# 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 programskelletten för metoderna method1(), method2() och method3() nedan:

```
public void method1() {
   //code block A
   method2();
   //code block B
}
```

```
public void method2() {
   //code block C
   method3();
   //code block D
}
```

```
public void method3() {
   //code block E
   readFile();
   //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().

i) 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.

ii) 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 subklass till Exception. Vidare har du definierat ytterligare en klass NotAPositiveNumberException som en subklass till NotANumberException.

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

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

ii) 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 statisk 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 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 exekveringar av programmet då 100k respektive 5M slumptal genererades. Vardera av x, y, z motsvarar varsin samling (datastruktur). Vilken datastrukturer 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 {
   private static final double NANO = 1/1000000      000.0;
   public static void main(String[ ] args) {
      // Use dierent 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]);
      Random randomizer = new Random(1234);
      for (Collection<Integer> c : colls) {
         // Add a lot. . .
         long startTime = System.nanoTime();
         for (int i = 0; i < NUMRUNS; i++) {
            c.add(randomizer.nextInt());
         }
         System.out.println("add " + c.getClass() + "\t" + ((System.nanoTime() - startTime) * NANO));
         // . . .and search a lot
         startTime = System.nanoTime();
         for (int i = 0; i < NUMRUNS; i++) {
            c.contains(randomizer.nextInt());
         }
         System.out.println("contains " + c.getClass() + "\t" + ((System.nanoTime() - startTime) * NANO));
         c.clear();
      }
   }
}
```

**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 "Ferrarati 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 krashar 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--;
      }
   }
}
```