Instuderingsuppgifter läsvecka 4 - LÖSNINGAR

1.

a) Exponering av den interna representationen innebär att den interna representationen oavsiktligt blir direkt tillgänglig för modifieringar utanför klassen, t.ex. att en instansvariabel borde vara privat blivit publik.

b) Här uppstår exponeringen genom att referensen till instansvariablerna startPoint och endPoint är tillgänglig utanför klassen. Vi måste därför se till att konstruktorn skapar kopior av referenserna som fås som parametrar, samt att metoden getStartpoint och getEndpoint istället för att returnera referenser till instansvariablerna returnerar referenser till kopior av instansvariablerna.

```
import java.awt.Point;
public class Line {
   private Point startPoint;
   private Point endPoint;
   public Line(Point startPoint, Point endPoint) {
      this.startPoint = (Point) startPoint.clone();
      this.endPoint= (Point) endPoint.clone();
   }
   public Point getStartpoint() {
      return (Point) startPoint.clone();
   }
   public Point getEndpoint() {
      return (Point) endPoint.clone();
   }
}//Line
```

2.

Icke-muterbara objekt har många fördelar:

- De är lätta att förstå.
- De är alltid trådsäkra.
- Det är ofarligt att lämna ut referenser till dem. Det gör inget om en massa olika objekt råkar använda ett och samma icke-muterbara objekt.
- Deras interna data kan återanvändas. Om ett nytt icke-muterbart objekt ska skapas där vissa fält inte skiljer sig från de i ett befintligt objekt av samma klass kan alla oförändrade fält referera till exakt samma instanser som i det befintliga objektet.
- Det är lätt att använda icke-muterbara objekt som tillstånd i andra objekt.
- Det är ofarligt att skicka dem till andra processer i distribuerade system.

3.

Se föreläsningsanteckningarna.

4.

Se föreläsningsanteckningarna.

5.

Se föreläsningsanteckningarna.

6.

Att en klass är konsistent innebär att gränssnitten för publika metoder har samma strukturella uppbyggnad avseende namn, parametrar, ordning på parametrar, returvärden och beeteende.

7.

Se föreläsningsanteckningarna.

8.

Se föreläsningsanteckningarna.

9.

Se föreläsningsanteckningarna.

10.

equals-metod skall vara reflexiv, symmetrisk, transitiv och konsistent.

11.

```
public boolean equals(Object o) {
   if (o == null)
      return false;
   if (getClass() != o.getClass())
      return false;
   Dog d = (Dog) o ;
      return breed.equals(d.breed) && name.equals(d.name) && gender.equals(d.gender);
}

public int hashCode() {
   return 23*breed.hashCode() + 17*name.hashCode() + 31*gender.hashCode();
}
```

12.

Se föreläsningsanteckningarna.

13.

Se föreläsningsanteckningarna.

14.

Alternativ d) är felaktig, eftersom undantagsklassen FirstException är en superklass till undantagsklassen SecondException.

15.

```
public class Maximum {
   public static void main(String[] args) {
      if (args.length >= 2 ) {
         try {
            int t1 = Integer.parseInt(args[0]);
            int t2 = Integer.parseInt(args[1]);
            int max = t1 + t2;
            System.out.println("The maximum of " + t1 + " and " + t2 + " is " + max);
         }
         catch (NumberFormatException e) {
            System.out.println("Invalid input values " + e.getMessage());
            System.out.println("The input values must be integers.");
         }
      }
      else
         System.out.println("Usage: java Maximum interger1 integer2");
   }//main
}//Maximum
```

16.

a) och b) är korrekta, medan c) är felaktig. En metod som överskuggar en metod får inte kasta ett kontrollerande undantag om inte metoden som överskuggas gör det. Om detta vore tillåtet skulle den överskuggade metoden i subklassen ställa större krav på klienten än vad metoden gör i superklassen (eller som i detta fall i interfacet), eftersom klienten måste ta hand om undantaget genom att antingen fånga det eller skicka det vidare.

17.

```
public class Printer {
   private Format format;
   public Printer(Format format) {
      this.format = format;
   }
   public void output(String s) {
      format.print(s);
   }
}
public interface Format {
   public void print(String str);
}
public class FormatOne implements Format {
   public void print(String str) {
      System.out.println(str);
   }
}
public class FormatTwo implements Format {
   public void print(String str) {
      System.out.println("<p>" + str + "</p>");
   }
}
public class FormatThree implements Format {
   public void print(String str) {
      System.out.println(str + "//");
   }
}
```

18.

a)

```
public class Book implements Comparable {
   private String title;
   private double price;

   public Book(String title, double price) {
      this. title = title;
      this. price = price;
   }

   public double getPrice() {
      return price;
   }
   public String getTitle() {
      return title;
   }
   public int compareTo(Object obj) {
      return title.compareTo(((Book)obj).getTit le());
   }
   public boolean equals(Object obj) {
      if (this == obj)
         return true;
      if (obj == null)
         return false;
      if (getClass() != obj.getClass())
         return false;
      Book other = (Book) obj;
      return title.equals(obj.title);
   }
}
```

b)

```
import java.util.*;
public class BookComparator implements Comparator <Book> {
    public int compare(Book a, Book b) {
        if (a.getPrice() < b.getPrice())
            return -1;
        if (a.getPrice() == b.getPrice())
            return 0;
        return 1;
    }//compare
}/BookComparator
```