Vorlesung 07: Collections

Peter Thiemann

Universität Freiburg, Germany

SS 2011

Inhalt

Literaturhinweis

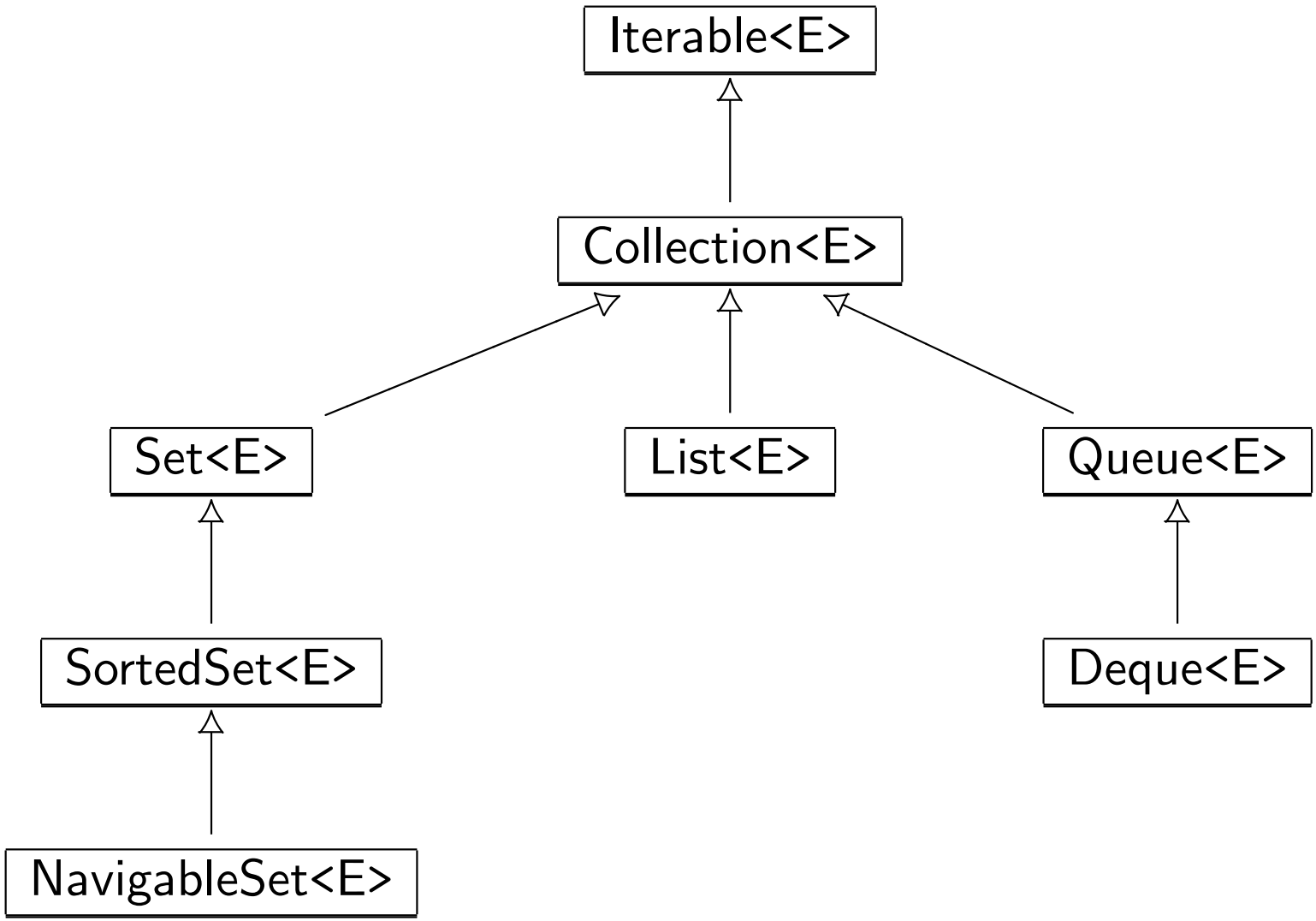
Java Generics and Collections
Maurice Naftalin, Philip Wadler
O'Reilly, 2006

Übersicht: Java Collections Framework

- Collection: Oberbegriff für Aggregatdatentypen (Containerdatentypen), in denen andere Elemente enthalten sind,
- Operationen: Hinzufügen, Entfernen, Suchen, Durchlaufen
- Hauptinterfaces (in `java.util`):
  - *`Collection`* Grundfunktionalität für alle Datentypen *außer für Abbildungen*. Keine konkrete Implementierung.
  - *`Set`* Aggregation ohne Wiederholung, Reihenfolge unerheblich. Zwei Spezialisierungen: `SortedSet` und `NavigableSet`.
  - *`Queue`* Warteschlange, FIFO. Spezialisierung: `Deque` (an beiden Enden anfügen und entfernen)
  - *`List`* Aggregation mit Wiederholung und fester Reihenfolge.
  - *`Map`* endliche Abbildung. Spezialisierung: `SortedMap` und `NavigableMap`

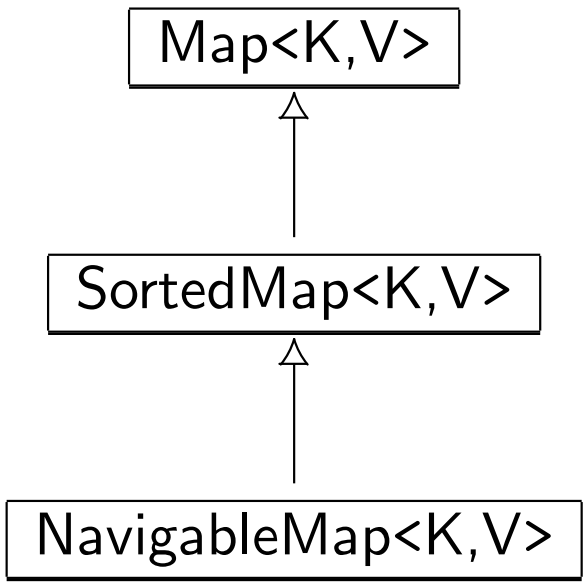
Übersicht Collections

Elementtyp `E`



Übersicht Abbildungen

Argumenttyp (Schlüssel) `K`, Ergebnistyp (Wert) `V`



Iteratoren (Durchlaufinterfaces)

Iterator

Ein *Iterator* implementiert das folgende (Durchlauf-) Interface

```
public Iterator<E> {
    // returns true if the iteration has more elements
    boolean hasNext();
    // returns the next element in the iteration
    E next();
    // removes the last element returned by the iterator (optional)
    void remove();
}
```

Jeder Aufruf von `next()` liefert ein anderes Element.

Durchlaufen mit Iterator

- (Veraltetes) Muster zum Verarbeiten einer Collection mit Iterator

```
Collection<E> coll;
...
for (Iterator<E> iter = coll.iterator(); iter.hasNext(); ) {
  E elem = iter.next ();
  System.out.println (elem);
}
```

Durchlaufen mit Iterator

- (Veraltetes) Muster zum Verarbeiten einer Collection mit Iterator

```
Collection<E> coll;
...
for (Iterator<E> iter = coll.iterator(); iter.hasNext(); ) {
  E elem = iter.next ();
  System.out.println (elem);
}
```

- Ab Java 5: neue Variante der `for`-Anweisung, die dieses Muster implementiert (`foreach`-Anweisung)

```
Collection<E> coll;
...
for (E elem : coll) {
  System.out.println (elem);
}
```

Das Interface `Iterable`

- Die `foreach`-Anweisung funktioniert mit jedem Datentyp, der das Interface `Iterable` implementiert:

```
public Iterable<E> {
    // returns an iterator over elements of type E
    Iterator<E> iterator();
}
```

- Verwendbarkeit von foreach:
  - Jede `Collection`:
    Das `Collection`-Interface erweitert das `Iterable`-Interface.
  - Arrays
  - beliebige Klassen, die `Iterable` implementieren
  - leider nicht: `String`

Beispiel: Ein Array mit `Iterable` durchlaufen

```
class Echo {
    public static void main (String[] arg) {
        for (String s : arg) {
            System.out.println(s);
        }
    }
}
```

Beispiel: Ein Array mit `Iterable` durchlaufen

```
class Echo {
    public static void main (String[] arg) {
        for (String s : arg) {
            System.out.println(s);
        }
    }
}
```

Ausführen liefert

```
> java Echo gibt ein schoenes Echo
gibt
ein
schoenes
Echo
```

Beispiel: Ein Zähler mit `Iterable`

Ein `Iterable` selbst machen

```
class Counter implements Iterable<Integer> {
    private int count;
    public Counter (int Count) { this.count = count; }
    public Iterator<Integer> iterator () {
        return new CounterIterator (this.count);
    }
}

class CounterIterator implements Iterator<Integer> {
    private int count;
    private int i;
    CounterIterator (int count) { this.count = count; this.i = 0; }
    public boolean hasNext () { return this.i <= this.count; }
    public Integer next() { this.i++; return this.i; }
    public void remove () { throw new UnsupportedOperationException(); }
}
```

Beispiel: Ein Zähler mit `Iterable`

Verwendung

```
int total = 0;
for (int i : new Counter (3)) {
  total += i;
}
assert total == 6;
```

- Die `assert`-Anweisung testet einen Ausdruck vom Typ boolean.
- Wenn der Ausdruck zu false auswertet, bricht das Programm mit einer Exception ab.

Zwischenspiel: Anonyme Klassen

- Die Klasse `CounterIterator` wird nur innerhalb von `Counter` verwendet.
- Daher ist es sinnvoll, sie innerhalb von `Counter` zu definieren.
- (Ähnliches gilt für die Implementierungen von `ISelect` und `ITransform`)
- Anwendungsfall für eine *anonyme Klasse*:
  - Definiert ein Objekt zu einem Interface oder einer abstrakten Klasse.
  - Im Objekt können Instanzvariable und (müssen) Methoden definiert werden.
  - Definition geschieht an der Stelle, wo das Objekt (einmalig) benötigt wird.

Beispiel: Count mit anonymer Klasse

```
class Counter implements Iterable<Integer> {
    private final int count;
    public Counter (int Count) { this.count = count; }
    public Iterator<Integer> iterator () {
        return new Iterator<Integer> () {
            private int i = 0;
            public boolean hasNext () { return this.i <= count; }
            public Integer next() { this.i++; return this.i; }
            public void remove () { throw new UnsupportedOperationException(); }
        };
    }
}
```

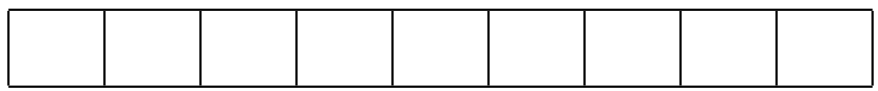
Implementierungen

Implementierungen

- Das Java Collection Framework besteht aus *Interfaces*.
- Zu jedem Interface gibt es *mehrere Implementierungen*.
- Grund:
  - Kompromisse beim Entwurf von Datenstrukturen
  - Für jede Datenstruktur sind gewisse Operationen sehr effizient, dafür sind andere weniger effizient.
  - Hauptoperationen
    - Zugreifen auf Elemente nach Position
    - Einfügen und Entfernen von Elementen nach Position
    - Auffinden von Elementen
    - Durchlaufen
  - Jede Anwendung hat einen anderen Mix von Operationen: Auswahl der Implementierung je nach Anwendung (ggf. auch dynamisch)
- **Programme sollten sich ausschließlich auf Interfaces beziehen**
- **Einzige Ausnahme**: Erzeugen der Datenstruktur (selbst das sollte konfigurierbar sein)

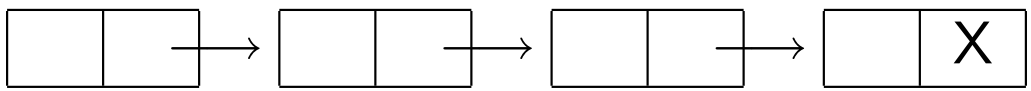
Implementierung mit Arrays

- Schneller Zugriff über Position $O(1)$
- Schnelles Durchlaufen
- Einfügen und Entfernen in $O(n)$, da andere Elemente nachrücken bzw. Platz schaffen müssen
- Verwendet für `ArrayList`, `EnumSet`, `EnumMap` sowie für viele `Queue` und `Deque`-Implementierungen



Verkettete Listen

- Liste ist entweder leer oder enthält ein Listenelement und einen Verweis auf eine Restliste.
- Langsamer Zugriff nach Position $O(n)$
- Einfügen und Löschen in $O(1)$
- Verwendet für `LinkedList`, `HashSet` und `LinkedHashSet`.



Hashtabellen

- Elemente werden über ihren Inhalt indiziert
- Ablage in Tabelle der Größe m
- Hashfunktion `hashCode()` : Inhalt $\rightarrow [0, m)$ (Hashcode)
- Ablage über den Hashcode
- **Kein Zugriff über Position**
- Zugriff über Inhalt, Einfügen, Löschen fast in $O(1)$
- Verwendet für `HashSet`, `LinkedHashSet`, `HashMap`, `LinkedHashMap`, `WeakHashMap`, `IdentityHashMap`

Bäume

- (Such-) Bäume, organisiert über ihren Inhalt
- Sortierte Ausgabe und Durchlauf einfach und effizient möglich
- Einfügen, Löschen und Zugriff auf Element in $O(\log n)$
- Verwendet für `TreeSet`, `TreeMap`, `PriorityQueue`

Das Interface `Collection`

Überblick

```
public interface Collection<E> {
    public boolean add (E o);
    public boolean addAll (Collection<? extends E> c);
    public boolean remove (Object o);
    public void clear();
    public boolean removeAll(Collection<?> c);
    public boolean retainAll(Collection<?> c);
    public boolean contains (Object o);
    public boolean containsAll (Collection<?> c);
    public boolean isEmpty();
    public int size();
    public Iterator<E> iterator();
    public Object[] toArray();
    public <T>T[] toArray (T[] a);
}
```

Einfügen von Elementen

```
// adds the element o
public boolean add (E o);
// adds all elements in collection c
public boolean addAll (Collection<? extends E> c);
```

- Liefern `true`, falls Operation erfolgreich
- Bei Mengen: `false`, falls Element schon enthalten
- Exception, falls Element aus anderem Grund illegal
- Argument von `addAll` verwendet den *Wildcard Typ ?*:
  Jedes Argument vom Typ Collection<T> wird akzeptiert, falls T ein Subtyp von E ist

Löschen von Elementen

```
// removes element o
public boolean remove (Object o);
// removes all elements
public void clear();
// removes all elements in c
public boolean removeAll(Collection<?> c);
// removes all elements not in c
public boolean retainAll(Collection<?> c);
```

- Argument von remove hat Typ `Object`, nicht `E`
- Argument von removeAll bzw. retainAll ist Collection mit Elementen von beliebigem Typ
- Argument/Element `null` entfernt eine `null`
- Argument/Element $\neq$ `null` entfernt Eintrag, der `equals` ist
- Rückgabewert `true`, falls die Operation die Collection geändert hat

Abfrage des Inhalts

```
// true if element o is present
public boolean contains (Object o);
// true if all elements of c are present
public boolean containsAll (Collection<?> c);
// true if no elements are present
public boolean isEmpty();
// returns number of elements (or Integer.MAX_VALUE)
public int size();
```

Alle Elemente verarbeiten

```
// returns an iterator over the elements
public Iterator<E> iterator();
// copies the elements into a new arry
public Object[] toArray();
// copies the elements into an array
public <T>T[] toArray (T[] a);
```

- Die "T" Methode kopiert die Elemente der Collection in ein Array mit Elementen von *beliebigem* Typ T.
- Laufzeitfehler, falls die Elemente nicht Typ T haben
- Wenn im Argumentarray a genug Platz ist, wird es verwendet, sonst wird ein neues Array angelegt.
- Verwendung: Bereitstellen von Argumenten für Legacy-Methoden, die Arrays als Argument erwarten

Die Methode <T>T[] toArray (T[] a)

In Collection<E>

- Verwendung:
  - Kopieren in Array von Supertyp von E (geht immer)
  - Kopieren in Array von Subtyp von E, "Spezialisieren der Collection"
- Typisches Muster:
  Argument ist leeres Arrays, das nur den Typ T anzeigt

```
Collection<String> cs = ...;
String[] sa = cs.toArray (new String[0]);
```

- Alternativ, bei mehreren Anwendungen

```
private static final String[] EMPTY_STRING_ARRAY = new String[0];
Collection<String> cs = ...;
String[] sa = cs.toArray (EMPTY_STRING_ARRAY);
```

Die Methode <T>T[] toArray (T[] a)

In Collection<E>

- Spezialisierung eines Arraytyps

```
List<Object> lo = Arrays.asList ("zero", "one");
String[] sa = lo.toArray (new String[0]);
```

- Fehlschlag bei der Spezialisierung

```
List<Object> lo = Arrays.asList ("zero", "one", 4711);
String[] sa = lo.toArray (new String[0]); // Laufzeitfehler
```

Die Methode <T>T[] toArray (T[] a)

In Collection<E>

- Spezialisierung eines Arraytyps

```
List<Object> lo = Arrays.asList ("zero", "one");
String[] sa = lo.toArray (new String[0]);
```

- Fehlschlag bei der Spezialisierung

```
List<Object> lo = Arrays.asList ("zero", "one", 4711);
String[] sa = lo.toArray (new String[0]); // Laufzeitfehler
```

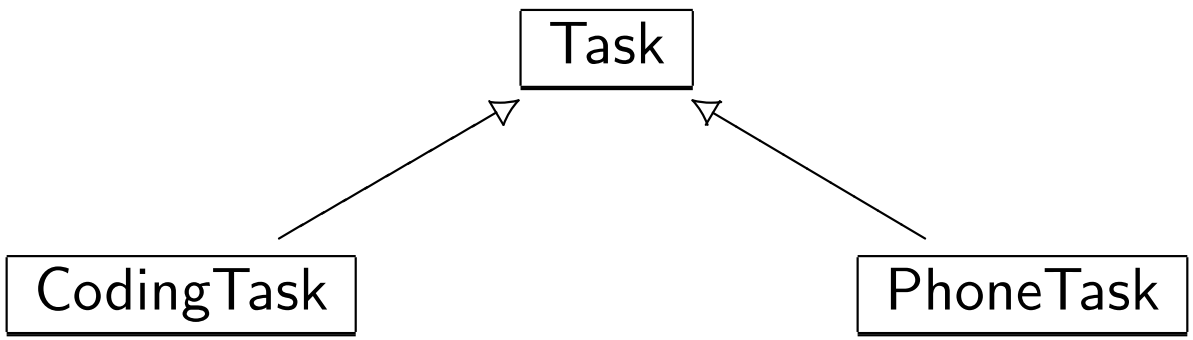
- Sinnvolle T-Typen:
  - Subtypen von E
  - Supertypen von E

Aber das lässt sich in Java nicht ausdrücken...

Verwendung der `Collection` Methoden

Beispiel: Eine Aufgabenliste

In einer Aufgabenliste können sich verschiedene Arten von Aufgaben befinden. Eine Aufgabe ist entweder eine Programmieraufgabe oder ein Telefonanruf, der erledigt werden muss.



Beispiel: Aufgaben, Implementierung

```
abstract class Task implements Comparable<Task> {
    protected Task() {}
    public boolean equals (Object o) {
        if (o instanceof Task) {
            return this.toString().equals (o.toString());
        } else {
            return false;
        }
    }
    public int compareTo (Task t) {
        return this.toString().compareTo (t.toString);
    }
    public int hashCode() {
        return this.toString().hashCode();
    }
    public abstract String toString();
}
```

Beispiel: Spezielle Aufgaben, Implementierung

```
final class CodingTask extends Task {
    private final String spec;
    public CodingTask (String spec) {
        this.spec = spec;
    }
    public String getSpec () { return spec; }
    public String toString() { return "code " + spec; }
}
```

Beispiel: Spezielle Aufgaben, Implementierung

```
final class PhoneTask extends Task {
    private final String name;
    private final String number;
    public PhoneTask (String name, String number) {
        this.name = name;
        this.number = number;
    }
    public String getName () { return name; }
    public String getNumber () { return number; }
    public String toString () { return "phone " + name; }
}
```

Verwendung der Task-Klassen

```
PhoneTask mikePhone = new PhoneTask ("Mike", "0123456789");
PhoneTask paulPhone = new PhoneTask ("Paul", "0190318318");
CodingTask dbCode = new CodingTask ("db");
CodingTask guiCode = new CodingTask ("gui");
CodingTask logicCode = new CodingTask ("logic");

Collection<PhoneTask> phoneTasks = new ArrayList<PhoneTask> ();
Collection<CodingTask> codeTasks = new ArrayList<CodeingTask> ();
Collection<Task> mondayTasks = new ArrayList<Task> ();
Collection<Task> tuesdayTasks = new ArrayList<Task> ();

Collections.addAll (phoneTasks, mikePhone, paulPhone);
Collections.addAll (codingTasks, dbCode, guiCode, logicCode);
Collections.addAll (mondayTasks, logicCode, paulPhone);
Collections.addAll (tuesdayTasks, dbCode, guiCode, mikePhone);

assert phoneTasks.toString().equals ("[phone Mike, phone Paul]");
assert codeTasks.toString().equals ("[code db, code gui, code logic]");
assert mondayTasks.toString().equals ("[code logic, phone Paul]");
assert tuesdayTasks.toString().equals ("[code db, code gui, phone Mike");
```

Elemente hinzufügen

Neue Elemente

```
mondayTasks.add (new PhoneTask ("Ruth", "01907263428"));
assert mondayTasks.toString().equals ("[code logic, phone Paul, phone Ruth]");
```

Pläne kombinieren

```
Collection<Task> allTasks = new ArrayList<Task>(mondayTasks);
allTasks.addAll (tuesdayTasks);
assert allTasks.toString().equals (
  "[code logic, phone Paul, phone Ruth, code db, code gui, phone Mike]")
```

Elemente entfernen

Wenn eine Aufgabe erledigt ist, kann sie entfernt werden.

```
boolean wasPresent = mondayTasks.remove(paulPhone);
assert wasPresent;
assert mondayTasks.toString().equals("[code logic, phone Ruth]");
```

Wenn alle Aufgaben erledigt werden, können alle entfernt werden.

```
mondayTasks.clear();
assert mondayTasks.toString().equals("[]");
```

Alle Aufgaben am Dienstag, die nicht Telefonanrufe betreffen

```
Collection<Task> tuesdayNonphoneTasks = new ArrayList<Task>(tuedayTasks);
tuesdayNonphoneTasks.removeAll (phoneTasks);
assert tuesdayNonphoneTasks.toString().equals("[code db, code gui]");
```

Mehr Elemente entfernen

Telefonanrufe am Dienstag

```
Collection<Task> tuesdayPhoneTasks = new ArrayList<Task>(tuedayTasks);
tuesdayPhoneTasks.retainAll (phoneTasks);
assert tuesdayPhoneTasks.toString().equals("[phone Mike]");
```

Telefonanrufe am Dienstag, Version 2

```
Collection<PhoneTask> tuesdayPhoneTasks2 =
  new ArrayList<PhoneTask> (phoneTasks);
tuesdayPhoneTasks2.retainAll (tuesdayTasks);
assert tuesdayPhoneTasks2.toString().equals("[phone Mike]");
```

In Version 2 wird ausgenutzt, dass das Argument von `retainAll` den Typ `Collection<?>` hat.

Abfragen des Inhalts einer Collection

```
assert tuesdayPhoneTasks.contains(paulPhone);
assert tuesdayTasks.containsAll(tuesdayPhoneTasks);
assert mondayTasks.isEmpty();
assert mondayTasks.size() == 0;
```

Iteratoren, Foreach und Löschen

Iteratoren, Foreach und Löschen

- Rumpf von `foreach` darf die unterliegende Collection nicht ändern:

```
// throws ConcurrentModificationException
for (Task t : tuesdayTasks) {
  if (t instanceof PhoneTask) { tuesdayTasks.remove(t); }
}
```

Iteratoren, Foreach und Löschen

- Rumpf von `foreach` darf die unterliegende Collection nicht ändern:

```
// throws ConcurrentModificationException
for (Task t : tuesdayTasks) {
  if (t instanceof PhoneTask) { tuesdayTasks.remove(t); }
}
```

- Auch das Auflösen des `foreach` hilft nicht:

```
// throws ConcurrentModificationException
for (Iterator<Task> it = tuesdayTasks.iterator() ; it.hasNext() ; ) {
  Task t = it.next();
  if (t instanceof PhoneTask) { tuesdayTasks.remove(t); }
}
```

Iteratoren, Foreach und Löschen

- Rumpf von `foreach` darf die unterliegende Collection nicht ändern:

```
// throws ConcurrentModificationException
for (Task t : tuesdayTasks) {
  if (t instanceof PhoneTask) { tuesdayTasks.remove(t); }
}
```

- Auch das Auflösen des `foreach` hilft nicht:

```
// throws ConcurrentModificationException
for (Iterator<Task> it = tuesdayTasks.iterator() ; it.hasNext() ; ) {
  Task t = it.next();
  if (t instanceof PhoneTask) { tuesdayTasks.remove(t); }
}
```

- **Während ein Iterator aktiv ist, dürfen Veränderungen an der Collection nur über den Iterator selbst vorgenommen werden.**

Iteratoren, Foreach und Löschen

Korrekte Verwendung

```
for (Iterator<Task> it = tuesdayTasks.iterator() ; it.hasNext() ; ) {
  Task t = it.next();
  if (t instanceof PhoneTask) {
    it.remove();
  }
}
```

- **Verwende `remove` Methode aus dem `Iterator`-Interface.**

Beispiel: Verschmelzen von Aufgabenlisten

Aufgabenlisten sollen immer sortiert sein. Wenn zwei Aufgabenlisten verschmolzen werden, soll das Ergebnis wieder sortiert sein.

Annahme dabei: die Aufgabenlisten enthalten nicht `null`.

Beispiel: Verschmelzen von Aufgabenlisten

Hilfsdefinition

```
static <E> E getNextElement(Iterator<E> itr) {
  if (itr.hasNext()){
    E nextElement = itr.next();
    if (nextElement == null) {
      throw new NullPointerException();
    } else {
      return nextElement;
    }
  } else {
    return null;
  }
}
```

- Signalisiert durch `null` das Ende des Durchlaufs.

Beispiel: Verschmelzen von Aufgabenlisten

Das eigentliche Verschmelzen (vereinfacht)

```
static <T extends Comparable<T>>
List<T> merge(Collection<T> c1, Collection<T> c2) {
  List<T> mergedList = new ArrayList<T>();

  Iterator<T> itr1 = c1.iterator();
  Iterator<T> itr2 = c2.iterator();

  T c1Element = getNextElement(itr1);
  T c2Element = getNextElement(itr2);
```

Beispiel: Verschmelzen von Aufgabenlisten

Das eigentliche Verschmelzen (vereinfacht, Teil 2)

```
    // each iteration will take a task from one of the iterators;
    // continue until neither iterator has any further tasks
    while (c1Element != null || c2Element != null) {

      // use the current c1 element if either the current c2
      // element is null, or both are non-null and the c1 element
      // precedes the c2 element in the natural order
      boolean useC1Element = c2Element == null ||
              c1Element != null && c1Element.compareTo(c2Element) < 0;

      if (useC1Element) {
        mergedList.add(c1Element);
        c1Element = getNextElement(itr1);
      } else {
        mergedList.add(c2Element);
        c2Element = getNextElement(itr2);
      }
    }
    return mergedList;
  }
```

Ausblick

- Sets
- Lists
- Maps