Programmieren in Java

Vorlesung 04: Collection API

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

SS 2017

Inhalt

Codequalität: Immutable Inputs

Immutable = unveränderlich

Codequalität: Immutable Inputs

Immutable = unveränderlich

Java unterscheidet

- primitive Datentypen: bool, byte, int, short, long, float, double, char
  werden direkt als 32 Bit Wort dargestellt (long, double: 64 Bit)
- Referenzdatentypen: Arrays, Objekttypen
  werden durch Referenz (Adresse eines Speicherbereichs) dargestellt

Parameterübergabe

Unterschied bei der Übergabe als Parameter

- Werte von primitiven Datentypen werden kopiert
  - Zuweisungen an Parameter sind nach außen nicht sichtbar
- Für Referenztypen wird die Adresse übergeben,
  - Änderungen am Parameter sind für den Aufrufer (global) sichtbar
  - Oft unerwartet, **muss** dokumentiert werden, falls dieses Verhalten gewünscht ist!

Parameterübergabe

Unterschied bei der Übergabe als Parameter

- Werte von primitiven Datentypen werden kopiert
  - Zuweisungen an Parameter sind nach außen nicht sichtbar
- Für Referenztypen wird die Adresse übergeben,
  - Änderungen am Parameter sind für den Aufrufer (global) sichtbar
  - Oft unerwartet, **muss** dokumentiert werden, falls dieses Verhalten gewünscht ist!

Konvention (ab sofort)

- Parameter von (getesteten) Methoden dürfen nicht verändert werden
- Zuweisungen an Parameter aller Art sind code smells und verboten

Null

- Referenztypen besitzen einen Extra-Wert: `null`
- `null` ist keine gültige Adresse
- Zugriffe auf `null` liefern eine Exception

```
1   int[] ia = null;
2   ia[0] = 1; // yields a NullPointerException
3   ia.length; // yields a NullPointerException
```

- Tony Hoare nennt den `null`-Wert den "Billion Dollar Mistake"
  Null References: The Billion Dollar Mistake

Null

Abhilfe: Conditional mit Test auf `null`

```
1       public static int ref0(int[] ia) {
2         if (ia != null) {
3           // after the test we can assume that ia is a valid address
4           int l = ia.length;
5           //...
6         } else {
7           // fallback code if ia is null
8         }
9       }
```

Null

Abhilfe: Conditional mit Test auf `null`

```
1       public static int ref0(int[] ia) {
2         if (ia != null) {
3           // after the test we can assume that ia is a valid address
4           int l = ia.length;
5           //...
6         } else {
7           // fallback code if ia is null
8         }
9       }
```

Konvention

- Bei jeder Variable vom Referenztyp muss der Programmierer davon ausgehen, dass sie `null` sein kann!
- Vor Verwendung muss explizit getestet werden, ob der Wert ungleich `null` ist!

Aus der Praxis

```
1     public static int secondHighest(int[] numbers){
2   int nrOfnumbers = numbers.length; //Length of inputArray
3   int secondhighest = Integer.MIN_VALUE; //start value for second highest
4   int highest = Integer.MIN_VALUE; //Start value for highest
5
6   for (int i =0; i < nrOfnumbers; i++){    //for loop with die duration of the length of t
7       if (numbers[i]== highest | numbers [i] == secondhighest){ // checking if the curre
8
9       }
10      else if (numbers[i] > highest){ // setting the current array spot as new highest num
11    secondhighest = highest;
12    highest =numbers[i];
13      }else if (numbers[i] > secondhighest){ //setting the current array spot as the new s
14    numbers[i] = secondhighest;
15      }
16
17  }
18  return secondhighest; //returning the second highest
19    }
```

Coding Conventions Applied

```
1  public static int secondHighest(int[] numbers){
2    int nrOfNumbers = numbers.length; //Length of inputArray
3    int secondHighest = Integer.MIN_VALUE; //start value for second highest
4    int highest = Integer.MIN_VALUE; //Start value for highest
5
6    for (int i =0; i < nrOfNumbers; i++) { //for loop with die duration of the length of
7      if (numbers[i] == highest | numbers [i] == secondHighest) {
8        // checking if the current array spot has the same value as secondHighest or high
9      } else if (numbers[i] > highest) { // setting the current array spot as new highest n
10       secondHighest = highest;
11       highest =numbers[i];
12     } else if (numbers[i] > secondHighest) { //setting the current array spot as the new
13       numbers[i] = secondHighest;
14     }
15   }
16   return secondHighest; //returning the second highest
17 }
```

Remaining Code Problems

Weitere Fehler

- Line 1: Javadoc fehlt
- Line 7

```
(numbers[i] == highest | numbers [i] == secondHighest)
```

  Operator | ist bitweises Oder. Es muss ein logisches Oder || sein.

- Line 13

```
numbers[i] = secondHighest;
```

  Zuweisung an die Eingabe führt in diesem Fall zum Versagen des Tests

- Neu: Test ob `numbers != null` fehlt.

Collection API

- Framework zum Arbeiten mit Listen, Mengen, Multimengen, etc von Werten (d.h. Collections:-)
- Bestehend aus
  - Interfaces, die die Operationen beschreiben und
  - Implementierungen, die die Operationen implementieren
- Typische Operationen: Hinzufügen, Entfernen, Suchen, Durchlaufen

Konvention

- Möglichst nur die Interfacetypen verwenden!
- Ausnahme: Erzeugung einer Collection

Listen und Iteratoren

Listen

- Das Interface List<X> ist eine Abstraktion zum Bearbeiten von Sequenzen von Elementen vom Typ X.
- List<X> ist ein *generischer Referenztyp*, bei dem für X ein beliebiger Referenztyp (Klasse, Interface, ...) eingesetzt werden kann.
- Beispiele
  - List<Integer> Liste von Zahlen
  - List<Object> Liste von beliebigen Objekten
  - List<Validation> Liste von **Validation** Objekten

Operationen auf Listen (Auszug aus der Interfacedefinition)

```
1  package java.util;
2
3  public interface List<X> {
4     // add new element at end of list
5     boolean add (X element);
6     // get element by position
7     X get (int index);
8     // nr of elements in list
9     int size();
10    // further methods omitted
11 }
```

- Weitere Methoden in der *Java API Dokumentation*
- Um eine Liste zu erzeugen, muss eine konkrete Implementierung gewählt werden
- Beispiele: **ArrayList**, **LinkedList**, **Stack**, ...
- Unterschiedliche Eigenschaften, Auswahl nach Anwendungsfall

Beispiel: Liste implementiert als LinkedList

```
1  public class ListTest {
2    @Test // JUnit specific: see later
3    public void testList() {
4      List<Integer> il = new LinkedList<Integer>(); // create an empty list
5      assertEquals(0, il.size());
6      il.add(1);
7      assertEquals(1, il.size());
8      il.add(4);
9      assertEquals(2, il.size());
10     il.add(9);
11     assertEquals(3, il.size());
12     assertEquals((int)1, (int)il.get(0));
13     assertEquals((int)4, (int)il.get(1));
14     assertEquals((int)9, (int)il.get(2));
15   }
16 }
```

Beispiel: Liste implementiert als ArrayList

```
1  public class ListTest {
2    @Test // JUnit specific: see later
3    public void testList() {
4      List<Integer> il = new ArrayList<Integer>(); // create an empty list
5      assertEquals(0, il.size());
6      il.add(1);
7      assertEquals(1, il.size());
8      il.add(4);
9      assertEquals(2, il.size());
10     il.add(9);
11     assertEquals(3, il.size());
12     assertEquals((int)1, (int)il.get(0));
13     assertEquals((int)4, (int)il.get(1));
14     assertEquals((int)9, (int)il.get(2));
15   }
16 }
```

Durchlaufen von Listen

- Durchlaufen einer Liste kann mittels get geschehen.
- Erfordert Manipulation von Indexen und der Länge der Liste
- Beispiel (Muster)

```
1  public static int sum(List<Integer> li) {
2    int result = 0;
3    for (int i = 0; i < li.size(); i++) {
4      result += li.get(i);
5    }
6    return result;
7  }
```

Das Interface **Iterable**

Generische Möglichkeit

Durchlaufen mittels Iterator

```
1 public interface Iterable<X> {
2   Iterator<X> iterator()
3 }
```

- Jede Liste kann einen Iterator liefern, mit dem die Liste durchlaufen werden kann.
- Alles was List<X> ist, ist auch Iterable<X>.

Das Interface **Iterator**

```
1 public interface Iterator<X> {
2   // true if there is a next element in the list
3   boolean hasNext();
4   // obtain next element and advance
5   X next();
6   // remove the last element returned by next (optional)
7   void remove();
8 }
```

Codemuster für **Iterator**

```
1  Iterable<X> collection = ...;
2  Iterator<X> iter = collection.iterator();
3  while (iter.hasNext()) {
4    X element = iter.next();
5    // process element
6    if (noLongerNeeded(element)) {
7      iter.remove();
8    }
9  }
```

For-Schleife mit **Iterator**

- Falls Löschen nicht erforderlich ist, kann die explizite Verwendung der **Iterator** Methoden vermieden werden
- Stattdessen: Verwende eine For-Schleife

```
1  Iterable<X> collection = ...;
2  for (X element : collection) {
3    // process element
4  }
```

For-Schleife mit **Iterator**

- Falls Löschen nicht erforderlich ist, kann die explizite Verwendung der **Iterator** Methoden vermieden werden
- Stattdessen: Verwende eine For-Schleife

```
1  Iterable<X> collection = ...;
2  for (X element : collection) {
3    // process element
4  }
```

Beispiel

```
1  public static int sum(List<Integer> li) {
2    int result = 0;
3    for (Integer v : li) {
4      result +=v;
5    }
6    return result;
7  }
```

Abbildungen (Maps)

Abbildung

- "Endliche Abbildung"
- Zuordnung von einem Key zu einem Value
- Auszug aus dem Map Interface

```
1  public interface Map<Key, Value> {
2    // Returns the value to which the specified key is mapped, or
3    // null if this map contains no mapping for the key.
4    Value get(Object key);
5    // Associates the specified value with the specified key in this
6    // map (optional operation). Returns previous value or null if none.
7    Value put (Key key, Value value);
8    // further methods omitted
9  }
```

Verwendung von Maps

- Auswahl einer Implementierung:
  vgl “All Known Implementing Classes:” in der Original Dokumentation
  die wichtigsten sind
  - HashMap (Implementierung mittels Hashing; einfacher zu verwenden)
    und
  - TreeMap (Implementierung mittels Suchbaum; erfordert eine Ordnung auf den Elementen)

Beispiel (Maps)

```
1  public static void mapsTest () {
2    // exchange rates for 1 EUR
3    Map<String, Double> currencyTable = new HashMap<String, Double>();
4    currencyTable.put ("USD", 1.12425);
5    currencyTable.put ("JPY", 125.224);
6    currencyTable.put ("CHF", 1.09150);
7    currencyTable.put ("GBP", 0.863911);
8
9    assertEquals(4, currencyTable.size());
10   assertEquals((double)1.12425, (double)currencyTable.get("USD"));
11   assertEquals((double)0.863911, (double)currencyTable.get("GBP"));
12   assertEquals((double)1.09150, (double)currencyTable.get("CHF"));
13 }
```

Reguläre Ausdrücke

- Java unterstützt matchen von regulären Ausdrücken
- Erklärung dazu findet sich in der Dokumentation von java.util.regex.Pattern
- Typische Verwendung (aus o.g. Dokumentation)

Verwendung von Regulären Ausdrücken

```
1  Pattern p = Pattern.compile("a*b");
2  Matcher m = p.matcher("aaaaab");
3  boolean b = m.matches();
```

- `compile` lohnt sich nur, wenn das Pattern mehrfach zum Matchen verwendet wird
- Einfachere Alternative:

```
1  boolean b = Pattern.matches("a*b", "aaaaab");
```

Die Scanner API beinhaltet Methoden, mit denen Patterns gelesen (next, hasNext) und übersprungen (skip) werden können.

Unit Testing mit JUnit

- Was ist Unit Testing?
- Coverage

Fragen