Sarajevo School of Science and Technology
CS/IS 160 Programming and Problem Solving II
June 8, 2015

Test 3 Solutions

(5% of the final grade, 90 minutes)

Problem 1: Short answers (20 points).

Answer the following questions (each part 2 points):

(a) What is an interface in Java?

Solution: 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.

(b) Why would we specify an exception superclass type such as `Exception` as the exception type variable in a `catch` block?

Solution: This enables us to catch related types of exceptions and process them in a uniform manner. However, it is also often useful to process the subclass type exceptions individually for a more precise exception handling.

(c) What are main differences between the `List` and `Set` data structures in Java?

Solution: In Java, `List` is an ordered collection that can contain duplicate elements, while `Set` is a collection of unique elements without a particular element order.

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

(d) A `Map` can contain duplicate keys.

Solution: False. Java's `Map` data structure cannot contain duplicate keys.

(e) A `Map` can contain duplicate values.

Solution: True.

(f) An `ArrayList` can contain duplicate values.

Solution: True.

(g) Values of primitive types such as `int`, `char` and `double` may be stored directly in a Java collection such as `ArrayList`.

Solution: Accepted either *True* or *False* here. Java's `ArrayList` collection can only store reference type variables (objects) such as `Integer`, `Character`, `String`, *etc*. However, it can appear to store primitive types using auto-boxing, a process that converts the primitive type into its equivalent reference type (for example, `int` is converted into its equivalent `Integer` type).

Write the following simple declarations and code (each part 3 points):

(h) Declare and initialize an ArrayList `strings` that can dynamically hold strings.

Solution: `ArrayList<String> strings = new ArrayList<String>();`

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

Solution: The following code segments will cause a null pointer exception.

```
String nullString = null;
int num = nullString.length();
```

Problem 2: Number of unique characters in a string (20 points).

Implement a static method `countUniqueChars`, which finds the number of unique characters in an input string.

For example, the number of unique characters for `"abraca"` is 4 since it contains characters `(a,b,c,r)`. You can obtain the character at the specified index in a string using the `charAt` method and its length using the `length` method, *e.g.*,

```
String str = "abraca";
char firstChar = str.charAt(0);
int strLength = str.length();
```

Use the following method prototype:

```
public static int countUniqueChars( String s )
```

Solution: We present couple of different solutions to this problem, which are both equally good. The first solution uses Java's `Set` collection to count the number of unique characters in the input string.

```
public static int countUniqueChars( String s ) {

      Set<Character> uniqueChars = new HashSet<Character>();

      for( int i=0; i < s.length(); i++ ) {
            uniqueChars.add(s.charAt(i));
      }

      return uniqueChars.size();
}
```

The second solution converts the input string to an array of characters, sorts it, and then counts different consecutive characters.

```
      public static int countUniqueChars( String s ) {

            // check if the input string is empty
            if(s.isEmpty()) {
                  return 0;
            }

            // convert the string to char array and sort it
            char[] letters = s.toCharArray();
            Arrays.sort(letters);

            // count each consecutive letter that is different
            int counter = 1;
            for( int i=0; i < letters.length - 1; i++ ) {
                  if(letters[i] != letters[i+1])
                        counter++;
            }

            return counter;
      }
```

Problem 3: Downsample a list by a factor of two (20 points).

Implement a static method `downsample`, which downsamples an input integer list by a factor of 2. For example, if the input list is (1, 2, 3, 4, 5, 6), then the output list is (1, 3, 5).

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

```
      public static List<Integer> downsample( List<Integer> list )
```

Solution: We present a solution where we use another `ArrayList` collection object to keep every other element from the original input list.

```
public static List<Integer> downsample( List<Integer> list ) {

      List<Integer> result = new ArrayList<Integer>();

      for( int i=0; i < list.size(); i+=2 ) {
            result.add(list.get(i));
      }

      return result;
}
```

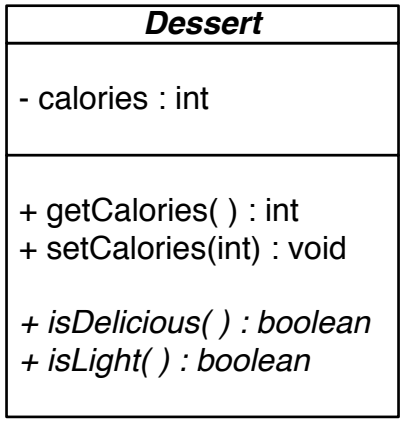
Problem 4: Abstract Dessert and concrete Baklava class (20 pts, each part 10 pts).

(a) Define an abstract class `Dessert` with getters and setters for its private instance variable `calories` (integer). Make sure that the calories variable must contain a positive integer value. If that is not the case, throw a `RuntimeException` with an appropriate message.

Also define abstract public methods `isDelicious` and `isLight` that have no arguments and return Boolean values. Therefore, the class should have the following method signatures and the class diagaram:

```
public int getCalories()
public void setCalories(int cal) throws RuntimeException

abstract boolean isDelicious()
abstract boolean isLight()
```



Solution: An implementation of the abstract `Dessert` class is shown below.

```
public abstract class Dessert {

   // instance variable
   private int calories;
```

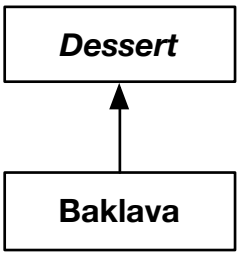
```
    // getter
    public double getCalories() {
        return calories;
    }

    // setter
    public void setCalories(int cal) throws RuntimeException {
        if(cal > 0) {
            calories = cal;
        }
       else {
         throw new RuntimeException("Calories must be positive.");
       }
    }

    // abstract methods
    abstract boolean isDelicious();
    abstract boolean isLight();

}
```

(b) Next define a concrete class `Baklava` that extends the `Dessert` class. Implement its `isDelicious` method to always return true. Implement its `isLight` method to return true if the baklava has less than 200 calories, and false otherwise.



Solution: An implementation of the concrete `Baklava` class is shown below.

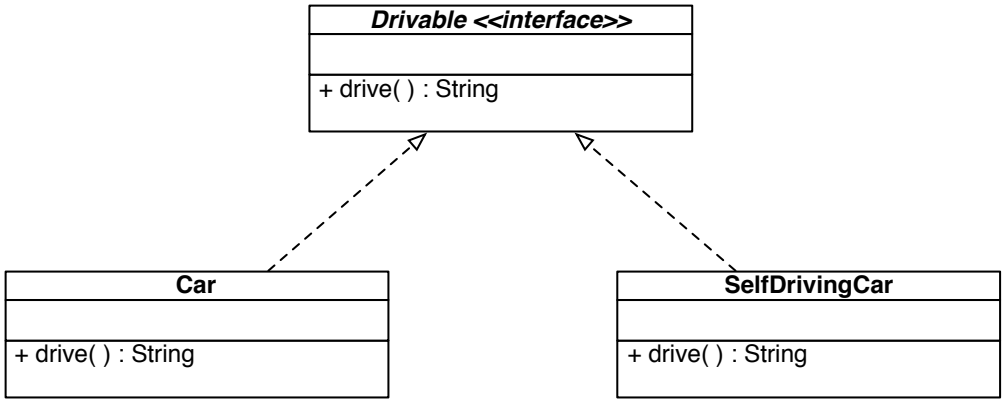
```
public class Baklava extends Dessert {

      @Override
      public boolean isDelicious() {
            return true;
      }

      @Override
      public boolean isLight() {
            return (getCalories() < 200);
      }
}
```

Problem 5: Inheritance and interfaces (20 points).

Implement the class hierarchy shown below, where the `Drivable` is an interface and the `Car` and `SelfDrivingCar` classes implement the interface. The `drive` method for the `Car` class should return the string "Regular car" while the `SelfDrivingCar` class should return "Look, I am a Google self-driving car."



Solution: The following code implements the `Car` and `SelfDrivingCar` classes. We first present the `Drivable` interface definition and then the concrete classes.

```
public interface Drivable {

	public String drive();
}

public class Car implements Drivable {

	@Override
	public String drive() {
		return "Regular car.";
	}
}

public class SelfDrivingCar implements Drivable {

	@Override
	public String drive() {
		return "Look, I am a Google self-driving car.";
	}
}
```