University of Zurich[UZH]

s.e.a.l. software evolution & architecture lab

Informatics I – EProg HS15

Exercise 13

1 Task: Exceptions

1.1 Learning Objectives

1. Know how exception handling works

1.2 Task Description

Mark the correct statements:

- ☐ `catch (Exception e) { ... }` catches all exceptions that occur within the preceding `try` block. If this is followed by another `catch` block (e.g. a `catch (NullPointerException e) { ... }`), then this subsequent catch block is ignored.
- ☐ If multiple catch statements are used, `catch (Exception e) { ... }` can be used at the end to catch all remaining exceptions.
- ☐ The super-class of all exceptions and errors is `Throwable`.
- ☐ The super-class of all exceptions and errors is `Exception`.
- ☐ If a method executes code that could potentially raise a checked exception, it should handle the exception itself (`try/catch`) or it should contain `throws` in its declaration.
- ☐ A `finally`-block following `try/catch` is only executed if an exception occurred.

2 Task: Generics - Stack 2.0

2.1 Learning Objectives

1. Use generic type parameters
2. Declare and use custom Exception types

2.2 Task Description

The stack from Exercise 11 shall be extended to support generic types and exceptions. The new interface is defined as follows:

```
1  /**
2   * The <code>IStack</code> interface defines the functionality of a last-
        in-first-out (LIFO) stack for arbitrary
3   * elements.
4   *
5   * @param <T>
6   * the type of elements that are handled by the stack
7   */
8  public interface IStack<T> {
9
10   /**
11    * Pushes an element onto the top of this stack.
12    *
13    * @param element
14    * the element to be pushed onto this stack.
15    * @return the <code>element</code> argument.
16    */
17   public T push(T element) throws StackException;
18
19   /**
20    * Removes the element at the top of this stack and returns that element
          as the value of this function.
21    *
22    * @return The element at the top of this stack or <code>null</code> if
          the stack is empty.
23    */
24   public T pop() throws StackException;
25
26   /**
27    * Looks at the element at the top of this stack without removing it from
          the stack.
28    *
29    * @return the element at the top of this stack or <code>null</code> if
          the stack is empty.
30    */
31   public T peek() throws StackException;
32
```

```
33    /**
34     * Tests if this stack is empty.
35     *
36     * @return <code>true</code> if and only if this stack contains no
           elements; <code>false</code> otherwise.
37     */
38    public boolean isEmpty() throws StackException;
39
40    /**
41     * Returns the 1-based position where an element is on this stack. If the
           element <tt>element</tt> occurs as a element
42     * in this stack, this method returns the distance from the top of the
           stack to the first occurrence of the element in
43     * the stack (top to bottom); the topmost number on the stack is
           considered to be at distance <tt>1</tt>.
44     *
45     * @param element
46     * the desired element.
47     * @return the 1-based position from the top of the stack where the
           element is located; the return value
48     * <code>-1</code> indicates that the element is not on the stack.
49     */
50    public int search(T element) throws StackException;
51  }
```

Listing 1: `IStack` interface

Re-write the three implementations of Exericse 11 (`ArrayStack`, `ArrayListStack`, and `LinkedStack`) such that they implement the interface above to allow to fill the stack with arbitrary objects. Write a `TestDriver` to test your implementations.

- Remark for `ArrayStack`: In the solution of exercise 11 the array, that actually stores the elements, is initialized in the constructor of the `ArrayStack`. This is not possible in the new solution, because arrays of a generic type cannot be instantiated in Java (e.g. `new T[10]` is not possible). Therefore you have to initialize the array outside of the `ArrayStack` class (in the position of the code where you know its concrete type) and pass it to the constructor of `ArrayStack`.
- The second difference to the solution of Exercise 11 is that the size of the array cannot be increased inside of `ArrayStack`, due to the same problem. Instead of increasing the array size, simply raise a `StackException` if the array is full and an attempt is made to push another element. Catch the exception in the `TestDriver`.
- Note that even though all methods of `IStack<T>` declare that they may throw a `StackException`, there is no need to do this except when something actually goes wrong, which is only the case when trying to add another element to a full `ArrayStack`.

3 Task: Optional - reading files, fun with data structures

3.1 Learning Objectives

1. Using generic data structures native to Java
2. Be provided with code to read files into a Java program
3. Understand the purpose, usage, and limitations of `Map<K,V>` and what multimaps are.

3.2 Task Description

This task is optional for people that feel like they want to practice the usage of Java data structures. Additionally, as this is the case in practice, you may want to have a look at the provided code for reading in files like simple text files into a Java program and then do something with that data. This code is also intended for you to be reused in case you need to read in a file in another situation. It requires at least Java 7 to run, so make sure your JDK is up-to-date.

```
1  import java.io.BufferedReader;
2  import java.io.IOException;
3  import java.nio.charset.Charset;
4  import java.nio.file.Files;
5  import java.nio.file.Path;
6  import java.nio.file.Paths;
7  import java.util.ArrayList;
8
9  public class FileUtils {
10    public static ArrayList<String> readLinesFromFile(String filePath) {
11       ArrayList<String> lines = new ArrayList<String>();
12       Path file = Paths.get(filePath);
13
14       Charset charset = Charset.forName("UTF-8");
15       try (BufferedReader reader = Files.newBufferedReader(file, charset))
              {
16          String line = null;
17          while ((line = reader.readLine()) != null) {
18             lines.add(line);
19          }
20       } catch (IOException x) {
21          System.err.format("IOException: %s%n", x);
22       }
23       return lines;
24    }
25 }
```

Listing 2: `FileUtils` class

Your task is to use the `FileUtils.readLinesFromFile(filePath)` method to read in the provided text file `text.txt` line-by-line from your local disk, e.g. from your desktop, and then create an inverted index that holds each word of the file and for each word, all the line numbers representing on which line this word occurred (similar to an index or "Stichwortverzeichnis/Register" at the end of a book, where for each (important) word, the page numbers indicating on

which page this word occurred is shown). The words, furthermore, shall be sorted alphabetically. If a word occurs multiple times on one line, the line number shall only be stored once for this word. Line numbers for each word, additionally, shall be sorted from smallest to largest line number. Finally, print out those words and corresponding line numbers that have more than 1 and less than 10 lines they occur on to the screen.

The resulting output should look something like that:

```
...
alderaan [15, 16, 19]
an [4, 11]
analysis [15, 25]
as [10, 13, 20, 27]
attack [25, 27, 28]
base [24, 30]
before [4, 13, 30]
ben [9, 10]
but [3, 16, 24, 28]
by [3, 5, 6, 12, 13, 20, 21, 29]
...
```

Hints:

- As a base idea, make use of a `Map<K, V>`, where keys K are words (`Strings`) and values V are line numbers (`Integer`).
- Make use of a `TreeMap<K, V>` to implement such an index as it is a `SortedMap`, meaning that keys K are sorted lexicographically, making it alphabetically ordered if keys are `Strings`. This property is due to the underlying tree implementation of this `Map<K, V>`.
- As presented in the lecture slides, a `Map<K, V>` may only store for each key K **one** value V (it "maps" one key K to one value V). As we want to store multiple values V (line numbers) for each key K (word), it is necessary to store a certain collection as V inside the `TreeMap<K, V>`. Furthermore, the requirement is that each line number stored for each word may only occur once. A collection having these properties is a `TreeSet<V>`. In a set, every item may only occur once and the tree implementation (as already seen in `TreeMap<K, V>`) by default sorts the items in increasing lexicographical order. Thus, to implement the inverted index, use a `TreeMap<String, TreeSet<Integer> >`. Such an implementation is called a *multimap*, as it maps one key to one or more values and may thus overcome the limitations of a 1-to-1 mapping of a simple map.
- Inserting into a map requires the `put(K, V)` method, and `get(K)` returns the value V for the stored key K. If there is no value stored for a key K, get will return `null`. Thus, make sure you initialise the `TreeSet` for this key K in the `TreeMap` before you insert a new line number. In a `Set`, you can use `add(I)` to store an item I (Removing an item from a set is done with `remove(I)` but this is not required for this exercise).
- Iterating over a `Map` can be accomplished e.g. using the `keySet()` method and then for each key K in the key set, the `get(K)` method is called to get the value(s) for that key.
- Make sure to convert all lines to lower case such that the same words are mapped to the same line numbers. You may additionally remove all punctuation marks that are not numbers nor letters using the `String.replace()` or `String.replaceAll()` methods, such that more words are mapped to the same key in the map.