TDA550 – Objektorienterad programvaruutveckling IT, forts. kurs
Övning vecka 5

Pelle Evensen & Daniel Wetterbro

20 oktober 2011

Sammanfattning

Denna vecka ska vi titta på samlingar, generics och designmönstren Decorator, Singleton och Iterator.
Övningarna är graderade (något subjektivt) från lätt (*) till svår (***). Svårighetsgraden hos en övning har inte nödvändigtvis med lösningens storlek att göra.

1 Samlingar

1.1 Frekvenstabell **

Kom ihåg att samlingar och avbildningar (eng. "map") endast kan lagra objektreferenser och inga primitiva värden. Om du försöker lagra primitiva värden, kommer Javakompilatorn automatiskt göra omslagsobjekt (wrapper objects) av motsvarande typer, d.v.s. automatisk typomvandling. (Om du inte är säker på vad omslagsobjekt är för något, diskutera detta med varandra och/eller handledaren).[1]

1. Skriv ett program som genererar en frekvenstabell av orden i argumentlistan till programmet. Frekvenstabellen mappar varje ord till antalet gånger det förekommer i argumentlistan. För att förstå vad som egentligen händer, så får ni för närvarande inte använda er av automatisk typomvandling (boxing, unboxing), utan ni måste göra detta explicit. Använd i stället till att börja med följande klassen **IncrementableInteger** i fig. 1, sida 2, som "värden" i avbildningen. Histogrammet bör alltså ha den *statiska* typen `Map<String, IncrementableInteger>`.

2. Skriv nu samma program utan att använda av **IncrementableInteger**. Använd automatisk typomvandling så ofta ni kan, d.v.s. programmet ser ut att spara och hämta primitiva värden (förutom deklarationen av avbildningen, som nu får statisk typ `Map<String, Integer>`). Kan ni se några nackdelar med denna version?

[1] Lånad av Bror Bjerner, Övning 4, 2008

```
class IncrementableInteger {
    private int intField;

    public IncrementableInteger(int i) {
        intField = i;
    }

    public void increment() {
        intField = intField + 1;
    }

    public int getIntField() {
        return intField;
    }

    public String toString() {
        return String.valueOf(intField);
    }
}
```

Figur 1: Klassen **IncrementableInteger**.

1.2 Iteratorrättning **

En något meningslös lista över slumpade datum skapas av klassen **DummyDates** i fig. 2. Den tillhandahåller även en iterator för att stega sig igenom de genererade datumen.[2] Men iteratorn har brister, identifiera dessa! Till din hjälp följer ett utdrag ur Suns Java 6 API:

boolean hasNext()

Returns true if the iteration has more elements. (In other words, returns true if **next** would return an element rather than throwing an exception.)

E next()

Returns the next element in the iteration. Throws **NoSuchElementException** if the iteration has no more elements.

void remove()

Removes from the underlying collection the last element returned by the iterator (optional operation). This method can be called only once per call to **next**. The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method. Throws **UnsupportedOperationException** if the remove operation is not supported by this Iterator. Throws **IllegalStateException** if the **next** method has not yet been called, or the **remove** method has already been called after the last call to the **next** method.

[2] Baserad på uppgift av Joachim von Hacht, Övning 4, 2009.

```
public class DummyDates implements Iterable<Date> {

    private final static int SIZE = 15;
    private Date[] dates = new Date[SIZE];
    private int actualSize;

    public DummyDates() {
        Random r = new Random();
        // Fill the array....
        for (int i = 0; i < SIZE; i++) {
            dates[i] = new Date(r.nextInt());
        }
        actualSize = SIZE;
    }

    public Iterator<Date> iterator() {
        return new DSIterator();
    }

    private class DSIterator implements Iterator<Date> {

        // Start stepping through the array from the beginning
        private int next = 0;

        public boolean hasNext() {
            // Check if a current element is the last in the array
            return (next < actualSize);
        }

        public Date next() {
            return dates[next++];
        }

        public void remove() {
            for (int i = next; i < SIZE - 1; i++) {
                dates[i] = dates[i + 1];
            }
            dates[SIZE - 1] = null;
            actualSize--;
        }
    }
}
```

Figur 2: Klassen **DummyDates**.

1.3 En exklusiv union *

Implementera metoden

public static <T> Set<T> exclusiveUnion(Set<T> s1, Set<T> s2) ...

som ger den mängd av element som finns antingen i s1 eller s2, men som inte finns i båda. Metoden får inte vara destruktiv, d.v.s. innehållet i s1 och s2 får inte förändras. I metodkroppen använder ni bara mängdoperationer och konstruktorer för mängder.[3]

[3]Lånad av Bror Bjerner, Övning 4, 2008

1.4 Under huven ***

I fig. 3 ses ett enkelt testprogram som lagrar två miljoner slumpmässigt genererade Integers i en samling och därefter söker efter två miljoner ytterligare slumpmässigt framtagna i samlingen.

Tre olika implementationer av samlingen har använts: en hashtabell, en trädstruktur och en länkad lista och deras körningstider (i sekunder) uppmätts. För varje datastruktur har vi mätt tiden för `add` respektive `contains` (100k tal och 5M tal). Resultaten kan avläsas nedan; avgör när de olika datastrukturerna använts och förklara varför resultaten ser ut som de gör. Vardera av *x, y, z* motsvarar varsin datastruktur. Är någon konsekvent bättre med avseende på tidsåtgång?

Implementation	100k `add`	100k `contains`	5M `add`	5M `contains`
x	0.04	51.00	0.72	> 100 min
y	0.13	0.10	8.30	4.80
z	0.06	0.04	2.89	0.96

```
public class Timings2 {

    public static void main(String[] args) {
        // Use different implementations of a collection
        Collection<Integer>[] colls = new Collection[3];
        colls[0] = new TreeSet<Integer>();
        colls[1] = new HashSet<Integer>();
        colls[2] = new LinkedList<Integer>();

        final int NUMRUNS = Integer.parseInt(args[0]);
        final int SEED = Integer.parseInt(args[1]);

        for (Collection<Integer> c : colls) {
            Random randomizer = new Random(SEED);
            // Add a lot...
            long now = System.nanoTime();
            for (int i = 0; i < NUMRUNS; i++) {
                c.add(randomizer.nextInt());
            }
            System.out.println("add " + c.getClass() + "\t"
                    + ((System.nanoTime() - now) / 1000000000.0));

            // ...and search a lot
            now = System.nanoTime();
            for (int i = 0; i < NUMRUNS; i++) {
                c.contains(randomizer.nextInt());
            }
            System.out.println("contains " + c.getClass() + "\t"
                    + ((System.nanoTime() - now) / 1000000000.0));
            c.clear();
        }
    }
}
```

Figur 3: Klassen **Timings2**.

2 Generics

2.1 Typsäkerhet **

En cykel är ett objekt, likaså är en racingbil en bil är ett objekt. I klassen **Vehicles** nedan (fig. 4) har programmeraren begått ett fel som gör att programmet krashar vid körning.

1. Varför krashar programmet?
2. Skriv om klassen **Vehicles** så att felet upptäcks vid kompilering istället för vid körning. Se gärna Item 23 i [Blo08].

```
class Bicycle {
    public String toString() {    return "Two wheeled bike"; }
}

class Car {
    public String toString() {    return "An ordinary car"; }
    public double getCO2EmissionLevel() { return 0.1; }
}

class RaceCar extends Car {
    public String toString() {    return "Ferrarati 7.4"; }
    public double getCO2EmissionLevel() { return 5.1; }
}

public class Vehicles {
    public static void main(String[] args) {
        Collection vehicles = new HashSet();
        vehicles.add(new Bicycle());
        vehicles.add(new Car());
        vehicles.add(new RaceCar());
        for(Object obj : vehicles) {
            Car c = (Car) obj;
            System.out.println(c.toString() + " emits " + c.getCO2EmissionLevel()
                    + " g CO2 / km");
        }
    }
}
```

Figur 4: Klassen **Vehicles**.

3 Singleton-mönstret **

Syftet med Singleton-mönstret är att vi vill kunna garantera att endast en instans av en klass existerar. Ofta (men inte alltid) bör man också kunna komma åt denna instans utifrån. I fig. 5 visas en klass med enbart klassmetoder. Är detta en singleton? Om inte, hur ska vi förändra den så att den blir det? Spelar det någon roll om den är det eller inte?

```
class GameTile {
    // ...
}

public class GameUtils {
    public static GameTile[][] newBoard(final int width, final int height,
            final GameTile baseTile) {
        GameTile[][] board = new GameTile[width][height];
        fillBoard(board, baseTile);

        return board;
    }

    public static void fillBoard(final GameTile[][] board,
            final GameTile baseTile) {
        for (GameTile[] row : board) {
            for (int j = 0; j < row.length; j++) {
                row[j] = baseTile;
            }
        }
    }
}
```

Figur 5: Klassen **GameUtils**.

4 En udda dekoratör och iterator **

Studera klassen **ForwardInputIterator** i fig. 6. Som vi ser *tar den bort* funktionalitet snarare än lägger till. Beskriv syftet med klassen. Om klassen fyller någon sund funktion, hade vi kunnat få samma funktionalitet om vi tänker i termer av Iterator-mönstret istället för hur Javas **Iterator**-gränssnitt ser ut?

Formulera lämpliga eftervillkor för de två metoderna `processIterator(Iterator<T>)` och `processIterator(ForwardInputIterator<T>)`.

Vad kan gå fel vid det sista anropet till `processIterator`?

Referenser

[Blo08] Joshua Bloch. *Effective Java.* Addison-Wesley, 2nd edition, 2008.

```
public class ForwardInputIterator<T> implements Iterator<T> {
    private final Iterator<T> it;

    public ForwardInputIterator(final Iterator<T> it) {
        this.it = it;
    }

    @Override
    public boolean hasNext() {
        return this.it.hasNext();
    }

    @Override
    public T next() {
        return this.it.next();
    }

    @Override
    public void remove() {
        throw new UnsupportedOperationException("remove() not supported.");
    }

    /**
     * @post What postcondition can this method guarantee with regards
     *   to the structure that it iterates on?
     */
    public static <T> void processIterator(Iterator<T> it) {
        // ... do something interesting here.
    }

    /**
     * @post What postcondition can this method guarantee with regards
     *   to the structure that it iterates on?
     */
    public static <T> void processIterator(ForwardInputIterator<T> it) {
        // ... do something interesting here.
    }

    public static void main(String[] args) {
        Collection<Object> coll = new LinkedList<Object>();
        // ... add a lot of elements to coll.

        // How is this different ...
        processIterator(coll.iterator());

        // From this ...
        processIterator(new ForwardInputIterator<Object>(coll.iterator()));

        // and this ...
        processIterator((Iterator<Object>) new ForwardInputIterator<Object>(coll.iterator()));
}
}
```

Figur 6: Klassen **ForwardInputIterator**.