Ergänzungen zum Collections Framework

Das Collections Framework ist beschrieben in der Ergänzung zur javadoc. Der derzeitige link ist[1]:
http://download.oracle.com/javase/6/docs/technotes/guides/collections/overview.html

Zweck

In der Hilfe von Oracle heißt es, dass dieses framework ein Objekt sei, das eine Gruppe von Objekten repräsentiere. Ich halte die Wahl des Begriffes Objekt hier für unangemessen. Der Begriff Struktur ist sicher passender. Das Ziel ist, mit einer einheitlichen Struktur die Schnittstelle von Sammlungen zu beschreiben, um auf sie in stets gleicher Weise unabhängig von der eigentlichen Implementation zugreifen zu können.

Vorteile

- geringerer Aufwand beim Programmieren
- man kann besser mit Änderungen der Implementation auf geänderte Bedingungen reagieren
- Verstehen, Entwurf und Kommunikation von Objekten untereinander werden leichter

Inhalt

- das interface Collection selbst
- vorgegebene grundlegende Lösungen für viele Typen, aber auch eine Vielfalt von spezialisierten Lösungen
- typische Algorithmen wie Sortierung

Was gehört dazu?

Noch einmal zur Erinnerung: Es geht um interfaces. Es gibt hier also nicht den Begriff der Vererbung: "wer nichts hat, kann auch nichts vererben". Trotzdem gibt es wie bei der Vererbung eine Baumstruktur von Implementationsbeziehungen und man spricht auch hier – völlig zu Recht – von "erweitern" [extends].

Collection Interfaces

Es gibt also ein interface Collection. Collection wird erweitert von

- Set, List, SortedSet, NavigableSet, Queue, Deque, BlockingQueue und BlockingDeque.

Obwohl auch sie zum Collections Framework gehören, erweitern die folgenden Map – interfaces nicht Collection :

- Map, SortedMap, NavigableMap, ConcurrentMap und ConcurrentNavigableMap

Grundlagen

Auf die weiteren Betrachtungen, die sich auf der o.a. Seite finden, gehe ich hier nicht ein. Wichtiger scheint mir eine Erläuterung, die das Unterscheiden der grundlegenden Sammlungstypen ermöglicht.
Grundsätzlich stellen – abgesehen von den statischen Varianten – alle natürlich die Möglichkeit bereit, Elemente hinzuzufügen, zu entfernen und nach Elementen zu suchen.

1 Dieser Text orientiert sich im ersten Teil stark an dem dortigen.

Dennoch gibt es wichtige Unterschiede.

Set

Set, also deutsch Menge, lässt sich leicht aus den mathematischen Anforderungen an eine Menge ableiten: Eine Menge kann ein Objekt nur genau einmal enthalten. Ein Set-Objekt darf also dasselbe Element nicht noch einmal hinzufügen lassen. Dabei kann es auf verschiedene Art reagieren. Eine Möglichkeit ist, die Anforderung zu ignorieren, eine andere ist, mit einer Fehlermeldung zu reagieren.
Man muss hier allerdings sehr darauf achten, dass es bei *"dasselbe"* wirklich um dasselbe im Sinne der Methode equals(...) für die Objekte geht.
Betrachten wir die folgende Klasse mit sechs Methoden zum Testen.

```
public class GleichheitTesten{
    // Vergleich von Strings mit ==
    public boolean test1(){
        String erster = "Hund";
        String zweiter = "Hund";
        return (erster==zweiter);
    }
    //Vergleich von Strings mit equals
    public boolean test2(){
        String erster = "Hund";
        String zweiter = "Hund";
        return erster.equals(zweiter);
    }
    //Vergleich von neu erzeugten String-Objekten
    public boolean test3(){
        String erster = new String("Hund");
        String zweiter = new String("Hund");
        return (erster==zweiter);
    }
    //Vergleich von neu erzeugten String-Objekten
    public boolean test4(){
        String erster = new String("Hund");
        String zweiter = new String("Hund");
        return erster.equals(zweiter);
    }
    // Vergleich von neu erzeugten Objekten
    public boolean test5(){
        Intern erster = new Intern("Hund");
        Intern zweiter = new Intern("Hund");
        return (erster==zweiter);
    }
    // Vergleich von neu erzeugten Objekten
    public boolean test6(){
        Intern erster = new Intern("Hund");
        Intern zweiter = new Intern("Hund");
        return erster.equals(zweiter);
    }

    private class Intern{
        String wort;
        public Intern(String wort){
            this.wort=wort;
        }
    }
}
```

Eine Testmethode aus einer Testklasse dazu:

```
public void test()
{
      GleichheitTesten gleichhe1 = new GleichheitTesten();
      assertEquals(true, gleichhe1.test1());
      assertEquals(true, gleichhe1.test2());
      assertEquals(false, gleichhe1.test3());
      assertEquals(true, gleichhe1.test4());
      assertEquals(false, gleichhe1.test5());
      assertEquals(false, gleichhe1.test6());
}
```

Der Grund für den Wert falsch bei der letzten Rückgabe ist, dass die interne Klasse nicht – wie es die Klasse String macht – die Methode equals überschreibt, so dass die von Object geerbte Methode equals verwendet wird, die allein testet, ob es sich um dasselbe Objekt handelt.
Interesssant ist in diesem Zusammenhang das erste Ergebnis, das zeigt, dass Java in dem Fall kein neues Objekt "Hund" erzeugt, sondern dasselbe verwendet. Man ist bei String-Objekten aber immer auf der sicheren Seite, wenn man nicht == verwendet, sondern equals.

List

Eine Liste (auch sequence) enthält die Objekte in einer wohldefinierten Reihenfolge. Als Beispiele von Klassen, die das interface List implementieren, kennen wir schon ArrayList und Linked List. Für beide lohnt sich ein Vergleich der Implementation, da an ihnen Fragen der Effizienz von Datenstrukturen im unterschiedlichen Anwendungskontext diskutiert werden können.

Queue

Eine Queue[1] – wer schon einmal in England war, kennt die Bedeutung von "queue up please" – ist eine Warteschlange. Man denkt zunächst nicht, dass es dafür verschiedene Varianten gibt, aber nicht nur die typische Warteschlange, bei der man sich hinten anstellen muss und vorn drankommt (first in first out) gehört dazu, sondern auch ein Stack, also ein LIFO-Struktur (last in first out), genauso wie eine andere wichtige Datenstruktur die Prioritätswarteschlange, bei der die Elemente nach einer Bewertungsfunktion eingeordnet werden.

Beispiel LinkedList

Die LinkedList ist eine dequeue, also eine "double ended queue". Sie stellt nicht eine einfach verkettete Liste dar, bei der man zunächst auf den Kopf zugreifen muss, um sich dann von Element zu Element hindurch zu hangeln, sondern eine vorwärts und rückwärts verkettete Liste, bei der jedes Element nicht nur seinen Nachfolger kennt – so es denn einen gibt – sondern auch seinen Vorgänger.

Map

Maps sind so organisiert, dass man die Objekte als Paar ablegt. Die Elemente sind also Schlüssel – Wert Paare [key-value] und man kann direkt auf einen Wert über die Angabe des Schlüssels zugreifen.

1 An dieser Stelle nur ein kleiner Hinweis, sich einmal mit dem Begriff der "abstrakten Datentypen" ADT zu beschäftigen.

Hashcode

Eine interessante Variante der Datenspeicherung nutzt intern eine solche Paarbildung. Dabei wird zu jedem Objekt ein hashcode berechnet, der der die Position des Objektes in der Datenstruktur beschreibt. Der Zugriff auf so abgespeicherte Daten ist sehr schnell, so dass man sie zu nutzen versucht, wenn es auf solche Schnelligkeit ankommt. Hier ist allerdings das Problem der Kollision zu lösen. Die Werte können sich wiederholen und man muss sich Gedanken machen, wie man dann vorgeht.

Tree

Eine weitere Methode der effektiven Speicherung von Objekten ist das Ablegen in Baumstrukturen. Binäre Bäume sollte man unbedingt verstanden haben. Wer mehr Interesse hat, kann sich mit weight-balanced trees beschäftigen.