Prof. Dr. Ralf Hinze
Sebastian Schweizer, M.Sc.
Markus Heinrich, M.Sc.

TU Kaiserslautern
Fachbereich Informatik
AG Programmiersprachen

Miniprojekt 1: Programmierpraktikum 2020

Ausgabe: 6. Mai 2020
Abgabe: 19. Mai 2020, 15 Uhr

GitLab Team Repositories

Für die Miniprojekte und Projekte nutzen wir das Versionsverwaltungssystem Git. Unser GitLab Server ist dabei eine zentrale Kopie des Repositories und bietet zudem eine Weboberfläche mit weiteren Funktionen.

Für jedes Abgabeteam haben wir ein eigenes GitLab Projekt erstellt. Darin befindet sich ein Ordner `MP1`, der die Vorlage für das erste Miniprojekt enthält. Clonen (`git clone`) Sie sich das Repository Ihres Teams und öffnen Sie dann den `MP1` Ordner in IntelliJ IDEA. Darin enthalten ist ein Gradle Projekt. IntelliJ IDEA lädt automatisch die definierten Abhängigkeiten herunter, wenn Sie das Projekt zum ersten Mal öffnen.

Ihre Lösungen müssen Sie in das lokale Repository committen (`git commit`) und dann auf den GitLab Server übertragen (`git push`). Diese beiden Git Funktionen stehen auch direkt in IntelliJ IDEA bereit. Sehen Sie sich dazu die weiteren Videos an. Sprechen Sie sich zur Bearbeitung der Aufgaben mit Ihren Abgabepartner*innen ab, damit Sie unnötige Konflikte vermeiden. In den Vorlesungen zu Git lernen Sie, wie Sie dennoch aufgetretene Konflikte beheben können.

Projektabnahmen und Betreuung

Bei der Durchführung der Miniprojekte und Projekte werden Sie von einer Tutorin/einem Tutor betreut. Die/der Tutor*in steht Ihnen für Rückfragen zur Verfügung und wird nach der Abgabe eine Projektabnahme durchführen, bei der jedes Teammitglied Teile der Abgabe erklären muss. So möchten wir sicherstellen, dass sich alle an der Abgabe beteiligt haben.

Die Schnittstelle Map

Im ersten Miniprojekt wollen wir eine `Map` Schnittstelle auf verschiedene Arten implementieren. Eine `Map` assoziiert Schlüssel mit Werten, sie ist also eine Abbildung. Zu jedem eingetragen Schlüssel ist ein Wert gespeichert. Wenn ein neuer Wert mit dem selben Schlüssel eingetragen wird, dann überschreibt dieser den bisher gespeicherten Wert.

```
public interface Map<K, V> {
    V get(K key);
    void put(K key, V value);
    void remove(K key);
    int size();
    void keys(K[] array);
}
```

Die `Map` ist dabei generisch bzgl. der Typen für Schlüssel und Werte. `K` ist der Typ für die Schlüssel, `V` der Typ für die Werte. Eine `Map<String, Integer>` weist also jedem eingetragenen `String` eine ganze Zahl zu und eine `Map<Integer, String>` weist Zahlen einen `String` zu.

Die Methode `V get(K key)` gibt den zum Schlüssel `key` gespeicherten Wert zurück, oder **`null`**, wenn kein Wert zu diesem Schlüssel gespeichert ist. Die Methode **`void`** `put(K key, V value)` speichert für den Schlüssel `key` den Wert `value`. Falls bereits ein Eintrag für `key` existiert hat, so wird dieser überschrieben. Die Methode **`void`** `remove(K key)` löscht den Eintrag für Schlüssel `key` falls dieser existiert, ansonsten bleibt die `Map` unverändert. Die Methode **`int`** `size()` gibt die Anzahl der in der `Map` gespeicherten Einträge zurück.

Die Methode **`void`** `keys(K[] array)` bietet eine Möglichkeit, alle in der `Map` gespeicherten Schlüssel abzufragen: Sie erwartet als Eingabe ein Array ausreichender Größe und speichert darin die in der `Map` eingetragenen Schlüssel ab. Die Reihenfolge der Schlüssel spielt dabei keine Rolle, jeder Schlüssel muss aber exakt ein Mal ins Array geschrieben werden. Falls das gegebene Array **`null`** ist oder wenn es zu klein ist, soll eine `IllegalArgumentException` geworfen werden[1]. Wenn das gegebene Array zu groß ist, werden die Indizes `0` bis `size() - 1` befüllt, der Rest des Arrays bleibt unverändert. Der Grund, warum die Methode *nicht* die Signatur `K[] keys()` hat, ist dass das Erstellen eines Arrays mit einem generischen Basistypen in Java nicht möglich ist. Das liegt an der sogenannten *Type Erasure*, die dazu führt, dass der Typparameter `K` zur Laufzeit nicht mehr bekannt ist.

Unsere Schnittstelle ähnelt stark der `java.util.Map`, bietet aber einen geringeren Funktionsumfang, sodass sie einfacher zu implementieren ist. Die Schnittstelle `Iterator` ist quasi eine Kopie von `java.util.Iterator`. Dieses Schema zum Iterieren wird Ihnen also auch bei einigen Bibliotheksfunktionen über den Weg laufen.

Zur Lösung der Aufgaben aus diesem Miniprojekt dürfen Sie die Java Standardbibliothek *nicht* verwenden! Sie müssen also alle benötigten Klassen und Methoden selbst implementieren. Dabei sollen Ihre Kompetenzen im Umgang mit den grundlegenden Sprachfeatures gefestigt werden. In der Praxis würde man niemals eine eigene `Map` Schnittstelle definieren sondern die `java.util.Map` mitsamt ihren Implementierungen aus der Standardbibliothek benutzen. Aber damit wären die Aufgaben hier hinfällig.

[1] `throw new IllegalArgumentException();`

Aufgabe 1 ListMap: Implementierung mit einer verketteten Liste

Implementieren Sie eine Klasse `public class ListMap<K, V> implements Map<K, V>`. Die Einträge der `Map` sollen in einer verketteten Liste verwaltet werden. Sie brauchen also eine separate Klasse für Listenelemente, in der Schlüssel, Wert und das nächste Listenelement gespeichert sind. Die einzelnen Methoden aus der `Map` Schnittstelle müssen diese Liste dann durchlaufen.

Aufgabe 2 ArrayMap: Implementierung mit einem Array

Wie Eingangs schon erwähnt ist das Erstellen von Arrays mit generischen Basistypen nicht möglich. Um Einträge intern in einem Array zu verwalten müssten Sie ein `Object[]` anlegen und bei jedem Zugriff einen Typcast vornehmen. Allerdings sind Typcasts mit generischen Typen unsicher (d.h. es können Typfehler an Stellen auftreten, an denen das eigentlich nicht der Fall sein sollte). Um diesen Problemen aus dem Weg zu gehen, geben wir die Typen in dieser Aufgabe fest vor. Für die Schlüssel nehmen wir hier `String`, für die Werte in der `Map` nehmen wir `Integer`.

Implementieren Sie eine Klasse `public class ArrayMap implements Map<String, Integer>`. Die Einträge der `Map` sollen in einem Array verwaltet werden. Da die Größe eines Arrays nach dem Erstellen nicht mehr verändert werden kann, müssen Sie in den Methoden `put` und `remove` ggf. ein neues Array mit abweichender Größe erstellen und die Einträge kopieren. Ob Sie im Array von vornherein einen zusätzlichen Puffer einplanen, um die Anzahl dieser Kopiervorgänge zu verringern, bleibt Ihnen überlassen.

Aufgabe 3 TreeMap: Implementierung mit einer Baumstruktur

Bei den bisherigen Implementierungen war es nötig, eine Liste oder ein Array komplett zu durchlaufen, um die Abwesenheit eines Schlüssels festzustellen. Auch das Auffinden eines vorhandenen Schlüssels dauert linear zur Größe der `Map`. Hier wollen wir die Elemente in einem Suchbaum speichern, sodass (wenn der Baum balanciert ist) nur logarithmisch viele Schritte nötig sind, um einen Schlüssel zu finden oder dessen Abwesenheit festzustellen.

Implementieren Sie eine Klasse `public class TreeMap<K, V> implements Map<K, V>`. Die Einträge der `Map` sollen in einem Suchbaum verwaltet werden. Sie brauchen also eine separate Klasse für Baumknoten, in der Schlüssel, Wert und zwei Teilbäume (Knoten) gespeichert sind.

Damit Sie Elemente des Schlüsseltyps `K` miteinander vergleichen können, erhält der Konstruktor `TreeMap` einen Parameter vom Typ `java.util.Comparator<K>`. Aus der Schnittstelle `Comparator<T>`[2] interessiert uns nur die Methode `int compare(T o1, T o2)`. Der zurückgegebene `int` ist

- < 0 wenn `o1` kleiner als `o2` ist,
- $= 0$ wenn beide Objekte gleich sind und
- > 0 wenn `o1` größer als `o2` ist.

Sie dürfen in dieser Aufgabe die Schnittstelle `Comparator` selbstverständlich importieren und die genannte Methode aufrufen, obwohl diese Teil der Standardbibliothek ist.

Tipp: Für die Implementierung der `keys` Methode müssen Sie sich überlegen, wie Sie sinnvoll alle Elemente des Baumes durchlaufen können. Da immer zwei Teilbäume zu betrachten sind, ist ein lineares Vorgehen mit einer Schleife nicht ohne Weiteres möglich. Sie können jedoch eine zusätzliche rekursive Methode definieren, die immer einen Knoten ins Array schreibt und sich dafür selbst zwei mal rekursiv aufruft.

Das Implementieren der Methoden `remove` ist in dieser Aufgabe optional. Wenn Sie diesen Teil auslassen möchten, dann werfen Sie einfach eine `UnsupportedOperationException`[3].

[2] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Comparator.html
[3] `throw new UnsupportedOperationException();`

Aufgabe 4 Häufigkeitsanalyse

Wir möchten die `Map` verwenden, um zu zählen, wie oft einzelne Wörter in der Bibel vorkommen. Wir haben Ihnen die Methode **`static`** `Iterable<String> Util.getBibleWords()` bereitgestellt, die Wort für Wort den Text aus der mitgelieferten Bibel Datei liefert. Das `Iterable` können Sie mit einer **`for`** Schleife iterieren:

```
for (String word : Util.getBibleWords()) {
    System.out.println(word);
}
```

Verwenden Sie eine derartige **`for`** Schleife, um in der Klasse `BibleAnalyzer` zunächst die Klassenmethode **`static void`** `countWords(Map<String, Integer> counts)` zu implementieren. Sie soll zählen, wie oft jedes Wort in der Bibel vorkommt. Die Häufigkeiten sollen in die übergebene `Map` eingetragen werden. Beachten Sie dabei, dass der Typ `Integer` ein Referenztyp ist, der neben den Werten des Basistyps **`int`** auch **`null`** sein kann. Berechnungen wie **`null`** `+ 1` oder die Umwandlung eines `Integer`s mit dem Wert **`null`** in einen **`int`** würden eine `NullPointerException` werfen. Sie müssen also bei Verwendung der `get` Methode prüfen, ob das Ergebnis **`null`** ist, bevor Sie weiter damit arbeiten.

Implementieren Sie dann die `main` Methode in dieser Klasse:

1. Sie soll eine `Map<String, Integer>` instanziieren und die zuvor implementierte Methode `countWords` damit aufrufen. Welche der drei Klassen Sie als konkrete `Map` Implementierung nutzen ist dabei zunächst egal. Zur Instanziierung der `TreeMap<String, Integer>` benötigen Sie einen `Comparator<String>`. Nutzen Sie `Comparator.<String>naturalOrder()`, um die Wörter lexikographisch sortiert zu betrachten.
2. Speichern Sie die Wörter (ohne Duplikate) in einem Array vom Typ `String[]`.
3. Sortieren Sie das Array aufsteigend nach Häufigkeit der einzelnen Wörter. Implementieren Sie dazu in der Klassenmethode **`static void`** `sort(String[] words, Map<String, Integer> counts)` ein geeignetes Sortierverfahren, welches das übergebene Array "in-place" sortiert. Wenn Sie Hilfsmethoden benötigen ergänzen Sie einfach weitere Klassenmethoden, diese können dann auch **`private`** sein.

   Rufen Sie die `sort` Methode dann von der `main` Methode aus auf.

   Gerne können Sie den Code für den Sortieralgorithmus aus der GdP Vorlesung, aus anderen Vorlesungen oder sonstigen Quellen übernehmen. Geben Sie in einem Kommentar an, um welches Sortierverfahren es sich handelt und ergänzen Sie einen Link zur Quelle. Sie dürfen jedoch *nicht* die Sortierfunktionalitäten der Java Standardbibliothek aufrufen.
4. Abschließend sollen alle Wörter auf die Konsole geschrieben werden. Eines pro Zeile, und zwar zuerst die Häufigkeit gefolgt von einem Leerzeichen und dem Wort. Beispiel:

   ```
   42 Engel
   123 Gott
   254 Amen
   ```

Die Antworten auf die folgenden Fragen müssen Sie nicht abgeben; sie werden in der Abnahme diskutiert:

Welches sind die 10 häufigsten Wörter in der Bibel? Welches ist das häufigste Nomen?

Untersuchen Sie, wie sich die unterschiedlichen `Map` Implementierungen auf die Ausführungszeit auswirken. Nennen Sie Gründe für die Unterschiede.