Collections und Generics Überblick

Seit JDK 1.0 gibt es die 'traditionellen' Collections

- **Vector, Stack, Dictionary, Hashtable, Properties, BitSet**.

Seit Java 2 JDK 1.2 wurde ein neues Collection-API eingeführt. Einige bisherige Klassen (Vector und Stack) wurden neu implementiert, viele neue Interfaces und Klassen eingeführt und die Struktur insgesamt verbessert.
Alle Collections sind im Package *java.util* enthalten.
Es gibt seitdem die Interfaces

- ***Collection*, *List*, *Set*, *Map, Queue***

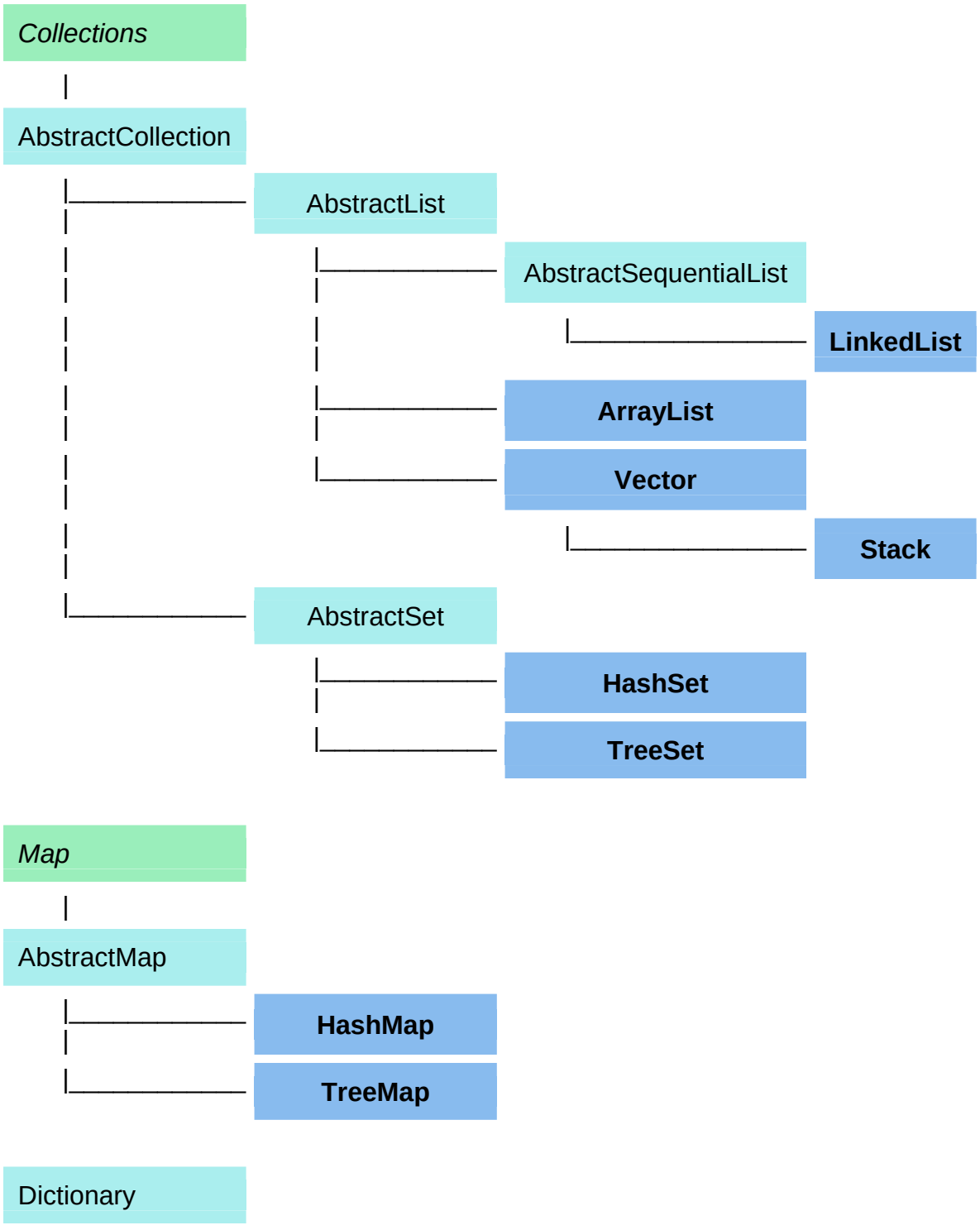
sowie die

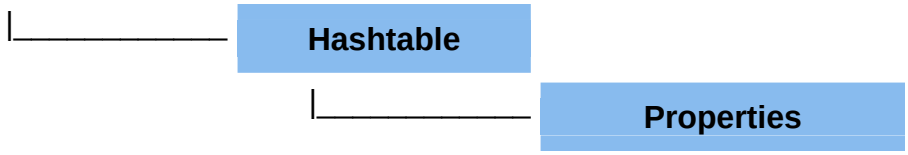
- *List* implementierenden Klassen
  - **LinkedList, ArrayList, Vector, Stack**
- *Set* implementierenden Klassen
  - **HashSet, TreeSet, LinkedHashSet**
- *Map* implementierenden Klassen
  - **HashMap, TreeMap, Hashtable**
- *Queue und Deque*
  - **LinkedList, PriorityQueue, ArrayBlockingQueue**

und noch einige weitere Interfaces und Klassen (z.B. **Collections**).
Bitte nicht das Interface *Collection* (ohne s) und die Klasse Collections (mit s) verwechseln.

Die im folgenden erläuterten Strukturen und Eigenschaften beziehen sich auf Java 2 JDK 1.3.

Vereinfachte Vererbungsbäume zu einigen Collections-Klassen mit dazugehörenden Container.





Was sind Generics?

Generics bzw. Generizität und generische Elemente sind ein Sprachmittel von Programmiersprachen. Im Falle von Java heißt dies, dass Klassen und Methoden (Methoden auch außerhalb von parametrisierten Klassen) mit Typparametern parametrisiert werden können. Es geht also u. a. um Konstrukte wie einen "Vector of String", also einen Vector, der nur String-Objekte aufnehmen kann.

Gründe für Generics

Einer der Hauptgründe für die Einführung von Generics mit der Java-Version 1.5 ("Tiger") war der Wunsch, die existierenden Collectionklassen wie z. B. Vector, Map, Set weiterhin wieder verwendbar zu halten, aber diese um Typsicherheit zu erweitern. So waren vor der Java-Version 1.5 häufig sogenannte Downcasts (explizite Verwendung eines spezielleren Typs für ein Objekt) nötig, wenn man die Objekte aus einem Container auslesen wollte. Hierzu ein Beispiel, wie die Summation einer Liste von Integern ohne Generics üblicherweise realisiert wurde:

```
1: List intList = new Vector();
2: intList.add(new Integer(1234));
3: intList.add(new Integer(5678));
4: intList.add(new Integer(4711));
5:
6: // Addition aller Werte
7: int sum = 0;
8: for(int i=0;i<intList.size();i++) {
9: sum += ((Integer)intList.get(i)).intValue(); // unkontrollierter
Downcast!
10: }
```

In Zeile 9 wird hier das aktuelle Element der Liste mit der Methode get() ausgelesen. Diese Methode liefert jedoch ein Objekt vom Typ Object, so dass sein Zahlenwert nicht direkt auszulesen ist (es könnte ja auch ein String sein). Daher musste der Programmierer hier einen Downcast (in diesem Falle auf den Typ Integer) vornehmen, um die Methode intValue() aufrufen und somit den Zahlenwert auslesen zu können.
Dies war bei einem kleinen, übersichtlichen Codefragment wie dem obigen natürlich ohne große Probleme möglich; der Programmierer hatte eben dafür zu sorgen, dass nur Integer-Objekte in der Liste landen.
Sollte aber nun diese Funktionalität durch eine Methode in einer Bibliothek zur Verfügung gestellt werden, musste eine manuelle Typüberprüfung durchgeführt werden, da der Programmierer der Bibliotheksfunktion nicht mehr sicherstellen konnte, dass seine Methode wirklich nur mit Integer-Objekten aufgerufen wird:

```
1: public int sum(List intList) {
2: int result = 0;
3: for(int i=0;i<intList.size();i++) {
4: Object o = intList.get(i);
5: if(o instanceof Integer) // Typüberprüfung
6: result += ((Integer)o).intValue(); // "sicherer" Downcast
7: else
8: throw new IllegalArgumentException("Keine Liste von Integern!");
9: }
10: return result;
11: }
```

Auch hier sind die Nachteile schnell ersichtlich. Es ist ein beträchtlicher Teil des Codes nur für die Typüberprüfung nötig, und bei mehreren Funktionen würde dieser Code immer wieder benötigt werden. Außerdem wird, im Falle eines Objekts vom falschen Typ in der Liste, eine Exception erst zur Laufzeit ausgelöst. Daher ist die aufrufende Funktion auch noch für eine Fehlerbehandlung verantwortlich, soll das Programm nicht mit einer Exception abstürzen.

Generics – Grundlagen

Anlegen einer generischen Klasse

```
class Box<T>
{
        private T val;

        void setValue( T val )
        {
                this.val = val;
         }

        T getValue()
         {
         return val;
         }
}
```

Erzeugen eines Objektes einer generischen Klasse.

```
Box<String>  stringBox = new Box<String>();
Box<Integer> intBox    = new Box<Integer>();
Box<Point>   pointBox  = new Box<Point>();
```

Anlegen einer generischen Methode

```
class Util
{
	public static <T> T random( T m, T n )
	{
		return Math.random() > 0.5 ? m : n;
	}
}
```

Aufruf einer generischen Methode

```
String s = Util.random( "Essen", "Schlafen" );
```

Erzeugung von generischen (paramentrisierten) Collections

Man muss keine generischen Collections verwenden!

```
List players = new ArrayList();
Player laraFarm = new Player();
players.add( laraFarm );
players.add( "ätsch" );

Player p1 = (Player) players.get( 0 );  // OK
Player p2 = (Player) players.get( 1 );  // BUM!
```

Man kann sie verwenden und damit leicht Typfehler vermeiden!

```
List<Player> players = new ArrayList<Player>();
```

Man kann generische Datentypen auch schachteln!

```
List<List<String>> las = new LinkedList<List<String>>();
```

Es handelt sich dabei um eine verkettete Liste deren Inhalt Strings sind.