In diesem Kapitel:

Collections

The problem with people, who have no vices is that generally you can be pretty sure they're going to have some pretty annoying virtues.
- Elizabeth Taylor

Das java.util Package enthält viele brauchbare Interfaces und Klassen. Grob kann man sie einteilen in zwei Kategoriene: Collections und alles andere. In dieser Zusammenstellung lernen Sie die Klassen und Interfaces der Collections kennen. Die anderen Klassen und Interfaces werden später behandelt.

1.1. Collections

Collections (manchmal auch *Container* genannt*)* sind Behälter, welche Objekte speichern und verwalten können, so dass die Objekte sinnvoll und effizient eingesetzt werden können. Was effizient heisst, hängt von Ihrem Einsatz der Collections ab. Daher bieten Collections in Java unterschiedliche Typen und Implementationsstufen an.

Im Package java.util finden Sie Interfaces und Klassen, welche eine generelle Basis für Collections (Containers) bieten, also ein generelles Framework. Dieses Framework liefert Ihnen ein Set von Collection Interfaces und brauchbare, sofort einsetzbare Implementationen dieser Interfaces.

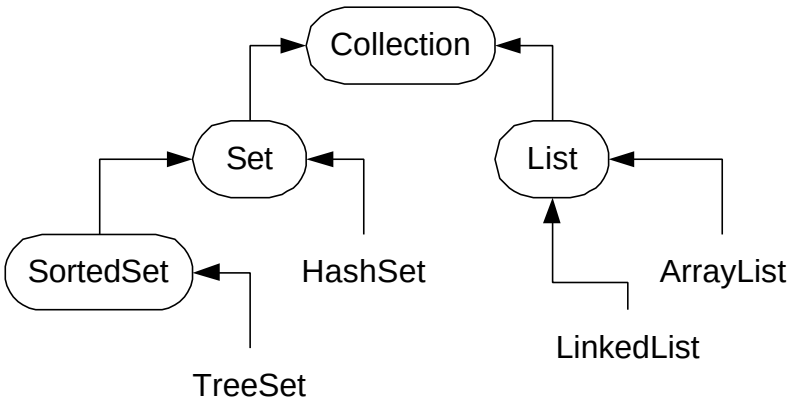
COLLECTIONS

Das Collection Framework wurde neu entworfen, nachdem in den ersten Versionen von Java bereits bestimmte Grundfunktionen, Klassen und Interfaces, zur Verfügung standen. Ziel des Designs des Collection Frameworks war es, wenige aber universell brauchbare Basisbausteine zu definieren und zu implementieren. Ziel war es nicht, eine möglichst umfangreiche Sammlung von Basisklassen für alle denkbaren Anwendungen zur Verfügung zu stellen, da ein solches Set in der Regel unübersichtlich und komplex wird.

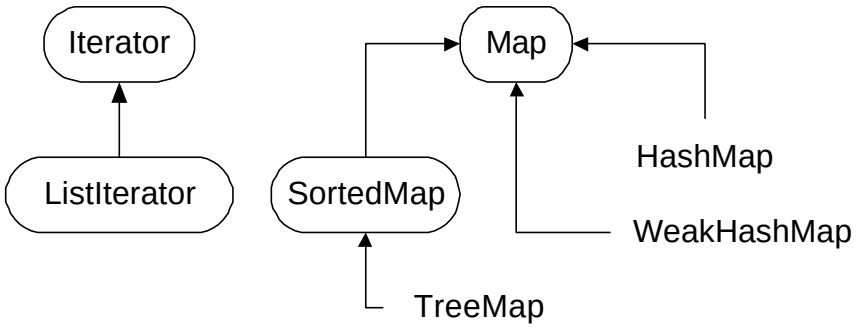
Eine Möglichkeit die Grösse des Frameworks klein zu halten besteht darin ein breites Anwendungsgebiet und hohe Abstraktion anzustreben an Stelle einer sehr detaillierten Liste von Klassen und Interfaces. Die Kern-Collection- Interfaces stellen Methoden zur Verfügung, gestatten die Definition aller geminsam nutzbaren Methoden und überlassen es den Implementationen die unbrauchbaren Methoden über Exceptions, java.lang.UnsupportedOperationException, zu verbieten und Fehler abzufangen.

Die Collections Interfaces und Implementationen in java.util sind:

- `Collection` - die Wurzel, das grundlegende Interface für Collections. Dieses Interface stellt beispielsweise Methoden wie `add(), remove(), size(), toArray()` und `iterator`.



- `Set` - dies ist eine Collection, welche jedes Element nur einmal aufnehmen kann (keine Duplikation) und deren Elemente nicht in einer bestimmten Reihenfolge abgespeichert sind. Dieses Interface erweitert `Collection`.
- `SortedSet` - eine Menge von Objekten, welche im Gegensatz zu oben sortiert sind. Dieses Interface erweitert das `Set` Interface.
- `List` - eine Collection, deren Elemente in einer bestimmten Ordnung angeordnet sind, ausser die Liste wird verändert. Die `List` erweitert das `Collection` Interface. Der Begriff Liste bedeutet nicht "Linked List", sondern wird hier allgemeiner verstanden, obschon linked Lists eine mögliche Implementation des Interfaces sind.
- `Map` - Dies sind Abbildungen von Schlüsseln auf Werte, jeweils ein Schlüssel auf einen Wert. `Map` erweitert `Collection` nicht sondern steht selbständig da.Die definierten Methoden haben zum Teil die selben Namen wie im `Collection` Interface, werden aber unabhängig davon definiert.
- `SortedMap` - Ein Map, dessen Schlüssel sortiert sind (erweitert das `Map` Interface).



- `Iterator` - ein Interface für Objekte, welche andere Objekte aus einer Collection zurück liefern, jeweils eines pro Aufruf der entsprechenden Methoden. Diese Objekttypen werden durch Aufruf der Methode `List.Iterator` generiert, zurückgeliefert.
- `ListIterator` - ein Iterator für `List` Objekte. Dieses Interface enthält weitere Methoden, welche für die Bearbeitung von Listen besonders nützlich sind.

Die Interfaces `SortedSet` und `SortedMap` garantieren, dass auf eine sortierte Art und Weise durch Elemente iteriert werden kann. Wie eine Ordnung definiert werden kann, werden wir noch sehen.

Das `java.util` Package stellt auch mehrere konkrete Implementationen dieser Interfaces zur Verfügung, mit deren Hilfe Sie wahrscheinlich die meisten Probleme bereits lösen können:

- `HashSet` - Dies ist ein Set, welches mit Hilfe einer Hash-Tabelle implementiert wird. Die Implementation stellt die üblicherweise benötigten Methoden zur Verfügung: Methoden zum Suchen , zum Hinzufügen, zum Entfernen von Objekten, in der Regel auf eine solche Art und Weise implementiert, dass die Performance weitestgehend unabhängig von der Grösse des Sets und der enthaltenen Objekte ist.
- `TreeSet` - Ein `SortedSet`, welches mit ausbalancierten binären Bäumen implementiert wird. Die Implementation von suchen und modifizieren ist langsamer als jene im `HashSet` aber die Elemente bleiben sortiert.
- `ArrayList` - Eine `List` Klasse, welche mittels eines Arrays veränderlicher Grösse implementiert ist. Das Einfügen oder Entfernen eines Elementes am Anfang der Liste ist recht zeitaufwendig, speziell bei langen Listen. Aber das Kreieren solcher Listen ist recht effizient und speziell für Zufallszugriffe (random access) sehr schnell.
- `LinkedList` - Eine `List` Klasse, deren Elemente doppelt verhängt sind. Modifikationen sind in der Regel recht schnell; aber Zufallszugriffe sind langsam. Eine sinnvolle Anwendung dieser Klasse wären Warteschlangen.
- `HashMap` - Eine Implementation eines Maps mittels einer Hash-Tabelle. Diese Klasse ist sehr universell einsetzbar mit brauchbaren Zugriffszeiten und Einfügungszeiten.
- `TreeMap` - Eine Implementation von `SortedMap` mittels balancierten binären Bäumen, welche die Elemente mittels deren Schlüssel sortiert hält. Diese Klasse ist sinnvoll einsetzbar, falls die Daten sortiert sind oder bleiben sollen. Die Zugriffszeiten mittels Lookup Keys sind mässig.
- `WeakHashMap` - Eine Implementation von `Map` mittels eines Hashtables. Die Schlüssel der Tabelle werden mittels schwachen Referenzobjekten referenziert. Limitierter Einsatz.

Alle obigen Implementationsklassen sind `Cloneable` und `Serializable`.

Zuerst werden wir uns nun mit Iterationen beschäftigen, da diese für alle `Collection` Klassen nützlich sind. Anschliessend widmen wir uns der Frage "Sortieren" oder genereller "Ordnungsrelation". Auch dieses Konzept spielt bei vielen `Collection` Klassen eine grosse Rolle. Anschliessend gehen wir auf die einzelnen `Collection`-basierten Datentypen, Klassen und Interfaces, ein. Danach betrachten wir spezielle Themen, wie unmodifizierbare (*unmodifiable*) und synchronisierte Collections. Als nächstes lernen Sie, wie Sie eigene Iterationen und Collections schreiben können, falls Sie spezielle Varianten oder spezielle Implementationen der Standard Collections benötigen. Zum Abschluss betrachten wir die sogenannten "legacy collections" in java.util, also jene Klassen, welche vor der Definition des gesamten Collection Konzepts bereits in Java definiert waren und daher in vielen Systemen eingesetzt sind, beispielsweise Properties, welche immer noch eine sehr wichtige Rolle spielen und auch in Zukunft spielen werden.

1.1.1. Konventionen betreffend Ausnahmen

Im Rahmen der Collection Klassen wurden einige Regeln oder Konventionen betreffend der Ausnahmen definiert, welche in Collection Klassen und Interfaces eingesetzt werden und welche so allgemein gültig sind, dass wir sie besser einmal, statt bei jedem Auftreten beschreiben.

- Methoden, welche in einer Implementation eines Interfaces optional sind, werfen eine `UnsupportedOperationException`, falls sie nicht implementiert werden. Bei der Besprechung der einzelnen Klassen und Interfaces werden wir diese Methoden speziell kennzeichnen.
- Methoden und Konstruktoren, welche Elemente (individuell oder als Teile einer anderen Collection) aufnehmen, um der Collection hinzugefügt zu werden, werfen eine `ClassCastException`, falls das Element nicht vom für die Collection passenden Datentyp ist.
- Methoden oder Konstruktoren, welche Elemente (individuell oder als Teile einer anderen Collection) aufnehmen, um der Collection hinzugefügt zu werden, werfen eine `IllegalArgumentException`, falls der Wert nicht zur Collection passt. Beispielsweise definieren Subsets Collections Wertebereiche, welche für die Elemente dieser Collection erlaubt sind.
- Methoden, welche individuelle Elemente der Collection zurückliefern, werfen eine `NoSuchElementException`, falls die Collection leer ist.
- Methoden oder Konstruktoren, welche Referenzen als Parameter akzeptieren, werfen in der Regel eine `NullPointerException`, falls eine `null` Referenz übergeben wird. Diese Regel hat eine Ausnahme: falls die Methode ein Objekt als Parameter erwartet, beispielsweise für das Hinzufügen, Entfernen oder Nachschlagen eines Objekts in einer Collection, dann wird auch das `null` Objekt als Element akzeptiert.

Die anderen Ausnahmen werden wir von Fall zu Fall diskutieren.

1.2. Iteration

Das `Collection` Interface definiert eine `iterator()` Methode, welche ein Objekt zurückliefert, welches das `Iterator` Interface implementiert. Dieses verfügt über folgende Methoden:

`public boolean` **`hasNext()`**
: liefert `true`, falls die Iteration (die Collection) weitere Elemente besitzt.

`public Object` **`next()`**
: liefert das nächste Element in der Iteration (Collection). Falls keine weiteren Elemente existieren, wird eine `NoSuchException` geworfen.

`public void` **`remove()`**
: entfernt aus der darunterliegenden Collection jenes Element, welches als letztes von der Iteration zurückgeliefert wurde. Pro `next()` Aufruf kann `remove()` lediglich einmal aufgerufen werden, sonst wird eine `IllegalStateException` geworfen.

Das folgende Programmbeispiel verwendet alle drei Methoden der `Iterator` Klasse und entfernt alle Zeichenketten, welche eine vorgegebene Länge überschreiten.

```
package einführendebeispiele;
import java.util.*;

class Iteration {
  Iteration() {
    Vector vec = new Vector();      // Vector ist eine Implementation
    for (int i=0;i<10; i++){        // des Collection Interfaces
      int jR = (int)(10*Math.random());  // generieren eines Strings
      String strTemp = "";
      for (int j=0; j< jR; j++ )
        strTemp = strTemp + j;      // Hinzufügen zum Vector
      vec.addElement(i+".tes Element:"+strTemp);
    }
    entferneLangeStrings(vec,20);  // Methodenaufruf
  }
  public static void main(String[] args) {
    Iteration iter = new Iteration();
  }
  public void entferneLangeStrings(Collection col, int maxStrLänge) {
    // Collection definiert iterator()
    // diese Methode liefert ein Iterator Objekt, mit dem
    // alle Elemente der betreffenden Collection abgefragt,.. werden können
    // (iteriert mittel hasN/(n)ext() über die Collection: alle Elemente)
    Iterator it = col.iterator();
    while(it.hasNext() ) {
      String str = (String)it.next();
      // nächstes Element der darunterliegenden Collection
      if (str.length() > maxStrLänge){
        // Element (String) ist änger als erlaubt
        it.remove();
 System.out.println("Das Element '"+str+"' wurde aus dem Vektor entfernt");
      }    }  }
}
```

Zuerst verwenden wir die Methode `iterator()`, um ein `Iterator` Objekt zu erhalten. Mit diesem wird der Inhalt der `Collection` durchlaufen, ein Element nach dem andern.

Dann durchlaufen wir die `while()` Schleife solange, bis kein weiteres Element mehr in der `Collection` vorhanden ist, also `hasNext() false` liefert.

Das Element des Durchlaufs erhalten wir mit der Methode `next()`. Da die `Collection` (der `Vector`) allgemeine Objekte enthält, müssen wir diese zuerst noch in Zeichenketten umwandeln (*casting*).

Falls diese Zeichenkette länger als der maximal erlaubte Wert ist, wird das Element, also jenes, welches wir gerade mit `next()` erhalten haben, aus der `Collection` mit `remove()` entfernt. `remove()` bezieht sich auf jenes Element, welches wir zuletzt mit der `next()` Methode erhalten haben. Die `remove()` Methode ist die einzige sichere Methode ein Element aus einer `Collection` zu entfernen.

Das `ListIterator` Interface erweitert Iterator und fügt weitere Methoden hinzu. Diese zusätzlichen Methoden gestatten die Manipulation einer geordneten Liste, eines `List` Objekts, während der Iteration. Dabei können Sie vorwärts, mit `hasNext()` und `next()`, oder rückwärts, mit `hasPrevious()` oder `previous()`, oder vorwärts und rückwärts durch Mischen dieser Methoden, durch die Liste hindurchwandern.

Das folgende Beispiel zeigt, wie Sie rüchwärts durch eine Liste, in diesem Fall in Form eines Vektors, hindurchwandern können.

```
package einführendebeispiele;
import java.util.*;
public class ListIteratorBeispiel {

  public ListIteratorBeispiel() {
    List l = new Vector();
    for (int i=0; i<10; i++)
      l.add(i+"tes Element");
    this.rückwärtsEineListeDurchlaufen(l);
  }
  public static void main(String[] args) {
   ListIteratorBeispiel listIteratorBeispiel1 = new ListIteratorBeispiel();
  }
  public void rückwärtsEineListeDurchlaufen(List list) {
    // List erweitert Collection definiert iterator() und listIterator()
    // diese Methode liefert ein Iterator/ListIterator Objekt, mit dem
    // alle Elemente der betreffenden Liste abgefragt, .... werden können
    // (iteriert mittel hasN/(n)ext() über die Liste: alle Elemente)

    // ListeIterator ist ein Interface, welches Iterator erweitert
    // wir gehen rückwärts, müssen als ganz hinten anfangen
    ListIterator it = list.listIterator(list.size());
    while(it.hasPrevious() ) { // ja, es gibt nach Elemente davor
      String str = (String)it.previous(); // beispielsweise dieses
      // vorgängiges Element der darunterliegenden Collection
      System.out.println("Das Element '"+str+"' wurde im Vektor gefunden");
    }
  }}
```

Das Programm bestimmt als erstes (Methode `rückwärtsEineListeDurchlaufen()`) einen `ListIterator` zur Liste (in unserem Fall ist das `List` Interface im `Vector` implementiert).

Dieser wird auf das Ende der Liste gesetzt (`list.size()`).

Nachher durchlaufen wir einfach die Liste solange wie wir noch Vorgängerelemente finden. Die Elemente unter dem `ListIterator` (-Fenster) werden angezeigt, aber nicht entfernt. Insgesamt werden die Elemente an den Positionen `0` bis `list.size()-1` durchlaufen.

Falls wir Elemente entfernen möchten, würde uns wieder die `remove()` Methode zur Verfügung stehen. Diese entfernt jenes Element, welches als letztes gelesen wurde, also mit `next()` oder `previous()` festgelegt wurde:

`public void` **`set(Object elem)`**

> Ersetzt das zuletzt gelesene Element (mit `next()` oder `previous()`) durch das Element `elem`. Falls Sie bereits einmal seit dem letzten Aufruf von `next()` oder `previous()` `remove()` oder `add()` aufgerufen haben, wird eine `IllegalStateException` geworfen.

`public void` **`add(Object elem)`**

> Fügt ein Element an der Stelle ein, bei der sich der `ListIterator` gerade befindet, also vor dem Element, welches als nächstes zurück geliefert würde, oder aber am Ende, falls `hasNext()` `false` liefert. Falls Sie `previous()` nach `add()` aufrufen, erhalten Sie genau das neu eingefügte Element.

Der Kontrakt, das Verhalten, der Methode `remove()` wird übernommen: es wird eine `IllegalStateException` geworfen, falls `remove()` aufgerufen wird und eine `add()` oder `set()` Methode nach dem letzten Aufruf von `next()` oder `previous()` aufgerufen wurde.

Die Kontrakte für `Iterator` und `ListIterator` enthalten keine *Snapshot* Garantie. In anderen Worten: es kann passieren, dass sich die Collection genau zu dem Zeitpunkt (durch einen anderen Thread oder eine Anwendung) verändert, zu dem der Iterator ein Element bestimmt.

Falls Sie Snapshot Garantie benötigen, müssen Sie vor dem Aufruf der Methoden eine Kopie der Collection herstellen, beispielsweise mittels einer Array Kopie:

```
public Iterator snapshotIterator(Collection col) {
      return new ArrayList(col).iterator();
}
```

Viele der in java.util implementierten Iteratoren implementieren ein *fail-fast* Verhalten: diese Iteratoren bestimmen, ob und wann eine Collection modifiziert wurde und ob dies durch den Iterator selbst geschah. Falls dies nicht der Fall ist, wird eine `ConcurrentModificationException` geworfen. Dies garantiert ein sicheres Verhalten der Iteratoren.

1.3. *Definition von Ordnungsrelationen mit* Comparator *und* Comparable

Das Interface `java.lang.Comparable` kann durch irgend eine Klasse implementiert werden, welche Objekte enthält oder definiert, welche sortiert werden können.

Das Interface besitzt lediglich eine einzige Methode:

`public int` **`compare(Object other)`**

liefert einen Wert der entweder kleiner, gleich oder grösser als null ist, je nachdem ob das Objekt auf das die Methode angwandt wird kleiner, gleich oder grösser als das Objekt `other` ist.
Falls der Wert 0 zurückgegeben wird, sollte die Methode `equals` angewandt auf dieses Objekt mit `other` als Parameter den Wert `true` liefern.
Falls die beiden Objekte nicht vergleichbar sind (beispielsweis ein `Integer` mit einem String `Objekt`), dann wird eine `ClassCastException` geworfen.

Die Ordnung, welche mittels `compareTo()` definiert wird, entspricht der *natürlichen Ordnung* dieser Objekte, also der Ordnung, welche der natürlichsten Ordnung entspricht, die für diese Objekte definiert werden kann. Dasselbe gilt für die `equals()` Methode.

Viele bestehende Klassen sind `Comparable`, beispielsweise `String, java.io.File, java.util.Date` und alle Basisdatentypen- Wrapperklassen. Beispielsweise besitzt die Klasse `Short` die Methoden `compareTo(Object)` und `compareTo(Short)`. Der Compiler wird falls möglich immer die zweite Form benutzen, da bei der ersten zuerst noch eine Typenumwandlung geschehen muss.

Falls Sie eine Klasse einsetzen, welche `Comparable` nicht implementiert, können Sie in der Regel ein `java.util.Comparator` Objekt zur Verfügung stellen. Das `Comparator` Interface besitzt folgende Methode:

`public int` **`compare(Object o1, Object o2)`**

stellt eine Ordungsrelation analog zu `Comparable.compareTo()` zur Verfügung.

Mit `Comparable` und `Comparator` Objekten können Sie Objekte sortieren und `List` Objekte durchsuchen (ordnen), oder die statischen `Collection` Methoden `sort()` und `binarySearch()` verwenden. Die `Collection` Klasse stellt Ihnen statische Methoden `min()` und `max()` zur Verfügung, mit denen das grösste oder das kleinste Element in einer `Collection` bestimmt werden kann.

Falls Sie Vergleiche ohne Berücksichtigung der Gross- und Kleinschreibung durchführen möchten, stellt Ihnen die `String` Klasse dafür einen `Comparator` zur Verfügung: `String.CASE_INSENSITIVE_ORDER`.

1.4. Das Collection *Interface*

Wir haben bereits gesehen, das die meisten Collection-Typen Untertypen des `Collection` Interfaces sind. Die einzige Ausnahme bildet `Map`. In diesem Abschnitt beschreiben wir das `Collection` Interface und die speziellen Implementation davon im `java.util` Package. Auf die `Map` Klasse werden wir später zurück kommen. Auch die Implemention eigener `Collection` Klassen verschieben wir auf später.

Die Basis für das Collection System in Java bildet das `Collection` Interface. Dies haben wir bereits weiter vorne gesehen: beispielsweise sind `List` oder `Set` (Interfaces) Erweiterungen des Interfaces `Collection`. Als macht es sicher Sinn, mit dem `Collection` Interface zu starten, falls wir Collections verstehen möchten.

`public int` **`size()`**
: liefert die Grösse dieser Collection, also die Anzahl Elemente, welche sie im Moment enthält. Der Maximalwert wird durch `Integer.MAX_VALUE` beschränkt, selbst wenn die Collection mehr Elemente enthält!

`public boolean` **`isEmpty()`**
: liefert `true`, falls die Collection kein Element enthält.

`public boolean` **`contains(Object elem)`**
: liefert `true`, falls die `Collection` das Objekt `elem` enthält. Das heisst, falls beim Aufruf der Methode `equals()` auf einem der Objekte der `Collection` mit `elem` als Parameter `true` resultiert, liefert auch `contains() true`. Falls `elem` das `null` Element ist, liefert die Methode `true` genau dann, falls die `Collection` ebenfalls `null` als Element enthält.

`public Iterator` **`iterator()`**
: liefert einen Iterator, welcher alle Elemente dieser Collection durchlaufen kann.

`public Object[ ]` **`toArray()`**
: liefert ein neues Array, welches Refernzen auf alle Elemente dieser Collection enthält.

`public Object[ ]` **`toArray(Object[ ] dest)`**
: liefert ein Array, welches Referenzen auf alle Elemente dieser Collection enthält. Falls alle Elemente in `dest` Platz haben, dann wird `dest` auch zurückgeliefert. Falls `dest` mehr Elemente als die Collection enthält, wird das erste Element in `dest`, welches keinen Inhalt der Collection enthält durch das Element `null` aufgefüllt.
  Falls die Elemente in `dest` nicht Platz haben, wird ein neues Array vom selben Typus wie `dest` generiert und dieses grössere Array zurückgeliefert.
  Falls der Datentyp von `dest` inkompatibel mit dem Datentyp der Elemente in der Collection ist, dann wird eine `ArrayStoreException` ausgelöst.

`public boolean` **`add(Object elem)`**
: stellt sicher, das das Element `elem` enthält. Falls die Collection beim Hinzufügen verändert wurde, wird `true` zurück geliefert.Falls die Collection das selbe Element mehrfach enthalten kann, wird auch `true` zurück geliefert.Falls die Collection jedes

Element genau einmal aufnehmen darf, aber bereits ein solches Element vorhanden ist, dann wird (optional) `false` zurückgeliefert.

`public boolean` **`remove(Object elem)`**
entfernt eine Instanz des Elements `elem` aus der Collection. Falls diese Änderung erfolgreich durchgeführt werden kann, liefert die Methode `true`. In diesem Fall war das Element in der Collection vorhanden. Falls `elem null` ist, liefert die Methode `true`, falls das `null` Objekt in der Collection vorkommt (optional).

Alle Methoden, welche den Begriff der Äquivalenz benötigen, wie beispielsweise `contains()` und `remove()`, verwenden die `equals()` Methode zur Überprüfung auf Gleichheit.

Die `toArray()` Methode ohne Parameter liefert ein Array von Objekten vom Typ `Object`. Falls Sie andere Datentypen, Objekttypen, benötigen, müssen Sie die andere Form der Methode verwenden. Falls Ihre Collection beispielsweise lediglich `String` Objekte enthält, wäre es innvoller ein `String Array` zu definieren:

```
String[ ] strings = new String[coll.size()];
strings = (String)coll.toArray(strings);
```

Beachten Sie die doppelte Information (`strings` als Rückgabe und als Parameter) und die Erklärung dazu weiter oben.

Wegen obigem Mechanismus könnten Sie dasselbe auch folgendermassen erzwingen:

```
String[ ] strings = (String[ ])coll.toArray(new String[0]);
```

Verschiedene Methoden von `Collection` operieren im Grossen mit anderen Collections. Diese Methoden wurden aus Bequemlichkeit entwickelt. Sie gestatten mächtige Vergleiche, sind aber selten nötig.

`public boolean` **`containsAll(Collection coll)`**
liefert `true`, falls diese Collection jedes der Elemente as `coll` enthält.

`public boolean` **`addAll(Collection coll)`**
fügt alle Elemente der Collection `coll` zur aktuellen Collection hinzu und liefert `true`, falls das Hinzufügen zu einer Änderung der Collection führte (optional).

`public boolean` **`removeAll(Collection coll)`**
entfernt alle Elemente dieser Collection, welche auch in `coll` vorkommen. Falls das Entfernen zu einer Veränderung der Collection führte, wird `true` (optional) zurück geliefert.

`public boolean` **`retainAll(Collection coll)`**
entfernt alle Elemente der Collection, welche nicht in `coll` vorkommen. Falls dieses Entfernen zu einer Veränderung der Collection führt, kann (optional) ein `true` zurück gegeben werden.

`public void` **`clear()`**
löscht alle Elemente der Collection.

Das Collection Interface ist bewusst allgemein gehalten. Jeder Collection Untertyp kann andere Verhaltensmuster oder Einschränkungen betreffend Datentypen verordnen.

Beispielsweise kann eine Collection das `null` Element abweisen oder abzeptieren. Auch die Elemente können unter Umständen auf bestimmte Datentypen beschränkt werden.

Wichtig ist, dass die spezifischen Einschränkungen sauber dokumentiert werden, so dass sofort klar ist, welche Möglichkeiten die konkrete Version der Collection dem Benutzer bietet.

1.5. Set und SortedSet

Das `Set` Interface erweitert Collection und stellt spezifische Kontrakte für die Methoden zur Verfügung, ohne allerdings neue Methoden zu definieren.Eine `Set` Collection enthält keine Elemente mehrfach. Falls Sie versuchen, ein Element ein zweites Mal hinzuzufügen, liefert die Methode das erste Mal den Wert `true`, das zweite Mal den Wert `false` (es wurde kein Element hinzugefügt: das Set wurde nicht verändert).

Entsprechend verhält es sich mit beim Entfernen eines Elements.

Jedes Set enthält maximal ein `null` Element, logischerweise, da jedes Element maximal einmal vorkommt.

Das `SortedSet` Interface erweitert `Set` um einen zusätzlichen Kontrakt - die Iteration auf einer sortierten Menge liefert die Elemente immer in einer bestimmten (Sortier-) Reihenfolge. Diese entspricht der natürlichen Reihenfolge der Elemente. In der Implementation von `SortedSet` in `java.util` können Sie auch noch ein `Comparator` Objekt angeben, um die natürliche Reihenfolge festzulegen.

Die Ordnung definiert eine Zusatzanforderung an die Elemente des Sets - die Elemente müssen in einer bestimmten Reihenfolge zueinander stehen. Jedes Einfügen eines Elements muss diese Reihenfolge beachten und Elemente entsprechend neu anordnen. Falls Sie versuchen unterschiedliche Objekttypen in ein `SortedSet` einzufügen, führt dies zu einer `ClassCastException`.

`SortedSet` definieren zusätzliche Methoden, welche für diese Datenstrukturen Sinn machen:

`public Comparator` **`comparator()`**
: liefert den Comparator, welcher zum Sortieren benutzt wird oder `null`, falls die natürliche Ordnung benutzt wird.

`public Object` **`first()`**
: liefert das erste (tiefste) Objekt dieses Sets.

`public Object` **`last()`**
: liefert das letzte (höchste) Element in diesem Set.

`public SortedSet` **`subSet(Object min, Object max)`**
: liefert eine Sicht auf die Elemente der Collection, welche zwischen `min` und `max` liegen. Falls `min > max` ist, oder dieses Set selber eine Sicht auf ein anderes Set ist

und `min` oder `max` ausserhalb dieser *anderen* Sicht liegen, dann wird eine `IllegalArgumentException` geworfen. Wie üblich würde dann eine `IllegalArgumentException` geworfen, falls Sie versuchen würden, ein Element zu modifizieren, welches ausserhalb dieses Bereiches liegt.

`public SortedSet` **`headSet(Object max)`**
liefert eine Sicht auf die Menge, welche alle Elemente enthält, welche einen Wert kleiner als `max` sind.

`public SortedSet` **`tailSet(Object min)`**
liefert eine Sicht auf die Menge, welche alle Elemente enthält mit Werten grösser als `min`.

Die Bezeichnung *Sicht* ist wesentlich in folgendem Sinne:
falls Sie auf die Teilmengen zugreifen, die durch obige Sichten definiert werden, greifen Sie in Wahrheit auf die *ursprünglichen* Elemente der zugrundeliegenden Menge zu. Sollte sich die urprüngliche Menge ändern, ändert sich auch das Ergebnis der Sicht! Mit andern Worten: es wird kein Snapshot erstellt. Sie sehen durch ein Fenster auf die Urdaten!

Aber mit den obigen Methoden können Sie leicht auch Snapshot Methoden selber definieren. Hier ein Beispiel für das Kopieren des Kopfes einer sortierten Menge:

```
public SortedSet copyHead(SortedSet set, Object max) {
      SortedSet head = set.headSet(max);
      return new TreeSet(head); // initialisiert mit dem Head
}
```

Dieses Beispiel verwendet einen sogenannten `copy`-Konstruktor, mit dem eine neue Collection kreiert werden kann, deren Elemente die Elemente einer Collection sind, welche als Parameter dem Konstruktor mitgegeben werden.

Beispiele für Implementationen von `Set` und `SortedSet` sind die in `java.util` vorhandenen Klassen `HasSet` und `TreeSet`.

1.5.1. HashSet

`HashSet` ist ein `Set`, welches mit Hilfe einer Hash-Tabelle implementiert wird. Das Modifizieren des Inhalts eines `HashSet` oder das Testen, ob ein Element im `HashSet` enthalten ist, sind sogn. *constant-time* Operationen, dh. sie sind unabhängig von der Grösse des Sets, sofern die Hashfunktion korrekt implementiert wurde.

`HashSet` Collections bieten folgende zusätzliche Methoden an:

`public` **`HashSet(int initialCapacity, float loadFactor)`**
kreiert ein neues `HashSet` mit `initialCapacity` Hash Buckets und einem vorgegeben `loadFactor`, einer positiven Zahl. Falls die durchschnittliche Anzahl Elemente im Set bezogen auf die Anzahl Buckets grösser oder gleich dem `loadfactor` ist, wird die Anzahl Buckets vergrössert.

`public` **`HashSet(int initialCapacity)`**
kreiert ein neues HashSet mit einer initialen Kapazität und einem standard Loadfaktor.

`public` **`HashSet()`**

kreiert ein neues `HashSet` mit einer Standard-Kapazität und einem Standard-Loadfaktor.

`public` **`HashSet(Collection coll)`**

liefert ein neues `HashSet`, dessen initialer Inhalt die Elemente der Collection `coll` sind. Die initiale Kapazität basiert auf der Anzahl Elemente der Collection `coll`. Der Standard-Loadfaktor wird verwendet.

1.5.2. TreeSet

Falls Sie ein `SortedSet` benötigen, können Sie beispielsweise ein `TreeSet` verwenden. Dieses speichert seine Elemente in einer Baumstruktur, welche balanciert bleibt, deren Äste also möglichst untereinander gleich lang bleiben. Dadurch beträgt die Durchschnittszeit, um den Baum zu durchsuchen oder zu modifizieren $O(\log n)$, das heisst, dass der Zeitbedarf grob mit dem Logorithmus der Anzahl Elemente zunimmt.

Die Klasse `TreeSet` besitzt folgende Konstruktoren:

`public` **`TreeSet()`**

liefert ein neues `TreeSet`, welches gemäss der natürlichen Ordnung der Elemente sortiert wird. Alle Elemente, welche hinzugefügt werden, müssen das `Comparable` Interface implementieren und jeweils gegenseitig vergleichbar sein, da sie sonst nicht sortiert werden können.

`public` **`TreeSet(Collection coll)`**

ist äquivalent zur Konstruktion eines `TreeSet` und dem anschliessenden Hinzufügen der Elemente der Collection `coll`.

`public` **`TreeSet(Comparator comp)`**

kreiert ein neues `TreeSet`, dessen Elemente gemäss dem Comparator `comp` soriert werden.

`public` **`TreeSet(SortedSet set)`**

kreiert ein neues `TreeSet`, dessen Anfangselemente die selben sind, wie im `SortedSet set` und dessen Sortierreihenfolge ebenfalls vom `SortedSet set` übernommen wird.

1.6. List

Das `List` *Interface* erweitert Collection und definiert eine Collection, deren Elemente eine definierte Ordnung besitzen - jedes Element existiert in einer bestimmten Position in der Collection, indexiert von 0 bis `list.size()`. Dadurch müssen verschiedene Kontrakte aus den übergeordneten Interfaces modifiziert werden. Beispielsweise muss beim Hinzufügen eines Elements dieses am Ende angefügrt werden; falls Sie das n-te Element entfernen, wird das n+1-te zum n-ten und alle Elemente darüber werden um eine Position verschoben. Der `java.util.Vector` ist eine *legacy* Struktur, welche das selbe Verhalten zeigt.

Die Klasse `List` definiert auch einige neue Methoden, welche für geordnete Collections Sinn machen:

`public` **`boolean get(int index)`**
liefert das `index`-te Element in der Liste.

`public boolean` **`set(int index, Object elem)`**
setzt das Element `elem` auf die `index`-te Position in der Liste und ersetzt dabei ein bereits vorhandenes Element.

`public boolean` **`add(int index, Object elem)`**
fügt das Element `elem` an der Stelle `index` in die Liste ein. Falls dort ein Element vorhanden ist, werden alle Elemente ab dort um eine Position verschoben (optional).

`public boolean` **`remove(int index)`**
entfernt das `index`-te Element aus der Liste. Optional werden die restlichen Elemente um eine, die geleerte Position, verschoben.

`public int` **`indexOf(Object elem)`**
liefert den Index des ersten Objekts in der Liste, welches gleich ist wie `elem` (oder `null` falls `elem null` ist). Falls kein Element gefunden wird, wird -1 zurückgeliefert.

`public int` **`lastIndexOf(Object elem)`**
liefert den Index des letzten Objekts in der Liste, welches gleich ist, wie `elem` (oder `null` falls das `elem null` ist). Falls kein passendes Element gefunden wird, wird - 1 zurückgegeben.

`public List` **`subList(int min, int max)`**
liefert eine Liste, als Sicht auf die darunterliegende Liste. Diese Sicht zeigt lediglich die Elemente im Bereich `min` bis `max`. Die resultierende Liste `List` ist eine Sicht. Das heisst also, dass jede Änderung in der ursprünglichen Liste auch in der Liste `List` sichtbar ist.

`public ListIterator` **`listIterator(int index)`**
liefert ein `ListIterator` Objekt, welches durch die Elemente der Liste iterieren kann, ab dem `index`-ten Element.

`public ListIterator` **`listIterator()`**
liefert ein `ListIterator` Objekt, mit dessen Hilfe durch die gesamte Liste gewandert werden kann.

Alle Methoden, welche eine Indexvariable als Parameter verwenden, können eine `IndexOutOfBoundsException` werfen, falls der Wert des Indices ausserhalb des erlaubten Bereiches liegt (Null bis Listgrösse-1).

Das java.util Package stellt zwei Implementationen der `List` Collection zur Verfügung - `ArrayList` und `LinkedList`.

1.6.1. ArrayList

`ArrayList` ist eine gute Basislistenimplementation, welche die Elemente in einem Array abspeichert. Das Hinzufügen oder Entfernen eines Elements am Ende der Liste ist sehr einfach und verbraucht $O(1)$ Zeit, wächst also linear mit der Grösse der Liste; anders gesagt: bei grossen Listen ist die Methode nicht gerade effizient (im Vergleich zu $O(\log n)$ Versionen). Auch das Herauslesen eines Elements verbraucht $O(1)$ Zeit. Falls sich das Element in der Mitte der Liste befindet, wird das Ganze wegen dem Reorganisieren noch langsamer : $O(n-i)$, wobei n die Grösse der Liste ist und i die Position des Elements, welches Sie entfernen wollen.

Eine Array Liste besitzt eine bestimmte Kapazität, einfach die Anzahl Elemente, welche sie aufnehmen kann, ohne mehr Speicher zu verlangen. Neue Elemente werden einfach im Array abgespeichert, solange Speicherplätze dafür zur Verfügung stehen. Falls kein Platz mehr vorhanden ist, wird das bestehende Array durch ein neues ersetzt und die Daten kopiert. Daher kann das Setzen einer Kapazität die Performance verbessern, sofern Sie in etwa wissen, wie gross das Array sein wird. Das Ganze ist etwas trickreich: wählen Sie ein viel zu grosses Array, wird die Manipulation eher langsam, weil Sie Speicherplatz benötigen, aber nie benutzen; ist die Kapazität zu klein, muss umkopiert werden.

Die Klasse `ArrayList` besitzt drei Konstruktoren:

`public` **`ArrayList()`**
: kreiert ein neues `ArrayList` Objekt mit einer Standardkapazität

`public` **`ArrayList(int initalCapacity)`**
: kreiert ein neues `ArrayList` Objekt, mit einer Kapazität `initialCapacity`. Das Array besitzt am Anfang die Speicherplatz gemäss `initialCapacity`.

`public` **`ArrayList(Collection coll)`**
: kreiert ein `ArrayList` Objekt, dessen Inhalt den Elementen der Collection `coll` entspricht. Die Kapazität des Arrays wird am Anfang auf 110% der Collection gesetzt, damit noch etwas Reserve vorhanden ist. Auch die Ordnung der Elemente ist durch die Collection vorgegeben.

Die Klasse `ArrayList` definiert auch zwei weitere Methoden:

`public void` **`trimToSize()`**
: setzt die Kapazität des Arrays genau so gross, wie unbedingt nötig. Sie können damit ein eventuell zu gross angelegtes Array auf die optimale Grösse / Länge reduzieren, wobei unter Umständen ein neues kleineres Array angelegt wird.

`public void` **`ensureCapacity(int minCapacity)`**
: setzt die Kapazität auf den garantierten Wert `minCapacity,` auch wenn das aktuelle Array eventuell kürzer ist. Die Methode können Sie beispielsweise dann einsetzen, wenn Sie eine bestimmte Anzahl Elemente in die Liste einfügen müssen und im voraus wissen wieviel Platz Sie dafür benötigen, da Sie dadurch vermeiden können, dass das System dauernd das Array herumkopieren und vergrössern muss.

1.6.2. LinkedList

Eine `LinkedList` ist eine doppelt-verbundene Liste, deren Performance Charakteristiken sich völlig anders verhält als die Array Liste. Falls Sie ein Element am Ende anfügen wollen, kostet Sie dies *O*(l) Operationen. Das Hinzufügen oder Entfernen eines Elements in der Mitte kostet Sie *O*(l) Operationen, weil Sie nichts kopieren müssen. Ein Element aus der Liste lesen kostet Sie *O*(i), wobei i die Position des gesuchten Elements ist.

`LinkedList` stellt zwei Konstruktoren zur Verfügung, welche die typischen Charaktersitiken doppelt verbundener Listen berücksichtigen:

`public` **`LinkedList()`**
: kreiert eine neue verbundene Liste.

`public` **`LinkedList(Collection coll)`**
: kreiert eine neue `LinkedList`, deren Elemente mit denen aus der Collection `coll` übereinstimmen. Auch die Ordnung der Elemente wird aus der Collection übernommen.

`public Object` **`getFirst()`**
: liefert das erste Element der Liste.

`public Object` **`getLast()`**
: liefert das letzte Element in der Liste.

`publc Object` **`removeFirst()`**
: entfernt das erste Element in dieser Liste.

`public Object` **`removeLast()`**
: entfernt das letzte/hinterste Element aus der Liste.

`public Object` **`addFirst(Object elem)`**
: fügt ein Element am Anfang der Liste hinzu.

`public Object` **`addLast(Object elem)`**
: fügt ein Element am Ende der Liste hinzu.

Eine `LinkedList` ist eine gute Basis für die Definition einer eigenen Warteschlange oder von anderen Listen, bei denen die meisten Operationen am Ende der Liste geschehen.

Falls Sie einen eigenen Stack definieren wollen, dann ist die Array Liste dafür besser geeignet. Array Listen können Sie auch besser durchsuchen als andere Listen, da Sie dafür keinen Iterator benötigen.

Als Anwendungsbeispiel für Listen hier ein einfaches Beispiel, eine Polygon Klasse:

```
import java.util.List;
import java.util.ArrayList;

public class Polygon {
      private List vertices = new ArrayList();

      public void add(Point p) {
            vertices.add(p);
      }

      public void remove(Point p) {
            vertices.remove(p);
      }

      public int numVertices() {
            return vertices.size();
      }

      // weitere Methoden

}
```

Beachten Sie, dass wir `vertices` als Liste `List` definieren und ein `ArrayList` Objekt kreieren. Grund dafür ist, dass Sie eine Variable so abstrakt wie möglich definieren sollten. In unserem Fall ist die Liste `List` abstrakter als die konkrete Implementation `ArrayList`. Der Vorteil ist beispielsweise der, dass Sie jederzeit die Implementation von `ArrayList` auf `LinkedList` verändern könnten, ohne dass Sie wesentliche Änderungen machen müssten, lediglich eine einzige Zeile müssten Sie ändern!

Selbsttestaufgabe 1

Schreiben Sie ein Programm, welches eine Datei öffnet und eine Zeile nach der andern liest und in eine Liste sortiert einträgt. verwenden Sie `String.compareTo` als Vergleichsoperator.

Versuchen Sie eine weitere Lösung des obigen Problems, indem Sie die Zeilen in eine Collection eintragen, beispielsweise ein `Set` und daraus eine sortierte Collection herstellen. Werden die Datensätze automatisch sortiert? Geben Sie die sortierte Collection aus.

1.7. Map und SortedMap

Das Interface `Map` gehört zu den Collections obschon es das `Collection` Interface nicht erweitert. Der Hauptgrund dafür ist, dass der Kontrakt einer Map Klasse wesentlich unterschiedlich von einem Kontrakt einer normalen Collection Klasse ist: einer Map Klasse fügen Sie keine einzelnen Elemente hinzu, sondern Schlüssel / Wert Paare. Ein Map gestattet Ihnen den Wert zu einem bestimmten Schlüssel abzufragen. Pro Schlüssel gibt es einen oder keinen Wert. Als Beispiel können Sie sich eine Namenskartei vorstellen. Falls Ihr Name in der Kartei vorkommt, werden Sie die Adresse als Wert erhalten. Sonst erhalten Sie keine Antwort. Es kann auch sein, dass zu mehreren Namen ein und die selbe Adresse gehört, beispielsweise könnten mehrere Personen an der selben Adresse wohnen.

Die grundlegenden Methoden des Map Interfaces sind:

`public int` **`size()`**
: liefert die Grösse dieses Maps, also die Anzahl Schlüssel/Werte Paare, die im Map enthalten sind. Die Anzahl Werte ist maximal Integer.MAX_VALUE, selbst falls die Map Collection mehr Elemente enthält.

`public boolean` **`isEmpty()`**
: liefert `true`, falls diese Collection keine Mappings enthält.

`public boolean` **`containsKey(Object key)`**
: liefert `true`, falls die Collection ein Mapping zum Schlüssel `key` enthält.

`public boolean` **`containsValue(Object value)`**
: liefert `true`, falls die Collection mindestens ein Mapping zum gegebenen Wert `value` enthält.

`public Object` **`get(Object key)`**
: liefert das Objekt zum Schlüssel `key` oder `null`, falls kein entsprechendes Mapping existiert. Falls `null` als Key und als Wert erlaubt ist, könnte ein `null` Objekt auch `null` als Wert Objekt zurück liefern.

`public Object` **`put(Object key, Object value)`**
: assoziiert den Schlüssel `key` mit einem Wert im Map Objekt. Falls bereits eine Zuordnung `key-value` existiert, wird dieses Mapping verändert und das alte Wert wird zurückgeliefert. Falls kein Mapping existiert, dann liefert die Methode `null` als Objekt zurück. Das kann aber auch heissen, das der ursprüngliche Wert zum Schlüssel das `null` Objekt war.

`public Object` **`remove(Object key)`**
: entfernt alle Mappings für diesen Schlüssel. Der Rückgabewert befolgt die selbe Semantik wie in der obigen `put()` Methode.

`public void` **`putAll(Map otherMap)`**
: stellt alle Mappings eines Map Objekts in dieses Map Objekt.

`public void` **`clear()`**
: entfernt alle Mappings aus diesem Map Objekt.

Bei diesen Methoden gilt:

- falls die Methoden einen Schlüssel verwenden, kann eine `ClassCastException` geworfen werden, falls der Schlüssel nicht vom korrekten Typ ist.
- falls der Schlüssel das `null` Objekt ist und die Map Klasse das `null` Objekt nicht gestattet, wird eine `NullPointerException` geworfen.

Beachten Sie, dass Map Methoden mit dem selben oder ähnlichen Namen wir Collection Klassen besitzt. Dies hilft als Gedankenstütze.

Zur Effizienz ist folgendes zu sagen:
jede Methode, welche mit Schlüsseln arbeitet, ist effizienter. Beispiel: `containsKey()` ist wesentlich effizienter als `containsValue()`. Schlüssel werden beispielsweise mit Hilfe von Hash-Tabellen verwaltet und entsprechend schnell gefunden. Werte werden in der Regel linear durchsucht werden, ein Wert nach dem andern (sequentiell).

Obschon Maps keine Collections sind, stellt Map Methoden zur Verfügung, mit deren Hilfe Maps als Collections betrachtet werden können:

`public Set` **`keySet()`**
liefert ein Set, welches als Elemente alle Keys dieses Map Objekts enthält.

`public Collection` **`valueSet()`**
liefert ein Set, welches die Werte dieses Map Objekts als Elemente enthält.

`public Set` **`entrySet()`**
liefert ein Set, dessen Elemente `Map.Entry` Objekte sind. Diese stellen die einzelnen Mappings im `Map` Objekt dar. `Map.Entity` ist in inneres Interface mit eigenen Methoden, die wir noch anschauen werden.

Collections (und die Sets oben), welche diesen Maps zugeordnet sind, entsprechen Sichten auf die Maps. Das heisst, dass bei Änderungen im Map auch die Werte in den Sets verändert werden. Eine weitere Spezialität besitzen diese Collections, weil sie Sichten auf das Map Objekt sind: Sie können keine weiteren Elemente in die Sets oder die Collection einfügen! Auch andere Methoden sind nicht anwendbar: immer wenn dadurch die Integrität der Daten (das Fenster auf das Map Objekt) verletzt werden könnte, wird die Methode nicht ausführbar!

Auch beim Iterieren über die Schlüsselmenge und gleichzeitig über der Wertemenge, können Sie nicht sicher sein, dass Sie gültige Schlüssel / Wertepaare erhalten. Wenn Sie dies möchten sollten Sie mit den `EntrySets` arbeiten.

Nun zum inneren Interface `Map.Entry`:

dieses definiert Methoden zum Manipulieren die Einträge in einem Map Objekt

`public Object` **`getKey()`**
liefert den Schlüssel dieser Entry

`public Object` **`getValue()`**
liefert den Wert für diese Entry, für diesen Map Eintrag.

`public Object` **`setValue()`**
setzt den Wert für diesen Eintrag im Map Objekt.

Beachten Sie, dass Sie den Schlüssel nicht neu setzen können. Das kennen Sie sicher aus anderen schlüsselorientierten Konzepten in der Informatik (Primärschlüssel in einer Datenbank lassen sich auch nicht verändern). Falls Sie einen Schlüssel verändern wollen, müssen Sie dessen zugehörigen Wert bestimmen, den Eintrag löschen und einen neuen Eintrag generieren.

`SortedMaps` sind Erweiterungen der Map Klasse, bei denen die Schlüssel sortiert sind. Zusätzlich werden einige neue Methoden für `SortedMaps` definiert:

`public Comparator` **`comparator()`**
liefert den Comparator, mit dessen Hilfe sortiert wird, falls nicht die natürliche Sortier Ordnung verwendet werden kann (in diesem Fall wird das `null` Objekt zurück geliefert).

`public Object` **`firstKey()`**
liefert den ersten (tiefsten) Schlüssel im Map Objekt.

`public Object` **`lastKey()`**
liefert den letzten (höchsten) Schlüssel im Map Objekt.

`public SortedMap` **`subMap(Object minKey, Object maxKey)`**
liefert eine Sicht auf das sortierte Map Objekt, mit dem Fenster von `minKey` bis `maxKey`.

`public SortedMap` **`headMap(Object maxKey)`**
liefert eine Sicht auf jenen Teil des Map Objekts, welches mit dem tiefsten Schlüssel beginnt und bis `maxKey` reicht.

`pblic SortedMap` **`tailMap(Object minKey)`**
liefert eine Sicht auf jenen Teil des Map Objekts, welcher bei `minKey` beginnt und bis zum höchsten Schlüssel reicht.

Auch hier gilt:
alle Maps, welche als Rückgabewerte auftreten, sind Fenster auf die Originaldaten. Ändern sich die Originaldaten, so ändern sich auh die Daten in der Sicht!

Beispiele für die Implementation von `SortedMaps` sind im `java.util` Package: `HashMap`, `TreeMap` und `WeakHashMap`.

1.7.1. HashMap

HashMaps implementieren Maps mittels Hash-Tabellen: die Hashfunktion wird benutzt, um einen Platz in der Hash- Tabelle zu finden. Hash-Tabellen sind recht effizient und vorallem ist die Performance nicht wesentlich von der Grösse der Tabelle abhängig.

Die Konstruktoren der HashMaps sind:

`public` **`HashMap(int initialCapacity, float loadFactor)`**
: kreiert ein neues HashMap Objekt mit einer Intialkapazität von Hash Buckets und einem vorgegebenen Ladefaktor (bis zum Ladefaktor werden die Buckets gefüllt). Beide Zahlen müssen positiv sein.

`public` **`HashMap(int initialCapacity)`**
: kreiert ein neues HashMap Objekt mit einer bestimmten Initialkapazität. Der Ladefaktor wird standardmässig gesetzt.

`public` **`HashMap()`**
: kreiert ein neues HashMap Objekt mit Standard Initialkapazität und Ladefaktor.

`public` **`HashMap(Map map)`**
: kreiert ein neues Hashmap Objekt, dessen initiale Mappings von einem Map Objekt stammen. Initialkapazität wird vom Map Objekt übernommen; der Ladefaktor wird standardmässig angenommen.

1.7.2. TreeMap

Die TreeMap Klasse implementiert `SortedMap`, sorgt also für sortierte Schlüssel. Diese Klasse ist recht ineffizient: die Performance (entfernen, finden, einfügen) geschieht etwa gemäss O(log n). TreeMaps verwendet man lediglich, wenn man die Sortierung wirklich benötigt.

TreeMap besitzt folgende Konstruktoren:

`public` **`TreeMap()`**
: kreiert ein neues TreeMap Objekt, deren Schlüssel gemäss der natürlichen Ordnung sortiert sind. Alle Schlüssel, welche diesem Map hinzugefügt werden, müssen das Comparable Interface implementieren, damit die Elemente gegenseitig verglichen werden können (sortiert).

`public` **`TreeMap(Map map)`**
: äquivalent zu `TreeMap()` und anschliessendem Hinzufügen aller Schlüssel / Wert Paare aus dem Map `map`.

`public` **`TreeMap(Comparator comp)`**
: kreiert ein TreeMap Objekt, welches gemäss dem Comparator Objekt sortiert / geordnet ist.

`public` **`TreeMap(SortedMap map)`**

kreiert ein neues TreeMap Objekt, dessen initialer Inhalt der selbe ist, wie jener im Objekt `map` und dessen Sortierreihenfolge auch aus jenem Objekt übernommen wird.

1.7.3. WeakHashMap

Die Collection Implementationen benutzen alle die (üblichen) starken Referenzen für Elemente, Werte und Schlüssel. Das ist auch das, was Sie üblicherweise verwalten wollen.

Falls Sie aus irgendwelchen Gründen an den sogenannten Weak Referenzen, den schwachen Referenzen, interessiert sind, also an den Objekten, die kurz vor dem Garbage Collector stehen, dann sollten Sie sich für WeakHaskMaps interessieren.

Der Unterschied zu den normalen HashMap Objekten ist lediglich die Art der Elemente: weak Referenzen an Stelle der üblichen Referenzen.

Weak Referenzen werden üblicherweise bei der Besprechung des Garbage Collectors besprochen, da ein Objekt mit einer weak Referenz eine Referenz ist, die vom Garbage Collector nächstens gelöscht wird.

1.8. Wrapped Collections und die Collection*s Klasse*

Die `Collections` Klasse definiert einige statische Hilfsprogramme, welche auf Collection Objekten operieren. Diese Hilfsmethoden können grob in zwei Gruppen eingeteilt werden:

- solche Methoden, welche zu gewrappten Collection Objekten führen
  und
- solche Methoden, welche dies nicht tun!

Gewrappte Collection sind

- synchronisierte Collection Objekte / Klassen
- unmodifizierbare Collection Objekte / Klassen

1.8.1. Der Synchronisations-Wrapper

Alle Collections in java.util sind nicht synchronisiert, ausser den Legacy Klassen aus den ersten Java Releases. Die Synchronisation wurde zum Teile aus Performance Gründen entfernt. Es liegt also an Ihnen die Zugriffe auf Collection Objekte zur synchronisieren!

Eine Möglichkeit besteht darin, dass Sie die Methoden als `synchronized` deklarieren oder algorithmisch für eine Sequenzialisierung sorgen.

Eine andere Möglichkeit besteht darin, den *Synchronisationswrapper* einzusetzen. Diese rufen alle Methoden auf, unter Beachtung aller benötigten Synchronisationen. Den Synchronisationswrapper erhalten Sie, indem Sie eine der statischen Collections Methoden aufrufen:

```
1. synchronizedCollection
2. synchronizedSet
3. synchronizedSortedSet
4. synchronizedList
5. synchronizedMap
6. synchronizedSortedMap
```

Diese Factory Methoden liefern Wrapper, deren Methoden voll synchronisiert sind.

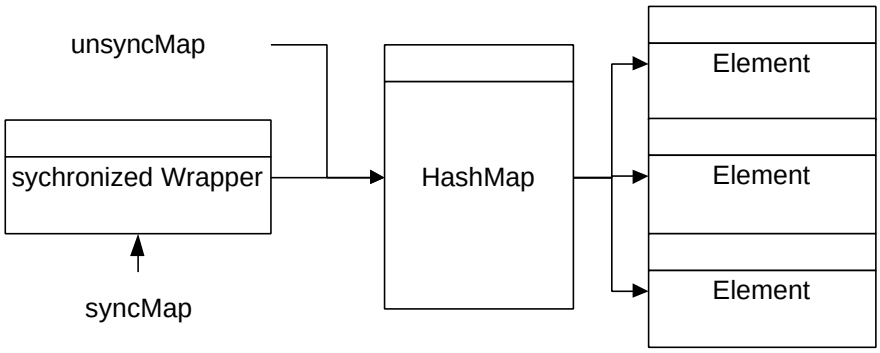
Beispiel:

```
Map unsyncMap = new HashMap();
Map syncMap = Collections.synchronizedMap(unsyncMap);
```

Das erste Map Objekt verwendet keine synchronisierte Methoden; daraus wird mit Hilfe der Factory Methode ein voll synchronisiertes Map Objekt.

Die zwei Objekte sind eigentlich ein einziges Objekt, einfach aus zwei Blickwinkeln betrachtet.

Anschaulich:



Falls Sie viele Elemente in ein Map Objekt einfügen wollen, kann es sinnvoll sein, den Zugriff zentral zu synchronisieren. Schematisch:

```
synchronized(syncMap) {
       for (int i=0; i<keys.length; i++)
            unsync.put(keys[i], values[i]);
}
```

Damit können wir den Synchronisationsoverhead reduzieren!

Falls Ihnen irgend eine Methode einen Iterator zurückliefert, muss auch dieser synchronisiert werden:

```
synchronized(syncMap) {
      System.out.println(" --- map Inhalte: ");
      Iterator it = syncMap.keySet().iterator();
      while (it.hasNext() ) {
            Object key = it.next();
            System.out.println(key + ": "+syncMap.get(key) );
      }
}
```

Falls Sie unsynchronisiert in einem Multithreading System arbeiten, können undefinierte Zustände resultieren.

1.8.2. Der Unmodifiable Wrapper

Die Collections Klasse besitzt auch statische Methoden, welche *unmodifizierbare Wrapper* für Ihre Collections zurückliefern :

```
1. unmodifiableCollection
2. unmodifiableSet
3. unmodifiableSortedSet
4. unmodifiableList
5. unmodifiableMap
6. unmodifiableSortedMap
```

Diese Collection Klassen enthalten unmodifizierbare Methoden in folgendem Sinne: falls eine der modifizierenden Methoden aufgerufen wird, wird dieser Aufruf nicht ausgeführt, sondern eine `UnsupportedOperation` Exception geworfen. Aber das Schema ist dasselbe wie eben: die ursprüngliche Klasse wird nicht berührt! Sie blicken durch ein Fenster auf diese Originalklasse.

Der Inhalt der Klasse kann sich ändern, aber auf den gewrappten Klassen kann man keine Modifikationen oder Einschübe in die darunterliegenden Klasse durchführen.

Wir könnten diese Collection Klassen beispielsweise einsetzen, um unmodifizierbare Sets zu definieren:

```
public final String suits[ ] = { "Hearts", "Clubs", "Diamonds", "Spades" };
```

Damit wird die Referenz final und nicht modifizierbar! Wir dachten aber wohl eher an die Konstanten auf der rechten Seite der Gleichung.

Als Schlitzohr könnten Sie also hingehen und beispielsweis den Inhalt des Arrays verändern:

```
suits[3] = "Armani";
```

Dieses Problem könnten wir mit einem unmodifiable Wrapper korrekter formulieren:

```
private final String suitNames[ ] = { "Hearts", "Clubs", "Diamonds",
"Spades" };

public final List suits =
Collections.unmodifiableList(Array.asList(suitNames));
```

Nun haben wir wahrscheinlich das korrekte Verhalten implementiert:
- als erstes haben wir die Namen in einem privaten Array definiert, so dass kein freier Zugriff möglich ist;
- dann haben wir eine Liste konstruiert, mittels der statischen Collections Methode, welche eine Ausnahme wirft, falls wir Elemente verändern.

1.8.3. Die Collections Hilfsmethoden

Die Collections Klasse (mit s am Schluss!) definiert wichtige Hilfsmethoden. Beispielsweise können Sie mit diesen Methoden das kleinste oder das grösste Element einer Collection finden.

`public static Object` **`min(Collection coll)`**
liefert das kleinste Element der Collection, klein im Sinne der natürlichen Ordnung.

`public static Object` **`min(Collection coll, Comparator comp)`**
liefert das kleinste Element der Collection, klein gemäss der angegebenen Ordnung im Comparator Objekt.

`public static Object` **`max(Collection coll)`**
liefert das grösste Element der Collection, gross im Sinne der natürlichen Ordnung.

`public static Object` **`max(Collection coll, Comparator comp)`**
liefert das grösste Element der Collection, gross im Sinne der Comparator Ordnung.

Wir können auch die Reihenfolge der Elemente umdrehen:

`public static Comparator` **`reverseOrder()`**
liefert einen Comparator, der die natürliche Ordnung der Elemente umkehrt.
Das heisst:

```
Collections.reverseOrder().compare(o1, o2);
// liefert
-o1.compareTo(o2);
```

Verschiedene Methoden arbeiten mit Listen:

`public static void` **`reverse(List list)`**
dreht die Ordnung der Elemente in der Liste um.

`public static void` **`shuffle(List list)`**
mischt die Elemente in der Liste.

`public static void` **`shuffle(List list, Random randomSource)`**
mischt die Liste mit Hilfe von Zufallszahlen, die mit der `randomSource` produziert werden.

`public static void` **`fill(List list, Object elem)`**
ersetzt jedes Element der Liste durch das Element `elem`.

`public static void` **`copy(List dst, List src)`**
kopiert jedes Element der `src`-Liste in die `dst`-Liste. Falls dst zu klein ist, um alle Elemente aufzunehmen, wird eine `IndexOutOfBoundsException` geworfen. Sie können auch Sublisten aufeinander kopieren.

`public static List` **`nCopies(int n, Object elem)`**
liefert eine unveränderliche Liste mit n Elementen, welche alle gleich dem Element `elem` sind. Dieses Konstrukt gestattet eine Speicheroptimierung: intern wird das Element nur einmal und zusätzlich der Wiederholungsfaktor gespeichert.

Zusätzlich existieren Methoden, welche das *Singleton* Design Pattern implementieren, also genau *eine* Instanzierung einer Klasse gestatten. Die folgenden Collections enthalten genau ein Element:
`public static Set` **`singleton(Object elem)`**
liefert ein nicht mutierbares Set, welches genau ein Element, `elem`, enthält.

`public static List` **`singletonList(Object elem)`**
liefert eine nicht mutierbare Liste, welche genau ein Element, `elem`, enthält.

`public static Map` **`singletonMap(Object key, Object value)`**
liefert ein nicht mutierbares Map, welches genau einen Eintrag enthält.

Zusätzlich existieren verschiedene Methoden zum Sortieren:
`public static void` **`sort(List list)`**
sortiert die Liste gemäss der natürlichen Ordnung.

`public static void` **`sort(List list, Comparator comp)`**
sortiert die Liste gemäss dem Comparator Objekt.

Und schliesslich bietet Collections auch noch Suchmethoden:

`public static int` **`binarySearch(List list, Object key)`**
diese Methode verwendet einen binären Suchalgorithmus, um den Schlüssel `key` in der Liste `list` zu finden. Die Liste muss gemäss der natürlichen Sortierreihenfolge angeordnet sein. int ist die Position, falls das Element gefunden wird, sonst wird eine negative Zahl zurück geliefert.

`public static int` **`binarySearch(List list, Object key, Comparator comp)`**
diese Methode entspricht der obigen mit Ausnahme der Reihenfolge, die in diesem Fall durch das Comparator Objekt gegeben ist.

Falls Sie eine der Such- oder Sortiermethoden aufrufen, aber eine Liste verwenden, welche nicht sortierbar ist, wird eine `ClassCastException` geworfen.

Hier ein einfaches Beispiel:
Sie möchten eine Liste mit 100 Elementen mit dem gleichen Wert -1 initialisieren.

```
Integer init = new Integer(-1);
List values = new ArrayList( (Collections.nCopies(100, init) );
```

Zum Schluss noch ein paar Konstanten der Collections Klasse, für den Fall, dass Sie leere Collection Objekte benötigen:
1. EMPTY_LIST
2. EMPTY_SET
3. EMPTY_MAP

1.9. Die Arrays *Hilfsklasse*

Die Klasse `Arrays` stellt nützliche statische Methoden für die Arbeit mit Arrays zur Verfügung. Die meisten dieser Methoden sind mehrfach überladen:

je eine Methode für Arrays bestehend aus den verschiedenen Basisdatentypen.(ausser `boolean` betreffend suchen und sortieren) und eine version für generelle `Object` Arrays.

Zudem existieren häufig zwei Versionen einer Methode:
je eine für die Maipulation eines gesamten Arrays bzw. Subarrays.

Die Methoden im Einzelnen:

- `sort` sortiert ein Array mit Hilfe einer Technik, die ein O(n log n) Verhalten hat.
- `binarySearch` durchsucht ein sortiertes Array, um einen bestimmten Schlüssel zu finden. Es gibt keine SubArray Version dieser Methode.
- `equals` liefert true, falls zwei Arrays das selbe Objekt repräsentieren. Es gibt keine SubArray Version dieser Methode.
- `fill` füllt ein Array mit spezifizierten Werten.

Die `sort()` und `binarySearch()` Methoden existieren auch in zwei überladenen Versionen für generelle Objekte:

1) in der ersten Version muss vorausgesetzt werden, dass die Elemente (Objekte) vergleichbar sind, also ein Comparable Interface implementiert wird.
2) in der zweiten Version muss ein Comparator Objekt mitgeliefert werden, mit dessen Hilfe verglichen wird.

Ein Array bestehend aus Objekten kann man auch als Liste ansehen. Dieses Kreieren einer Listensicht geschieht mit Hilfe der Methode `asList()` Methode. Diese Methode liefert eine Sicht auf, also keine Kopie des Arrays. Sie können demnach auch keine Elemente aus der Liste entfernen oder neue hinzufügen.

1.10. Schreiben von Iterator Implementationen

Iteratoren sind sehr hilfreiche Konstrukte: sie liefern Ihnen beispielsweise jeweils das nächste Element aus einer Liste. Wie würde ein selbstgeschriebener Iterator aussehen?

Hier ein mögliches Gerüst für einen eigenen Iterator:

```
public class ShortStrings implements Iterator {
      private Iterator strings;     //Quelle für Strings
      private String nextShort;     // null falls es kein next gibt
      private final int maxLen;     //

      public ShortStrings(Iterator strings, int maxLen) {
            this.strings = strings;
            this.mexLen = maxLen;
            nextShort = null;
      }

      public boolean hasNext() {
            if (nextShort != null)  // gefunden
                  terurn true;
            while (strings.hasNext() ) {
                  nextShort = (String)strings.next();
                  if (nextShort.length() <> maxLen)
                        return true;
            }
      }

      public Object next() {
            if (nextShort == null && !hasNext() )
                  throw new NoSuchElementException();
            String n = nextShort;   // zwischenspeichern
            nextShort = null;
            return n;
      }

      public void remove() {
            throw new UnsupportedOperationException();
      }
}
```

Um Ihnen eine Idee zu geben:
die Klasse `ShortStrings` ist eine Art Iterator, der String Objekte liest, hier mittels eines andern Iterators. Dieser Iterator liefert nur jene Strings zurück, welche eine bestimmte Länge nicht überschreiten. .

Der Konstruktor akzeptiert einen Iterator, der die Zeichenketten liefern wird und die maximale Länge.

Das Datenfeld `nextShort` speichert die nächste kurze Zeichenkette oder `null`, falls es keine weitere Zeichenketten mehr gibt. Falls `next null` ist, sucht die `hasNext` Methode die nächste Zeichenkette und speichert sie in `nextShort`. Falls sie an das Ende gelangt, das Ende gemäss dem andern Iterator, dann liefert sie `false`.

Die Methode `next` überprüft ob es eine nächste kurze Zeichenkette überhaupt gibt und liefert diese Zeichenkette oder wirft eine `NoSuchElementException`, falls kein Element gefunden wurde. Die Methode `hasNext` macht die ganze Arbeit: das Suchen der kurzen Zeichenketten; die Methode `next` liefert einfach das Ergebnis ab. Falls das nächste Element noch gesucht werden soll, wird `nextShort` auf null gesetzt.

Die Methode `remove()` wird nicht unterstützt durch diese Iterator Implementation, wirft also die `UnsupportedOperationException`.

Einige Bemerkungen:

`hasNext` ist so geschrieben, dass ein mehrfacher Methodenaufruf nicht zum Absturz führt.

`next` ist so geschrieben, dass es auch dann funktioniert, wenn `hasNext` noch nie aufgerufen wurde.

`remove` wird nicht unterstützt, weil diese Operation in diesem Beispiel kaum Sinn macht: die Sequenz

```
it.next();
it.hasNext();
it.remove();
```

könnte zu einem undefinierten Zustand führen

Selbsttestaufgabe 2

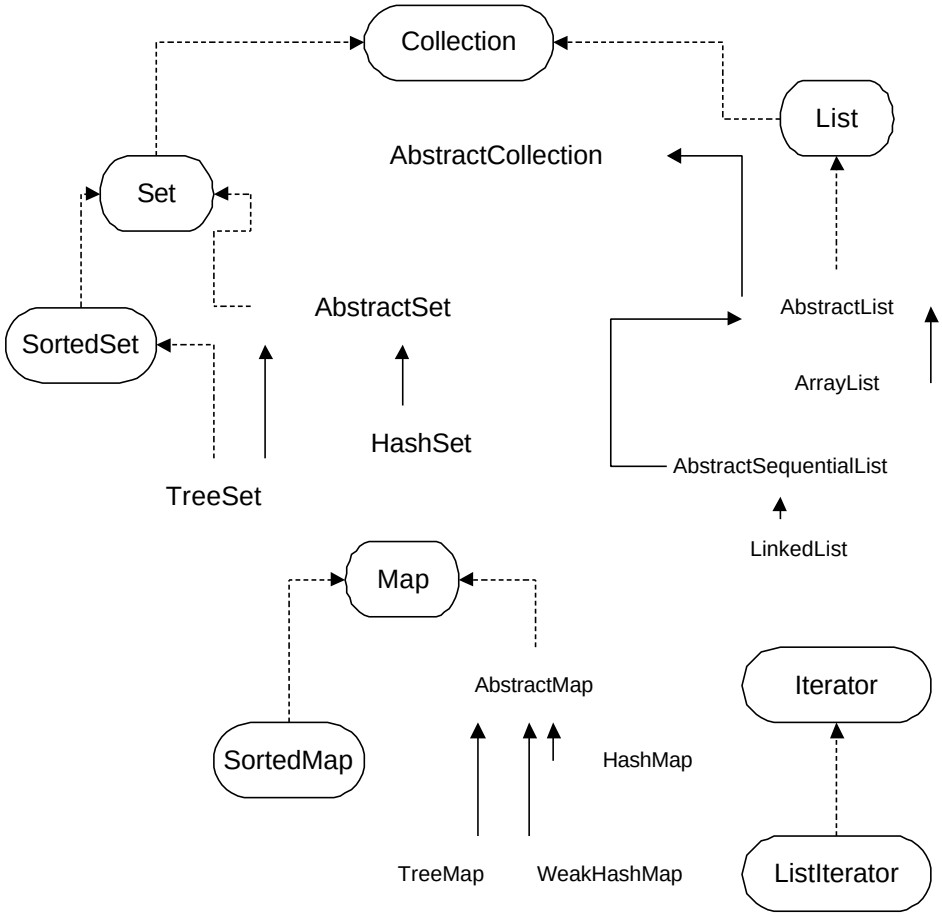
Schreiben Sie ein Programm, welches wie oben die Klasse `ShortStrings` implementiert aber `ListIterator` verwendet. Können Sie die Klasse einfach erweitern oder müssen Sie sie abändern?

1.11. Schreiben von Collection Implementationen

In der Regel werden Sie mit den implementierten Interfaces auskommen oder andere Bibliotheken dazu kaufen und einsetzen. Falls nicht, dann könnten Sie eines der Interfaces implementieren.

In den abstrakten Klassen:
- `AbstractCollection`
- `AbstractSet`
- `AbstractList`
- `AbstractSequentialList`
- `AbstractMap`

finden Sie Skelete für solche Klassen. Diese erleichtern Ihnen die Arbeit statt von einer Interface Definition starten zu müssen. Hier die schematische Liste der Interfaces, abstrakten und Inplementations- Klassen.



Die abstrakten Collection Klassen wurden entworfen, um Ihnen einfache Ausgangspunkte für eigene Implementation zu liefern.Sie können aber auch direkt auf die Interfaces zurück greifen.

Zu jeder abstrakten Collection wurden mehrere Methoden definiert, die in den Standardimplementationen auch eingesetzt werden. Zum Beispiel besitzt die `AbstractList` zwei abstrakte Methoden: `size()` und `get()`. alle anderen Methoden der `AbstractList` werden mit deren Hilfe implementiert, inklusive der Iteratoren. Sie brauchen lediglich eigene Implementationen der anderen Methoden schreiben wenn Sie dies unbedingt wollen.

Die Wurzel der abstrakten Klassen ist `AbstractCollection`. Falls Sie eine Collection Klasse benötigen, welche weder eine Liste, noch ein Map ist, dann sollten Sie direkt diese abstrakte Klasse implementieren. Falls Sie eine Liste oder sonst eine Klasse schreiben wollen, welche dichter an einer der anderen abstrakten Klassen ist, dann sollten Sie versuchen die entsprechenden Unterklassen direkt einzusetzen.

Falls die Collection, die Sie entwickeln wollen, nicht modifiziert werden darf, müssen Sie dafür sorgen, dass die Methoden die `UnsupportedOperationException` wirft. Falls Sie beispielsweise eine unmodifizierbare `AbstractCollection` implementieren wollen, müssen Sie einen Iterator für Ihre Collection schreiben. Falls Ihre Implementation modifizierbare Collections erzeugen soll, müssen Sie auch die `add()` Methode implementieren und Ihr Iterator muss auch noch die `remove()` Methode unterstützen.

`AbstractSet` erweitert `AbstractCollection` und Ihre Methoden können deren Methoden überschreiben.

`AbstractList` verlangt, dass Sie `size()` und `get(int)` implementieren. Dies reicht, falls die Klasse unmodifizierbar sein soll. Falls Sie auch noch `set(int, Object)` implementieren, erhalten Sie eine modifizierbare Liste, aber eine Liste, deren Grösse noch unveränderlich ist. Falls Sie zudem die Methoden `add(int, Object)` und `remove(int)` implementieren, wird Ihre Liste auch noch modifizierbar.

Schauen wir uns ein Beispiel an:

```
public class ArrayBunchList extends AbstractList {
      private Object[ ] arrays;
      private int size;

      public ArrayBunchList(Object[ ][ ] arrays) {
            this.arrays = (Object[][])arrays.clone();
            int s=0;
            for (int i=0; i< arrays.length; i++) {
                  s += arrays[i].length;
            size = s;
      }

      public int size() {
            return size;
      }
```

```
	public Object get(int index) {
		int off=0;  // offset vom Beginn der Collection
		for (int i=0; i < arrays.length; i++) {
			if (index < off + arrays[i].length)
				return arrays[i][jindex-off];
			off += arrays[i].length;
		}
		throws new ArrayOutOfBoundsException(index);
	}

	public Object set(int index, Object value) {
		int off = 0;
		for (int i=0; i < arrays.length; i++) {
			if(index < off + arrays[i].length) {
				Object ret = arrays[i][index-off];
				arrays[i][index-off] = value;
				return ret;
			}
			off += arrays[i].length;
		}
		throw new ArrayIndexOutOfBoundsException(index);$
	}
}
```

Wenn ein `ArrayBunchList` Objekt kreiert wird, werden alle Arrays, aus den die Collection aufgebaut wird, intern unter `arrays` abgespeichert; die gesamte Grösse der Collection in `size`. `ArrayBunchList` implementiert `size, get` und `set`, aber kein `add` oder `remove`. Das heisst, dass diese Klasse eine modifizierbare Liste liefert, aber eine, deren Länge nicht verändert werden kann (`size`). Alle Zugriffe auf die darunterliegenden Daten geschehen mittels der `get()` Methode. Alle Änderungen werden mit Hilfe der `set()` Methode durchgeführt.

`AbstractList` stellt Iterator und ListIterator Implementationen für Sie zur Verfügung. Diese Implementationen verwenden Methoden der darunterliegenden Klasse `AbstractList`.

Der Iterator implementiert eine `AbstractList`, um die Werte zu lesen. Die Klasse `ArrayBunchList` besitzt eine `get()` Methode, mit deren Hilfe weitere Werte im darunterliegenden Array gespiechert werden.

Die Iterator Implementationen der `AbstractList` Klasse verwendet `get()` , um die Werte zu lesen. `ArrayBunchList` besitzt eine `get()` Methode, um die Werte im darunterliegenden Array zu manipulieren.

Ein optimierter Iterator könnte folgendermassen aussehen:

```
private class ABLIterator implements Iterator {
	private int off;
	private int array;
	private int pos;

	ABLIterator() {
		off = 0;
		array = 0;
		pos = 0;
```

```
            for (array = 0; array< arrays.length; array++)
                  if (arrays[array].length > 0)
                        break;

      }

      public boolean hasNext() {
            return off + pos < size();
      }

      public Object next() {
            if (!hasNext() )
                  throw new NoSuchElementException();
            Object ret = arrays[array][pos++];

            // zum nächsten Element
            while(pos>= arrays[array].length) {
                  off += arrays[array++].length;
                  pos = 0;
                  if (array >= arrays.length)
                        break;
            }
            return ret;
      }

      public void remove() {
            throw new UnsuppoertedOperationException();
      }
}
```

Diese Implementation benutzt zusätzliches Wissen über die darunterliegende Datenstruktur, in unserem Fall ein Array. Daher ist die Implementation effizienter, als im allgemeinen Fall.

Man kann oft die Performance drastisch verbessern, falls Sie eine veränderliche Liste haben und die `removeRange()` Methode überschreiben. Diese Methode verwendet zwei `int` Parameter, `min` und `max`, und entfernt alle Elemente zwischen `min` und `max`. Die `clear()` Methode verwendet `remove...()`. Die Standardversion ruft `remove()` einmal pro Element auf. Aber in vielen Fällen kann die darunterliegende Datenstruktur bereits diese Funktion übernehmen. Damit lässt sich diese Funktion wesentlich effizienter implementieren.

`AbstractSequentialList` erweitert `AbstractList`, um Ihnen die Implementation von sequentiellen Listen zu erleichtern. Eine `LinkedList` ist eine sequentielle Liste. Sie sehen das auch daran, dass `ArrayList AbstractList` erweitert, aber `LinkedList` die `AbstractSequentialList` erweitert.

Bei Maps sieht das Ganze komplexer aus, weil Sie Entries konstruieren müssen.

Selbsttestaufgabe 3

Implementieren Sie einen ListIterator für die ArrayBunchList, er möglichst effizienter ist, als im obigen Beispiel.

1.12. Die Legacy Collection Types

Das Collection Framework - Interfaces und Klassen - sind relativ neu im Java, wenigstens die hier besprochenen. Früher besahs Java in java.util bereits einige Klassen, die auch unter der Bezeichnung Collection zusammengefasst waren, aber mit dem hier definierten Framework wenig zu tun hatten.

Diese Datentypen sind weiterhin einsetzbar und werden in vielen Standard Java Packaes eingesetzt. Daher sollten Sie einiges über dies Klassen wissen, ohne jedoch das neue Framework zu ignorieren. Diese alten / legacy Collection Klassen waren:

- `Enumeration` - analog zu `Iterator`
- `Vector` - analog zu `ArrayList`
- `Stack` - eine Unterklasse der Klasse `Vector` (mit `pop()` und `push()`)
- `Dictionary` - analog zu `Map` (`Dictionary` ist abstrakt)
- `Hashtable` - analog zu `HashMap`
- `Properties` - Unterklasse von `Hashtable` (<Schlüssel, Wert> Paare)

`Properties` sind speziell wichtig, weil sie auch systemseitig eingesetzt werden. Deswegen besprechen wir diesen Datentyp noch genauer.

1.12.1. Enumeration

`Enumeration` ist analog zu einem Iterator, besitzt aber genau zwei Methoden:
1. `hasMoreElements()` , welche sich analog zu `hasNext()` und
2. `nextElement()`, welches sich analog zu `next()` verhält.

Falls Sie eine `Enumeration` benötigen, können Sie diese auch aus einer Collection konstruieren, mittels der statischen Methode: `Collections.enumeration`.

Selbsttestaufgabe 4

Schreiben Sie ein Programm, welches eine `Enumeration` aus einer `Collection` gewinnt und verwenden Sie diese `Enumeration`, um die Elemente der `Collection` abzufragen.

1.12.2. Vector

Die `Vector` Klasse ist analog zu `ArrayList`. Obschon `Vector` eine Legacy Klasse ist, wurde die Klasse so definiert, dass sie `List` implementiert und damit auch mit allen Collection Klassen zusammenarbeiten kann. Alle Methoden, welche auf Elemente des Vectors zugreifen, sind synchronisiert, also eher langsam. Die Klasse besitzt viele Methoden, welche analog zu den Methoden in der ArrayList Klasse aufgebaut sind bzw. zu jenen des Interfaces List.

Die alten / legacy Konstruktoren der Klasse `Vector` sind analog zu jenen der `ArrayList` Klasse:

`public` **`Vector()`**
kreiert ein `Vector` Objekt mit einer Standardkapazität

`public` **`Vector(int initalCapacity)`**
kreiert ein neues `Vector` Objekt, mit einer Kapazität initialCapacity. Der Vector besitzt am Anfang die Speicherplatz gemäss initialCapacity.

`public` **`Vector(Collection coll)`**
kreiert ein `Vector` Objekt, dessen Inhalt den Elementen der Collection `coll` entspricht. Die Kapazität des Vectors wird am Anfang auf 110% der Collection gesetzt, damit noch etwas Reserve vorhanden ist. Auch die Ordnung der Elemente ist durch die Collection vorgegeben.

Die Klasse `Vector` definiert auch Methoden:

```
public final void addElement(Object elem)
```
analog zu `add(elem)`

```
public final void insertElementAt(Object elem, int index)
```
analog zu `add(index, elem)`

```
public final void setElementAt(Object elem, int index)
```
analog zu `set(index, elem)`

```
public final void removeElementAt(int index)
```
analog zu `remove(index)`

```
public final boolean removeElement(Object elem)
```
analog zu `remove(elem)`

```
public final void removeAllElements()
```
analog zu `clear()`

```
public final Object elemetAt(int index)
```
analog zu `get(index)`

```
public final void copyInto(Object[ ] anArray)
```
spezielle Methode, welche `IndexOutOfBounds` Exception wirft.

`public final int indexOf(Object elem, int index)`
sucht das erste Auftreten des Elements elem beginnend beim Index `index`.

`public final int lastIndexOf(Object elem, int index)`
sucht rückwärts ab dem Index `index`.

`public final Enumeration elements()`
analog zu `iterator()`, äquivalent zu `Collections.enumeration`

`public final Object firstElement()`
analog zu `get(0)`

`public final Object lastElement()`
analog zu `get(size() -1)`

`public final void setSize(int newSize)`
falls die Grösse `newSize` kleiner als die Grösse des Arrays ist, wird der Rest abgeschnitten. Sonst werden neue Elemente hinzugefügt.

`public final int capacity()`
liefert die Kapazität des Vektors.

1.12.3. Stack

Der `Stack` ist eine Erweiterung des Vectors mit zusätzlichen Methoden, zum Bearbeiten mit First-In und First-Out Methoden. Diese werden typischerweise als `pop()` und `push()` bezeichnet: `push()` legt ein Objekt auf den Stack; `pop()` liest und entfernt ein Objekt. Die `peek()` Methode liefert das oberste Element. Die `empty()` Methode liefert true, falls der Stack leer ist. Falls Sie dann noch versuchen Elemente zu manipulieren, wird eine `EmptyStackException` geworfen.

Sie können auch Elemente suchen, mit `search()`: 1 ist `TopOfStack`; -1 besagt, dass das Element nicht vorhanden ist. Search verwendet die `equals()` Methode.

Alle diese Methoden verwenden letztlich die Vector oder `ArrayList` Methoden.

Selbsttestaufgabe 5

Versuchen Sie selber einen Stack mit einer ArrayListe zu implementieren. Verwenden Sie die `ArrayList` Methoden um die obigen Methoden zu implementieren.

1.12.4. Dictionary

Die `Dictionary` Klasse ist in Wahrheit ein Interface, analog zum `Map` Interface. Die Methoden sind analog zum Map definiert: `get, put, remove, site` und `isEmpty`.

Die Methoden `keys()` und `elements()` sind nicht gleich definiert. Beim Dictionary müssen Sie mit der Enumeration arbeiten, falls Sie die Methoden `keys()` und `elements()` verwenden wollen.

1.12.5. Hashtable

Die `Hashtable` Klasse ist analog zu `HashMap` definiert und implementiert dieselben Methoden wie `Dictionary`. Alle Methoden der `Hashtable` sind synchronisiert. `Hashtable` implementiert das `Map` Interface nicht! Der Grund ist der, dass die Keys *inkompatibel* sind.

Die Methoden haben aber analoge Namen:

`containsKey, containsValue, outAll, keySet, entrySet, values, clear, equals` und `hashcode`.

Zusätzlich werden folgende Methoden implementiert:

`public Hashtable()`
analog zu `HashMap`

`public Hashtable(int initialCapacity)`
analog zu `HashMap(initialCapacity)`

`public Hashtable(int initialCapacity, float loadFactor)`
analog zu `HashMap(initialCapacity, loadFactor)`

`public Hashtable(Map map)`
analog zu `HashMap(map)`

`public boolean contains(Object elem)`
analog zu `containsValue(elem)`

1.13. Properties

Ein `Property` wird benutzt um Zeichenketten abzuspeichern und zugeordnete Zeichenketten oder Werte. Diese Art Hash-Tabellen werden (`Properties` erweitern Hashtable) werden zum manipulieren von `Properties` Objekten eingesetzt: zum Setzen und Bestimmen von Werten zu bestimmten Schlüsseln.

Weitere Methoden werden spezifisch für `Properties` definiert.

`public Properties()`
kreiert einen leeren Property Map

`public Properties(Properties defaults)`
kreiert einen leeren Property Map mit speziellen Standardproperties.
Falls der Lookup für ein Objekt fehlschlägt, wird ein Standardobjekt geliefert.

`public String getProperty(String key)`
liefert das `Property` Element für den Schlüssel `key`. Falls kein Wert gefunden wird könnte ein Standardwert zurückgeliefert werden, sofern die Standardproperties definiert wurden.

`public String getProperty(String key, String defaultElement)`
liefert das `Property` Element zu key. Falls `key` nicht gefunden wird, wird der Standardwert gesucht und falls keiner gefunden wird, wird `defaultElement` weiterverwendet.

`public String setProperty(String key, String value)`
fügt einen Property `key` hinzu, mit gegebenem Wert `value`. Standardproperties sind nicht betroffen.

`public void store(OutputStream out, String header) throws IOException`
speichert die Properties in einem Ausgabestrom. Dies funktioniert nur dann, falls es sich bei den Properties um Zeichenketten handelt, sonst wird eine `ClassCastException` geworfen.

`public void loaf(InputStream in) throws IOException`
lädt die Properties aus einem `InputStream`, wobei diese in der Regel mit der `store...()` Methode gespeichert wurden.

`public Enumeration propertyNames()`
listet alle Schlüssel auf, auch die Standardschlüssel.

`public void list(PrintWriter out)`
druckt die Properties auf den DruckerStrom.

`public void list(PrinStream out)`
wie oben, aber mit einem `PrintStream`

Die Standardproperties können Sie nach dem Kreieren eines `Property` Objekts nicht mehr verändern, was eigentlich einleuchten sollte: das wäre ein volles Chaos.

Science is facts;
just as houses are made of stones, so is science made of facts;
but a pile of stones is not a house and a collection of facts is not necessarily science.
- Henri Poincaré

COLLECTIONS