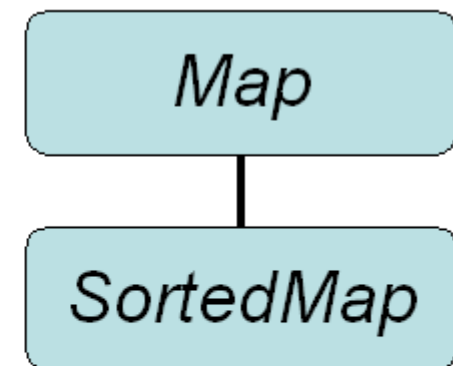
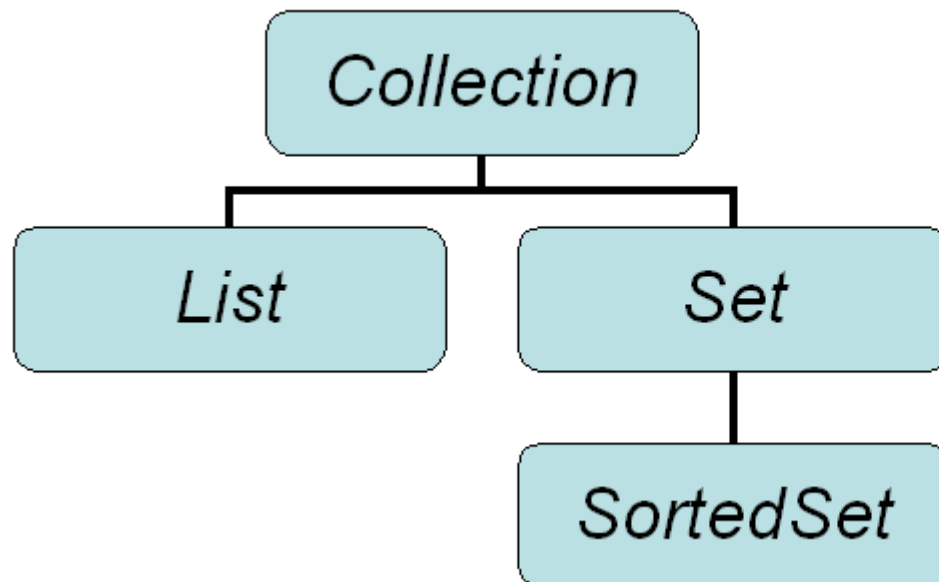


# 4.4.9 Das Java Collection Framework

<http://java.sun.com/j2se/1.4.2/docs/api/java/util/Collection.html>

- eine Sammlung von Interfaces, die die Organisation von Objekten in “Container” unterstützt
  - Spezifikation von Methoden zur Unterstützung der Erstellung von Collections
    - Hinzufügen und Löschen von Objekten, etc.
- Die wichtigsten Container-Interfaces:



# Die wichtigsten Elemente

- **java.util.Collection**
  - Interface, um eine Gruppe von Objekten zu organisieren
  - Basis-Definitionen für Hinzufügen und Entfernen von Objekten
- **java.util.List**
  - Collection Interface, das zusätzlich jedem Element eine fixe Position zuweist
- **java.util.Set**
  - Collection Interface, das keine doppelten Elemente erlaubt (Mengen)
- **java.util.Map**
  - Interface, das die Zuordnung von Elementen zu sogenannten *Schlüsseln* unterstützt
  - erlaubt, Elemente mit dem zugehörigen Schlüssel anzusprechen
  - Map ist keine Unterklasse von Collection!

# Nützliche Hilfs-Interfaces

- **java.util.Iterator**
  - Interface, das Methoden spezifiziert, die es erlauben, alle Elemente einer Collection aufzuzählen
  - ersetzt weitgehend das ältere `java.util.Enumeration` Interface
- **java.util.Comparator**
  - Interface, das Methoden zum Vergleich von Elementen einer Collection spezifiziert

# Wichtige Methoden des Collection Interfaces

- `boolean add(Object obj)`
  - füge `obj` zur `Collection` hinzu
  - `return true` wenn sich die `Collection` dadurch verändert hat
- `boolean contains(Object obj)`
  - `return true` wenn `obj` bereits enthalten ist
- `boolean isEmpty()`
  - `return true` wenn die `Collection` keine Elemente enthält
- `Iterator iterator()`
  - `return` ein `Iterator` Objekt, mit dem man die Elemente einer `Collection` auf zählen kann
- `boolean remove(Object obj)`
  - entfernt ein Element, das `equal` zu `obj` ist, falls eins existiert
  - `return true`, falls sich die `Collection` dadurch verändert hat
- `int size()`
  - `return` die Anzahl der Elemente in der `Collection`

# Weitere Methoden des Collection Interface

- `boolean addAll(Collection c)`
  - fügt zur **Collection** alle Objekte aus der **Collection c** hinzu
- `boolean containsAll(Collection c)`
  - `true` wenn alle Objekte aus der **Collection c** enthalten sind
- `boolean removeAll(Collection c)`
  - entfernt alle Objekte aus der **Collection**, die sich in einer anderen **Collection c** befinden
- `boolean retainAll(Collection c)`
  - behalte nur Objekte, die sich auch in der **Collection c** befinden
- `void clear()`
  - entfernt alle Objekte aus der **Collection**
- `Object[] toArray()`
  - retourniert die Elemente der **Collection** in einem **Array**
- `Object[] toArray(Object[] a)`
  - retourniert einen **Array** vom selbem (dynamischen) Typ wie **a**

# java.util.Iterator

- Das Iterator Interface dient zur Iteration über Elemente einer Collection
  - kann aber auch von anderen Klassen implementiert werden
- Festgelegte Methoden:
  - `boolean hasNext()`
    - überprüft, ob die Collection noch zusätzliche Elemente hat
  - `Object next()`
    - retourniert das letzte Objekt in der Collection
  - `void remove()`
    - entfernt das letzte Element, das vom Iterator retourniert wurde, aus der Collection
- Anmerkung:
  - seit der Java Version 1.5. gibt es auch eine spezielle Version der for-Schleife zur Iteration über ein Collections

# Beispiel 1

- Das Programmstück aus dem letzten Beispiel von 4.4.8

```
for ( int i=0; i<v.size(); i++ ) {  
    Integer x = (Integer) (v.get(i));  
    System.out.print ( x.intValue() );  
}
```

- könnte man auch so schreiben

```
Iterator it = v.iterator();  
while (it.hasNext()) {  
    Integer x = (Integer) it.next();  
    System.out.println (x.intValue());  
}
```

leeres Update, update von x  
erfolgt im Schleifen-Rumpf!

- oder so

```
for (Iterator it = v.iterator(); it.hasNext(); ) {  
    Integer x = (Integer) it.next();  
    System.out.println (x.intValue());  
}
```

# Beispiel 2

- Methode zum Filtern von Elementen aus einer Collection

```
static void filter(Collection c) {  
    Iterator i = c.iterator();  
    while (i.hasNext()) {  
        if (filterTest(i.next()))  
            i.remove();  
    }  
}
```

- Beachte, daß diese Methode für jede beliebige Klasse funktionieren würde, die eine boolean Methode names `filterTest` definiert hat
  - ganz egal, wie die Datenkomponenten aussehen
  - solange die Klasse angibt, daß sie das Interface `Iterator` implementiert



# Interface `java.util.List`

- Spezifiziert eine Collection, bei der die Elemente durchnummeriert sind
    - ähnlich wie in einem Array
    - aber mit flexibleren Zugriffsmöglichkeiten
      - z.B. veränderlicher Größe
  - realisiert als Unterklasse von `java.util.Collection`
    - das heißt, Objekte, die dieses Interface implementieren, müssen alle Collection-Methoden unterstützen
    - und zusätzlich noch Methoden, die einen Zugriff über die Position des Elements erlauben
  - Die totale Ordnung aller Objekte erlaubt auch, daß man die Objekte in beide Richtungen durchlaufen kann
    - es gibt daher auch eine spezielle Unterklasse des Iterator-Interfaces, das erlaubt, in beide Richtungen zu laufen
      - z.B. `hasPrevious()`, `previous()`, etc.
- Interface `ListIterator`

# Die wichtigsten zusätzlichen Methoden

- `Object get(int i)`
  - **retourniere das i-te Element**
- `Object set(int i, Object o)`
  - **weise dem i-ten Element das Objekt o zu**
- `int indexOf(Object o)`
  - **Index des ersten Objekts, für das `equals(o)` gilt**
  - **-1 falls es kein so ein Element gibt**
- `void add(int i, Object o)`
  - **fügt o an der i-ten Stelle der Liste ein**
  - **retourniert das Element, das sich vorher an dieser Stelle befunden hat**
- `Object remove(int i)`
  - **entfernt und retourniert das Objekt an der i-ten Stelle**
- `List subList(int von, int bis)`
  - **retourniert die Teil-Liste beginnend mit `von`, endend mit `bis-1`**

# Vordefinierten Listen-Klassen

- Klasse `LinkedList`
  - Implementiert eine Liste mit expliziter Verkettung
    - d.h. in den Datenkomponenten wird ein Verweis auf das nächste und vorhergehende Listen-Element abgespeichert
  - ähnlich zu der Implementieren, die wir im Abschnitt über rekursive Datenstrukturen kennengelernt haben
- Klasse `ArrayList`
  - Implementiert eine Liste mittels eines Arrays
    - d.h. die Elemente der Liste werden in einem Array abgespeichert
- Vor- und Nachteile:
  - `ArrayList` ist schneller im Zugriff auf indizierte Elemente
    - da sich die Adresse direkt berechnen läßt
  - `LinkedList` ist schneller im Einfügen und Entfernen
    - da die restlichen Einträge der Liste unberührt bleiben.
- Aber nach außen hin können sehen beide gleich aus
  - da beide das Interface `List` implementieren

# java.util.Vector

- Haben wir bereits kennengelernt
- Implementiert das `List`-Interface
  - intern mittels eines Arrays
  - sehr ähnlich zu `ArrayList`
    - Unterschiede nur beim Zugriff aus mehreren Threads
    - Daseinsberechtigung hauptsächlich aus historischen Gründen
- Beachte:
  - die Klasse war bereits Teil von Java, bevor in der Version 1.2 das Collection Framework definiert wurde
  - daher gibt es für einige Methoden auch noch andere Namen
    - z.B. `elementAt(i)` und `get(i)` machen genau das gleiche

# Interface `java.util.Set`

- Spezifiziert eine `Collection`, in der kein Element doppelt vorkommen darf
  - also eine Menge
- Dieses Interface implementiert genau die Methoden, die für `Collection` vorgeschrieben sind
  - aber keine zusätzlichen Methoden!
- Wozu wurde dann ein eigenes Interface definiert?
  - die Bedeutung der Methoden ist eine andere
  - Beispiel:
    - für ein Objekt, das `List` implementiert, verändert die Methode `add` die `collection`.
    - für ein Objekt, das `Set` implementiert, wird die Methode nur verändert, wenn das hinzuzufügende Element nicht schon in der `Collection` enthalten ist.
      - Zur Überprüfung Aufruf von `equals`!
  - Unterschied liegt also in der Semantik, nicht in der Syntax

# Interface `java.util.HashSet`

- implementiert das Interface `Set` mit Hilfe einer Hash-Tabelle
  - was eine Hash-Tabelle ist werden wir in Kürze sehen

# Beispiel

```
import java.util.*;

public class LottoZiehung
{
    public static void main(String[] args)
    {
        HashSet zahlen = new HashSet();
        //Lottozahlen erzeugen
        while (zahlen.size() < 6) {
            int num = (int) (Math.random() * 49) + 1;
            if (zahlen.add(new Integer(num))) {
                System.out.println("Neue Zahl " + num);
            }
            else {
                System.out.println("DoppelteZahl " + num
                    + " ignoriert");
            }
        }
        //Lottozahlen ausgeben
        Iterator it = zahlen.iterator();
        while (it.hasNext()) {
            System.out.println(((Integer) it.next()).toString());
        }
    }
}
```

Für ein Set retourniert add true, wenn ein neues Element hinzugefügt wurde, false sonst.

# Interface `java.util.SortedSet`

- Im vorigen Beispiel werden die sechs Lotto-Zahlen in der Reihenfolge, in der sie gezogen wurden, präsentiert
  - schöner wär's natürlich, wenn Sie in sortiert wären
  - dazu kann man sie entweder selber sortieren...
- ... oder eine Klasse verwenden, die das Interface `java.util.SortedSet` implementiert
  - wie `java.util.Set`, nur daß die Elemente sortiert werden
- Problem: Wie vergleiche ich zwei beliebige Elemente?



# java.util.Comparator und java.util.Comparable

- `java.util.Comparable`
  - Interface das angibt, daß auf Objekten, die diese Interface implementieren, eine totale Ordnung definiert ist.
    - die sogenannte **natürliche Ordnung** des Objekts
  - Einzige Methode: `int compareTo(Object o)`
    - vergleicht dieses Objekt mit dem Objekt `o`
    - Rückgabewert:
      - 0 wenn beide Objekte gleich sind
        - äquivalent zu `this.equals(o)`
      - positive Zahl, wenn dieses Objekt als größer als `o` anzusehen ist
      - negative Zahl, wenn dieses Objekt kleiner als `o`
- `java.util.Comparator`
  - ein eigenes Objekt zum Durchführen von Vergleichen
  - Einzige Methode: `int compare(Object o1, Object o2)`
    - vergleicht zwei Objekte `o1` und `o2`
    - Rückgabewert analog zu oben

# Neue Methoden von SortedSet

- `Comparator comparator()`
  - retourniert den Vergleichsoperator, der verwendet wird, oder null, falls die natürliche Ordnung der Elemente verwendet wird
- `Object first()`
  - Das erste (niedrigste) Element in der sortierten Menge
- `Object last()`
  - Das letzte (höchste) Element in der sortierten Menge
- `SortedSet headSet(Object o)`
  - alle Elemente des Sets, die kleiner als `o` sind
- `SortedSet tailSet(Object o)`
  - die Elemente des Sets, die größer oder gleich `o` sind
- `SortedSet subSet(Object o1, Object o2)`
  - die Teilmenge von Elementen, die größer oder gleich `o1` sind und kleiner als `o2` sind

# Klasse `java.util.TreeSet`

- Implementiert das `SortedSet` Interface
  - Speicherung der Mengen erfolgt in einer Baum-Struktur
  - stellt sicher, daß in jedem Knoten nur ein Element vorhanden ist
  - stellt auch die Sortierung sicher
- Beispiel:
  - Um sortierte Lotto-Zahlen zu erhalten, muß man in unserem Beispiel nur die Zeile

```
HashSet zahlen = new HashSet();
```

ersetzen durch

```
TreeSet zahlen = new TreeSet();
```

- So einfach ist das dank der Verwendung von Interfaces!
  - da alle Methoden-Namen durch das gemeinsame Interface festgelegt sind.

# Konstruktoren von `TreeSet`

- `TreeSet ()`
  - erzeugt ein leeres `TreeSet`
    - analog zu Konstruktoren von `HashSet`
  - die Elemente werden nach Ihrer natürlichen Ordnung sortiert
    - das heißt, die Objekte müssen das Interface `Comparable` implementieren
    - Sortierung erfolgt durch Aufruf der `compareTo`-Methode der zu sortierenden Objekte
- `TreeSet (Comparator c)`
  - erzeugt ein leeres `TreeSet`
  - die Elemente werden durch Aufruf des Comparators `c` sortiert
    - das heißt, die Objekte müssen nicht das Interface `Comparable` implementieren
    - oder die Sortierung kann auch nach einer anderen Methode als der natürlichen Ordnung erfolgen

# Interface `java.util.Map`

- realisiert einen assoziativen Speicher
  - ein assoziativer Speicher besteht aus einer Tabelle von Paaren von Schlüssel und Wert
- **Schlüssel** (Keys):
  - sind beliebige Objekte
  - jeder Schlüssel kann nur maximal einmal vorkommen
- **Werte** (Values):
  - sind ebenfalls beliebige Objekte
  - jedem Schlüssel wird genau ein Wert zugeordnet
- die Tabelle ist als Abbildung von beliebigen Objekten (Schlüsseln) auf andere Objekte (Werte) zu verstehen
  - genauso wie **bei einem Array** oder bei einer Liste **jeder integer** Zahl (dem Index) **ein Objekt** zugeordnet ist
  - kann **bei einem Map einem beliebigen Objekt** (dem Schlüssel) **ein anderes Objekt** zugeordnet werden (mapping = Abbildung)

# Schematische Illustration

- List (bzw. Array)

0 → object1

1 → object2

2 → object3

...

n-1 → objectn

```
l.set(2, object3)
Object o = l.get(2);
```

- Map

“dog” → object1

“cat” → object2

“pig” → object3

...

“last” → objectn

```
m.put("pig", object3)
Object o = m.get("pig");
```

# Wichtige Methoden von `java.util.Map`

- `Object get(Object key)`
  - retourniert den dem Schlüssel `key` zugeordneten Wert
- `Object put(Object key, Object val)`
  - ordnet dem Schlüssel `key` den Wert `val` zu
- `Object remove(Object key)`
  - Entfernt den Schlüssel `key` (und damit den zugehörigen Wert)
- `boolean containsKey(Object key)`
  - `true` wenn es einen Eintrag mit dem Schlüssel `key` gibt
- `boolean containsValue(Object val)`
  - `true` wenn es einen Eintrag mit dem Wert `val` gibt
- `Set keySet()`
  - retourniert alle Schlüssel als ein `Set` (keine Duplikate)
- `Collection values()`
  - retourniert alle Werte als eine `Collection`

# Klasse `java.util.HashMap`

- realisiert eine Map mit einer sogenannten Hash-Tabelle
- Hash-Tabellen ordnen jedem Objekt eine fixe Zahl zu
  - die Zuordnung sollte nach Möglichkeit eindeutig sein (also verschiedene Objekte werden mit verschiedenen Zahlen identifiziert)
  - und die (große) Anzahl der möglichen Objekte auf eine relativ kleine Anzahl von möglichen Werten reduzieren
  - und relativ effizient zu berechnen sein
- Einige bekannte Hashing-Funktionen
  - MD5
  - SHA-1
- Die Idee von Checksummen ist ähnlich der Idee von Hash-Funktionen
  - im Prinzip kann eine Checksumme als eine Hash-Funktion verwendet werden und umgekehrt



# Hash-Codes

- Wo kommen die Hash-Codes her?
  - Methode `hashCode()` von `java.lang.Object`
    - kann ererbt, aber auch speziell implementiert werden
- Anforderungen
  - für jedes Objekt muß ein Integer Code retourniert werden
  - während eines Ablaufs des Programms muß immer der gleiche Code retourniert werden
  - bei verschiedenen Abläufen können es auch verschiedene Codes sein
  - es ist *nicht* verlangt, daß das zwei verschiedene Objekte verschiedene Hash-Codes retournieren (wär aber gut)
- Die Default-Implementierung, die von `java.lang.Object` geerbt wird retourniert üblicherweise die Speicheradresse des Objekts
  - muß aber nicht so sein

# Beispiel

```
import java.util.*;

public class MailAliases
{
    public static void main(String[] args)
    {
        HashMap h = new HashMap();

        //Pflege der Aliase
        h.put("Fritz", "f.mueller@test.de");
        h.put("Franz", "fk@b-blabla.com");
        h.put("Paula", "user0125@mail.uofm.edu");
        h.put("Lissa", "lb3@gateway.fhdto.northsurf.dk");

        //Ausgabe
        Iterator it = h.keySet().iterator();
        while (it.hasNext()) {
            String key = (String)it.next();
            System.out.println(key + " --> "
                               + (String)h.get(key) );
        }
    }
}
```

# Weitere Map-Klassen

- **Klasse** `java.util.Hashtable`
  - alte Implementation von Hash-Tables
  - Unterschiede zu `HashMap` minimal
- **Interface** `java.util.SortedMap`
  - Eine Map, bei der die Schlüssel sortiert bleiben
  - d.h. die Schlüssel bilden kein `Set`, sondern ein `SortedSet`
- **Klasse** `java.util.TreeMap`
  - Klasse, die das `SortedMap` Interface implementiert

# Klasse `java.util.Collections`

- Eine Sammlung von statischen Methoden zum Arbeiten mit Collections
- Methoden zum
  - Sortieren
    - `static void sort(List list)`
      - Sortieren nach natürlicher Ordnung der Objekte
    - `static void sort(List list, Comparator c)`
      - Sortieren nach dem Comparator-Objekt
  - Suchen
  - Kopieren
  - Mischen
  - ....
- Details → Java API Dokumentation

# Beispiel

- Im vorigen Beispiel können die E-mail-Adressen in alphabetischer Reihung ausgegeben werden:

```
//Ausgabe
ArrayList keys = new ArrayList(h.keySet());
Collections.sort(keys);
Iterator it = keys.iterator();
while (it.hasNext()) {
    String key = (String)it.next();
    System.out.println(key + " --> "
        + (String)h.get(key) );
}
```

- wird die Sortierung häufig verwendet, würde sich in diesem Beispiel natürlich die Verwendung von `TreeMap` statt `HashMap` empfehlen
  - das Programm müßte außer bei der Erzeugung von `h` wiederum nicht verändert werden!

# Abstrakte Basis-Klassen

- Zur Neu-Definition von Objekten, die Collection Interfaces implementieren, kann man (wenn man möchte) von existierenden abstrakten Basis-Klassen ableiten
  - `AbstractCollection`
  - `AbstractList`
  - `AbstractSet`
  - `AbstractMap`
- Insbesondere sind alle bisher besprochenen Klassen von diesen abstrakten Klassen abgeleitet