Lösningsförslag: Övning vecka 4.

Uppgift 1

a) De två invarianterna

@invariant getStart() consistently returns the same value after object creation
@invariant getEnd() consistently returns the same value after object creation

anger att klassen ska vara icke-muterbar.

b) Programmeraren har gjort två fel:

- Typerna hos konstruktorns två parametrar är inte primitiva eller icke-muterbara, vilket gör att kopior av objekten måste tas. Annars kan den som skapar ett Period-objekt behålla referenser till dessa objekt. När konstruktorn har kört färdigt och kontrollerat att start är mindre eller lika med end, kan den utomstående då förändra detta förhållande. Kopior behöver därför tas av objekten innan dessa tilldelas instansvariablerna start och end.
- I metoderna getStart() och getEnd() returneras referenser till Period-objektets privata Date-objekt. En utomstående får då möjlighet att förändra innehållet i dessa objekt. Istället måste kopior göras av objekten och referenser till dessa nya objekt returneras.

En korrekt lösning:

```
/**
 * @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; must not precede start
     * @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 = new Date(start.getTime());
        this.end = new Date(end.getTime());
    }

    public Date getStart() {
        return new Date(start.getTime());
    }

    public Date getEnd() {
        return new Date(end.getTime());
    }
}
```

Uppgift 2

i) Ja, equals-metoden är reflexiv. x.equals(x) ger **true** för alla objekt x av typen Float och för alla objekt x av typen TolerantFloat.

ii) Nej, equals-metoden är inte symmetrisk. Om vi skapar objekten

```
Float f = new Float(1);
TolerantFloat tf = new TolerantFloat(1.01);
```

ser vi att

```
tf.equals(f)  --> true
```

men

```
f.equals(tf)  --> false
```

iii) Nej, equals-metoden är inte transitiv. Om vi skapar objekten

```
TolerantFloat tf1 = new TolerantFloat(1);
TolerantFloat tf2 = new TolerantFloat(1.01);
TolerantFloat tf3 = new TolerantFloat(1.02);
```

ser vi att

```
tf1.equals(tf2) --> true
tf2.equals(tf3) --> true
```

men att

```
tf1.equals(tf3) --> false
```

b) I klassen NamedPoint har man infört en värdekomponent genom instansvariabeln name. För att åtgärda symmetriproblemet testar equals-metoden i klassen NamedPoint om det objekt (o) som aktuellt **this** jämförs med är av typen Point, i sådana fall anropas equals-metoden för objektet o. Men det blir ändå problem, i detta fall med transitiviteten. Här följer ett exempel som visar ett sådant brott. En Point representeras som en tvåtupel med sina x- och y-koordinater, (x,y), och en NamedPoint som en tretupel med x- och y-koordinater följt av namnet, (x,y,namn).

```
p1 = (0, 0, "centrum")
p2 = (0, 0)
p3 = (0, 0, "origo")
p1 equals p2 --> true
p2 equals p3 --> true
p1 equals p3 --> false
```

De två första jämförelserna är ok, eftersom endast koordinaterna testas då en Point är del av jämförelsen. Men det hela fallerar i tredje jämförelsen när namnen tas med.

Det finns ingen annan lösning på problemet än att endast utföra jämförelser mellan objekt av samma typ om man inte kan garantera att klasserna *har samma uppsättning instansvariabler och att samma instansvariabler används för likhetsjämförelse*.

Observera att **instanceof**-operatorn returnerar sant för subtyper och därmed kommer jämförelser med subtyper att tillåtas. Dessa kan innehålla ytterligare instansvariabler, vilket gör att equals-kontraktet bryts. För att undvika detta kan metoden getClass() användas istället för instanceof. Men att inte tillåta jämförelser med subklasser medför ett brott mot LSP, vilket kan ställa till det, t.ex. om objekt av subtyper ska kunna lagras i samlingar. Det måste övervägas om equals-kontraktet eller LSP ska uppfyllas.

En bättre lösning kanske är att använda sig av delegering i stället för arv, enligt nedan:

```
public class NamedPoint {
    private final Point point;
    private final String name;
    public NamedPoint(int x, int y, String name) {
        point = new Point(x, y);
        this.name = name;
    }
    public Point asPoint() {
        return point;
    }
```

```
    @Override
    public boolean equals(Object o) {
        if (!(o instanceof NamedPoint))
            return false;
        NamedPoint np = (NamedPoint) o;
        return np.point.equals(point) && np.name.equals(name);
    }

    // övriga metoder utelämnade
}
```

Uppgift 3

Kontraktet som *The Java Language Specification* specificerar för equals och hashCode är:

x.equals(y) ⇒ x.hashCode() == y.hashCode()

Detta betyder att kontraktet bryts om beräkningen av hashkoden involverar andra instansvariabler än de som används vid beräkningen av likhet. Exempelvis bryts kontrakt om equals variant 1 kombineras med hashCode variant 2, eftersom equals betraktar två objekt som lika om instansvariabeln productNumber har samma värden i de båda objekten, medan beräkningen av hashkoden i hashCode också involverar instansvariabeln name.

Samtliga instansvariabler som används vid beräkningen av likhet behöver inte ingå i beräkningen av hashkoden, eftersom två objekt med samma hashkod inte behöver vara lika.

		equals version 1	equals version 2	equals version 3
a)	hashCode version 1	OK	OK	
b)	hashCode version 2		OK	
c)	hashCode version 3		OK	OK

d) Version 1 är lämpligast, eftersom det endast är instansvariabeln productNumber som vi med säkerhet vet inte förändrar sitt värde.

Uppgift 4

a) *Liskov Substitution Principle* säger att en supertyp är utbytbar med sina subtyper. Med den design som gjorts innebär detta alltså att var som helst där vi förväntar oss en cirkel går det lika bra med en cylinder. Eller med andra ord gäller det att *en cylinder är en cirkel*, vilket naturligtvis är helt galet. Den som gjort designen verkar ha infört implementationsarvet mellan klasserna Cylinder och Circle enbart med återanvändning av kod som motiv, vilket inte är ett tillräckligt motiv.

b) Synligheten i GeometricObject är **protected**. Synligheten bör dock vara **private**. Man skall alltid eftersträva att exponera så lite som möjligt av superklassens interna representation till dess subklasser. Orsaken är givetvis att om den interna representationen är exponerad och används av subklasserna, måste subklasserna förändras ifall superklassen byter representation. I vårt specifika fall tillhandahåller dessutom klassen GeometricObject publika accessmetoder för samtliga instansvariabler, varför inga ytterligare förändringar av klassen behöver göras.

c) Det saknas konstruktorer för att ange var det skapade objektet skall placeras i det två-dimensionella rummet. Är det möjligt skall en instans ges sitt fullständiga tillstånd när instansen skapas.

Uppsättningen konstruktorer är inkonsekvent mellan de olika klasserna. Exempelvis borde ComparableCylinder ha samma uppsättning konstruktorer som Cylinder.

Ur konsekvenssynpunkt kan även ordningen på parametrarna till konstruktorerna ifrågasättas. Exempelvis har klassen Cylinder konstruktorerna Cylinder(radius, length) och Cylinder(radius, color, length), det hade varit lämpligare att den sistnämnda konstruktorn ersatts med Cylinder(radius, length, color).

d) Det hade varit lämpligare att avbilda en färg med exempelvis klassen java.awt.Color (eller en egendefinierad klass) än en sträng. Likaledes hade det varit lämpligare att avbilda en position med exempelvis klassen java.awt.Point (eller en egendefinierad klass) än som två heltal.

e) Genom att implementera gränssnittet Comparable<E> kommer kompilatorn att kontrollera typtillhörighet, dvs att endast objekt av samma typ jämförs. Går ett anrop av metoden compareTo igenom kompileringen vet vi att objekten som jämförs är av samma typ och ingen explicit typomvandling eller felhantering behövs (i metoden compareTo eller i anropande metod).

```
public class ComparableCircle extends Circle implements Comparable<ComparableCircle> {
    //kod för konstruktorerna utelämnad
    public int compareTo(ComparableCircle cc) {
        if (getRadius() > cc.getRadius()) {
            return 1;
        } else if (getRadius() < cc.getRadius()) {
            return -1;
        } else {
            return 0;
        }
    }
}
```

Uppgift 5

a)

```
public class Point implements Cloneable {
    // som tidigare
    public Point clone() {
        try {
            return (Point) super.clone();
        }
        catch (CloneNotSupportedException e) {
            throw new InternalError();
        }
    }
}
```

b)

```
public class Polygon implements Cloneable {
    // som tidigare
    public Polygon clone() {
        try {
            Polygon copy = (Polygon) super.clone();
            copy.points = (ArrayList<Point>) points.clone();
            for (int i = 0; i < points.size(); i++)
                copy.points.set(i, points.get(i).clone());
            return copy;
        }
        catch (CloneNotSupportedException e) {
            throw new InternalError();
        }
    }
}
```

Uppgift 6

- Java gör s.k. *interning* av String-objekt d.v.s. under huven bevaras tidigare skapade strängkonstanter i en samling. När programmeraren skapar en ny String (observera att String är icke-muterbar!), tittar Java först i sin pool om motsvarande textsträng redan existerar där. Om så är fallet, skapas inte något nytt objekt utan en referens till det existerande String-objektet returneras. Men som framgår av övningen är Java begränsat i sin härledningsförmåga - härledningen görs vid kompilering och avser därför endast strängliteraler. Därför kommer ett nytt objekt att skapas, trots att ett med samma värde redan existerar, för uttryck som inte endast består av literaler.

- Det existerar omslagstyper (wrapper types) till alla primitiva typer i Java. Dessa är referenstyper som i sitt objekt lagrar samma typ av värde som den motsvarande primitiva typen. Typen **int** har således en motsvarande omslagsklass Integer, **boolean** har omslagsklassen Boolean o.s.v. En omslagstyp kan i viss omfattning användas på samma plats som sin motsvarande primitiva typ och tvärtom, eftersom det sker automatiska konverteringar. När en omslagstyp används på platsen för en primitiv typ kallas konverteringen "unboxing" (det inneboende primitiva värdet i omslagsklassen packas upp ur sin ”låda”) och det omvända fallet kallas "boxing". I fallet med operatorn "==" sker dock inte någon "unboxing", eftersom Java prioriterar jämförelse mellan objektens referenser (vilket är vad som sker för alla sorters referenstyper [inte bara omslagstyper]).