DHBW
Duale Hochschule
Baden-Württemberg

AIRBUS
DEFENCE & SPACE

Collections, Iteratoren und Exceptions

Lukas Abelt
lukas.abelt@airbus.com

DHBW Ravensburg
Wirtschaftsinformatik

Ravensburg
22. April 2019

Inhalt

Inhalt

Inhalt

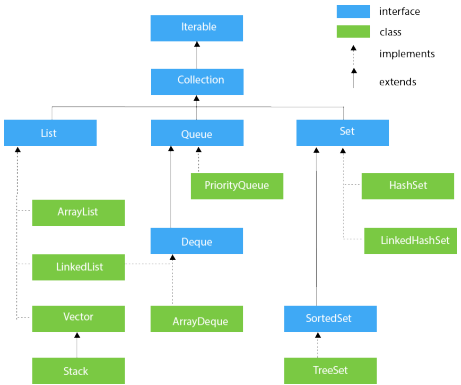
Collections

Was genau ist das eigentlich? (Vgl. [1]

- Kann sowohl bezeichnen:
  - Das Collections *Framework*
  - Das Collections *Interface* (Als Teil des Collection Frameworks)
- Collections sind grundlegend:
  - Datenstrukturen um eine Gruppe von Elementen zu speichern...
  - ...und zu manipulieren
- Das Collections Framework ist eine Zusammenfassung aus Klassen, Interfaces und Algorithmen

Collections Framework

Übersicht



Quelle: [1]

Collections Framework

Übersicht

- Bestes Anwendungsbeispiel für Generics
  - Listen sind für **alle** Klassen verwendbar...
  - ...ohne, dass irgendwelche Änderungen vorgenommen werden müssen
- C++ Äquivalent: STL Containers
  - Seit C++11 teil des Standards
  - Umfasst einige Datenstrukturen, die Collections nicht haben

Inhalt

Collection

Der Grundstein (Vgl. [3]

- Grundlegendes Interface für alle Subinterfaces und Klassen
- Definiert grundlegende Methoden zum:
  - Hinzufügen...
  - Entfernen...
  - Vergleichen...
  - Zählen...
  - ...von Elementen
- Noch keine (direkte) Methode zum *lesen* von Elementen

List

- Grundsätzliche Struktur für Listen von Elementen
- Keine Einschränkung der enthaltenen Elemente
- Erlaubt random access von Elementen
- Reihenfolge der Elemente wird beibehalten
  - Heißt, Elemente liegen in der Reihenfolge vor, wie sie hinzugefügt wurden
  - Sofern die Liste nicht anderweitig modifiziert wurde (Sortieren o.Ä.)

ArrayList

Implementierung des List Interfaces

- Daten werden in dynamsichen Array gespeichert
- Größe von diesem wird nach Bedarf (im Hintergrund oder auf Anfrage) angepasst
- Sehr ähnlich zur `Vector` Implementierung...
  - Jedoch nicht synchronisiert...
  - ...und deshalb nicht für multithreaded Anwendungen geeignet

LinkedList

Implementierung des List Interfaces

- Daten werden als Double-Linked-List gespeichert
- Ermöglicht Zugriff von beiden „Enden“ der Liste
- Implementiert sowohl `List` Interface, als auch `Deque` Interface
- Ähnlich wie `ArrayList`: Nicht thread-safe

Vector

- Existierte schon vor dem `Collections` Framework
- Wurde in dieses aufgenommen
- Im Grunde ähnlich wie `ArrayList`
- Aber: Thread-Safe

Queue

- Repräsentiert eine „Warteschlange“
- Für Elemente gilt **FIFO**:
  - **F**irst **I**n **F**irst **O**ut
  - Nur Zugriff auf vorderstes Element
- Reihenfolge der Elemente nicht unbedingt beibehalten
  - `PriorityQueue` sortiert Element automatisch

Deque

- **D**ouble **e**nded **Que**ue
- Subklasse von `Queue`
- Erlaubt jedoch Zugriff auf erstes und letztes Element der Liste
- Somit Verwendung auch zB. als **LIFO** Liste
  - **L**ast **I**n **F**irst **O**ut

Set

- Vergleichbar mit einer mathematischen Menge
  - Jedes Element kann genau einmal vorkommen
- Je nach Implementierung...
  - ...wird die Reihenfolge der Daten beibehalten
  - ...werden die Daten strukturiert

Map

Die Collection die keine ist

- Gehört mit zum Collections Framework
- Erbt jedoch **nicht** vom Collections Interface
- Implementiert jedoch sog. *collection-views*
- Speichert eine Gruppe an *KeyValuePairs*
  - Wobei hier die *Keys* nicht mehrfach vorkommen
- Je nach Implementierung sortiert oder nicht

Vgl. [6], [2]

Inhalt

Iteratoren

- Dienen zum traversieren von Listenstrukturen
- „Kennen“ das nächste Element in der Liste
- In C++: Ähnlich zur Nutzung von Pointern in Arrays
  - Überschreiben die increment/decrement (++) bzw. -) Operatoren
  - Vermeiden jedoch das „abdriften“ in unerlaubte Speicherbereiche
- In Java über zwei Interfaces definiert:
  - `Iterator`
  - `Iterable`

Iterable

Das Interface für Listen

- Ist das Super-Interface zum `Collection` Interface
- Somit in jedr Listenstruktur vorhanden
- Definiert drei Methoden:
  - `forEach()` - Führt für jedes Element die gegebene Aktion aus (Definiert über Lambda-Expressions)
  - `iterator()` - Gibt das `Iterator` Element für diese Collection zurück
  - `spliterator()` - Gibt ein `Spliterator` Element für diese Collection zurück

Siehe [4]

Iterator

Allgemeines

- Definiert im Grunde eine Position in einer Liste
- Über Methoden kann das Element „vor“ dem Iterator ausgelesen werden
- Je nach Implementierung auch das „dahinter“ (z.B. `ListIterator`)
  - Dadurch wird der Iterator jedoch in die entsprechende Richtung bewegt

Siehe [5]

Iterator

Methoden

- Das `Iterator` Interface definiert die Methoden:
  - `forEachRemaining()` - Führt die angegebene Operation für alle verbleibenden Elementen aus
  - `hasNext()` - Prüft, ob ein weiteres Element in der Collection vorhanden ist
  - `next()` - Gibt das nächste Element der Collection zurück
  - `remove()` - Entfernt das zuletzt zurückgegebene Element aus der Collection

Siehe [5]

Iteratoren

Vergleich zu C++

- `iterator()` Methode gibt in der Regel Iterator am Beginn der Collection zurück (Java)
  - `begin()`/`end()` Methode geben Iterator vor dem ersten bzw. hinter dem letzten Element des Containers zurück
- Iteratoren können (Standardmäßig) nur vorwärts bewegt werden (Java)
  - Iteratoren können vor und zurück bewegt werden (C++)
- C++ verfügt zusätzlich noch über „Reverse Iterator“
  - Diese starten hinter dem letzten Element und bewegen sich bei Inkrementieren rückwärts in der Liste
  - Einige Collections (z.B. `LinkedList`) implementieren ähnliches Verhalten über `descendingIterator()` Methode

Inhalt

Exceptions

Allgemein

- Signalisieren Fehler im entworfenen Programm
- Können auftreten:
  - Zur Laufzeit (Unchecked Exceptions)
  - Beim compilen (Checked Exceptions)
- Beenden die Ausführung des Programms vorzeitig
  - Wenn Sie nicht abgefangen werden

Checked Exceptions

- Werden bereits durch den Compiler erkannt
- **Müssen** abgefangen werden
  - Entweder direkt in der Methode
  - Oder durch „spezifizieren“ in der Methodensignatur
- Beispiel:
  - `FileNotFoundException`

Check Exception

Beispiel

```
1  import java.io.*;
2
3  class Main {
4      public static void main(String[] args) {
5          FileReader file = new FileReader("C:\\\\test\\\\a.
               ↪ txt");
6          BufferedReader fileInput = new BufferedReader(file)
               ↪ ;
7
8          // Print first 3 lines of file "C:\\test\\a.txt"
9          for (int counter = 0; counter < 3; counter++)
10             System.out.println(fileInput.readLine());
11
12         fileInput.close();
13     }
14 }
```

Unchecked Exceptions

Probleme zur Laufzeit

- Werden nicht durch den Compiler erkannt
  - Müssen nicht abgefangen werden
  - Beenden die Ausführung aber vorzeitig bei auftreten
- Können somit zur Laufzeit des Programmes auftreten
- Sind meist Indikator für Programmierfehler
  - Logikfehler
  - Falsche Nutzung von APIs
- Bekannte Vertreter:
  - `NullPointerException`
  - `ArrayIndexOutOfBoundsException`
  - `IndexOutOfBoundsException`
  - `NumberFormatException`

Checked Exceptions

Beispiel

```
1  class Main {
2     public static void main(String args[]) {
3        int x = 0;
4        int y = 10;
5        int z = y/x;
6     }
7  }
```

Behandeln von Exceptions

try-catch

- Fehler können über `try-catch` Konstrukte abgefangen werden
- Für verschiedene Fehler können unterschiedliche Abfangmethoden deklariert werden
- `finally` kann angegeben werden um Code auszuführen, nachdem der `try` Block bearbeitet wurde
  - Unabhängig davon, ob in diesem ein Fehler auftrat
  - Oder dieser behandelt wurde
  - Kann zum Beispiel genutzt werden, um offene Streams zu schließen
- Ein `catch` Block kann mehrere Fehler behandeln

Behandeln von Exceptions

Beispiel

```
1  public void readFile(String[] fNames, int index){
2      try{
3          String fName=fNames[index];
4          FileReader file = new FileReader(fName);
5          BufferedReader fileInput = new BufferedReader(file)
               ↪ ;
6      } catch(IOException e){
7      //Fehlerbehandlung für nicht gefundene Datei
8      }  catch(ArrayIndexOutOfBoundsException e){
9      //Fehlerbehandlung für invaliden Array Index
10     }
11     /*Weiterer Code*/
12 }
```

Behandeln von Exceptions

Beispiel

```
1 public int calc(int[] data, int index, int divisor){
2     try{
3         return data[index]/divisor;
4     } catch(ArrayIndexOutOfBoundsException|
        ↪ NumberFormatException e){
5         return 0;
6     }
7 }
```

Behandeln von Exception

Spezifizieren von Ausnahmen

- Fehler müssen nicht direkt durch die aufrufende Methode behandelt werden
- Alternativ Nutzung des `throws`-Keywords möglich
  - Gibt an, dass diese Methode den angegebenen Fehler auslösen kann
  - Verantwortung zum behandeln des Fehlers liegt dann am Aufrufer der Methode
- (Checked) Exceptions müssen spätestens in der `main()` Methode behandelt werden

Throws-Keyword

Beispiel

```
1 public int getElement(int[] data, int i) throws
      ↪ ArrayIndexOutOfBoundsException{
2     return data[i];
3 }
```

Auslösen von Exceptions

- Excpetions können auch „manuell“ ausgelöst werden
- Um dem Anwender zu signalisieren, dass ein Fehler aufgetreten ist
- Häufig in Verbindung mit selbstdefinierten Fehlerklassen
- Auslösen über Nutzung des `throw` Keywords
  - Ausgelöster Fehler muss das `Throwable` Interface implementieren

Throws-Keyword

Beispiel

```
1 public void throwDemo(){
2     throw new IOException();
3 }
```

Custom Exceptions

- Eigene Fehlertypen können über Klassen definiert werden, die das `Throwable` Interface implementieren
- Hierüber kann beispielsweise eine eigene Fehlermeldung definiert werden
- Bei eigenen Fehlern ist zu entscheiden, ob diese als Checked oder Unchecked Exceptions betrachtet werden sollen.
  - Unchecked Exceptions erben von `RunTimeExceptions`
  - Die Verwendung von unchecked Exceptions ist abzuwägen...
  - ...Auch wenn es die Entwicklung „vereinfacht“ (weil keine Compilerfehler auftreten)

Quellen I

[1] JavaTPoint. *Collections in Java*. 2018. URL: https://www.javatpoint.com/collections-in-java (besucht am 09. 04. 2019).

[2] JavaTPoint. *Java Map Interface*. 2018. URL: https://www.javatpoint.com/java-map (besucht am 09. 04. 2019).

[3] Oracle. *Interface Collection<E>*. 2018. URL: https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html (besucht am 09. 04. 2019).

[4] Oracle. *Interface Iterable<E>*. 2018. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html (besucht am 09. 04. 2019).

Quellen II

[5] Oracle. *Interface Iterator<E>*. 2018. URL: https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html (besucht am 09. 04. 2019).

[6] Oracle. *Interface Map<K,V>*. 2018. URL: https://docs.oracle.com/javase/8/docs/api/java/util/Map.html (besucht am 09. 04. 2019).

Kontakt

- E-Mail: `lukas.abelt@airbus.com`
- GitHub: `https://www.github.com/LuAbelt`
- GitLab: `https://www.gitlab.com/LuAbelt`
- Telefon(Firma): 07545 - 8 8895
- Telegram: LuAbelt