Instuderingsuppgifter läsvecka 5 - LÖSNINGAR

1.

Om vi använder interfacet List behöver vi inte bry oss om den konkret implementation som används, därför kan implementationen bytas ut genom att endast ändra på ett enda ställe i koden. *Depend upon abstractions, not upon concrete implementations.*

2.

Deklarationen

```
Set<String> stringSet;
```

är att föredra. Det finns två anledningar till detta:

1. vi erhåller ett kompileringsfel istället för få ett exekveringsfel om försöker lägga in något annat än en String i mängden stringSet
2. vi behöver inte göra explicita typomvandlingar när vi hämtar element ur mängden stringSet.

3.

Om man vill använda Collection-ramverkets sorteringshjälpmedel så måste man se till att alla objekt som ska sorteras implementerar gränssnittet Comparable vilket betyder att metoden compareTo () som jämför två objekt av klassen måste finnas.

```
public class Point implements Comparable{
   . . .
   public int compareTo(Point p) { . . . }
}
```

4.

```
TreeSet<Double>  mySet = new TreeSet<Double>( );
mySet.add( new Double(1.0) );
. . .
mySet.add( new Double(0.5) );
Double d = mySet.first();
```

5.

Utskrift av alla elementen i listan med användning av en iterator:

```
Iterator<String> iter = list.iterator();
while (iter.hasNext()) {
      String str = iter.next();
      System.out.println(str);
}
```

Utskrift av alla elementen i listan med användning av den förenklade **for**-satsen:

```
for (String str : list) {
   System.out.println(str);
}
```

6.

Felen inträffar vi satserna:

```
c = cigs.remove(0);   respektive   Cigarette c = cigs.get(i);
```

Metoderna remove och get returnerar typen Object medan variabeln c i båda fallen är av typen Cigarette.

Istället för att göra explicit typomvandling skall följande åtgärder göras:

```
private ArrayList< Cigarette> cigs;
public CigarettePack() {
   cigs = new ArrayList<Cigarette>();
}
```

7.

i. Ja - String implementerar Comparable

ii. Nej - Set är inte en subtyp till List

iii. Ja - ArrayList är en subtyp till List och String implementerar Comparable

iv. Nej - Dog implementerar inte Comparable

8.

Om ordningen som låtarna spelas i, är av betydelse, så är en lista det rätta valet. Om däremot ordningen inte har någon betydelse, men att samma låt inte får spelas flera gånger, så är en mängd rätta valet.

9.

a)

```
public static ArrayList<String> bornThisYear(Person[] people, int year) {
   ArrayList<String> result = new ArrayList<String>();
   for (int i = 0; i < people.length; i = i + 1) {
      if (people[i].getBirthYear() ==  year)
         result.add(people[i].getName());
   }
   return result;
}
```

b)

```
public static ArrayList<String> bornThisYear(ArrayList<Person> people, int year) {
   ArrayList<String> result = new ArrayList<String>();
   for (Person p : people) {
      if (p.getBirthYear() ==  year)
         result.add(p.getName());
   }
   return result;
}
```

```
public static ArrayList<String> bornThisYear(ArrayList<Person> people, int year) {
   ArrayList<String> result = new ArrayList<String>();
   Iterator<Person> p =  people.iterator();
   while (p.hasNext()) {
      Person obj = p.next();
      if (obj.getBirthYear() ==  year)
         result.add(obj.getName());
   }
   return result;
}
```

c)

```
public static void removeNames(ArrayList<Person> people,  ArrayList<String>  names) {
   Iterator<String> n =  names.iterator();
   while (n.hasNext()) {
      String name = n.next();
      Iterator<Person> p =  people.iterator();
      while (p.hasNext()) {
         if (p.next().getName() ==  name)
         p.remove();
      }
   }
}
```

10.

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.

11.

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.

12.

```
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
```

13.

```
public class NewSetAdapter<T>  implements Set<T>{
   private NewSet<T> ns = new NewSet<T>();
   public void add(T e) {
      ns.insert(e);
   }

   public void delete(T e) {
      ns.remove(e);
   }

   public int size() {
      return ns.size();
   }

   public boolean has(T e) {
      return ns.contains (e);
   }
}//NewSetAdapter
```

14.

a) Syftet med designmönstret Decorator är att tillhandahålla ett sätt att dynamiskt lägga till nya beteenden och tillstånd till ett objekt.

b)

```
public interface IA {
   public void f();
}//IA

public class B implements IA {
   private IA ia;
   public B(IA ia) {
      this.ia = ia;
   }
   public void f() {
      ia.f();
      System.out.println("B");
   }
}//B

public class A implements IA {
   public void f() {
      System.out.println("A");
   }
}//A

public class C implements IA {
   private IA ia;
   public C(IA ia) {
      this.ia = ia;
   }
   public void f() {
      ia.f();
      System.out.println("C");
   }
}//C

public static void main(String[] args) {
   IA a = new A();
   IA b = new B(new A());
   IA c = new C(new A());
   IA bc = new C(new B(new A()));
   IA cb = new B(new C(new A()));
   a.f();
   b.f();
   c.f();
   bc.f();
   cb.f();
}//main
```

15.

```
public class A extends Observable {
   private int value = 0;
   public void compute(int x) {
      value += x;
      setChanged();
      notifyObservers();
   }
   public int getValue() {
      return value;
   }
}//Observable

public class B implements Observer {
   private A theAObject;
   public B(A anAObject) {
      theAObject = anAObject;
      theAObject.addObserver(this);
   }
   public void update(Observable o, Object arg) {
      if (o instanceof A) {
         System.out.println(theAObject.getValue());
      }
   }
}//Observer
```