# Instuderingsuppgifter läsvecka 4

**1.**

a) Förklara vad det betyder att den interna representationen i en klass exponeras!

b) Vilka förändringar måste göras i klassen Line nedan, för att förhindra exponering av representationen?

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

**2.**

Ett objekt som är en instans av en *icke-muterbar* klass förändrar inte sitt tillstånd under sin livstid. Redogör för vilka fördelar det finns med icke-muterbara objekt.

**3.**

Hur åstadkommer man att en klass blir icke-muterbar?

**4.**

Redogör för hur tolkningen är av konstruktionen **final** när den används tillsammans med

a) en klass
b) en metod
c) en parameter
d) en instansvariabel

**5.**

En av de viktigaste principerna vid programdesign är *The Single Responsibility Principle*. Förklara vad denna princip innebär! Vad finns det för samband mellan *kohesion*, *koppling* och *The Single Responsibility Principle*?

**6.**

Det publika gränssnittet för en klass skall vara konsistent. Förklara vad det betyder.

**7.**

Förklara med ett exempel vad principen *Expert pattern* innebär.

**8.**

Essensen i *Law of Demeter* beskrivs ofta med parollen *Don't talk to strangers, talk only to your immediate friends*. Förklara *Law of Demeter* och hur denna paroll kommer in i bilden!

**9.**

Förklara innebörden av *The Interface Segregation Principle*.

**10.**

Vilka egenskaper skall en equals-metod uppfylla?

**11.**

Antag att klassen Dog är definierad på följande sätt:

```
public class Dog {
    private String breed;
    private String name;
    private String gender;
    public Dog(String aBreed, String aName, String aGender) { … }
    public String getBreed() { … }
    public String getName() { … }
    public String getGender() { … }
    //Two dogs are equal if they have equal breeds, genders and names.
    public boolean equals(Object o) {…)
    public int hashCode() { … }
    …
}
```

Skriv kod för metoderna equals och hashCode.

**12.**

Vad är det för skillnad mellan kontrollerande respektive icke-kontrollerande undantag? När skall kontrollerande respektive icke-kontrollerande undantag användas?

**13.**

Vad har de olika delarna i **try-catch-finally**-konstrukten för uppgifter? Vilka av dessa delar kan utelämnas?

**14.**

Undantagsklasserna E, F och G samt klasserna Base och Sub definieras enligt:

```
public class E  extends Exception {...}
public class F  extends E {...}
public class G  extends F {...}
public class Base {
    public void method()  throws F {
        ...
    }
}
public class Sub  extends Base {
    @Override
    public void method()  throws ...
}
```

En av nedanstående kan inte vara en överskuggning av method i Sub, vilken?

a) **public void** method() **throws** F,G {...}

b) **public void** method() **throws** F {...}

c) **public void** method() **throws** G {...}

d) **public void** method() **throws** E {…}

**15.**

Utöka nedanstående program med lämplig undantagshantering.

```
public class Maximum {
    public static void main(String[] args) {
        if (args.length >= 2 ) {
            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);
        }
        else
            System.out.println("Usage: java Maximum interger1 integer2");
    }//main
}//Maximum
```

**16.**

Är nedanstående deklarationer korrekta? Om inte, förklara varför!

a)
```
public interface IB {
    public void doIt() throws AnException;
} //IB

public class ImpIB implements IB {
    public void doIt() {
        //the code not shown
    }//doIt
}//ImpplIB
```

b)
```
public class AnException extends RuntimeException {. . .}

public interface IC {
    public void doIt();
} //IC

public class ImpIC implements IC {
    public void doIt() throws AnException{
        //the code not shown
    } //doIt
}// ImpIC
```

c)
```
public class AnException extends Exception {. . .}

public interface ID {
    public void doIt();
} //ID

public class ImpID implements ID {
    public void doIt() throws AnException{
        //the code not shown
    } //doIt
}// ImpID
```

**17.**

Betrakta följande klass:

```
public class Printer {
   private int format;
   public Printer(int format) {
      this.format = format;
   }
   public void output(String s) {
      if (format == 1) {
         System.out.println(s);
      }
      else if (format == 2) {
         System.out.println("<p>" + s + "</p>");
      }
      else if (format == 3) {
         System.out.println(s + "//");
      }
   }//output
}//Printer
```

Designen bryter mot *Open/Closed-principen*; man kan inte lägga till ytterligare utskriftsformat utan att behöva modifiera klassen Printer. Eliminiera detta problem genom att göra om designen med användning av *Strategy*-mönstret.

**18.**

Betrakta klassen Book nedan:

```
public class Book {
   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;
   }
}
```

a) Gör nödvändiga tillägg i klassen så att det införs en naturlig ordning på böckerna, vilken skall definieras efter alfabetisk växande ordning på böckernas titel. Överskugga samtidigt equals-metoden.

b) Det finns också ett behov av att kunna sortera böckerna efter växande ordning på priset (utan att förstöra den naturliga sorteringsordningen efter böckernas titel). Visa hur detta går till.