In today’s lab, we’ll extend the `Counter` class, deriving other classes from it to illustrate the mechanics of code reuse through inheritance.

**Setup**

1. Open Eclipse, create a new project called `inheritance`, and in that project create a new package called `tallying` for the classes we develop in this lab.

2. Recover the `Counter.java` class definition that you wrote for the “Objects and Classes” lab and place it in the `tallying` package. (If you don’t have a copy of that class definition, you can import one from the `/home/reseda/object-oriented-programming/code/` directory.)

**Adding a Memory**

One useful way of extending the `Counter` class is to give counters a memory, allowing them to store the tally at a particular time and subsequently recover the stored tally, even after additional click operations have overwritten the tally field itself.

3. Design this facility. What additional fields, if any, will a counter need in order to keep track of the stored tally? What additional public methods, if any, will the extended counters need? What should the “store the current tally” operation do if there is already a stored value in the memory when it is invoked again? What should the “recover the stored tally” operation do if no value has yet been stored? Should the `reset` method clear the stored tally as well as the current one, or should there be a separate method for clearing the stored tally? Resolve these questions and write down your answers to them. (What you write will be useful as comments when the new class is implemented.)

4. Write the definition of a new class, `CounterWithMemory`, that extends the `Counter` class and implements the memory facility that you designed in the previous exercise. Simultaneously, write a test program (`CounterWithMemoryTester`) that exercises all of the methods that you add to the `CounterWithMemory` class and ensures that they work correctly.

**An Abstract Method**

Another potentially useful method returns a `String` representation of the current tally (as opposed to the `report` method, which returns an `int`). But we can imagine that, for some applications (such as logging the counts over a long period of time), we might want the string representation to include the name of the counter and a time stamp that records the moment at which the value was inspected, while in other cases we just want the base-ten numeral corresponding to the current tally.

One way to arrange this is to add an abstract method to the `CounterWithMemory` class and then to derive two (or possibly more) classes from it, containing different implementations of the abstract method.

5. Add abstract method `show` that takes no arguments and returns a `String` to the definition of the `CounterWithMemory` class.

6. Define the derived `BasicCounterWithMemory` and `LoggingCounterWithMemory` classes, giving the former a `show` method that returns just the base-ten numeral for the current tally and the latter a `show` method that returns a string that also contains the counter’s name and a time stamp. (You can get the current date and time by importing the `java.util.Date` class and calling...
its zero-argument constructor. Objects of this class have a `toString` procedure that return time stamps accurate to the second.)