Övning 4.

I denna övning ska vi titta på icke-muterbarhet kontra muterbarhet, samt metoderna equals, hashCode och clone.

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;
   }
}
```

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

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);
   }
   @Override
   public boolean equals(Object o) {
      if (!(o instanceof Float))
         return false;
      float f1 = floatValue();
      float f2 = ((Float) o).floatValue();
      return (Math.abs(f1 - f2) <= TOLERANCE);
   }
}
```

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 exempel som visar detta.

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

iii) Är TolerantFloat.equals() transitiv? Om inte, 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;
        }
    }
}

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.equals(name);
    }
}
```

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

Uppgift 3

Betrakta nedanstående klass:

```
public class Product {
   final private int productNumber;
   private String name;
   private String description;
   private int price;
   //code not of interest here
}
```

Klassen behöver förses med metoderna equals och hashCode. Nedan ges tre möjliga implementationer av equals respektive tre möjliga implementationer av hashCode :

equals version 1:

```
public boolean equals(Object obj) {
   if (this == obj)
      return true;
   if (obj == null)
      return false;
   if (getClass() != obj.getClass())
      return false;
   Product other = (Product) obj;
   return productNumber == other.productNumber ;
}
```

equals version 2:

```
public boolean equals(Object obj) {
   if (this == obj)
      return true;
   if (obj == null)
      return false;
   if (getClass() != obj.getClass())
      return false;
   Product other = (Product) obj;
   return name.equals(other.name) &&
          productNumber == other.productNumber ;
}
```

equals version 3:

```
public boolean equals(Object obj) {
   if (this == obj)
      return true;
   if (obj == null)
      return false;
   if (getClass() != obj.getClass())
      return false;
   Product other = (Product) obj;
   return name.equals(other.name) &&
          description.equals(other.description);
}
```

hashCode version 1:

```
public int hashCode() {
   return productNumber;
}
```

hashCode version 2:

```
public int hashCode() {
   return name.hashCode() + productNumber;
}
```

hashCode version 3:

```
public int hashCode() {
   return name.hashCode();
}
```

a) Med vilka versioner av equals kan *version 1* av hashCode kombineras utan att bryta det kontrakt som *The Java Language Specification* specificerar. Motivera ditt svar!

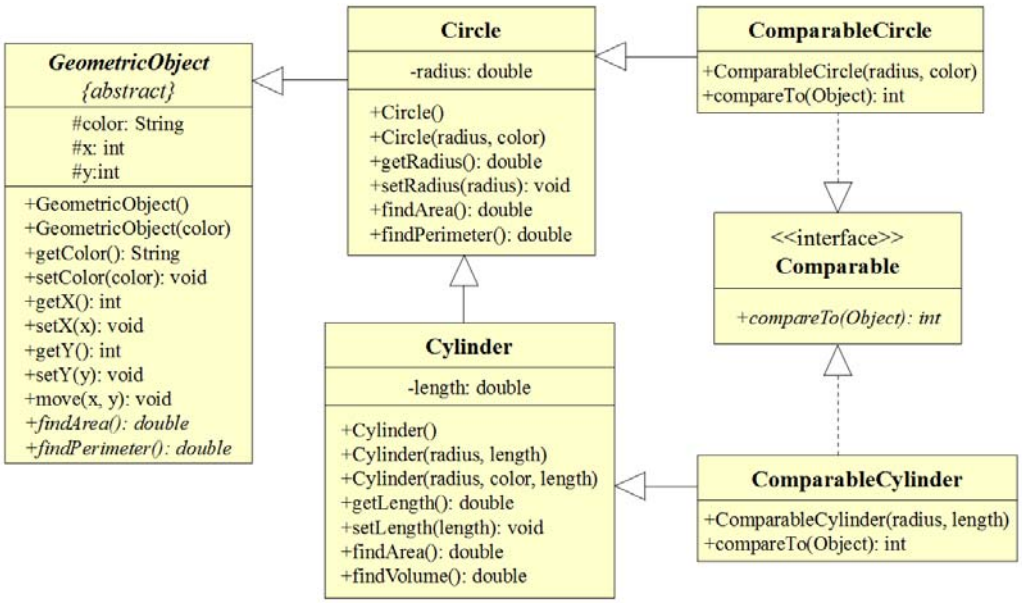
b) Med vilka versioner av equals kan *version 2* av hashCode kombineras utan att bryta det kontrakt som *The Java Language Specification* specificerar. Motivera ditt svar!

c) Med vilka versioner av equals kan *version 3* av hashCode kombineras utan att bryta det kontrakt som *The Java Language Specification* specificerar. Motivera ditt svar!

d) Vilken version av equals är lämpligast att avvända. Motivera ditt svar!

Uppgift 4

På en nyligen avslutad grundkurs i objektorienterad programutveckling illustrerades arv med följande klasser och interface (för att avbilda geometriska objekt som har en viss färg och en viss placering i det tvådimensionella rummet):



Det är i och för sig något suspekt att kunna positionera tredimensionella objekt i ett tvådimensionellt rum, men bortse från detta när du besvarar nedanstående frågor.

a) Identifiera och förklara de brister som finns i ovanstående design med avseende på de arvsrelationer som finns.

b) Betrakta synligheten av instansvariablerna i respektive klass. Kommentera!

c) Betrakta uppsättningen konstruktorer i respektive klass. Kommentera!

d) Finns det något att säga om hur tillstånden i klassen GeometricObject har avbildats (dvs vilka instansvariabler som har specificerats och typerna på dessa).

e) Implementationen av klassen ComparableCircle har följande utseende:

```
public class ComparableCircle extends Circle implements Comparable {
    //kod för konstruktorer utelämnad
    public int compareTo(Object o) {
        if (o instanceof  ComparableCircle) {
            if (getRadius() > ((Circle) o).getRadius()) {
                return 1;
            } else if (getRadius() < ((Circle) o).getRadius()) {
                return -1;
            } else {
                return 0;
            }
        }
        throw new IllegalArgumentException();
    }
}
```

Klassen borde implementera det generiska gränssnittet Comparable<E> och inte det *råa* gränssnittet Comparable. Varför? Skriv om klassen så den implementerar gränssnittet Comparable<E>.

Uppgift 5

Betrakta nedanstående två klasser för att representera punkter i planet respektive polygoner:

```
public class Point {
   private int x, y;
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
   public int getX() {
      return x;
   }
   public void setX(int x) {
      this.x = x;
   }
   public int getY() {
     return y;
   }
   public void setY(int y) {
      this.y = y;
   }
}
```

```
public class Polygon {
   private ArrayList<Point> points;
   private int noOfPoints;
   //constructors and methods not shown here.
}
```

a) Skriv en clone-metod för klassen Point, samt ange vad som i övrigt behöver göras för att klassen Point skall bli kloningsbar.

b) Skriv en clone-metod, som returnerar en djup kopia, för klassen Polygon, samt ange vad som i övrigt behöver göras för att klassen Polygon skall bli kloningsbar.

Uppgift 6 Samma kontra lika

Betrakta nedanstående klass:

```
public class References {
    public static void main(String[ ] args) {
        stringRefs();
        integerRefs();
    }

    private static void printIntegerEq(String exp, boolean b) {
        System.out.println(exp + " -> " + b);
    }

    private static void printStringIdEq(String aName, String bName ,Object a, Object b) {
        System.out.println(aName + " == " + bName + " -> " + (a == b) + "\t"
                + aName + ".equals(" + bName + ") -> " + a.equals(b));
    }

    private static void stringRefs() {
        String s1 = new String("False");
        String s2 = new String("False");
        printStringIdEq ("s1", "s2", s1, s2);
        String s3 = "True";
        String s4 = "Tr" + "ue";
        printStringIdEq ("s3", "s4", s3, s4);
        String s5 = "False";
        String sx = "F";
        String s6 = sx + "alse";
        printStringIdEq ("s5", "s6", s5, s6);
    }

    private static void integerRefs() {
        Integer i = new Integer(2);
        Integer j = new Integer(2);
        printIntegerEq ("i >= j", i >= j);
        printIntegerEq ("i == j", i == j);
        printIntegerEq ("i == 2", i == 2);
    }
}
```

När main-metoden exekveras erhålls följande utskrifter:

```
s1 == s2 -> false       s1.equals(s2) -> true
s3 == s4 -> true        s3.equals(s4) -> true
s5 == s6 -> false       s5.equals(s6) -> true
i >= j -> true
i == j -> false
i == 2 -> true
```

Troligen är inte detta vad man förväntar sig! För att förstå varför utskriften blir som den blir måste man söka förklaringen i hur Java har implementerar klassen String och omslagsklassen Integer. Slå upp detta i *The Java Language Specification, Java SE 8 Edition* (delkapitel 3.10.5 respektive 5.1.1 – 5.1.8) och förklara sedan resultatet i ovanstående utskrift.