1. Java Collections (Sets und Bags)

1.1. Motivation

Im Modul *Algorithmen und Datenstrukturen* 2 stehen die Datenstrukturen im Vordergrund. Sie dienen – wie der Name sagt – der strukturierten Speicherung von Daten. Die dabei verwendete Struktur bestimmt den Aufwand zum Einfügen, Suchen und Löschen von Daten. Je nach Anwendung und Anforderungen eignen sich einzelne Datenstrukturen mehr oder weniger gut. Bei der Software-Entwicklung gilt es also die jeweiligen Trade-Offs zu verstehen und zu bewerten, um eine gut begründete Wahl treffen zu können.

Programmierer und Software-Architekten müssen die richtigen Entscheidungen treffen. Projektleiter, Requirements-Ingenieure, Auftraggeber und andere Stakeholder sollten die Spielräume und Konsequenzen bei solchen Entscheidungen verstehen und ggf. bei der Priorisierung von Anforderungen berücksichtigen.

Für Java-Programmierer stellt das *Java Collection Framework* eine verbreitete Programmbibliothek mit einer Grundausstattung an allgemein verwendbaren Datenstrukturen dar. Die Schnittstellen werden als im Wesentlich bekannt vorausgesetzt (z.B. aus dem Modul OOP 2). Zur Bewertung verschiedener Implementierungen wird auf Werkzeuge wie die *asymptotische Komplexität* oder *Vor- und Nachbedingungen* zurückgegriffen (als bekannt vorausgesetzt z.B. aus dem Modul *Algorithmen und Datenstrukturen 1*).

Als Anknüpfungspunkt an Bekanntes und Auslegeordnung grundsätzlicher Konzepte bei Datenstrukturen werden zunächst einige Varianten Array-basierter Datenstrukturen betrachtet.

1.2. Lernziele

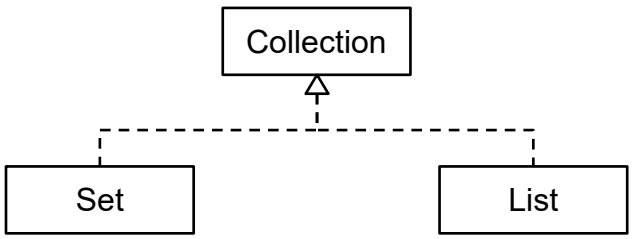
- Sie können den Unterschied zwischen *Bag-Semantik* und *Set-Semantik* erklären und für gegebene Anforderungen eine geeignete Variante auswählen.
- Sie können Vor- und Nachteile der sortierten und unsortierten Datenablage im Array angeben und für gegebene Anforderungen eine geeignete Variante begründet auswählen.
- Sie können einfache Array-basierte Collections für Bag-Semantik und Set-Semantik (mit sortierter und unsortierter Datenablage) implementieren.
- Sie können für eine gegebene Implementierung die Komplexität der wichtigsten Zugriffsoperationen *add(…)*, *contains(…)*, *remove(…)* ermitteln.

1.3. Das Java Collection Framework

Wenn von Daten im Zusammenhang mit *Collections* gesprochen wird, sind immer mehrere *Elemente* eines Typs gemeint, also beispielsweise mehrere Integer-Zahlen, mehrere Strings usw. (manchmal aber auch beliebige *Object*s). Solche Daten können in verschiedenen Strukturen gespeichert werden. Alle diese Strukturen zur Datenablage haben Vor- und Nachteile und man muss je nach Verwendungszweck eine passende auswählen. Änderungen des Use-Case oder zusätzliche Anforderungen machen es bisweilen notwendig, die Datenstruktur zu wechseln.

Um dies zu erleichtern, bietet das Java Collection Framework eine Reihe von Interfaces als Abstraktionsmechanismen an. Solange die darin definierten Eigenschaften gleich bleiben, kann die Implementierung ausgetauscht werden, ohne dass die Zugriffe angepasst werden müssen.

Hier ein Ausschnitt aus der Interface-Hierarchie des Java Collection Frameworks:



Jede Datenstruktur des Collection Frameworks ist eine *Collection* (implementiert dieses Interface). Für alle Collections sind elementare Methoden definiert, wie *Element hinzufügen*, *Element entfernen*, *Element suchen*,…). Die Implementierung der verwendeten Datenstruktur kann ausgetauscht werden, ohne dass die Teile des Programms, die diese Datenstruktur benutzen, angepasst werden müssen. Ein Wechsel der Datenstruktur kann sich aber bemerkbar auf die Performance auswirken. Es kann sich auch die Funktionsweise verändern. Möchte man dies nicht, sollte man evtl. eines der Sub-Interfaces wie *Set* oder *List* benutzen, um damit genauere Anforderungen an die Implementierung zu stellen.

Die Java 11 API schreibt zum Interface *Collection*:

> *The root interface in the collection hierarchy. A collection represents a group of objects, known as its elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The JDK does not provide any direct implementations of this interface: it provides implementations of more specific subinterfaces like Set and List. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.*

1.4. Set-Semantik vs. Bag-Semantik

Betrachtet man Datenstrukturen als *Sammlungen* von Objekten (bzw. *Elementen*), braucht man im Wesentlichen drei Operationen: Ein Element der Sammlung hinzufügen, ein Element daraus entfernen und prüfen, ob ein Element darin enthalten ist.

Ein Element, das hinzugefügt wird, ist anschliessend enthalten, bis es wieder entfernt wird.

Doch Vorsicht: Was soll passieren, wenn ein Element mehr als einmal hinzugefügt wird?

Hier gibt es zwei Möglichkeiten, die beide je nach Anwendung nützlich sein können. In einer mathematischen *Menge* ist jedes Element höchstens einmal enthalten, egal wie oft es hinzugefügt wird. Eine andere Möglichkeit sind *Bags*, in denen die gleichen Elemente mehrfach vorkommen können[1].

Entsprechend werden auch im Java Collection Framework (und in entsprechenden Bibliotheken anderer Sprachen) zwei Fälle unterschieden:

- **Set-Semantik:** Die Set-Semantik erlaubt nicht, dass ein Element mehrmals in der Collection erscheint. Wird ein bereits enthaltenes Element noch einmal hinzugefügt, ändert sich nichts. In Java implementieren solche Collections das Interface *Set* und damit wegen der Vererbungshierarchie auch das Interface *Collection*.
- **Bag-Semantik:** Die Bag-Semantik erlaubt Mehrfachvorkommen eines Elementes. In Java implementieren solche Collections das Interface *Collection*, aber nicht das Interface *Set*.

Beispiel: In eine leere Sammlung werden die Werte *1, 5, 3, 2, 4, 1, 3, 6* nacheinander eingefügt.

- Enthaltene Elemente bei Bag-Semantik:

- Enthaltene Elemente bei Set-Semantik:

1.5. Realisierung im Java Collection Framework: Methoden *add*, *remove*, *contains*

Wir konzentrieren uns im Folgenden auf die drei Methoden *add, remove* und *contains* des Collection Interface zum Vergleich verschiedener Datenstrukturen. Das Verhalten dieser Methoden ist in der Dokumentation zu den Schnittstellen *Collection* und *Set* beschrieben[2].

Für die Methoden `boolean add(E e)` und `boolean remove(E e)` kann man folgende Merkregel verwenden:

- Wird keine Exception geworfen, war der Aufruf im Sinne der Spezifikation erfolgreich: *add* hat ein entsprechendes Element hinzufügt – gemäss Set- oder Bag-Semantik – bzw. *remove* hat ein Element entfernt, falls eines vorhanden war.
- Der *boolean* Rückgabewert signalisiert, dass die Collection verändert wurde. Wird also *false* zurückgegeben, war die Operation erfolgreich, brauchte aber zum Etablieren der Nachbedingung die Collection nicht zu verändern.

---

[1] Relationale Datenbanken kennen ebenfalls die Unterscheidung zwischen *Set* und *Bag*. Grundsätzlich werden dort Daten in *Bags* gespeichert. Jedoch gilt für Primärschlüssel bzw. mit UNIQUE gekennzeichnete Attribute die *Set*-Semantik und es gibt eine Operation zur *Duplicate Elimination*, die aus einem *Bag* ein *Set* macht.

[2] s. z.B. http://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Collection.html und http://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Set.html

Einige Aufrufbeispiele für eine Variable `Collection<Integer> c`:

1. `c.add(1)` gibt `false` zurück. Mögliche Erklärung(en):

2. `c.add(1)` gibt `true` zurück. Mögliche Erklärung(en):

3. `c.add(null);` Mögliche Ergebnisse:

4. `c.remove(1); c.remove(1);` gibt beide Male `true` zurück. Mögliche Erklärung(en):

5. Aufruf um zu prüfen, ob `c` mindestens ein Element `1` enthält:

6. Programm zum Löschen aller Elemente `1` aus `c`, wenn gilt `c instanceof Set`:

7. Programm zum Löschen aller Elemente `1` aus `c`, wenn gilt `!(c instanceof Set)`:

1.6. Anordnung der Daten in einem Array

Collections – egal ob Set oder Bag – können mit einem Array implementiert werden. Elemente, für die eine Ordnungsrelation definiert ist, können dabei grundsätzlich auf zwei Arten gespeichert werden: Entweder in beliebiger Reihenfolge oder geordnet. Bei geordneter Speicherung kann mit Binärer Suche effizienter gesucht werden, dafür ist der Aufwand zum Hinzufügen und Entfernen von Elementen grösser.

Beispiel: In eine leere Sammlung werden die Werte *1, 5, 3, 2, 4, 1, 3, 6* nacheinander eingefügt.

- Elemente im Array bei sortierter Speicherung mit Bag-Semantik:

- Elemente im Array bei sortierter Speicherung mit Set-Semantik:

1.7. Vergleich verschiedener Collection-Varianten

Für *Collections* gibt es also zwei verschiedene Arten von Verhalten (*Set* und *Bag*) sowie zwei verschiedene Implementierungsstrategien mit Arrays (*sortierte* und *unsortierte* Speicherung). Dabei sind beliebige Kombinationen möglich. Insgesamt gibt es also vier Varianten zu betrachten:

		Sortierung	
		Sortiert	**Nicht sortiert**
Semantik	**Set**	*SortedSet*	*UnsortedSet*
	Bag	*SortedBag*	*UnsortedBag*

		UnsortedBag	SortedBag	UnsortedSet	SortedSet
	Inhalt nach dem Einfügen von: 12, 5, 28, 47, 12, 8				
Asyptotische Komplexität im **Worst Case** (n Elemente in der Collection)	add(E e);				
	contains(Object o)				
	remove(Object o)				
	Besonders gut geeignet für:				