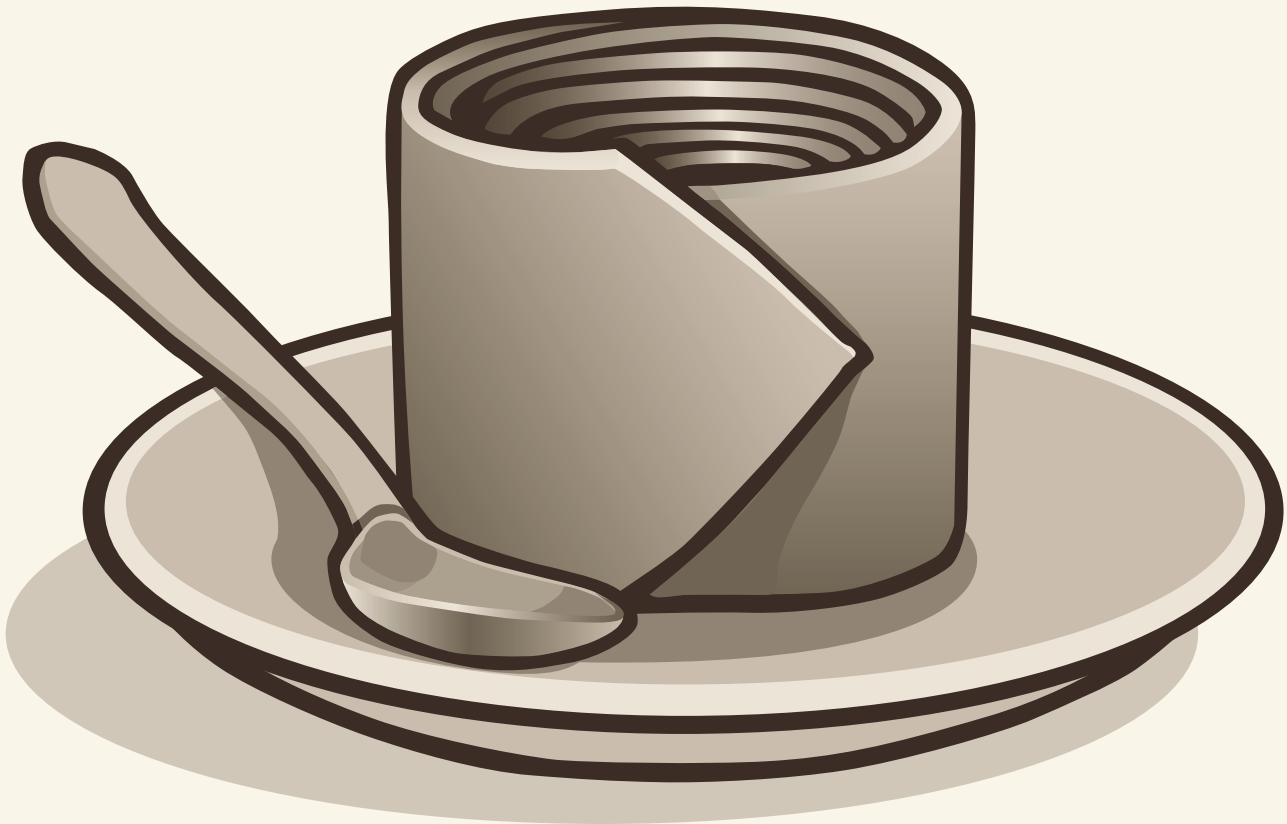
KaffeeKlatsch

Das Magazin rund um Software-Entwicklung

ISSN 1865-682X

09/2014

Jahrgang 7



Editorial

Warum?

Neulich habe ich mich mit einem Freund getroffen. So wie es aussieht, wechselt er demnächst den Arbeitgeber. Im Laufe des Gesprächs entwickelte sich eine Gedankenkette, die zu einer ganz einfachen Frage führte. Wenn der Arbeitgeber nach einem Mitarbeiter mit einem ganz bestimmten Profil sucht und diesen auch findet, warum muss dieser dann nach der Einstellung überhaupt noch besser werden?

Ungeachtet dessen, wie man die Güte eines Software-Entwicklers bestimmt, lässt sich diese Frage gar nicht so einfach beantworten. Dabei helfen auch keine Sprüche wie etwa: „Wer aufhört, besser zu werden, hat aufgehört, gut zu sein!“[1] Wenngleich sie zeigen, dass das Verbessern an und für sich eine Notwendigkeit des Fortschritts zu sein scheint oder diesen mit sich bringt.

Sowohl aus Sicht des Arbeitgebers als auch aus der des Arbeitnehmers genügt ja eigentlich ein lokales Maximum – der „Beste“ im Team zu sein (was immer das bedeutet) ist also völlig ausreichend. Wenn dann auch noch die geschriebene Software funktioniert, gibt es noch nicht einmal einen Grund, warum man sich verbessern sollte. Wenn das Team dann auch noch zeitig zum gewünschten Termin und im Rahmen des Budgets fertig wird, gibt es auch aus wirtschaftlicher Sicht keinen Anlass, das aktuell etablierte Vorgehen zu ändern.

Wenn man ein bisschen im Internet forscht, findet man sehr viel darüber heraus, wie man ein besserer Programmierer wird. So wird dort etwa empfohlen, auch mal Bücher zu lesen, jedes Jahr eine neue Programmiersprache zu lernen, jährlich ein neues Spaßprojekt zu beenden und an Programmierwettbewerben teilzunehmen. Oder man soll eine Fremdsprache lernen, Chinesisch oder Arabisch; also eine Sprache, die mit der unsrigen relativ wenig zu tun hat.

Gelegentlich wird man dort auch dadurch ein guter Programmierer, wenn man bestimmte Code-Idiome verwendet, eine bestimmte Technik beherrscht oder *Patterns* und *Frameworks* kennt. Auch werden oft Metriken definiert, die den Fortschritt des Besserwerdens beschreiben: Kürzere Formulierungen, sprechendere Bezeichner, bessere Dokumentation oder eine bessere Testabdeckung.

Darüber hinaus kann man erfahren, was einen guten, „echten“ Programmierer ausmacht – unabhängig davon, wie er zu den Fähigkeiten gekommen ist. Ein guter Programmierer kann in *Assembler* programmieren, kennt alle *include*-Dateien der *C*-Bibliothek, zitiert aus *Der Herr der Ringe, Star Wars* oder *Monty Python* und hat einen personalisierten Scheck über 2,56 $ von Donald Knuth an der Wand hängen.

Aber kaum eine Seite im Internet kümmert sich darum, die Frage nach dem Warum zu beantworten. Um besser werden zu können, muss sich zumindest der Kandidat im Klaren darüber sein, warum er denn für sich selbst besser werden will. Das setzt aber voraus, dass überhaupt das Bewusstsein und der Wille zu einer Veränderung da ist, was aber nicht bei jedermann gegeben scheint.

Es gibt viele Gründe, warum man besser werden und sich weiterentwickeln muss. Aber der für mich wichtigste Grund ist die Konkurrenzfähigkeit und die daraus resultierende Flexibilität, auch über das eigene Unternehmen hinaus. Denn nur so ist und bleibt man unabhängig von seinem Arbeitgeber, was in allen Zeiten von unschätzbarem Wert ist.

Möge also auch der KaffeeKlatsch zur Erweiterung des Horizonts beitragen…

Ihr Michael Wiedeking

[1] Wird in diversen Internet-Quellen (leider ohne konkreten Beleg) Robert Bosch nachgesagt.

Beitragsinformation

Der KaffeeKlatsch dient Entwicklern, Architekten, Projektleitern und Entscheidern als Kommunikationsplattform. Er soll neben dem Know-how-Transfer von Technologien (insbesondere Java und .NET) auch auf einfache Weise die Publikation von Projekt- und Erfahrungsberichten ermöglichen.

Beiträge

Um einen Beitrag im KaffeeKlatsch veröffentlichen zu können, müssen Sie prüfen, ob Ihr Beitrag den folgenden Mindestanforderungen genügt:

- Ist das Thema von Interesse für Entwickler, Architekten, Projektleiter oder Entscheider, speziell wenn sich diese mit der Java- oder .NET-Technologie beschäftigen?
- Ist der Artikel für diese Zielgruppe bei der Arbeit mit Java oder .NET relevant oder hilfreich?
- Genügt die Arbeit den üblichen professionellen Standards für Artikel in Bezug auf Sprache und Erscheinungsbild?

Wenn Sie uns einen solchen Artikel, um ihn in diesem Medium zu veröffentlichen, zukommen lassen, dann übertragen Sie Bookware unwiderruflich das nicht exklusive, weltweit geltende Recht

- diesen Artikel bei Annahme durch die Redaktion im KaffeeKlatsch zu veröffentlichen
- diesen Artikel nach Belieben in elektronischer oder gedruckter Form zu verbreiten
- diesen Artikel in der Bookware-Bibliothek zu veröffentlichen
- den Nutzern zu erlauben diesen Artikel für nicht-kommerzielle Zwecke, insbesondere für Weiterbildung und Forschung, zu kopieren und zu verteilen.

Wir möchten deshalb keine Artikel veröffentlichen, die bereits in anderen Print- oder Online-Medien veröffentlicht worden sind.

Selbstverständlich bleibt das Copyright auch bei Ihnen und Bookware wird jede Anfrage für eine kommerzielle Nutzung direkt an Sie weiterleiten.

Die Beiträge sollten in elektronischer Form via E-Mail an redaktion@bookware.de geschickt werden.

Auf Wunsch stellen wir dem Autor seinen Artikel als unveränderlichen PDF-Nachdruck in der kanonischen KaffeeKlatsch-Form zur Verfügung, für den er ein unwiderrufliches, nicht-exklusives Nutzungsrecht erhält.

Leserbriefe

Leserbriefe werden nur dann akzeptiert, wenn sie mit vollständigem Namen, Anschrift und E-Mail-Adresse versehen sind. Die Redaktion behält sich vor, Leserbriefe – auch gekürzt – zu veröffentlichen, wenn dem nicht explizit widersprochen wurde.

Sobald ein Leserbrief (oder auch Artikel) als direkte Kritik zu einem bereits veröffentlichten Beitrag aufgefasst werden kann, behält sich die Redaktion vor, die Veröffentlichung jener Beiträge zu verzögern, so dass der Kritisierte die Möglichkeit hat, auf die Kritik in der selben Ausgabe zu reagieren.

Leserbriefe schicken Sie bitte an leserbrief@bookware.de. Für Fragen und Wünsche zu Nachdrucken, Kopien von Berichten oder Referenzen wenden Sie sich bitte direkt an die Autoren.

Werbung ist Information

Firmen haben die Möglichkeit Werbung im KaffeeKlatsch unterzubringen. Der Werbeteil ist in drei Teile gegliedert:

- Stellenanzeigen
- Seminaranzeigen
- Produktinformation und -werbung

Die Werbeflächen werden als Vielfaches von Sechsteln und Vierteln einer DIN-A4-Seite zur Verfügung gestellt.

Der Werbeplatz kann bei Frau Natalia Wilhelm via E-Mail an anzeigen@bookware.de oder telefonisch unter 09131/8903-16 gebucht werden.

Abonnement

Der KaffeeKlatsch erscheint zur Zeit monatlich. Die jeweils aktuelle Version wird nur via E-Mail als PDF-Dokument versandt. Sie können den KaffeeKlatsch via E-Mail an abo@bookware.de oder über das Internet unter www.bookware.de/abo bestellen. Selbstverständlich können Sie das Abo jederzeit und ohne Angabe von Gründen sowohl via E-Mail als auch übers Internet kündigen.

Ältere Versionen können einfach über das Internet als Download unter www.bookware.de/archiv bezogen werden.

Auf Wunsch schicken wir Ihnen auch ein gedrucktes Exemplar. Da es sich dabei um einzelne Exemplare handelt, erkundigen Sie sich bitte wegen der Preise und Versandkosten bei Natalia Wilhelm via E-Mail unter natalia.wilhelm@bookware.de oder telefonisch unter 09131/8903-16.

Copyright

Impressum

KaffeeKlatsch Jahrgang 7, Nummer 9, September 2014
ISSN 1865-682X
Bookware – eine Initiative der
MATHEMA Verwaltungs- und Service-Gesellschaft mbH

Henkestraße 91, 91052 Erlangen
Telefon: 0 91 31 / 89 03-0
Telefax: 0 91 31 / 89 03-55
E-Mail: redaktion@bookware.de
Internet: www.bookware.de

Herausgeber/Redakteur: Michael Wiedeking
Anzeigen: Natalia Wilhelm
Grafik: Nicole Delong-Buchanan

Inhalt

Artikel

Kolumnen

Alte Zöpfe – neue Besen – Seids gewesen?

von Frank Ganske

In Java gibt es schon seit J2SE 1.4 ein Paket mit dem Namen java.nio, das zumindest ich mir mit New I/O übersetzt und gedanklich zu den Akten gelegt hatte. In Unternehmensanwendungen kommt man mit diesen Gefilden ja selten in Berührung. Außerdem bringt dieses java.nio nichts, dass man mit java.io nicht erledigen konnte. Ja – aber: das NIO bedeutet Non-blocking I/O [1] und enthält Technologien, die bestimmte Aufgaben deutlich effektiver erledigen. Es ist kein ersetzendes, sondern ein alternatives I/O.

Vorbelegt

von Ilker Yümsek

Anhand von weniger komplexen Aufgabenstellungen kann man sich mit neuen Funktionalitäten und Änderungen in einer Programmiersprache besser auseinandersetzen. In diesem Artikel geht es um die Nutzung der mit Java 8 eingeführten Default-Methoden.

Summa summarum

von Michael Wiedeking

Wie auch immer man seine Zahlen repräsentiert, irgendwann einmal kommt der Zeitpunkt, an dem man sie addieren oder subtrahieren möchte. Aber auch hier stellt sich die Frage, wie man dies am besten anstellt.

Alte Zöpfe – neue Besen – Seids gewesen?

Java I/O und Netty (NIO), Top Down gemachte Erfahrungen mit einem Framework

von Frank Ganske

In Java gibt es schon seit J2SE 1.4 ein Paket mit dem Namen java.nio, das zumindest ich mir mit New I/O übersetzt und gedanklich zu den Akten gelegt hatte. In Unternehmensanwendungen kommt man mit diesen Gefilden ja selten in Berührung. Außerdem bringt dieses java.nio nichts, dass man mit java.io nicht erledigen konnte.
Ja – aber: das NIO bedeutet Non-blocking I/O [1] und enthält Technologien, die bestimmte Aufgaben deutlich effektiver erledigen. Es ist kein ersetzendes, sondern ein alternatives I/O.

Es ist völlig anders und man kann viele neue Dinge falsch machen. Spätestens wenn man mit einem modernen I/O-Framework wie *Netty* [2] arbeitet, muss man Java NIO kennen und anwenden können, ohne seine Vorteile wieder zunichte zu machen.

Hintergrund

Datenverarbeitung gliedert sich immer in Eingabe/Verarbeitung/Ausgabe. Der Prozess ist also entscheidend von Eingabe und Ausgabe, kurz I/O, abhängig. Hier werden wesentliche nicht funktionale Anforderungen, wie Geschwindigkeit und Speicherverbrauch, erfüllt. Tatsächlich war Netty der Auslöser, sich mit dem Begriff NIO zu beschäftigen. Mit Netty wurde ein Projekt realisiert, dessen Netzwerkkomponente in der Android-App und in der Desktop-Anwendung eingesetzt wird. Daten aus Netty werden analysiert und in Dateien geschrieben – Dateien gelesen, verändert und an Netty übergeben. Netty ist ein NIO-Framework, das die Entwicklung von Netzwerkanwendungen, eigener Protokolle, Clients und Server „quick and easy“ ermöglicht. Netty ist ab Java 1.5 verfügbar und kann unter Android verwendet werden [3]. Einerseits erweitert es die NIO-Technologien, wie mit der Klasse *io.netty.buffer.ByteBuf* statt *java.nio.ByteBuffer*. Andererseits stellt es auch Adapter zum klassischen I/O, wie *io.netty.buffer.ByteBufInputStream /ByteBufOutputStream* bereit. Es gibt also eine Menge neuer Begriffe, die im Auge zu behalten sind. Also erst mal zurück auf Start.

Vergleich zwischen Java I/O und NIO

Das klassische Java I/O behandelt Ein- und Ausgaben mit Streams, zeichenorientiert *Reader/Writer* genannt. Die vielfältigen Aufgaben können durch Kombination bzw. Aneinanderreihung solcher Filter komfortabel gelöst werden. Alles ist in Java implementiert, plattformunabhängig und vertraut. Das bedeutet jedoch, dass es im Kern immer das Byte-weise Kopieren von *Arrays* mit Java ist. NIO nutzt dagegen integrale I/O-Konzepte heutiger Betriebssysteme und macht sie verfügbar. Als Beispiel kann es Speicherbereiche des Systems referenzieren, ohne sie erst in ein *Byte-Array* zu kopieren und sie dabei sowohl lesen als auch schreiben. Das kann zu massiven Geschwindigkeitsvorteilen führen. Java delegiert dazu Verantwortung an die Plattform und das führt zu unterschiedlichem Verhalten. Die Verwaltung solcher Operationen, wenn es nicht um einen Zehnzeiler zum unveränderten Kopieren einer Datei oder zum simplen Ausgeben von Zeichen auf die Konsole geht, ist komplexer.

Neue Begriffe und Konzepte

Für jede Ein- und Ausgabeoperation werden *java.nio.Channel* und *java.nio.Buffer* benötigt. Sie bilden den Kern der API von NIO.

Channel dienen zum Zugriff auf Daten. Sie sind an jeder Lese- oder Schreiboperation beteiligt. Daten werden vom Channel in den *Buffer* gelesen, sowie vom Buffer in den Channel geschrieben. Ein Channel kann sowohl zum Lesen als auch zum Schreiben verwendet werden. Bei *Streams* ist Lesen und Schreiben getrennt implementiert. Einige Channel-Implementierungen können mit einem *Selector* asynchron gelesen und geschrieben werden, ohne zu blockieren. Das bedeutet dann keine Garantie, dass die erwarteten Daten schon da sind [4]. Channel benötigen immer Buffer (bzw. Systemressourcen beim *Channel to channel transfer* im Beispiel weiter unten).

Buffer sind die hinzugekommenen Datenhalter, die den deutlichsten Unterschied zwischen Java NIO und dem klassischen Java I/O ausmachen. Im Stream-orientierten I/O wurden die Daten direkt mit den Stream-Objekten behandelt, jetzt sind immer Buffer beteiligt. Buffer kommunizieren mit dem Channel und können auch direkt Daten erhalten und abgeben. Ihre Methoden bilden die Schnittstelle zu den Daten der Anwendung.

Der grundsätzliche Prozess zur Nutzung eines Buffer ist das Schreiben in den Buffer, der Aufruf der Methode *flip()*, das Lesen aus dem Buffer und der Aufruf der Methoden *clear()* oder *compact()*. Der Buffer verwaltet, neben seiner Kapazität, wie viele Daten in ihn geschrieben wurden.

Beispiel: HTML-Response von Netty in Datei schreiben

Das folgende Beispiel benötigt die Downloads *Apache IOUtils* [5] und Netty [6]. Von Netty erhält man so etwas wie den *DefaultFullHttpResponse*, der einen *ByteBuf* enthält. Falls der *Netty-ByteBuf* mehrere NIO-Buffer enthält, werden sie in einer Schleife ausgegeben. Die Header sind Textinformationen, die über einen *java.nio.charset.Charset* codiert werden müssen, bevor sie als *ByteBuffer* geschrieben werden.

```
package snippet;

import io.netty.buffer.ByteBuf;
import io.netty.handler.codec.http.DefaultFullHttpResponse;
import io.netty.handler.codec.http.FullHttpResponse;
import io.netty.handler.codec.http.HttpResponseStatus;
import io.netty.handler.codec.http.HttpVersion;

import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.charset.Charset;

import org.apache.commons.io.IOUtils;

public class ChannelWriteSnippet {

  private static final Charset DEFAULT_ENCODING =
    Charset.forName("ISO-8859-1");

  public static void main(String[] args)
    throws IOException {
    // Ein leeres Beispiel ...
    FullHttpResponse content =
      new DefaultFullHttpResponse(
        HttpVersion.HTTP_1_1, HttpResponseStatus.OK
      );
    String headerText = "header text...";
    ByteBuf nettyBuffer = content.content();

    File tempFile = new File("ChannelWriteSnippet.txt");
    if (!tempFile.exists()) {
      tempFile.createNewFile();
    }
    RandomAccessFile output = null;
    FileChannel channel = null;
    try {
      output = new RandomAccessFile(tempFile, "rw");
      channel = output.getChannel();
      ByteBuffer headerBuffer =
        DEFAULT_ENCODING.encode(headerText);
      channel.write(headerBuffer);
      for (ByteBuffer each : nettyBuffer.nioBuffers()) {
        channel.write(each);
      }
    } finally {
      IOUtils.closeQuietly(channel);
      IOUtils.closeQuietly(output);
    }
  }

}
```

Beispiel: Textzeilen aus einem ByteBuffer lesen

In Java I/O gab es mit dem *BufferedReader* eine komfortable Möglichkeit Text zeilenweise zu lesen. Verwendet man das mit einem *InputStreamReader* und einem *ByteBufInputStream*, und will wie im Beispiel anschließend den binären Teil der Nachricht einlesen, funktioniert das nicht. Der Puffer des *BufferedReader* hat schon über die letzte Zeile hinaus gelesen. Wird der Stream weiter gelesen, fehlen Daten.

Das folgende Beispiel zeigt das Byte-weise Lesen von Daten aus einem *NIO-ByteBuffe*r zur Ermittlung eines Nachrichten-Headers. Besonders ist hier die *get*()-Methode mit *Cast* auf *char* statt der *getChar*()-Methode, um die richtigen Zeichen aus dem *ByteBuffer* zu bekommen. Der *ByteBuffer* kann so direkt zum Lesen der folgenden binären Daten weiter verwendet werden, nachdem die Leerzeile erkannt wurde. Ohne Leerzeile wird hier eine *Exception* ausgelöst, weil die Nachricht ungültig ist.

```
package snippet;

import java.nio.BufferUnderflowException;
import java.nio.ByteBuffer;
import java.util.ArrayList;
import java.util.List;

public class ReadLinesSnippet {

  private static final char CR = 13, LF = 10;
```

```
    private List<String> readLines(ByteBuffer byteBuffer)
        throws BufferUnderflowException {
      List<String> lines = new ArrayList<String>();
      if (byteBuffer.remaining() == 0) {
        return lines;
      }
      StringBuilder b = new StringBuilder();
      char c;
      while (true) {
        String line = "";
        while (true) {
          c = (char) byteBuffer.get();
          if (c == LF) {
            line = b.toString();
            b.setLength(0);
            break;
          } else if (c == CR) {
            line = b.toString();
            b.setLength(0);
            char c2 = (char) byteBuffer.get();
            if (c2 == CR) {
              return lines; // empty line
            }
            if (c2 != LF) {
              b.append(c2);
              c = c2;
            }
            break;
          } else {
            b.append(c);
          }
        }
        if (line.isEmpty()) {
          return lines; // empty line
        }
        lines.add(line);
      }
    }
  }
```

Verhalten und Eigenschaften von Buffern

Der am häufigsten verwendete NIO-Buffer ist *java.nio.buffer.ByteBuffer.* Zum strukturierten Schreiben und Lesen sind Methoden für die primitiven Datentypen vorhanden, aber es gibt weiter spezialisierte Buffer für Datentypen oder zeichenorientierte Daten.

Mit *flip()* muss man den Modus umschalten, um Daten zu lesen. Erst danach ist die Positionsmarke entsprechend gesetzt – *flip()* darf nicht vergessen werden. Netty vermeidet diese Stolperfalle in seiner Basisklasse *ByteBuf* mit getrennten Positionen zum Lesen und Schreiben.

Nach dem Auslesen muss der Buffer freigegeben werden. Die Methode *clear()* löscht den Inhalt. Die Methode *compact()* löscht den gelesenen Bereich. Neue Daten werden dann hinter dem ungelesenen Rest geschrieben.

Ein Buffer enthält einen Speicherbereich, in den man Daten schreibt und später wieder daraus liest. Das Objekt unterstützt den Zugriff mit Methoden und verwaltet dazu drei wesentliche Eigenschaften: *Capacity, Position* und *Limit*. Dabei sind Limit und Position abhängig vom Schreib- oder Lesemodus.

Die Capacity, dass Fassungsvermögen eines Speicherbereichs, ist zunächst eine feste Größe. Ist die Capacity erreicht, muss der Buffer geleert werden, um ihn wieder neu zu füllen.

Die Position, die Schreib- oder Lesemarkierung, ist ein Index ab 0. Geschriebene Daten erhöhen den Zähler, maximal bis zum Wert der Capacity –1. Wird der Buffer mit *flip()* in den Lesemodus geschaltet, beginnt der Zähler wieder bei 0 und wird entsprechend erhöht.

Das Limit ist im Schreibmodus identisch mit der Capacity. Im Lesemodus zeigt es an, wie viel Daten lesbar sind, weil der Buffer durchaus größer als die enthaltenen Daten dimensioniert sein kann. Mit *flip()* wird das Limit auf die Position des vorhergegangenen Schreibmodus gesetzt.

Anders als bei I/O-Streams können Buffer mit der Methode *rewind()* zurückgesetzt werden, um noch einmal zu lesen oder zu schreiben. Auch *mark()* und *reset()* zum Merken und Zurücksetzen von Positionen ist jederzeit möglich.

Sehr interessant ist, dass *NIO-CharBuffer* das Interface *CharSequence* implementieren und so direkt an die *Java Regular Expressions*-Implementierung in *java.util.regex* übergeben werden können.

Man erhält einen Buffer entweder über eine Schnittstelle, wie die Netzwerkdaten aus Netty oder instanziiert ihn über seine *Factory*-Methoden wie *java.nio.ByteBuffer.allocate()*. Außerdem gibt es *wrap()*-Methoden, die vorhandene Daten, wie *byte[]*, entsprechend dem Typ des Buffer übernehmen. Zur Entkopplung kann man die Methode *ByteBuf.copy()* einsetzen, die den Speicherbereich kopiert. Die Methoden *slice()* und *duplicate()*, die es auch in Java NIO gibt, erzeugen dagegen eine weitere Referenz mit neuen unabhängigen Schreib- und Lesemarken. Weil Änderungen in allen Referenzen wirken, sind unterschiedliche Restriktionen zu beachten. NIO-Buffer können mit der Methode *asReadOnlyBuffer* eine schreibgeschützte Referenz erzeugen.

Beispiele: Dateien kopieren mit Zeitmessung

Direct Buffer ermöglichen Implementierungen direkt mit der Speicherverwaltung des Betriebssystems zu arbeiten, um das System bestmöglich zu nutzen. Ein Beispiel ist der *MappedByteBuffer* für Dateien. Er ermöglicht

Memory-mapped file I/O, mit dem Kopiervorgänge beschleunigt werden können. Allerdings steht Messen vor blindem Vertrauen. Auf meinem alten Laptop unter Linux schaffen die Apache IOUtils mit I/O-Streams: Copied 1 GB in 27.710 ms. Das NIO-Beispiel gibt Copied 1 GB in 26.356 ms aus (Zieldateien nicht ersetzen, sondern neu schreiben):

```
package snippet;

import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.MappedByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.channels.FileChannel.MapMode;

import org.apache.commons.io.IOUtils;

public class MappedByteBufferCopySnipped {

  private static final int GB = 1024 * 1024 * 1024;

  public static void main(String[] args)
      throws IOException {
    RandomAccessFile source = null;
    RandomAccessFile target = null;
    try {
      File sourceFile = new File("source.img");
      if (!sourceFile.exists()) {
        sourceFile.createNewFile();
      }
      source = new RandomAccessFile(sourceFile, "rw");
      long length = 1 * GB;
      source.setLength(length);


      MappedByteBuffer sourceBuffer =
        source.getChannel().map(
          MapMode.READ_ONLY, 0, length
        );
      target = new RandomAccessFile(
        "target.img", "rw"
      );
      FileChannel targetChannel =
        target.getChannel();
      long ms = System.currentTimeMillis();
      while (sourceBuffer.hasRemaining()) {
        targetChannel.write(sourceBuffer);
      }
      System.out.println(
        String.format(
          "Copied %,d GB in %,dms",
          length / GB, System.currentTimeMillis() - ms
        )
      );
    } finally {
      IOUtils.closeQuietly(source);
      IOUtils.closeQuietly(target);
    }
  }

}
```

Es geht ebenso andersherum (Definition oben). Ja, mit der *read()*-Methode des *sourceChannel* und dem *targetBuffer* wird wieder in die Zieldatei geschrieben:

```
...
      target = new RandomAccessFile("target.img", "rw");
      MappedByteBuffer targetBuffer =
        target.getChannel().map(
          MapMode.READ_WRITE, 0, length
      );
      long ms = System.currentTimeMillis();
      sourceChannel.read(targetBuffer);
      // langsam!
...
```

Channel können direkt transferiert werden, wenn mindestens ein *FileChannel* beteiligt ist.

```
...
      FileChannel sourceChannel = source.getChannel();
      target = new RandomAccessFile("target.img", "rw");
      FileChannel targetChannel = target.getChannel();
      long ms = System.currentTimeMillis();
      targetChannel.transferFrom(sourceChannel, 0, length);
...
```

Auch das geht wieder andersherum:

```
...
      sourceChannel.transferTo(targetChannel, 0, length);
...
```

Vorsicht bei nicht blockierenden *SocketChanel*-Implementierungen. Hier kann die Übertragung unerwartet aufhören, wenn ein interner Buffer gefüllt ist. Dann fehlen Daten, ohne dass es eine Exception gibt. Ein solcher Fehler ist schwer zu lokalisieren.

Zum Vergleichen, Experimentieren und Nachmessen zeigt dieses Beispiel die klassische Implementierung mit Streams:

```
package snippet;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

import org.apache.commons.io.IOUtils;

public class StreamIoTransferSnippet {
  private static final int GB = 1024 * 1024 * 1024;
```

```
    public static void main(String[] args)
      throws IOException {
      InputStream is = null;
      OutputStream os = null;
      try {
        File source = new File("source.img");
        long length = source.length();
        is = new FileInputStream(source);
        os = new FileOutputStream("target.img");
        long ms = System.currentTimeMillis();
        IOUtils.copy(is, os);
        System.out.println(
          String.format(
            "Copied %,d GB in %,dms", length / GB,
            System.currentTimeMillis() - ms
          )
        );
      } finally {
        IOUtils.closeQuietly(is);
        IOUtils.closeQuietly(os);
      }
    }
}
```

Direct Buffer sind auf jedem System anders. JDK ist hier nicht gleich JDK und Android schon gleich gar nicht. Hier wurde in Netty viel Arbeit investiert, um das Verhalten zu vereinheitlichen. Netty bietet über die Faktories *Pooled* und *Unpooled* mit den Methoden *directBuffer()* eigene Implementierungen an, die ihre Capacity dynamisch anpassen können. Hier findet man auch *Wrapper*-Methoden für Java NIO-Buffer.

Die Netty *ByteBuf* sind wie NIO-Buffer eine Sicht auf Speicher, der sich aber aus mehreren Teilen zusammensetzen kann und gleichzeitig in mehreren Buffer referenziert sein kann. Die Umschaltung der Modi gibt es nicht. Dazu werden Lese- und Schreibzeiger, sowie Verwendungszähler verwaltet. Netty verwaltet den Speicher der Buffer explizit und verlässt sich nicht auf die *Garbage Collection*.

Bei der Nutzung von Netty ging es um das Lesen aus und das Schreiben in Dateien. Für Dateizugriff erhält ein Channel entweder *java.io.FileInputStream.getChannel(),* der ein *java.io.File* geöffnet hat, oder *java.io.RandomAccessFile.getChannel()*. Netty verwaltet die Channel der Netzwerkkommunikation und erhält oder übergibt Daten als *io.netty.buffer.ByteBuf*. Der *NettyByteBuf* gibt über die Methode *nioBuffer()* einen *java.nio.Buffer* zurück.

Netty ist mehr als NIO mit Channel und Buffer. Die Entscheidung für Netty wurde getroffen, weil es die Netzwerkprogrammierung so sehr vereinfacht. Es enthält *HTTP-Client-* und *Server-Codec*, *Decompressor*, *Chunk Aggregator* und vieles mehr, das man kombinieren kann. Dennoch wurde die Nutzung von Netty auf das Netzwerk-Paket beschränkt und die weitere Daten- und Dateiverarbeitung ausschließlich mit NIO umgesetzt. Diese Trennung ermöglicht die Austauschbarkeit von Netty.

Gegenanzeigen und Kritik

Wie eingangs gesagt ist Java NIO kein ersetzendes, sondern ein ergänzendes Verfahren. Es gibt durchaus Argumente, die gegen seinen Einsatz sprechen [4]. Die nicht blockierenden Channel erfordern ein hohes Maß an Verwaltung, weil man prüfen muss, ob alle Daten übermittelt wurden. NIO ermöglicht mehrere Verbindungen in einem *Thread* und spart damit Ressourcen. *Multithreading* wird von den heutigen Systemen aber immer besser unterstützt. Hat man gute, passende und flexible I/O-Komponenten und plant keine Hochlast, braucht man keine neuen Pfade einzuschlagen. Bei der Ressourcen-Situation auf einem mobilen Gerät ist NIO kein Pluspunkt. Klassisches I/O funktioniert auch auf ältesten Android-Versionen. Nutzt man ein Framework mit Non-blocking I/O, darf man das Messen, Testen und Vergleichen, auf allen geplanten Plattformen, nicht vernachlässigen.

Weitere Aspekte von Java NIO

Buffer und Channel sind nur ein Teil von Java NIO. *Selector* ermöglichen mehrere Verbindungen gleichzeitig. Eben diese Aufgaben sind in Netty umgesetzt. Diese Architekturentscheidung ist durch das Framework getroffen worden. Java NIO enthält mit *Scatter/Gather* eine Technik zur gesplitteten Übermittlung über Buffer und Cannel. Ein Channel überträgt mehrere Buffer und dabei nur die Daten zwischen Position und Limit. Das ist sehr nützlich für Netzwerkübertragung mit Paketen oder die Trennung einer Nachricht in Header (Text) und Inhalt (binär), die ganz unterschiedlich verarbeitet werden. Der N*etty-ByteBuf* kapselt das Konzept weiter und liefert mit *nioBufferCount()* und *nioBuffers()* alle ggf. enthaltenen NIO-Buffer.

Beispiel: Die Sache mit den Zeichensätzen

Auch Zeichensätze liegen im Paket java.nio. Die Ermittlung des korrekten Zeichensatzes aus HTTP/HTML ist komplex und in dieser Anwendung unnötig. Falls Umlaute bei der Verarbeitung irrelevant sind und nur richtig wieder zurück codiert werden müssen, kann man die aufwändige Zeichensatzerkennung weglassen. Der Zeichensatz ISO-8859-1 überträgt Bytes immer 1:1 in *Chars* und ist somit transparent, selbst wenn der Zei-

chensatz ein anderer ist. In Java werden Umlaute völlig falsch angezeigt, wenn sie statt ISO-8859-1 doch UTF-8 waren, was ja immer öfter vorkommt. Der Trick ist, falls man doch ausnahmsweise Umlaute benötigt, es einfach mit *Try* und *Error* zu versuchen.

Das Beispiel simuliert falsch als ISO-8859-1 gespeicherten UTF-8-Text und gibt ihn aus. Dann werden korrekter und falscher Text konvertiert angezeigt. Beide Ausgaben sind richtig. Besonders ist, statt *Charset.decode()* den *CharsetDecoder* mit den *REPORT-Actions* zu verwenden. Das Standardverhalten sind Fragezeichen, nur so wird eine *CharacterCodingException* ausgelöst.

```
package snippet;

import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.charset.CharacterCodingException;
import java.nio.charset.Charset;
import java.nio.charset.CharsetDecoder;
import java.nio.charset.CodingErrorAction;

public class WrongEncodingSnippet {

  private static final Charset DEFAULT_ENCODING =
    Charset.forName("ISO-8859-1");
  private static final Charset UTF8_ENCODING =
    Charset.forName("UTF-8");
  private static final String INPUT = "äöüßÄÖÜ€";

  public static void main(String[] args) {
    String wrongSaved =
      DEFAULT_ENCODING.decode(
        UTF8_ENCODING.encode(INPUT)
      ).toString();
    System.out.println("Wrong saved: " + wrongSaved);
    System.out.println(
      "Convertet input: " + convertEncoding(INPUT)
    );
    v.out.println(
      "Convertet wrong: "convertEncoding(wrongSaved)
    );
  }

  private static String convertEncoding(String input) {
    try {
      ByteBuffer in = ByteBuffer.wrap(
        input.getBytes(DEFAULT_ENCODING)
      );
      CharsetDecoder decoder =
        UTF8_ENCODING.newDecoder() //
      .onMalformedInput(
        CodingErrorAction.REPORT
      ) //
      .onUnmappableCharacter(
        CodingErrorAction.REPORT
      );
      CharBuffer buffer = decoder.decode(in);
      return buffer.toString();
    } catch (CharacterCodingException e) {
      return input;
    }
  }
}
```

Mit Java 7 wurde NIO um eine neue API für Dateizugriffe erweitert. Unter anderem gibt es ein neues *Zip*-Dateisystem. Links können bearbeitet und große Verzeichnisse über Buffer schneller aufgelistet werden.

Zusammenfassung

Der Artikel will kein umfassendes Tutorial über Java NIO sein. Er zeigt die wesentlichen Kernkonzepte und gibt dem Leser ein Gefühl für die Zusammenhänge in der Technologie.

Das völlig andere Vorgehen kann zu Überraschungen führen. Die Laufzeitumgebung wird verstärkt genutzt und das erfordert besonderes Augenmerk auf den Test und die Messung des Verhaltens auf jedem unterstützten System. Mit den enthaltenen Beispielen kann man sofort vergleichen und experimentieren. Niemand muss eine funktionierende Java I/O-Lösung umstellen. Bringt ein verwendetes Framework Java NIO mit, ist eine konsistente NIO-Implementierung über die Schnittstellen zu prüfen.

Referenzen

[1] Wikipedia *Non-blocking I/O (Java)* http://en.wikipedia.org/wiki/Non-blocking_I/O_%28Java%29 -

[2] I/O von JBoss AS und Play Framework *Netty project* http://netty.io

[3] Netty projekt *Netty 4.1.0.Beta1 released* Netty ab Version 4.1 unterstützt offiziell Android ab Version 4.0 Ice Cream Sandwich, http://netty.io/news/2014/07/04/4-1-0-Beta1.html

[4] DZone *Architekturbewertung, Kritik und Einsatzempfehlung* http://java.dzone.com/articles/java-nio-vs-io [5] http://search.maven.org/remotecontent?filepath=io/netty/netty-all/4.1.0.Beta1/netty-all-4.1.0.Beta1.jar

[5] http://search.maven.org/remotecontent?filepath=io/netty/netty-all/4.1.0.Beta1/netty-all-4.1.0.Beta1.jar

[6] http://search.maven.org/remotecontent?filepath=commons-io/commons-io/2.4/commons-io-2.4.jar

Kurzbiografie

Frank Ganske (frank.ganske@mathema.de) ist Senior Developer bei der MATHEMA Software GmbH in Erlangen. Er entwickelt seit 1991 betriebswirtschaftliche Individual- und Standard-Software. Besonderes Interesse hat er an Veränderungsprozessen und deren Absicherung. Das umfasst sowohl Bauprozesse, Testautomatisierung, statische Code-Analyse und die Feststellung von Systemeigenschaften und Abweichungen.

Vorbelegt

Default-Methoden und Co. in Java 8

Von Ilker Yümsek

Anhand von weniger komplexen Aufgabenstellungen kann man sich mit neuen Funktionalitäten und Änderungen in einer Programmiersprache besser auseinandersetzen. In diesem Artikel geht es um die Nutzung der mit Java 8 eingeführten Default-Methoden.

Als Basis der Aufgabenstellung dient die Übung in Abbildung 1 aus einem Arbeitsheft.

Bei der Aufgabe geht es darum, angefangen von einer Startzahl durch nacheinander folgende Divisionsoperationen eine Zielzahl herauszufinden. Um auf Basis einer Startzahl ähnliche Aufgaben programmatisch generieren zu lassen, wird die Übung wie in Abbildung 2 als Objekt abgebildet.

In diesem Modell wird die Startzahl als Basis für die Aufgabe vom Rest unabhängig betrachtet. Auf die Startzahl folgen weitere Bestandteile und wenn man sie genauer betrachtet, stellt man Folgendes fest: Es geht dabei um Gruppen, die in gleich bleibender Reihenfolge bestimmte Elemente der Aufgabe zusammenfassen und in beliebiger Anzahl vorkommen können. Diese Gruppen bekommen den Namen Rechenschritt. Ein Rechenschritt enthält dabei

Abbildung 1

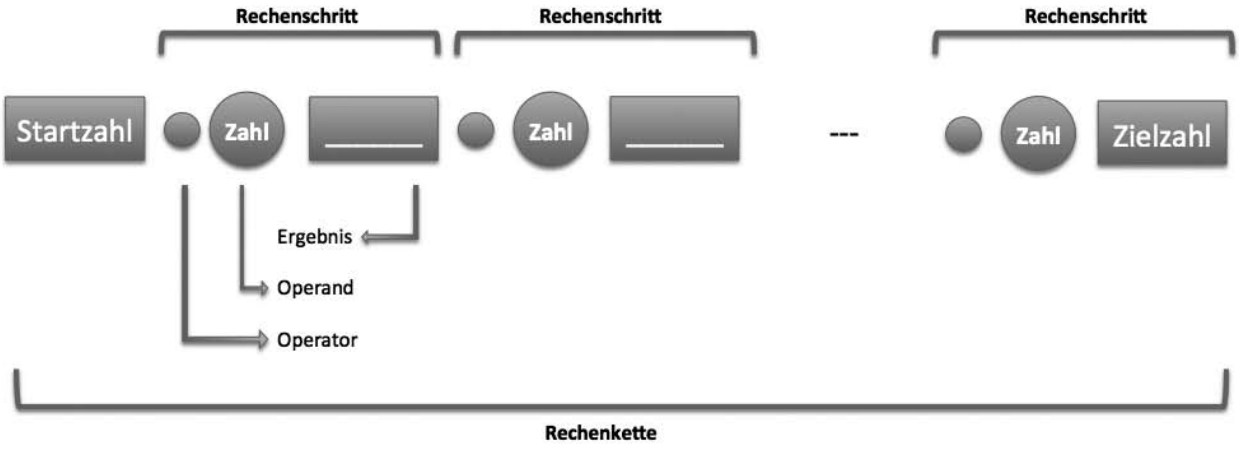


Abbildung 2

- einen Operanden und
- ein Operationsergebnis, das sich durch das Anwenden eines Operators (in der Aufgabe oben der Divisionsoperator) auf die vorhergehende Zahl (Startzahl oder Ergebnis des früheren Rechenschritts) und auf den Operanden ergibt.

Das Ergebnis des letzten Rechenschrittes ist damit die Zielzahl. Das Gesamtkonstrukt bekommt den Namen „Rechenkette".

Das Interface *IRechenschritt* umfasst dabei die Eigenschaften eines Rechenschritts:

```
public interface IRechenschritt{
   public int getOperand();
   public void setOperand(int operand);
   public int getErgebnis();
   public void setErgebnis(int ergebnis);
   public List<IRechenschritt> getFolgeschritte();
   public void addRechenschritt(IRechenschritt r);
...
}
```

Wie aus dem *Interface* erkennbar, enthält ein Rechenschritt eine Liste von möglichen Folgeschritten. Damit entsteht eine Baumstruktur, in der jeder eindeutige Pfad von der Wurzel bis zu einem Knoten in der untersten Ebene eine Rechenkette darstellt (siehe Abbildung 3).

Um die Erstellung der Folgeschritte mit möglichst wenig Programmieraufwand hinzubekommen, realisieren wir die Generierung und das Hinzufügen der generierten Rechenschritte in zwei *Default*-Methoden des Interfaces *IRechenschritt*.

Bei diesen Default-Methoden geht es darum, neue Rechenschritt-Instanzen entgegen zu nehmen, sie mit Daten zu füllen und in die Liste der Folgeschritte hinzuzufügen. In diesem Zusammenhang ist noch darauf hinzuweisen, dass die Berechnung des Ergebnisses – also die Rechenfunktionalität – in einem Rechenschritt bewusst außerhalb des Interfaces liegt.

Die erste Default-Methode (im Folgenden Kernmethode genannt) nimmt ein *IRechenschritt-Supplier,* den Operanden und die Rechenfunktionalität (*IntBinaryOperator*) entgegen, erstellt einen Folgeschritt und fügt diesen über die Methode *addRechenschritt* (IRechenschritt *r*) in die Liste der Folgeschritte ein:

```
default void processSchritt(
   Supplier<IRechenschritt> supplier,
   int operandRechts,
   IntBinaryOperator op
){
   IRechenschritt rs = supplier.get();
   rs.setOperand(operandRechts);
   int ergebnis = op.applyAsInt(getErgebnis(), operandRechts);
   rs.setErgebnis(ergebnis);
   addRechenschritt(rs);
}
```

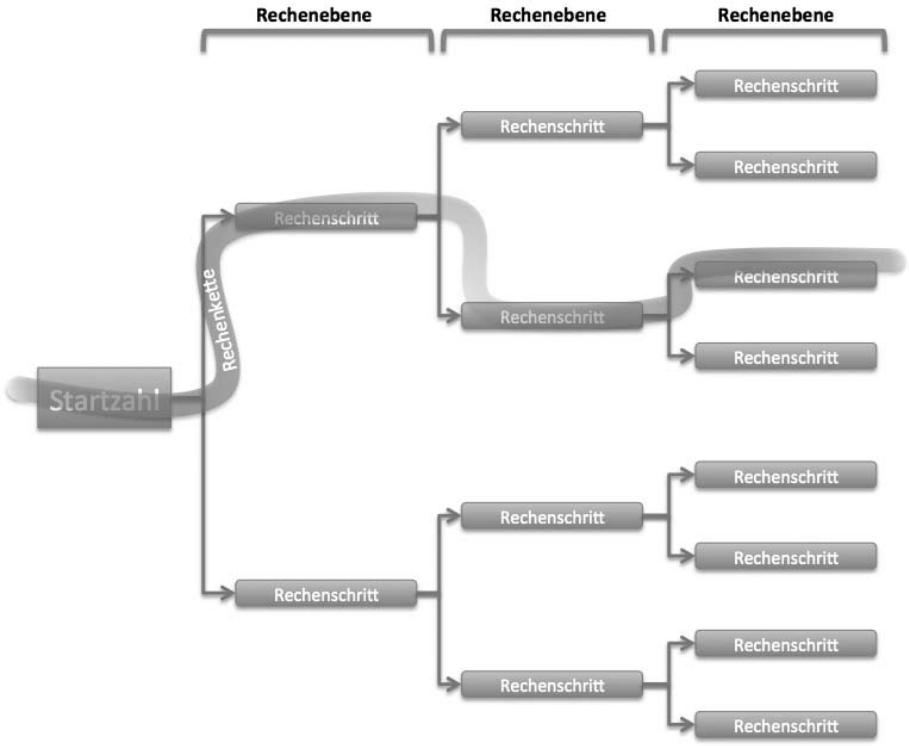


Abbildung 3

In der zweiten Methode werden neben den Parametern der Kernmethode die Nummer der aktuellen Ebene und ein *Supplier* der Operanden für eine bestimmte Ebene vom Aufrufer erwartet. Die Methode fragt die Liste der Operanden für die aktuelle Ebene ab und lässt für jeden Operanden ein Rechenschritt durch die Kernmethode erstellen:

```
default void processOperandList(
  int ebene,
  Function<Integer, List<Integer>> operandListSupplier,
  Supplier<IRechenschritt> supplier,
  IntBinaryOperator op
){
  List<Integer> operandList = operandListSupplier.apply(
    Integer.valueOf(ebene)
  );
  if(operandList != null){
    operandList.forEach(
      operand -> processSchritt(supplier, operand, op)
    );
    final int incEbene = ebene + 1;
    getFolgeschritte().forEach(
      fs -> fs.processOperandList(
        incEbene,
        operandListSupplier,
        supplier,
        op
      )
    );
  }
}
```

Durch den rekursiven Aufruf (für jeden neu erstellten Rechenschritt) am Ende der Methode wird die komplette Generierung der Rechenschritte bis zur untersten Ebene möglich.

Als grundlegende Klasse für die Generierung von Rechenaufgaben dient eine Klasse, die

- mehrere Rechenschritte (der ersten Rechenebene),
- eine Liste der Operanden pro Rechenebene,
- die Rechenfunktionalität und
- die Startzahl

enthält:

```
public class Rechenpfade {
  ArrayList<IRechenschritt> rechenschritte;
  ArrayList<List<Integer>> operandListen;
  int basiszahl;
  ...
  public List<Integer> getOperandList(int ebene) {
    List<Integer> operandList = null;
    if(operandListen.size() > ebene) {
      operandList = operandListen.get(ebene);
    }
    return operandList;
  }
  ...
  public int rechne(int operandLinks, int operandRechts) {
    ...
  }
  ...
}
```

Wenn die ersten Rechenschritte in dieser Klasse (auf Basis der Startzahl) erstellt und die Operanden für die (gewünschte Anzahl von) Rechenebenen definiert worden sind, können anhand dieser Informationen die weiteren Rechenebenen generiert werden:

```
public class Rechenpfade {
  ...
  public static void main(String[] args) {
    Rechenpfade rp = new Rechenpfade();
    ...
    rp.rechenschritte.forEach(
      s -> s.processOperandList(
        0,
        rp::getOperandList,
        (Supplier<IRechenschritt>) Rechenschritt::new,
        rp::rechne
      )
    );
    ...
  }
}
```

Damit haben wir es geschafft eine Grundfunktionalität für die Implementierungen des Interfaces bereitzustellen und diese – mithilfe der Default-Methoden – direkt im Interface selbst zu platzieren. Dies wäre vor Java 8 nur über eine abstrakte Implementierung des Interfaces möglich gewesen. Der Einsatz von abstrakten Klassen wird aber mit Java 8 nicht komplett obsolet, es gibt immer noch gute Gründe sie einzusetzen. Als Entscheidungshilfe kann die Gegenüberstellung der beiden Features (abstrakte Klassen und Interfaces) im Java-Tutorial über „Interfaces and Inheritance“[1] genommen werden.

Fazit

Zusammenfassend bringt der neue Ansatz mit den Default-Methoden und Methodenreferenzen folgende Vor- und Nachteile:

[1] http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html

(+) Da die Grundfunktionalität direkt im Interface realisiert wird und die Elemente im Interface aufeinander abgestimmt sein müssen, wird dadurch ein sauberes Design erreicht.

(+) Die Default-Methoden stellen ein gutes Lösungsmittel für solche Fälle dar, bei denen die Implementierungen von Interfaces sowohl bestimmte Grundfunktionalitäten nutzen als auch von einer anderen Klasse erben wollen.

(+) Die bereits existierenden Implementierungen müssen beim Hinzukommen von Default-Methoden keine Anpassungen machen.

(+) Durch die Möglichkeit, Methodenreferenzen zu übergeben, wird es bequemer, mit Operationen umzugehen, die an die Implementierungen delegiert werden. Die Operationen müssen nicht mehr (wie bei den abstrakten Methoden) zwangsläufig in der Implementierung des Interfaces selbst liegen.

(–) Über die Übergabe von Methodenreferenzen hinweg bleibt natürlich weiterhin die Möglichkeit, die zu delegierenden Operationen als Methode des Interfaces zu definieren. Dies birgt aber die Gefahr, dass Interfaces viele zusätzliche Methoden bekommen, die nur für die Delegierung bestimmter Operationen aus Default-Methoden heraus gedacht sind. Die Herausforderung beim Einsetzen von Default-Methoden wird darin liegen, das richtige Maß zwischen der Übergabe von Methodenreferenzen und dem Definieren von Interface-Methoden zu finden.

(–) Beim Einsetzen von Default-Methoden können keine (*non final*) Felder definiert oder genutzt werden.

(–) Die Default-Methoden sind für alle sichtbar. Beim Definieren von Default-Methoden muss stets folgende Frage gestellt werden: Geht es um eine allgemeine Funktionalität und soll diese von allen Nutzern des Interfaces aufgerufen werden können, oder nimmt man diesen Nebeneffekt in Kauf, weil die Vorgehensweise geschickter ist und favorisiert wird? In diesem Bezug könnte die Möglichkeit, den Zugriff auf die Default-Methode einzuschränken (z. B. über *private*), in Zukunft (in einer späteren Java-Version) mehr Gestaltungsspielraum geben.

Kurzbiografie

Ilker Yümsek arbeitet als Software-Architekt bei der MATHEMA Software GmbH und unterstützt Kunden bei der Konzeption und Realisierung von Software-Lösungen im Enterprise-Umfeld. Sein Schwerpunkt liegt auf Java-basierten Lösungen im Bereich der System-Integration und Enterprise-Anwendungen.

Des Programmierers kleine Vergnügen

Summa summarum

von Michael Wiedeking

Wie auch immer man seine Zahlen repräsentiert, irgendwann einmal kommt der Zeitpunkt, an dem man sie addieren oder subtrahieren möchte. Aber auch hier stellt sich die Frage, wie man dies am besten anstellt.

Um zwei beliebig lange Zahlen *x* und *y* zu addieren, kann man getrost den aus der Schule bekannten Algorithmus verwenden. Dabei bestehen die Ziffern idealerweise aus Maschinenwörtern, die man unter Beachtung etwaiger Überträge addiert. Der Einfachheit halber sollen die Zahlen hier nur positiv sein und als *Array* repräsentiert werden, wobei die niederwertigste Ziffer den Index 0 haben soll. Um nun *x* und *y* addieren zu können, die spaßeshalber auch noch gleich lang sein sollen und *n* Ziffern haben, benötigt man eine geeignete Funktion, die nicht nur die Summe zweier solcher Ziffern sondern auch deren Übertrag liefert:

(word, word) addWithCarry(word *x*, word *y*, word *c*)

Mit Hilfe einer solchen Funktion, die ein Wertepaar mit der Summe und dem Überlauf liefert, lässt sich die Summe für die Ziffern vom Typ *word* relativ leicht berechnen. Hierbei ist noch zu erwähnen, dass der Übertrag nur 0 oder 1 sein kann.

```
word[] add(word[] x, word[] y) {
  assert x.length == y.length;
  int n = x.length;
  word[] z = new word[n];
  word carry = 0;
  for (int i = 0; i < n; i++) {
    (sum, carry) = addWithCarry(x[i], y[i], carry);
    z[i] = sum;
  }
  if (carry != 0) {
    z = makeCopy(z, n + 1);
    z[n] = carry;
  }
  return z;
}
```

Wie man sieht, ist genau dann ein zusätzliches Wort nötig, wenn die letzte Teilsumme zu einem Übertrag führt. Bei der Repräsentation muss man also entscheiden, ob man ein überflüssiges Wort in Kauf nehmen möchte oder nicht. Möchte man etwa, dass die Repräsentation immer eindeutig ist, dann ist das nicht möglich, denn sonst könnte etwa die Zahl 1 als 1-Wort-Variante ⟨1⟩ oder 2-Wort-Variante ⟨0, 1⟩ repräsentiert werden. Letzteres führt dann aber zu Sonderfällen etwa bei der Bestimmung der Größe des Resultats.

Das Erzeugen eines neuen *Arrays* mit *makeCopy*, das um ein Wort größer ist, braucht selbst bei großen Zahlen relativ wenig Zeit. So benötigt eine 1000-stellige Zahl nur knapp einhundert 32-Bit-Wörter und ist dementsprechend schnell kopiert. Ein vergleichbares Problem tritt übrigens fast immer bei einer Subtraktion auf, wenn die beteiligten Operanden ähnlich groß sind. Zieht man etwa im Dezimalsystem 1000 von 1001 ab, so ergibt sich daraus 1001 − 1000 = 1. Damit ist das Ergebnis um drei Stellen kleiner als es ein reiner Längenvergleich der Zahlen vermuten lässt.

Natürlich kann man nicht davon ausgehen, dass die Zahlen immer gleich lang sind. Im Zusammenhang mit der Addition hat dies auch den positiven Nebeneffekt, dass das mit *n* dimensionierte Ergebnis in den meisten Fällen ausreichend ist. Deswegen kann die Schleife auch oft nach weniger als *n* Schritten beendet werden. Um es wieder einfach zu machen, ist es recht günstig, wenn *x* immer länger oder genau so lang wie *y* ist, was sich etwa durch folgendes Umdrehen erreichen lässt (wenn es denn die benutzten Mehrfachzuweisungen gibt).

```
int n = x.length;
int m = y.length;
```

```
if (m > n) {
  (x, y) = (y, x);
  (n, m) = (m, n);
}
```

Dann kann man die Addition in zwei Teile zerlegen: Der erste Teil berechnet die Summe der ersten *m* Elemente und der zweite Teil addiert dann nur noch die Überträge, so weit es nötig ist. Da die Addition mit Überlauf auf jeden Fall teurer ist als eine Kopie, lohnt es sich, sobald es keine Überträge mehr gibt, die verbleibenden Werte einfach zu kopieren.

```
word[] z = new word[n];
word carry = 0
for (int i = 0; i < m; i++) {
  (sum, carry) = addWithCarry(x[i], y[i], carry);
  z[i] = sum;
}
for (int i = m; i < n; i++) {
  (sum, carry) = addWithCarry(x[i], 0, carry);
  z[i] = sum;
  if (carry == 0) {
    for (i = i + 1; i < n; i++) {
      z[i] = x[i];
    }
    break;
  }
}
if (carry != 0) {
  z = makeCopy(z, n + 1);
  z[n] = carry;
}
return z;
```

Nachdem es nun grundsätzlich möglich ist, die Addition durchzuführen, bleibt nur noch die Frage, wie man denn *addWithCarry* implementiert. Das ist eigentlich schon in vergangenen Vergnügen (die – wie es der Zufall will – auch noch beide den gleichen Titel haben) im Zusammenhang mit doppelt breiten Wörtern [1] und Überlauferkennung [2] beantwortet worden. Allerdings gab es dabei einen kleinen „Schönheitsfehler“: Die Additionen hatten es nicht auch noch mit einem Übertrag zu tun.

Wie oben schon erwähnt ist der Überlauf bei zwei Wörtern gleicher Breite entweder 0 oder 1. Bei obiger Berechnung haben wir es aber mit drei Summanden zu tun. Auch wenn der dritte Summand – der Überlauf – maximal 1 sein kann, ist es nicht selbstverständlich, dass der Übertrag von $x + y + 1$ auch höchstens 1 ist. Es sei aber an dieser Stelle versichert, dass dies tatsächlich der Fall ist.

Das kann man sich eigentlich auch ganz leicht vorstellen. Im Dezimalsystem gibt es doch die Ziffern von 0 bis 9. Das damit zu erzielende Maximum für eine Stelle ist damit 9 + 9 = 18. Und diese Summe bietet noch Platz für einen Überlauf, der nicht größer ist als 1, um nicht selber überzulaufen. Jetzt stellt sich natürlich die Frage, ob das auch für Zahlensysteme gilt, die eine kleinere Basis haben. Im Dreiersystem etwa ist das Maximum $2 + 2 = 11_3 = 4_{10}$. So bietet auch das noch Platz für einen Überlauf von 1. Und das gilt nun natürlich auch noch für das Binärsystem, denn auch hier bietet das Maximum $1 + 1 = 10_2$ den nötigen Platz.

```
(word, word)  addWithCarry(word x, word y, word carry) {
  word sum = x + y + carry;
  carry = ((x & y) | ((x | y) & ~sum)) >>> (word_size – 1);
  return (sum, carry);
}
```

Unabhängig davon sei hier noch eine Variante erwähnt, die aus verschiedenen Gründen mit den anderen, bedingungsfreien Varianten konkurrieren kann. Ersetzt man etwa den Typ *word* durch *int*, so kann man auf den meisten Plattformen auf ein *long* zurückgreifen, um die Berechnungen überlauffrei durchführen zu können.

```
(int, int) addWithCarry(int x, int y, int carry) {
  long sum =
    (x & 0xFFFFFFFFL) + (y & 0xFFFFFFFFL) + carry;
  int result = (int) sum;
  int carry = (int) (sum >>> 32);
  return (result, carry);
}
```

Das ist sehr schnell, insbesondere auch deswegen, weil ein optimierender (Just-in-Time-)*Compiler* erkennen kann, dass man das *carry* einfach über die höherwertige Hälfte des Worts ermitteln kann, so dass gar kein *Shift* durchgeführt werden muss. Hier sei noch erwähnt, dass nur das *L* am Ende der Konstanten sicherstellt, dass das höchstwertigste Bit des *int* nicht als Vorzeichen interpretiert wird und so aus dem *long*-Äquivalent fälschlicherweise eine negative Zahl gemacht wird.

Ob sich die eine oder andere Variante besser eignet, muss genau analysiert werden. Auf einer 64-Bit-Maschine benötigt die allgemeine *word*-Variante 8 Instruktionen für jede Ziffer, auch bei einem 64-Bit-Wort. Die Doppelwort-Variante benötigt optimiert nur 4 Operationen, muss dafür aber doppelt so oft aufgerufen werden, da nur die halb so großen 32-Bit-Wörter verarbeitet werden können. Demgegenüber muss die erste Variante doppelt so viele Daten aus dem Speicher laden, dafür be-

nötigt die Schleife nur halb so viele Abbruchtests. Und so weiter und so fort.

Hier und da können noch kleine Optimierungen gemacht werden, was sich bei der Implementierung einer Standardbibliothek immer lohnt. Beispielsweise kann das *addWithCarry*(*x*, 0, *carry*) stark vereinfacht werden. Setzt man nämlich *y* = 0, dann verbleibt nur noch

```
(word, word) addWithCarry(word x, word carry) {
  word sum = x + carry;
  carry = (x & ~sum) >>> word_size;
  return (sum, carry);
}
```

Anstatt des *breaks* kann direkt *return* aufgerufen werden. In diesem Fall kann auf den abschließenden Test *carry != 0* verzichtet und direkt die größere Kopie angelegt werden, da die Schleife nur noch dann ordentlich verlassen wird, wenn es einen abschließenden Überlauf gegeben hat.

Alles in allem ist es schon schade, dass einem in Hochsprachen in der Regel die Addition mit Überlauf verwehrt ist. Aber in diesem Fall war es ja ganz nützlich, denn sonst würde es diese Kolumne ja nicht geben. Und trotzdem bleiben immer noch einige Fragen offen. Aber die werden wohl bis zum nächsten Mal warten müssen.

Referenzen

[1] Wiedeking, Michael *Des Programmierers kleine Vergnügen – Überläufer* KaffeeKlatsch, Jahrgang 4, Nr. 1, Seite 33f, Bookware, Januar 2011 http://www.bookware.de/kaffeeklatsch/archiv/KaffeeKlatsch-2011-01.pdf

[2] Wiedeking, Michael *Des Programmierers kleine Vergnügen – Überläufer KaffeeKlatsch*, Jahrgang 5, Nr. 2, Seite 18ff, Bookware, Februar 2012 http://www.bookware.de/kaffeeklatsch/archiv/KaffeeKlatsch-2012-02.pdf

Kurzbiographie

Michael Wiedeking (michael.wiedeking@mathema.de) ist Gründer und Geschäftsführer der MATHEMA Software GmbH, die sich von Anfang an mit Objekttechnologien und dem professionellen Einsatz von Java einen Namen gemacht hat. Er ist Java-Programmierer der ersten Stunde, „sammelt" Programmiersprachen und beschäftigt sich mit deren Design und Implementierung.

Kaffeesatz

Ges(ch)ichtsbuch

von Michael Wiedeking

Neulich war Klassentreffen. Vor dreißig Jahren haben wir unser Abitur gemacht. Das ist so lange her, dass bei dem Treffen der eine oder andere da war, der ganz sicher nicht bei mir auf der Schule war. Aber es ist ja auch schön, wenn man mal ganz neue Leute kennenlernen kann.

Natürlich will man nun auch irgendwie in Kontakt bleiben. Mit den Freunden von damals steht man natürlich immer noch in Kontakt. Aber einige der Mitschüler sind so interessant, dass man sie vielleicht mal besuchen möchte, wenn sich eine passende (geographische) Gelegenheit ergibt. Oder man möchte wissen, was sie so tun und treiben, oder wie es ihnen vielleicht gerade geht. Und da bietet sich doch irgendwie Facebook an.

Also habe ich Facebook neu für mich entdeckt. „Neu" deswegen, weil ich den Account eigentlich schon seit Jahren besitze, ihn aber nicht wirklich genutzt habe. Und das hat einen ganz einfachen Grund: Ich komme mit der Oberfläche einfach nicht zurecht. Aber das mag dem Umstand geschuldet sein, dass ich nicht weiß, wofür das Werkzeug gut sein soll, weil ich auch nicht weiß, was ich mitteilen könnte.

Das erste was mich verwunderte, als ich mich damals angemeldet hatte, war der Umstand, dass ich nur wenige Minuten später die erste Freundschaftsanfrage bekommen habe. Das konnte nur mit meiner E-Mail-Adresse zusammenhängen, und das wiederum konnte nur bedeuten, dass die potenziellen Freunde ihr Adressbuch an Facebook weitergegeben haben (und die sich das gemerkt haben). Auf meinem iPhone ist die Facebook-App zwar noch nicht installiert, aber legt man dort den Facebook-Schalter um, wird man gleich gefragt, ob man zum Freundefinden nicht sein Adressbuch freigeben möchte.

Selbst wenn ich mein Adressbuch nicht preisgebe, werde ich recht einfach über die indirekten Verknüpfungen gefunden. So habe ich über wenige Verbindungen eine Klassenkameradin vorgeschlagen bekommen, die vor Jahrzehnten in die Vereinigten Staaten ausgewandert ist und dort unter einem Namen auftritt, der mit dem aus der Schule praktisch nichts zu tun hat. Natürlich gibt es noch andere Indizien, über die sich jemand ausfindig machen lässt, und ich bin schon sehr erstaunt darüber, wie bereitwillig Angaben über den Wohnort, den Arbeitgeber, die Schule und das Geburtsdatum gemacht werden.

Alle Sorgfalt und Paranoia nutzten aber nichts, wenn das die anderen Beteiligten nicht so sehen. So bin ich nach dem Klassentreffen in eine Gruppe eingetragen worden, über die sich alle Schüler des Jahrgangs verabreden können sollen. Ich bin mir da nicht sicher, wer das alles mitbekommen hat. Dann bin ich wohl schon vor Jahren auf irgendeinem Bild markiert worden, das jemand gemacht hat. Auf dem Oktoberfest. Als Entschuldigung lässt sich nur sagen, dass ich von einem wichtigen Münchener Partner dorthin eingeladen worden bin – denn wer braucht schon das Oktoberfest, das es erst seit 1810 gibt, wenn er seit 1755 die Bergkirchweih vor der Tür hat. Also dass man mich bloß nicht für einen Oktoberfestgänger hält; aber auf den Berg gehe ich auch nicht.

Das Geburtsdatum lässt sich nicht wirklich geheim halten. Mit diesem Freibrief für telefonische Transaktionen sollte man meiner Meinung nach zwar vorsichtig umgehen, aber was nutzt meine Vorsicht, wenn mir auch nur einer meiner Freunde gedankenlos dort zum Geburtstag gratuliert; möglichst noch mit der Angabe meines genauen Alters, was insbesondere bei runden Geburtstagen sehr verlockend ist. Wobei ich da noch

eher meiner Kreditkartengesellschaft einen Vorwurf machen muss, die sich neben der Kreditkartennummer mit meinem Geburtstag (Handelsregister) und meiner Adresse (Telefonbuch) zufrieden gegeben hat.

Dann verstehe ich nicht, warum ich Nachrichten über ein weiteres Medium bekommen muss. Mein E-Mail-Verhalten ist nicht beanstandungsfrei, aber die meisten E-Mails werden wenigstens gelesen. Das ist nicht der Fall, wenn man diesen sozialen Portalen nicht regelmäßig einen Besuch abstattet, und dann auch noch die Feinheiten nicht beachtet. So habe ich zufällig eine Nachricht entdeckt, die ich vor 18 Monaten geschickt bekommen habe. Die konnte ich dann auch irgendwo hinschieben. Aber ich weiß nicht, was ich damit verändert habe, und ob die Nachricht aufgehoben oder gelöscht wurde.

Das wäre ja weiter nicht schlimm, wenn nicht die Dauernutzer insgeheim davon ausgehen würden, dass die Nachrichten sofort gelesen werden. Na, immerhin sind die meisten Mitteilungen unidirektional, so dass sie keiner Reaktion bedürfen. Man kann so etwas wohl „liken" oder „teilen", aber das übersteigt schon meinen Horizont. Also auf die Knöpfe drücken könnte ich schon noch, aber ich habe nicht verstanden, was für Auswirkungen das hat – außer dass Facebook weiß, was mir gefällt.

Am schlimmsten ist es aber, das Facebook nichts zu vergessen scheint. So habe ich schon Verweise auf (echte) Freunde gefunden, die sich bei Facebook abgemeldet haben; aber was sie bis dahin gemacht haben bleibt unverändert enthalten. Es handelt sich also um ein jederzeit verfügbares Gedächtnis, das bei falschen Einstellungen jedermann möglicherweise intime Einblicke gewährt, durchaus auch über den Tod hinaus.

Allerdings muss ich gestehen, es gleichermaßen beeindruckend wie beängstigend zu finden, dass just in diesem Augenblick 164.194.350 Menschen Facebook gefällt. Und dass es, bis Sie das hier gelesen haben werden, noch deutlich mehr geworden sind.

Kurzbiographie

Michael Wiedeking (michael.wiedeking@mathema.de) ist Gründer und Geschäftsführer der MATHEMA Software GmbH, die sich von Anfang an mit Objekttechnologien und dem professionellen Einsatz von Java einen Namen gemacht hat. Er ist Java-Programmierer der ersten Stunde, „sammelt" Programmiersprachen und beschäftigt sich mit deren Design und Implementierung.

User Groups

Fehlt eine User Group? Sind Kontaktdaten falsch? Dann geben Sie uns doch bitte Bescheid.

Bookware, Henkestraße 91, 91052 Erlangen
Telefon: 0 91 31 / 89 03-0, Telefax: 0 91 31 / 89 03-55
E-Mail: redaktion@bookware.de

Java User Groups

Deutschland

JUG Berlin Brandenburg
http://www.jug-bb.de
Kontakt: Herr Ralph Bergmann (orga@jug-bb.de)

Java UserGroup Bremen
http://www.jugbremen.de
Kontakt: Rabea Gransberger (rgransberger@gmx.de)

JUG DA
Java User Group Darmstadt
http://www.jug-da.de
Kontakt: jug-da-orga@googlegroups.com

Java User Group Saxony
Java User Group Dresden
http://www.jugsaxony.de
Kontakt: Herr Falk Hartmann
(falk.hartmann@jugsaxony.org)

rheinjug e.V.
Java User Group Düsseldorf
Heinrich-Heine-Universität Düsseldorf
http://www.rheinjug.de
Kontakt: Herr Heiko Sippel (info@rheinjug.de)

ruhrjug
Java User Group Essen
Glaspavillon Uni-Campus
http://www.ruhrjug.de
Kontakt: Herr Heiko Sippel (heiko.sippel@ruhrjug.de)

JUGF
Java User Group Frankfurt
http://www.jugf.de
Kontakt: Herr Alexander Culum
(alexander.culum@web.de)

JUG Deutschland e.V.
Java User Group Deutschland e.V.
c/o Stefan Koospal
http://www.java.de (office@java.de)

JUG Hamburg
Java User Group Hamburg
http://www.jughh.org

JUG Karlsruhe
Java User Group Karlsruhe
http://jug-karlsruhe.de
(jugkarlsruhe@gmail.com)

JUGC
Java User Group Köln
http://www.jugcologne.org
Kontakt: Herr Michael Hüttermann
(michael@huettermann.net)

jugm
Java User Group München
http://www.jugm.de
Kontakt: Herr Andreas Haug (ah@jugm.de)

JUG Münster
Java User Group für Münster und das Münsterland
http://www.jug-muenster.de
Kontakt: Herr Thomas Kruse (tkjugi@sforce.org)

JUG MeNue
Java User Group der Metropolregion Nürnberg
c/o MATHEMA Software GmbH
Henkestraße 91, 91052 Erlangen
http://www.jug-n.de
Kontakt: Frau Natalia Wilhelm
(info@jug-n.de)

JUG Ostfalen
Java User Group Ostfalen
(Braunschweig, Wolfsburg, Hannover)
http://www.jug-ostfalen.de
Kontakt: Uwe Sauerbrei (info@jug-ostfalen.de)

JUGS e.V.
Java User Group Stuttgart e.V.
c/o Dr. Michael Paus
http://www.jugs.org
Kontakt: Herr Dr. Micheal Paus (mp@jugs.org)
Herr Hagen Stanek (hs@jugs.org)
Rainer Anglett (ra@jugs.org)

Schweiz

JUGS
Java User Group Switzerland
http://www.jugs.ch (info@jugs.ch)

.NET User Groups

Deutschland

.NET User Group Bonn
.NET User Group "Bonn-to-Code.Net"
http://www.bonn-to-code.net (mail@bonn-to-code.net)
Kontakt: Herr Roland Weigelt

.NET User Group Dortmund (Do.NET)
c/o BROCKHAUS AG
http://do-dotnet.de
Kontakt: Paul Mizel (pmizel@do-dotnet.de)

Die Dodnedder
.NET User Group Franken
http://www.dodnedder.de
Kontakt: Herr Udo Neßhöver, Frau Ulrike Stirnweiß
(info@dodnedder.de)

.NET UserGroup Frankfurt
http://www.dotnet-usergroup.de

.NET User Group Hannover
http://www.dnug-hannover.de
Kontakt: (dnug@indisoftware.de)

INdotNET
Ingolstädter .NET Developers Group
http://www.indot.net
Kontakt: Herr Gregor Biswanger
(gregor.biswanger@web-enliven.de)

DNUG-Köln
DotNetUserGroup Köln
http://www.dnug-koeln.de
Kontakt: Herr Albert Weinert (info@der-albert.com)

.NET User Group Leipzig
http://www.dotnet-leipzig.de
Kontakt: Herr Alexander Groß (agross@dotnet-leipzig.de)
Herr Torsten Weber (tweber@dotnet-leipzig.de)

.NET Developers Group München
http://www.munichdot.net
Kontakt: Hardy Erlinger (hardy_erlinger@hotmail.com)

.NET User Group Oldenburg
c/o Hilmar Bunjes und Yvette Teiken
http://www.dotnet-oldenburg.de
Kontakt: Herr Hilmar Bunjes
(hilmar.bunjes@dotnet-oldenburg.de)
Frau Yvette Teiken (yvette.teiken@dotnet-oldenburg.de)

.NET Developers Group Stuttgart
Tieto Deutschland GmbH
http://www.devgroup-stuttgart.de
(GroupLeader@devgroup-stuttgart.de)
Kontakt: Herr Michael Niethammer

.NET Developer-Group Ulm
c/o artiso solutions GmbH
http://www.dotnet-ulm.de
Kontakt: Herr Thomas Schissler (tschissler@artiso.com)

Österreich

.NET User Group Austria
c/o Global Knowledge Network GmbH,
http://usergroups.at/blogs/dotnetusergroupaustria/default.aspx
Kontakt: Herr Christian Nagel (ug@christiannagel.com)

Software Craftsmanship Communities

Deutschland, Schweiz, Österreich

Softwerkskammer – Mehrere regionale Gruppen und Themengruppen unter einem Dach
http://www.softwerkskammer.org
Kontakt: Nicole Rauch (nicole.m@gmx.de)

MATHEMA

meet the experts of enterprise infrastructure

Entwicklung mobiler Anwendungen mit iOS
20. – 22. April 2015, 5. – 7. Oktober 2015
1.250,– € (zzgl. 19 % MwSt.)

HTML5, CSS3 und JavaScript
9. – 12. März 2015, 21. – 24. September 2015
1.650,– € (zzgl. 19 % MwSt.)

Anwendungsentwicklung mit der Java Enterprise Edition
6. – 11. Oktober 2014, 22. – 26. Juni 2015
2.150,– € (zzgl. 19 % MwSt.)

Fortgeschrittenes Programmieren mit Java
Ausgewählte Pakete der Java Standard Edition
13. – 15. Oktober 2014, 4. – 6. Mai 2015
1.350,– € (zzgl. 19 % MwSt.)

Weiterführende Programmierung unter C#
Umstieg auf C# 4.5/5 und Einführung in fortgeschrittene Konzepte
24. – 27. November 2014, 11. – 13. Mai 2015
1.350,– € (zzgl. 19 % MwSt.)

AngularJS
18. – 19. Mai 2015, 7. – 8. Dezember 2015
950,– € (zzgl. 19% MwSt.)

Lesen bildet. **Training macht fit.**

MATHEMA Software GmbH | Telefon: 09131 / 89 03-0 | Internet: www.mathema.de
Henkestraße 91, 91052 Erlangen | Telefax: 09131 / 89 03-55 | E-Mail: info@mathema.de

MATHEMA

„Die Herausforderung, jeden Tag etwas Neues zu lernen, habe ich gesucht und bei MATHEMA gefunden."

Tim Bourguignon, Senior Consultant

Wir sind ein Consulting-Unternehmen mit Schwerpunkt in der Entwicklung unternehmenskritischer, verteilter Systeme und Umsetzung von Service-orientierten Architekturen und Applikationen von Frontend bis Backend. Darüber hinaus ist uns der Wissenstransfer ein großes Anliegen: Wir verfügen über einen eigenen Trainingsbereich und unsere Consultants sind regelmäßig als Autoren in der Fachpresse sowie als Speaker auf zahlreichen Fachkonferenzen präsent.

join the experts of enterprise infrastructure

MATHEMA Software GmbH | Telefon: 09131 / 89 03-0 | Internet: www.mathema.de
Henkestraße 91, 91052 Erlangen | Telefax: 09131 / 89 03-55 | E-Mail: info@mathema.de | Ansprechpartnerin: Britta Kolb

Das Allerletzte

„Die zwei Folgefehler dürfen nicht passieren,
weil der eigentliche Fehler nicht passieren darf.“

Dies ist kein Scherz!
Diese Aussage wurde tatsächlich in der freien
Wildbahn angetroffen.

Ist Ihnen auch schon einmal ein Exemplar dieser
Gattung über den Weg gelaufen?
Dann scheuen Sie sich bitte nicht, uns das mitzuteilen.

Der nächste KaffeeKlatsch erscheint im Oktober.

Herbstcampus

Wissenstransfer par excellence

31. August – 3. September 2015
in Nürnberg