Exercise Sheet 7

1. Extend the following program with the aliasing modes `rep` and `arg`:

```
class Human {}

class Room {
    private Human m;

    void enter( Human m ) { /*...*/ }

    Human leave() { /*...*/ }
}

public class Building {
    private Room[] rooms;

    void enter( Human m ) { rooms[0].enter( m ); }

    Human leave() { rooms[0].leave( m ); }
}
```

2. Are these aliasing modes correctly used in the following program? Which expressions are legal? Which are invalid?

```
public class Assignments {

   /* rep */ Integer attr_x;
   /* arg */ Integer attr_y;
             Integer attr_z;


   public void m( /* arg */ Integer a ) {
      /* rep */ Integer x = new /* rep */ Integer(4);
      /* arg */ Integer y = new Integer(3);
      Integer z = new Integer(2);

      x = y;
      y = x;

      x = z;
      y = z;

      attr_x = x;
      attr_y = a;
   }
}
```

3. Read the article „*Universes: Lightweight Ownership for JML*“ written by Werner Dietl and Peter Müller.
   The article is available online at:

   `http://www.jot.fm/issues/issue_2005_10/article1.pdf`

   The concepts in the article will be explained in the next lecture and will be used in the next exercise.

4. **Aliasing question from last year's exam.** The following source code is used.
   Given is the following implementation for a sorted tree, which stores pairs (Key,Value). The Keys are of type **int**, the values are of type **Object**.

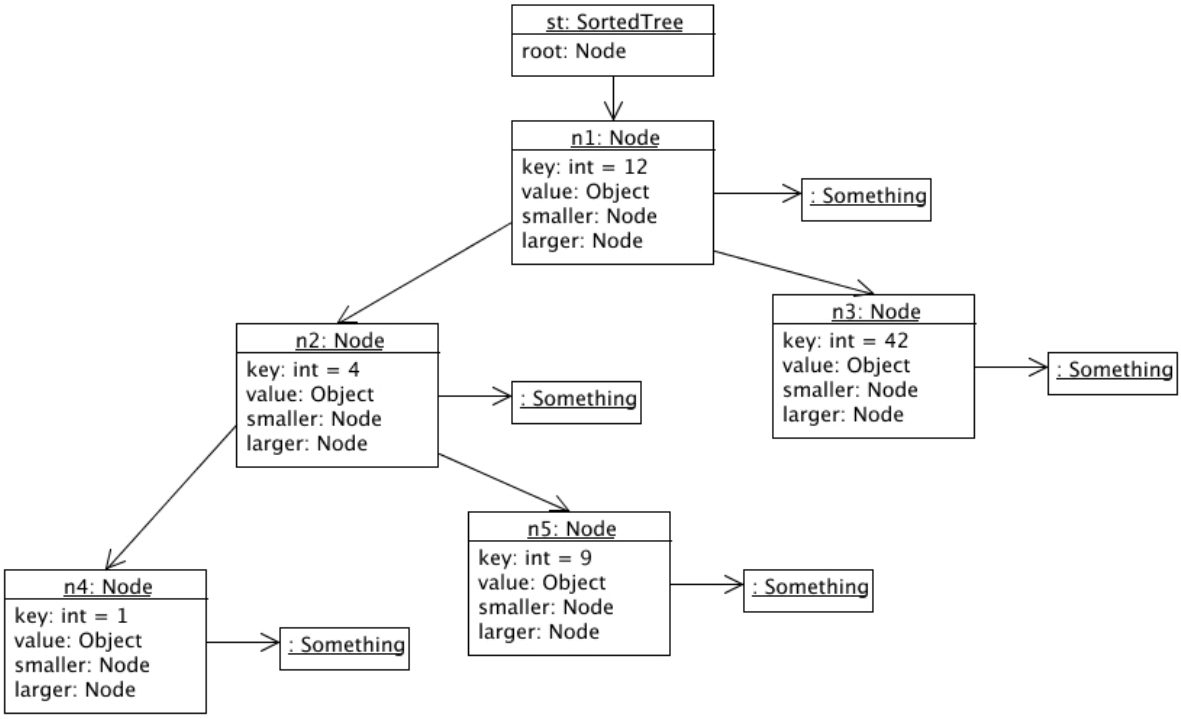
```
public class SortedTree {
  // the root of the sorted tree
  private Node root;

  // insert a new element at the correct location in the tree
  public void insert(int ke, Object val) {
       if (root == null) { root = new Node(ke, val); }
       else { root.insert(ke, val); }
  }

  // merge the given tree with the current tree
  public void merge(SortedTree toinsert) {
       if (root == null) { root = toinsert.root; }
       else { root.merge(toinsert.root); }
  }
}
```

The helper class **Node** looks like this:

```
// this class is a helper class and uses default
// visibility everywhere
class Node {
     // subtree with smaller elements
     Node smaller;
     // subtree with larger elements
     Node larger;
     // the key of this Node
     int key;
     // the value of this Node
     Object value;

     Node(int ke, Object val) {
          this.key = ke; this.value = val;
     }

     // insert the new key-value pair at the correct position
     void insert(int ke, Object val) {
```

```
            if (ke <= this.key) {
                if (smaller == null) {
                    smaller = new Node(ke, val); }
                else { smaller.insert(ke, val); }
            } else {
                if (larger == null) {
                    larger = new Node(ke, val); }
                else { larger.insert(ke, val); }
            }
        }

        // insert the given subtree at the correct position
        void merge(Node toinsert) {
            if (toinsert.key <= this.key) {
                if (smaller == null) { smaller = toinsert; }
                else { smaller.merge(toinsert); }
            } else {
                if (larger == null) { larger = toinsert; }
                else { larger.merge(toinsert); }
            }
        }
}
```

The following diagram shows the valid object structure:



a) Use an example to show that a **SortedTree** object does not completely encapsulate its **Node** structure.

b) The programmer wants to enhance the encapsulation of the **Node** structure using the Universe Type System. There are two reasonable typings for the **Node** structure. Write down the field declarations of both classes for both possibilities. For each case, draw the corresponding Ownership Diagram for the example.