Answer Key

1. **General Understanding** (10 points)

   (a) What is the difference between a `List` and a `Set`?

   (b) What is a checked exception?

   (c) What is the type of `x` after '`Collection x = new HashSet();`'?

   (d) What is the difference between an `InputStream` and a `Reader`?

   (e) Describe one (any) purpose of design patterns?

   Answer:

   (a) A `Set` is an unordered collection of objects. A `List` is an ordered collection of objects, where the objects are indexed subsequentially.

   (b) Every `Throwable` object which is neither an `Error` nor a `RuntimeException` is a *checked exception*.

   (c) `Collection`

   (d) `InputStream` is a byte stream, whereas `Reader` is a character stream.

   (e) One main purpose of design patterns is the reuse of designs.

2. **Interface Stack** (10 points)

   A `Stack` is a data structure where elements are added and removed *at the same side*, namely at the top. This is like a stack of boxes where you can only take away the topmost box, the one which is added at latest. Adding an element (necessarily at the top) is performed by the operation 'push'. Removing an element (which necessarily was the topmost element then) is performed by the operation 'pop'. One can always ask for the topmost element, with an operation called 'top'.

   A stack is sometimes called *Last-In-First-Out (FIFO)* data structure. Here is the interface for a *stack of characters*:

```
public interface Stack {

    /** place c on top of this stack */
    public void push(char c);

    /** remove the topmost entry of this stack;
     *  throw a RuntimeException with detail message
     *  "Tried to pop empty stack!", if the stack is empty */
    public void pop();

    /** return the topmost entry of this stack;
     *  throw a RuntimeException with detail message
     *  "Tried to get top of empty stack!", if the stack is empty */
```

```
    public char top();

    /** return the current number of entries in this stack */
    public int size();
}
```

Your task is to write a class `MyStack` that implements the interface `Stack`. In `MyStack`, make use of the collections framework.

(Hint: For how to construct a `RuntimeException` object with the specified detail message, see the appended `javadoc` pages. If you want to use `LinkedList` for your solution, then also see the appended `javadoc` pages.)

Answer:

```
import java.util.*;

/** Use LinkedList -- top of stack is first element of list */
public class MyStack implements Stack {

    private LinkedList stack;

    public MyStack() {
        stack = new LinkedList();
    }

    public void push(char c) {
        stack.addFirst(new Character(c));
    }

    public void pop() {
        if (size() > 0)
          stack.removeFirst();
        else
          throw new RuntimeException(
                      "Tried to pop empty stack!"
                    );
    }

    public char top() {
        if (size() > 0)
          return ((Character)stack.getFirst()).charValue();
        else
          throw new RuntimeException(
                      "Tried to get top of empty stack!"
                    );
    }

    public int size() {
```

```
            return stack.size();
        }
    }
```

3. **Input** (10 points)

   Write a program `CountPairs.java`, which counts in a given input text file the number of characters which *immediately* follow an identical character.

   For example, consider a text file `file.txt`, containing just the following two lines:

   ```
   jw3Xxv44js
   snnnms8@@lvdu
   ```

   Calling
   `java CountPairs file.txt`
   has the output:
   `file.txt contains 4 characters immediately following an identical character`

   In this example, the 4 counted characters are the second `4`, the second and third `n` and the second `@`. Note that `x` is not the same as `X`, and that the second `s` immediately follows a 'newline' character, not the first `s`.

   Answer:

```
import java.io.*;

class CountPairs {
  public static void main(String[] args)
                          throws IOException {
    FileReader input = new FileReader(args[0]);
    char previous = (char)input.read();
    char current;
    int in;
    int counter = 0;
    while ((in = input.read()) != -1) {
      current = (char)in;
      if (current == previous) {
        counter++;
      }
      previous = current;
    }
    System.out.println(args[0] +
                       " contains " +
                       counter +
                       " characters immediately following" +
                       " an identical character");
  }
}
```

4. **Collections and Output** (15 points)

   Consider the class `Person`:

```
public class Person {

  private String personNumber;

  private String name;

  // The argument persNum of the constructor is
  // assumed to be a String containing 6 digits,
  // followed by a '-', followed by 4 digits,
  // like the String "771224-8743".
  public Person (String persNum, String aName) {
    personNumber = persNum;
    name = aName;
  }

  public String getPersonNumber () {
    return personNumber;
  }
  public String getName () {
    return name;
  }
}
```

Suppose we have a database which contains several `Set`s of such `Person` objects. These `Set`s are *not sorted*. From time to time, we need to print such a set of `Person`s to a text file, in such a way that the 'persons' appear *sorted after their age*, starting with the oldest. Each name is followed by the according person number. An example output file might look like:

```
Petra Setterberg
670520-8746
Anna Lundborg
671118-9223
Lennart Augustsson
690113-2342
Anders Magnusson
710514-4271
Per Blomqvist
710529-8382
```

For printing out such an *sorted* listening of an *unsorted* set, we provide a helper class `PrintSortedPersons`, which contains nothing but a `static` method:

```
public static void printSortedPersons(Set s, String outputFile)
```

`printSortedPersons` should print out the elements of `s` to the file `outputFile`, in a sorted way. (We assume that, whenever `printSortedPersons` is called with a certain `Set s`, then `s` contains *only* `Person` objects.)

Your task is to implement the method `printSortedPersons`. (You may assume that all persons considered are born between 1900 and 1999.)

Hints:

- Looking at the above example output, you may realize that the order of person numbers reflects the standard ordering of `String`s. For instance, `"670520-8746"` is a 'smaller' `String` than `"671118-9223"`, because the character '`0`' is smaller than the character '`1`', and both are the first characters which are different in both `String`s. This means: The older a person is, the smaller is the person number (as a `String`).
- As the `Person`s in `s` are not sorted, the method has to sort them before printing them to `outputFile`. The suggestion is not to sort these objects directly, but to split them up into 'keys' and 'values', and store this 'key-value' pairs into a `SortedMap`, particularly a `TreeMap`. Printing out then means to iterate over the `SortedMap` and print values and keys.
- Adding a persNum/name pair to a `TreeMap tmap` could be done by:
  `tmap.put( persNum, name );`
  where `tmap` is a reference to a sorted map object. Thereby, the person number strings are used as keys, while the names are used as values. Sorting is done automatically by the `put`-method, using the standard (alphabetical) order over `String`. (So you don't have to implement any order yourself.) The effect of the sorting becomes visible, when we iterate over the result of `tmap.entrySet();`, as the iterator respects the ordering of the keys automatically.
- For how to iterate over an entry set, see slides of lecture 10, 'Map by Example'.
- To write to a file, e.g. named `text.txt`, use a `PrintWriter`, like:
  ```
  PrintWriter out = new PrintWriter(new FileWriter("text.txt"));
  ...
  out.println( someString );
  ```

Answer:

```
import java.io.*;
import java.util.*;

public class PrintSortedPersons {

  // The parameter s is assumed to contain only Person objects.
  public static void printSortedPersons(Set s, String outputFile)
                                              throws IOException {
    SortedMap smap = new TreeMap();
    //sort s into smap:
    Iterator it1 = s.iterator();
    while (it1.hasNext()) {
      // ausprobieren ohne casten:
      Person p = (Person)it1.next();
      smap.put(p.getPersonNumber(), p.getName());
    }
    //print out smap, by iterating through all entries
    PrintWriter out = new PrintWriter(new FileWriter(outputFile));
    Set entries = smap.entrySet();
```

```
        Iterator it2 = entries.iterator();
        while (it2.hasNext()) {
          Map.Entry entry = (Map.Entry)it2.next();
          String keyString = (String)entry.getKey();
          String valueString = (String)entry.getValue();
          out.println( valueString );
          out.println( keyString );
        }
        out.close();
      }
    }
```

5. **Threads** (15 points)

   Consider the thread class `MyThread`:

```
class MyThread extends Thread {

   private int whatToAdd;
   private int howOften;

   static int shared = 0; //*** shared by all MyThread objects

   MyThread( int toAdd, int often ) {
      whatToAdd = toAdd;
      howOften = often;
   }

   public void run() {
      int temp = 0;
      for ( int i = 0; i < howOften; i++ ) {
         temp = shared + whatToAdd;
         shared = temp;
      }
   }
}
```

   Any such thread adds `howOften` times the number `whatToAdd` to the number `shared`. (Note that `whatToAdd` can be negative.) `shared` is a 'shared' field of all instances of `MyThread`, not because of its name, but because it is `static`.

   (a) (6 points) Write a class `IncrAndDecr`, such that the `main` method starts two threads. The first thread adds one million times `+1` to `shared`, the second thread adds one million times `-1` to `shared`. After starting both threads, the `main` method waits until both threads stop running, and then prints out the value of `shared`.

   Hint: Waiting until a thread `t` stops running is done by the statement:
   `t.join();`
   See also `join` *in the appended* `jacadoc` *pages.*

(b) (2 points) `java IncrAndDecrOneMillion` will most certainly *not* output '`0`'. Why?

(c) (5 points) Modify `MyThread` such that `java IncrAndDecrOneMillion` will certainly output '`0`'. The modification should still allow the threads to truly run in parallel, which means that the execution can switch back and forth between adding `+1` and adding `-1`.

Hint: You may use 'synchronized statements' rather than a 'synchronized method' for this. See also lecture 6 or Jia section 8.2.1 . Note that the `synchronized` block can contain more than one statement. To use the *same* object lock for *both* threads, you may synchronize on a `static` object `lockObj`, which you can define by `static Object lockObj = new Object();`.

(d) (2 points) Modify the class `IncrAndDecrOneMillion` such that the number of iterations both threads perform is given as a command line argument.

Answer:

(a)

```
class IncrAndDecrOneMillion {
   public static void main( String[ ] args ) {
      int iterations = Integer.parseInt(args[0]);
      MyThread t1 = new MyThread( +1, iterations );
      MyThread t2 = new MyThread( -1, iterations );
      t1.start();
      t2.start();
      // wait for both threads to complete before printing result
      try {
        t1.join();
        t2.join();
      } catch( InterruptedException e ) { }
      System.out.println( MyThread.shared );
   }
}
```

(b)

As nothing is synchronized here, two `for`-iterations of both threads could interfere. It might e.g. happen that the result of `shared + +1` is not assigned to `shared` before `shared + -1` is computed *and* assigned to `shared`.

(c)

```
class MyThread extends Thread {

   private int whatToAdd;
   private int howOften;

   static int shared = 0; //*** shared by all MyThread objects
   static Object lockObj = new Object();

   MyThread( int toAdd, int often ) {
```

```
      whatToAdd = toAdd;
      howOften = often;
   }

   public void run() {
      int temp = 0;
      for ( int i = 0; i < howOften; i++ ) {
        synchronized (lockObj) {
          temp = shared + whatToAdd;
          shared = temp;
        }
      }
   }
}
```

(d)

```
class IncrAndDecrOneMillion {
   public static void main( String[ ] args ) {
      int iterations = Integer.parseInt(args[0]);
      MyThread t1 = new MyThread( +1, iterations );
      MyThread t2 = new MyThread( -1, iterations );
      .
      .
      .
```