UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFR08014 INFORMATICS 1 - OBJECT-ORIENTED PROGRAMMING

Wednesday 14th May 2014

09:30 to 12:30

INSTRUCTIONS TO CANDIDATES

1. **Note that all questions are compulsory.**
2. **Remember that a file that does not compile, or does not pass the simple JUnit tests provided, will get no marks.**
3. **This is an Open Book exam.**

Convener: J. Bradfield
External Examiner: C. Johnson

THIS EXAMINATION WILL BE MARKED ANONYMOUSLY

Question 1

In this question you will implement part of a simple holiday planning application.

(a) Implement class `Cost`. This has a private integer attribute `amount` and a private `String` attribute `currency`, to represent the cost of an item in a certain currency. Your code must ensure that `amount` is non-negative, and that `currency` is one of "pounds sterling", "US dollars", "euros". (Do *not* use an enum.)

Your class must have a zero-argument constructor that sets `amount` to 0 and `currency` to "pounds sterling", and getters and setters for its attributes. If a setter is given an argument that is not acceptable, it should simply do nothing: that is, it must not change the object, but it must not report this in any way as a mistake.

Your class must also have a method `convert` taking a `String` `newCurrency` and a `double` `rate` and returning nothing. The parameter `rate` represents the number of units of the *old* currency that equal 1 of the *new* currency. If `newCurrency` is one of the acceptable currencies, the `convert` method must change the `currency` field of the object to be `newCurrency` and change the `amount` field to be the appropriately converted amount (rounded to the nearest integer, with .5s rounded up in the usual way).

Again, if the `newCurrency` argument is not acceptable, the method should simply do nothing. You may assume, and need not check, that `rate` > 0.

Finally the `toString` method should return "37 pounds sterling" if `amount` is 37 and `currency` is "pounds sterling", and so on (with a single space character between the amount and the currency, as shown).

In summary `Cost` has this public interface, with behaviour as explained above:

```
public class Cost
-------------------------------------------------
        Cost()
    int getAmount()
   void setAmount(int n)
 String getCurrency()
   void setCurrency(String s)
   void convert(String newCurrency, double rate)
 String toString()
-------------------------------------------------
```

[25 marks]

(b) You are given, in file `Holiday.java`, the code of a class `Holiday` with the following public interface:

public class Holiday	
Holiday(String destination, int days)	*constructor*
String getDestination()	*getter*
void setDestination(String destination)	*setter*
int getDays()	*getter*
void setDays(int days)	*setter*
String toString()	*return "Destination (days)" e.g. "Glasgow (14)"*

Your task is to implement the class `ActivityHoliday` to extend `Holiday`. This class should have an additional private attribute `activities` of type `HashMap<String, Cost>`, representing the names and costs of various optional activities offered on the holiday, in terms of the Cost class you implemented in part (a). In the constructor of `ActivityHoliday` you must initialise `activities` with an initial capacity for 10 activities. Your constructor must, like the constructor of `Holiday`, take a `String destination` and an `int days`, and must pass these to `Holiday`'s constructor.

Remember that you will need the line

```
import java.util.HashMap;
```

at the top of your file.

Provide a public method `addActivity`, taking a `String` representing the name of the activity and a `Cost` representing its cost, that adds an activity to `activities`, if necessary replacing the existing activity of that name. You should assume that the `Cost` object is being maintained elsewhere, e.g., updated if the cost of the activity changes; make sure you keep a reference to, not a copy of, the given `Cost` object.

Override the method `toString()` so that it:

(a) first gives the destination and length of the holiday exactly as `Holiday`'s `toString()` does;

(b) then on a new line by itself gives "Activities:";

(c) then lists the activities in this holiday, each on a new line, by giving the key string of the activity, then one space, then the cost of the activity as given in the corresponding value.

Take care that `toString()`'s output does *not* end with a new line.

For example, if the holiday is 14 days in Glasgow, and the activities available are called "Hunterian Museum" and "Mackintosh House", each at a cost of £5, then sending `toString()` to an object representing this holiday should result in:

```
Glasgow (14)
Activities:
Hunterian Museum 5 pounds sterling
Mackintosh House 5 pounds sterling
```

(or:

```
Glasgow (14)
Activities:
Mackintosh House 5 pounds sterling
Hunterian Museum 5 pounds sterling
```

– the order in which activities are listed does not matter).

In summary `ActivityHoliday` has this public interface, with behaviour as explained above:

```
public class ActivityHoliday
-------------------------------------------------------
       ActivityHoliday(String destination, int days)
  void addActivity(String name, Cost cost)
String toString()
-------------------------------------------------------
```

[*25 marks*]

The files that you must submit for this question are the following:

- `Cost.java`
- `ActivityHoliday.java`

Question 2

Let $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ be lists of integers, and let a *kerfuffle* of the two lists be a real number given by the sum $a_1 b_{i_1} + a_2 b_{i_2} + \ldots + a_n b_{i_n}$, where each a_i and each b_j is used exactly once: that is, the list $i_1 \ldots i_n$ is some rearrangement of $1 \ldots n$.

The *rearrangement inequality* states that any kerfuffle of two lists is less than or equal to the kerfuffle given by putting the largest a_i with the largest b_i and so on, down to the smallest a_i with the smallest b_i.

In this question you will write code to demonstrate the rearrangement inequality.

Note: it is expected that you may want to use one or more methods of the Arrays class that you may not have used before. You should consult the Java documentation whenever you need to. **Hint:** you are likely to find a method that sorts an array useful. To use the Arrays class, you will need to add the line:
`import java.util.Arrays;`
at the top of your file.

(a) Create a public class `Rearrangement`.

In the class `Rearrangement`, implement the public static method

```
int dotWith(int[] a, int[] b)
```

that, if `a` and `b` have the same length n, returns $\Sigma_{i=1}^{i=n} a_i b_i$, that is, the kerfuffle that takes the elements of the arrays in the orders given. (You might recognise this as the dot product of `a` and `b` regarded as vectors, hence the name of the function, but you are not required to remember any Linear Algebra for this question.)

If `a` and `b` do not have the same length, return 0.

Expected behaviour:

```
dotWith(new int[] {2, 1}, new int[] {3, 4})
```

should return 10, because this is $2 \times 3 + 1 \times 4$.

```
dotWith (new int[] {2, 1, 3}, new int[] {3, 4})
```

should return 0, because the input arrays have different lengths. [*10 marks*]

(b) In the class `Rearrangement`, implement the public static method

```
void rotate (int[] b)
```

that modifies the given array by moving each of its elements one place to the right, except for the last element, which is moved to the beginning of the array.

Expected behaviour:

```
int[] b = new int[] {1, 2, 3};
rotate(b);
```

should return nothing, but afterwards b should be $\{3, 1, 2\}$. [*10 marks*]

(c) In the class `Rearrangement`, implement the public static method

```
int useRotations(int[] a, int[] b)
```

that performs all possible rotations on `b` (using your `rotate` method), checks the result of `dotWith` applied to `a` and each rotation, and returns the highest value found. If `a` and `b` do not have the same length, return 0.

Expected behaviour:

```
useRotations (new int[] {2, 1}, new int[] {3, 4})
```

should return 11, because this is $2\times4+1\times3$, which is greater than $2\times3+1\times4 = 10$.

```
useRotations (new int[] {2, 1, 3}, new int[] {3, 4})
```

should return 0, because the input arrays have different lengths. [*10 marks*]

(d) In the class `Rearrangement`, implement the public static method

```
int useSorted(int[] a, int[] b)
```

that returns the result of sorting the elements of `a` and of `b` and applying `dotWith` to the resulting sorted arrays. (Note that it does not matter whether the arrays are sorted from least to greatest or vice versa, provided both are sorted in the same order.)

If `a` and `b` do not have the same length, return 0.

Expected behaviour:

```
useSorted (new int[] {2, 1}, new int[] {3, 4})
```

should return 11, because this is $1 \times 3 + 2 \times 4$.

```
useSorted (new int[] {2, 1, 3}, new int[] {3, 4})
```

should return 0, because the input arrays have different lengths. [*10 marks*]

(e) Finally, in the class `Rearrangement`, implement the public static `main` method that will exercise your methods. The user will give a list of arguments on the command line, all of which will be integers. **You are not required to provide any error handling in this part.**

The first argument, say n, is the length of each of two arrays. The next n arguments are the elements of the first array. The remaining n arguments are the elements of the second array. For example, if the user runs the program as

```
Rearrangement 2 2 1 3 4
```

then you will be investigating the kerfuffles of the arrays `[2,1]` and `[3,4]`.

Once your `main` method has parsed the input and built the arrays, it must do three calculations and print their results to standard output. Use exactly the given format, and note that there is a single space after each colon.

(a) Calculate `dotWith` on the two given arrays, and print the result: in the example of user input given above we expect:

```
dotWith gave: 10
```

(b) Calculate `useRotations` on the two given arrays, and print the result: in the example, we expect:

```
useRotations gave: 11
```

(c) Calculate `useSorted` on the two given arrays, and print the result: in the example, we expect:

```
useSorted gave: 11
```

If you have done everything correctly, you will find that these numbers are weakly increasing, for any correctly formatted input. [*10 marks*]

The file that you must submit for this question is:

- `Rearrangement.java`

Final Checklist

You are reminded again that any file that does not compile, or does not pass the simple tests provided to you, will get no marks. You are advised to check this just before each submission you make.
Here is a complete list of all the files that you must submit for this exam:

```
Cost.java
ActivityHoliday.java
Rearrangement.java
```