Övning vecka 4.

Denna vecka ska vi titta på gränssnitten Comparable och Comparator, exceptions, designmönstret Strategy samt icke-muterbarhet kontra muterbarhet. Vi gör också återbesök på temat "programming by contract".

Uppgift 1 Comparator & Comparable

Vi kan för en viss klass göra så att den får en total ordning genom att låta den implementera gränssnittet Comparable. I de fall då vi vill tillhandahålla mer än en ordning för en klass använder vi oss av gränssnittet Comparator. Detta kan vi så klart också använda för att ordna objekt av en klass som inte redan implementerar Comparable.

a) Betrakta klassen Car nedan:

```
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;
    }

    // Accessors go here. Not shown.

    public String toString() {
        return " manufacturer: " + this.manufacturer+ " modelName: " + this.modelName
                + " modelYear: " + this.modelYear + " serialNumber: " + this.serialNumber;
    }
}//Car
```

Vi vill att objekt av klassen Car skall kunna sorteras i stigande ordning med avseende på följande variabler:

1. modelYear
2. manufacturer
3. modelName

Om två bilar är av samma årsmodell så jämför man också tillverkare, om tillverkare också är lika, jämför också modellnamn.

Skriv om klassen Car så att den implementerar gränssnittet Comparable. Metoden compareTo i Car skall ge den totala ordningen enligt ovan.

The Java Language Specification rekommenderar att metoden compareTo() är konsistent med metoden equals(), d.v.s. att om a.compareTo(b) ger att a och b är lika skall även a.equals(b) ge att a och b är lika samt omvänt. Implementera därför även metoden equlas i klassen Car.

b) Det kan tänkas att man inte kan förutspå alla totala ordningar som kan vara av intresse vid en senare tidpunkt.

Ett vanligt fall är då vi i en applikation visar en tabell och man genom att klicka på kolumnernas rubriker bestämmer om posterna ska sorteras i stigande eller fallande ordning med avseende på värdet i kolumnen. Givet att vi har n kolumner så kan dessa sorteras (med ett av alternativen stigande eller fallande) på n! olika sätt.

Om vi dessutom lägger till egenskapen att vi ska kunna ha både stigande och fallande ordning för varje fält individuellt så får vi $2^n n!$ möjliga ordningar. Även för små n blir detta snabbt ohanterligt. Vi skulle t.ex. behöva 384 olika jämförare för att få alla totala ordningar av 4 fält ($2^4 4! = 16 * 24 = 384$).

Klassen CarComparator nedan är ett skelett för att kunna tillhandahålla alla möjliga jämförare för att ordna objekt av typ Car.

```
public enum Field {
   MANUFACTURER, MODELNAME, MODELYEAR, SERIAL
}//Field
public enum Order {
   ASCENDING, DESCENDING
}//Order

/**
  * A comparator that can provide all total orderings for a Car-object.
  * @inv Describe reasonable class invariants.
*/
class CarComparator implements Comparator<Car> {
   private final Field[ ] f;
   private final Order[ ] o;
   /**
   * @pre Describe reasonable preconditions.
   */
   public CarComparator(Field[ ] f, Order[ ] o) {
       // Your implementation.
   }
   public int compare(Car c1, Car c2) {
       // Your implementation.
   }
}//CarComparator
```

Vilka förvillkor för konstruktorn är rimliga? Har t.ex. alla par av fält och ordning en vettig tolkning? Ger alla unika val av t.ex. par och tripplar distinkta totala ordningar?

Tips: Jämför storleken på definitionsmängden för konstruktorn med antalet totala ordningar för 2 fält. Definitionsmängdens storlek är antalet element i den cartesiska produkten, alltså antalet möjliga distinkta tupler för parameter1 gånger antalet möjliga indata för parameter2, et.c. Om vi t.ex. har en metod method(**int** a, **int** b) så är definitionsmängdens storlek $2^{32} * 2^{32} = 2^{64}$ - en **int** i Java har ju 32 bitar. Två **int** efter varandra blir då 64 bitar som tillsammans kan anta 2^{64} olika värden.

Den cartesiska produkten för ett jämförelsekriterium är alltså i detta fall antalet datafält (tillverkare, modellnamn, årsmodell och serienummer) gånger antalet ordningar för dessa fält (stigande, fallande). Om vi kallar storleken på denna mängd *p* så blir definitionsmängdens storlek för n jämförelsekriteria p^n.

Det bör gå att ställa upp åtminstone sju separata förvillkor för f och o tillsammans.

För att skapa en någorlunda flexibel, om än kanske inte så snabb, lösning på det allmänna fallet passar designmönstret Decorator utmärkt. Om detta ännu inte gåtts igenom på kursen kan man senare återkomma hit och fundera på hur en sådan lösning kan se ut.

Uppgift 2

Strategier är en lämplig taktik att ta till då bara delar av en implementation varierar. Gränssnittet Comparator är ett exempel inom Javas standard-API som är en ren strategi. I en typisk sorteringsalgoritm är det lämpligt att bryta ut själva jämförelsen av elementen från resten av algoritmen. Som vi sett i problemet med CarComparator ovan så kan jämförelsekriterierna också förändras under körning.

a) Betrakta klassen Integrator nedan.

Metoden integrate gör en numerisk skattning av integralen

$$\int_a^b f_k(x)\,dx$$

där $f_k(x)$ är någon av funktionerna $f_1(x) = \log(\log(x))$, $f_2(x) = x^x$, $f_3(x) = \sin(\sin(x))$.

```
public class Integrator {
   public enum Function {
      FUNCTION1, FUNCTION2, FUNCTION3
   };
   public static double integrate(Function f, double a, double b, int steps) {
      double sum = 0.0;
      double previous = fun(f, a);
      double delta = (b - a) / steps;
      for (int i = 1; i <= steps; i++) {
         double x = a + (b - a) * i / steps;
         double current = fun(f, x);
         // Compute the integral through Simpson's 3/8 rule.
         sum += delta / 8 * (previous + current
                             + 3 * (fun(f, (2 * (x - delta) + x) / 3.0)
                             + fun(f, (3 * x - delta) / 3.0)));
         previous = current;
      }
      return sum;
   }
   private static double fun(Function f, double x) {
      switch (f) {
         case FUNCTION1: { return Math.log(Math.log(x)); }
         case FUNCTION2: {return Math.pow(x, x); }
         case FUNCTION3:{ return Math.sin(Math.sin(x)); }
         default:
            throw new IllegalArgumentException("Illegal function!");
      }
   }
}// Integrator
```

Ett allvarligt problem med klassen Integrator är att vi måste förändra den varje gång vi numeriskt vill integrera en ny funktion, trots att metoden för själva integrationen inte förändras. Vi kan därmed inte skapa nya funktioner under körning.

Skriv om klassen Integrator så att den istället för fasta funktioner tar en strategi som parameter till metoden integrate. Du behöver bara skapa ett korrekt gränssnitt (interface) samt förändra metoden integrate på lämpligt vis.

Tips: Enum-typen Function ska inte finnas med i lösningen.

Uppgift 3 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 boolskt 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 4

Betrakta klassen Pair nedan:

```
/**
 * Immutable class for generic pairs.
 * @inv State your expected invariant(s) here.
*/
public final class Pair<A, B> {
    /**
     * The first element of the pair.
    */
    public final A first;
    /**
    * The second element of the pair.
    */
    public final B second;
    /**
     * Makes a new pair of type <A,B>.
     * @param first the first element of the pair.
     * @param second the second element of the pair.
    */
    public Pair(final A first, final B second) {
        this.first = first;
        this.second = second;
    }
    @Override
    public boolean equals(final Object o) {
        return o instanceof Pair<?, ?> && ((Pair<?, ?>) o).first.equals(this.first)
                && ((Pair<?, ?>) o).second.equals(this.second);
    }
    @Override
    public int hashCode() {
        return this.first.hashCode() * 13579 + this.second.hashCode() * 23456789;
    }
    @Override
    public String toString() {
        return "Pair: <" + this.first.toString() + "," + this.second.toString() + ">";
    }
}//Pair
```

Avsikten med denna klass är att den ska kunna fungera då man behöver en ad-hoc sammansättning av data där det inte finns någon tydlig operation på just denna kombination av data. Denna klass tillåter sammansättning av par.

Då Java inte har något inbyggt stöd för tupler kan detta (för specialfall, t.ex. par, triplar, etc.) simuleras som i klassen Pair. Vi har inte pratat om generics på kursen ännu. Om du inte har en klar bild av vad <A, B> i klassdeklarationen betyder, strunta i det, man måste inte förstå det för denna uppgift.

Ett exempel på användning av klassen Pair finns i (den något artificiella) klassen SizeAndMin nedan:

```
import java.util.*;
public class SizeAndMin<E extends Comparable<E>> {
    /**
    * Returns a pair containing the number of elements and the minimum element
    * of an iterable collection.
    */
    public Pair<Integer, E> sizeAndMin(final Iterable<E> elements) {
        final Iterator<E> it = elements.iterator();
        if (!it.hasNext()) {
            throw new IllegalArgumentException("min and max undefined for empty sets.");
        }
        E min = it.next();
        int i = 0;
        while (it.hasNext()) {
            E e = it.next();
            if (e.compareTo(min) < 0) {
                min = e;
            }
            i++;
        }
        return new Pair<Integer, E>(Integer.valueOf(i), min);
    }
    public static void main(final String[ ] args) {
        SizeAndMin<String> m = new SizeAndMin<String>();
        Pair<Integer, String> p = m.sizeAndMin (Arrays.asList(args));
        System.out.println("Elements: " + p.first + " Min: " + p.second);
    }
}//SizeAndMin
```

a) Klassen Pair har publika instansvariabler. Detta bryter rimligtvis mot god sed med avseende på inkapsling och synlighet. Ställer detta till det på något vis i klassen Pair?

b) Variablerna first och second är deklarerade **final**. Vad är den tänkta klassinvarianten (formulera informellt) och kan man garantera att den håller? Jämför gärna med uppgift 3 från vecka 3:s övning.