**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].

1. 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 Tand returns a result of type T.

1. 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.

1. 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 aB.

1. 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 aC.

1. In a singleton class named Part2, put the following:
   * 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](http://www.scala-lang.org/api/current/index.html#scala.collection.immutable.List)).
   * 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.
   * 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.
   * 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.
   * A method test() containing the following code:
   * def test() {
   * def faa(a:A):A = new A(a.value+10)
   * def fab(a:A):B = new B(a.value+20)
   * def fba(b:B):A = new A(b.value+30)
   * def fbb(b:B):B = new B(b.value+40)
   * def fbc(b:B):C = new C(b.value+50)
   * def fcb(c:C):B = new B(c.value+60)
   * def fcc(c:C):C = new C(c.value+70)
   * def fac(a:A):C = new C(a.value+80)
   * def fca(c:C):A = new A(c.value+90)
   * val myBTree: Tree[B] = Node(new B(4),Node(new B(2),Leaf(new B(1)),Leaf(new B(3))),
   * Node(new B(6), Leaf(new B(5)), Leaf(new B(7))))
   * val myATree: Tree[A] = myBTree
   * println("inOrder = " + inOrder(myATree))
   * println("Sum = " + treeSum(myATree))
   * println(BTreeMap(faa,myBTree))
   * println(BTreeMap(fab,myBTree))
   * println(BTreeMap(fba,myBTree))
   * println(BTreeMap(fbb,myBTree))
   * println(BTreeMap(fbc,myBTree))
   * println(BTreeMap(fcb,myBTree))
   * println(BTreeMap(fcc,myBTree))
   * println(BTreeMap(fac,myBTree))
   * println(BTreeMap(fca,myBTree))
   * println(treeMap(faa,myATree))
   * println(treeMap(fab,myATree))
   * println(treeMap(fba,myATree))
   * println(treeMap(fbb,myATree))
   * println(treeMap(fbc,myATree))
   * println(treeMap(fcb,myATree))
   * println(treeMap(fcc,myATree))
   * println(treeMap(fac,myATree))
   * println(treeMap(fca,myATree))
   * }

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.

* + 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))))