Ausgabe 03/2019

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 03

Java aktuell

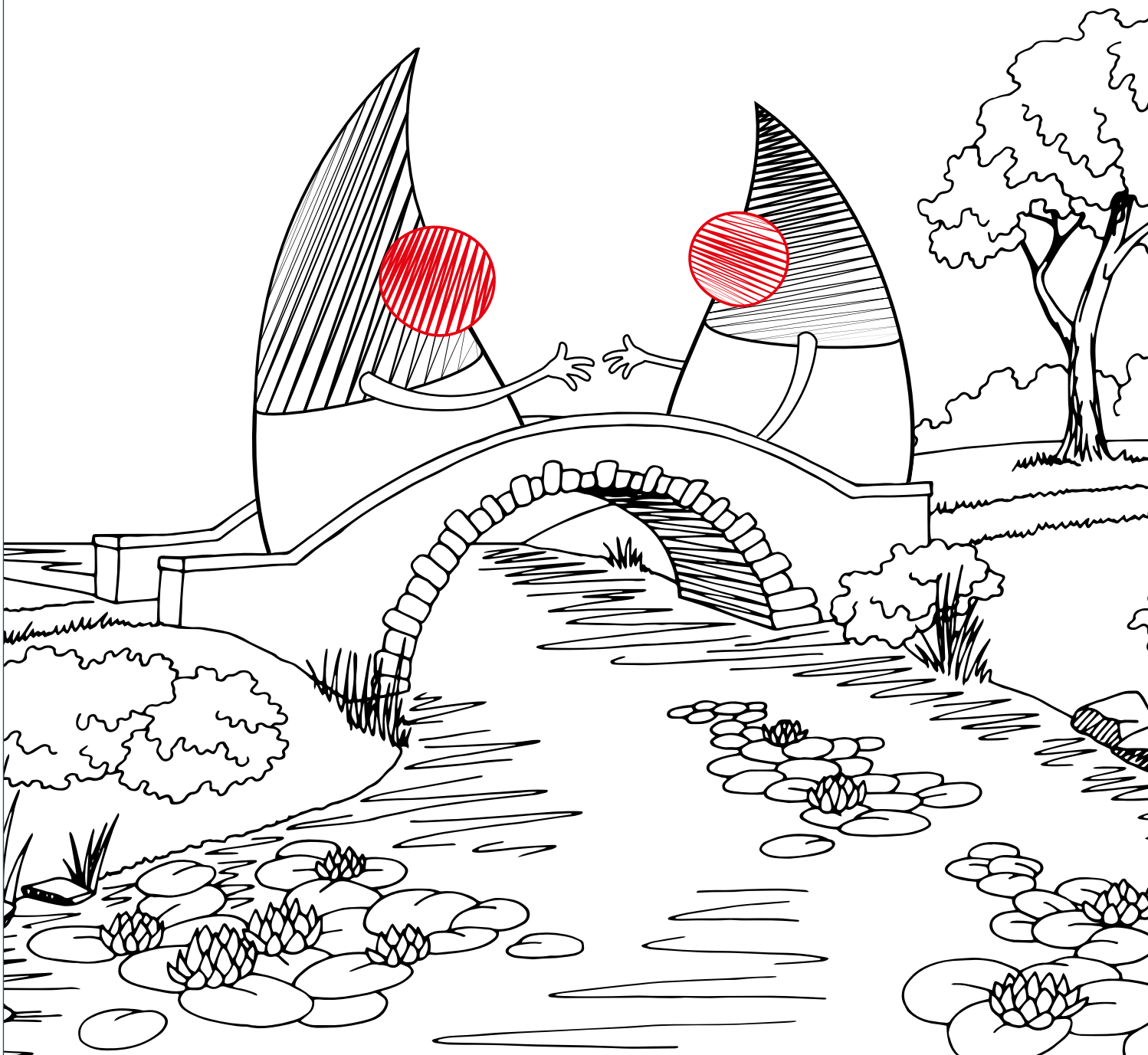
iJUG Verbund
www.ijug.eu

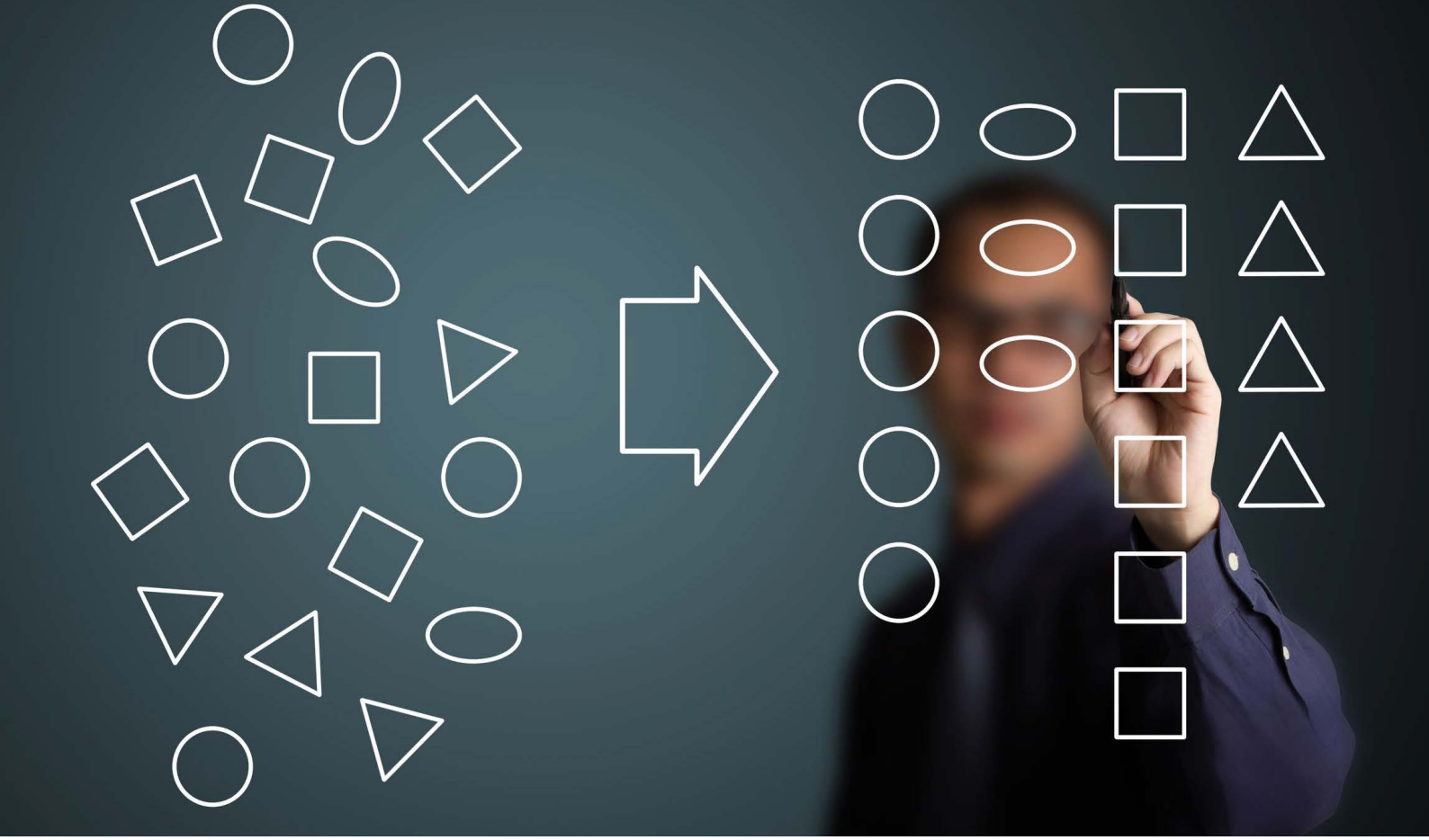
Jubiläum
Der iJUG feiert sein zehnjähriges Bestehen

Lizenzänderung
Was Java-Nutzer jetzt tun können

Mobile
iOS-Apps in Java

Java verbindet





Eine moderne und konsistente Implementierung natürlicher Ordnung bei Java-Objekten – Teil 1

Christian Heitzmann, SimplexaCode AG

Eigene Java-Klassen sortierbar, also komparabel zu machen, ist eine elementare und wichtige Aufgabe. Gemessen an ihrer Bedeutung wird dieses Thema in der Grundlagen-Literatur jedoch nur sehr spärlich behandelt. Hinzu kommt, dass mit neuen Java-Versionen und somit neuen Sprachmerkmalen immer wieder neue Varianten entstanden sind, wie sich die „compareTo"- beziehungsweise „compare"-Methoden mehr oder weniger elegant implementieren lassen. Dieser zweiteilige Artikel in dieser und in der nächsten Ausgabe zeigt die Evolution dieser Methoden über die verschiedenen Java-Versionen auf und bietet eine Code-Vorlage für eine saubere Implementierung ab Java 8 an.

Auch wenn die Implementierung einer natürlichen Ordnung in eigenen Java-Klassen durchaus bedeutsam sein mag, so ist es zumindest nicht notwendig, diese Definition für alle eigenen Klassen vorzunehmen. Hingegen ergibt es durchaus Sinn, dies für sogenannte „Werteklassen" zu tun, also für Klassen, deren Schwerpunkt mehr auf Daten als auf Verhalten liegt.

Die Definition spezifischer natürlicher Ordnung wird spätestens dann unvermeidbar, wenn die Klassen in Algorithmen oder Datenstrukturen (vor allem „Collections") verwendet werden, die auf irgendeine Art eine Sortierung vornehmen. Das wird ganz klar bei Methoden wie „Arrays.sort", „Collections.sort" oder „List#sort" (seit Java 8), kann aber bei Collections wie „TreeSet" oder „TreeMap" auch weniger offensichtlich sein, weil diese Datenstrukturen ihre Sortierung (und sogar Balancierung) intern und üblicherweise für den API-Benutzer unsichtbar durchführen.

Zum Glück kann ein Programmierer nicht aus Versehen eine Methode oder Collection benutzen, die ein vergleichbares (komparables)

Objekt benötigt, ohne dass das Objekt auch tatsächlich komparabel ist. Entweder implementiert die eigene Klasse die generische Schnittstelle „Comparable<T>“, die den Programmierer wiederum zwingt, ihre einzige Methode „compareTo(T)“ mit Funktionalität zu füllen, oder der Programmierer kann einen externen generischen „Comparator<T>“ implementieren, der dann der sortierenden Methode oder Datenstruktur zusammen mit den eigentlichen Klassenobjekten übergeben wird. Mit anderen Worten, die Implementierung von „Comparable“ stellt eine „eingebaute interne“ Definition der Klassenreihenfolge dar, während „Comparator“ einer „nachgerüsteten externen“ Definition entspricht.

Streng genommen sollte man nur bei Ersterem, also bei der internen Implementierung, von „natürlicher Ordnung“ sprechen, denn das „natürlich“ bezieht sich darauf, dass die Objekte „von selbst“, also „natürlich“ wissen, in welcher Reihenfolge sie sich anzuordnen haben. Bei der externen Implementierung wird dies von außen „auferdiktiert“, was nicht zwingend „natürlich“ sein muss. Nachdem nun auf diese Finesse hingewiesen wurde, wird in diesem Artikel auf eine derart strenge Unterscheidung verzichtet.

Die alte Art bis Java 6

Abgesehen vom aufgezeigten Hauptunterschied zwischen „Comparable“ und „Comparator“ bleibt die eigentliche Implementierung der natürlichen Ordnung einer Klasse für beide Varianten ähnlich. Hier muss man unterscheiden zwischen der Art, wie „Comparable compareTo(T)“- und „Comparator compare(T, T)“-Methoden in der Vergangenheit, bis und mit Java 6, implementiert wurden, wie sie dann in Java 7 implementiert werden dürfen und wie sie letztlich heute, seit Java 8, am konsistentesten implementiert werden können.

Um die unterschiedlichen Arten zu demonstrieren, wird die folgende Klasse eingeführt. Sie stellt eine künstliche Werteklasse dar, die alle möglichen Datentypen für ihre Attribute (englisch „fields“) ent-

```
public final class ComparableJava6DemoClass implements Comparable<ComparableJava6DemoClass> {
  private boolean booleanField;
  private byte byteField;
  private char charField;
  private short shortField;
  private int intField;
  private long longField;
  private float floatField;
  private double doubleField;
  private OtherComparableClass comparableClassField;

  /* [...] Constructors omitted. */
  @Override
  public int compareTo(ComparableJava6DemoClass otherDemoClass) {
     int difference;
     /* 1. booleanField */
     if (!this.booleanField && otherDemoClass.booleanField) return -1;
     if (this.booleanField && !otherDemoClass.booleanField) return +1;
     /* 2. byteField */
     if (this.byteField < otherDemoClass.byteField) return -1;
     if (this.byteField > otherDemoClass.byteField) return +1;
     /* 3. charField */
     if (this.charField < otherDemoClass.charField) return -1;
     if (this.charField > otherDemoClass.charField) return +1;
     /* 4. shortField */
     if (this.shortField < otherDemoClass.shortField) return -1;
     if (this.shortField > otherDemoClass.shortField) return +1;
     /* 5. intField */
     if (this.intField < otherDemoClass.intField) return -1;
     if (this.intField > otherDemoClass.intField) return +1;
     /* 6. longField */
     if (this.longField < otherDemoClass.longField) return -1;
     if (this.longField > otherDemoClass.longField) return +1;
     /* 7. floatField */
     difference = Float.compare(this.floatField, otherDemoClass.floatField);
     if (difference != 0) return difference;
     /* 8. doubleField */
     difference = Double.compare(this.doubleField, otherDemoClass.doubleField);
     if (difference != 0) return difference;
     /* 9. comparableClassField */
     if (this.comparableClassField == null) {
        difference = (otherDemoClass.comparableClassField == null ? 0 : +1);
     } else {
        difference = (otherDemoClass.comparableClassField == null ? -1
           : this.comparableClassField
           .compareTo(otherDemoClass.comparableClassField));
     }
     if (difference != 0) return difference;
     return 0;
  }
  /* [...] Other methods omitted. */
}
```

Listing 1

hält, es gibt also ein Attribut für jeden primitiven Java-Datentyp, wie „short" oder „double" (mit den entsprechenden Namen „shortField" und „doubleField"), sowie ein Attribut für einen Referenztyp (mit dem Namen „comparableClassField", der in diesem Fall eine andere komparable Klasse namens „OtherComparableClass" referenziert).

Es wurde darauf verzichtet, auch Arrays zu referenzieren, da die Implementierung von Ordnung auf Arrays normalerweise das Durchlaufen all seiner Elemente in einer Schleife benötigt, den ganzen Code aufbläht, in der Praxis nicht sehr geläufig ist und daher auch nicht viel zur Kernaussage dieses Artikels beiträgt *(siehe Listings 1 und 2)*.

Ein paar Punkte sind erwähnenswert:

- Die Reihenfolge, in der die Attribute verglichen werden, spielt auf jeden Fall eine Rolle. In dieser künstlichen Demoklasse werden die Klassen-Attribute in derselben Reihenfolge verglichen, in der sie als Felder deklariert sind. Es gibt keinen Grund, dies zwingend so zu tun. In realen Klassen wird es viel wahrscheinlicher sein, dass eine andere als die Deklarations-Reihenfolge verwendet wird. Der erste Vergleich (hier kommentiert mit „/* 1. booleanField */") entspricht dem höchstwertigen Attribut. Nur wenn sich die zwei booleschen Werte als gleich herausstellen, wird das zweit-höchstwertige Attribut (in diesem Fall „byteField", kommentiert mit „/* 2. byteField */") geprüft und verglichen. Man sieht, dass es in der Demoklasse insgesamt neun Vergleichsschritte gibt. Erst wenn sich alle neun Attribute als gleich herausstellen, dann, und nur dann, wird die „compareTo"-Methode „0" zurückgeben und somit Gleichheit anzeigen.
- Einen externen „Comparator" zu implementieren, funktioniert ähnlich, ist jedoch nicht exakt das Gleiche. Seine „compare(T, T)"-Methode benötigt zwei Parameter: die zwei Klassen, die verglichen werden sollen. Die „Comparable compareTo(T)"-Methode benötigt nur einen Parameter, da sie sich selbst bereits kennt und darum nur eine weitere Referenz auf das andere Objekt benötigt. Der gezeigte Code-Ausschnitt verwendet daher „demoClass1" und „demoClass2" anstelle von „this" und „otherDemoClass" *(siehe Listing 3)*.

```
public abstract class OtherComparableClass implements
Comparable<OtherComparableClass> {
  /* [...] Body omitted. */
}
```

Listing 2

Ein weiterer Unterschied ist die Verwendung von Getter-Methoden, da ein externer „Comparator" normalerweise nicht direkt auf die Klassenattribute, die verglichen werden sollen, zugreifen kann (das nennt sich „Kapselung"). Wann immer möglich, sollte man versuchen, die interne Methode zu implementieren, anstatt eine externe „compare"-Methode zu schreiben.

Es gibt verschiedene Arten, die Attribute zu vergleichen, abhängig von ihrem Typ:

- Primitive, ganzzahlige Typen (also „byte", „char", „short", „int" und „long", siehe Vergleiche 2 bis 6) scheinen die einfachsten zu sein, sollten aber ab Java 8 trotzdem geändert werden (siehe unten). Es ist einfach, ihre Funktionalität zu verstehen. Im Falle der ganzzahligen „compareTo"-Methode versetzt man sich selbst in die Position der Klasse, die sie implementiert. Dann sagt man: „Wenn mein (also „this") Wert kleiner als der Wert des anderen (also „Parameters") ist, dann gib eine negative Zahl zurück. Wenn mein Wert größer als der Wert des anderen ist, dann gib eine positive Zahl zurück."
- Man kann auch ähnlich bei der externen „compare"-Methode vorgehen: „Wenn der Wert des ersten Parameters kleiner als der Wert des zweiten Parameters ist, dann gib eine negative Zahl zurück. Wenn der Wert des ersten Parameters größer als der Wert des zweiten Parameters ist, dann gib eine positive Zahl zurück."
- Der Vergleich von „boolean" (siehe Vergleich 1) ist weniger offensichtlich, aber immer noch einfach nachzuvollziehen, wenn man den

```
import java.util.*;

public final class DemoComparator implements Comparator<ComparableJava6DemoClass> {
  @Override
  public int compare(ComparableJava6DemoClass demoClass1, ComparableJava6DemoClass demoClass2) {
     int difference;
     /* 1. booleanField */
     if (!demoClass1.isBooleanField() && demoClass2.isBooleanField()) return -1;
     if (demoClass1.isBooleanField() && !demoClass2.isBooleanField()) return +1;
     /* 2. byteField */
     if (demoClass1.getByteField() < demoClass2.getByteField()) return -1;
     if (demoClass1.getByteField() > demoClass2.getByteField()) return +1;
     /* [...] Other comparisons omitted. */
     /* 9. comparableClassField */
     if (demoClass1.getComparableClassField() == null) {
        difference = (demoClass2.getComparableClassField() == null ? 0 : +1);
     } else {
        difference = (demoClass2.getComparableClassField() == null ? -1
           : demoClass1.getComparableClassField()
           .compareTo(demoClass2.getComparableClassField()));
     }
     if (difference != 0) return difference;
     return 0;
  }
}
```

Listing 3

```
public final class ComparableJava7DemoClass implements Comparable<ComparableJava7DemoClass> {
  /* [...] Fields omitted. */

  /* [...] Constructors omitted. */
  @Override
  public int compareTo(ComparableJava7DemoClass otherDemoClass) {
     int difference;
     /* 1. booleanField */
     difference = Boolean.compare(this.booleanField, otherDemoClass.booleanField);
     if (difference != 0) return difference;
     /* 2. byteField */
     difference = Byte.compare(this.byteField, otherDemoClass.byteField);
     if (difference != 0) return difference;
     /* 3. charField */
     difference = Character.compare(this.charField, otherDemoClass.charField);
     if (difference != 0) return difference;
     /* 4. shortField */
     difference = Short.compare(this.shortField, otherDemoClass.shortField);
     if (difference != 0) return difference;
     /* 5. intField */
     difference = Integer.compare(this.intField, otherDemoClass.intField);
     if (difference != 0) return difference;
     /* 6. longField */
     difference = Long.compare(this.longField, otherDemoClass.longField);
     if (difference != 0) return difference;
     /* 7. floatField */
     difference = Float.compare(this.floatField, otherDemoClass.floatField);
     if (difference != 0) return difference;
     /* 8. doubleField */
     difference = Double.compare(this.doubleField, otherDemoClass.doubleField);
     if (difference != 0) return difference;
     /* 9. comparableClassField */
     if (this.comparableClassField == null) {
        difference = (otherDemoClass.comparableClassField == null ? 0 : +1);
     } else {
        difference = (otherDemoClass.comparableClassField == null ? -1
           : this.comparableClassField
           .compareTo(otherDemoClass.comparableClassField));
     }
  if (difference != 0) return difference;
     return 0;
  }
  /* [...] Other methods omitted. */
}
```

Listing 4

Quelltext liest. In diesem Fall kommt „false" vor „true", also eine Instanz von ComparableDemoClass, die das „booleanField" auf „false" gesetzt hat, ist kleiner als eine Instanz der gleichen Klasse, die „booleanField" auf „true" gesetzt hat. Man beachte, dass dies lediglich eine Definitionssache und nicht in Stein gemeißelt ist.

- Fließkommazahlen (also „float" und „double", siehe Vergleiche 7 und 8) sollten die statische „compare"-Methode ihrer Wrapper-Klassen „Float" beziehungsweise „Double" verwenden. „float" und „double" sind viel komplizierter als ganzzahlige Typen, weil sie besondere Werte besitzen können: positive und negative Nullen, positive und negative „Unendlichkeiten" (englisch „infinities") sowie NaN („Not-a-Number") als Resultat gewisser ungültiger Operationen, wie die Division von Null durch Null. Da man sich wahrscheinlich nicht mit all diesen Sonderfällen herumschlagen möchte, verwendet man einfach die oben erwähnte „compare"-Methode, die einem all die schwere Arbeit abnimmt.
- Die alte Art, Referenzattribute zu vergleichen (siehe Vergleich 9), ist langatmig und fehleranfällig. Es lohnt sich nicht, durch die Details des Code-Blocks zu gehen, weil er nicht mehr relevant ist. Die Grundidee ist, dass die „compareTo"-Methoden der Referenz-Typen rekursiv aufgerufen werden. Da mein Referenz-Attribut, das andere Referenz-Attribut (also das Referenz-Attribut des Parameters) oder beide „null" sein können, muss eine komplizierte Fallunterscheidung durchgeführt werden, die den Code aufbläht. In obigem Quelltext ist „null" größer als jeder andere (gültige) Wert. Mit anderen Worten, „null" kommt in der Reihenfolge zuletzt.

Die Zwischenlösung in Java 7

In Java 7 wurden alle Wrapper-Klassen für boolesche Werte („Boolean" für „boolean") und alle ganzzahligen primitiven Typen („Byte" für „byte", „Character" für „char", „Short" für „short", „Integer" für „int" und „Long" für „long") mit statischen „compare"-Methoden ausgestattet. Ihre Verwendung ist konsistent zu denjenigen von „Float" und „Double". Joshua Bloch, der Autor des großartigen Buchs „Effective Java", schreibt, die Verwendung von „relational operators „<" and „>" in compareTo-methods is verbose and error-prone and no longer recommended."

Der Autor stimmt dieser Aussage nicht ganz zu. Wie man in unten stehendem Code-Beispiel sehen kann, hat sich die „Code-Verbosität" – sofern man überhaupt von „Verbosität" sprechen kann – nicht wirklich geändert. In Bezug auf die behauptete Fehleranfälligkeit warf er einen Blick in die JDK-Implementierungen der statischen „compare"-Methoden der Wrapper-Klassen, um herauszufinden, ob er irgendwelche Fallstricke im Zusammenhang mit den Vergleichs-

```
import java.util.*;
public final class ComparableJava8DemoClass implements Comparable<ComparableJava8DemoClass> {
  private static final Comparator<ComparableJava8DemoClass> DEMO_CLASS_COMPARATOR
     = Comparator.comparing((ComparableJava8DemoClass dc)
        -> Boolean.valueOf(dc.booleanField))                          /* 1. */
     .thenComparingInt(dc -> dc.byteField)                            /* 2. */
     .thenComparingInt(dc -> dc.charField)                            /* 3. */
     .thenComparingInt(dc -> dc.shortField)                           /* 4. */
     .thenComparingInt(dc -> dc.intField)                             /* 5. */
     .thenComparingLong(dc -> dc.longField)                           /* 6. */
     .thenComparingDouble(dc -> dc.floatField)                        /* 7. */
     .thenComparingDouble(dc -> dc.doubleField)                       /* 8. */
     .thenComparing(dc -> dc.comparableClassField,
        Comparator.nullsLast(OtherComparableClass::compareTo));       /* 9. */
  /* [...] Fields omitted. */

  /* [...] Constructors omitted. */
  @Override
  public int compareTo(ComparableJava8DemoClass otherDemoClass) {
     return DEMO_CLASS_COMPARATOR.compare(this, otherDemoClass);
  }
  /* [...] Other methods omitted. */
}
```

Listing 5

operatoren „<" und „>" für ganzzahlige Typen übersehen hat, aber nein, in diesen Methoden finden wirklich keinerlei Zaubereien statt. In dieser Hinsicht sieht er also keinen Vorteil in der Verwendung der statischen „compare"-Methoden der Wrapper-Klassen, aus Sicht der Konsistenz ist es jedoch absolut sinnvoll. Die Benutzung statischer „compare"-Methoden ist nun für alle primitiven Typen in Java gleich, was mehr als Grund genug ist, daran festzuhalten *(siehe Listing 4)*.

Die neue Art seit Java 8

Wie man sehen kann, haben sich die Dinge für den Vergleich des Referenz-Typs (siehe Vergleich 9) noch immer nicht geändert. Sein Code-Block ist wirklich langatmig und fehleranfällig. Java 8 führte „Optional" ein, die als kleine Container für einzelne Objekte aufgefasst werden können und entweder leer sind oder eben das Objekt enthalten. Ihr Vorteil beruht auf der Tatsache, dass ihr Inhalt nie „null" sein kann.

Dieses Konzept könnte vielleicht hilfreich sein, um die Fallunterscheidung ein wenig zu vereinfachen. Man denke dabei an die Verwendung der „Optional"-Methoden „ifPresent" oder „ifPresentOrElse" (seit Java 9), die die Menge an „if-else"-Befehlen reduzieren könnten. Da ich jedoch einen komplett anderen Ansatz vorstellen werde, müssen wir diese Idee nicht weiter verfolgen.

Beginnend mit Java 8 können Schnittstellen nun auch statische und Default-Implementierungen enthalten. Das ist genau das, was „Comparator" macht: Er wurde mit einer Vielzahl von Methoden ausgestattet, die ganz grob als „Fabrik-Methoden" verstanden werden können. Sie erlauben das schrittweise Erstellen von „Comparatoren", die wiederum entweder direkt oder innerhalb der „compareTo"-Methode einer Klasse aufgerufen werden können *(siehe Listing 5)*.

Erneut bedürfen ein paar Punkte einer Erläuterung:

- Die Klasse enthält nun ein statisches und finales „Comparator"-Attribut (mit dem Namen „DEMO_CLASS_COMPARATOR"). Es wird innerhalb der Klassenmethode „compareTo" aufgerufen, was die „compareTo"-Methode auf eine einzige Zeile reduziert. Wenn der „Comparator" als „innere Klasse" deklariert wird (am besten als „statische innere Klasse"), kann er auf alle (üblicherweise privaten) Attribute der äußeren Klasse zugreifen. Wird der „Comparator" irgendwo außerhalb deklariert, dann ist er normalerweise abhängig von Getter-Methoden, um auf die Attribut-Werte zugreifen zu können. Es ist allerdings nicht wirklich entscheidend, ob solch ein „Comparator" nun „intern" oder „extern" verwendet wird – so oder so ist seine Benutzung deutlich konsistenter geworden.
- Der „Comparator" wird Schritt für Schritt erstellt. Man beginnt mit einer der statischen „Comparator.comparing[XXX]"-Methoden für das höchstwertige Attribut und hängt dann die anderen Attribute in der gewünschten Reihenfolge unter Benutzung der „thenComparing[XXX]"-Methoden an. Für „boolean" und alle Referenz-Attribute verwendet man „comparing" und „thenComparing". Für „byte", „char", „short" und „int" verwendet man „[then]comparingInt". Für „long" nimmt man „[then]comparingLong", für „float" und „double" kommt „[then]comparingDouble" zum Einsatz . Wenn man die Methoden für primitive Typen nicht überall dort verwendet, wo sie möglich wären, dann wird Java Autoboxing durchführen, was fehleranfällig sein und die Performance signifikant beeinträchtigen kann. Leider stellt Java keine „[then]comparingBoolean"-Methode zur Verfügung, sodass man sich für diesen Typ ausnahmsweise mit Autoboxing abfinden muss (oder das Boxing manuell vornimmt, so wie ich im obigen Code-Beispiel).
- Jede „[then]comparing[XXX]"-Methode benötigt eine sogenannte „Key-Extractor-Funktion" als Parameter, was üblicherweise mithilfe von Lambdas implementiert wird. Die Eingabe für so einen Key-Extractor ist diejenige Klasse, mit der verglichen werden soll (hier die Demoklasse, referenziert durch „dc"); die Ausgabe ist das extrahierte Attribut. Wenn der Lambda-Ausdruck direkten Zugang zu den privaten Attributen hat, dann ist es einfach; wenn nicht, dann muss man die entsprechenden Getter-Methoden aufrufen, wie oben erwähnt.
- Der Java-Compiler ist in der Regel sehr gut darin, die korrekten Typen bei Lambda-Ausdrücken herauszufinden. Allerdings scheint es hier Probleme zu geben. Aus diesem Grund muss der Typ der Eingabe zumindest im ersten Befehl ausdrücklich spezi-

fiziert sein, wie im Beispiel zu sehen ist: „(ComparableJava8DemoClass dc) -> Boolean.valueOf(dc.booleanField))". Die Typen der darauffolgenden Lambdas sollten danach automatisch bestimmt werden können.

- Die „[then]comparing"-Methoden (nicht für „int", „long" oder „double") erlauben die Angabe eines weiteren „Comparator" im zweiten Methoden-Parameter. Wenn man sich Vergleich 9 im Code-Beispiel anschaut, sieht man die zusätzliche Spezifikation „Comparator.nullsLast(OtherComparableClass::compareTo)". „Comparator.nullsLast" ist eine statische Methode, die einen „Comparator" mit weiterer Funktionalität im Umgang mit „null"-Werten ausstattet – anstatt eine „NullPointerException" zu werfen und ohne langatmige Fallunterscheidungen, wie in den Code-Beispielen zu sehen war. Wenn „Comparator.nullsLast" verwendet wird, dann kommt „null" zuletzt. Wer möchte, dass „null" zuerst kommt, verwendet einfach „Comparator.nullsFirst".

Zusammenfassung und Ausblick

Das war's für den ersten Teil. Die Art der Implementierung natürlicher Ordnung in Java 8 sieht modern und prägnant aus. Ihre Verwendung ist konsistent, sicher und – am wichtigsten – einfach. Die vollständigen Quelltexte zu diesem Artikel können unter *link.simplexacode.ch/8sk8* zur privaten Nutzung heruntergeladen werden. Im Teil 2 zu diesem Thema werden in der nächsten Ausgabe einige praxisrelevante Beispiele für die Implementierung von Ordnung und die Sortierung realistischerer Werteklassen gezeigt.

Hinweis: Die englische Version des Artikels finden Sie unter *link.simplexacode.ch/vq29*.

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.