— Solution notes —

COMPUTER SCIENCE TRIPOS Part IA – 2014 – Paper 1

4 Object-Oriented Programming (RKH)

A Lecturer wishes to create a program that lists his students sorted by the number of practical assignments they have completed. The listing should be greatest number of assignments first, sub-sorted by name in lexicographical order (A to Z).

A class `StudentInfo` stores the name and number of assignments completed for a student. Amongst other methods, it contains a `void setCompleted(int n)` method that allows changes to the number of completed assignments.

Core OOP

(*a*) Provide a definition of `StudentInfo` with an `equals()` method and a natural ordering that matches the given requirement. [9 marks]

Answer:

- Define a sensible encapsulated class with appropriate fields and access modifiers (private state with public getters and setters etc) **(2)**;
- override `equals(...)` (note: failure to recollect it has an `Object` argument would not lose marks, nor did forgetting to create a `hashcode()` method.) **(1)**;
- compare the fields correctly in `equals(...)`**(1)**;
- implement `Comparable<StudentInfo>` (forgetting the to use Generics was not acceptable) **(1)**;
- create a `int compareTo(StudentInfo s)` method **(1)**;
- implement a logic that compares nCompleted and then compares String iff they are equal **(1)**;
- create a reverse sorting by assignments (e.g. `(s.nCompleted-nCompleted);`) **(1)**.
- use `name.compareTo(s.name)` when comparing `String`s **(1)** .

```
public class StudentInfo implements Comparable<StudentInfo>{
  private String name; // student name
  private int nCompleted=0; // Number of assignments completed
  public StudentInfo(String n) { name=n; }
  public int getCompleted() { return nCompleted; }
  public void setCompleted(int n) { nCompleted=n; }
  public String getName() { return name; }

  @Override
  public int compareTo(StudentInfo s) {
    int c=(s.nCompleted-nCompleted);
    if (c==0) return name.compareTo(s.name);
    return c;
  }

  @override
  public boolean equals(Object o) {
    StudentInfo s = (Studentinfo)o;
    return (nCompleted==s.nCompleted) && (name.equals(s.name));
  }
}
```

(*b*) A `TreeSet` is used to maintain the `StudentInfo` objects in appropriate order. When `setCompleted(...)` is called on a `StudentInfo` object it is necessary to remove the object from the set, change the value and then reinsert it to ensure the correct ordering. This is to be automated by applying the Observer design pattern via classes `UpdatableTreeSet` and `SubscribableStudentInfo`. A partial definition of `UpdatableTreeSet` is provided below.

```
public class UpdatableTreeSet extends
             TreeSet<SubscribableStudentInfo> {
  // To be called just before the StudentInfo object is updated
  public void beforeUpdate(SubscribableStudentInfo s) {
    remove(s);
  }
  // To be called just after the StudentInfo object is updated
  public void afterUpdate(SubscribableStudentInfo s) {
    add(s);
  }
}
```

Observer

(*i*) Extend `StudentInfo` to create `SubscribableStudentInfo` such that: multiple `UpdatableTreeSet` objects can subscribe and unsubscribe to receive updates from it; and the `beforeUpdate(...)` and `afterUpdate(...)` methods are called appropriately on the subscribed `UpdatableTreeSet` objects whenever `setCompleted(...)` is called. [6 marks]

Answer: This requires an appropriate structure to hold the updatables **(1)**; a subscribe method to add to the structure **(1)**; an unsubscribe method to remove from the structure **(1)**; two loops **(1)** in `setCompleted`, each calling the correct update method **(1)** in the correct place **(1)** . E.g.

```
public class SubscribableStudentInfo extends StudentInfo {

  private Set<UpdatableTreeSet> mRegistered =
     new HashSet<UpdatableTreeSet>();

  public void subscribe(UpdatableTreeSet u) {
    mRegistered.add(u);
  }

  public void unsubscribe(UpdatableTreeSet u) {
    mRegistered.remove(u);
  }

  @Override
  public void setCompleted(int n) {
    Set<UpdatableTreeSet> tmp =
       new HashSet<UpdatableTreeSet>(mRegistered);
    for (UpdatableTreeSet u : tmp) {
      u.beforeUpdate(this);
    }
```

```
        nCompleted=n;
       for (UpdatableTreeSet u : tmp) {
         u.afterUpdate(this);
       }
      }
    }
```

In 2014, some candidates favoured the used of a lists to store the registered objects. Whilst this was acceptable, in later parts this may have required a check for duplicates.

Note that the correct implementation of `setCompleted()` affected the answer expected to the following part. Specifically, the candidate could choose not to copy `mRegistered` and adapt their solution to the next part so that the mRegistered structure was not touched by the calls to `beforeUpdate` and `afterUpdate`. Either approach attracted full marks here, but placed restrictions on the following part (see below).

Observer

(*ii*) Give a complete definition of `UpdatableTreeSet` that overrides the inherited methods `boolean add(SubscribableStudentInfo)` and `boolean remove(Object)` to automatically subscribe and unsubscribe to their arguments, as appropriate. You may ignore all other methods inherited from `TreeSet`. [5 marks]

Answer: If the partial code is used as provided, it is necessary to have copied the collection in the loops of the previous part (otherwise `beforeUpdate` will attempt to deregister and produce a ConcurrentModificationException in the loop, etc). This approach requires **(1 mark each)**:

- calling super.add in the add method and super.remove() in the remove method;
- Correctly casing the input to remove();
- (De)-registration in the add/remove methods;
- spotting the concurrency issue in the loops;
- solving the concurrency issue.

```
public class UpdatableTreeSet extends TreeSet<SubscribableStudentInfo> {

  @Override
  public boolean add(SubscribableStudentInfo s) {
    s.subscribe(this);
    return super.add(s);
  }

  @Override
  public boolean remove(Object s) {
    SubscribableStudentInfo si = (SubscribableStudentInfo) s;
    si.unsubscribe(this);
    return super.remove(s);
  }

  public void beforeUpdate(SubscribableStudentInfo s) {
    remove(s);
  }
```

```
  public void afterUpdate(SubscribableStudentInfo s) {
    add(s);
  }
}
```

If the answer to the previous part used `mRegistered` directly (no copying), a valid strategy was to explicitly call the parent methods (which had no notion of registration incorporated):

```
public class UpdatableTreeSet extends TreeSet<StudentInfo> {

  // ... Other methods as above

  public void beforeUpdate(StudentInfo s) {
    super.remove(s);
  }

  public void afterUpdate(StudentInfo s) {
    super.add(s);
  }
}
```
