04-2014 | Winter | www. ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler
Aus der Community – für die Community

Mit Lambda-Ausdrücken einfacher programmieren

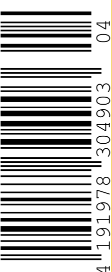
Go for the Money
Währungen und Geldbeträge in Java

Oracle-Produkte
Die Vorteile von Forms und Java vereint

Pimp my Jenkins
Noch mehr praktische Plug-ins

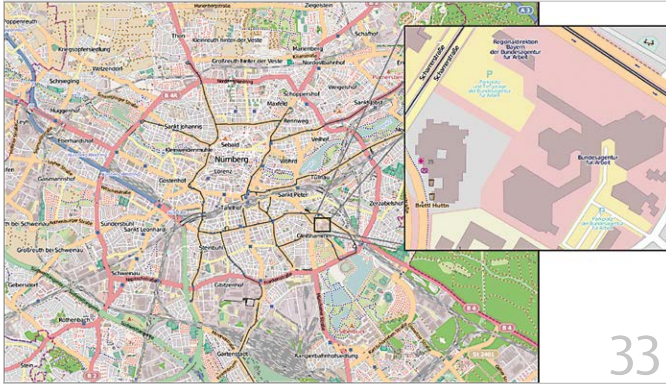
Java aktuell

D: 4,90 EUR A: 5,60 EUR CH: 9,80 CHF Benelux: 5,80 EUR ISSN 2191-6977



iJUG Verbund

Sonderdruck



OpenStreetMap ist vielen kommerziellen Diensten überlegen, Seite 33

Alles über Währungen und Geldbeträge in Java, Seite 20

5 Das Java-Tagebuch
Andreas Badelt, Leiter der DOAG SIG Java

8 JDK 8 im Fokus der Entwickler
Wolfgang Weigend

15 Einmal Lambda und zurück – die Vereinfachung des TestRule-API
Günter Jantzen

20 Go for the Money – eine Einführung in JSR 354
Anatole Tresch

24 Scripting in Java 8
Lars Gregori

28 JGiven: Pragmatisches Behavioral-Driven-Development für Java
Dr. Jan Schäfer

33 Geodatensuche und Datenanreicherung mit Quelldaten von OpenStreetMap
Dr. Christian Winkler

38 Pimp my Jenkins – mit noch mehr Plug-ins
Sebastian Laag

41 Apache DeviceMap
Werner Kei

46 Schnell entwickeln – die Vorteile von Forms und Java vereint
René Jahn

50 Oracle-ADF-Business-Service-Abstraktion: Data Controls unter der Lupe
Hendrik Gossens

54 Neue Features in JDeveloper und ADF 12*c*
Jürgen Menge

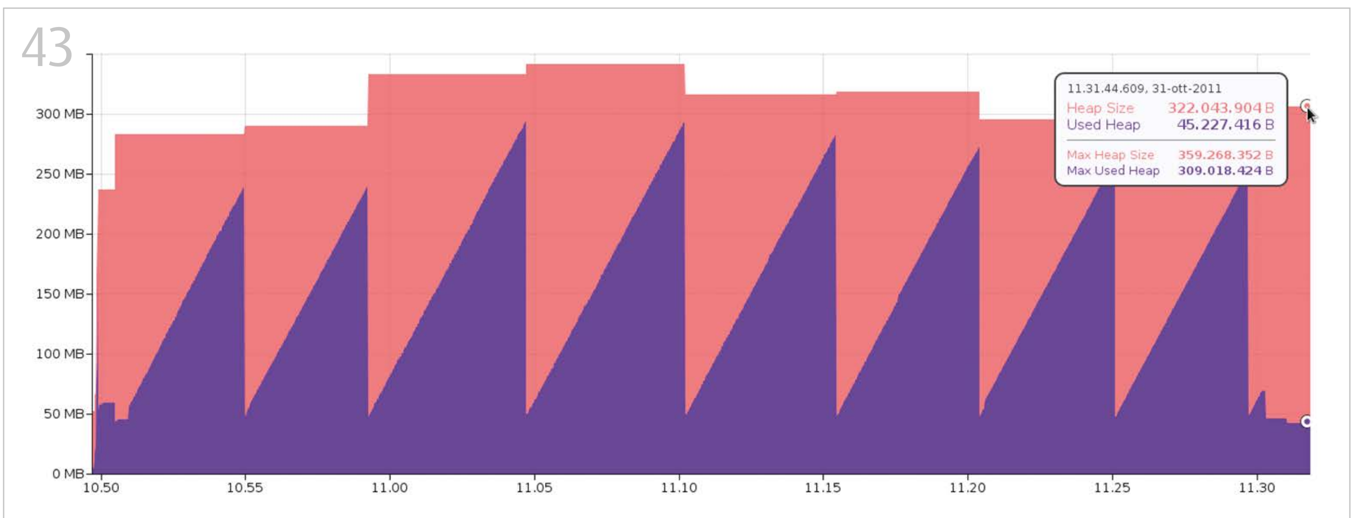
57 Der (fast) perfekte Comparator
Heiner Kücker

60 Clientseitige Anwendungsintegration: Eine Frage der Herkunft
Sascha Zak

64 Unbekannte Kostbarkeiten des SDK Heute: Die Klasse „Objects“
Bernd Müller

66 Inserenten

66 Impressum



WURFL ist verglichen mit selbst dem nicht reduzierten OpenDDR-Vokabular deutlich speicherhungriger, Seite 43

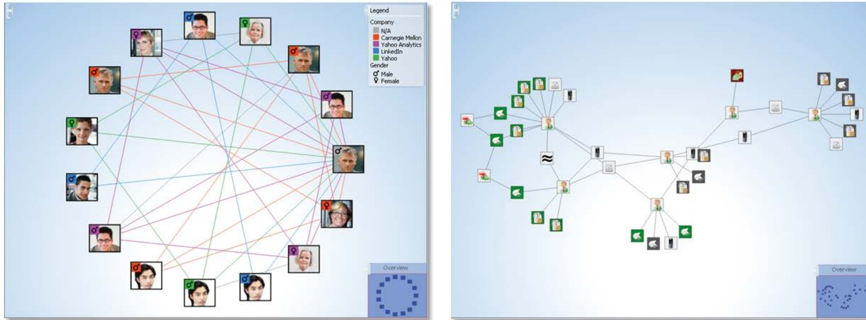


Abbildung 6: Diagram

darstellen und bietet großen Freiraum in Bezug auf das Layout (siehe Abbildung 6).

Für viele Komponenten ist die Bedienung durch Gesten hinzugekommen, um dem wachsenden Bedarf an mobilen Applikationen gerecht zu werden. Eine Live-Darstellung aller zur Verfügung stehenden Komponenten findet man unter „http://jdevadf.oracle.com".

Mit dem JDeveloper 12*c* sind neue Skins (Neuer Default Skin: „Skyros") hinzugekommen. Der grafische Skin.Editor hat einige Verbesserungen erfahren. Zudem wurde Maven 3 als Standard Build System integriert. Applikationen können bereits beim Anlegen in einem Maven Repository verwaltet oder zu einem späteren Zeitpunkt in ein solches überführt werden. Neben der Untersützung für SVN und anderen verbreiteten Systemen kommt mit dem JDeveloper 12*c* git/GitHub als weiteres populäres Versionskontrollsystem hinzu.

Weitergehende Informationen

- Oracle JDeveloper 12*c* (12.1.2.0.0.): www.oracle.com/technetwork/developer-tools/jdev/documentation/1212-nf-1964675.html
- Oracle JDeveloper 12*c* (12.1.3.0.0): www.oracle.com/technetwork/developer-tools/jdev/documentation/1213nf-2222743.html?msgid=3-10243408667

Dr. Jürgen Menge
juergen.menge@oracle.com

Dr. Jürgen Menge hat bei Oracle zunächst sechs Jahre in der Schulung als Trainer für die Entwicklungs-Werkzeuge Oracle Designer und Oracle Forms gearbeitet, um dann als technischer Berater in den Lizenzvertrieb zu wechseln. Seine fachlichen Schwerpunkte sind sowohl die klassischen Entwicklungs-Werkzeuge als auch die neuen, Standard-basierten Tools (JDeveloper/ADF, BI Publisher etc.).

http://ja.ijug.eu/14/4/13

Der (fast) perfekte Comparator

Heiner Kücker, Freiberufler

Eine Liste ganz einfach sortieren, entsprechende Methode aufrufen, Comparator übergeben. Dieser Artikel zeigt, dass auch ein einfacher Comparator Verbesserungsmöglichkeiten birgt.

Jeder Java-Programmierer hat sicher schon mal so einen Comparator geschrieben (siehe Listing 1).

Wenn nach einer anderen „Property (Member)" unserer Bean sortiert werden soll, müssen der gesamte Code kopiert und die abzufragenden Properties angepasst werden. Gibt es mehrere Beans, sind außerdem die beiden Parameter-Typen der „compare"-Methode zu ändern, also bereits vier Code-Stellen – eine langweilige und fehleranfällige Angelegenheit.

Getter ohne Reflektion

Basis dieses Alternativ-Vorschlags ist ein Property-Accessor, der ohne Reflektion auskommt, in diesem Fall nur der lesende Anteil, der „Getter" (siehe Listing 2).

Die beiden Typ-Parameter (Generics) „B-(Bean)" und „P(Property)" bestimmen die jeweils beteiligten Typen. Der Sinn der „get"-Methode sollte klar sein. Dieses Interface wird für alle benötigten Properties (Member) implementieren (siehe Listing 3).

Die Getter-Objekte sind statisch, existieren also nur einmal je JVM. Durch die Typ-Parameter sind Namensverwechslungen

und Typ-Fehler wie bei der Benutzung von Reflektion ausgeschlossen. Die Laufzeit ist sicher auch ok, wahrscheinlich besser als mit Reflektion. Das Getter-Objekt wird nun dem verbesserten wiederverwendbaren Comparator (Code der Java-Klasse mit Member und Initialisierung im Konstruktor hier weggelassen) übergeben: „new AccessorComparator(Customer.SUR_NAME_GETTER);“. Jetzt ist nur noch ein Wert zu ändern. Die Getter-Objekte müssen auch vorhanden sein, man kann sie allerdings noch an anderen Codestellen verwenden. Listing 4 zeigt die „compare“-Methode des verbesserten Comparators.

Ein weiterer Wunsch ist die „null“-Sicherheit. „null“-Werte sind nicht vergleichbar, sie verstoßen gegen die totale Ordnung, in realen Applikationen tauchen sie jedoch oft auf. Hier wird davon ausgegangen, dass „null“ kleiner als Nicht-„null“ ist (siehe Listing 5).

Kaum ist dieses Problem gelöst, ist nach mehreren Properties in einer definierten Rangfolge zu sortieren. Man übergibt die Getter-Objekte als Array mit einem „vararg“-Parameter oder dynamisch als Liste (siehe Listing 6).

Diese Lösung ist effektiv, der Aufruf der „compareTo“-Methode und damit der Zugriff auf die Properties erfolgt nur, wenn unbedingt nötig. Dies könnte aufwändig sein und das Abfragen der Properties könnte im Hibernate-/JPA-Umfeld eine Datenbank-Abfrage auslösen. Nun kommt noch jemand daher und möchte nach bestimmten Properties absteigend sortieren. Ein Problem dabei ist, dass wir nicht wie mit „Collections.reverseOrder(Comparator)“ die Sortier-Richtung für eine einzelne Property umkehren können. Aber das ist möglich (siehe Listing 7).

Die Methode „invert“ ist eine statische Methode des „AccessorComparator“, im obigen Code-Schnipsel per statischem Import ohne Klassen-Präfix notierbar. Sie liefert ein spezielles magisches Objekt mit dem originalen Getter als Member (siehe Listing 8, Generics aus Platzgründen weggelassen). Die Klasse „InvertGetter“ ist „private“ im „AccessorComparator“ versteckt. Beim Sortieren wird per „instanceOf“ geprüft, ob der magische Getter auftaucht (siehe Listing 9).

Das ist nicht besonders schön, bleibt aber unter dem API des „AccessorComparator“ verborgen. Dynamisch kann man die Sortier-Properties festlegen, indem man sie in einer Liste sammelt und die Liste dann per „toArray“-Methode in ein Array für den „vararg“-Parameter umwandelt.

Unter [2] kann der gesamte Code mit einer Beispiel-Bean und Unit-Tests heruntergeladen werden. Für die Benutzung gibt es keine Einschränkungen, jeder kann mit dem Code alles tun, zum Beispiel in sein

```
Comparator<Customer> comparator =
        new Comparator<>() {
            @Override
            public int compare(
                    final Customer c1 ,
                    final Customer c2 ) {
                return c1.getSurName().compareTo(
                        c2.getSurName() );
            }
        };
```

Listing 1

```
interface Getter<B, P> {
    P get( B bean );
}
```

Listing 2

```
public static final Getter<Customer, String> NAME_GETTER =
        new Getter<Customer, String>() {
            @Override
            public String get( Customer bean ) {
                return bean.getName();
            }
        };
```

Listing 3

```
@Override
 public int compare(
         final T o1 ,
         final T o2 ) {
     return this.getter.get( o1 ).compareTo(
             this.getter.get( o2 ) );
 }
```

Listing 4

```
    P p1 = getter.get( o1 );
    P p2 = getter.get( o2 );
    if ( p1 == null && p2 == null ) {
        return 0;
    } else if ( p1 == null ) {
        return -1;
    } else if ( p2 == null ) {
        return 1;
    } else
        ...
```

Listing 5

```
    for ( Getter g : getters ) {
        int compareResult =
            getter.get( o1 ).compareTo(
                getter.get( o2 );
        if ( compareResult != 0 ) {
            return compareResult;
        }
    }
    return 0;
```

Listing 6

```
  Comparator<Customer> comparator =
          new AccessorComparator<>(
                  invert(
                      Customer.SUR_NAME_GETTER ) );
```

Listing 7

```
  public static Getter invert(
          final Getter getterToInvert )
  {
      return new InvertGetter<>( getterToInvert );
  }
```

Listing 8

```
    if ( getter instanceof InvertGetter )
        // absteigende Sortierung
    {
        final InvertGetter<B, ?> invertGetter = (InvertGetter<B,
?>) getter;
        p1 = (P) invertGetter.getterToInvert.get( o1 );
        p2 = (P) invertGetter.getterToInvert.get( o2 );
    }
```

Listing 9

Projekt kopieren und das Package entsprechend anpassen. In einem weiteren Package ist ein Beispiel für einen PropertyComparator abgelegt, von dem es auch eine „NullIsLesserPropertyComparator"-Implementierung zum Behandeln von „null"-Werten gibt und der, wie beim klassischen Comparator üblich, durch die Methode „Collections.reverseOrder()" umgedreht werden kann. Schließlich ist eine andere Lösung immer nur ein paar Refactoring-Schritte entfernt.

Der fast perfekte Comparator

Der verbesserte Comparator vermeidet „Copy & Paste" und arbeitet effektiv bezüglich des Zugriffs auf die benötigten Properties. Die Anforderungen sind:

- Typsicher
- Effizient
- Multi-Property-/Field-fähig
- „null"-sicher
- Property-individuell aufsteigend/absteigend sortierbar

Die Lösung zum Umkehren der Sortier-Richtung mit „instanceOf" hat zwar den Geschmack der Unsauberkeit im Sinne der objektorientierten Programmierung, dem Autor ist jedoch keine bessere Lösung eingefallen. Wer eine bessere Idee hat, kann sich gern bei ihm melden.

Das Erstellen der Property-Accessoren ist erstmal ein höherer Aufwand, der sich allerdings bei entsprechend häufiger Benutzung auszahlt. Durch die Verwendung von Typ-Parametern („Generics") in der „Getter"-Klasse müssen primitive Properties in die jeweiligen Wrapper-Objekte verpackt werden. Bei der Arbeit mit „vararg"-Parametern wird jedes Mal ein Array erzeugt; Konstruktoren mit unterschiedlichen Parameter-Listen könnten effektiver sein. Mit den Java8-Lambdas ergeben sich eventuell neue Möglichkeiten, die der Autor hier nicht berücksichtigt hat.

Fazit

Dieser Comparator kann sicher kein Projekt retten, aber die zugrunde liegende Philosophie, eine ausdrucksstarke und der Problematik angemessene Lösung zu suchen, tut sicher jedem Projekt gut. Es ist oft zu beobachten, dass etliche Frameworks oder Tools alle möglichen Probleme lösen sollen. Das ist wie mit den Dampfmaschinen in der Anfangszeit der Industrialisierung: je größer, desto besser. Das einfache Programmierhandwerk wird nicht geschätzt und wenn, dann werden oft Prinzipien wie Clean-Code oder testgetriebene Entwicklung pauschal bis zur Umkehrung ihres Sinns betrieben.

Links

[1] http://www.nosid.org/java-compound-comparator.html
[2] http://www.heinerkuecker.de/Comparator.html

Heiner Kücker
mail@heinerkuecker.de

Heiner Kücker ist seit dem Jahr 2000 freiberuflicher Java-Entwickler und freut sich, wenn das Java-Typ-System ihn vor seinen Flüchtigkeitsfehlern (macht er ständig) schützt. Zur Rationalisierung seiner Arbeit schreibt er sich selbst kleine Main-Methoden-Tools. Außerdem interessiert er sich für fortschrittliche Programmiertechniken wie funktionale Programmierung.

http://ja.ijug.eu/14/4/14