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)
   Can an `abstract` class have non-`abstract` methods?

   (b) (2 points)
   Assume that `o1.hashCode()` returns the same value as `o2.hashCode()`. Must then `o1.equals(o2)` return `true`?

   (c) (2 points)
   Is it possible to write a (compilable) program such that, during execution, a *checked* exception is thrown, but never *caught* (fångad)?

   (d) (2 points)
   Assume that class `A` extends class `B`, and assume that class `B` has only one constructor `B()`, and `B()` is `public`. Can this constructor `B()` be overridden in class `A`?

   (e) (2 points)
   Assume that `ClassI`, `ClassII`, and `ClassIII` are three different classes. Is it possible that `ClassI` is a sub*type* of both `ClassII` and `ClassIII`?

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

   (a) (2 points)
   yes.
   (It is the whole point of an `abstract` class to encapsulate implementations common to all concrete sub-classes. Otherwise, an interface is preferable.)

   (b) (2 points)
   no.
   (It is the other way round:
   "If `o1.equals(o2)` returns `true`, then `o1.hashCode()` must return the same value as `o2.hashCode()`.")

   (c) (2 points)
   yes.
   (If all 'surrounding' methods, including `main`, declare this exception in their `throws` clause.)

   (d) (2 points)
   no.
   (Constructors are not inherited, and can therefore not be overridden.)

   (e) (2 points)
   yes.
   (For instance, if `ClassI extends ClassII` and `ClassII extends ClassIII`.
   Example: `FileInputStream` is a subtype of both, `InputStream` and `Object`.)

2. **Sorting** (12 points)
Many computer games keep track of the scores which players have achieved so far. A ranking of the scores is usually displayed in an order where 'better' scores are displayed higher up than 'less good' scores. When printing to a screen (like `System.out`), this means that the best scores are printed first and less good scores are printed later.

Consider the following class `Score`:

```
public class Score {

    private String name;

    private int points;

    private String date;

    public Score(String name, int points, String date) {
        this.name = name;
        this.points = points;
        this.date = date;
    }

    public String toString() {
        return (points + "  " + name + "  (" + date + ")");
    }
}
```

Also, consider the interface `HighScoreRanking`:

```
import java.util.*;

public interface HighScoreRanking {
    boolean add(Comparable c);
    Iterator iterator();
}
```

A `HighScoreRanking` object is meant to store all rankings which were added using `add`. The `Iterator` returned by `iterator()` should return these scores in an order such that the best score comes out first, then the second best, and so on. Of course, a score with more points is better than a score with less points. If the points are the same, than the one who played later is better. If the points and the dates are the same, we order the scores alphabetically after the names of the players.

An implementation of `HighScoreRanking` shall be stored in a *sorted* collection. Recall that standard iterators on *sorted* collections first return the *smallest* element, then the second but smallest, and so on. Therefore, to get the *best* score first, we must implement the order on the scores in such a way that a *better* score is *smaller* in this order.

(a) (6 points)
The method `add` of `HighScoreRanking` requires arguments of type `Comparable`.

But instances of the class `Score` are not (yet) of type `Comparable`. Change the class `Score` such that `Score` objects can be given as arguments to `add`. You are not allowed to change the interface `HighScoreRanking`.
(*Hints:* Lookup the enclosed API of `Comparable`. Implementing an `equals` is not required. Observe again the correspondence between 'smaller' and 'better' mentioned above. You can expect the `date` strings being of the form: `"YYMMDD"`, like `"031207"` or `"031130"`. Hereby, the built-in alphabetical order on strings fits to the order in time.)

(b) (3 points)
Write a class `MyScoreRanking` which `implements HighScoreRanking`. Observe the above explanations.

(c) (3 points)
Complete the following test class, like indicated by the comments:

```
import java.util.*;

public class HighScoreTest {
    public static void main(String[] args) {
        HighScoreRanking ranking = new MyScoreRanking();
        // add at least four scores to ranking
        System.out.println("Here comes the current ranking:");
        // print all scores of ranking to System.out
    }
}
```

Also, write down the output which is produced by running your `HighScoreTest`.

Answer:

(a)

```
public class Score implements Comparable {

    ...

    public int compareTo(Object o) {
        Score other = (Score) o;
        int result;
        result = other.points - points;
        if (result != 0)
            return result;
        result = other.date.compareTo(date);
        if (result != 0)
            return result;
        return name.compareTo(other.name);
    }

    ...
}
```

(b)

```
import java.util.*;

public class MyScoreRanking implements HighScoreRanking {

    private Set theScores = new TreeSet();

    public boolean add(Comparable c) {
        return theScores.add(c);
    }

    public Iterator iterator() {
        return theScores.iterator();
    }
}
```

(c)

```
import java.util.*;

public class HighScoreTest {

    public static void main(String[] args) {

        HighScoreRanking ranking = new MyScoreRanking();
        ranking.add(new Score("Martin", 700, "031207"));
        ranking.add(new Score("Michael", 900, "031208"));
        ranking.add(new Score("Ingunn", 900, "031209"));
        ranking.add(new Score("Philipp", 800, "031206"));
        ranking.add(new Score("Ingunn", 700, "031207"));

        System.out.println("Here comes the current ranking:");

        Iterator it = ranking.iterator();

        while (it.hasNext())
            System.out.println(it.next());

    }
}
```

3. **Template Methods** (12 points)
   The standard collections framework contains several *abstract* collection classes which factor out common code from concrete collection classes, and provide a basis for new implementations. For instance, the collections framework contains an abstract class `AbstractCollection` which implements `Collection`.

To simplify things, we now consider an interface `ExamCollection` which offers less methods than the standard interface `Collection`.

```
import java.util.*;

public interface ExamCollection {
    boolean add(Object o);
    void clear();
    boolean contains(Object o);
    boolean isEmpty();
    Iterator iterator();
    boolean remove(Object o);
    int size();
}
```

This interface does not have any 'optional' methods. Therefore, an implementation of the interface must 'support' all operations of the interface.

We assume that the methods offered by `ExamCollection` have exactly the same contract (API description) as the corresponding methods in `Collection`. (See also the enclosed API descriptions of `Collection` and `Iterator`. Those API descriptions are by purpose taken from older Java Platforms, to not confuse you with type parameters.)

Write an abstract class `AbstractExamCollection` which `implements ExamCollection`, and which has three `abstract` methods: `add`, `iterator`, and `size`. All other methods must be concrete (i.e. non-abstract).

Your class `AbstractExamCollection` should not have any fields (instance variables).

Recall that collections can also contain `null` references, and correspondingly the argument to `add`, `contains`, and `remove` can be `null`. We divide the task into two subtasks.

(a) (9 points) In the first version of the program `AbstractExamCollection`, you can assume that `contains` and `remove` are never called with `null` as argument. In other words, you can ignore the case where the parameter is `null`.
(Remind that `clear` is the only method which is `void`.)

(b) (3 points) Now, rewrite the methods `contains` and `remove`, such that they also react correctly if the parameter is `null`.

Answer:

(a)
```
import java.util.*;

public abstract class AbstractExamCollection
                      implements ExamCollection {

    protected AbstractExamCollection() {}

    public abstract boolean add(Object o);

    public void clear() {
```

```
        Iterator it = iterator();
        while (it.hasNext()) {
            it.next();
            it.remove();
        }
    }

    public boolean contains(Object o) {
        Iterator it = iterator();
        while (it.hasNext())
            if (o.equals(it.next()))
                return true;
        return false;
    }

    public boolean isEmpty() {
        return (size() == 0);
    }

    public abstract Iterator iterator();

    public boolean remove(Object o) {
        Iterator it = iterator();
        while (it.hasNext())
            if (o.equals(it.next())) {
                it.remove();
                return true;
            }
        return false;
    }

    public abstract int size();

}
```

(b)

```
    public boolean contains(Object o) {
        Iterator it = iterator();
        if (o == null)
            while (it.hasNext()) {
                if (it.next() == null)
                    return true;
            }
        else
            while (it.hasNext())
                if (o.equals(it.next()))
                    return true;
```

```
        return false;
    }

    public boolean remove(Object o) {
        Iterator it = iterator();
        if (o == null)
            while (it.hasNext()) {
                if (it.next() == null) {
                    it.remove();
                    return true;
                }
            }
        else
            while (it.hasNext())
                if (o.equals(it.next())) {
                    it.remove();
                    return true;
                }
        return false;
    }
```

4. **I/O and Collections** (12 points)

Write a program `CountOccurrences` which prints out how many *different* words of length 1, of length 2, and so forth, up to length 10, appear in a certain text file. We count words of length longer than 10 as if they had length 10. The text file is given as an argument.

Example:
Suppose we have the following input file `asterix.txt`:

```
The year is 50 B.C.
Gaul is entirely occupied by the Romans
Well, not entirely
One small village (andSoOnAndSoOn)
```

Then, when calling
`> java CountOccurrences asterix.txt`
the following should be printed to `System.out`:
(Output follows this line:)

```
Frequency of word lengths:
--------------------------
1: 0
2: 3
3: 4
4: 3
5: 2
```

6: 1
7: 1
8: 2
9: 0
10: 1

(End of Output, and End of Example)

We consider words being separated by any number of the usual whitespace characters, like the space character, the tab character, the newline character, and so on (all of which are normally 'invisible' in editors). The exact definition of 'invisible' or 'whitespace' characters does not matter here, as long as the mentioned ones are covered. But all 'visible' characters, including `'.'` and `','`, are part of the words. For instance, "`B.C.`" is a word of length 4, and "`Well,`" is a word of length 5. Note that, in the above example, the word "`entirely`" is only counted once. The same holds for "`is`". Also note that the word "`(andSoOnAndSoOn)`" is counted as if it had length 10.

In case any input/output problem occurs at runtime (like the argument file does not exist), the program should print an according message for the user. (To simplify your task, the program does not have to be robust against being called with the wrong number of arguments.)

Hints:

- Consider using an array of sets. (But other solutions are possible, too.)
- Consider using `BufferedReader`, and `StringTokenizer` (see the enclosed API description). 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 be continued on the next page)*
- Alternatively, you could use `Scanner`. (A summary of the `Scanner` API is enclosed.)
- Even if the default delimiters (separators) for `StringTokenizer` and `Scanner` are different, that difference does not matter here. In either case, you can just use the default delimiters (separators), meaning you do not need to specify them explicitly in any way.

Answer:

As always, there are different solutions possible. Three of them are given here. *Note that these are only partial solutions, as the exception handling performed by the programs is not sufficient. Particularly, the user is not notified in case IOExceptions are raised.*

(Variant using an Array of `Set`*s, and a* `BufferedReader`*:)*

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

public class CountOccurrences {

  public static void main(String[] args)
```

```
                        throws IOException {

    BufferedReader in =
           new BufferedReader(new FileReader(args[0]));

    Set[] occurrences = new Set[11];

    for (int i = 1; i <= 10; i++)
        occurrences[i] = new HashSet();

    String line, word;
    int len;

    while ((line = in.readLine()) != null) {
        StringTokenizer st = new StringTokenizer(line);
        while(st.hasMoreTokens()) {
            word = st.nextToken();
            len = word.length();
            if (len >= 10)
                len = 10;
            occurrences[len].add(word);
        }
    }

    in.close();

    System.out.println("Frequency of word lengths:");
    System.out.println("-------------------------");

    for (int i = 1; i <= 10; i++) {
        Set lookup = occurrences[i];
        System.out.println(i + ": " + lookup.size());
    }
  }
}
```

(Variant using an Array of Set*s, and a* Scanner*:)*

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

public class CountOccurrences {

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

    Scanner in = new Scanner(new File(args[0]));
```

```
    Set[] occurrences = new Set[11];

    for (int i = 1; i <= 10; i++)
        occurrences[i] = new HashSet();

    String line, word;
    int len;

    while(in.hasNext()) {
        word = in.next();
        len = word.length();
        if (len >= 10)
            len = 10;
        occurrences[len].add(word);
    }

    in.close();

    System.out.println("Frequency of word lengths:");
    System.out.println("-------------------------");

    for (int i = 1; i <= 10; i++) {
        Set lookup = occurrences[i];
        System.out.println(i + ": " + lookup.size());
    }
  }
}
```

(Variant using a `Map` *from* `Integer`*s to* `Set`*s, and a* `BufferedReader`*:)*

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

public class CountOccurences {

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

    BufferedReader in
           = new BufferedReader(new FileReader(args[0]));

    Map occurrences = new TreeMap();

    for (int i = 1; i <=10; i++)
        occurrences.put(new Integer(i), new HashSet());

    String line, word;
    int len;
```

```
        Integer lenWrap;

        while ((line = in.readLine()) != null) {
            StringTokenizer st = new StringTokenizer(line);
            while(st.hasMoreTokens()) {
                word = st.nextToken();
                len = word.length();
                if (len > 9)
                    len = 10;
                lenWrap = new Integer(len);
                ((Set) occurrences.get(lenWrap)).add(word);
            }
        }

        in.close();

        System.out.println("Frequency of word lengths:");
        System.out.println("-------------------------");
        Iterator it = occurrences.keySet().iterator();
        while(it.hasNext()) {
            Integer aKey = (Integer) it.next();
            Set lookup = (Set) occurrences.get(aKey);
            System.out.println(aKey + ": " + lookup.size());
        }
    }
}
```

5. **Input/Output** (14 points)

Write a program `SimpleCompress` that does a simplistic text compression. The name of the input file is given as a command line argument. The compressed output is written to another text file, the name of which is the name of the input file plus the suffix "`.ezip`" (`ezip` stands for "exam zip").

During compression, each sequence of n repeating characters c is compressed into <n/c>. For instance, the sequence `aaaaaaaaaa` is compressed into `<10/a>`.

Example:
Suppose we have the following input file `in.txt`:

```
Thisssssss            is   aaaaaaaa
testtt 22225555555555
```

Then, after calling
`> java SimpleCompress in.txt`
the file `in.txt.ezip` should look like:

```
Thi<8/s><11/ >is<3/ ><8/a>
tes<3/t> <4/2><10/5>
```

(End of Example)

In case any input/output problem occurs at runtime (like the argument file does not exist), the program should print an according message for the user. (To simplify your task, the program does not have to be robust against being called with the wrong number of arguments.)

Hint:

- The class `PrintWriter` offers the method `write(int c)` (like any other `Writer`), but also `print(String s)` for printing strings, `print(int i)` for printing integers, and similar print methods for all primitive types. A `PrintWriter` which would, for instance, write to the file `text.txt`, would be created by the expression:
  `new PrintWriter("text.txt")`

Answer: *Note that the following is only a partial solution, as the exception handling performed by the program is not sufficient. Particularly, the user is not notified in case IOExceptions are raised.*

```
import java.io.*;

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

    FileReader in = new FileReader(args[0]);
    String outname = args[0] + ".ezip";
    PrintWriter out = new PrintWriter(outname);

    int match, next, count;

    match = in.read();
    while (match != -1) {
        count = 1;
        next = in.read();
        while (next == match) {
            count++;
            next = in.read();
        }
        if (count == 1)
            out.write(match);
        else
            out.print("<" + count + "/" + (char)match + ">");
        match = next;
    }
    in.close();
    out.close();
  }
}
```