Assignments

Assignment - In progress

Add attachment(s), then choose the appropriate button at the bottom.

Title
OOP (Java and Scala) Programming Assignment
Due
Dec 16, 2019 11:55 PM
Number of resubmissions allowed
Unlimited
Accept Resubmission Until
Dec 16, 2019 11:55 PM
Status
Not Started
Grade Scale
Points (max 20.00)
Modified by instructor
Dec 5, 2019 6:41 PM

Instructions

Object Oriented Programming (Java & Scala) Assignment

Due Monday, December 16

There are two parts to this assignment, a Java part and a Scala part. Please complete the Java part as soon as possible so you can get started on the Scala part.

Part 1: Java

Familiarize yourself with the `List<>` and `Comparable<>` generic interfaces, as well as the `ArrayList<>` generic class in the Java API.

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

1. Define a generic class `ComparableList<>` that implements both the `List` and `Comparable` interfaces, such that two objects of type `ComparableList<T>` can be compared using the `compareTo` method. You can extend a built-in generic class, e.g. `ArrayList` that already implements the `List` interface, if you want, but that is up to you.

   The comparison method, `compareTo()`, required by the `Comparable` interface, should take another `ComparableList<T>` object (for the same `T`) and perform a lexicographic comparison. That is, a list *L1* is less than list *L2* if:

   - there is a *k* such that the first *k-1* elements of *L1* and *L2* are equal and the *k*th element of *L1* is less than *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.

   Two lists are equal if all of their corresponding elements are equal.

   Note that, in order for `ComparableList<T>` to be defined, you'll need to place constraints on the type parameter `T`.

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

2. Define a class `A` that can be used to instantiate `ComparableList<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 comparing the `x` values that the two objects were initially constructed with. That is, given

  ```
  A a1 = new A(6);
  A a2 = new A(7);
  ```

  the result of `a1.compareTo(a2)` should return `-1`, indicating that `a1` is less than `a2`.

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

3. Define a class `B` that extends `A` and overrides the inherited `compareTo()` method. 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 be overridden 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(3,5);
     ```

     the results of the comparisons should be:

     ```
     a1.compareTo(b1);  //returns 0, since 6 = (2+4)
     a1.compareTo(b2);  //returns -1, since 6 < (3+5)
     b1.compareTo(a1);  //returns 0, since (2+4) = 6
     b2.compareTo(a1);  //returns 1, since (3+5) > 6
     b1.compareTo(b2);  //returns -1, since (2+4) < (3+5)
     ```

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

4. In a separate class named `Part1`, define the static `main()` method. In that same class, define a polymorphic static method, `addToCList()`, which takes two parameters, *z* and *L*, where *L* can be any `ComparableList` and *z* can be inserted into *L*. The method `addToCList()` should insert *z* onto the end of *L*.

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

   ```
   static void test() {
       ComparableList<A> c1 = new ComparableList<A>();
       ComparableList<A> c2 = new ComparableList<A>();
       for(int i = 0; i < 10; i++) {
           addToCList(new A(i), c1);
           addToCList(new A(i), c2);
       }

       addToCList(new A(12), c1);
       addToCList(new B(6,6), c2);

       addToCList(new B(7,11), c1);
       addToCList(new A(13), c2);

       System.out.print("c1: ");
       System.out.println(c1);

       System.out.print("c2: ");
       System.out.println(c2);

       switch (c1.compareTo(c2)) {
   ```

```
        case -1:
            System.out.println("c1 < c2");
            break;
        case 0:
            System.out.println("c1 = c2");
            break;
        case 1:
            System.out.println("c1 > c2");
            break;
        default:
            System.out.println("Uh Oh");
            break;
        }

    }
```

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

```
c1: [[A<0> A<1> A<2> A<3> A<4> A<5> A<6> A<7> A<8> A<9> A<12> B<7,11> ]]
c2: [[A<0> A<1> A<2> A<3> A<4> A<5> A<6> A<7> A<8> A<9> B<6,6> A<13> ]]
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.

Part 2: Scala

Read the web page describing the Scala `Ordered` trait (click here).

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

1. Define a class, `OInt`, that implements the `Ordered` trait. `OInt` should define the `compare` method required by `Ordered` and should override the `toString` method so that it prints something sensible. Each object of the `OInt` class should be instantiated with an integer parameter, e.g. by saying `new OInt(6)`. The result of comparing two `OInt`'s should be based on the values of the integers they were created with. For example, the expression

   ```
   (new OInt(5)).compare(new OInt(6))
   ```

   should return -1, because 5 is less than 6.

2. Define an abstract generic class `OTree[T]` such that:

   - `T` must itself implement the `Ordered` trait, allowing two `T`'s to be compared.
   - `OTree[T]` also implements the `Ordered` trait, allowing two `OTree`'s to be compared
   - Two case classes, `OLeaf[T]` and `ONode[T]`, extend the `OTree[T]` class.
   - `OLeaf[T]` should be parameterized by a `T` object and `ONode[T]` should be parameterized by a list of `OTree`'s. For example, the following creates an `OTree[OInt]` object:

     ```
     val myTree = ONode(List(OLeaf(new OInt(3)),
                             OLeaf(new OInt(4))))
     ```

   Since the `OTree` class implements the `Ordered` trait, the `compare` method must be defined. The comparison between two `OTree`'s must satisfy the following rules.

   - Two `OTree`'s are equal if their structure is identical, i.e. they have the same arrangement of nodes and leaves, and the same values at the leaves.
   - An `OTree` *A* is less than an `OTree` *B*, when

- *A* is a leaf and *B* is a node, or
- *A* and *B* are both leaves, and the value at *A* is less than the value at *B*, or
- *A* and *B* are both nodes, and the list of children of *A* is lexicographically less than the list of children of *B* (where the lexicographic comparison is defined above).

3. In a singleton class named `Part2`, put the `main()` method. Also in the `Part2` singleton class, put the following.

- A method `compareTrees()` that takes two trees of type `OTree[T]` as parameters, for the same `T`. If the first tree is less than the second, `compareTrees()` should print "Less". If the two trees are equal, then `compareTrees()` should print "Equal". Otherwise, `compareTrees()` should print "Greater".
- The following `test()` method:

```
def test() {

  val tree1 = ONode(List(OLeaf(new OInt(6))))

  val tree2 = ONode(List(OLeaf(new OInt(3)),
                         OLeaf(new OInt(4)),
                         ONode(List(OLeaf(new OInt(5)))),
                         ONode(List(OLeaf(new OInt(6)),
                                    OLeaf(new OInt(7))))));

  val treeTree1: OTree[OTree[OInt]] =
    ONode(List(OLeaf(OLeaf(new OInt(1)))))

  val treeTree2: OTree[OTree[OInt]] =
    ONode(List(OLeaf(OLeaf(new OInt(1))),
               OLeaf(ONode(List(OLeaf(new OInt(2)),
                                OLeaf(new OInt(2)))))))


  print("tree1: ")
  println(tree1)
  print("tree2: ")
  println(tree2)
  print("treeTree1: ")
  println(treeTree1)
  print("treeTree2: ")
  println(treeTree2)
  print("Comparing tree1 and tree2: ")
  compareTrees(tree1, tree2)
  print("Comparing tree2 and tree2: ")
  compareTrees(tree2, tree2)
  print("Comparing tree2 and tree1: ")
  compareTrees(tree2, tree1)
  print("Comparing treeTree1 and treeTree2: ")
  compareTrees(treeTree1, treeTree2)
  print("Comparing treeTree2 and treeTree2: ")
  compareTrees(treeTree2, treeTree2)
  print("Comparing treeTree2 and treeTree1: ")
  compareTrees(treeTree2, treeTree1)

}
```

The `main()` procedure should call `test()` and the resulting output should look something like the following.

```
tree1: ONode(List(OLeaf(<6>)))
tree2: ONode(List(OLeaf(<3>), OLeaf(<4>), ONode(List(OLeaf(<5>))), ONode(List(OLeaf(<6>), OLeaf(<7>)))))
treeTree1: ONode(List(OLeaf(OLeaf(<1>))))
treeTree2: ONode(List(OLeaf(OLeaf(<1>)), OLeaf(ONode(List(OLeaf(<2>), OLeaf(<2>))))))
Comparing tree1 and tree2: Greater
Comparing tree2 and tree2: Equal
Comparing tree2 and tree1: Less
Comparing treeTree1 and treeTree2: Less
Comparing treeTree2 and treeTree2: Equal
Comparing treeTree2 and treeTree1: Greater
```

Submission

Attachments

No attachments yet

Select a file from computer Choose File No file chosen

Submit Preview Save Draft Cancel

> Don't forget to save or submit!

Change Profile Picture

Error removing image
Error uploading image
Upload Choose File No file chosen

Save Cancel Connections ✖ Remove

Search for people ...
View More

My Connections | Pending Connections

You don't have any connnections yet. Search for people above to get started.
You have no pending connections.
←Back to My Connections
Search for people ...
${{cmLoader.getString("connection_manager_no_results")}

Done