Carleton University
Department of Systems and Computer Engineering
SYSC 2004 - Object-Oriented Software Development - Winter 2015

Lab 5 - Class Inheritance

Objective

- To understand the syntax and semantics of class inheritance in Java.
- To add a new class to an existing class hierarchy.

In Exercises 1-5, you will explore and extend the class inheritance example that was presented in last week's lectures; i.e., you'll investigate the use of inheritance in classes that model different kinds of counters. In Exercises 6 and 7, you'll apply your knowledge of inheritance to the design and implementation of a new counter subclass.

Attendance/Demo

To receive credit for this lab, you must demonstrate your work. **Also, you must submit your lab work to cuLearn**. (Instructions are provided in the *Wrap Up* section at the end of this handout.)

When you have finished all the exercises, call a TA, who will grade the code you wrote. For those who don't finish early, a TA will ask you to demonstrate whatever code you've completed, starting about 30 minutes before the end of the lab period. **Any unfinished exercises should be treated as "homework"; complete these on your own time, before Lab 6.**

References

Objects First with Java

- Chapter 8 (*Improving Structure with Inheritance*)
- Chapter 9 (*More About Inheritance*)

Getting Started

Step 1: Create a new folder named Lab 5.

Step 2: Download counters.zip from cuLearn and move it to your Lab 5 folder.

Step 3: Extract the *counters* project from the zip file into a project folder named counters.

Step 4: Open the counters folder. Double-click on the BlueJ project file, which is named package. This will launch BlueJ and open the *counters* project.

Exercise 1 - Exploring Class `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

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.

Class `RollOverCounter` is an implementation of this type of counter.

Step 1: Open class `RollOverCounter` in an editor window and read the source code. How many fields (instance variables) and methods are defined in this class?

Step 2: This class defines two constructors. The zero-parameter constructor initializes the newly created counter so that it counts between 0 and 999, and the two-parameter constructor initializes the counter so that it counts between the specified minimum and maximum values.

Create a `RollOverCounter` object that counts between 1 and 5, inclusive, and place it on the object bench.

Step 3: Open an object inspector on the `RollOverCounter` object and move it, if necessary, so that it doesn't overlap the main BlueJ window. How many fields are displayed in the object inspector? What are their names? (Record the answers - you'll need them in Exercise 3.)

Step 4: You now going to ask the `RollOverCounter` object to count up. Using the object's pop-up menu, call the `countUp` method. Which of the fields displayed in the object inspector change when you call `countUp`? Repeatedly call `countUp` and use the object inspector to verify that the `RollOverCounter` counts from 1 to 5, then rolls back to 1.

Step 5: You can call several other methods on a `RollOverCounter` object; for example, method `getCount` returns the current value of the counter, and method `reset` sets the counter to its minimum value. Right-click on icon for the `RollOverCounter` object and explore the pop-up menu until you find the list of methods. Experiment with the `RollOverCounter` object; that is, call all of its methods while using the object inspector to observe its state, until you understand the counter's behaviour.

Exercise 2 - Exploring Class `LimitedCounter`

Now suppose we want to model a counter object that increments by 1 each time the count-up button is pushed, but stops counting when it reaches its maximum value. This limited 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.

Class `LimitedCounter` is an implementation of this type of counter.

Steps 1-5: Repeat all the steps in Exercise 1, this time using class `LimitedCounter` instead of `RollOverCounter`.

A Summary of What You've Observed So Far

- The `RollOverCounter` and `LimitedCounter` classes don't define any fields, yet the object inspector clearly shows that instances of these classes have three fields. A question we need to answer is, where are these fields defined?
- The `RollOverCounter` and `LimitedCounter` classes each define one method (`countUp`), yet we can call several other methods on instances of these classes. Another question we need to answer is, where are these methods defined?

Exercise 3 - Exploring Class `Counter`

Step 1: Looks at the headers for the `RollOverCounter` and `LimitedCounter` classes (the statements that start with the keywords `public class`). These headers declare that these two classes `extend` a class named `Counter`. In other words, `RollOverCounter` and `LimitedCounter` *inherit* field and method definitions from `Counter`. We say that `Counter` is the *superclass* of `RollOverCounter` and `LimitedCounter`, and that `RollOverCounter` and `LimitedCounter` are *subclasses* of `Counter`.

This inheritance relationship is represented in the class diagram in BlueJ's main window by solid arrows with triangular, hollow arrowheads. Each arrow depicts one superclass/subclass relationship. The arrowhead touches the superclass, and the arrow's tail touches the subclass.

Step 2: Open class `Counter` in an editor window and read the source code.

How many fields (instance variables) are defined in this class? What are their names? Compare these names to the field names that were displayed in the object inspectors for the `RollOverCounter` and `LimitedCounter` objects, in Exercises 1 and 2.

What are the names of the methods defined in `Counter`? Compare these names to the method names that were listed in the pop-up menus for the `RollOverCounter` and `LimitedCounter` objects, in Exercises 1 and 2.

Exercise 4 - Inheritance and Method Calls

Step 1: Rebuild the project (from the menu bar, select Tools > Rebuild Package).

Step 2: The *Code Pad* is the pane beside the object bench. If the Code Pad is not visible, from the menu bar select View > Show Code Pad. When the Code Pad is visible, there will be a check mark to the left of this command.

Step 3: We can type Java statements and expressions in the Code Pad. These are executed one at a time, as we type them.

Type the following statements in the Code Pad to create a new `RollOverCounter` object and call `countUp` on that object, twice.

```
RollOverCounter r = new RollOverCounter(1, 5);
r.countUp();
r.countUp();
```

Now type the following expression, which calls `getCount`. **Don't type a terminating semicolon after the method call.** This will cause the Code Pad to display the value returned by the method:

```
r.getCount()   // Don't type a terminating ';'
```

What value was returned by the method? Is this value correct?

Now type the following statement:

```
r.reset();
```

Once again, call `getCount` and note the value returned by the method. Is this value correct?

Step 4: Now we're going to use the debugger and the Code Pad together, to trace the execution of method calls. From the menu bar, select View > Show Debugger.

Step 5: Open class `Counter` in an editor window. Place a breakpoint in method `reset` at the statement:

```
count = minimumCount;
```

Step 6: Type the following statements in the Code Pad:

```
r.countUp();
r.countUp();
r.reset();
```

Look at the editor window displaying class `Counter` and notice that execution stopped at the breakpoint inside the `reset` method. You called `reset` on a `RollOverCounter` object, but that method isn't defined in class `RollOverCounter`, so Java called the `reset` method that `RollOverCounter` inherits from `Counter`.

Click Step until the `reset` method returns.

Step 7: Open class `Counter` in an editor window. Place a breakpoint in method `countUp` at the statement:

```
count++;
```

Open class `RollOverCounter` in an editor window. Place a breakpoint in method `countUp` at the statement:

```
    if (isAtMaximum()) {
```

Step 8: Type the following statement in the Code Pad:

```
    r.countUp();
```

Look at the editor window displaying class `RollOverCounter` and notice that execution stopped at the breakpoint inside the `countUp` method. Java is about to execute the highlighted statement:

```
    if (isAtMaximum()) {
```

The `if` statement contains the internal method call `isAtMaximum()`, but that method isn't defined in class `RollOverCounter`, so Java will call the `isAtMaximum` method that `RollOverCounter` inherits from `Counter`.

Step 9: Click Step Into (not Step).

Look at the editor window displaying class `Counter` and notice that Java is now executing the `isAtMaximum` method defined in that class.

Step 10: Click Step to cause control to return to `countUp` (in `RollOverCounter`).

Click Step once more. Notice that Java is about to execute the highlighted statement:

```
    super.countUp();
```

This statement will call the `countUp` method that `RollOverCounter` inherits from its superclass, `Counter`. We have to use the syntax `super.`*`methodName`*`()` because the calling method (`countUp` in `RollOverCounter`) and the called method (`countUp` in `Counter`) have the same names.

Click Step Into (not Step).

Look at the editor window displaying class `Counter` and notice that that Java is now executing the `countUp` method defined in that class.

Step 11: Repeatedly click Step until `countUp` in `RollOverCounter` has finished.

A Summary of What You've Learned

- When we call a method on an object, we never specify if the method is defined in the object's class or inherited from a superclass. The syntax for the call is simply *`objectName.methodName`*`()`. If the method is defined in the object's class, Java executes that method. If the method is not defined in the object's class, but is instead inherited from a superclass, Java executes the inherited method.

- A method in a subclass can use an internal method call to execute an inherited method, as long as the two methods *have different names*. For example, notice that the `countUp` method in `RollOverCounter` calls the `isAtMaxium` method inherited from `Counter`, this way:

```
isAtMaximum();
```

Class `RollOverCounter` doesn't define the `isAtMaximum` method, so Java executes the `isAtMaximum` method that is inherited from class `Counter`.

- A method in a subclass can call an inherited method *that has the same name*, by using the `super` keyword. For example, notice that the `countUp` method in `RollOverCounter` calls the `countUp` method inherited from `Counter`, this way:

```
super.countUp();
```

Exercise 5 - Adding a Method to `Counter`

We want `LimitedCounter` and `RollOverCounter` objects to provide a `countDown` method that decrements the counter by 1. For both types of counters, if `countDown` is called when the counter is at its minimum value, the object's state will not change; in other words, the count cannot be decremented below its minimum.

Instead of defining identical `countDown` methods in `LimitedCounter` and `RollOverCounter`, we should instead define the method in the `Counter` class. This method will be inherited by the counter subclasses.

Class `Counter` contains an incomplete definition of `countDown`. Read the method's Javadoc comment, then code the method.

Create `LimitedCounter` and `RollOverCounter` objects on the object bench. Open an object inspector for each counter. Verify that the count-up behaviour is unchanged from what you observed in Exercises 1 and 2, and that the counters now provide the required count-down behaviour.

Exercise 6 - Defining Another Counter Class

One of things biologists study is cell division, which is the process by which living organisms are constructed and repaired. Each time cell division occurs, the total number of cells doubles: 1 cell divides into 2 cells, then the 2 cells both divide, for a total of 4 cells, then the 4 cells each divide, for a total of 8 cells, and so on. When a cell dies, the total number of cells decreases by 1 (the death of one cell does not affect other cells).

A cell division counter mimics the cell division process. The counter has a minimum value of 1 and a maximum value of 2147483647; that is, $2^{31} - 1$, which is the largest value of Java's `int` type.

This counter always starts at 1. Counting up always doubles the count, unless doubling would cause the count to exceed the maximum value, in which case the count does not change. Counting down reduces the counter's value by 1, but when a counter is at its minimum value, requesting it to count down has no effect on the count.

The *counters* project contains an incomplete implementation of a class named `CellDivisionCounter`. Read the Javadoc comments and finish the implementation of the class.

- You are not permitted to make any changes to your `Counter` class from Exercise 5.
- Notice that two constants named `MIN_COUNT` and `MAX_COUNT` have been defined in the class. These represent the counter's minimum and maximum values. Your code should use these constants.
- You are not permitted to define any fields (instance variables) in `CellDivisionCounter`; however, you can define local variables in the constructor and methods.
- Your `CellDivisionCounter` class can override (redefine) any methods inherited from `Counter`, but you are not permitted to define any additional methods `CellDivisionCounter`.

Hint: this counter's maximum value is always 2^{31} - 1. When writing `countUp`, you can't use an expression like this:

```
    if (2 * currentCount > maximumCount) ...
```

to determine if doubling the current count would exceed the counter's maximum value. When `currentCount` is >= 2^{30}, multiplying that value by 2 yields a value that cannot be represented by a Java `int`, because it is >= 2^{31}. In such situations, the condition may yield `false` when you expect it to be `true`. When writing `countUp`, you have to figure out another way to determine if doubling the current count would exceed the counter's maximum value.

Here are some examples of how a `CellDivisionCounter` counts:

```
CellDivisionCounter counter = new CellDivisionCounter();
// The count is currently 1

counter.countUp();   // count doubles to 2
counter.countUp();   // count doubles to 4
counter.countUp();   // count doubles to 8
counter.countDown(); // count is now 7
counter.countUp();   // count doubles to 14
counter.reset();     // count is now 1

counter.countUp();   // count doubles to 2
counter.countDown(); // count is now 1
counter.countDown(); // count remains 1
```

```
CellDivisionCounter counter2 = new CellDivisionCounter();
// The count is currently 1

counter2.countUp(); // count doubles to 2 (i.e., 2^1)
counter2.countUp(); // count doubles to 4 (i.e., 2^2)
counter2.countUp(); // count doubles to 8 (i.e., 2^3)
counter2.countUp(); // count doubles to 16 (i.e., 2^4)
...
counter2.countUp(); // count doubles to 2^30
counter2.countUp(); // count remains 2^30, because 2 * 2^30
                    // is greater than the largest Java int, 2^31 - 1
```

Use the object bench or the Code Pad to test your method interactively.

Exercise 7

Class `CellDivisionCounterTest` is a JUnit test class containing a *suite* of *test cases* that test class `CellDivisionCounter`. Run all the test cases. A Test Results dialogue box will appear, listing the test cases that were executed. If your class is correct, there should be green check-marks to the left of all the test cases. An `x` to the left of a test case indicates that it failed.

If any of the test cases fail, you'll need to locate and fix the bugs. Use an object inspector and the debugger to help you determine where the problems are (e.g., before and after you execute a method, what values are stored in the object's fields? Which values are correct? Which values are incorrect? What section of code in the method you executed changes those values?) Edit `CellDivisionCounter` and rerun the JUnit test cases until every test passes.

A Summary Of What You've Learned

When we define a method in a superclass, that method is inherited by all subclasses, but this does not guarantee that the behaviour of the inherited method is appropriate for a particular subclass. A subclass can *override* an inherited method by redefining it; i.e., the method in the subclass has the same name and parameter list as the inherited method, but the method body is changed to provide the correct operation for the subclass.

Your `CellDivisionCounter` class overrides the `countUp` method it inherits from `Counter`.

Wrap-Up

1. Review your `CellDivisionCounter` class and make sure that the constructor and every method has a Javadoc comment. Make sure that the class has no fields (instance variables). Novice Java programmers sometimes define fields when they should instead define variables that are local to methods. If your class has too many fields, you'll need to edit it to fix this.

2. With one of the TAs watching, run the JUnit tests in `CellDivisionCounterTest`. The

TA will note how many test cases pass. The TA will review your solutions to the exercises, assign a grade (Satisfactory, Marginal or Unsatisfactory) and have you initial the demo/sign-out sheet.

3. The next thing you'll do is package the project in a *jar* (Java archive) file named counters.jar. To do this:

   3.1. From the menu bar, select Project > Create Jar File... A dialog box will appear. Click the Include source and Include BlueJ project files check boxes. A check-mark should appear in each box. Do not modify the Main class field.

   3.2. Click Continue. A dialog box will appear, asking you to specify the name for the jar file. Type counters or select the BlueJ icon named counters in the list of files. **Do not use any other name for your jar file** (e.g., lab5, my_project, etc.).

   3.3. Click Create. BlueJ will create a file named counters that has extension .jar. (Note: you don't type this extension when you specify the filename in Step 3.2; instead, it's automatically appended when the jar file is created.) The jar file will contain copies of the Java source code and several other files associated with the project. (The original files in your counters folder will not be removed).

4. Before you leave the lab, log in to cuLearn and submit counters.jar. To do this:

   4.1. Click the Submit Lab 5 link. A page containing instructions and your submission status will be displayed. After you've read the instructions, click the Add submission button. A page containing a File submissions box will appear. Drag counters.jar to the File submissions box. Do not submit another type of file (e.g., a .java file, a RAR file, a .txt file, etc.)

   4.2. After the icon for the file appears in the box, click the Save changes button. At this point, the submission status of your file is "Draft (not submitted)". If you're ready to finish submitting the file, jump to Step 4.4. If you instead want to replace or delete your "draft" file submission, follow the instructions in Step 4.3.

   4.3. You can replace or delete the file by clicking the Edit my submission button. The page containing the File submissions box will appear.

      4.3.1. To overwrite a file you previously submitted with a file having the same name, drag another copy of the file to the File submissions box, then click the Overwrite button when you are told the file exists ("There is already a file called..."). After the icon for the file reappears in the box, click the Save changes button.

      4.3.2. To delete a file you previously submitted, click its icon. A dialogue box will appear. Click the Delete button., then click the OK button when you are asked, "Are you sure you want to delete this file?" After the icon for the file disappears, click the Save changes button.

4.4. Once you're sure that you don't want to make any changes to the project you're submitting, click the Submit assignment button. A Submit assignment page will be displayed containing the message, "Are you sure you want to submit your work for grading? You will not be able to make any more changes." Click the Continue button to confirm that you are ready to submit your lab work. This will change the submission status to "Submitted for grading".