TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Lernziele

- Abstrakte Datentypen Stack und Queue zu implementieren und anzuwenden
- Vorteile von parametrischer Polymorphie nennen zu können
- Parametrisierte Datentypen mit Java Generics zu implementieren
- Java Collections Framework mit ihren parametrisierten Container-Datentypen und Iteratoren anzuwenden

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Abstrakte Datentypen

Abstrakte Datentypen

- Ein **abstrakter Datentyp (ADT)** ist eine Menge von Elementen (bzw. Objekten) zusammen mit den Operationen, die für die Elemente der Menge charakteristisch sind.
- ADTs können unabhängig von ihrer konkreten Implementierung in verschiedenen Kontexten eingesetzt werden, einzig basierend auf einer wohldefinierten Schnittstelle.
- Java unterstützt abstrakte Datentypen durch das Interface-Konzept.

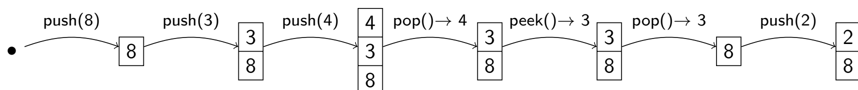
Abstrakter Datentyp: Liste

```
1 interface ListOfInt {
2   // Haengt ein Element an das Ende der Liste an
3   void add(int element);
4   // Liefert das Element an Position index
5   int get(int index);
6   // Anzahl der Elemente
7   int size();
8 }
```

Konkrete Implementierungen des ADTs Liste sind z.B. einfachverkettete Listen, doppeltverkette Listen, Array-Listen.

Abstrakter Datentyp: Stapel (engl. *stack*)

- Basiert auf dem LIFO-Prinzip (*Last-in-First out*)
- Das Element, welches zuletzt eingefügt wurde, wird als erstes wieder entfernt.
- Anwendungsbeispiel:
  - Verwaltung von Hyperlinks im Browser (Back-Button)
  - Verwaltung von Prozedurinkarnationen bei verschachtelten Methodenaufrufen
  - Auswertung von arithmetischen Ausdrücken in polnischer Notation



Schnittstelle von Stapel

```
1  interface StackOfInt {
2
3    // Testet, ob der Stapel leer ist.
4    boolean empty();
5
6    // Legt Element oben auf den Stapel.
7    void push(int item);
8
9    // Gibt das oberste Element vom Stapel zurueck.
10   int peek();
11
12   // Gibt das oberste Element vom Stapel zurueck
13   // und entfernt es vom Stapel.
14   int pop();
15 }
```

Stack-Implementierung mit Arrays

```
1  import java.util.EmptyStackException;
2
3  public class ArrayStackOfInt implements StackOfInt {
4    private int[] elems;
5    private int n;
6
7    public ArrayStackOfInt(int max) {
8      this.elems = new int[max];
9    }
10   public boolean empty() {
11     return n == 0;
12   }
13   public void push(int item) {
14       elems[n] = item;
15       n++;
16   }
17   public int peek() {
18     if (empty()) throw new EmptyStackException();
19     return elems[n-1];
20   }
21   public  int pop() {
22     int result = this.peek();
23     n--;
24     return result;
25   }
26 }
```

Aufgabe

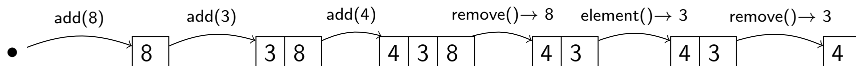
- Wie viele Operationen bzw. einzelne Schritte benötigt man, um ein Element einzufügen bzw. zu löschen?
- Ist diese Anzahl abhängig von der aktuellen Anzahl der Elemente auf dem Stack?
- Was sind die Nachteile dieser Implementierung?

Stack-Implementierung mit einfach verketteten Listen

```
1  import java.util.EmptyStackException;
2
3  public class ListStackOfInt implements StackOfInt {
4    private Node first;
5
6    public ListStackOfInt() { }
7
8    public boolean empty() {
9      return (first == null);
10   }
11   public void push(int item) {
12     Node n = new Node(item, first);
13     first = n;
14   }
15   public int peek() {
16     if (first == null) {
17       throw new EmptyStackException();
18     }
19     return first.getItem();
20   }
21   public int pop() {
22     int result = peek();
23     first = first.getNext();
24     return result;
25   }
26 }
```

Warteschlange (engl. *queue*)

- Basiert auf dem FIFO-Prinzip (*First-in-First-out*)
- Anwendungsbeispiele: Musik-Playlists, Warteschlangen beim Einkaufen, Beantworten von Server-Anfragen, Druckaufträge
- Aspekt der Fairness!



Schnittstelle der Queue

```
1  public interface QueueOfInt {
2    // Fuegt ein Element hinten in die Warteschlange an.
3    void add(int e);
4
5    // Gibt das erste Element in der Warteschlange zurueck.
6    int element();
7
8    // Gibt das erste Element in der Warteschlange zurueck
9    // und entfernt es aus der Warteschlange.
10   int remove();
11
12   // Testet, ob die Warteschlange leer ist.
13   boolean isEmpty();
14 }
```

Queue-Implementierung mit einfachverketteten Listen I

```
1  import java.util.NoSuchElementException;
2
3  public class ListQueueOfInt implements QueueOfInt {
4    private Node first;   // Element mit laengster Verweildauer
5    private Node last;    // Neuestes Element
6
7    // Testet, ob die Warteschlange leer ist.
8    public boolean isEmpty() {
9      return first == null;
10   }
11
12   // Fuegt ein Element hinten in die Warteschlange an.
13   public void add(int e) {
14     Node n = new Node(e,null);
15     if (isEmpty()) {
16       first = n;
17     } else {
18       last.setNext(n);
19     }
20     last = n;
21   }
22
23   // Gibt das erste Element in der Warteschlange zurueck.
24   public int element(){
25     if (isEmpty()) {
```

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Queue-Implementierung mit einfachverketteten Listen II

```
26          throw new NoSuchElementException();
27        }
28        return first.getItem();
29      }
30
31      // Gibt das erste Element in der Warteschlange zurueck
32      // und entfernt es aus der Warteschlange.
33      public int remove(){
34        if (isEmpty()) {
35          throw new NoSuchElementException();
36        }
37        int result = first.getItem();
38        first = first.getNext();
39        if (isEmpty()) {
40          last = null;
41        }
42        return result;
43      }
44    }
```

Ausblick

- Was mache ich, wenn ich eine Liste/Stack/Queue mit Strings oder beliebigen anderen Objekten haben will?
- Weitere Datentypen zum Verwalten von Objekt-Sammlungen: Java Collections Framework

Beispiel: Listen mit Strings I

```
1  interface ListOfString {
2    // Haengt ein Element an das Ende der Liste an
3    void add(String element);
4
5    // Liefert das Element an Position index
6    String get(int index);
7
8    // Anzahl der Elemente
9    int size();
10 }
```

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

```
1  class NodeOfString {
2        private String value;
3        private NodeOfString next;
4
5        NodeOfString(String value, NodeOfString next) {
6            this.value = value;
7            this.next = next;
8        }
9        String getValue() {
10         return value;
11       }
12       NodeOfString getNext() {
13         return next;
14       }
15       void setNext(NodeOfString n) {
16         next = n;
17       }
18 }
```

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)

Typvariablen

- **Typvariablen** sind an den Stellen erlaubt, an denen Typen im Programm verwendet werden.
- Typvariablen müssen vor ihrer Verwendung deklariert (analog zu Variablen, die Werte repräsentieren).
- Die Deklarationsstellen für Typvariablen sind Klassen- bzw. Interface-**Namen**.
- (Sie können auch in Methodensignaturen deklariert werden.)

Beispiel: Generische Listen I

```
1  interface List<T> {
2    // Haengt ein Element an das Ende der Liste an
3    void add(T element);
4    // Liefert das Element an Position index
5    T get(int index);
6    // Anzahl der Elemente
7    int size();
8  }
```

Beispiel: Generische Listen II

```
1  public class LinkedList<T> implements List<T> {
2    private Node<T> first;
3    private int size;
4
5    public void add(T value) {
6      Node<T> newNode = new Node<T>(value, null);
7      if (first == null) {
8        first = newNode;
9      } else {
10       Node<T> n = first;
11       while (n.getNext() != null) {
12         n = n.getNext();
13       }
14       n.setNext(newNode);
15     }
16     size++;
17   }
18
19   public int size() {
20     return size;
21   }
```

Beispiel: Generische Listen III

```
22
23     public T get(int pos) {
24       Node<T> n = first;
25       int i = 0;
26       while (i < pos) {
27         n = n.getNext();
28         i++;
29       }
30       return n.getValue();
31     }
32   }
```

Beispiel: Generische Listen IV

```
1  class Node<T> {
2      private T value;
3      private Node<T> next;
4
5      Node(T value, Node<T> next) {
6          this.value = value;
7          this.next = next;
8      }
9      T getValue() {
10       return value;
11     }
12     Node<T> getNext() {
13       return next;
14     }
15     void setNext(Node<T> n) {
16       next = n;
17     }
18 }
```

Beispiel: Generische Listen V

- Typparameter müssen bei der Objekt-Erzeugung instantiiert werden.
- Objekte haben daher immer einen grundlegenden Typen, d.h. einen Typen ohne Parameter.

```
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();
}
```

Typausdrücke

Durch die Einführung von generischen Typen sind in Java Typen durch **Typausdrücke** repräsentiert.

Ein Typausdruck ist dabei entweder

- ein Typbezeichner (ohne Parameter) oder (z.B. `String`, `Objekt`, `int`)
- eine Typvariable (z.B. `T`, wobei `T` deklariert sein muss) oder
- ein Typkonstruktor angewandt auf Typausdrücke (z.B. `List<T>`, `LinkedList<String>`).

Problem: Instantiierung mit Basisdatentypen

- Typvariablen können nur für Referenztypen stehen!
- Bei der Instantiierung 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`:

```
public class Integer {
static int MAX_VALUE; // Konstante fuer groessten int-Wert
static int MIN_VALUE; // Konstante fuer kleinsten int-Wert

// Konstruktoren fuer Integer-Objekte
Integer (int value) {...}
Integer (String s)  {...}

//liefert den int-Wert fuer ein Integer-Objekt
int intValue()      {...}
// wandelt einen String in ein int um
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**).

Variante mit Autoboxing:

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

Variante ohne Autoboxing:

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

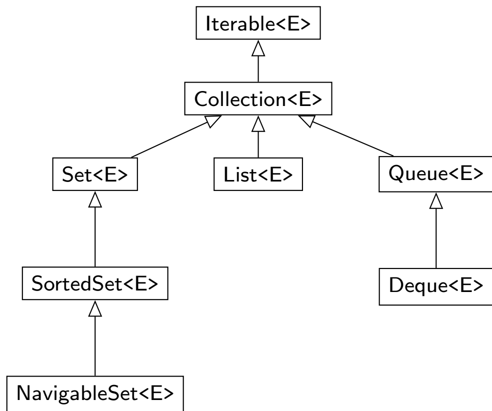
Das Java Collection Framework

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 erfordert nur eine Operation bei Arrays unabhängig von der Größe und mind. N Operationen bei verketteten Listen mit N Elementen
- 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 Elemente aus c enthalten sind
boolean containsAll (Collection<?> c);

// true, falls kein Element enthalten
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: 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