Instuderingsuppgifter läsvecka 4 - LÖSNINGAR

1.

equals-metod skall vara :

reflexiv: För varje icke-**null** referens x skall det gälla att x.equals(x) returnerar **true**.

symmetrisk: För alla referenser x och y skall det gälla att x.equals(y) returnerar **true** om och endast om y.equals(x) returnerar **true**.

transitiv: För alla referenser x, y och z skall gälla att om x.equals(y) returnerar **true** och y.equals(z) returnerar **true** så skall också x.equals(z) returnera **true**.

konsistent: Om objekten till vilka x och y refererar inte har förändrats skall upprepade anrop av x.equals(y) returnera samma värde.

2.

```
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();
}
```

3.

a) Man låter metoden clone() kasta ett undantag enligt:

```
public class NoCopy {
    . . .
   public NoCopy clone() throws CloneNotSupportedException {
      throw new CloneNotSupportedException();
   }//clone
}//NoCopy
```

b)

```
public class C implements Cloneable {
   private int x;
   private B b;
   . . .
   public C clone() {
      try {
         C copy = (C) super.clone();
         copy.b = b.clone();
         return copy;
      } catch (CloneNotSupportedException e) {
            return null; //never invoked
   }
}
```

c)

```
import java.util.ArrayList;
final public class D implements Cloneable {
   private int x;
   private ArrayList<C> cs;
   . . .
   public D(D other) {
      this.x = other.x;
      cs = new ArrayList<C>();
      for (C item: other.cs) {
         cs.add(item.clone());
      }
   }
   public D clone() {
      return new D(this);
   }
}
```

d) Tekniken med *kopieringskonstruktor* för att implementera clone() skall endast användas när klassen är deklarerade som **final**, dvs när det inte går att skapa subklasser.

4.

Ett designmönster är en lösningen på ett återkommande problem skriven på ett sådant sätt att lösningen kan användas i olika situationer utan att man i detalj behöver använda den på samma sätt i dom olika situationerna.

5.

Designmönstret Singleton är användbart i alla system där exakt en instans av en service får förekomma, t.ex. skrivarkö som används för att fördela arbetet på flera skrivare.

6.

Eftersom konstruktorn är privat! Om synligheten ändras för konstruktorn är det inte en singleton längre, eftersom andra klasser då kan anropa konstruktorn.

7.

```
public abstract class AbstractButton extends JButton implements ActionListener {
   protected Door door;
   public AbstractButton(Door door, String str) {
      super(str);
      this.door = door;
      addActionListener(this);
   }
   public void actionPerformed(ActionEvent event) {
      operation();
   }
   public abstract void operation();
}//AbstractButton

public class OpenButton extends AbstractButton {
   public OpenButton(Door door) {
      super(door, "Open");
   }
   public void operation() {
      door.open();
   }
}//OpenButton

public class CloseButton extends AbstractButton {
   public CloseButton(Door door) {
      super(door, "Close");
   }
   public void operation() {
      door.close();
   }
}//CloseButton
```

8.

```
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 + "//");
   }
}
```

9.

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).getTitle());
   }
   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(other.title);
   }
}//Book
```

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
```

10.

Syftet med designmönstret *State* att göra det möjligt för ett objekt att ändra sitt beteende när objektet ändrar sitt tillstånd.