Answer Key

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

   (a) What is the relationship between a method's `throws` clause and that *checked* exceptions the can be thrown during the execution of the method?

   (b) If a non-static field of a class `A` is declared `protected`, where might it be accessed?

   (c) What is the difference between `start()` and `run()` in the context of threads?

   (d) What value of which type does `read()` return to indicate the end of a stream?

   (e) Describe the difference between *functional* and *structural* testing. (Two sentences may suffice.)

   Answer:

   (a) In a method's `throws` clause you have to declare all checked, uncaught exceptions which might be thrown during execution of the method.

   (b) In the class `A` itself, in all subclasses of `A`, and in all classes that belong to the same package than `A`.

   (c) `run()` executes the thread, and the thread is finished when `run()` returns. `start()` only *starts* the execution of the thread, and returns immediately, regardless of how long the thread runs.

   (d) The `int` value -1.

   (e) *Functional* testing is exclusively performed on the basis of a unit's *specification* (such as a description of the input/output relation). *Structural* is performed by properly taking the implementation into account.

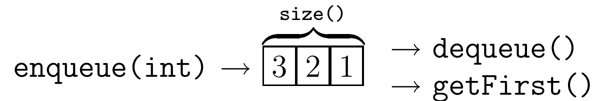
2. **Interface Queue** (10 points)

   A `Queue` is a data structure where new elements are always *added at the end*, while elements are still only *taken out at the front*. This is like a queue of people waiting in a shop: they can only enter the queue at the end (the queue's tail), and the queue only gets smaller when the first person of the queue (the queue's head) is served. A queue is sometimes called *First-In-First-Out (FIFO)* data structure. Here is the interface for a *queue of integers*:

```
public interface Queue {

    /** append i at the tail of this queue and return i */
    public int enqueue(int i);

    /** remove the head entry of this queue and return it;
     * throw a RuntimeException, if the queue is empty */
    public int dequeue();

    /** return the head entry of this queue; throw
```

```
    *  a RuntimeException, if the queue is empty */
    public int getFirst();

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

Here is a schematic picture of a queue (1 was enqueued first, then 2, then 3):



Your task is to write a class `MyQueue` that implements the interface `Queue`. In `MyQueue`, make use of the collections framework. (Hint: The classes in the collections framework store *objects*, whereas the `Queue` operations receive and return the *primitive type* `int`. Therefore, you may use the wrapper class `Integer`.)

Answer:

```
import java.util.*;

/** Use LinkedList -- head of queue is head of list */
public class MyQueue implements Queue {

    private List queue;

    public MyQueue() {
        queue = new LinkedList();
    }

    public int enqueue(int i) {
        queue.add(new Integer(i));
        return i;
    }

    public int dequeue() {
        if (size() > 0)
          return ((Integer)queue.remove(0)).intValue();
        else
          throw new RuntimeException(
                    "Tried to dequeue empty queue!"
                  );
    }

    public int getFirst() {
        if (size() > 0)
          return ((Integer)queue.get(0)).intValue();
        else
          throw new RuntimeException(
```

```
                        "Tried to get first of empty queue!"
                    );
        }

        public int size() {
            return queue.size();
        }
}
```

Remark: It is also correct to not explicitly throw a `RuntimeException` if the called method throws the exception itself. That means: the above program remains to be a correct answer if the methods `dequeue` and `getFirst` only contain the `return` line.

3. **Input/Output** (10 points)

   Write a program `FindChar.java` which prints the line number where the first instance of a particular character is found in an input file. The character is specified by the first command line argument, the input file by the second command line argument. (The program is assumed to be executed in the directory where the input file resides.)

   The output of the program follows the subsequent examples:

   - If the first instance of the character '`I`' in file `text.txt` is in line 7, then
     `java FindChar I text.txt`
     prints out:
     `The character I appears first in line 7 of file text.txt.`
   - If the character '`y`' does not appear in file `text.txt`, then
     `java FindChar y text.txt`
     prints out:
     `The character y does not appear in file text.txt.`

   What is the output of your program, if called with the following command line?
   `java FindChar x FindChar.java`
   (Line numbers are counted starting with 0.)

   Answer:

```
import java.io.*;

class FindChar {
  public static void main(String[] args)
                           throws IOException {
    char match = args[0].charAt(0);
    FileReader fileIn = new FileReader(args[1]);
    LineNumberReader in = new LineNumberReader(fileIn);
    int i;
    while ((i = in.read()) != -1) {
      char ch = (char)i;
      if (ch == match) {
```

```
            System.out.println("The character " + match
                               + " appears first in line "
                               + in.getLineNumber()
                               + " of file " + args[1] + ".");
            return;
          }
        }
        System.out.println("The character " + match
                           + " does not appear in file "
                           + args[1] + ".");
      }
    }
```

4. **I/O and Collections** (15 points)

   We consider a simple format for a text file serving as address book. A sample file might look like:

```
Setterberg Petra
070/188329
Lundborg Anna
031/498711
Magnusson Anders
031/471992
Blomqvist Per
073/214947
```

   Regardless how this file is created (may be it is edited by hand), such an address book should be sorted from time to time. The result of sorting is, in the example, a file looking like:

```
Blomqvist Per
073/214947
Lundborg Anna
031/498711
Magnusson Anders
031/471992
Setterberg Petra
070/188329
```

   The second functionality we want to have is the merging of two address book files. The outcome should be sorted, no matter if the two files were sorted or not. In the following, you can assume that any input file, taken as an address book, follows the pattern sketched above: the lines with names and telephone numbers alternate, and there is an even number of lines in total. (Even if you can assume well-formed files, the compiler requires exception handling.)

   Look at both of the following tasks, before you start one of them. *To obtain maximal points, you have to avoid duplicate code.* Therefore, place the code common to both in some extra class.

- Write a public class, the `main` method of which expects two file names as command line input. The first argument is the input address book file, the second is the output address book file, to which the sorted entries are 'printed'.
- Write a public class, the `main` method of which expects three file names as command line input. The first two arguments are input address book files, the second is the output address book file, to which the merged and sorted entries are 'printed'.

Your implementation should use the class `TreeMap`, which implements the interface `SortedMap`. Adding a name/number pair could be done by:
`smap.put( nameString, numberString );`
where `smap` is a reference to a sorted map object. Thereby, the names are used as keys, and the telephone numbers 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 `smap.entrySet();`, as the iterator respects the ordering of the keys automatically.

Hints:

- For how to iterate over an entry set, see slides of lecture 10, 'Map by Example'.
- The method `readLine()` of the class `BufferedReader` returns a String, either containing the contents of the line, or null if the end of the stream has been reached.
- To write to a file, e.g. named `text.txt`, use a `PrintWriter`, like:
```
PrintWriter out = PrintWriter(new FileWriter("text.txt"));
...
out.println( someString );
```
  Thereby, a line containing `someString` is printed to the file `text.txt`, not to the standard output.

Answer:

- Sorter:

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

public class Sorter{

    private SortedMap smap = new TreeMap();

    public void insertFromReader(BufferedReader r)
                                throws IOException{
        String nameString = r.readLine();
        String numberString = r.readLine();

        while(nameString != null && numberString != null){
            smap.put(nameString,numberString);
            nameString = r.readLine();
            numberString = r.readLine();
```

```
        }
    }

    public void putToWriter(BufferedWriter w)
                            throws IOException{
        Iterator it = smap.entrySet().iterator();
        Map.Entry entry;

        while(it.hasNext()) {
            entry = (Map.Entry) it.next();
            w.write(entry.getKey() + "\n");
            w.write(entry.getValue() + "\n");
        }
    }

}
```

- MergeAndSortBook:

```
import java.io.*;

public class MergeAndSortBook{

    public static void main(String[] args){
        BufferedReader in, in2;
        BufferedWriter out;
        Sorter sorter = new Sorter();
        int index = 0;

        try{
            in = new BufferedReader(new FileReader(args[index]));
            sorter.insertFromReader(in);
            in.close();
            index++;
            if(args.length == 3){
                in2 = new BufferedReader(
                                    new FileReader(args[index]));
                sorter.insertFromReader(in2);
                in2.close();
                index++;
            }
            out = new BufferedWriter(new FileWriter(args[index]));
            sorter.putToWriter(out);
            out.close();
        }
        catch (FileNotFoundException e) {
```

```
                throw new RuntimeException("File not found: "
                                             + args[0]);
            }
            catch (IOException e) {
                throw new RuntimeException("Error writing file: "
                                             + args[1]
                                             + " or reading file: "
                                             + args[0]);
            }
        }
    }
```

5. **Threads: A Bakery** (15 points)

   We consider a swedish bakery featuring the typical *nummerlapp* system: When a customer enters a shop he or she draws a number from a dispenser. We assume there is only one clerk, announcing who's next by looking at a display showing the next number to serve. After serving a customer, the clerk presses some button, causing the displayed number to increment.

   Our task is to build a multithreaded simulation that uses a model of the dispenser and the incrementable display to coordinate the behaviour of the customers and the single clerk. The simulation only models a small part of the real world scenario. Particularly, we do not model the bread/money exchange. Therefore, in the simulation, the only action performed by a customer is taking a number. The clerk's action of serving the customer consists just in push the button (i.e. invoking a method) which causes the serving number to increment.

   Problem Decomposition

   We will have one thread per customer and one thread for the clerk. The data that must be visible for different threads is held by a single object, which keeps track of the next number to take and the next number to serve.

   We have the following classes:

   - `TakeNumServeNum` — Represents both the dispenser and the display. This class is also responsible for reporting the number taking/serving actions, by printing out what happens. This class will only have one instance (one object).
   - `Customer` — Represents a customer who will use the `TakeNumServeNum` object to 'get a number'. Instances (objects) of this class execute threads, which (according to the above remark) finish running after a number is taken.
   - `Clerk` — Will use the `TakeNumServeNum` object to 'serve a number'. The one and only instance (object) of this class executes a thread.
   - `Bakery` — Responsible for creating the threads and starting the simulation.

   Here is the (first) implementation of the `TakeNumServeNum` class.

```
/*
 * Description: An instance of this class serves as
 * a shared resource for the customers and clerk threads
```

```
 * of the bakery simulation. This object contains two
 * instance variables, both of which are initialized to 0.
 * The variable nextNumToTake represents the next "ticket"
 * given to customers as they arrive. The variable
 * nextNumToServe represents the next customer to be served.
 */

class TakeNumServeNum {
    // Next number to take by a customer:
    private int nextNumToTake = 0;
    // Next number to serve by the clerk:
    private int nextNumToServe = 0;

    // called once by each customer:
    public void takeANumber( int custId ) {
        nextNumToTake++;
        System.out.println( "Customer " + custId
                            + " takes ticket " + nextNumToTake );
    }

    // repeatedly called by the clerk:
    public void serveANumber() {
        nextNumToServe++;
        System.out.println( "  Clerk serves ticket "
                            + nextNumToServe );
    }
}
```

In the following, you are asked to implement three versions of the program, possibly modifying also the `TakeNumServeNum` class. The first version does neither care about synchronization nor about cooperation of the threads. The second version introduces synchronization, the third introduces cooperation.

(a) (8 points) Implement the other three classes. In the first version, do *not* use the `synchronized` modifier or the `synchronize` statement. Also don't care about cooperation, i.e. don't care if the clerk servers numbers not yet being taken by a customer. Here are some ingredients for the classes:

- `Customer`:
  To give every Customer a unique ID, the `Customer` class needs a static variable which counts ID numbers. It is initialized to `100`:
  `private static int idCounter = 100;`
  Then, each `Customer` object has two fields, and `id` and a reference `nextNums` to the `TakeNumServeNum` object:
  `private int id;`
  `private TakeNumServeNum nextNums;`
  This reference is given as an argument to the constructor:
  `public Customer( TakeNumServeNum tsNum ) {`

```
        id = idCounter;
        idCounter++;
        nextNums = tsNum;
    }
```

In order to make the simulation less predictable, use a randomized delay before a customer takes a number:
`sleep((int)(Math.random() * 1000));`
As we said above: a `Customer` thread is done after taking one number.

- `Clerk`:
  Similarly, the `Clerk` works with a randomized delay before serving a number:
  `sleep((int)(Math.random() * 50));`
  Serving a number means nothing but calling the `serveANumber()` method of the `TakeNumServeNum` object. Like a `Customer`, the `Clerk` needs to have a reference to the `TakeNumServeNum` object.
- `Bakery`:
  In its `main` method, the `Bakery` creates all objects and starts all threads. The number of customers appearing in the simulation is given as command line argument to the `Bakery`.

(b) (2 points) So far, the critical sections of the threads are not synchronized. Which are the critical sections, and what can happen if we do not synchronize them?

(c) (2 points) Synchronize the critical sections. You can choose between writing the classes again or *clearly* describe the code modification.

(d) (3 points) As long as the `Clerk` thread does not care if numbers it serves are at all taken by any `Customer` thread, we might get the following output:

```
Clerk serves ticket 1
Clerk serves ticket 2
Customer 101 takes ticket 1
Customer 100 takes ticket 2
...
```

It is no problem that `Customer 100` randomly takes a number after `Customer 101`, but it is a problem that `Clerk` serves numbers before they are taken. Repair this, not using 'busy waiting' of the `Clerk` thread, but using `wait()` and `notifyAll()`. (Hint: A comparison between `nextNumToTake` and `nextNumToServe` can indicate if a number is waiting for being served or not.) You can choose between writing the classes again or *clearly* describe the code modification.

Answer:

(a) (8 points)

- `Customer`:

```
public class Customer extends Thread {

    private static int idCounter = 100;
    private int id;
    private TakeNumServeNum nextNums;
```

```
    public Customer( TakeNumServeNum tsNum ) {
      id = idCounter;
      idCounter++;
      nextNums = tsNum;
    }

    public void run() {
        try {
            sleep((int)(Math.random() * 2000));
            nextNums.takeANumber(id);
        } catch (InterruptedException e) { }
    }
}
```

- Clerk:

```
public class Clerk extends Thread {
    private TakeNumServeNum nextNums;

    public Clerk(TakeNumServeNum tsNum) {
      nextNums = tsNum;
    }

    public void run() {
        while (true) {
            try {
                sleep((int)(Math.random() * 1000));
                nextNums.serveANumber();
            } catch (InterruptedException e) { }
        }
    }
}
```

- Bakery:

```
public class Bakery {
    public static void main(String[] args) {
        TakeNumServeNum nextNums = new TakeNumServeNum();
        Clerk clerk = new Clerk(nextNums);
        clerk.start();
        int appearingCustomers = Integer.parseInt(args[0]);
        for (int k = 0; k < appearingCustomers ; k++) {
            Customer customer = new Customer(nextNums);
            customer.start();
        }
```

```
    }
}
```

(b) (2 points)

The critical sections are the two methods of the `TakeNumServeNum` class, which are: `takeANumber` and `serveANumber`. If not synchronized, these methods may interfere, with each other or with themselves. E.g., it may happen that two customers are printed out to take the same number.

(c) (2 points)

change `TakeNumServeNum` to:

```
class TakeNumServeNum {
    // Next number to take by a customer:
    private int nextNumToTake = 0;
    // Next number to serve by the clerk:
    private int nextNumToServe = 0;

    // called once by each customer:
    public synchronized void takeANumber( int custId ) {
        nextNumToTake++;
        System.out.println( "Customer " + custId
                            + " takes ticket " + nextNumToTake );
    }

    // repeatedly called by the clerk:
    public synchronized void serveANumber() {
        nextNumToServe++;
        System.out.println( "  Clerk serves ticket "
                            + nextNumToServe );
    }
}
```

(d) (3 points)

change `TakeNumServeNum` to:

```
class TakeNumServeNum {
    // Next number to take by a customer:
    private int nextNumToTake = 0;
    // Next number to serve by the clerk:
    private int nextNumToServe = 0;

    // called once by each customer:
    public synchronized void takeANumber( int custId ) {
        nextNumToTake++;
```

```
            System.out.println( "Customer " + custId
                                + " takes ticket " + nextNumToTake );
            notifyAll();
        }

        // repeatedly called by the clerk:
        public synchronized void serveANumber() {
            while (nextNumToTake <= nextNumToServe) {
              try {
                wait();
              } catch(InterruptedException e) { }
            }
            nextNumToServe++;
            System.out.println( "  Clerk serves ticket "
                                + nextNumToServe );
        }
    }
```