Java 8 im Praxiseinsatz

10 August 2015 | **Software Engineering** | **Michael Inden**, **Marvin Minder**
Lesezeit: 6 Minutes

In diesem Blogbeitrag stelle ich mit freundlicher Erlaubnis der Firma Swisscom eine abgespeckte Variante eines Log-File-Analysetools vor, das ich im Rahmen meiner dortigen Tätigkeit erstellt habe. An dessen hier gezeigter Implementierung wird deutlich, wie sich die Features aus Java 8 sowie JavaFX kombinieren lassen und wie sich dadurch die Arbeit im Vergleich zu JDK 7 vereinfacht. Darüber hinaus schauen wir auf Streams, das Filter-Map-Reduce-Framework sowie eine ansprechende grafische Aufbereitung mithilfe von JavaFX und insbesondere Charts.

Problemkontext und Aufgabenstellung

Sicherlich kennen Sie es aus dem eigenen Alltag, dass Ihre Programme mitunter nicht die gewünschte und benötigte Performance liefern. In solchen Fällen ist oftmals der Einsatz eines Profilers und zunächst von VisualVM ein guter Ausgangspunkt. Für das hier als Grundlage dienende Programmsystem waren aber vor allem die Aufrufe von Umsystemen zeitaufwändig. Das Profiling lieferte keine stichhaltigen Anhaltspunkte zur Verbesserung. Weil die Softwarearchitekten schon im Vorhinein aufmerksam waren, haben sie die Möglichkeit geschaffen, Performance-Indikatoren in Log-Dateien zu schreiben. Insbesondere werden die Aufrufe von speziellen Methoden, die Fremdsysteme betreffen, protokolliert.

Nachfolgend soll ein Tool geschrieben werden, mit dessen Hilfe diese Log-Dateien analysiert werden können, um die dort protokollierten Laufzeiten von Methoden zu bestimmen und geeignet grafisch aufzubereiten.

Erste Schritte zur Informationsaufbereitung

Die Implementierung der Auswertung von Log-Dateien beginnen wir damit, dass wir uns das Format der auszuwertenden Daten anschauen. Danach zeige ich, wie sich daraus Informationen zur Laufzeit sowie zu den aufgerufenen Methoden ermitteln lassen.

Datenbasis

Die auszuwertenden Daten finden sich in Log-Dateien und die dort für uns relevanten Informationen sind in Zeilen mit folgendem schematischen Aufbau protokolliert:

```
<Zeitstempel> invoked: <Methodename> ... call_time:= 1000 ms
<Zeitstempel> invoked: <Methodename> ... calltime 250 ms
```

Wir sehen hier zwei nahezu identische Formate mit minimalen Abweichungen. So etwas findet man in der Realität immer mal wieder. Für uns stellt dies hier eine kleine zusätzliche Herausforderung dar, um die Informationen geeignet auslesen zu können.

Extraktion der Laufzeiten

Mithilfe von Lambdas und den Neuerungen aus Java 8 schreiben wir folgende Methode zur Extraktion der jeweiligen Aufrufdauern unter Beachtung eines Schwellwerts:

```
public static Stream<String> extractRelevantLines(final Path
pathToLogFile,
                           final long thresholdInMs)
                           throws IOException
{
   final Stream<String> lines = Files.lines(pathToLogFile);
   final Predicate<String> hasCallTime =
                                line -> line.contains("call_time") ||
                                        line.contains("calltime");

   final Stream<String> relevantLines = lines.filter(hasCallTime);
   return relevantLines.filter(line -> extractCallTime(line) >=
thresholdInMs);
}
```

Die Extraktion der relevanten Aufrufinformationen nutzt das Stream-API und im Speziellen die Methode `filter()`. Für die dadurch verbliebenen relevanten Zeilen wird jeweils geschaut, ob die dort protokollierte Laufzeit einen Schwellwert überschreitet. Insbesondere sollen in diesem Falle alle Aufrufe mit einer Dauer unter 500 ms herausgefiltert werden, weil diese für die Performance-Probleme unbedeutend sind. Dazu rufen wir die Methode `extractCallTime(String)` auf, die mitsamt einer Hilfsmethode sowie etwas Stringverarbeitungslogik wie folgt implementiert wird:

```
public static long extractCallTime(final String line)
{
   final long callTime1 = extractCallTimeValueInMs(line, "call_time:=
");
   final long callTime2 = extractCallTimeValueInMs(line, "calltime ");

   return Math.max(callTime1, callTime2);
}

private static long extractCallTimeValueInMs(final String line,
                                             final String prefix)
{
   final int posPrefix = line.indexOf(prefix);
   final int endPosPrefix = posPrefix + prefix.length();
   final int posPostfix = line.indexOf("ms", endPosPrefix);

   if (posPrefix >= 0 && posPostfix >= 0)
   {
      final String durationAsString = line.substring(endPosPrefix,
                                                     posPostfix - 1);
      return Long.parseLong(durationAsString);
   }
   return -1;
}
```

Extraktion des Methodennamens

Neben den Laufzeiten wollen wir natürlich auch die Verursacher langer Laufzeiten, also die aufgerufenen Methoden, ermitteln. Dazu benutzen wir einige Stringfunktionalitäten und suchen zunächst nach dem Präfix `invoked:` und ermitteln dann die Position der öffnenden Klammer, wodurch sich der Name der aufgerufenen Methode leicht wie folgt extrahieren lässt (vereinfachend betrachten wir hier keine Überladungen):

```
public static String extractCall(final String line)
{
   final String prefix = "invoked: ";
   final int posPrefix = line.indexOf(prefix);
   final int endPosPrefix = posPrefix + prefix.length();
   final int endPosName = line.indexOf("(", endPosPrefix);

   if (posPrefix >= 0 && endPosName >= 0)
   {
      return line.substring(endPosPrefix, endPosName);
   }
   return "";
}
```

Aufbereiten eines Histogramms

Aus den ermittelten Daten wollen wir ein Histogramm erzeugen. Dazu müssen Aufrufe an die gleiche Methode zusammengefasst werden. Dazu nutzen wir die Utility-Klasse Collectors und insbesondere die Aufrufe von `groupingBy()`sowie `counting()`, die wir der Lesbarkeit halber statisch importieren:

```
public static <T> Map<T, Long> histogram(final Stream<T> stream,
                                         final Function<T, T>
groupFunction)
{
   return stream.collect(groupingBy(groupFunction, counting()));
}
```

Die Methode machen wir generisch und weshalb dort eine `Function<T,T>` übergeben wird. In diesem Fall wissen wir, dass die Funktionalität auch an anderer Stelle benötigt wird und führen daher schon etwas mehr Komplexität als für diesen Anwendungsfall notwendig ein.

Histogramme der Aufrufe und Aufrufzeiten

Mit den bisher vorgestellten Basisbausteinen können wir nun mit der Implementierung der Extraktion und der Aufbereitung der Performance-Analysedaten beginnen. Im einfacheren Fall der Auswertung der Anzahl der Aufrufe extrahieren wir zunächst alle relevanten Zeilen, ermitteln daraus den jeweiligen Aufruf und fassen das Ganze zu

einem Histogramm zusammen, wobei wir hier die Identität als Funktion verwenden:

```
public static int thresholdInMs = 500;

public static Map<String, Long> analyzeCallCounts(final Path
pathToLogFile)
                                throws IOException
{
   final Stream<String> relevantLines =
extractRelevantLines(pathToLogFile,
thresholdInMs);
   final Stream<String> calls = relevantLines.map(line ->
extractCall(line));

   final Map<String, Long> callCountHistogram = histogram(calls,
Function.identity());
   return callCountHistogram;
}
public static Map<String, Long> analyzeCallTimes(final Path
pathToLogFile)
                                throws IOException
{
   final Stream<String> relevantLines =
extractRelevantLines(pathToLogFile,
thresholdInMs);

   final Map<String, Long> callTimeHistogram = relevantLines.collect(
                                    toMap(line -> extractCall(line),
                                          line -> 0L +
extractCallTime(line),
                                          (time1, time2) -> time1 +
time2));
   return callTimeHistogram;
}
```

Wie man leicht sieht, ist die Aufbereitung der Informationen zur Gesamtsumme der Aufrufzeiten pro Methode ein wenig komplexer. Wir profitieren hier wiederum von

den Neuerungen aus Java 8 und im Speziellen von der Methode `toMap()`. Diese erhält einen Schlüsselextraktor und danach für die Werte eine Berechnungsvorschrift für das erste Auftreten und die Kombination bei vorhandenem Wert. Weil wir an der Gesamtzeit interessiert sind, nutzen wir hier eine einfache Addition, was wir auch als Methodenreferenz `Long::sum` schreiben könnten.

Grafische Darstellung

Zur Erleichterung der Analyse ist eine grafische Darstellung wünschenswert. Hier können wir von den vielfältigen in JavaFX verfügbaren Charts profitieren. Wir wählen das `BarChart<X,Y>`, das eine Spezialisierung von `XYChart<X,Y>` ist.

Aufbereitung der Datenmodelle

Die bislang aufgesammelten Daten liegen als `Map<K,V>` vor. Allerdings erwarten die JavaFX-Chart-Komponenten die Daten nicht als `Map<K,V>`, sondern in einem speziellen Datenmodell, und zwar abhängig vom verwendeten Charttyp. Die Transformation von einer `Map<K,V>` in das hier benötigte Format einer `ObservableList<XYChart.Series<String, Number>>` lässt sich ohne viel Mühe wie folgt umsetzen:

```
public static ObservableList<XYChart.Series<String, Number>>
                    toXyChartDataForMap(final String name,
                                        final Map<String, Long> map)
{
   final Series<String, Number> series = new Series<>();
   series.setName(name);

   for (final Map.Entry<String, Long> entry : map.entrySet())
   {
      series.getData().add(new XYChart.Data<>(entry.getKey(),
                                              entry.getValue()));
   }

   final ObservableList<XYChart.Series<String, Number>> chartData =
   observableArrayList();
   chartData.add(series);
   return chartData;
}
```

Darüber hinaus benötigen wir eine Methode, die die erzeugten Daten dem eigentlichen `BarChart<String,Number>` zuordnet. Damit bei sich verändernden Daten auch die Darstellung korrekt erfolgt, musste ich den im Listing gezeigten Trick des Aktivierens und Deaktivierens der Animation nutzen:

```
public static void populateBarChart(final BarChart<String, Number>
barChart,
                                    final String name,
                                    final Map<String, Long>
originalData)
{
   barChart.setAnimated(true);

   final ObservableList<Series<String, Number>> dataForMap =
                                       toXyChartDataForMap(name,
originalData);
   for (final Series<String, Number> currentSeries : dataForMap)
   {
      barChart.getData().add(currentSeries);
   }
   barChart.setAnimated(false);
}
```

Danach ist es nur noch ein kleiner Schritt zur Darstellung ähnlich zu Abbildung 1-1:

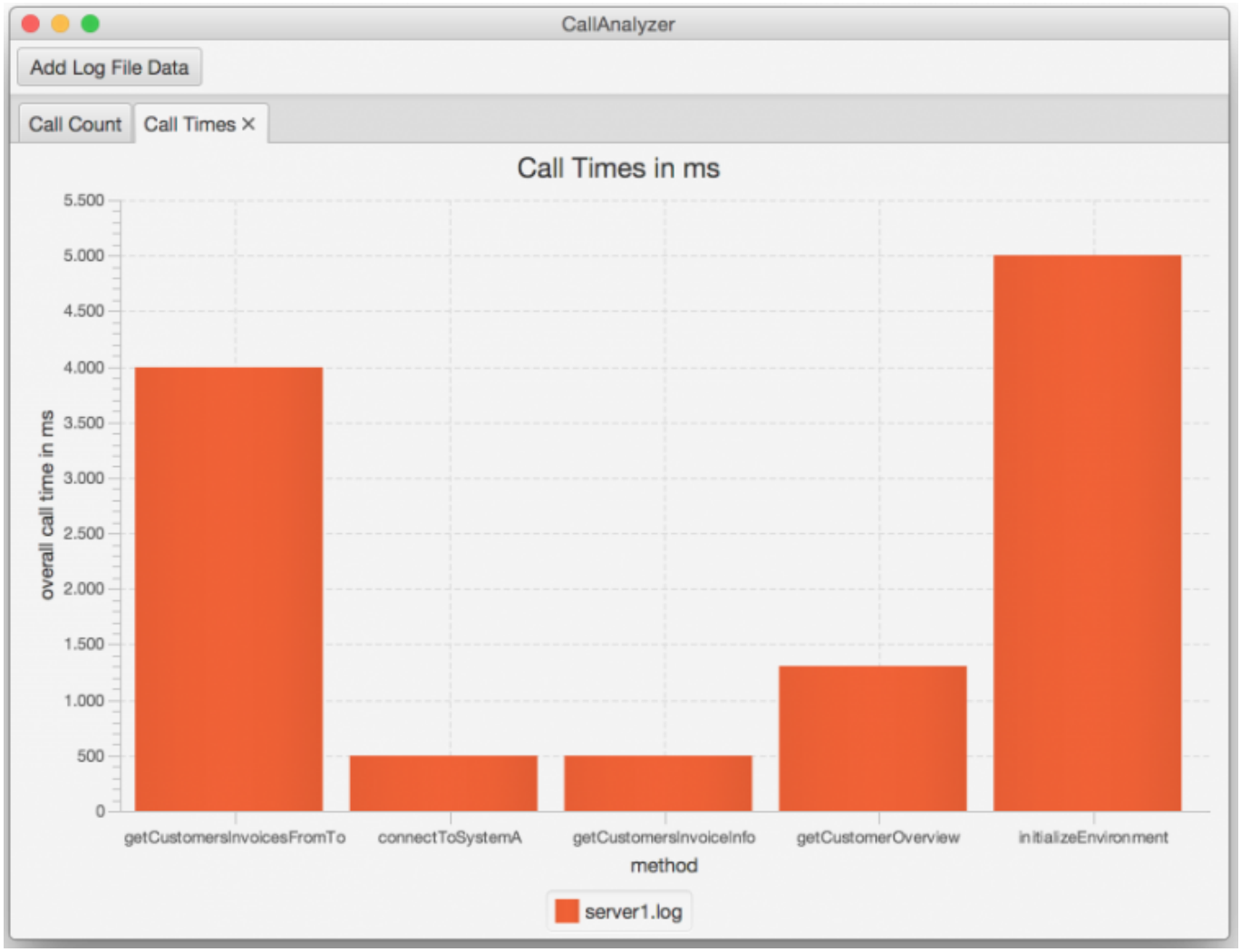

Abbildung 1-1: Darstellung des Programms CALLANALYZER

Feintuning der Funktionalität – Maps nach Wert sortieren

Wenn man nun aber diese Werte – wie eingangs angedeutet – zur Bestimmung der Hotspots einsetzen möchte, wäre eine Sortierung optimal. Überlegen wir kurz: Die Werte sind in einer `Map<K,V>` gespeichert. Bei Sortierungen und Maps kommt einem eine `TreeMap<K,V>` in den Sinn. So verlockend deren Einsatz scheint, ist dies jedoch nicht ausreichend für unsere Anforderungen. Warum? Eine `TreeMap<K,V>` sortiert die Schlüssel. In diesem Fall würden die Einträge also gemäß dem Methodennamen alphabetisch geordnet und nicht nach dem Wert, also nicht nach Ausführungsdauer bzw. Anzahl an Aufrufen. Praktischerweise kann man einer `TreeMap<K,V>` bei ihrer Konstruktion einen `Comparator<T>` übergeben.

Dies setzt voraus, dass die Werte den Typ `Comparable<T>` besitzen, was hier für Long gegeben ist. Beim Vergleich der Schlüssel erfolgt hier ein Zugriff auf die Werte, die verglichen werden. Anschließend wird das Ergebnis dieses Vergleichs zurückgeliefert. Dabei muss man noch beachten, dass bei Gleichheit der Werte explizit nicht der Wert 0 zurückgegeben werden darf, weil man sonst gleiche Werte fälschlicherweise zusammenfassen würde.

Übrigens ist es unbedeutend, ob der Wert 0 auf den Wert 1 oder -1 abgebildet wird. Mit diesem Wissen schreiben wir folgende Hilfsmethode zum Sortieren einer Map nach Wert (die Idee entstammt dieser Seite):

```
public static Map<String, Long> sortMapByValue(final Map<String, Long>
map)
{
   // Keine Behandlung von null-Werten, weil hier nicht benötigt
   class ValueComparator implements Comparator<String>
   {
      public int compare(final String first, final String second)
      {
         final int result = Long.compare(map.get(second),
map.get(first));
         if (result == 0) // Der Wert 0 würde ungewünschte Ergebnisse
liefern
         {
            return -1;
         }
         return result;
     }
   }

   final ValueComparator comparator = new ValueComparator();
   final TreeMap<String, Long> valueSortedMap = new
TreeMap<>(comparator);
   valueSortedMap.putAll(map);
   return valueSortedMap;
}
```

Abschließende Verbesserungen des Komparators

Wenn man möchte, kann man nach den numerisch sortierten Werten zusätzlich auch die Schlüssel alphabetisch sortieren. Erneut kann man von den Neuerungen in Java 8 (hier im Speziellen von den Komparatoren) profitieren. Schlussendlich modifizieren wir die Methode `compare(String, String)` folgendermaßen:

```
public int compare(final String first, final String second)
{
   final Comparator<String> longCompare =
Comparator.comparingLong(map::get);
   // Achtung: Größte Werte zuerst, daher hier andere
Parameterreihenfolge
   final int result = longCompare.compare(second, first);
   if (result == 0)
   {
      return first.compareTo(second);
   }
   return result;
}
```

Wenn man diese verbesserte Variante eines Komparators nutzt, bekommt man eine Darstellung ähnlich zu der in Abbildung 1-2.

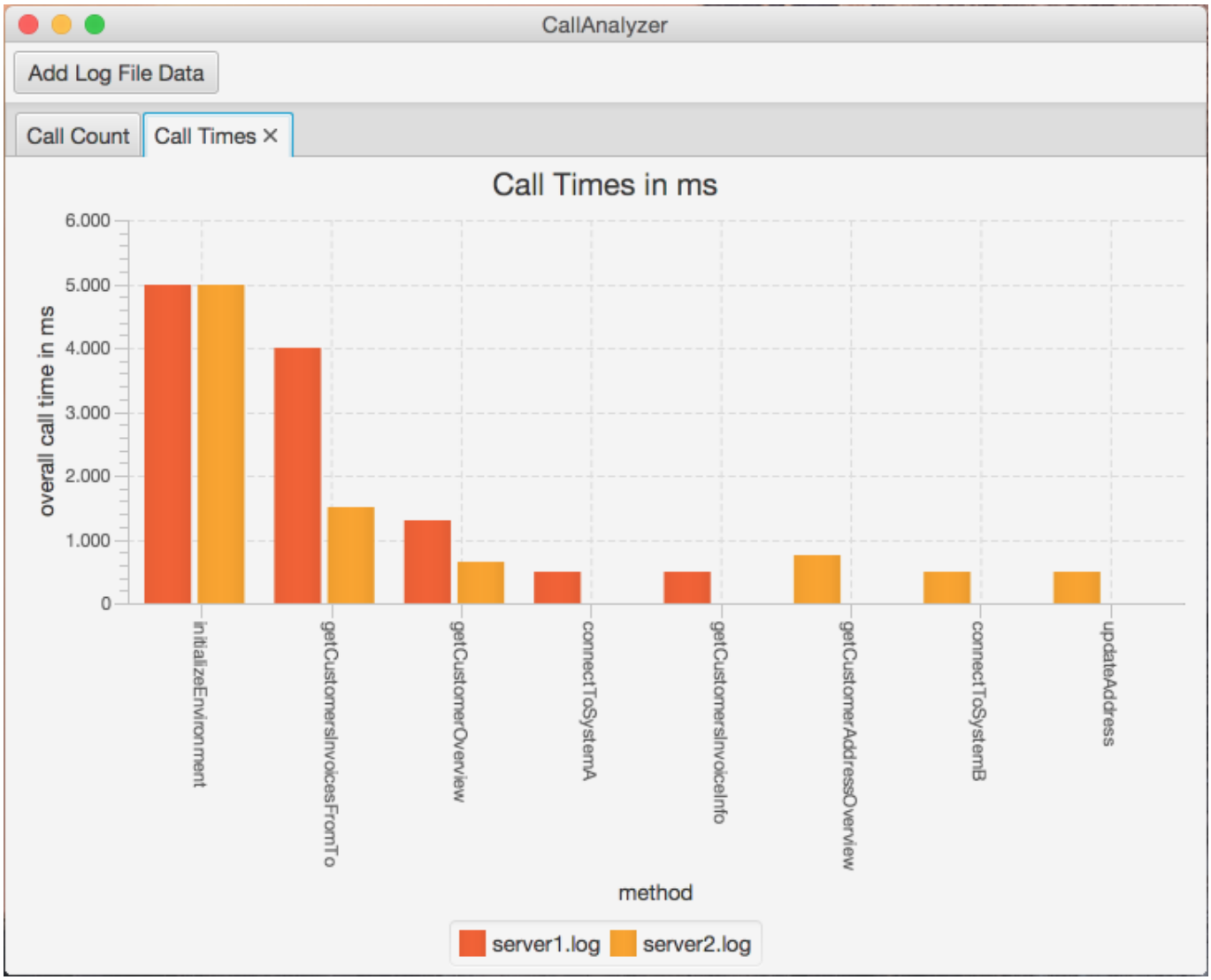

Abbildung 1-2: Darstellung des Programms CALLANALYZER

Fazit

Das hier vereinfacht vorgestellte Tool bietet in der Realität komplexere Auswertungen. Zudem ist die grafische Darstellung ausgeklügelter: Neben Zooming und Panning kann man die Daten auf die Top 20 beschneiden und die Grafiken ausdrucken. All das sind Ideen, die Sie als Anregung für eigene Erweiterungsversuche dieses Tools nutzen können. Insbesondere wollte ich Ihnen zeigen, dass man mit überschaubarem Aufwand nützliche Applikationen erstellen kann, wenn man die richtigen Techniken geeignet kombiniert.

Ohne den Einsatz der Sprachfeatures von Java 8 wäre etwas Ähnliches ungleich schwieriger zu realisieren. Zum einen fehlen einem die Streams und deren Möglichkeiten zur

Verarbeitung von Daten. Zum anderen bietet Swing keine Charts und man müsste entweder eine externe Bibliothek einbinden oder aber etwas selbst schreiben, was unverhältnismäßig aufwändig wäre.

Literatur

Für eine ausführliche Darstellung der Neuerungen in Java 8 verweise ich Sie auf meine Bücher:

- Der Weg zum Java-Profi, 3. Auflage, 2015, dpunkt.verlag
- Java 8 – Die Neuerungen, 2. Auflage, 2015, dpunkt.verlag

Möchten Sie mehr zu JavaFX erfahren, werfen Sie doch einen Blick in das Buch von Anton Epple:

- JavaFX, Anton Epple, 2015, dpunkt.verlag

Kurs zum Thema

Die Zühlke Academy bietet einen eintägigen Java 8 Workshop an, der Ihnen einen fundierten Überblick über die mit Java 8 eingeführten Neuerungen gibt.