

Övning vecka 5.

Denna vecka ska vi titta på samlingar, generics och designmönstren Decorator, Singleton och Iterator.

Uppgift 1

Kom ihåg att samlingar och avbildningar (eng. "map") endast kan lagra objektreferenser. Om du försöker lagra värden av en primär datatyp, kommer Javakompilatorn att 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).

- a) Skriv ett program som genererar en frekvenstabell av orden i argumentlistan till programmet. Frekvenstabellen avbildar varje ord till antalet gånger det förekommer i argumentlistan. För att förstå vad som egentligen händer, får ni för närvarande inte använda er av automatisk typomvandling (boxing, unboxing), utan ni måste göra detta explicit. Använd därför objekt av klassen `IncrementableInteger` nedan, som "värden" i avbildningen. Frekvenstabellen skall alltså ha den statiska typen `Map<String, IncrementableInteger>`.

```
public 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);
    }
}
//IncrementableInteger
```

- b) Skriv nu samma program utan att använda `IncrementableInteger`. Använd automatisk typomvandling där ni kan, d.v.s. programmet ser ut att spara och hämta `int`:s. Deklarationen av avbildningen får nu den statisk typen `Map<String, Integer>`. Kan ni se några nackdelar med denna version?

Uppgift 2

Betrakta klassen `DummyDates` på nästa sida. Klassen skapar en (något meningslös) lista över slumpade datum. Klassen tillhandahåller även en iterator för att stega sig igenom de genererade datumen. Men iteratorn har brister, identifiera dessa!

Till din hjälp har du följande utdrag ur Java's 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 <code>NoSuchElementException</code> 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 <code>UnsupportedOperationException</code> if the remove operation is not supported by this Iterator. Throws <code>IllegalStateException</code> 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.

```

import java.util.Date;
import java.util.Iterator;
import java.util.Random;
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--;
        }
    } //DSIterator
} //DummyDates

```

Uppgift 3

Implementera metoden

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

som returnerar mängden av element som antingen finns i mängden **s1** eller mängden **s2** (elementen får alltså inte förekomma både i **s1** och **s2**). 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.

Uppgift 4

Nedan ges ett enkelt testprogram som lagrar NUMRUNS slumpmässigt genererade objekt av typen `Integer` i en samling och därefter generera ytterligare NUMRUNS slumpmässigt genererade objekt av typen `Integer` samt söker efter dessa i samlingen.

Tre olika samlingar använts i testprogrammet: en trädstruktur (`TreeSet`), en hashtabell (`HashSet`) och en länkad lista (`LinkedList`). Körningstider (i sekunder) för att utföra `add`-operationerna och `contains`-operationerna beräknas för respektive samling. I tabellen nedan redovisas resultaten för för exekveringar av programmet då 100k respektive 5M slumpstal genererades. Vardera av x, y, z motsvarar varsin samling (datastruktur). Vilken datastruktur motsvaras av x, y respektive z? Förklara varför resultaten ser ut som de gör. Är någon konsekvent bättre med avseende på tidsåtgång?

Implementation	100k add	100k contains	5M add	5M contains
x	0.04	51	0.72	> 100 min
y	0.13	0.1	8.3	4.8
z	0.06	0.04	2.89	0.96

```
import java.util.*;
public class Timings {
    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();
        }
    }
}
```

Uppgift 5

Betrakta klasserna nedan:

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

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

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

import java.util.*;
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");
        }
    }
} //Vehicles
```

- När `main`-metoden i klassen `Vehicles` exekveras krashar programmet. Varför?
- Skriv om klassen `Vehicles` så att felet upptäcks vid kompilering istället för vid exekveringen.

Uppgift 6

Syftet med Singleton-mönstret är att kunna garantera att *endast en instans* av en viss klass existerar. Oftast skall också denna instans komma åt utifrån klassen. Nedan ges en klass med enbart klassmetoder. Är denna klass en singleton? Om inte, hur ska vi förändra klassen så att den blir en singleton? Spelar det någon roll om klassen är en singleton eller inte?

```
public class GameTile {
    // code not show here
} //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;
            }
        }
    }
} //GameUtils
```

Uppgift 7

I uppgift 2 på övningen vecka 4 behandlades gränssnittet `Comparator`, och vi implementerade en klass `public class CarComparator implements Comparator<Car>`

för att sortera objekten i klassen `Car` nedan i stigande ordning med avseende på:

1. `manufacturer`
2. `modelYear`
3. `modelName`

```
public final class Car {
    private final String manufacturer;
    private final String modelName;
    private final int modelYear;
    private final String serialNumber;
    public Car(String manufacturer, String modelName, int modelYear, String serialNumber) {
        this.manufacturer = manufacturer;
        this.modelName = modelName;
        this.modelYear = modelYear;
        this.serialNumber = serialNumber;
    }
    public String getManufacturer() {
        return manufacturer;
    }
    public String getModelName() {
        return modelName;
    }
    public int getModelYear() {
        return modelYear;
    }
    public String getSerialNumber() {
        return serialNumber;
    }
    public String toString() {
        return "manufacturer: " + this.manufacturer + " modelName: " + this.modelName
            + " modelYear: " + this.modelYear + " serialNumber: " + this.serialNumber;
    }
} //Car
```

Objekten i `Car` kan dock sorteras efter många andra ordningar än den som implementeras av `CarComparator` ovan. Eftersom `Car` har 4 instansvariabler kan dessa ordnas på $4! = 24$ olika sätt. Dessutom kan varje instansvariabel sorteras efter både stigande och fallande ordning. Det totala antalen ordningar som objekten i `Car` kan sorteras på blir därför $2^4 * 4! = 16 * 24 = 384$ olika sätt. Det behövs således 384 olika implementationer av gränssnittet `Comparator` för att få alla totala ordningar för objekten i klassen `Car`.

Ett betydligt flexiblare sätt är att nyttja designmönstren *Decorator* och *Template Method*, genom att skapa en jämförare för varje instansvariabel och seriekoppla dessa efter behov. Implementera denna idé för klassen `Car`.