Övning vecka 4.

Denna vecka ska vi titta på icke-muterbarhet kontra muterbarhet, gränssnitten Comparable och Comparator, designmönstret Strategy, exceptions samt equals-metoden .

Uppgift 1 Icke-muterbarhet kontra muterbarhet

Betrakta klassen Period nedan:

```
/**
* @invariant getStart() is before or at the same time as getEnd()
* @invariant getStart() consistently returns the same value after object creation
* @invariant getEnd() consistently returns the same value after object creation
*/
import java.util.Date;
public final class Period {
   private final Date start;
   private final Date end;
   /**
   * @param   start the beginning of the period
   * @param   end the end of the period
   * @pre     start <= end
   * @post    The time span of the returned period is positive.
   * @throws  IllegalArgumentException if start is after end
   * @throws  NullPointerException if start or end is null
   */

   public Period(Date start, Date end) {
      if (start.compareTo(end) > 0) {
         throw new IllegalArgumentException(start + " after " + end);
      }
      this.start = start;
      this.end = end;
   }

   public Date getStart() {
      return start;
   }

   public Date getEnd() {
      return end;
   }
}//Period
```

Klassen Period representerar ett datumintervall genom att innehålla ett startdatum och ett slutdatum. I kommentarerna till klassen visas dessutom vilket kontrakt implementationen av klassen måste uppfylla ("Programming by Contract"). En *invariant* är ett villkor som måste vara uppfyllt före och efter varje metodanrop, ett *förvillkor* utgör krav som måste vara uppfyllda innan en metod anropas och *eftervillkor* visar samband som gäller efter att metoden körts.

a) Några klassinvarianter för Period innebär att avsikten är att klassen skall vara *icke-muterbar*. Vilka?

b) Att en klass är icke-muterbar innebär att när ett objekt av klassen väl instansierats så kommer objektet under resten av sin livslängd att ha samma värden som då det skapades.

Har programmeraren gjort sitt jobb eller finns det sätt att bryta mot klassens invarianter? Om så är fallet, vad blir konsekvensen och hur kan vi åtgärda felen?

Uppgift 2 Comparator & Comparable

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;
   }
   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
```

a) 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 jämför man också tillverkare, om tillverkare också är lika, jämförs även 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. för två objekt a och b där a.compareTo(b) == 0 skall gälla att a.equals(b) == true. Om detta inte uppfylls kan det t.ex. inträffa underligheter när objekten lagras i samlingar. Implementera därför även metoden equlas i klassen Car.

b) 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).

Skriv en klass

```
public class CarComparator implements Comparator<Car>
```

som kan användas för att sortera objekten i klassen Car i stigande ordning med avseende på:

1. manufacturer
2. modelYear
3. modelName

Om två bilar har samma tillverkare jämför man också årsmodell, om årsmodellen också är lika, jämförs även modellnamn.

Uppgift 3 Designmönstret Strategy - Encapsulate what varies

Strategier är en lämplig taktik att ta till då bara delar av en implementation varierar. Gränssnittet Comparator är ett exempel inom Javas standard-API på en strategi. I en typisk sorteringsalgoritm är det lämpligt att bryta ut själva jämförelsen av elementen från resten av algoritmen, därigenom kan jämförelsekriterierna förändras under körning.

Betrakta klassen Integrator nedan:

```
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
```

Metoden integrate gör en numerisk skattning av integralen

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

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

Ett allvarligt problem med klassen Integrator är den inte stödjer *Open-Closed principen*. Vi måste förändra klassen varje gång vi 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 en korrekt strategi samt förändra metoden integrate på lämpligt vis.

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

Uppgift 4 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
}//method1
```

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

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

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 5 Likhet (equals)

Anta att standardklassen Float, som representerar reella tal som objekt, är implementerad enligt följande:

```
public class Float {
   private float f;
   public Float(float f) {
      this.f = f;
   }//constructor
   public float floatValue() {
      return f;
   }//floatValue
   public boolean equals(Object o) {
      if (! (o instanceof Float))
         return false;
      float f1 = this.floatValue();
      float f2 = ((Float) o).floatValue();
      return (f1 == f2);
   }//equal
   // more methods here that don't interest us now
}//Float
```

Pelle Hacker beslutar att skriva en förbättrad variant av Float-klassen genom att göra en subklass TolerantFloat som tolererar små avvikelser när man jämför huruvida två reella värden är lika. Pelles kod har följande utseende:

```
public class TolerantFloat extends Float {
   public static final float TOLERANCE = 0.01f;
   public TolerantFloat(float f) {
      super (f);
   }//constructor
   public boolean equals(Object o) {
      if (!(o instanceof Float))
         return false;
      float f1 = this.floatValue();
      float f2 = ((Float) o).floatValue();
      return (Math.abs(f1 - f2) <= TOLERANCE);
   }//equals
}// TolerantFloat
```

Java Language Specification säger att metoden equals -metoden skall uppfylla följande relationer;

Reflexivitet:	a.equals(a)
Symmetri:	a.equals(b) $\Rightarrow$ b.equals(a)
Transitivitet:	a.equals(b) && b.equals(c) $\Rightarrow$ a.equals(c)

i) Är TolerantFloat.equals() reflexiv? Om inte ge ett, ge ett exempel som visar detta.

ii) Är TolerantFloat.equals() symmetrisk? Om inte ge ett, ge ett exempel som visar detta.

iii) Är TolerantFloat.equals() transitiv? Om inte ge ett, ge ett exempel som visar detta.

b) Betrakta klasserna Point och NamedPoint nedan:

```
public class Point {
   private final int x;
   private final int y;
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }

   @Override
   public boolean equals(Object o) {
      if (!(o instanceof Point)) {
         return false;
      } else {
      Point p = (Point) o;
      return p.x == x && p.y == y;
      }
   }
}//Point

public class NamedPoint extends Point {
   private final String name;
   public NamedPoint(int x, int y, String name) {
      super(x, y);
      this.name = name;
   }

   @Override
   public boolean equals(Object o) {
      if (!(o instanceof Point)) {
         return false;
      }
      // If o is a normal Point, do a name-blind comparison
      if (!(o instanceof NamedPoint)) {
         return o.equals(this);
      }
      // o is a NamedPoint; do a full comparison
      return super.equals(o) && ((NamedPoint) o).name == name;
   }
}// NamedPoint
```

Håller relationerna reflexivitet, symmetri och transivitet för equals-metodenför dessa två klasser? Om inte, vilken eller vilka relationer håller inte? Om någon av relationerna inte håller, hur borde man implementerat equals istället?