Answer Key

1. **General Understanding** (10 points)
   Answer the following questions with 'yes' or 'no'. Read the questions carefully and think before you decide.

   (a) (2 points)
   Does the interface `java.util.SortedSet` implement `java.util.Collection`?

   (b) (2 points)
   Does the initializer '`Object o = new Integer(5);`' compile?
   (Assume `o` was not declared before?)

   (c) (2 points)
   Is branch coverage testing a white box technique?

   (d) (2 points)
   Does the expression
   `(new Integer(11) == new Integer(11))`
   evaluate to `true`?

   (e) (2 points)
   Is `String` a reference type?

   Answer: (The explenations given here were not required.)

   (a) (2 points)
   no.
   (An interface never *implements* anything. It can only *extend* another interface.)

   (b) (2 points)
   yes.
   (The type `Integer` is compatible with the type `Object`.)

   (c) (2 points)
   yes.
   (To know the different *branches* of program execution, we have to know the implementation.)

   (d) (2 points)
   no.
   (Each '`new`' operator creates a *different* object and returns a reference to it. Therefore, '`==`' here compares two different references.)

   (e) (2 points)
   yes.
   (`String`s are objects.)

2. **Interfaces** (10 points)

(a) (2 points)
Please remind yourself of the `Iterator` interface, which is also documented in the enclosed API descriptions. Implement a small helper class `IteratorPrinter`, only providing a `static` method `printElements(Iterator it)`, which prints all elements you can get out of the iterator `it` to `System.out` (one element per line).

(b) (6 points)
The `Iterator` interface is meant to be used in combination with `Collection` types. Nevertheless, the task here is to write an implementation of this interface not for collections, but for *arrays*, more precisely for *arrays* the element type of which is a reference type (not a primitive type).

Write a class `ObjectArrayIterator` implementing the `Iterator` interface ('`implements Iterator`'). Each `ObjectArrayIterator` object is 'bound' to an underlying array. Using the `next()` method of an `ObjectArrayIterator` object, one can iterate over the elements of the underlying array.

You have to carefully follow the contracts described in the '**MethodDetail**' section of the enclosed API description. This includes implementing what is required by the respective '**Throws:**' sections in the API description.

The class should not support the `remove()` *method.* That means: you still have to implement `remove()`, but only in a certain way. (See the according '**Throws:**' section.)

Beside the `Iterator` methods, you must implement a constructor
`public ObjectArrayIterator(Object[] a)`
which creates an `ObjectArrayIterator` for the array.

(Hint: Store the array `a` in a field, and provide an additional `nextElement` field. In this exam, don't worry about independent iterations.)

(c) (2 points)
Now the `ObjectArrayIterator` class is to be used, in a very small (and silly) example.

Implement a small class `PrintArgs`, the `main` method of which prints out all command line arguments, *but only by calling the* `printElements` *method of the class* `IteratorPrinter`. (So don't use a `for` or `while` in this class.)

Hints:

- All arrays of type `Type[]`, where `Type` is a *reference type*, are compatible with the array type `Object[]`. For example, you can pass an array of type `String[]` to a method with argument type `Object[]`.
- To throw an exception of type `SomeException`, use the following statement:
  `throw new SomeException();`
- Both, the `NoSuchElementException` and the `UnsupportedOperationException`, are unchecked exception types, as they extend `RuntimeException`. Therefore, exceptions of both types neither have to be caught (using `try catch`) nor have to be declared in a method's head.

Answer:

(a) (2 points)

```
import java.util.*;

public class IteratorPrinter {
  public static void printElements(Iterator i) {
    while (i.hasNext())
      System.out.println(i.next());
  }
}
```

(b) (6 points)

```
import java.util.*;

public class ObjectArrayIterator implements Iterator {

        private Object[] a;
        private int nextElement;

        public ObjectArrayIterator(Object[] a) {
                this.a = a;
                nextElement = 0;
        }
        public boolean hasNext() {
                return nextElement < a.length;
        }
        public Object next() {
          if (!hasNext())
            throw new NoSuchElementException();
          nextElement = nextElement + 1;
          return a[nextElement - 1];
        }
        public void remove () {
          throw new UnsupportedOperationException();
        }
}
```

(c) (2 points)

```
import java.util.*;

public class PrintArgs {
  public static void main(String[] args) {
    Iterator it = new ObjectArrayIterator(args);
    IteratorPrinter.printElements(it);
  }
}
```

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

Write a program `LineAndColumn.java` which seeks for the first occurrence of a particular character in a text file and prints out the *line number* and *position within the line* of that occurrence. The program in it's normal use takes two command line arguments, the first being the name of the text file, the second being the character to be searched. The program should complain if it gets to the wrong number of arguments, or if the second argument is a `String` longer than one character. Also, it should tell when there appears an input/output problem. (The program is assumed to be executed in the directory where the input file resides.)

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

```
jw3Xxv44js
snnnms8@@lvdu
```

Moreover, assume there is no file `xxx.txt` in the current directory. Here are example outputs:

- `java LineAndColumn test.txt j`
  prints out:
  `the character j appears first in line 1 at position 0`
- `java LineAndColumn test.txt m`
  prints out:
  `the character m appears first in line 2 at position 4`
- `java LineAndColumn test.txt a`
  prints out:
  `the character a does not appear in the file`
- `java LineAndColumn test.txt`
  prints out:
  `wrong number of arguments`
- `java LineAndColumn test.txt bb`
  prints out:
  `second argument must be a single character`
- `java LineAndColumn xxx.txt j`
  prints out:
  `there appeared an input/output problem`

(It doesn't matter if your program considers the first line being line number 1 or line number 0. The same holds for the positions within the lines.)

Hints:

- One can exit a `void` method at any point in the method body by using the statement '`return;`'. But use this only if you think it simplifies your code.
- Consider using a `LineNumberReader`.
- Consider the methods `charAt(i)` and `indexOf(c)` of the `String` class. (See Jia.)

Answer:

```
import java.io.*;

public class LineAndColumn {

  public static void main(String[] args) {

    if (args.length != 2) {
      System.out.println("wrong number of arguments");
      return;
    }
    if (args[1].length() != 1) {
      System.out.println("second argument must be " +
                         "a single character");
      return;
    }
    String file = args[0];
    char toFind = args[1].charAt(0);

    String line;
    int indexInLine;
    boolean found = false;
    try {
      LineNumberReader input =
        new LineNumberReader(new FileReader(file));

      while (((line = input.readLine()) != null) && !found) {
        indexInLine = line.indexOf(toFind);
        if (indexInLine != -1) {
          found = true;
          System.out.println("the character " + toFind +
                             " appears first in line " +
                             input.getLineNumber() +
                             " at position " + indexInLine);
        }
      }
      if (!found)
        System.out.println("the character " + toFind +
                           " does not appear in the file");
      input.close();
    }
    catch (IOException e) {
      System.out.println("there appeared " +
                         "an input/output problem");
    }
  }
}
```

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

We consider a scenario where simple *price lists* are available in a text format. As an example, here is the content of the file `priceList.txt`:

```
butter    20
bread     18
jam       25
cola      15
milk       9
salami    30
```

(For simplicity, all prices are full SEK, so Öre are not considered.)
You can assume all price lists having such a form. Each line starts with a word, followed by an arbitrary number of white spaces (`' '`), followed by digits representing a number. White spaces appear only between the words and the numbers. No TABs are used.

Another form of text files are *orders*, such as the following file `order.txt`

```
butter
jam
bread
milk
beer
```

You can assume all orders having such a form. Each line contains one word, and the file does neither contain white spaces nor TABs. Moreover, for simplicity we assume that no word appears twice in an order.

Write a java program which takes such a *price list* and an *order* and prints an according receipt ('kvitto') to `System.out`. On our example files, it behaves like this:

calling:
`java Receipt priceList.txt order.txt`
results in the following output:

```
bread for 18
jam for 25
butter for 20
beer not availabe
milk for 9
= = = = = = = = =
total: 72
```

(The order in which items are listed in the output is not significant. Your program is allowed to list them in an arbitrary order.)

To simplify your task, the program does not have to be robust this time. This means, you can receive full points even if your program does not react properly to unexpected situations, like a wrong number of arguments, non-existing files, or files having a different format than explained above. *Nevertheless, the program has to compile. Therefore, you have to do as much exception handling as the compiler requires.*

The program is not allowed to read through the input files again and again, whenever the price of an item has to be looked up. Instead, store items and prices in a suitable data structure.

Hints:

- Consider using `BufferedReader` and `StringTokenizer`.
- 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 iterate over a `Collection` (for example a `Set`), you can create an iterator by calling the `iterator()` method.
- The `get(k)` method of `Map` returns the value to which this map maps the specified key. It returns `null` if the `Map` contains no mapping for this key.
- The return type of `get(k)` is `Object`.
- The argument type of `Integer.parseInt(str)` is `String`.

Answer:

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

public class Receipt {

  public static void main(String[] args) throws IOException {

    String priceFile = args[0];
    String orderFile = args[1];

    BufferedReader priceReader =
                   new BufferedReader(new FileReader(priceFile));
    BufferedReader orderReader =
                   new BufferedReader(new FileReader(orderFile));

    // read the prices into the price map

    Map priceMap = new HashMap();

    String line;

    while ((line = priceReader.readLine()) != null) {
      StringTokenizer st = new StringTokenizer(line);
      priceMap.put(st.nextToken(), st.nextToken());
    }

    priceReader.close();

    // read the orders into the set of orders
```

```
        Set orders = new HashSet();

        while ((line = orderReader.readLine()) != null)
          orders.add(line);

        orderReader.close();

        // iterate over the orders and sum up the prices

        int total = 0;

        Iterator it = orders.iterator();
        while (it.hasNext()) {
          Object item = it.next();
          Object price = priceMap.get(item);
          if (price == null)
            System.out.println(item + " not available");
          else {
            System.out.println(item + " for " + price);
            total = total + Integer.parseInt((String) price);
          }
        }

        System.out.println("========");
        System.out.println("total: " + total);
      }
    }
```

5. **Threads** (15 points)

   This assignment talks about print jobs, being produced by clients, collected in a print queue, and being printed by a print server.

   We represent the information to be printed by a (potentially very long) `String`. Consider the class `PrintJob`, wrapping stings which are going to be printed:

```
class PrintJob {
  public String whatToPrint;
  public PrintJob followingJob;

  public PrintJob(String str) {
    whatToPrint = str;
    followingJob = null;
  }
}
```

   As `PrintJob` objects have a reference to another `PrintJob` object, they can be chained up to a queue which is to be served by a print server. We now present an according class `PrintQueue`. This class allows to add strings, and to take them out again. It is

`PrintQueue` itself which does the wrapping and unwrapping of strings into (and from) print jobs. Therefore, it is appropriate to make `PrintJob` visible only in `PrintQueue`. Nobody else might use the class. This is achieved by moving the class `PrintJob` *into* the class `PrintQueue`, as a `private static` inner class. (`private static` inner classes act like normal classes, beside that they are visible only within code of their outer class.) Better ignore the 'critical point' comments in the first reading.

```
public class PrintQueue {

  private static class PrintJob {
    public String whatToPrint;
    public PrintJob followingJob;

    public PrintJob(String str) {
      whatToPrint = str;
      followingJob = null;
    }
  }

  private PrintJob firstToPrint = null;
  private PrintJob lastToPrint = null;
  private int length = 0;

  public void addToQueue(String str) {
    PrintJob pJob = new PrintJob(str);

    if (length == 0)
      firstToPrint = pJob;
    else
      lastToPrint.followingJob = pJob;

    lastToPrint = pJob;
    //(critical point here?)
    length = length + 1;
  }

  public String takeFromQueue() {
    while (length == 0) {
      // waiting until no longer empty
    }

    PrintJob forReturning = firstToPrint;
    firstToPrint = firstToPrint.followingJob;
    //(critical point here?)
    length = length - 1;
```

```
    if (length == 0)
      lastToPrint = null;

    return forReturning.whatToPrint;
  }
}
```

(a) (2 points)
Describe *briefly* how `addToQueue` and `takeFromQueue` work.

(b) (4 points)
Now consider the `PrintServer` class:

```
public class PrintServer extends Thread {
  private PrintQueue queue;

  public PrintServer(PrintQueue pq) {
    queue = pq;
  }

  public void run() {
    while (true) {
      String toPrint = queue.takeFromQueue();
      // do the real printing ...
    }
  }
}
```

A `PrintServer` object stores a reference to its corresponding `PrintQueue` object. `PrintServer` extends `Thread`, because it is executed in parallel to *clients*.

Implement a class `Client`, also extending `Thread`. Different clients run in parallel to each other and to the server. For simplicity, your `Client` should only add the *same* `String` again and again to the *same* `PrintQueue`. Both, the `String` and the `PrintQueue`, are given as arguments to the constructor. For not creating *too* many print jobs, execute '`Thread.sleep(100);`' after each adding. (`sleep()` is declared to possibly throw an `InterruptedException`, which is a checked exception! The same holds for `wait()`, see below.)

(c) (3 points)
Implement a tester class `PrintTest`, the `main` method of which creates a print queue, and starts three clients (with different strings to print) and one server. All clients and the server refer to the same print queue.

(d) (3 points)
With the given implementation, a `PrintQueue` can get corrupted, in at least two different ways. Explain how. What can be done to prevent this? Indicate clearly the necessary changes to the code. Hint: Look at the 'critical point' comments now.

(e) (3 points)
The current implementation of `takeFromQueue` potentially spends time in a exe-

cuting a 'waiting' loop, if the queue is empty. This prevents other threads from executing immediately. In combination with `synchronized`, it can even lead to a deadlock. What should you do instead? Indicate clearly the necessary changes to the code.

Answer:

(a) (2 points)

`addToQueue`:

- takes a string as an argument
- it wraps the string into a `PrintJob`
- if there are no jobs in the print queue, this job becomes the front and end of the `PrintQueue`
- else this job is appended at the end of the `PrintQueue`
- the `length` is incremented

`takeFromQueue`:

- waits until jobs are in the queue
- remembers the first `PrintJob`, which is then discarded from the `PrintQueue`
- the `length` is decremented
- if the discarded job was the only one, then there is no last job left
- unwraps the string from the remembered job and returns it

(b) (4 points)

```
public class Client extends Thread {

  private PrintQueue queue;

  private String toPrintRepeatedly;

  public Client(PrintQueue pq, String str) {
    queue = pq;
    toPrintRepeatedly = str;
  }

  public void run() {
    while (true) {
      queue.addToQueue(toPrintRepeatedly);
      try {
        Thread.sleep(100);
      } catch (InterruptedException e) {}
    }
  }
}
```

(c) (3 points)

```
public class PrintTest {

  public static void main(String[] args) {

    PrintQueue pQueue = new PrintQueue();

    Client cl1 = new Client(pQueue, "Sent from client 1.");
    Client cl2 = new Client(pQueue, "Sent from client 2.");
    Client cl3 = new Client(pQueue, "Sent from client 3.");

    PrintServer server = new PrintServer(pQueue);

    cl1.start();
    cl2.start();
    cl3.start();
    server.start();
  }
}
```

(d) (3 points)

The `addToQueue` could be interrupted right after setting `lastToPrint` to the new job, without having increased the `length`. The queue then might seem to be empty but is not, which might cause this job being 'overwritten' by another job.

The `takeFromQueue` could be interrupted right after discarding the so far first job, without having decreased the `length`. The queue then might seem to contain one more job but actually is empty, causing a `NullPointerException` if another server uses the same queue to take out jobs.

To prevent this, both methods should be `synchronized`. See the code in the answer to (e).

(e) (3 points)

```
  public synchronized void addToQueue(String str) {

    System.out.println("adding: " + str);
    System.out.println();
    PrintJob pJob = new PrintJob(str);
    if (length == 0)
      firstToPrint = pJob;
    else
      lastToPrint.followingJob = pJob;
    lastToPrint = pJob;
    length = length + 1;

    notifyAll();
```

```
}

public synchronized String takeFromQueue() {

  while (length == 0)
    try {
      wait();
    } catch (InterruptedException e) {}

  PrintJob forReturning = firstToPrint;

  ....
```