

Övning 6

Denna vecka ska vi titta på undantag, samlingar, generiska enheter, testning (demo) och designmönstret Iterator.

Uppgift 1 Exceptions

- a) En fördel med exceptions är möjligheten att propagera felrapportering uppåt i anropsstacken.

Betrakta programskelleten för metoderna `method1()`, `method2()` och `method3()` nedan:

```
public void method1() {           public void method2() {           public void method3() {
    //code block A                //code block C                //code block E
    method2();                   method3();               readFile();
    //code block B                //code block D                //code block F
}                                }                               }
```

Antag att metoden `readFile()` är den fjärde metoden som anropas i en serie av nästlade anrop: vår `main`-metod anropar `method1()`, som anropar `method2()`, som anropar `method3()`, som slutligen anropar `readFile()`.

Antag vidare att `method1()` är den enda metoden som är intressant för hantering av fel som uppstått i `readFile()`.

- Gör en ny implementation av metoderna ovan så att fel som uppstår i metoden `readFile()` hanteras i metoden `method1()`. Implementationen skall inte använda exceptions.
Tips: Låt metoden `readFile()` returnera ett booleskt värde för att rapportera om fel har uppstått.
 - Gör en ny implementering av metoderna ovan med användning av exceptions. Antag att `readFile()` kastar en exception av typen `ReadFileFailedException` om ett fel uppstår.
- b) Antag att du i Java har definierat en egen klass `NotANumberException` som en subclass till `Exception`. Vidare har du definierat ytterligare en klass `NotAPositiveNumberException` som en subclass till `NotANumberException`.

- Kommer **catch**-satsen nedan att fånga ett `NotAPositiveNumberException`? Varför/Varför inte?

```
catch(NotANumberException nane) {
    System.out.println("Fångade ett NotANumberException");
}
```

- Kommer **try-catch**-satsen nedan att fungera som avsett? Varför/Varför inte?

```
try {
    ...
}
catch(NotANumberException nane) {
    System.out.println("Fångade " + "ett NotANumberException");
}
catch(NotAPositiveNumberException napne) {
    System.out.println("Fångade ett " + "NotAPositiveNumberException");
}
```

Uppgift 2

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);  
    }  
}
```

- 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 statiska typen `Map<String, Integer>`. Kan ni se några nackdelar med denna version?

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 konstruktörer för mängder.

Uppgift 4

Nedan ges ett enkelt testprogram som lagrar NUMRUNS slumpmässigt genererade heltal i en datasamling och beräknar därefter genomsnittlig söktid för ett stort antal slumpmässigt genererade heltal 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 genomsnittlig tid (i millisekunder) för contains-operationerna beräknas för respektive samling. I tabellen nedan redovisas resultaten för exekveringar av programmet då 100k respektive 200k slumpstal adderades till en tom samling. 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 (s)	Av. contains (ms)	200k add (s)	Av. Contains (ms)
x	0,035	607	0,057	1170
y	0.059	0,68	0,13	0,75
z	0,043	0,25	0,11	0,26

```
import java.util.*;
public class Timings {
    private static final double NANO = 1/1000000000.0;
    private static final Random randomizer = new Random();
    public static void main(String[] args) {
        // Use different collection implementations
        Collection<Integer>[] collections = new Collection[3];
        collections[0] = new TreeSet<Integer>();
        collections[1] = new HashSet<Integer>();
        collections[2] = new LinkedList<Integer>();
        final int NUMRUNS = Integer.parseInt(args[0]);
        for (Collection<Integer> c : collections) {
            System.out.print(c.getClass().getName() + "\t");
            time(c, NUMRUNS);
            System.out.println();
            c.clear();
        }
    }
    private static void timeAdds(Collection<Integer> c, int runs) {
        // Add a lot. . .
        long startTime = System.nanoTime();
        for (int i = 0; i < runs; i++)
            c.add(randomizer.nextInt());
        System.out.print(runs + " add: " + (System.nanoTime() - startTime)*NANO + "\t");
        timeContains(c);
    }
    private static void timeContains(Collection<Integer> c) {
        // Search a lot. . .
        long startTime = System.nanoTime();
        for (int i = 0; i < 10000; i++)
            c.contains(randomizer.nextInt());
        System.out.print("Av. Contains (ms): " + (System.nanoTime() - startTime)/10000.0/1000 + "\t");
    }
}
```

Uppgift 5

Betrakta klasserna nedan:

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

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

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

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");
        }
    }
}
```

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

Uppgift 6

Betrakta klassen **DummyDates** nedan. 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 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.

```
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 the 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--;
        }
    }
}
```