

Assigned: Tuesday 15 February 2011

Due: Tuesday 22 February 2011

Topics: Parameterized types, abstract classes, inheritance, generics, collections

Collaboration: This homework assignment may be completed individually or in pairs.

Submission: Follow the instructions for submitting programs via P-Web and handing in a printed copy. Be sure to generate unit tests and a statement justifying their sufficiency.

Warning: This assignment involves subtlety with Java typing and generics. The programming is not difficult, but it will require careful thought about generics. Read the entire assignment before you begin. Start early and ask lots of questions!

In mathematics, a *magma* is a type of algebraic structure that has a set of elements, all of the same type, and one binary operation. Such a pairing of a set (e.g., positive integers) with this operation is a *semigroup* when the operation is associative and produces an element from the same set.

Another common algebraic structure features a unary operation that is its own inverse, commonly called the *complement*. One example is the boolean complement operator, which takes a boolean and produces a boolean. By definition, applying a complement operator twice gives the original value.

In this assignment, you will create a generic complementable interface and semigroup abstract class that support some general operations. You will then create fully concrete implementations of some familiar semigroups: positive integers paired with common addition, and RGB colors paired with a combination operation (e.g., averaging the color components of the two operands). You will also implement the complementable interface.

Complementable Interface

Define a typed interface `Complementable<AnyType>` that specifies the `complement` operation. Because this will be an object method, it takes no argument but produces an object of the (parameterized) complementable type. This is similar to the `java.lang.Iterator<T>` interface.

BinaryWord Class

Create a `BinaryWord` class implementing the `Complementable` interface. Use `java.util.BitSet` as the underlying storage container. While the `BinaryWord` constructor implementation is up to you, using a `String` argument may be the most straightforward.

The complement operation should yield a bitwise inversion of the word. For instance, the complement of 001011 would be 110100.

You should also implement `equals` and `toString` methods for later testing.

What should the parameterized type be for the `implements` clause? If you're confused, you might refer to the `java.lang.Comparable<T>` interface and also see the declaration of the `Integer` class in the Java API, as well as the `java.lang.Iterator<T>` interface and the declaration of the `java.util.Scanner` class.

Semigroup Class

Recall that a semigroup requires a binary operation, which means the operation takes two arguments.

Define an abstract, typed class `Semigroup<AnyType>` that specifies an `operator` operation, which takes one argument. (Note that the other implicit argument of the operation is the object receiving the method call.) Some aspects of this will be similar both to the `Comparator<T>` and `java.lang.Iterable<T>` interfaces.

`Semigroup` objects should be immutable; that is, any operations should return *new* objects of the generic type, rather than being mutators. Note that we do not yet have any shared implementations, so this class will not actually “do” much—yet. (That is, its functionally similar to an interface.)

PositiveInteger Class

Create a (concrete) subclass of `Semigroup` called `PositiveInteger`.

What should the parameterized type be for the `extends` clause? This is similar to `Comparable`, except `BinaryWord` implements a generic interface while `PositiveInteger` extends a generic class.

Implement the required method of the class (`operator`) using the typical add operation on integers. You should also implement `equals` and `toString` methods for later testing.

RGBColor Class

Create another (concrete) subclass of `Semigroup` called `RGBColor`. It should store three integers in the range 0-255. Because colors have complements, `RGBColor` should also implement the `Comparable` interface.

The `operator` of the `Semigroup` will be color blending. That is, the components of the new color should be the (integer) average of the components of the two input colors. The `complement` operation should give a new color whose components are each 255 minus the original.

For example, if [R/G/B] represents the three color components, the operator on [32/96/128] and [0/99/255] would yield [16/97/191]. The complement of the former would be [223/159/127].

Expanding Semigroup

An abstract class may as well be an interface if it has no members or concrete methods. Let’s rectify that.

Add one static method to your `Semigroup` class called `combine` that will compute accumulate the `operator` results sequentially for all the elements in a `Collection` (the parent interface of `ArrayList`) of any `Semigroup` objects. For example, employing the `combine` method on a collection of `PositiveIntegers` would yield their sum.

You should use Java’s enhanced `for` loop. We will take as a precondition that the collection is non-empty.

Note that every item in the given `Collection` must be the same *kind* of `Semigroup` object, otherwise you probably would not be able to combine them (e.g., what’s the operation on 5 and yellow??!). Thus, you will need to use type bounds on these static methods (c.f. Weiss 4.7.3 and 4.7.4). Note that `Collection` will need to be imported from `java.util`.

Testing

Demonstrate the functionality of your concrete classes and the additional static methods with a suite of appropriate unit tests that show the generic static methods working for both types of semigroups.

