Sarajevo School of Science and Technology
CS/IS 160 Programming and Problem Solving II
June 2015

Test 3 Practice Problem Solutions

Problem 1: Short answers (20 points).

Answer the following questions:

(a) What is a default constructor?

A default constructor is a constructor with no arguments provided by the compiler, which initializes class instance variables to their default values.

(b) What is an interface?

An interface in Java describes a set of methods that a class needs to implement to satisfy that interface. Interfaces do not provide implementation of its methods.

(c) What is the difference between the `throws` clause and the `throw` statement?

The `throws` clause informs the compiler that a method throws one or more exceptions. The `throw` statement causes an exception to be thrown.

(d) What is the difference between a checked exception and an unchecked exception?

Unchecked exceptions are those that are derived from the `Error` or from the `RuntimeException` class. All of the remaining exceptions are checked exceptions, which you have to handle in your program by either declaring your method to throw them or by using a try-catch handler block.

State whether each of the statements that follows are *true* or *false*. If *false*, explain why.

(e) A method declared *final* in a superclass can be overridden in a subclass.

False. A method declared final in a superclass cannot be overridden in a subclass.

(f) An *abstract* class can contain more than one *abstract* method.

True.

(g) The superclass constructor always executes before the subclass constructor.

True.

Write the following simple declarations and code:

(h) Declare and initialize an array list variable `intList` that can dynamically hold integer numbers.

```
ArrayList<Integer> intList = new ArrayList<Integer>();
```

(i) Write a "bad" code segment that generates an `ArithmeticException`.

```
int badResult = 1 / 0;
```

Problem 2: Diagonal entries of a square matrix (20 points).

Implement a static method called `diagonal`, which returns the diagonal elements of an input square integer matrix. For example, if the input matrix is

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 2 & 3 \end{bmatrix},$$

then the diagonal elements are returned as the array (1, 1, 3).

Use the following method prototype, where `matrix` is the input square matrix with integer entries:

```
public static int[] diagonal( int[][] matrix )
```

Solution: We use a single indexing variable `i` to iterate over the diagonal elements of the input matrix and copy their values into an output array. Note that the elements on the main diagonal have the same row and column index.

```
public static int[] diagonal( int[][] matrix ) {

      int[] diagArray = new int[matrix.length];

      for (int i = 0; i < matrix.length; i++) {
            diagArray[i] = matrix[i][i];
      }

      return diagArray;
}
```

Problem 3: Count duplicate elements in an array (20 points).

Implement a static method `countDuplicates` that returns the number of duplicate elements in an integer array. For example, if the input array is (1, 2, 2, 3, 5, 1), then this method returns 2, since there are two duplicate elements (1, 2) in the array.

Use the following method prototype, where `array` is an input integer array:

```
public static int countDuplicates( int[] array )
```

Solution: First we consider the case when we count the total number of duplicate elements. For example, if the input array is (1, 2, 2, 2, 2, 3, 5, 1), then this method needs to return 4, since there are four duplicate elements (1, 2, 2, 2) in the array.

The first solution method we present uses dynamic array list to find all unique elements in the input array and then we get the total number of duplicates by subtracting the number of unique elements from the total number of the input elements.

```
public static int countDuplicates(int[] array) {

      // array list to keep track of unique elements in the array
      ArrayList<Integer> uniques = new ArrayList<Integer>();

      for( int element : array ) {
            if( ! uniques.contains(element)) {
                  uniques.add(element);
            }
      }

      // number of total duplicates equals to the
      // total number of input elements - total number of uniques
      return array.length - uniques.size();
}
```

The second solution method we present first sorts the array and then it counts equal consecutive elements. At the end, this count will give us the total number of duplicates.

```
public static int countDuplicatesWithSorting( int[] array ) {

      // first copy array and sort its copy
      // (so we do not change the original array)
      int[] arr = Arrays.copyOf(array, array.length);
      Arrays.sort(arr);

      // start counting consecutive elements that are equal,
      // those are duplicates
      int count = 0;
      for (int i = 0; i < arr.length-1; i++) {
            if( arr[i] == arr[i+1]) {
                  count++;
            }
      }

      return count;
}
```

Finally, we present a solution method for our problem variation where we are only interested in unique duplicate elements; that is, if there are multiple duplicates of an element we should only report it once. For example, if the input is (1, 2, 2, 2, 2, 3, 5, 1), then the method should return 2, since there are two unique duplicate elements (1, 2). Our solution uses two array lists to keep track of the unique and duplicate elements.

```
public static int countUniqueDuplicates(int[] array) {

      // array lists to keep track of uniques and duplicates
      ArrayList<Integer> uniques = new ArrayList<Integer>();
      ArrayList<Integer> dups = new ArrayList<Integer>();

      for( int element : array ) {

            if( ! uniques.contains(element) ) {
                  // add element to uniques, first time we see it
                  uniques.add(element);
            }
            else if( ! dups.contains(element) ) {
                  // element is in uniques, but not in dups, add it
                  dups.add(element);
            }
            else {
                  // ignore duplicate element, already in dups
            }
      }

      return dups.size();
}
```

Problem 4: Abstract and concrete classes (20 points, each part 10 points).

(a) Define an abstract class `Animal` with a private instance variable `age` (integer) and an abstract method `sound` that has no arguments and returns a string.

Implement a getter and setter for the age variable and make sure that it must be set to a positive integer value or zero. If that is not the case, throw a `RuntimeException` with an appropriate message.

Therefore, the `Animal` class should have the following method signatures:

```
public int getAge()
public void setAge(int age) throws RuntimeException
abstract String sound()
```

Solution: We implement the abstract `Animal` class as follows.

```
public abstract class Animal {

      private int age;

      // getter
      public int getAge() {
            return age;
      }

      // setter
      public void setAge(int age) throws RuntimeException {
         if (age > 0 ) {
             this.age = age;
         }
         else {
             throw new RuntimeException("Age should be positive");
         }
      }

      // abstract method
      abstract String sound();

}
```

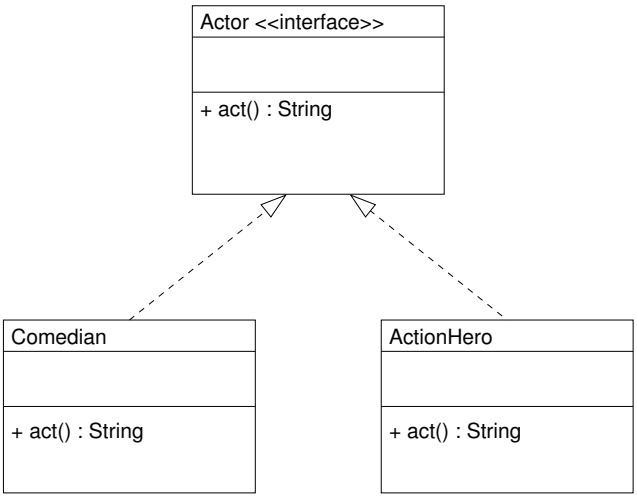
(b) Next define two concrete classes that extend the `Animal` class: `Dog` and `Duck`. Implement their `sound` method to return an appropriate sound specific to that species.

Solution: Here is our implementation of the concrete `Dog` and `Duck` classes.

```
public class Dog extends Animal {

      public String sound() {
            return "Woof, woof";
      }
}

public class Duck extends Animal {

      public String sound() {
            return "Quack, quack";
      }
}
```

Problem 5: Inheritance and interfaces (20 points).

Implement the class hierarchy shown below, where the `Actor` is an interface and the `Comedian` and `ActionHero` classes implement the interface. The `act` method for the `Comedian` class should return the string "I'm Austin Powers," while the `ActionHero` class should return "My name is Bond, James Bond."



Solution: The following code implements the `Comedian` and `ActionHero` classes. We first present the `Actor` interface definition and then the concrete classes.

```
public interface Actor {
      String act();
}

public class Comedian implements Actor {
      public String act() {
            return "I'm Austin Powers";
      }
}

public class ActionHero implements Actor {
      public String act() {
            return "My name is Bond, James Bond";
      }
}
```