UNIVERSITÄT DUISBURG ESSEN

NES Networked Embedded Systems

Miniprojekt 4 – Collections, Generics, Exceptions, IO und GUIs

In diesem Miniprojekt erstellen Sie einen graphischen Editor für Konfigurationsdateien, die eine Liste von Schlüssel-Wert-Paaren (engl. Key-Value-Pair) enthalten soll. Der Aufbau der Konfigurationsdateien entspricht dabei folgendem Schema:

<Schlüssel> = <Wert><Zeilenumbruch>
<Schlüssel> = <Wert><Zeilenumbruch>
…

Namen und Werte dürfen mehrfach vorkommen und die Reihenfolge der Schlüssel-Wert-Paare soll keine Rolle spielen.

Bei der Entwicklung des Editors üben Sie den Umgang mit den einfachen Collection Klassen aus dem java.util Paket, um komplexere Datentypen durch Komposition zu erzeugen. Darüber hinaus üben Sie die Erstellung generischer Klassen und nutzen Exceptions um Fehlerfälle anzuzeigen und zu behandeln. Schließlich üben Sie noch den Umgang mit den IO Klassen aus dem java.io Paket und Sie lernen ein einfaches Beispiel für eine Swing-basierte graphische Oberfläche kennen.

Aufgabe 1 – MultiHashMap

Die Schnittstelle **java.util.Map** und die zugehörige **java.util.HashMap** definieren und implementieren eine Datenstruktur, die eine Menge von Schlüssel-Wert-Paaren speichern kann. Die **Map** bildet dabei jeden Schlüssel auf genau einen Wert ab. D.h. es ist nicht möglich mehrere Werte für einen Schlüssel zu hinterlegen. Da unsere Konfigurationsdateien jedoch den selben Schlüssel mehrfach enthalten dürfen, brauchen wir eine andere Datenstruktur, welche die Funktionalität der **HashMap** entsprechend erweitert.

Die Schnittstelle **MultiMap** definiert hierfür eine Reihe von Methoden, die den Methoden von **java.util.Map** ähneln. Sie sind jedoch darauf ausgelegt, dass einem Schlüssel mehrere Werte zugeordnet sein können. D.h. ähnlich wie die **put**(key,value)-Methode einer Map, gibt es eine **addValue**(key,value)-Methode, die einen Wert für einen Schlüssel einfügt. Ebenso gibt es eine Methode **get**- und **remove**-Methode. Die **get**-Methode liefert jedoch nicht nur einen Wert, sondern eine Liste von Werten zurück und die **remove**-Methode entfernt nicht den Schlüssel, sondern nur einen bestimmten Wert, der diesem zugeordnet ist. Analog zur Schnittstelle **Map** verwendet auch die Schnittstelle **MultiMap** zwei Typparameter (**K** und **V**), um beliebige Typen von Schlüsseln (K für Key) und Werten (V für Value) typsicher zu verwalten.

Datenstruktur

Implementieren Sie die Schnittstelle **MultiMap** in der Klasse **MultiHashMap**. Genau wie **MultiMap** soll Ihre Implementierung dabei generisch sein. D.h. die Typparameter K und V sollen von Ihrer Implementierung nicht festgelegt werden. Nutzen Sie dabei als interne Datenstruktur Ihrer

Implementierung eine **HashMap**, die Sie mit einer **ArrayList** kombinieren, um so mehrere Werte für jeden Schlüssel speichern zu können. Konkret bedeutet das, dass Ihre interne **HashMap** einen Schlüssel **K** auf eine **List<V>** abbilden soll. Die Programmlogik sieht dann so aus:

1. Wird ein Wert für einen neuen Schlüssel eingefügt, muss zunächst eine Liste erzeugt werden, die den Wert enthält. Diese Liste wird dann für den Schlüssel in der **HashMap** eingetragen.
2. Wird ein Wert für einen bestehenden Schlüssel eingefügt, muss dieser nur ans Ende der Liste, die zu dem Schlüssel gehört, eingefügt werden.
3. Wird ein Wert für einen bestehenden Schlüssel entfernt, muss dieser aus der zugehörigen Liste entfernt werden. Ist die Liste nach dem Entfernen des Werts leer, soll auch die Zuordnung zwischen Schlüssel und Liste entfernt werden.

Punkt 1 und 2 ermöglichen somit die Zuordnung mehrerer Werte zu einem Schlüssel. Punkt 3 stellt sicher, dass zu jedem Schlüssel in der **HashMap** mindestens ein Wert in der zugehörigen Liste enthalten ist.

Exceptions

Im Gegensatz zu einer **HashMap** soll Ihre Implementierung der **MultiHashMap** keine null-Schlüssel und null-Werte zulassen. Versucht ein Programmierer einen null-Schlüssel oder null-Wert zu verwenden, soll Ihre Implementierung eine NullPointerException mit einer aussagekräftigen Fehlermeldung ausgeben (z.B. „Key must not be null.“ oder „Value must not be null.“).

Darüber hinaus sollen alle Operationen, die mit Hilfe eines Schlüssels Werte abrufen oder löschen eine KeyNotFoundException liefern, sofern der entsprechende Schlüssel nicht in der internen HashMap enthalten ist. Implementieren Sie hierfür zunächst die Klasse KeyNotFoundException, indem Sie von der Klasse Exception ableiten und einen Konstruktur definieren, der einen String als Parameter erhält und an die Basisklasse weiterleitet. Integrieren Sie danach die entsprechenden Prüfungen in Ihre Implementierung der MultiHashMap. Die genaue Liste der betroffenen Methoden entnehmen Sie der Schnittstelle MultiMap, die bereits Methodendeklarationen mit KeyNotFoundExceptions enthält.

Test

Testen Sie Ihre MultiHashMap mit Hilfe von Strings. Erzeugen Sie sich hierfür eine main-Methode, die eine MultiHashMap<String,String> erzeugt. Fügen Sie danach Werte ein und lassen Sie sich den Zustand ausgeben. Entfernen Sie danach die Werte und lassen Sie sich den Zustand wieder ausgeben. Um den Zustand der MultiHashMap einfacher ausgeben zu können, können Sie die toString-Methode überschreiben und den Aufruf an die interne HashMap weiterleiten.

Aufgabe 2 – IO

Mit der **MultiHashMap** haben wir jetzt die Möglichkeit den Inhalt einer Konfigurationsdatei programmintern darzustellen. Uns fehlt jedoch noch die Möglichkeit, den Inhalt der **MultiHashMap** aus einer Datei zu laden und wieder in eine Datei zu speichern. Die Schnittstelle MultiMapIO definiert hierfür die Methode **void write(BufferedWriter, MultiMap)** zum Speichern einer MultiMap sowie die Methode **MultiMap read(BufferedReader)** zum Laden einer MultiMap. Wie die Schnittstelle **MultiMap** ist auch die Schnittstelle **MulitMapIO** mit zwei Typparametern versehen um beliebige Typen für Schlüssel und Werte typsicher zu behandeln.

Implementieren Sie die Klasse StringMapIO so, dass diese die Schnittstelle MultiMapIO für MultiMaps implementiert, deren Schlüssel und Werte vom Typ String sind.

Write

In der write-Methode soll dabei der Inhalt der MultiMap in den BufferedWriter geschrieben werden. Für jedes Schlüssel-Wert-Paar der MultiMap soll dabei eine Zeile nach dem folgenden Schema geschrieben werden:

<Schlüssel> = <Wert><Zeilenumbruch>

D.h. also die Zeichenkette des Schlüssels, dann ein Leerzeichen, dann ein Gleichzeichen, dann ein Leerzeichen, dann die Zeichenkette des Wertes gefolgt von einem Zeilenumbruch („\n“ unter Linux oder „\r\n“ unter Windows).

Hinweis: Aufgrund der Schnittstellendefinition ist es erforderlich, dass Sie beim Iterieren über die Schlüssel und Wertpaare in der MultiMap eine KeyNotFoundException beim Aufruf der Methode **getValues(Key)** abfangen und behandeln. Bei einer korrekten Implementierung der MultiMap kann dieser Fehler während der Ausführung niemals auftreten, da die Methode **getKeys()** nur existierende Schlüssel zurückliefern soll. Der Compiler kann dies aber nicht feststellen. Um den Fehler zu behandeln, können Sie ihn mittels eines try-catch-blocks in eine IOException umwandeln. Ein gebräuchlicher Weg dabei ist die Umwandlung durch Verkettung der Exceptions (engl. exception chaining). D.h. Sie geben bei der Erstellung der neuen IOException, die KeyNotFoundException als Auslöser an. Das sieht dann z.B. so aus:

```
try {
        …. // code that may throw a KeyNotFoundException
} catch (KeyNotFoundException knf) {
        throw new IOEXception(“The MultiMap implementation is erroneous.”, knf)
}
```

Read

Die read-Methode soll zunächst eine neue MultiMap erzeugen und dann den gesamten Inhalt des BufferedReaders zeilenweise lesen. Der Inhalt jeder Zeile soll dabei nach dem String " = " (also Leerzeichen, Gleichzeichen, Leerzeichen) durchsucht werden.

1. Wird er gefunden, soll die Zeile in den Schlüssel (alle Zeichen vor der Zeichenkette " = ") und den Wert (alle Zeichen nach der Zeichenkette " = ") aufgeteilt werden. Das so gelesene Schlüssel-Wert-Paar soll dann in die MultiMap eingefügt werden.
2. Wird der String " = " nicht gefunden, soll eine IOException mit aussagekräftiger Fehlerbeschreibung geworfen werden (z.B. <line> does not contain separator. wobei <line> durch den Zeileninhalt ersetzt wird).

Ist der gesamte Inhalt des BufferedReaders ausgewertet (d.h. BufferedReader.readLine() == null), soll schließlich die MultiMap zurückgegeben werden.

Test

Testen Sie Ihre read- und write-Methoden, indem Sie eine MultiHashMap<String, String> mit einigen Schlüssel-Wert-Paaren erzeugen und diese dann in einer Datei speichern. Anschließend laden Sie die Datei wieder und vergleichen den Inhalt der MultiHashMap. Um den Inhalt in eine Datei zu speichern müssen Sie sich zunächst einen BufferedWriter erzeugen, dessen Ausgabe in der Datei erfolgt. Dazu erzeugen Sie zunächst einen FileWriter mit Hilfe eines Files, z.B. so:

```
BufferedWriter writer = new BufferedWriter(new FileWritter(new File(“test.config”)));
```

Behandeln Sie die dabei möglicherweise auftretenden Exceptions durch einen try-catch-finally Block und stellen Sie sicher, dass der writer nach dem Aufruf der write-Methode in StringMapIO mittels writer.close() auf jeden Fall geschlossen wird.

Analog zum BufferedWriter können Sie einen BufferedReader mit Hilfe eines FileReaders und eines Files erzeugen, z.B.:

BufferedReader reader = new BufferedReader(new FileReader(new File("test.config")));

Wichtig: Beachten Sie, dass der FileWriter bestehende Dateien direkt – also ohne Warnung – überschreiben wird und dass Dateinamen, die keinen absoluten Pfad spezifizieren, relativ zum Arbeitsverzeichnis (engl. working directory) sind. Die Standardeinstellung für eine Launch Configuration in Eclipse ist das Projektverzeichnis. Wenn Sie das Programm über die Kommandozeile ausführen ist es das Verzeichnis in dem Sie den java-Befehl zum Start der Java Virtual Machine eingeben. Wenn Sie sich nicht sicher sind, sollten Sie lieber einen absoluten Pfad angeben (z.B. „c:/test/test.config" unter Windows oder „~/test.config" unter Linux). Sollten Sie einen absoluten Pfad verwenden, stellen Sie vor der Ausführung sicher, dass das Zielverzeichnis angelegt ist.

Zeichenfüllung (optional)

Wenn Sie sich das Dateiformat der Konfigurationsdateien genauer anschauen, werden Sie feststellen, dass die Zeichenkette „ = " (also Leerzeichen, Gleichzeichen, Leerzeichen) zur Trennung von Schlüsseln und Werten verwendet wird. Dadurch ergibt sich das Problem, dass diese Zeichenkette nicht mehr in Schlüsseln und Werten vorkommen darf. Denn wenn Sie einen Schlüssel mit der Zeichenkette „Schlü = ssel" und einen Wert mit der Zeichenkette „We = rt" in die Datei schreiben, können Sie später bei der resultierenden Zeile „Schlü = ssel = We = rt" nicht mehr eindeutig sagen, welche Zeichenkette die eigentliche Trennung markiert.

Eine einfache Lösung wäre es, diesen Fehler beim Schreiben zu erkennen indem Sie die Schlüssel und Werte auf die Zeichenkette, die zur Trennung dienen soll, testen und ggfs. eine IOException auslösen. Diese Lösung ist jedoch suboptimal, da Sie damit die gültigen Zeichenketten für Schlüssel und Werte einschränken.

Als Alternative bietet sich manchmal die Anpassung des Ausgabeformats an. Eine einfache Möglichkeit ist dabei die sogenannte Zeichenfüllung (engl. character stuffing) bei der Sie das Auftreten eines Trennzeichens dadurch verhindern, dass Sie dieses in allen Fällen, in denen es kein Trennzeichen darstellt, durch eine Kombination von Zeichen ersetzen.

In unserem Fall ist eine Möglichkeit die Ersetzung aller Gleichzeichen in einem Schlüssel oder Wert durch zwei Gleichzeichen. D.h. beim Schreiben ersetzen wir für jeden Schlüssel und für jeden Wert, die Gleichzeichen durch doppelte Gleichzeichen, z.B. mit String.replace(„=", „=="). Beim Lesen machen wir diese Ersetzung wieder rückgängig, z.B. mit String.replace(„==", „="). Dadurch kann ein einfaches Gleichzeichen in keinem Schlüssel und in keinem Wert im Dateiformat mehr vorkommen. Da unsere Trennung aus Leerzeichen, Gleichzeichen, Leerzeichen besteht, können wir dieses somit immer eindeutig identifizieren, denn obiges Beispiel würde nach Anwendung der Zeichenfüllung so aussehen: „Schlü == ssel = We == rt"

Implementieren Sie diesen Ansatz indem Sie die Codierung der Schlüssel und Werte vor dem Schreiben sowie die Dekodierung der Schlüssel und Werte nach dem Lesen vornehmen. Ändern Sie dazu die read- und write-Methode in StringMapIO.

Hinweis: Ein ähnliches Problem haben wir natürlich auch mit dem Zeilenumbruch in unserem Dateiformat, da dieser ebenfalls ein Trennzeichen (zwischen einzelnen Schlüssel-Wert-Paaren)

darstellt. Für unseren Editor ignorieren wir dieses Problem jedoch einfach, da die graphische Oberfläche später die Eingabe von Zeilenumbrüchen nicht unterstützen wird.

Aufgabe 3 – GUI (ohne Jack-Tests)

Die Klassen JEditor, JEditorModel und JEditorDialog implementieren eine einfache graphische Oberfläche zum Editieren der Inhalte einer MultiHashMap<String,String>. JEditor enthält eine main-Methode, mit der Sie die Anwendung starten können.

Die Anwendung besteht aus einem Fenster (JEditor). Das Fenster enthält eine Menüleiste (menu bar) mit einem Menü mit dem Namen „File“. In diesem Menü befinden sich drei Einträge (menu item) zum Laden und Speichern von Dateien und zum Beenden der Anwendung. Unterhalb der Menüleiste befindet sich der eigentliche Inhalt des Fensters (die sog. content pane). Darin befindet sich eine Werkzeugleiste (tool bar) mit zwei Schaltflächen (button) und eine Tabelle, die Schlüssel-Wert-Paare auflistet. Die Anordnung des Fensterinhalts erfolgt mit Hilfe eines sogenannten Layouts, das dafür sorgt, dass die Werkzeugleiste oberhalb der Tabelle platziert wird. Im speziellen wird ein BorderLayout verwendet.

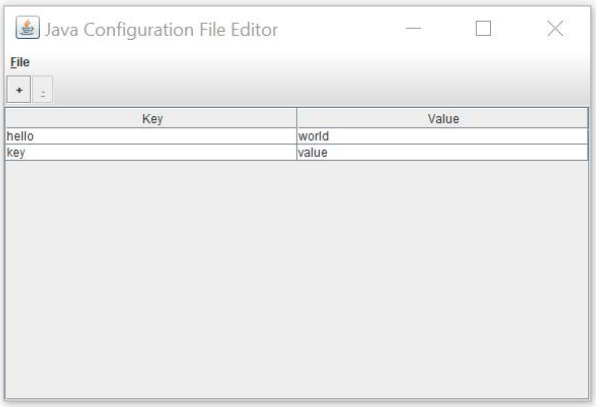

Abbildung 1 – Graphischer Editor

Der gesamte Aufbau dieses Inhalts befindet sich im Konstruktor der Klasse JEditor. Schauen Sie sich diesen einmal an und versuchen Sie den Inhalt des Fensters den Aufrufen zuzuordnen.

Mit Hilfe von Lambda Ausdrücken werden den Schaltflächen und Menüeinträgen sogenannte ActionListener zugewiesen, die Aufrufe an die „on…“ Methoden der Klasse JEditor weiterleiten. Die ActionListener werden von Swing genau dann aufgerufen, wenn der Nutzer auf den zugehörigen Menüeintrag oder die zugehörige Schaltfläche klickt.

Beispiel: Die Zeile remove.addActionListener(event->onRemove(event)) sorgt dafür, dass bei einem Klick auf die Schaltfläche zum Entfernen eines Tabelleneintrags die „onRemove“-Methode aufgerufen wird.

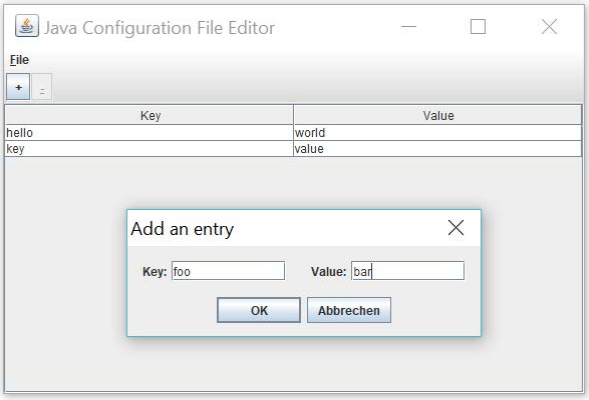

Abbildung 2 – Dialog zum Hinzufügen eines Schlüssel-Wert-Paares

Die einzelnen Funktionen der Schaltflächen und Menüeinträge sind bereits weitestgehend implementiert. Bei einem Klick auf die Schaltfläche „+“ wird beispielsweise mit Hilfe der Klasse JEditorDialog ein modaler Eingabedialog für ein Schlüssel-Wert-Paar geöffnet. Bei einem Klick auf die OK-Schaltfläche des Dialogs wird das eingegebene Schlüssel-Wert-Paar gespeichert. Analog dazu löscht die Schaltfläche „-“ das in der Tabelle ausgewählte Schlüssel-Wert-Paar und bei Auswahl der Menüeinträge werden modale Dateiauswahldialoge zum Laden und Speichern geöffnet. Die eigentliche Logik zum Laden und Speichern von Dateien ist jedoch noch nicht implementiert. Machen Sie das wie folgt:

Laden

Implementieren Sie die Methode **MultiMap<String, String> loadFromFile(File inputFile)** so, dass diese den Inhalt der Datei „inputFile“ mit Hilfe der **StringMapIO** Klasse lädt und zurückgibt. Dazu müssen Sie sich wie in Aufgabe 2 einen **BufferedReader** erzeugen. Stellen Sie beim Laden sicher, dass alle Exceptions behandelt werden und dass ihr **BufferedReader** in allen Fällen nach erfolgreichem Öffnen auch wieder geschlossen wird. Nutzen Sie dazu eine geeignete Kombination von try-catch-finally oder try-finally Blöcken.

Sollte ein Fehler beim Laden auftreten, behandeln Sie diesen, indem Sie zuerst einen Fehlerdialog anzeigen und nach Bestätigung des Fehlers durch den Nutzer eine leere **MultiMap** zurückgeben. Zum Anzeigen des Dialogs können Sie entweder **JOptionsPane** nutzen oder einfach die Hilfsmethode **JEditorDialog.showErrorDialog(title, message)** aufrufen.

Speichern

Implementieren Sie danach die Methode **void saveToFile(File outputFile, MultiMap<String, String> data)** so, dass diese den Inhalt von „data“ in die Datei „outputFile“ geschrieben wird. Dazu müssen Sie sich wie in Aufgabe 2 einen BufferedWriter erzeugen. Fangen Sie beim Speichern alle auftretenden Exceptions ab, indem Sie wie beim Laden einen Fehlerdialog anzeigen.

Erweiterungen (optional)

Edit-Schaltfläche: Fügen Sie dem Editor eine neue Schaltfläche zum Editieren eines Schlüssel-Wert-Paares hinzu. Dazu müssen Sie zunächst eine Schaltfläche erzeugen und der Menüleiste hinzufügen. Außerdem müssen Sie einen ActionListener registrieren. Schließlich müssen Sie dann die Programmlogik der Methoden „onAdd“ und „onRemove“ in geeigneter Weise kombinieren. D.h. Sie müssen das Schlüssel-Wert-Paar für die ausgewählte Zeile ermitteln und einen Dialog mit den Werten öffnen. Für den Fall, dass der Nutzer die OK-Schaltfläche klickt müssen Sie dann den alten Wert löschen, den neuen Wert einfügen und den Inhalt der Tabelle mittels „model.fireTableDataChanged()“ aktualisieren.

Dateinamen: Fügen Sie im Fensterbereich eine Beschriftung (Klasse: JLabel) ein, die am unteren Fensterrand angezeigt wird (getContentPane().add(label, BorderLayout.SOUTH)). Ersetzen Sie den Inhalt des Labels bei jeder erfolgreichen Lade- und Speicheroperation durch den vollständigen Pfad der Datei.

New-Menüeintrag: Fügen Sie dem „File“-Menü einen weiteren Menüeintrag „New“ hinzu. Beim Aufruf des Eintrags sollen alle Schlüssel-Wert-Paare gelöscht werden und die Beschriftung mit dem Dateinamen auf eine leere Zeichenkette zurückgesetzt werden.