TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa / Arnd Poetzsch-Heffter

AG Softech
FB Informatik
TU Kaiserslautern

Überblick

- Parametrisierte Datentypen mit Java Generics
- Java Collections Framework
  - Parametrisierte Container-Datentypen
  - Iteratoren
  - foreach-Anweisung

Parametrisierte Datentypen

Motivation: Parametrische Polymorphie

- Generische Klassen, Interfaces und Methoden erlauben die Abstraktion von den konkreten Typen der Objekte, die in Instanzvariablen und lokalen Variablen gespeichert werden oder als Parameter übergeben werden.
- Hauptanwendungsbereich: Containerklassen (Collections)

Beispiel: Listen mit Strings I

```
interface ListOfString {
  // Haengt ein Element an das Ende der Liste an
  void add(String element);

  // Liefert das Element an Position index
  String get(int index);

  // Anzahl der Elemente
  int size();
}
```

Beispiel: Listen mit Strings II

```
public class LinkedListOfString implements ListOfString {
  private NodeOfString first;
  private int size;

  public void add(String value) {
    NodeOfString newNode = new NodeOfString(value, null);
    if (first == null) {
      first = newNode;
    } else {
      NodeOfString n = first;
      while (n.getNext() != null) {
        n = n.getNext();
      }
      n.setNext(newNode);
    }
    size++;
  }
  // etc.
}
```

Beispiel: Listen mit Strings III

```
class NodeOfString {
    private String value;
    private NodeOfString next;

    NodeOfString(String value, NodeOfString next) {
        this.value = value;
        this.next = next;
    }
    String getValue() {
      return value;
    }
    NodeOfString getNext() {
      return next;
    }
    void setNext(NodeOfString n) {
      next = n;
    }
}
```

Beispiel: Listen mit Strings IV

```
// Beispiel zur Verwendung

public static void main(String[] args) {
  LinkedListOfString l = new LinkedListOfString();
  l.add("Eine");
  l.add("Liste");
  l.add("mit");
  l.add("Strings!");

  for (int i = 0; i < l.size(); i++) {
    System.out.print(l.get(i) + " ");
  }
  System.out.println();
}
```

Beispiel: Generische Listen I

```
public interface List<T> {
  // Haengt ein Element an das Ende der Liste an
  void add(T element);

  // Liefert das Element an Position index
  T get(int index);

  // Anzahl der Elemente
  int size();
}
```

Beispiel: Generische Listen II

```
public class LinkedList<T> implements List<T> {
  private Node<T> first;
  private int size;

  // fuegt ein Element am Ende der Liste ein
  public void add(T value) {
    Node<T> newNode = new Node<T>(value, null);
    if (first == null) {
      first = newNode;
    } else {
      Node<T> n = first;
      while (n.getNext() != null) {
        n = n.getNext();
      }
      n.setNext(newNode);
    }
    size++;
  }
```

Beispiel: Generische Listen III

```
public T get(int index) {
  Node<T> n = first;
  int i = 0;
  while (i < index) {
    n = n.getNext();
    i++;
  }
  return n.getValue();
}
```

Beispiel: Generische Listen IV

```
class Node<T> {
    private T value;
    private Node<T> next;

    Node(T value, Node<T> next) {
        this.value = value;
        this.next = next;
    }
    T getValue() {
      return value;
    }
    Node<T> getNext() {
      return next;
    }
    void setNext(Node<T> n) {
      next = n;
    }
}
```

Beispiel: Generische Listen V

```
public static void main(String[] args) {
    LinkedList<String> l = new LinkedList<String>();
    l.add("Eine");
    l.add("Liste");
    l.add("mit");
    l.add("Strings!");

    for (int i = 0; i < l.size(); i++) {
      System.out.print(l.get(i) + " ");
    }
    System.out.println();
}
```

Begriffsklärung

- `List<T>` ist ein **generisches Interface**.
- `LinkedList<T>` und `Node<T>` sind generische Klassen.
- `T` ist eine Typvariable, die für einen beliebigen Referenztyp steht.

Vorteile

- Keine Codeduplikation, um Instanzen für bestimmte Typen zu haben
- Vermeidet evtl. Laufzeitprüfungen bei homogenen Datenstrukturen (effizienter) und bietet mehr statische Prüfbarkeit

Beispiel: Generisches Paar

```
// Repraesentiert ein generisches 2-Tupel
public class Pair<X,Y> {
  private X fst;
  private Y snd;

  public Pair(X fst, Y snd) {
    this.fst = fst;
    this.snd = snd;
  }

  public X getFirst() {
    return fst;
  }

  public Y getSecond() {
    return snd;
  }
}
```

Problem: Instantiierung mit Basisdatentypen

- Typvariablen können nur für Referenztypen stehen!
- Bei der Instantierung mit Basisdatentypen (`int`, `double`, `boolean`, `etc`.) müssen daher **Wrapperklassen** verwendet werden.
- Die Umwandlung von Werten der elementaren Datentypen in Objekte der Wrapper-Klassen nennt man **Boxing**, die umgekehrte Umwandlung **Unboxing**.

Wrapper-Klassen

- Ein Wrapper-Objekt für den elementaren Datentyp D besitzt ein Attribut zur Speicherung von Werten des Typs D.
- Wrapper-Klassen beinhalten auch (statische) Methoden und Attribute zum Umgang mit Werten des zugehörigen Datentyps.
- Javas Wrapper-Klassen sind im Paket `java.lang` definiert und müssen daher nicht importiert werden. Beispiele:

`int`	`java.lang.Integer;`
`double`	`java.lang.Double;`
`boolean`	`java.lang.Boolean;`

Beispiel: Wrapper-Klasse für Integer

`Integer` ist die Wrapper-Klasse für den Typ `int`:

```
static int MAX_VALUE;
static int MIN_VALUE;

Integer (int value)
Integer (String s)

int intValue()
static int parseInt(String s)
```

Anwendungsbeispiel:

```
Integer iv = new Integer(7);
Object  ov = iv;
int     n  = iv.intValue() + 23 ;
```

Autoboxing

Wo nötig führt Java mittlerweile Boxing und Unboxing automatisch durch (**Autoboxing**).
Das folgende Programmfragment mit Autoboxing

```
List<Integer> l = new LinkedList<Integer>();
l.add(1);
int i = l.get(0);
```

ist eine Abkürzung für

```
List<Integer> l = new LinkedList<Integer>();
l.add(new Integer(1));
int i = l.get(0).intValue();
```

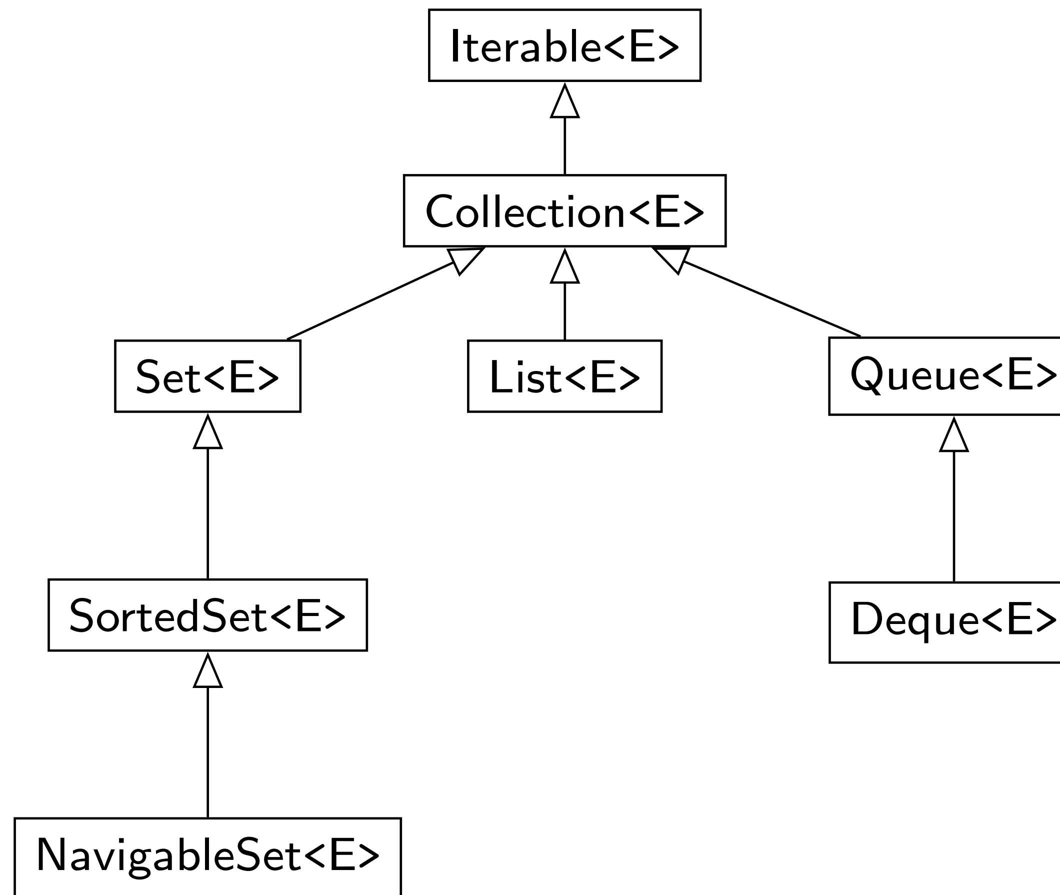
Das Java Collection Framework

Übersicht: Java Collections Framework

- "Collection" ist der Oberbegriff für **Containerdatentypen**, mit denen Ansammlungen von Elementen verwaltet werden
- Typische Operationen dabei sind das Hinzufügen, Entfernen, Suchen, Durchlaufen.
- Hauptinterfaces (in `java.util`):
  - `Collection` enthält die Grundfunktionalität für alle Datentypen des Frameworks *außer für Maps (⇒ nächste Vorlesung)*
  - `Set`: ohne Duplikate, Reihenfolge unwichtig
    Zwei Spezialisierungen: `SortedSet` und `NavigableSet`
  - `Queue`: Warteschlange, FIFO
    Spezialisierung: `Deque` (double ended queue) erlaubt das effiziente Einfügen und Entfernen an beiden Enden
  - `List`: mit Wiederholung, Elemente in fester Reihenfolge
  - `Map` endliche Abbildung, Indexstrukturen
    Spezialisierung: `SortedMap` und `NavigableMap`

Übersicht: Collections

Elementtyp ist `E`



Implementierungen

- Das Java Collection Framework besteht aus *Interfaces*.
- Zu jedem Interface gibt es *mehrere Implementierungen*, basierend auf Arrays, Listen, Bäumen, oder anderen Datenstrukturen.
- Grund: Für jede Datenstruktur sind einige Operationen sehr effizient, dafür sind andere weniger effizient.
- Beispiel: Zugreifen auf Elemente nach Position ist in $O(1)$ bei Arrays und in $O(N)$ bei verketteten Listen
- Auswahl der Implementierung sollte den Anforderungen der Anwendung angepasst sein
- **Programme sollten sich ausschließlich auf die Interfaces beziehen**
- **Einzige Ausnahme**: Erzeugen der Datenstruktur

Das Collection Interface

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

Einfügen von Elementen

```
// Fuegt das Element o ein
boolean add (E o);

// Fuegt alle Elements aus Collection c ein
boolean addAll (Collection<? extends E> c);
```

- Liefern `true`, falls die Operation erfolgreich ist
- Bei Mengen: `false`, falls das Element schon enthalten ist
- Löst Exception aus, falls das Element aus anderem Grund nicht erlaubt
- Das 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

```
// Entfernt Element o
boolean remove (Object o);

// Entfernt alle Elemente
void clear();

// Entfernt alle Elemente in Collection c
boolean removeAll(Collection<?> c);

// Entfernt alle Elemente, die nicht in Collection c
sind
boolean retainAll(Collection<?> c);
```

- Argument von `remove` hat Typ `Object`, nicht `E`
- Argument von `removeAll` bzw. `retainAll` ist Collection mit Elementen von beliebigem Typ
- Rückgabewert ist jeweils `true`, falls die Operation die Collection geändert hat

Testen des Inhalts

```
// true, falls Element o enthalten ist
boolean contains (Object o);

// true, falls alle Elementa aus c enthalten sind
boolean containsAll (Collection<?> c);

// true, falls kein Element enhalten
boolean isEmpty();

// Liefert die Anzahl der Elemente
int size();
```

Alle Elemente verarbeiten

```
// Liefert einen Iterator ueber die Elemente
Iterator<E> iterator();

// Kopiert die Elemente in ein neues Array
Object[] toArray();

// Kopiert die Elemente in ein Array
<T> T[] toArray (T[] a);
```

- Die letzte 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 Methoden, die Arrays als Argument erwarten

Subtyping und generische Klassen

- Für generische Klassen gelten nur deklarierte Subtyp-Beziehungen, d.h. `List<A>` ist beispielsweise ein Subtyp von `Collection<A>`.
- Insbesondere:
  - Falls A Subklasse von B, dann **gilt nicht**, dass `Collection<A>` Subtyp von `Collection<B>` ist.
  - `Collection<A>` und `Collection<B>` haben keinerlei (Vererbungs-) Beziehung zueinander.
  - Gilt analog für alle anderen generischen Klassen.

Wiederholung: Ein Designfehler in Java

- In Java gilt:

  Falls A Subklasse von B, dann ist auch $A[]$ Subtyp von $B[]$

- Dies ist erzwingt spezielle Tests zur Laufzeit!
- Dieser wurde Fehler wurde beim Einführen von Generics in Java vermieden.

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Wo liegt das Problem?

```
class B { }
class A extends B {
  void m() { }
}

class ArrayCheck {
  void test() {
    A[] a = new A[1];
    a[0] = new A();
    a[0].m();
    blub(a);
    a[0].m(); // Problem, da Methode m in B nicht
    vorhanden
  }
  void blub(B[] b) {
    b[0] = new B(); // ArrayStoreException!!
  }
}
```

Wiederholung: Iterator

Iteratoren erlauben es, elementweise über Collections zu laufen, so dass alle Elemente der Reihe nach besucht werden.

```
// Generisches Interface fuer Iteratoren
public interface Iterator<E> {
  // Liefert true, falls weitere Elemente vorhanden
  boolean hasNext();

  // Liefert das naechste Elemente
  E next();

  // optional: Entfernt das letzte Element aus der
    Collection
  // das der Iterator geliefert hat
  void remove();
}
```

Durchlaufen mit Iterator

- (Veraltetes) Muster zum Verarbeiten einer Collection mit Iterator

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

Durchlaufen mit Iterator

- (Veraltetes) Muster zum Verarbeiten einer Collection mit Iterator

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

- Ab Java 5: `foreach`-Anweisung

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

Das Interface `Iterable`

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

```
public interface Iterable<T> {
    Iterator<T> iterator();
}
```

- Dazu zählen:
  - jede `Collection`, da das `Collection`-Interface das `Iterable`-Interface erweitert
  - Arrays
  - beliebige Klassen, die `Iterable` implementieren

Beispiel: Ein Array mit `Iterable` durchlaufen

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

Literaturhinweis

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