Object Oriented Programming (Java & Scala) Assignment

Due Thursday, December 21

There are two parts to this assignment, a Java part and a Scala part.

Java

Briefly familiarize yourself with the `ArrayList<>` generic class and the `List<>` and `Comparable<>` generic interfaces in the Java API (such as found here). To use them, be sure to put

```
import java.util.*;
```

at the top of your Java code.

Put the following in a file named "Part1.java".

1. **Define a generic class `SortedList<>` that implements both the `List` and `Comparable` interfaces, such that two objects of type `SortedList<>` can be compared using the `compareTo` method. You can extend a built-in generic class, e.g. `ArrayList`, if you want, but that is up to you (it will make things easier).**

   The elements of `SortedList` must remain in sorted order. Therefore, the element type must also implement the `Comparable` interface.

   The `SortedList` class should implement an `add` method (e.g. by overriding the `add()` method of `ArrayList`), so that any new element added to a `SortedList` is added to the right place in the list in order to keep the `SortedList` in sorted order (instead of just being added at the end of the list).

   The comparison method, `compareTo()`, required by the `Comparable` interface, should take another `SortedList<T>` object (for the same `T`) and perform a lexicographic comparison. Based on a lexicographic comparison, a list L1 is less than a list L2 if one of the following is true:

   - **there is a k such that the first k-1 elements of L1 and L2 are equal and the kth element of L1 is less than the kth element of L2, or**
   - **L1 is of length k1 and L2 is of length k2, such that k1 < k2 and the first k1 elements of the two lists are equal.**

2. **For example, [1,2,3,4,5,8] < [1,2,3,4,6,7] and [2,4,6,8] < [2,4,6,8,10]. Note that we use a lexicographic comparison to order words alphabetically.**

   You may want to override the `toString()` method, if you don't like the way your `SortedList` objects print out.

3. **Define a class `A` that can be used to instantiate `SortedList<A>`, which also means that two `A`'s must be able to be compared to each other. You can define `A`any way you like, the only requirements are:**

   - **`A` includes a constructor, `A(Integer x) {...}`.**
   - **When comparing two `A` objects, the result of the comparison should be based on the `x` values that each object was initially constructed with. That is, given**

```
A a1 = new A(4);
A a2 = new A(5);
```

   the result of `a1.compareTo(a2)` should return -1, indicating that `a1` is less than `a2` (because 4 < 5).
   You'll also want to override the `toString()` method, so `A` objects print nicely.

4. **Define a class `B` that extends `A`. You may override the inherited `compareTo()` method if you want, but you don't have to. You can define `B` any way you like, the only requirements are:**

   - **`B` includes a constructor, `B(Integer x, Integer y) {...}`.**
   - **the `compareTo` method should work so that the value of `x+y` is used as the basis for comparison. For example, given**

```
A a1 = new A(6);
B b1 = new B(2,4);
B b2 = new B(5,8);
```

   the results of the comparisons should be:

```
a1.compareTo(b1);  //returns 0, since 6 = (2+4)
a1.compareTo(b2);  //returns -1, since 6 < (5+8)
b1.compareTo(a1);  //returns 0, since (2+4) = 6
b2.compareTo(a1);  //returns 1, since (5+8) > (6)
b1.compareTo(b2);  //returns -1, since (2+4) < (5+8)
```

   You'll want to override the `toString()` method, so `B` objects print nicely.

5. **In a separate class named `Part1`, define the static `main()` method. In that same class, define a static method `addtoSortedList()` that is polymorphic over any type `T` and which takes two parameters, `z` and `L`, where `z` is of type `T` and `L` can be any `SortedList` into which an object of type `T` can be inserted. `addtoSortedList()` should add `z` to `L` (which will automatically remain sorted, of course).**

6. **Finally, in class `Part1`, put the following method definition.**

```
7.     static void test() {
8.   SortedList<A> c1 = new SortedList<A>();
9.   SortedList<A> c2 = new SortedList<A>();
10.  for(int i = 35; i >= 0; i-=5) {
11.      addToSortedList(c1, new A(i));
12.      addToSortedList(c2, new B(i+2,i+3));
13.  }
14.
15.  System.out.print("c1: ");
16.  System.out.println(c1);
17.
18.  System.out.print("c2: ");
19.  System.out.println(c2);
20.
21.  switch (c1.compareTo(c2)) {
22.  case -1:
23.      System.out.println("c1 < c2");
24.      break;
25.  case 0:
26.      System.out.println("c1 = c2");
27.      break;
28.  case 1:
29.      System.out.println("c1 > c2");
30.      break;
31.  default:
32.      System.out.println("Uh Oh");
33.      break;
34.  }
35.
36.    }
```

Have `main()` call this `test()` method. The result should look something like:

```
c1: [[A<0> A<5> A<10> A<15> A<20> A<25> A<30> A<35>]]
c2: [[B<2,3> B<7,8> B<12,13> B<17,18> B<22,23> B<27,28> B<32,33> B<37,38>]]
```

```
c1 < c2
```

We'll be testing your code on other test functions, so try different versions of the above test code to see if your code works well.

Scala

In a file named "Part2.scala", put the following.

1. **Define a generic abstract class `Tree`, parameterized by a type `T`, such that:**
   - **A generic case class `Node`, also parameterized by type `T`, extends `Tree[T]` and represents an interior node that has a label of type `T`, a left subtree, and a right subtree (both of type `Tree[T]`).**
   - **A generic case class `Leaf`, parameterized by type `T`, extends `Tree[T]` and represents a leaf that has a label of type `T`.**
   - **`Tree[T]` is covariantly subtyped. That is, if `B` is a subtype of `A`, then `Tree[B]` is a subtype of `Tree[A]`.**

2. **Define a generic trait `Addable`, parameterized by a type `T`, that requires any class implementing the `Addable` trait to have a + method that takes a parameter of type `T` and returns a result of type `T`.**

3. **Define a class `A` that implements the `Addable` trait, such that**
   - **An `A` object is constructed using an integer parameter, e.g. new A(6). That integer should be stored within the `A` object.**
   - **The result of adding two `A` operands together is an `A` object constructed with the sum of the integers within the two operand objects. For example, new A(6) + new A(7) would create a new `A` object with the parameter 13.**
   - **The toString() method of `A` is overridden to show the integer value stored within it as well as to indicate that the object is of type `A`.**

4. **Define a class `B` that extends `A`, such that `B` also takes an integer parameter and overrides the toString() method to show the integer and to indicate that the object is a `B`.**

5. **Define a class `C` that extends `B`, such that `C` also takes an integer parameter and overrides the toString() method to show the integer and to indicate that the object is a `C`.**

6. **In a singleton class named Part2, put the following:**

    1. **A generic function, `inOrder`, that is parameterized by type `T` and computes the list of labels found in a tree, in in-order order. `inOrder` should take a `Tree[T]` as a parameter and return a `List[T]` as the result (where `List[]` is a generic class defined in the Scala API).**

    2. **A generic function `treeSum`, parameterized by type `T` such that any such `T` has to implement the `Addable` trait, which computes the sum of all the labels in a tree. It should take a `Tree[T]` as a parameter and and return a `T` as the result.**

    3. **A generic function `treeMap` (analogous to MAP in Scheme or ML) which applies a function to every label in a tree, returning a tree of the results. `treeMap`, for any types `T` and `V`, should take a function of type `T=>V` and a tree of type `Tree[T]` as parameters and return a tree of type `Tree[V]` as a result. The resulting tree should have the same structure (relationship of parent and child nodes) as the original tree.**

    4. **A function `BTreeMap` that takes a function of type `B=>B` and a tree of type `Tree[B]` and (just like for TreeMap, above) applies the function to every label in the tree, returning a tree of type `Tree[B]` as the result.**

    5. **A method `test()` containing the following code:**

```
6.   def test() {
7.     def faa(a:A):A = new A(a.value+10)
8.     def fab(a:A):B = new B(a.value+20)
9.     def fba(b:B):A = new A(b.value+30)
10.    def fbb(b:B):B = new B(b.value+40)
11.    def fbc(b:B):C = new C(b.value+50)
12.    def fcb(c:C):B = new B(c.value+60)
13.    def fcc(c:C):C = new C(c.value+70)
14.    def fac(a:A):C = new C(a.value+80)
15.    def fca(c:C):A = new A(c.value+90)
16.
17.    val myBTree: Tree[B] = Node(new B(4),Node(new B(2),Leaf(new B(1)),Le
   af(new B(3))),
18.                           Node(new B(6), Leaf(new B(5)), Leaf(new B(7))
   ))
19.
20.    val myATree: Tree[A] = myBTree
21.
```

```
22.    println("inOrder = " + inOrder(myATree))
23.    println("Sum = " + treeSum(myATree))
24.
25.    println(BTreeMap(faa,myBTree))
26.    println(BTreeMap(fab,myBTree))
27.    println(BTreeMap(fba,myBTree))
28.    println(BTreeMap(fbb,myBTree))
29.    println(BTreeMap(fbc,myBTree))
30.    println(BTreeMap(fcb,myBTree))
31.    println(BTreeMap(fcc,myBTree))
32.    println(BTreeMap(fac,myBTree))
33.    println(BTreeMap(fca,myBTree))
34.
35.    println(treeMap(faa,myATree))
36.    println(treeMap(fab,myATree))
37.    println(treeMap(fba,myATree))
38.    println(treeMap(fbb,myATree))
39.    println(treeMap(fbc,myATree))
40.    println(treeMap(fcb,myATree))
41.    println(treeMap(fcc,myATree))
42.    println(treeMap(fac,myATree))
43.    println(treeMap(fca,myATree))
44.  }
```

Note that some of the above lines will generate compile-time type errors. Comment out only those erroneous lines -- the comment should also indicate (in your own words) why there was a type error.

45. **A** `main()` **method that simply calls the** `test()` **method.**

The output of your program should look something like:

```
inOrder = List(B(1), B(2), B(3), B(4), B(5), B(6), B(7))

Sum = A(28)

Node(B(24),Node(B(22),Leaf(B(21)),Leaf(B(23))),Node(B(26),Leaf(B(25)),Leaf(B(27))))

Node(B(44),Node(B(42),Leaf(B(41)),Leaf(B(43))),Node(B(46),Leaf(B(45)),Leaf(B(47))))
```

```
Node(C(54),Node(C(52),Leaf(C(51)),Leaf(C(53))),Node(C(56),Leaf(C(55)),Leaf(C(57))))

Node(C(84),Node(C(82),Leaf(C(81)),Leaf(C(83))),Node(C(86),Leaf(C(85)),Leaf(C(87))))

Node(A(14),Node(A(12),Leaf(A(11)),Leaf(A(13))),Node(A(16),Leaf(A(15)),Leaf(A(17))))

Node(B(24),Node(B(22),Leaf(B(21)),Leaf(B(23))),Node(B(26),Leaf(B(25)),Leaf(B(27))))

Node(C(84),Node(C(82),Leaf(C(81)),Leaf(C(83))),Node(C(86),Leaf(C(85)),Leaf(C(87))))
```

Upload your two files, Part1.java and Part2.scala, to the course web site.