

CS 94SI Assignment #2 – Object-Oriented Programming

Due: Before class on Wednesday, April 16, 2008

E-mail to: cs94si-spr0708-staff@lists.stanford.edu

Exercise – Sets as Objects

Last week we saw an implementation of sets that represented them as functions on Ints. This week, we will take a more object-oriented approach. Consider the following trait (included in the starter code, `Assignment2.scala`):

```
trait AbstractSet {
  def contains(n: Int): Boolean
  def add(n: Int): AbstractSet
  def remove(n: Int): AbstractSet

  def addAll(lst: Iterable[Int]): AbstractSet =
    lst.foldLeft(this)((s, elem) => s add elem)
  def removeAll(lst: Iterable[Int]): AbstractSet =
    lst.foldLeft(this)((s, elem) => s remove elem)
}
```

Traits are almost like Java interfaces. They can specify methods that concrete classes must implement in order to mix-in the trait. In this example, the `AbstractSet` trait has three methods that concrete set classes must implement: a method to determine membership (`contains`), a method to add a member (`add`), and a method to remove a member (`remove`).

Unlike Java interfaces, traits can also implement methods. These methods are included (for free) in any class that mixes in the trait. The `AbstractSet` trait has two such methods: a method to add an entire `List` of members (`addAll`) and a method to remove an entire `List` of members (`removeAll`). These methods are implemented in the trait itself, even though they rely on methods that have no implementation yet. (Don't worry if you don't understand the implementation of `addAll` and `removeAll`, just know that they work as advertised.)

- a) Write a concrete class `MutableSet` that extends the `AbstractSet` trait and implements the three methods that that trait requires. As its name suggests, your class should be *mutable*. That is, calling the `add` or `remove` methods on a `MutableSet` should modify the instance of the set on which the method was called, such that it now contains (or no longer contains), the specified `Int`.

Note that the `AbstractSet` trait specifies that the `add` and `remove` methods must return an `AbstractSet`. In the case of `MutableSet`, it is sufficient to return the current set (`this`). This has been implemented for you in the starter code. (Remember that using the `return` keyword is optional; any code block automatically returns its last expression.)

There are many ways to implement `MutableSet`. This assignment is purposefully open-ended. You may want to look into collections like `List[Int]` or `Array[Int]` in the standard library. (Granted, the standard library also includes `Set` collections. Don't "cheat" and implement your `MutableSet` with a `Set` from the standard library.) Regardless of your implementation choices, the starter code includes some "unit tests" that your `MutableSet` should pass.

- b) Write a concrete class `ImmutableSet` that extends the `AbstractSet` trait and implements the three methods that that trait requires. As its name suggests, your class should be *immutable*.

That is, calling the `add` or `remove` methods on an `ImmutableSet` should **not** modify the instance of the set on which the method was called. Instead, the method should return a new set which contains all the previous elements plus the added element (or minus the removed element). Hopefully this motivates the return type requirements of the `Set` trait.

Again, this assignment is purposefully open-ended; there are many ways to implement `ImmutableSet`. People coming from Java or similar languages often find it difficult to think in immutable data types. Be creative. **A word of warning, however:** it is entirely possible that we haven't covered enough of the language syntax for you to implement `ImmutableSet` easily. If you find yourself getting stuck, e-mail us what you've tried so far and we can help you out.

As with `MutableSet`, the starter code includes some "unit tests" that your `ImmutableSet` should pass, if implemented correctly.

- c) Since Scala allows method definitions in traits and sets no limit on how many traits can be mixed in, this provides for something akin to multiple inheritance. Consider the following trait:

```
trait LockingSet extends AbstractSet {
  abstract override def add(n: Int): AbstractSet =
    synchronized { super.add(n) }
  abstract override def remove(n: Int): AbstractSet =
    synchronized { super.remove(n) }
  abstract override def contains(n: Int): Boolean =
    synchronized { super.contains(n) }
}
```

The `LockingSet` trait can be mixed into any `AbstractSet` to create a set that can safely be used in the presence of multiple threads. The `synchronized` keyword works as you would expect from Java: each `AbstractSet` instance has a lock that must be acquired before the synchronized method can be performed. (Note: In reality, `synchronized` is not a keyword but a method on the base `AnyRef` type.) Given this definition, a thread-safe `MutableSet` can be instantiated as follows:

```
val s = new MutableSet with LockingSet
```

Now write a trait `LoggingSet` that extends the `AbstractSet` trait and provides logging functionality. Your trait should write to standard output (using the `println` function) each time the `contains`, `add` or `remove` methods are called.

Now a `MutableSet` that is both thread-safe and logs its usage to standard output can be instantiated as follows:

```
val s = new MutableSet with LoggingSet with LockingSet
```

Note that our definitions for `LockingSet` and `LoggingSet` are agnostic to the actual implementation of our set. These traits can just as easily and without modification be mixed into our `ImmutableSet` class.

- d) Do you expect there to be a difference between `MutableSet with LoggingSet with LockingSet` and `MutableSet with LockingSet with LoggingSet`? What might that difference be?
- e) Does it make sense to mix the `LockingSet` trait into an `ImmutableSet`? Why or why not?

Please hand in all the code you've written for this assignment.