Interfaces, Abstract Classes, Object recursion

Purpose:

The purpose of this lab is to use interfaces and abstract classes, and to solve a problems with object recursion.

Set up:

Create a directory named *Lab29/counters* in your *Java* directory. Copy all the files from *~labCourse/Labs/Lab29/counters* to your *counters* directory.

Specifications:

We want to design and implement a class that models odometer. An odometer is a counter with a specified number of digits. There is no *a priori* limit on the number of digits an odometer can have. The count can be incremented or decremented by one. If all digits are 9, incrementing will cause all digits to become 0. If all digits are 0, decrementing will cause all digits to become 9. The commands an odometer must support are *increment*, *decrement*, and *reset*. A query *count* provides the current value.

Design:

Since there is no limit on the number of digits, we cannot use a simple integer counter. In fact, we cannot return the value of the odometer as an *int*. An appropriate choice for the odometer value might be a *java.math.BigInteger*. But we'll return the value of the odometer as a *String*, with high-order 0's suppressed.

We design the odometer as a sequence of digits, each with a value in the range 0 through 9. When the odometer is incremented, if the right-most digit is less than 9, it is incremented by 1. If that digit is 9, it is set to zero and process repeated for the next digit. For any given digit, the increment algorithm will be:

```
void increment () {
   if (value < 9)
      value = value + 1;
   else {
      value = 0;
      increment digit to the left
   }
}
```

Decrement is similar:

```
void decrement () {
   if (value > 0)
      value = value - 1;
   else {
      value = 9;
      decrement digit to the left
```

```
        }
    }
```

Two questions need to be addressed:

- how is a digit represented?
- what happens with the left-most digit?

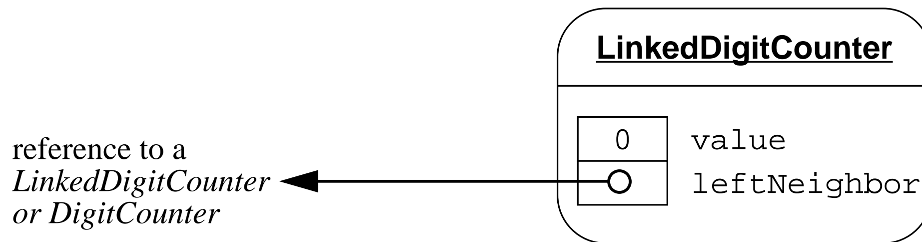
We can represent a digit as a counter that counts from zero to nine, and has the essentially the same operations as an odometer: commands *reset*, *increment*, and *decrement*, and a *String*-returning query *count*. We name the class representing digits *LinkedDigitCounter.*

The left-most digit behaves just as the others, except that it never references the "digit to the left." We need a class to model this kind of digit: that is, a digit with no left neighbor. We call it *DigitCounter.*

How do we put the digits together to form an odometer? The answer to this will simply be based on what a *LinkedDigitCounter* should know:

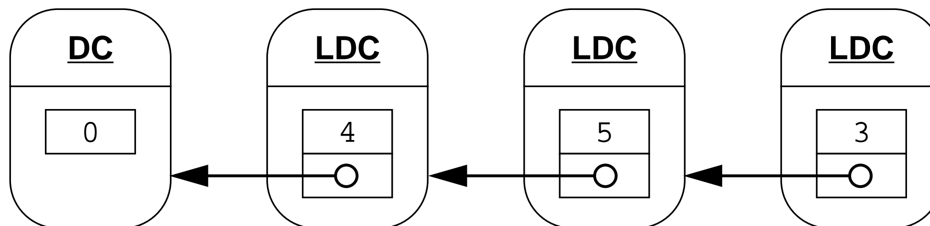
- it must know its current value;
- it must know its left neighbor digit.

A *LinkedDigitCounter* will look like this:



A *DigitCounter*, on the other hand, need only know its current value.

A four-digit odometer, for instance, will consist of three *LinkedDigitCounter* instances, and one *DigitCounter*:



The above grouping represents the value 0453.

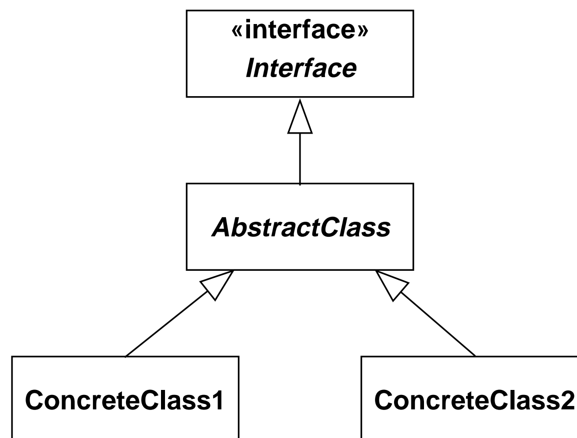
DigitCounter, *LinkedDigitCounter*, and *Odometer* have the same functionality. We can specify the functionality using an interface.

```
    /**
```

```
 * A basic up/down counter.
 */
public interface Counter {

    /**
     * The current value of this Counter as a String of digits.
     */
    String count();

    /**
     * Increment this Counter.
     */
    void increment();

    /**
     * Decrement this Counter.
     */
    void decrement();

    /**
     * Reset this Counter.
     */
    void reset();

}
```

Implementing the classes:

A common structural pattern is to define an abstract class that implements an interface, defines data common to all interface implementations, and provides some default method implementations. Concrete classes can then extend the abstract class:



We will do that here, even though our classes are not particularly complex. We define an abstract class *AbstractCounter* that inherits the abstract methods of the interface *Counter* and defines an *int* component variable to contain the value of the counter. The abstract class also defines a constructor that initializes the value to 0. The classes *DigitCounter* and *LinkedDigitCounter* will both extend this abstract class.

- Specify and implement this class *AbstractCounter*. Be aware that the data will be shared by its subclasses, and those subclasses should be able to modify it.
- Implement the classes *DigitCounter* and *LinkedDigitCounter*. The class *LinkedDigitCounter* adds a new attribute, *leftNeighbor*. The value of *leftNeighbor* might reference a *LinkedDigitCounter* or a *DigitCounter*. Its type should be *Counter*, and the *LinkedDigitCounter* constructor should require a *Counter* as argument.
- Would it be reasonable to give a *LinkedDigitCounter* an *Odometer* as left neighbor? Why or why not?
- Compile and test your implementation. A test driver, *OdometerTUIStart*, is provided for you.

Post-lab:

Submit the following, as directed by your lab instructor:

- listings of classes *AbstractCounter*, *DigitCounter*, and *LinkedDigitCounter*.
- Test plan and script of test execution.