Informatik B - Objektorientierte Programmierung in Java

Vorlesung 07:
Collections 1

Inhalt

- Collections
  - Rückblick
  - Überblick
  - Basisinterface **`Collection`**
  - Collections des Typs **`List`**
  - Iteratoren
  - Exkurs: Interface Enumeration
  - Exkurs: Innere Klassen
  - Beispiele

Rückblick: Inf A - Algorithmen

- Abstrakte Datentypen
  - Liste
    - VerweisListe
  - Keller
    - VerweisKeller
    - Implementation mit einem Array
  - Schlange
    - ArraySchlange
  - Baum
    - VerweisBaum
  - Suchbaum
  - Hashing

Überblick

- Collections
  - Collections sind Klassen, die „Behälter“ realisieren, um „beliebig viele“ Objekte ablegen und bei Bedarf wieder herausholen zu können.
  - Die Collections in der Java-Klassenbibliothek implementieren viele in der Informatik gebräuchliche Datenstrukturen wie z.B. Listen, Hash-Tabellen, Bäume etc.
  - Die Collections implementieren diese Datenstrukturen im Sinne von abstrakten Datentypen, d.h. ihre Eigenschaften sind durch die zur Verfügung gestellten Operationen definiert.

Überblick

- Collections
  - Die interne Realisierung der einzelnen Collections wird vor dem Benutzer (Programmierer) weitestgehend verborgen.
  - Die Daten werden gekapselt abgelegt, und der Zugriff ist nur mit Hilfe vorgegebener Operationen möglich.
  - Collections können Referenzen auf Objekte beliebiger Klassen speichern.
  - Alle Collection-Klassen sind im Paket **`java.util`** enthalten.

Überblick

- Historische Entwicklung
  - Die Collection-Klassen wurden in zwei Schritten in die Java-Klassenbibiliothek aufgenommen.
  - Bereits seit dem JDK 1.0 existieren die „traditionellen“ Collection-Klassen **`Vector`**, **`Stack`**, **`HashTable`**, **`Properties`** und **`BitSet`**.
  - Seit dem JDK 1.2 ist eine Sammlung von gut 20 Klassen und Interfaces hinzugekommen, die auch als „Collection-Framework“ bezeichnet werden.
  - Aus diesem Grund enthält die Klassenbibliothek z.T. Klassen mit ähnlichen Namen und ähnlicher Funktionalität (Operationen) aber Unterschieden im Detail.

Überblick

- Unterschiede
  - Die wichtigen Operationen der traditionellen Collections sind synchronisiert, d.h. auf eine Collection kann von mehreren parallel ablaufenden Threads (vereinfacht betrachtet ist ein Thread die Ausführung eines Programmstücks) zugegriffen werden, ohne dass es zu inkonsistenten Daten kommt. Dies beeinträchtigt jedoch die Geschwindigkeit, mit der die Operationen ausgeführt werden.
  - Die Operationen des Collection-Frameworks sind durchweg unsynchronisiert.

Überblick

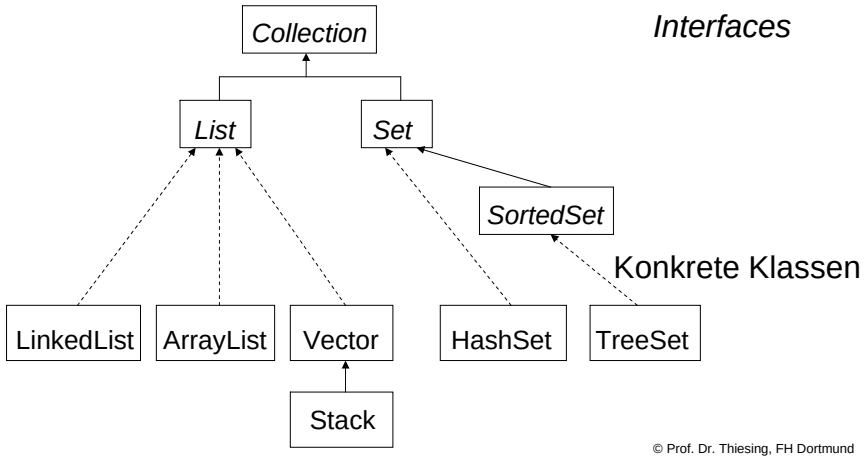
- Grundformen der JDK-1.2 Collections
  - **`List`**: beliebig große lineare Liste von Elementen beliebigen Typs, auf die sowohl sequentiell als auch wahlfrei (über einen Index) zugegriffen werden kann
  - **`Set`**: Ansammlung von Elementen, auf die mit typischen Mengenoperationen zugegriffen werden kann und die keine Elemente doppelt enthält
  - **`Map`**: Abbildung von Elementen eines Typs auf Elemente eines anderen Typs, also eine Menge zusammengehöriger Paare von Objekten

Überblick

- Grundformen der JDK-1.2 Collections
  - Jede der drei Grundformen ist als Interface unter dem oben genannten Namen in der Java-Klassenbibliothek vorhanden.
  - Zu jedem Interface existieren eine oder mehrere abstrakte Klassen, mit deren Hilfe das Erstellen eigener Collection-Klassen erleichtert wird.
  - Zudem gibt es für jedes Interface eine oder mehrere konkrete Implementierungen, die sich in den verwendeten Datenstrukturen und Algorithmen unterscheiden.

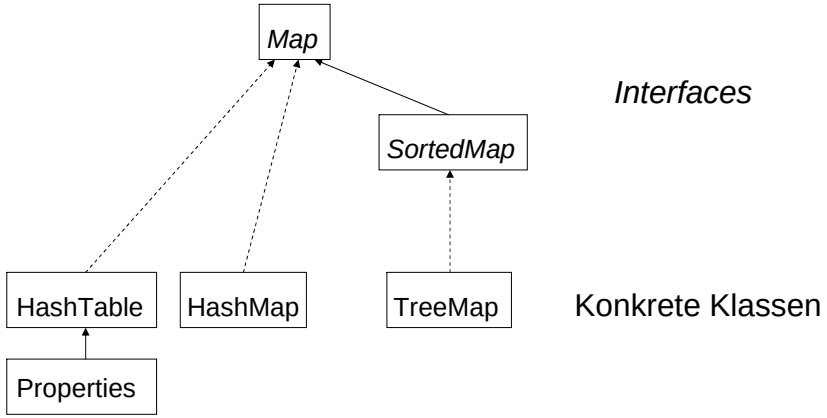
Überblick

- Klassenhierarchie



Überblick

- Klassenhierarchie

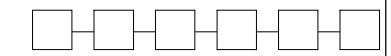


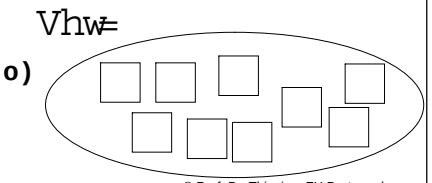
Basisinterface Collection

- Das BasisInterface **`Collection`** fasst die wesentlichen Operationen der Collections vom Typ **`Set`** und **`List`** zusammen:

- Einfügen von Elementen
  - Einfügen eines Objektes
    - **`boolean add(Object o)`**
- Entfernen von Elementen
  - Entfernen eines Objekts
    - **`boolean remove(Object o)`**
  - Entfernen aller Elemente
    - **`void clear()`**

List:





Basisinterface Collection

- Abfragen
  - Überprüfung, ob eine Collection leer ist
    - **`boolean isEmpty()`**
  - Bestimmung der Anzahl der gespeicherten Elemente
    - **`int size()`**
  - Überprüfung, ob eine Collection ein bestimmtes Objekt **`o`** enthält
    - **`boolean contains(Object o)`**
    - Beachte: Die Operation **`contains`** ruft intern die Operation **`equals`** des übergegebenen Objekts auf, um festzustellen, ob eines der Elemente der Collection zu **`o`** inhaltsgleich ist.

Basisinterface Collection

- Umwandlung
  - Speicherung der Elemente einer Collection in ein neues Feld
    - **`Object[] toArray()`**
- Zugriff auf Elemente
  - Erzeugung eines Iterators für den sequentiellen Zugriff auf die Elemente
    - **`Iterator iterator()`**

Basisinterface Collection

- Typsicherheit
  - Eine Collection erlaubt die Speicherung von Objekten beliebiger Klassen, denn die Einfügeoperation besitzt einen Parameter vom Typ **`Object`**.
  - Leider ist der Zugriff auf die gespeicherten Objekte damit nicht typsicher.
  - Der Compiler kann nicht wissen, von welchem Typ die Objekte sind, die in einer Collection gespeichert wurden, und geht daher davon aus, dass beim Zugriff auf ein Element der Collection eine Instanz der Klasse **`Object`** geliefert wird.
  - Mit Hilfe des Castings muss diese dann in den ursprünglichen Objekttyp zurück-verwandelt werden.

Basisinterface Collection

- Typsicherheit
  - Die Verantwortung für die korrekte Typisierung liegt damit beim Programmierer. Beispiel:

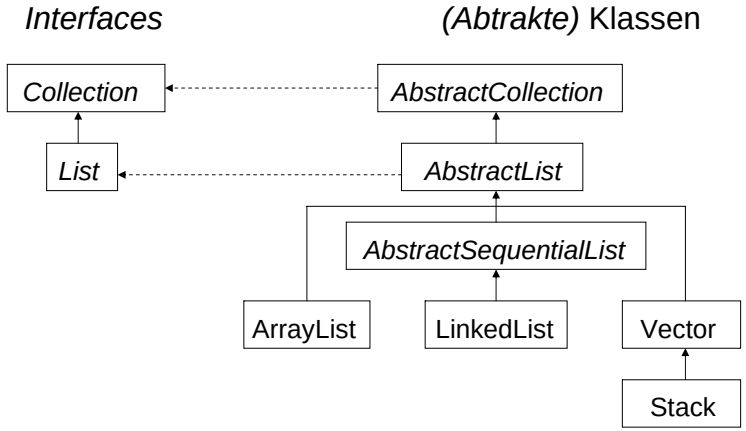
```
List list; ...
ListIterator it = list.listIterator();
String s = (String) it.next();
```

  - Mit Hilfe des Operators **`instanceof`** kann bei Bedarf zur Laufzeit überprüft werden, von welchem Typ ein Objekt ist. Beispiel:

```
Object o = it.next();
if (o instanceof String) s = (String) o;
```

Collections des Typs List

- Klassenhierarchie



Collections des Typs List

- Eigenschaften
  - Eine Collection vom Typ **`List`** ist eine geordnete Folge von Objekten, die entweder sequentiell oder über ihren Index (d.h. ihre Position in der Liste) zugegriffen werden können.
  - Wie bei einem Feld (Array) hat das erste Element den Index 0 und das letzte Element den Index **`size()`**-1.
  - Es ist möglich, an einer beliebigen Stelle der Liste ein Element einzufügen oder zu löschen.

Collections des Typs List

- Interface **`List`**
  - Das Interface **`List`** erbt alle Operationsdeklarationen des Interface **`Collection`**.
  - Das Interface **`List`** präzisiert die Operationen **`add`** und **`remove`** gegenüber **`Collection`** wie folgt:
    - **`boolean add(Object element)`**
      fügt ein neues Element am Ende der Liste ein
    - **`boolean remove(Object o)`**
      entfernt das (im Sinne eines sequentiellen Durchlaufs) erste Element der Liste, das mit o inhaltsgleich ist. Wurde ein Element entfernt, ist der Ergebniswert **`true`**, ansonsten **`false`**.

Collections des Typs List

- Zusätzlich zu den Operationen von **`Collection`** definiert **`List`** u.a. die folgenden Operationen:
  - **`void add(int index, Object element)`**
    fügt ein neues Element an der Position **`index`** ein und schiebt alle nachfolgenden Elemente eine Position weiter.
  - **`Object remove(int index)`**
    entfernt das Element an der Position **`index`** und schiebt alle nachfolgenden Elemente eine Position nach vorne.
  - **`Object set(int index, Object element)`**
    ersetzt das Element an der Position **`index`** durch das Objekt **`element`**
  - **`Object get(int index)`**
    liefert das Element an der Position **`index`** als Ergebnis

Collections des Typs List

- **`int indexOf(Object o)`**
  liefert die Position des ersten Elements der Liste, das mit **`o`** inhaltsgleich ist oder das gleich **`null`** ist, wenn **`o`** gleich **`null`** ist. Wird kein Element gefunden, ist das Ergebnis –1.
- **`int lastIndexOf(Object o)`**
  liefert die Position des letzten Elements der Liste, das mit **`o`** inhaltsgleich ist oder das gleich **`null`** ist, wenn **`o`** gleich **`null`** ist. Wird kein Element gefunden, ist das Ergebnis –1.
- **`List subList(int fromIndex, int toIndex)`**
  erzeugt eine Teilliste von der Position **`fromIndex`** (einschließlich) bis zur Position **`toIndex`** (ausschließlich)
  Beachte: Änderungen, die in der Teilliste vorgenommen werden, sind auch in der Gesamtliste wirksam.

Collections des Typs List

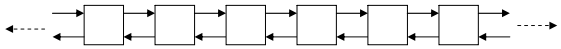
- **`ListIterator listIterator()`**
  erzeugt ein **`ListIterator`**-Objekt, mit dessen Hilfe die Liste beginnend am Anfang der Liste sequentiell durchlaufen werden kann.
- **`ListIterator listIterator(int index)`**
  erzeugt ein **`ListIterator`**-Objekt, mit dessen Hilfe die Liste beginnend an der Position **`index`** sequentiell durchlaufen werden kann.

➢ Anmerkungen
- Alle Positionen **`index`** müssen jeweils größer gleich 0 und kleiner **`size()`** sein, sonst wird eine Exception ausgelöst.
- Bei den Operationen **`add`** und **`listIterator`** bzw. **`subList`** darf die Position **`index`** bzw. **`toIndex`** auch gleich **`size()`** sein.

Collections des Typs List

■ Implementierungen
➢ Klasse **`LinkedList`**:
- realisiert eine Liste, deren Elemente als doppelt verkettete lineare Liste gehalten werden
- Im Vergleich zu **`ArrayList`** und **`Vector`** ist der wahlfreie Zugriff langsamer, aber die Einfüge- und Löschoperationen sind insbesondere bei großen Elementzahlen schneller.

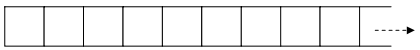
LinkedList:



Collections des Typs List

■ Implementierungen (2)
➢ Klasse **`ArrayList`**:
- implementiert die Liste als Feld von Elementen, das bei Bedarf vergrößert wird
- Im Vergleich zur **`LinkedList`** ist der wahlfreie Zugriff schneller, aber das Einfügen und Löschen kann insbesondere bei großen Elementzahlen länger dauern.
- Wird die interne Kapazität des Feldes überschritten, muss das Feld umkopiert werden.

ArrayList:

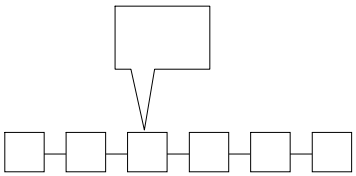


Collections des Typs List

- Klasse **`Vector`**
  - Eigenschaften, s. **`ArrayList`** (außer Synchronisation)
  - Aus Gründen der Vereinheitlichung implementiert seit dem JDK 1.2 auch die Klasse **`Vector`** das **`List`**-Interface, d.h. sie stellt alle im **`List`**-Interface definierten Operationen zur Verfügung.
  - Daneben existieren noch „alte“ Operationen, wie z.B: **`addElement`**, **`insertElementAt`**, **`removeElement`**, **`removeElementAt`**, **`elementAt`** und eine eigenes Interface für Iteratoren mit dem Namen **`Enumeration`**.
  - Beachte: Die wichtigen Operationen der Klasse **`Vector`** sind synchronisiert.
- Alle drei Klassen besitzen einen Konstruktor ohne Parameter.

Iteratoren

- Ein Iterator ist eine Abstraktion für den sequentiellen Zugriff auf alle Elemente einer Collection.
- Ein Iterator für Collection-Klassen wird in Java durch das Interface **`Iterator`** definiert.
- Das Interface **`Iterator`** deklariert die folgenden Operationen:
  - **`boolean hasNext()`**
  - **`Object next()`**
  - **`void remove()`**



Iteratoren

- Sequentieller Durchlauf
  - Nach der Initialisierung zeigt ein Iterator-Objekt auf eine Stelle vor dem ersten Element der Collection.
  - Durch Aufruf von **`hasNext`** kann geprüft werden, ob weitere Elemente enthalten sind.
  - Die Operation **`next`** liefert das nächste Element bzw. löst eine Ausnahme des Typs **`NoSuchElementException`** aus, wenn es keine weiteren Elemente gibt.

Iteratoren

- Modifikation der Collection
  - Zusätzlich besteht die Möglichkeit, die Collection während des sequentiellen Durchlaufs zu verändern, indem das zuletzt geholte Element mit der Operation
    `void remove()`
    gelöscht wird.
  - Dies ist die einzig erlaubte Möglichkeit, die Collection während der Verwendung eines Iterators zu verändern.
  - Alle direkt ändernden Zugriffe auf die Collection machen das weitere Verhalten des Iterators undefiniert.

Iteratoren

- Für Collections des Typs **`List`** steht das aus **`Iterator`** abgeleitete Interface **`ListIterator`** zur Verfügung, welches zusätzliche Operationen definiert:
  - Durchlauf durch die Liste in beiden Richtungen
    - **`boolean hasPrevious()`**
      prüft, ob vor der aktuellen Position ein Element existiert
    - **`Object previous()`**
      liefert das Element vor der aktuellen Position, soweit vorhanden
  - Zugriff auf Index von Elementen
    - **`int nextIndex()`**
      Wird **`nextIndex`** am Ende der Liste aufgerufen, liefert es **`size()`** als Rückgabewert.

Iteratoren

    - **`int previousIndex()`**
      Wird **`previousIndex`** am Anfang der Liste aufgerufen, liefert es -1 als Rückgabewert.
  - Veränderung der Collection
    - **`void add(Object o)`**
      Einfügen eines neuen Elements an der Stelle in die Liste, die unmittelbar vor dem nächsten Element des Iterators liegt.
    - **`void set(Object o)`**
      erlaubt es, das durch den letzten Aufruf von **`next`** bzw. **`previous`** beschaffte Element zu ersetzen.

Iteratoren

- Beispiel
  - ListTest

Die Klasse java.util.Collections

- Diese Klasse stellt eine Reihe von statischen Hilfsfunktionen für Collections zur Verfügung.
- z.B.: sortieren, shuffeln, min oder max liefern, kopieren, Reihenfolge umkehren, Elemente vertauschen, etc.
- **`public static void shuffle(List list)`**
  - Vertauscht die in list enthaltenen Elemente zufällig. Alle möglichen Permutationen sind gleich wahrscheinlich.

Exkurs: Enumeration-Interface

- Idee: Für eine Datenstruktur, die Elemente hält, sollen diese Elemente aufgezählt werden
- Typisch für Collection-Klassen, in denen Objekte gehalten werden
- Sehr ähnlich zum Iterator-Interface
- Vordefiniertes Enumeration-Interface mit zwei Methoden:
  - **`boolean hasMoreElements():`** true, wenn die Aufzählung noch weitere Elemente enthält, sonst false
  - **`Object nextElement()`**: liefert das nächste Element, falls es existiert, sonst wird eine NoSuchElementException ausgelöst

Beispiel: Enumeration-Interface

- MyList.java
- ListEnumerator.java

Exkurs: Innere Klassen

- Innere Klassen seit Java 1.1
- Zur Definition von Hilfsklassen an der Stelle, wo sie gebraucht werden
- Klarerer Code, weniger Quell-Dateien

Innere Klassen im Überblick

- Mitgliedsklassen (Member Classes)
- Statische Mitgliedsklassen (Nested Top-Level-Classes=Geschachtelte Top-Level-Klassen)
- Lokale Klassen
- Anonyme Klassen

Mitgliedsklassen (Member Classes)

- Typische, „echte“ innere Klassen
- Erweiterung der Sprache Java (1.1), aber nicht der JVM
- impliziter Zugriff auf Instanz der einschließenden Klasse
- Ein Objekt der inneren Mitgliedsklasse kann nur zusammen mit einem Objekt der umschließenden Klasse existieren.
- Typische Anwendung: Mitgliedsklassen **`Entry`**, **`Iterator`** der Collections-Klassen.

Beispiel: Mitgliedsklasse

```
public class Hi {
  static String a = "A";
  private String b = "B";
  class B { // Elementklasse B
      void hi () {
            System.out.print("member: " + a + b);
            System.out.println(Hi.this.a + Hi.this.b);
            /* dies ist die explizite Form */
      }
  }
  void hi() { ...
      new B().hi(); // fuehrt hi() der Klasse B aus
  ...}
}
```


Geschachtelte Top-Level-Klassen

- **`static`**-Modifikator
- Verhalten wie normale Top-Level-Klasse
- Haben Zugriff auf alle statischen Komponenten der umschließenden Klasse.
- Statische innere Klassen dienen vor allem der Strukturierung von Programmcode.

Beispiel: Statische Mitgliedsklasse

```
public class Hi {
  static String a = "A";
  private String b = "B";

  static class A {
  // geschachtelte Top-Level-Klasse A
      void hi () {
            System.out.println("top-level: " + a);
            // System.out.println(b);
            /* geht nicht: b ist nicht static! */
      }
  }

  static public void main (String args []) {
  new Hi.A().hi();  // fuehrt hi() der Klasse A aus
  }
}
```

Name der Klasse: Hi.A

Lokale Klassen

- definiert im Java-Block, nur dort sichtbar
- Wie Member-Klassen: Zugriff auf alle Komponenten der umschließenden Klasse. Zusätzlich auf alle im Block sichtbaren **`final`** Parameter und Variablen.

Beispiel: Lokale Klasse

```
public class Hi {
  static String a = "A";
  private String b = "B";
  void hi (final String c) {
      final String d = "D";
      String e = "E";
      class C { // lokale Klasse C
            void hi () {
            System.out.println("lokal: " + a + b + c + d);
            // System.out.println(e);
            // geht nicht: e ist nicht final!
            }
      }
      new C().hi(); // fuehrt hi() der Klasse C aus
  }
}
```


Anonyme Klassen

- weitgend wie lokale Klasse
- Deklaration und Instantiierung in einer Anweisung
- kein Name, kein Konstruktor, aber Instanzinitialisierer
- Typische Anwendung: Implementierung von Event-Listenern bei GUI (AWT, Swing)

Beispiel: Anonyme Klasse

```
public class Hi {...
  static String a = "A";
  private String b = "B";
  void hi (final String c) {
      final String d = "D";
      Object o = new B() {
      // anonyme Klasse ist Subklasse von Klasse B
      // (oder implementiert das Interface B)
      { System.out.println("Instanzinitialisierer der
      anonymen Klasse");
      }
      void hi ()
            { System.out.println("anonym: " +a+b+c+d);
            }
      };
      ((B)o).hi();  // fuehrt hi() der anonymen Klasse aus
  }
}
```

Das Beispiel im Überblick

- Source: Hi.java
- Das Übersetzen des Programms Hi.java erzeugt die Class-Dateien:
  - Hi.class
  - Hi$A.class
  - Hi$B.class
  - Hi$1.class
  - Hi$1C.class.