Informatik B

Vorlesung 10

Collection Framework

Rückblick Informatik A

- ADT (Abstrakter Datentyp)
  - **`Liste`**
    - **`Verweisliste`**
  - **`Keller`**
    - **`VerweisKeller`**
    - **`ArrayKeller`**
  - **`Schlange`**
    - **`VerweisSchlange`**
    - **`ArraySchlange`**
  - **`Baum`**
    - **`VerweisBaum`**
  - **`SuchBaum`**
  - **`AVLBaum`**
  - Hashing

Collection Framework

- *Collection* = Sammlung
- *Framework* = Rahmenstruktur
- Das *Collection Framework* ist eine vereinheitlichte Architektur um Sammlungen von Objekten zu verwalten
- Die Verwaltung erfolgt unabhängig von den enthaltenen Objekten selbst
- Eine C*ollection* ist ein Objekt welches eine Gruppe von Objekten repräsentiert
- Das Collection Framework verringert den Programmieraufwand und erhöht die Performance bei der Verwaltung von Objekten
- Es enthält Implementierungen der verschiedener Collections und Algorithmen, um diese zu verändern

Collection Framework

- Die Vorteile des *Collection Framework* sind:
  - Vermindert den Programmieraufwand da Datenstrukturen und Algorithmen vorhanden sind
  - Steigert die Performance durch effiziente Implementierung der Datenstrukturen und Algorithmen und durch den einfachen Wechsel zwischen unterschiedlichen Implementationen, je nach Anwendungsgebiet
  - Einfache Zugriffsmöglichkeiten auf eine Collection
  - Minimaler Lernaufwand, da viele Collections vereinheitlichte Methoden anbieten
  - Keine Notwendigkeit selber komplexe Collections zu implementieren, in der Regel reicht eine Unterklasse einer Collection aus

Collection Framework

- Das Collections Framework besteht aus:
  - **Collection Interfaces** Verschiedene Typen von Collections, wie Sets, Listen und Maps, die Basis-Interfaces des Frameworks
  - **General-purpose Implementations** Grundlegende Basiscollections
  - **Legacy Implementations** Collection-Klassen aus älteren Javaversionen
  - **Special-purpose Implementations** Spezielle Collections für spezielle Aufgaben, ggf. mit verändertem Laufzeitverhalten
  - **Concurrent Implementations** Collections für die Verwendung mit Threads
  - **Wrapper Implementations** Hinzufügen von Eigenschaften bei bereits bestehenden Collections (z.B. Synchronisation)
  - **Algorithms** Statische Methoden, um Collections zu verändern (z.B. sortieren)
  - **Array Utilities** Verschiedene Hilfsmethoden für Arrays mit primitiven Datentypen oder Referenztypen

Collection Interfaces

- Es gibt neun Collection Interfaces
- Das Basisinterface ist Collection
- Fünf Interfaces erweitern Collection: **`Set, List, SortedSet, Queue, BlockingQueue`**
- Die anderen drei Interfaces **`Map`**, **`SortedMap`**, **`ConcurrentMap`** erweitern nicht das Collection Interface, da sie eher Zuordnungen darstellen und weniger eine Collection
- Diese Interfaces enthalten aber *collection view* Methoden, wodurch sie wie eine Collection manipuliert werden können

Collection Interfaces

- **Alle** Methoden zur Modifikation einer Collection werden als optional gekennzeichnet
- Es kann also eine Implementation eines Interfaces geben in der nicht alle Methoden unterstützt werden
- In einem solchen Falle wird bei Aufruf der Methode eine **`UnsupportedOperationException`** geworfen
- In der Dokumentation wird angegeben, welche Operationen implementiert sind
- Das nicht alle Methoden unterstützt werden, liegt an der unterschiedlichen Implementierung der einzelnen Collections

Collection Interfaces

- Es werden im Sprachgebrauch folgende Vokabeln für Collections verwendet:
  - *Unmodifiable*: Collections, die nicht die Methoden **`add()`**, **`remove()`** und **`clear()`** implementieren
  - *Modifiable:* Gegenteil von *Unmodifiable*
  - *Immutable*: Änderungen an einem Collection-Objekt werden nicht nach außen hin sichtbar
  - *Mutable:* Gegenteil von *Immutable*
  - *Fixed-size lists:* Listen die immer eine feste Anzahl von Elementen beinhalten
  - *Variable-size:* Listen mit variabler Elementanzahl
  - *Random access lists*: Diese Listen garantieren einen schnellen (meist in konstanter Laufzeit) Zugriff auf die enthaltenen Elemente
  - *Sequential access* lists: Haben meist einen langsameren Zugriff auf einzelne Elemente, dafür können schnell Elemente eingefügt oder gelöscht werden

Collection Interfaces

- Einige Implementationen legen genau fest, welche Elemente gespeichert werden können
- Mögliche Einschränkungen sind:
  - Bestimmter Referenztyp
  - Keine **`null`**-Werte
  - Befolgen vordefinierter Eigenschaften
- Der Versuch zu solchen Collections nicht passende Elemente hinzuzufügen führt zu einer **`ClassCastException`**, einer **`IllegalArgumentException`** oder einer **`NullPointerException`**

Collection Framework

- Im wesentlichen gibt es drei unterschiedliche Grundkonzepte:
  - **`Set`**
  - **`List`**
  - **`Map`**

`Set`

- Ein **`Set`** ist eine Collection, die keine doppelten Elemente beinhaltet
- Formal bedeutet das:
  - Ein Set enthält keine zwei Elemente **`e1`** und **`e2`** für die gilt: **`e1.equals(e2)`**
  - Der Wert **`null`** darf nur einmal in einem **`Set`** enthalten sein
- Im Interface **`Set`** sind einige Bedingungen aufgeführt, die erfüllt sein müssen:
  - Alle Subklassen müssen Konstruktoren bereits stellen, die ein Set erstellen, in dem keine doppelten Elemente vorkommen
  - **`add(), addAll(), clear(), remove(), removeAll(), retainAll()`** sind optionale Methoden
  - Methoden, die Elemente hinzufügen, können beliebige Elemente ausschließen

`Set`

- Achtung: Falls Mutable Objekte in einem **`Set`** enthalten sind, kann dies zu Problemen führen, da durch die Änderung eines Objektes zwei Objekte innerhalb eines **`Set`** gleich sein könnten
- Außerdem kann durch eine Änderung eines Objektes dieses ggf. nicht wieder gefunden werden, da sich durch die Änderung ggf. auch der Hashwert ändert
- Beispiel: **`set1`**

`List`

- Eine **`List`** ist eine geordnete Collection (auch *sequence* genannt)
- In einer Implementation einer **`List`** kann genau angegeben werden, wo ein Element gespeichert werden soll, es gibt also eine Ordnung auf den Elementen
- Man kann direkt auf Elemente an einer bestimmten Position zugreifen und Elemente können gesucht werden
  - **`add(int index , Object element), get(int index)`**
  - Das erste Listenelement ist an Stelle **`0`**
  - Einfüge- und Löschoperationen können je nach Implementation der Liste unterschiedlich lange dauern (z.B. **`LinkedList`**)

List

- Listen erlauben in der Regel doppelte Elemente (auch **`null`** Werte)
- **`add(), addAll(), clear(), remove(), removeAll(), retainAll()`** sind optionale Methoden
- Das List-Interface definiert zwei Methoden zum Suchen von Elementen:
  - **`indexOf(Object o)`**
  - **`lastIndexOf(Object o)`**
- Achtung: Eine Suche kann viel Rechenzeit verschlingen, da sie oftmals linear verfolgt

Map

- Eine **`Map`** ist eine Collection die Schlüssel/Wert Paare verwaltet (Assoziativer Speicher)
- Vergleichbar mit einem Wörterbuch
- Jeder Schlüssel kann nur einmal in einer **`Map`** enthalten sein
- Jeder Schlüssel kann nur zu einem Wert gehören
- Das Interface **`Map`** ist nicht von **`Collection`** abgeleitet
- Dies wäre auch gar nicht möglich, da eine **`Map`** (die ja key/value-Paare verwaltet) nichts mit einer Methode **`add(Object o)`** anfangen könnte
- Eine **`Map`** definiert drei *collection views*, mit denen es möglich ist den Inhalt einer **`Map`** zu betrachten als:
  - Ein **`Set`** von Schlüsseln
  - Eine **`Collection`** von Werten
  - Ein **`Set`** von Schlüssel/Wert Paaren

Map

- Auf der **`Map`** ist keine bestimmte Reihenfolge der Elemente definiert
- Es gibt aber **`Map`** Implementationen, wie die Klasse **`TreeMap`**, die bestimmte Reihenfolgen garantieren
- Achtung: Falls Mutable Objekte in einer **`Map`** als Schlüssel enthalten sind, kann dies zu Problemen führen, da durch die Änderung eines Schlüssel-Objektes zwei Schlüssel innerhalb einer **`Map`** gleich sein könnten
- Eine Grundlegende **`Map`** sollte zwei Konstruktoren definieren:
  - Einen *Default Constructor* der eine leere Map erzeugt
  - Einen *Copy Constructor*
- Diese Vorgabe wird allerdings nur in der Dokumentation gemacht und kann nicht erzwungen werden
- **`clear(), put(), putAll() und remove()`** sind optionale Methoden

Collection Implementierungen

- Klassen die ein Collection-Interface implementieren haben meist einen Klassennamen der Form **`<ImplementationStyle><Interface>`**
- Einen Überblick über die allgemeinen Collection-Klassen (*General-purpose Implementations*) gibt folgende Tabelle:

		Implementationen				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Inter faces	**`Set`**	**`HashSet`**		**`TreeSet`**		**`LinkedHashSet`**
	`List`		**`ArrayList`**		**`LinkedList`**	
	`Map`	**`HashMap`**		**`TreeMap`**		**`LinkedHashMap`**

Collection Implementierungen

- Die allgemeinen Collection-Klassen unterstützen alle optionalen Methoden und können beliebige Objekte aufnehmen
- Die grundlegenden Collections sind nicht synchronisiert, sind also nicht geschützt durch die Manipulation durch mehrere Threads
- In der Klasse Collections sind so genannte *synchronization wrappers* enthalten
- Dies sind Methoden, die eine unsynchronisierte Collection in eine synchronisierte Collection *wrappen*
- Die abstrakten Klassen **`AbstractCollection`**, **`AbstractSet`**, **`AbstractList`**, **`AbstractSequentialList`** und **`AbstractMap`** bieten ein Grundgerüst der Basis Collection Interfaces, um den Programmieraufwand für eigene Collections zu minimieren
- In dem API wird genau beschrieben, wie die Methoden zu implementieren sind

Klassenbeispiele

- **`ArrayList`**:
  - Implementiert Listen-Funktionalität mit einem Array
  - Schneller Zugriff auf einzelne Elemente
  - Beim Einfügen und Löschen müssen ggf. die Werte umkopiert werden
  - Falls zu viele Elemente enthalten sind muss ein ganz neues Array angelegt und die Werte umkopiert werden
- **`LinkedList`**:
  - **`LinkedList`** ist eine doppelt verkettete Liste
  - Die Einträge besitzen eine Referenz auf den jeweiligen Nachfolger und Vorgänger
  - Schnelles einfügen oder löschen
  - Langsamer Zugriff auf bestimmte Elemente anhand eines Indexwertes

Klassenbeispiele

- **`HashSet`**:
  - Die Elemente werden anhand des Hashwertes (**`hashCode()`**) eingefügt
  - Schnelles Auffinden und Einfügen von Elementen
- **`TreeSet`**:
  - Elemente werden sortiert in einem Baum vorgehalten
  - Langsameres Einfügen und Löschen
  - Sortierte Ausgaben möglich
- **`LinkedHashSet`**
  - Eine schnelle Mengen-Implementierung, die sich parallel auch die Reihenfolge der eingefügten Elemente merkt
  - Schnelles Einfügen/Löschen

Klassenbeispiele

- **`HashMap`**
  - Implementiert einen assoziativen Speicher durch ein Hashverfahren
  - Es werden also key/value-Paare verwaltet
  - Schnelles Einfügen
- **`TreeMap`**
  - Exemplare dieser Klasse halten ihre Elemente in einem Binärbaum sortiert vor
  - Langsameres Einfügen/Löschen
  - Sortierte Ausgabe
- **`LinkedHashMap`**
  - Ein schneller Assoziativspeicher, der sich parallel auch die Reihenfolge der eingefügten Elemente merkt
  - Schnelles Einfügen/Löschen

Typsicherheit

- Ein großes Problem mit Datenstrukturen bis Java 5 war, dass sie prinzipiell offen für jeden Typ sind, da sie Objekte vom Typ Objekt beim Speichern entgegennehmen und diesen auch als Rückgabe liefern
- Dies birgt einige Fehlerquellen und Unbequemlichkeiten
- Sollen nur Objekte eines Typs verwaltet werden, müsste dies gesondert abgefragt werden
- Kümmert man sich nicht um die enthaltenen Objekt-Typen könnte man z.B. Äpfel und Birnen innerhalb einer Collection durcheinander werfen (was ja nicht immer gewünscht ist)
- Werden Objekte aus eine Collection geholt, müssen diese zunächst *gecastet* werden, da eine Collection Objekte vom Typ **`Object`** verwaltet
- Das bedeutet zusätzliche Tipparbeit und kann zur Laufzeit zu einer **`ClassCastException`** führen

Generics

- Seit Java 5 bietet die Collection-API Generics
- Sie gewährleisten bessere Typsicherheit, da dann nur spezielle Objekte in einer Collection verwaltet werden können
- Mit Generics lässt sich bei der Konstruktion einer Collection angeben, welche Typen zum Beispiel in einer Liste erlaubt sind
- Die Methodensignaturen werden durch die Angabe des generischen Typs automatisch angepasst
- Das ist zum einen eine Sicherheit für den Programmierer, hat aber noch einen weiteren Vorteil: zum Beispiel können die Typanpassungen weggelassen werden

Generics

- Generics werden in den Interfaces und Klassen definiert
- Das Interface Collection ist z.B. wie folgt definiert: **`interface Collection<E>`**
- Die Methoden im Interface sind z.B. wie folgt definiert: **`boolean add(E o), Iterator<E> iterator()`**
- **`E`** stellt dabei den generischen Typ dar
- Der Programmierer gibt dann bei der Erstellung eines Collection-Objektes an, welchen Typ **`E`** annehmen soll: **`List<MyData> l = new ArrayList<MyData>();`**
- In der so erzeugten Liste **`l`** dürfen jetzt nur noch Objekte der Klasse **`MyData`** und deren Unterklassen verwaltet werden
- Javadoc

Generics

- Die Generics stellen also Platzhalter für einzelne Typen dar, die dann bei der Programmierung konkretisiert werden können
- Verwendet man eine Collectionklasse ohne die Benutzung von Generics (also ohne Typsicherheit) gibt der Compiler eine *type safety* Warnung aus
- Generell sollten Collections nur typsicher verwendet werden
- Beispiel: **`generics1`**

Geschachtelte Generics

- Generics können auch geschachtelt werden
- Soll innerhalb einer Liste eine Liste mit Strings verwaltet werden, so muss folgendes angegeben werden:
  `List<List<String>> l = new ArrayList<List<String>>();`
- Die Schachtelung ist beliebig tief möglich
- Beispiel: **`generics2`**

Durchlaufen einer Collection

- Oftmals will man die Elemente einer Collection der Reihe nach bearbeiten
- Eine bequeme Art der Abarbeitung ist die Verwendung eines **`Iterator`**-Objektes
- Jede Collection-Klasse implementiert das Interface **`java.lang.Iterable`**
- Innerhalb des Interfaces ist lediglich die Methode **`iterator()`** definiert, die ein Objekt vom Typ **`Iterator`** zurückliefert
- Mit dem **`Iterator`**-Objekt kann eine Collection durchlaufen werden, dazu werden die Methoden **`hasNext()`**, **`next()`** und **`remove()`** zur Verfügung gestellt
- Beispiel: **`generics3`**

Fail-Fast Iterator

- Wird eine Collection mit einem **`Iterator`** durchlaufen, sollte diese nicht verändert werden, da sich dann das Iterator-Objekt in einem inkonsistenten Zustand befindet
- Daher können die Iteratoren eine Veränderung der Collection erkennen
- Wird eine Collection während des Durchlaufens verändert, wirft der **`Iterator`** beim nächsten Methodenaufruf eine Exception
- In früheren Javaversionen wurde keine Exception geworfen und der **`Iterator`** befand sich in einem nicht definierten Zustand, der eine weitere korrekte Verwendung des Iterators nicht garantierte
- Beispiel: **`generics4`**

`remove()`

- Will man dennoch beim Durchlaufen einer Collection einzelne Elemente löschen, kann dies mithilfe des **`Iterator`**-Objektes geschehen
- Die Methoden **`remove()`** löscht das zuletzt von **`next()`** gelieferte Element
- **`remove()`** kann nicht direkt hintereinander aufgerufen werden, sondern erst wieder nach dem Aufruf **`next()`**
- Würde man **`remove()`** dennoch hintereinander aufrufen kommt es zu einer **`IllegalStateException`**
- Die Methode **`remove()`** ist allerdings optional und wird daher nicht zwingend von jedem **`Iterator`**-Objekt unterstützt
- Beispiel: **`generics5`**

foreach

- Sollen alle Elemente einer Collection abgelaufen werden, braucht man nicht explizit das **`Iterator`**-Objekt
- Eine *foreach*-Schleife bietet ebenfalls das Gewünschte
- Bei einer *foreach*-Schleife wird in den runden Klammern zunächst der Typ angegeben, der in der Collection (oder auch Array) enthalten ist, darauf folgt der Variablenbezeichner und nach dem Doppelpunkt folgt die Collection (oder das Array)
- Beispiel: **`generics6`**

foreach

- Eine *foreach*-Schleife ist bequem zu implementieren
- Allerdings wird eine Collection (ein Array) immer komplett durchlaufen
- Eine *foreach*-Schleife besucht immer jedes Element
- Die Reihenfolge ist immer von vorne nach hinten
- Elemente können abgefragt aber nicht gesetzt werden
- Der Index ist nicht sichtbar
- Muss eine Schleife von obigen Bedingungen abweichen, muss eine herkömmliche **`for`**-Schleife verwendet werden
- Im Java Bytecode ist zwischen der herkömmlichen **`for`**-Schleife und der *foreach*-Schleife kein Unterschied, es handelt sich lediglich um eine etwas andere schreibweise

Zusammenfassung

- *Collection Framework*
- **`List`**
- **`Set`**
- **`Map`**
- **`Iterator`**
- foreach

Ausblick

- Konkrete Beispiele des Collection Framework
- Unterschiede zwischen den Implementationen
- Die Klasse **`Collections`**
- Generics selber implementieren
- **`Visitor`** vs. **`Iterator`**