SYSC 2004 Object-0riented Software Development

Lab 9

Lab 9:

Background Reading

- *Objects First with Java*, Chapters 10-12.

The objective of this lab is reinforce your understanding of inheritance and polymorphism.

Getting Started

1. Download file `lab9-counters.zip` from cuLearn. Save the file to the desktop.
2. Locate your solution to Lab 8 and put the `lab8-counters` folder on the desktop.
3. Right-click on the `lab9-counters.zip` folder and select Extract All... to extract all the files into a folder called `lab9-counters`.
4. Launch BlueJ.
5. Open the *lab9-counters* project in BlueJ.
6. You need to include your work from Lab 8 in this project:
    - From the BlueJ main menu, select Edit > Add Class from File... A dialogue box will open.
    - Browse to the `lab8-counters` folder that contains your work from Lab 8. Select class `Counter`, and click the Add button. A copy of the class will be placed in the *lab9-counters* project. (This will not delete the `Counter` class from your *lab8-counters* project, and any changes that you make to `Counter` in *lab9-counters* will not affect the original class in *lab8-counters*.)
7. Repeat step 6 for class `LimitedCounter` and again for class `RollOverCounter`, both of which you also implemented in Lab 8.

Part 1 - Revisiting The Counter Class Hierarchy

1. Method `testAllRollOverCounterMethods()` in `CounterTest` tests all of the methods in `RollOverCounter`. Open the source code for the editor and read the method. Run the unit test. It should pass.
2. In `CounterTest`, change the type of field `c1` from `RollOverCounter` to `Counter`.

`c1` is a polymorphic variable, so we can assign this variable a reference to objects of any subtype (subclass) of `Counter`. For example, `setUp()` contains the statement:

```
c1 = new RollOverCounter(1, 10);
```

Compile `CounterTest`. When the Java compiler attempts to compile the statement:

```
c1.countUp();
```

you should get an error: cannot find symbol - method countUp().

`setUp()` initializes `c1` to refer to a `RollOverCounter` object, and `RollOverCounter` has a `countUp()` method, so why does this error occur? Hint: what is the static type of `c1`? What is the dynamic type of `c1`?

4. In `Counter`, add a method called `countUp()`. This method should have an empty body; i.e., this method should not alter any of the fields defined in `Counter`. Recompile the project. it should now compile. Why? Run the unit test. It should nowpass.

   Explain why

   ```
   c1.countUp();
   ```

   causes the count to be incremented, even though `c1` is declared to have type `Counter`, and the `countUp()` method you just added to `Counter` does no useful work.

5. In `Counter`, delete the `countUp()` method you wrote in the previous step, and rename `incrementCount()` to `countUp()`. In `RollOverCounter`, modify `countUp()` to invoke the `countUp()` method in `Counter`. Recompile the project. `RollOverCounter` should compile without errors, but you'll get an error when the Java compiler compiles `LimitedCounter`. Why? Edit `LimitedCounter` to fix this problem. Run the unit test. It should pass. If it doesn't, ask for help.
6. We don't intend to create instances of the `Counter` class, but currently there's nothing to stop us from doing this. To verify this, interactively create a `Counter` object.
7. In the lectures you've learned that classes that we don't intend to instantiate, and which are intended only to be used as superclasses, are abstract classes. Change the declaration of `Counter` so that it is an abstract class:

   ```
   public abstract class Counter
   {
      ...
   }
   ```

   Recompile the project. Try to interactively create a `Counter` object. Were you successful?

8. Suppose we want to be able to invoke a `countDown()` method on instances of both counter subclasses. This method will be similar to `countUp()`, except that it will decrement the counter instead of incrementing it. One way to do this is to define the method in superclass `Counter`.

   Add these two methods to `Counter`:

```
/**
 * Decrement this counter by 1.
 */
public void countDown()
{
        count--;
}

/**
 * Sets the counter to its maximum value.
 */
public void setToMaximum()
{
    count = maximumCount;
}
```

   Add this method to `RollOverCounter`:

```
/**
 * Decrement this counter by 1. If we've reached the minimum count,
 * invoking this method rolls the counter over to its maximum value.
 */
public void countDown()
{
    if (isAtMinimum()) {
        setToMaximum();
    } else {
        super.countDown();
    }
}
```

9. Interactively create a `RollOverCounter` object that counts between 1 and 5, inclusive. Invoke `countDown()` on this object. Verify that a `RollOverCounter` counts down properly, wrapping around from the minimum count to the maximum count.
10. Interactively create a `LimitedCounter` object that counts between 1 and 5, inclusive. Invoke `countDown()` on this object. (You'll find the method on the *inherited from Counter* submenu.) Use an inspector to examine the state of the object. The value of field `count` will be 0, but that's not what we want. Why is the value stored in `count` less than 1 (i.e. less the specified minimum value for this counter)?
11. You now observed that, even though a subclass inherits methods from its superclass, an inherited method does not necessarily implement the correct behaviour for the subclass. Notice that the compiler does not (and cannot) tell us that we need to override

`countDown()` in `LimitedCounter` to provide the correct count-down operation for that class.

12. In `Counter`, rename `countDown()` to `decrementCount()`. In `RollOverCounter`, modify `countDown()` to invoke `decrementCount()`. Recompile the project.
13. Interactively create a `LimitedCounter` object (**not** a `RollOverCounter` object) that counts between 1 and 5, inclusive. We want to be able to invoke `countDown()` on this counter, but we can't. Why?
14. In `Counter`, define `countDown()` as an abstract method:

    ```
    public abstract void countDown();
    ```

    Recompile the project. What compilation error occurs? Why?

15. Correct this error by implementing `countDown()` in `LimitedCounter`. Recompile the project, and test your method interactively.
16. Get Part 1 checked by a TA.

What You Have Learned

1. (Steps 1 through 3) When compiling statements that contain a variable whose type is a class, the compiler always uses the variable's *static* type. At run-time, polymorphic method dispatch always uses the variable's *dynamic* type.
2. (Step 4) If a class overrides an inherited method, the method in the subclass can invoke the method in the superclass using the keyword `super`.
3. (Steps 5 and 6) You can create subclasses of abstract classes. At run-time, you **cannot** create objects from abstract classes.
4. (Steps 7 through 12) Defining a method in a superclass is one way to ensure that all subclasses have the method, but this does not guarantee that the behaviour of the inherited method is appropriate for the subclass.
5. (Steps 13 and 14) If an abstract superclass defines an abstract method, the compiler checks that a concrete implementation of the method is defined in all concrete subclasses. The methods in the subclasses will need to invoke accessor or mutator methods that are defined in the superclass in order to obtain or change the values of the private fields that are defined in the superclass.

If you don't have a solid understanding of all the concepts listed this summary, repeat the lab exercise, and ask for help when you reach something you don't understand.

Part 2 - 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 2 checked by a TA.

Part 3 - Improve the Unit Tests

1. Modify `CounterTest` to thoroughly test your revised implementations of `RollOverCounter` and `LimitedCounter`. A suggested approach follows:
    a. Add method `testAllLimitedCounterMethods()`, which will be very similar to `testAllRollOverCounterMethods()`, except for minor changes due to the different nature of the two counters.
    b. Add method `testNewRollOverCounterMethods()` that will test the new methods in class `RollOverCounter`, i.e. method `setToMaximum()` (inherited from class `Counter`), and method `countDown()`.
    c. Add method `testNewLimitedCounterMethods()` that will test the new methods in class `LimitedCounter`, i.e. method `setToMaximum()` (inherited from class `Counter`), and method `countDown()`. It will be very similar to method `testNewRollOverCounterMethods()`.
2. Don't forget your Javadoc comments!
3. Get Part 3 checked by a TA.
