SYSC 2004 Object-0riented Software Development

Lab 8

Lab 8:

Background Reading

- *Objects First with Java*, Chapter 10.

The objective of this lab is to reinforce your understanding of inheritance.

Getting Started

1. Download file `lab8-counters.zip` from cuLearn. Save the file to the desktop.
2. Right-click on the `lab8-counters.zip` folder and select Extract All... to extract all the files into a folder called `lab8-counters`.
3. Launch BlueJ.
4. Open the *lab8-counters* project in BlueJ.

Part 1 - Exploring `RollOverCounter`

Suppose we want an object that models a hand-held counter. The counter has two buttons:

- the *reset button* clears the current count to 0
- the *count-up button* increments the current count by 1

Most physical counters have a limited range; e.g., a counter with a 3-digit display can count between 0 and 999, inclusive. When the count is 999 and the count-up button is pressed, the count "rolls over" to 0.

`RollOverCounter` is an implementation of this type of counter. By default, the counter counts between 0 and 999. A two-argument constructor provides a way to construct a counter that counts between specified minimum and maximum values.

1. Read the Java source for `RollOverCounter`. How would you create a `RollOverCounter` object that counts between 1 and 5, inclusive (don't create the object, just identify which constructor you would use). How may instance variables (fields) would this object have? What would their initial values be?
2. Interactively create the `RollOverCounter` object that was described in the previous step, and save it on the object workbench. Open an inspector on this object. How many

fields does the object have? What values are stored in these fields? Were your answers to the questions in Step 1 correct?

3. Interactively invoke the counter's methods. Before you invoke an accessor method, predict the value that will be returned. Is your prediction correct? Before you invoke a mutator method, predict how the object's state will change? Is your prediction correct? Make sure you understand the implementation of every method.

Part 2 - Exploring `LimitedCounter`

Now suppose we want a limited-counter object that is similar to a roll-over counter, except that it stops counting when it reaches its maximum value. This counter must be reset to its minimum value before it can resume counting. For example, if a limited-counter counts between 0 and 999, pressing the count-up button has no effect after the count reaches 999. Instead, the reset button must be pressed to return the count to 0.

`LimitedCounter` is an implementation of this type of counter. By default, the counter counts between 0 and 999. A two-argument constructor provides a way to construct a counter that counts between specified minimum and maximum values.

Repeat the steps in Part 1, this time using class `LimitedCounter` instead of `RollOverCounter`.

Part 3 - A Counter Superclass

Compare the `RollOverCounter` and `LimitedCounter` classes. The two classes have identical fields, as well as identical constants that define the default minimum and maximum counter values. The constructor bodies and all the methods except `countUp()` are identical. The `countUp()` methods are different, because they are specialized to provide the required counting behaviour for each class.

In this part, we're going to move the code that is common to both counter classes into a common superclass.

1. Define a new class called `Counter`. Use the editor's copy-and-paste feature to copy all of the code in `RollOverCounter` to `Counter`.
2. Edit `Counter`, as follows:
    a. Change the name of the class to `Counter`.
    b. Change the names of the two constructors to `Counter`.
    c. Delete the `countUp()` method.
    d. Add the following method:

```
/**
 * Increment this counter by 1.
 */
public void incrementCount()
```

```
    {
        count++;
    }
```

3. Verify that your `Counter` class has 3 fields, two constants, two constructors, and seven methods.
4. Compile `Counter`. If you have compile errors that you cannot resolve, ask for help.
5. Get part 3 checked by a TA.

Part 4 - Inheritance and `RollOverCounter`

1. Edit `RollOverCounter`, as follows:
    a. Change the statement `public class RollOverCounter` so that this class is now a subclass of `Counter`
    b. Delete all fields and constants.
    c. Don't change the constructors.
    d. Delete all methods except `countUp()`.
    e. Comment out `countUp()` (e.g., put `//` in front of the method declaration and every statement in the method body).
2. Verify that your `RollOverCounter` class has no fields and constants, two constructors, and one commented-out method (`countUp()`).
3. Compile `RollOverCounter`. What error message to you see? What caused this error?
4. Edit the `RollOverCounter` constructors so that they use `super()` to invoke the constructor in class `Counter`. Remember, `super()` can be called without parameters, or it can be called with a parameter list. For each constructor, pick the most appropriate form of `super()`. Edit and compile your class until it compiles without errors. `countUp()` should remain commented out.
5. Using your edited class, how would you create a `RollOverCounter` object with counting limits of 1 and 5 (don't create the object, just identify which constructor you would use). How may instance variables (fields) would this object have? What would their initial values be?
6. Interactively create the `RollOverCounter` object that was described in the previous step, and save it on the object workbench. Open an inspector on this object. How many fields does the object have? What values are stored in these fields? Were your answers to the questions in Step 5 correct?
7. Interactively invoke the counter's methods. (When you right-click on the object, select the inherited from Counter submenu to see the list of methods). Before you invoke an accessor method, predict the value that will be returned. Is your prediction correct? Before you invoke a mutator method, predict how the object's state will change? Is your prediction correct? What method must you invoke to cause the counter to count up? What happens to the object's state when the current count is 5, and you ask the counter to count up? What you observe isn't what you'd expect from a `RollOverCounter`. What caused the problem?

8. Remove the `//`'s from the front of the statements in `countUp()` method. Compile `RollOverCounter`. What error message do you see? What caused this error?
9. In class `RollOverCounter`, rewrite `countUp()` so that it provides the same counting operation as the original version of this class. In other words, when the count is at its maximum value, counting up should cause the count to roll over to its minimum value.When doing this step, **do not modify class `Counter`**. Remember that inherited methods can be invoked using internal method calls.
10. Edit and compile your class until it compiles without errors. Verify that your new version of `countUp()` is correct.
11. Get part 4 checked by a TA.

Part 5 - Inheritance and `LimitedCounter`

1. Repeat the steps in Part 4, this time using class `LimitedCounter` instead of `RollOverCounter`.
2. Get part 5 checked by a TA.

Part 6 - Javadoc Comments

1. If you didn't do so while editting the code, go back and review all the classes and methods to ensure that all three classes have complete Javadoc comments.
2. Get part 6 checked by a TA.
3. You will use your Lab 9 solution in Lab 10, so be sure to save your work.
