Kapitel 14: Collections und Generics

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

In diesem Kapitel führen wir das Konzept von abstrakten Datentypen ein. Wir haben mit Listen bereits ein Beispiel für einen abstrakten Datentyp kennengelernt. Hier führen zunächst zwei weitere abstrakte Datentypen, Stacks und Queues, als Beispiele für weitere Schnittstellenabstraktion.
Die Implementierungen von Datensammlungen (Collections), die wir bisher betrachtet haben, können nur für einen bestimmten Elementtyp verwendet werden. Wir werden in diesem Abschnitt sehen, wie man eine Collection über den Elementtyp parametrisiert. Das Kapitel schließt mit einer Einführung in das Java Collections Framework, das eine umfangreiche Bibliothek mit Collection-Implementierungen für Java bereitstellt.

Lernziele dieses Kapitels:

- 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

1 Abstrakte Datentypen

Ein ***abstrakter Datentyp (ADT)*** ist i.A. 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. Die Spezifikation der Operationen und der dabei verwendeten Datentypen ist durch die Signaturen der Methoden gegeben, die das Interface vorgibt.
Die Semantik der ADTs kann entweder in einer formale Spezifikationssprache auf mathematisch-axiomatsiche Weise definiert werden oder zumindest informell durch Kommentare beschrieben werden. Wir wählen für diese Vorlesung letztere Variante.

1.1 Abstrakter Datentyp: Liste

Wir haben bereits mehrere Implementierungen von Listen diskutiert: einfachverkettete Listen, doppeltverkettete Listen, Array-Listen. All diese Implementierung haben folgendes gemeinsames Verhalten:

- Die Elemente der Liste sind in einer Reihenfolge geordnet.
- Elemente können in der Liste mehrfach vorkommen.
- Elemente können der Liste am Ende hinzugefügt werden (bisweilen haben wir auch weitere Einfügeoperationen implementiert).

Aus diesen Beobachtungen können wir nun folgende Listen-Schnittstelle für Listen mit `int`-Elementen ableiten:

```
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 }
```

Wenn die konkreten Implementierungen des ADTs, wie die einfach-verkettete Liste oder Array-Liste, dieses Interface implementieren, können die Listen-Implementierungen einfach gegeneinander ausgetauscht werden. Dazu muss lediglich der Konstruktoraufruf geändert werden.

```
ListOfInt l = new IntList();  // alternativ: new IntArrayList(10);
l.add(1);
l.add(43);
l.add(21);

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

1.2 Abstrakter Datentyp: Stapel (engl. stack)

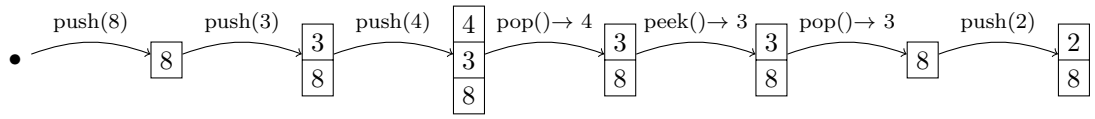
Ein Stapel ist eine Collection, die auf dem LIFO-Prinzip (*Last-in-First out*) basiert: Das Element, welches zuletzt eingefügt wurde, wird als erstes wieder entfernt. Anwendungsbeispiel für Stapel sind das Verwalten von Hyperlinks im Browser (Back-Button) oder die Verwaltung von Prozedurinkarnationen bei verschachtelten Methodenaufrufen.
Das API für Stapel, das `int`-Elemente verwaltet, kann so aussehen:

```
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  }
```

Verwendung von Stacks

```
StackOfInt stack = new ArrayStackOfInt(10); // mit max.
     10 Elementen
stack.push(8);
stack.push(3);
stack.push(4);
StdOut.println(stack.pop());  // 4
StdOut.println(stack.peek()); // 3
StdOut.println(stack.pop());  // 3
stack.push(2);
```



Stack-Implementierung mit Arrays Wie für Listen kann man auch für Stacks verschiedene Varianten implementieren. Hier zuerst die Implementierung basierend auf einem Array:

```
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  }
```

Wir orientieren uns bei unserer Implementierung von `peek()` und `pop()` an der entsprechenden Java-Bibliothek und liefern einen Fehler (`EmptyStackException`), wenn der Stack leer ist. Dazu wird die entsprechende Klasse aus der Bibliothek importiert und mittels `throw new EmptyStackException()` bei leerem Stack ein Fehler ausgelöst. Mehr dazu in Kapitel 17.
Alternativ kann auch als Vorbedingung verlangt werden, dass die beiden Methoden nicht auf einem leeren Stack aufgerufen werden dürfen.

Frage 1:

- Wie viele Operationen benötigt man, um ein Element einzufügen bzw. zu löschen?
- Was sind die Nachteile dieser Implementierung?

Stack-Implementierung mit einfach verketteten Listen Eine weitere einfache Variante erhält man analog zu einer einfach verketteten Liste:

```
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  }
```

1.3 Warteschlange (engl. queue)

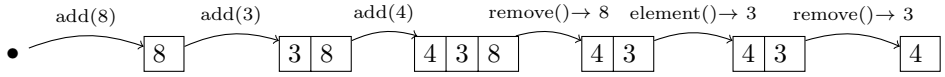
Eine Warteschlange verfolgt eine andere Strategie als der Stack. Sie basiert auf dem FIFO-Prinzip (*First-in-First-out*): Das Element, das als erstes eingefügt wurde, wird als erstes wieder entfernt. Auch hierfür gibt es zahlreiche praktische Anwendungsbeispiele: Musik-Playlists, Warteschlangen beim Einkaufen, Beantworten von Server-Anfragen, Druckaufträge, usw. Warteschlangen kommen oft zum Einsatz, wenn Fairness bei der Problemlösung berücksichtigt werden soll.
Hier die API von Warteschlangen mit `int`-Elementen:

```
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 }
```

Verwendung von Queues

```
QueueOfInt queue = new ListQueueOfInt();
queue.add(8);
queue.add(3);
queue.add(4);
StdOut.println(queue.remove());   // 8
StdOut.println(queue.element()); // 3
StdOut.println(queue.remove());   // 3
```

Die Funktionsweise einer Queue lässt sich folgendermaßen visualisieren:



Wir orientieren uns bei unserer Implementierung von `remove()` und `element()` an der entsprechenden Java-Bibliothek und liefern einen Fehler (`NoSuchElementException`), wenn die Queue leer ist. Dazu wird die entsprechende Klasse aus der Bibliothek importiert und mittels `throw new NoSuchElementException()` bei leerer Queue ein Fehler ausgelöst. Mehr dazu in Kapitel 17.

Queue-Implementierung mit einfachverketteten Listen Auch für die Warteschlangen-API können wir in verschiedenen Varianten implementieren. Hier als verkettete Liste von Einträgen:

```
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()) {
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 }
```

Wir werden weitere Datentypen zum Verwalten von Objekt-Sammlungen in Abschnitt 3 kennenlernen.

2 Parametrisierte Datentypen

Wir haben bisher hauptsächlich Listen/Stacks/Queues mit `int`-Elementen behandelt. Und wenn ich eine Liste/Stack/Queue mit Strings oder beliebigen anderen Objekten haben will?
Eine Möglichkeit ist es, die Implementierung an den jeweiligen Elementtyp anzupassen. Dabei müssen das Interface, die Knotenklassen, die Datenstruktur selbst und auch die Iteratoren angepasst werden. Hier zum Beispiel eine Implementierung für eine Liste mit String-Elementen. Zunächst die Interface-Definition:

```
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();
}
```

Die Implementierung der einfachverketteten Liste muss ebenfalls an den Elementtyp angepasst werden ebenso wie die `Node`-Klasse:

```
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;
    }
}
```

```
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();
```

```
13          }
14          n.setNext(newNode);
15        }
16        size++;
17      }
18
19      public int size() {
20        return size;
21      }
22
23      public String get(int pos) {
24        NodeOfString n = first;
25        int i = 0;
26        while (i < pos) {
27          n = n.getNext();
28          i++;
29        }
30        return n.getValue();
31      }
32    }
```

Der folgenden Code zeigt die Verwendung der String-Liste:

```
ListOfString 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();
```

Will man nun eine Liste für andere Objekte haben, müsste wiederum eine entsprechende Implementierung erstellt werden. Dies ist sehr umständlich: Alle Änderungen (z.B. Hinzufügen weiterer Methoden) müssen für alle Varianten durchgeführt werden.

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. Sie sind ein typisches Beispiel für ***parametrische Polymorphie***. Diese Art der Polymorphie wird von vielen modernen Programmiersprachen (C++, C#, Java ab Version 5, etc.) zusätzlich zur Subtyp-Polymorphie unterstützt. Ein Hauptanwendungsbereich für parametrische Polymorphie sind dabei Containerklassen, wie Listen, Stack, Queues. Der Vorteil von generischen Typen ist das Vermeiden von Codeduplikation, da es nun nicht mehr notwendig ist Implementierungen für jeden benötigten Typ zu haben.
Die Idee von generischen Typen ist es ***Typvariablen*** an Stelle von Typbezeichnern an den Stellen zu erlauben, an denen Typen im Programm verwendet werden. Typvariablen werden (ähnlich wie Variablen für Werte) vor ihrer Verwendung deklariert. Die Deklarationsstellen für Typvariablen sind Klassen- bzw. Interface-Namen[1].

[1] Sie können auch in Methodensignaturen deklariert werden; diese Möglichkeit behandeln wir in SE1 nicht näher

Beispiel: Generische Listen `List<T>` ist ein ***generisches Interface***. `T` ist eine Typvariable, die für einen beliebigen Referenztyp steht.

```
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();
}
```

`List<T>` ist ein ***generischer/parametrisierter Typ***. `List<_>` ist ein ***Typkonstruktor***.
`LinkedList<T>` und `Node<T>` sind generische Typen, die durch folgende Klassendeklarationen definiert werden:

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

  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++;
  }

  public int size() {
    return size;
  }

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

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

    Node(T value, Node<T> next) {
        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  }
```

Typparameter müssen bei der Objekt-Erzeugung instantiiert werden. Objekte haben daher immer einen grundlegenden Typen, d.h. einen Typen ohne Parameter. Im folgenden Beispiel wird der Typ des durch `l` referenzierten Objekts als `LinkedList<String>` festgelegt; dadurch erhält man ein Listenobjekt mit String-Elementen.

```
List<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();
```

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>`).

2.1 Wrapper-Klassen

Wir haben bei der Verwendung von Generics in Java allerdings ein Problem: Wir können Typvariablen nur mit Referenztypen instantiieren. 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***. Ein Wrapper-Objekt für einen elementaren Datentyp D besitzt dabei 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`[2]:

```
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-Objekte können folgendermaßen verwendet werden:

```
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 Verwendung von Autoboxing:

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

Variante mit expliziter Verwendung von Wrapper-Klassen:

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

3 Das Java Collection Framework

Das Verwalten von Objekt-Sammlungen ist zentral für eine Vielzahl von Programmen. Java stellt Programmierern eine umfangreiche Bibliothek mit verschiedenen Typen zur Verfügung. "Collection" ist der Oberbegriff für ***Containerdatentypen***, mit denen Ansammlungen von Elementen verwaltet werden. Typische Operationen dabei sind

[2] Dokumentation zu `java.lang.Integer`: `https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html`

das Hinzufügen, Entfernen und Suchen von Elementen sowie das Durchlaufen der gesamten Sammlung.

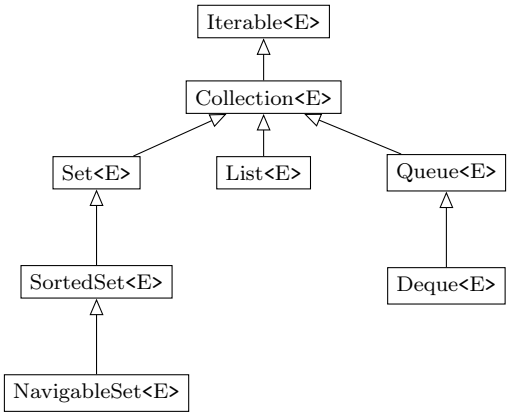
Wir geben in diesem Abschnitt eine kurze Einführung in die wichtigsten Interfaces und Klassen des Collections-Frameworks. Die Dokumentation mit weiteren Details finden Sie unter `https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html`

Eine sehr detaillierte Übersicht finden Sie in:
Maurice Naftalin, Philip Wadler: Java Generics and Collections. O'Reilly, 2006.

Die wichtigsten Schnittstellentypen aus dem Collections-Framework sind (in `java.util`):

- `Iterable`: Enthält Elemente, die mit einem Iterator gelesen werden können.
- `Collection`: Eine allgemeine Ansammlung von Objekten mit Operationen zum Lesen und Verändern.
- `List`: Eine Liste von Elementen.
- `Set`: Eine Menge, d.h. Elemente können nicht mehrfach eingefügt werden und beim Einfügen in die Menge kann keine Position angegeben werden. Die zwei Spezialisierungen `SortedSet` und `NavigableSet` ordnen die Elemente der Menge nach einem bestimmten Kriterium.
- `Queue`: Eine Warteschlange mit Operationen zum Einfügen am Ende der Schlange und Lesen bzw. Entfernen vom Anfang der Schlange. Die Spezialisierung `Deque` (engl. *double ended queue*) erlaubt das effiziente Einfügen und Entfernen an beiden Enden
- `Map`: Eine Abbildung, bei der zu einem Schlüssel jeweils ein Wert gespeichert werden kann. Die Spezialisierung `SortedMap` und `NavigableMap` definieren, analog zu `SortedSet` und `NavigableSet`, eine Ordnung auf den Schlüsseln. (Mehr zu Maps erfahren Sie im nächsten Kapitel.)

Um das Collections-Framework in einer Klasse bzw. Interface zu verwenden, muss die entsprechende Klasse importiert werden (z.B. mittels `import java.util.Set;`). Die folgende Grafik zeigt einen Ausschnitt der Subtyp-Beziehung zwischen den Interfaces des Collection-Frameworks. Dabei bezeichnet `E` den Elementtyp.



Zur Implementierungen des Frameworks Das Java Collection Framework besteht aus *Interfaces*. Zu jedem Interface gibt es *mehrere Implementierungen*, basierend auf Arrays, Listen, Bäumen, oder anderen Datenstrukturen. Für jede Datenstruktur sind einige Operationen sehr effizient, dafür sind andere weniger effizient.
Zum Beispiel ist das Zugreifen auf Elemente nach Position bei `ArrayList`-Objekten in konstanter Zeit möglich, während bei einem `LinkedList` dazu über die Elemente iteriert werden muss, die in der Liste davor stehen (potentiell über alle Elemente). Dafür ist das Einfügen eines Elements am Anfang der Liste bei einer `LinkedList` sehr schnell, bei einer `ArrayList` müssen dazu alle anderen Einträge verschoben werden.
Die Auswahl der Implementierung sollte daher den Anforderungen der Anwendung angepasst sein. In der Regel sollten Programme sich jedoch nur auf die Interfaces beziehen (Ausnahme: Erzeugen der Datenstruktur).

3.1 Das `Iterable` Interface

Die Interfaces `Iterable` und `Iterator` sind in Java (ab Version 7) wie folgt definiert:

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

public interface Iterator<E> {
  boolean hasNext();
  E next()
  void remove();
}
```

Ein Objekt vom Typ `Iterable` erlaubt es nur mit einem `Iterator` die enthaltenen Elemente zu betrachten und eventuell zu entfernen, falls der Iterator die `remove`-Operation unterstützt.

3.2 Das `Collection` Interface

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

Zum Einfügen von Elementen gibt es zwei Möglichkeiten:

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

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

Diese Methoden liefern `true`, falls die Operation den Inhalt der `Collection` ändert. Bei `Set`-Collections ist der Rückgabewert `false`, falls das Element schon enthalten ist, da in einer Menge jedes Element max. einmal enthalten sein kann. Die Methoden lösen eine Exception aus, falls das Element aus anderem Grund nicht erlaubt ist.

Das Argument von `addAll` verwendet den ***Wildcard-Typ*** `?`; d.h. bei Aufruf von `addAll` wird jedes Argument vom Typ `Collection <T>` akzeptiert, falls `T` ein Subtyp von `E` ist

Es gibt vier Varianten Elemente zu löschen:

```
// 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);
```

Der Parameter von `remove` hat dabei den (allgemeineren) Typ `Object`, nicht `E`. Die Methoden `removeAll` bzw. `retainAll` nehmen ebenso als Parameter eine Collection mit

Elementen von beliebigem Typ. Der Rückgabewert dieser Methoden ist jeweils `true`, falls die Operation die Collection geändert hat.
Das Entfernen von Elementen basiert auf deren Gleichheit, das heißt dem Rückgabewert der `equals`-Methode. Für eigene Klassen liefert die `equals`-Methode standardmäßig nur dann `true`, wenn die beiden Objekte die gleiche Identität haben. Das Anpassen der `equals`-Methode durch Überschreiben behandeln wir in einem folgenden Kapitel.
Das folgende Beispiel zeigt, dass es durch die Definition von `equals` zu unerwartetem Verhalten kommen kann.

```
1   public class Punkt {
2     private int x;
3     private int y;
4     public Punkt(int x, int y) {
5       this.x = x;
6       this.y = y;
7     }
8     public String toString() {
9       return "(" + x + ", " + y + ")";
10    }
11  }
12
13  public class PunktTest {
14    @Test
15    public void test() {
16      Collection<Punkt> punkte = new ArrayList<Punkt>();
17      punkte.add(new Punkt(1,2));
18      assertEquals(false, punkte.isEmpty());
19      punkte.remove(new Punkt(1,2));
20      assertEquals(false, punkte.isEmpty());
21    }
22  }
23
```

Der Punkt wird hier in Zeile 19 nicht aus der `Collection` entfernt. Er hat zwar die gleichen Koordinaten wie der zuvor hinzugefügte Punkt. Es handelt sich aber um ein anderes Objekt, weshalb dieser nicht entfernt wird.
Folgende Methoden erlauben es den Inhalt einer Collection zu inspizieren:

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

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

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

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

Um mit allen Elementen in einer Collection zu arbeiten, werden die folgenden Methoden bereitgestellt:

```
// 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 zweite Methode `toArray(T[] a)` kopiert die Elemente der Collection in ein Array mit Elementen von *beliebigem* Typ `T`. Sie liefert einen Laufzeitfehler, falls die Elemente nicht den Typ `T` haben. Wenn im Argumentarray `a` genug Platz ist, wird es für das Kopieren verwendet, sonst wird ein neues Array angelegt.
Die Methode `toArray` wird insbesondere dann verwendet, wenn bestehende Methoden aufgerufen werden müssen, die ein Array erwarten. Im folgenden Beispiel nehmen wir an, dass es schon eine Methode `int gesamtLaengeArray(String[] strings)` gibt, welche die Summe der Längen aller Strings in einem Array berechnet. Wir können diese Methode dann wie folgt mit `toArray` benutzen, um die Gesamtlänge aller Strings in einer Collection zu berechnen:

```
public int gesamtLaenge(Collection<String> strings) {
  String[] stringArray = strings.toArray(new String[strings.size()]);
  return gesamtLaengeArray(stringArray);
}
```

3.3 Iteratoren

Iteratoren erlauben es, elementweise über Collections zu laufen, so dass alle Elemente der Reihe nach besucht werden. Das Collection-Framework nutzt Iteratoren, um die Elemente der Collection zu besuchen. Das generisches Interface für Iteratoren hat dabei folgende Schnittstelle:

```
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();
}
```

Wir können nun Iteratoren verwenden, um Collections zu verarbeiten.

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

Diese Muster gilt seit Java 5 als veraltet, da die Sprachdefinition von Java in Version 5 um eine neue Anweisung erweitert wurde, die für eben dieses eine syntaktische Variante bereitstellt. Die `foreach`-Anweisung liefert eine kurze und elegante Art der Iteration:

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

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 Auch Arrays können mit `Iterable` durchlaufen werden:

```
1 public class Echo {
2     public static void main (String[] arg) {
3       for (String s : arg) {
4         System.out.println(s);
5       }
6   }
7 }
```

3.4 Listen

Listen sind Collections, in denen die Elemente in einer bestimmten Reihenfolge abgelegt werden können. Dementsprechend gibt es neben den Methoden, die durch Implementierung des `Collection`-Interfaces erfordert werden, noch Methoden wie das Einfügen an einer bestimmten Position und Methoden zum Lesen der Elemente an einer bestimmten Position.

```
// Einfuegen an gegebener Position:
void add(int index, E element);
// Fuegt alle Elemente aus c an gegebener Position ein:
boolean addAll(int index, Collection<? extends E> c);
// Ersetzen von Element an Position:
// (gibt altes Element an der Position zurück)
```

```
E set(int index, E element)
// Element an gegebener Position:
E get(int index);
// Position von erstem Vorkommen von Objekt
// (-1 falls nicht enthalten):
int lastIndexOf(Object o)
// Listen-Iteratoren können vorwärts und rückwärts laufen
// und bieten Operationen zum Verändern der Liste
ListIterator<E> listIterator()
// Listen-Iterator mit gegebener Start-Position
ListIterator<E> listIterator(int index)
```

Implementierungen

Die am häufigsten verwendete Implementierung von Listen in Java ist die `ArrayList`. Hier ist der Zugriff auf Elemente an gegebener Position effizient. Für manche Anwendungsfälle eignet sich jedoch die `LinkedList` besser, beispielsweise wenn Elemente häufig hinzugefügt und auch entfernt werden.

Beispiel Wir können Listenobjekte folgendermaßen erzeugen (hier: Listen mit String-Elementen):

```
List<String> a = new ArrayList<String>();
List<String> b = new LinkedList<String>();
```

Die Verwendung der Listenobjekte, die durch `a` und `b` referenziert werden, unterscheiden sich nach dem Aufruf des Konstruktors nicht – die Details der Implementierung sind hinter der gemeinsame Schnittstelle verborgen.
Um ein Array oder eine feste Zahl von Objekten in eine Liste umzuwandeln, gibt es die statische Methode `Arrays.asList()`. Diese erstellt eine Liste, die intern das übergebene Array verwendet und die daher nicht in der Größe verändert werden kann (beim Versuch Elemente hinzuzufügen oder zu entfernen gibt es eine `UnsupportedOperationException`). Die Methode eignet sich gut zur Angabe der erwarteten Liste beim Schreiben von JUnit-Tests.

```
@Test
public void test() {
  List<String> li = new ArrayList<String>();
  li.add("c");
  li.add("d");
  li.add("e");
  assertEquals(Arrays.asList(new String[] {"c", "d", "e"}), li)
  ;
  assertEquals(Arrays.asList("c", "d", "e"), li);
}
```

3.5 Sets

Das Interface `Set` bietet keine weiteren Methoden außer den Methoden von `Collection`. Das besondere an Objekten vom Typ `Set` ist, dass beim mehrmaligen Hinzufügen des gleichen Elements nur ein Eintrag in der Menge entsteht.
Beliebte Implementierungen von `Set` sind `HashSet`, `LinkedHashSet` und `TreeSet`.
`TreeSet` sortiert die Einträge und kann daher nur für Elemente verwendet werden, die sich in Java ordnen lassen (d.h. die Elemente müssen das Interface `Comparable` implementieren oder es muss beim Erstellen des `TreeSet` ein `Comparator` zum Vergleichen angegeben werden).
`HashSet` verwendet die `hashCode()`-Methode von Objekten, welche wir in einem späteren Kapitel betrachten werden. An dieser Stelle ist wichtig zu wissen, dass jedes Objekt eine Standard-Implementierung der `hashCode()`-Methode besitzt und das `HashSet` deshalb für beliebige Elemente benutzt werden kann. Es ist daher eine sehr oft verwendete Implementierung.

Falls die `equals()`-Methode für eine Klasse angepasst wird, muss auch die `hashCode()`-Methode entsprechend angepasst werden. Mehr dazu erfahren Sie im weiteren Verlauf von SE1.

Das `LinkedHashSet` ist ähnlich zum `HashSet`, bietet aber eine schnellere und deterministische Ordnung beim Durchlaufen der Einträge mit einem `Iterator`. Es ist daher in der Regel dem `HashSet` vorzuziehen, wenn der `Iterator` (oder `toArray()`) benutzt wird. In nichtdeterministischen Programmen ist es schwieriger Fehler zu finden.

3.6 Maps

Indexstrukturen (engl. *maps* oder *dictionary*) erlauben eine effiziente Verwaltung von Daten (hier: Objekten). Ein Eintrag in der `Map` besteht aus einem eindeutigen **Schlüssel** und dem ihm zugeordneten **Wert**.
Die Verwaltung von Einträgen basiert auf den folgenden drei Grundoperationen:

- Einfügen eines Eintrags
- Suchen eines Werts zu einem gegebenen Schlüssel K
- Löschen eines Eintrags mit gegebenen Schlüssel K

Das Interface `Map` bietet darüber hinaus auch noch weitere Methoden. Die Typparameter `K` und `V` stehen hierbei für den Typ der Schlüssel (engl. **K**ey) und für den Typ der Werte (engl. **V**alue).

```
interface Map<K, V> {
  // Löscht alle Einträge in der Map
  void clear();
  // Prüft, ob Schlüssel enthalten
  boolean containsKey(Object key);
  // Prüft, ob Wert enthalten (in der Regel langsame Operation)
  boolean containsValue(Object value);
```

```
  // Liefert alle Einträge als Menge
  Set<Map.Entry<K,V>> entrySet();
  // Liefert Wert zu gegebenen Schlüssel
  V get(Object key);
  // Prueft, ob Map leer ist
  boolean isEmpty();
  // Liefert die Menge aller Schlüssel
  Set<K> keySet();
  // Fuegt Eintrag in die Map ein; liefert den alten Wert oder null
  V put(K key, V value);
  // Übertragen von Einträgen von m in diese Map
  void putAll(Map<? extends K,? extends V> m);
  // Entfernen eines Eintrags mit gegebenen Schlüssel; liefert den Wert
  V remove(Object key);
  // Anzahl der Einträge
  int size();
  // Collection der Werte:
  Collection<V> values();

  // Interface fuer Einträge:
  interface Entry<K,V> {
    K getKey();
    V getValue();
    V setValue();
  }
}
```

Wie bei Mengen gibt es hier auch drei häufig benutzte Implementierungen: `HashMap`, `LinkedHashMap` und `TreeMap`. Es gelten die gleichen Hinweise wie für die `Set`-Implementierungen.

Beispiel zur Verwendung In Büchern findet man oft einen "Index" am Ende des Buchs. Der Index ist eine alphabetisch sortierte Liste von Begriffen, in der zu jedem Begriff die Seitennummern stehen, auf denen dieser Begriff vorkommt.
Die Methode `createIndex()` nimmt eine Liste von Wörtern und erstellt einen solchen Index. Dazu wird für jedes Wort eine Menge von Seiten gespeichert, auf denen das Wort vorkommt.

```
1  import static org.junit.Assert.assertEquals;
2  import org.junit.Test;
3  import java.util.*;
4
5  // Worteintraege in einem Dokument bzw. Buch
6  public class Word {
7    private int pageNumber; // Seitenzahl
8    private String word;    // zu indizierendes Wort
9
10   public Word(String word, int pageNumber){
11     this.word = word;
12     this.pageNumber = pageNumber;
13   }
14
15   public String getWord() {
16     return this.word;
17   }
18
```

```
19    public int getPageNumber() {
20      return this.pageNumber;
21    }
22
23    // Erstellt einen Index, d.h. Zuordnung von String auf sortierte Menge
        der Seitenzahlen
24    public static Map<String, SortedSet<Integer>> createIndex(List<Word>
        words) {
25
26      Map<String, SortedSet<Integer>> index = new TreeMap<String, SortedSet
        <Integer>>();
27
28      // Ueber alle Woerter iterieren und in den Index einfuegen:
29      for (Word word : words) {
30          SortedSet<Integer> lines = index.get(word.getWord());
31          if (lines == null) {
32              // Wenn noch keine Zeilen fuer dieses Wort gespeichert sind,
33              // wird ein neues TreeSet angelegt. Durch das TreeSet ist
34              // gegeben, dass die Zeilen-Nummern sortiert
35              // und ohne Duplikate erscheinen
36              lines = new TreeSet<Integer>();
37              index.put(word.getWord(), lines);
38          }
39          lines.add(word.getPageNumber());
40      }
41      return index;
42    }
43  }
```

3.7 Subtyping und generische Klassen

Für generische Klassen gelten nur deklarierte Subtyp-Beziehungen, d.h. `List<A>` ist beispielsweise ein Subtyp von `Collection<A>`. Insbesondere gelten die folgenden Regeln:

- 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.

Dies gilt analog für alle anderen generischen Klassen.

Wiederholung: Subtyping bei Arrays In Java gilt:

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

Dies erzwingt spezielle Tests zur Laufzeit, um zu verhindern, dass Elemente vom "falschen" Typ in ein Array eingetragen werden. Der folgende Code illustriert das Problem mit den Arrays nochmals:

```
1  class B { }
2  class A extends B {
3    void m() { }
```

```
4  }
5
6  public class ArrayCheck {
7    public static void main(String[] args) {
8      A[] a = new A[1];
9      a[0] = new A();
10     a[0].m();
11     replaceElement(a);
12     a[0].m(); // Problem, da Methode m in B nicht vorhanden
13   }
14   public static void replaceElement(B[] b) {
15     b[0] = new B(); // ArrayStoreException!!
16   }
17 }
```

Dieser Fehler wurde beim Einführen von Generics in Java vermieden:

> Falls A Subklasse von B, dann **gilt nicht**, dass `Collection<A>` Subtyp von `Collection<B>` ist.

Hinweise zu den Fragen

Hinweise zu Frage 1:

- Um ein Element einzufügen, wird ein Array-Eintrag geschrieben. Das Entfernen eines Elements erfordert nur ein Dekrement von `n`. Diese Operationen brauchen immer konstant viele Operationen, unabhängig davon, wie viele Einträge der Stack hat.
- Der Stack kann höchstens `max`-viele Elemente aufnehmen. Dieses Problem kann behoben werden, indem man z.B. ein Array mit doppelter Größe erstellt, die Elemente in das neue, größere Array kopiert und das neue Array für den Stack verwendet.