

Instuderingsuppgifter läsvecka 5 (generiska, samlingar, designmönster)

1.

Om vi i ett program vill deklarerar en lista av strängar bör vi skriva

```
List<String> theList = new ArrayList<String>();
```

och inte

```
ArrayList<String> theList = new ArrayList<String>();
```

Förklara varför!

2.

Om vi i ett program behöver lagra strängar i en mängd `stringSet` kan vi deklarerar `stringSet` på följande vis:

i) `Set stringSet;`

ii) `Set<String> stringSet;`

Vilken av deklARATIONerna är att föredra och varför?

3.

Antag att vi har en klass

```
public class Point {  
    ...  
}
```

för att avbilda punkter. Om man vill sortera en lista med objekt av `Point` kan man t.ex. använda sig av metoden `sort`

```
Collections.sort(listOfPoints);
```

där `listOfPoints` har typen `ArrayList<Point>`. Vad måste läggas till i klassen `Point` för att sorteringen ska gå att utföra?

4.

Klasserna i Java Collection Framework stödjer generiska typer från och med version 1.5. Skriv om nedanstående kod så att generiska typer används:

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

5.

Antag att vi har följande lista:

```
List<String> list = new LinkedList<String>();  
list.add("abc123");  
list.add("trl673");  
...
```

Iterera genom alla elementen i listan och skriv ut dem genom att använda en iterator resp. en förenklad **for**-sats.

6.

Betrakta nedanstående program.

```
public class Cigarette {
    //omitted
}

import java.util.*;
public class CigarettePack {
    private ArrayList cigs;
    public CigarettePack() {
        cigs = new ArrayList();
    }
    public void addCigarette(Cigarette c) {
        cigs.add(c);
    }
    public Cigarette getCigarette() {
        Cigarette c;
        if (cigs.size() > 0)
            c = cigs.remove(0);
        else
            c = null;
        return c;
    }
    public String toString() {
        String result = "pack of [";
        for (int i=0; i < cigs.size(); i++) {
            Cigarette c = cigs.get(i);
            result += ((i==0) ? "" : ",") + c.toString();
        }
        return result + "]";
    }
}
```

Klassen `CigarettePack` går inte att kompilera. Felutskriften blir något i stil med:

`CigarettePack.java:13: incompatible types . Found java.lang.Object , required Cigarette`

`CigarettePack.java:21: incompatible types . Found java.lang.Object , required Cigarette`

Förklara vad som är fel! Åtgärda felet!

7.

Klassen `Collections` innehåller följande metod

```
public static void sort(List<T> list)
```

där det gäller att

```
T extends Comparable<? super T>
```

Ange för var och en av deklarationerna nedan huruvida den deklarerade samlingen kan eller inte kan sorteras genom att använda metoden `sort` ovan. Motivera ditt svar! Klassen `Dog` har följande utseende:

```
public class Dog {
    ...
}
```

i. `List<String>`

ii. `Set<String>`

iii. `ArrayList<String>`

iv. `List<Dog>`

8.

Antag att du i ett program vill hålla reda på de låtar som ditt favoritband tänker spela på sin nästa konsert. Skall du spara låtarna i en lista eller i en mängd? Motiviera ditt svar! Om du behöver mer information för att fatta ditt beslut, ange vilken information du behöver.

9.

Nedan finns definitionen till klassen **Person**:

```
public class Person {  
    private String name;  
    private int birthYear;  
    ...  
    public String getName() {  
        return name;  
    }  
    public int getBirthYear() {  
        return birthYear;  
    }  
    ...  
}
```

a) Skriv en metod

```
public static String[] bornThisYear1(Person[] people, int year)
```

som tar ett fält **people** av **Person**-objekt och ett årtal **year**, och returnerar ett fält som innehåller namnen på de personer i fältet **people** som är födda år **year**.

b) Skriv en metod

```
public static ArrayList<String> bornThisYear3(ArrayList<Person> people, int year)
```

som gör samma sak som metoden i deluppgift a) förutom att parametern **person** nu är en **ArrayList** istället för ett fält.

Gör två implementeringar av metoden, dels en som använder en *förenklad for-sats* vid genomlöpnigen av listan **people**, dels en som använder en *iterator* vid genomlöpnigen av listan **people**,

c) Skriv en metod

```
public static void removeNames(ArrayList<Person> people, ArrayList<String> names)
```

som tar en lista **people** och en lista **names**, och tar bort alla objekt i listan **people** vars namn finns med i listan **names**.

10.

Förklara kortfattat vad ett designmönster är, samt vilka fördelar det finns med att använda designmönster.

11.

Ge ett exempel på där designmönstret *Singleton* kan vara användbart.

12.

Klasserna `OpenButton` och `CloseButton` är nästan identiska.

```
public class OpenButton extends JButton implements ActionListener {
    private Door door;
    public OpenButton(Door door) {
        super("Open");
        this.door = door;
        addActionListener(this);
    }
    public void actionPerformed(ActionEvent event) {
        door.open();
    }
}

public class CloseButton extends JButton implements ActionListener {
    private Door door;
    public CloseButton(Door door) {
        super("Close");
        this.door = door;
        addActionListener(this);
    }
    public void actionPerformed(ActionEvent event) {
        door.close();
    }
}
```

Gör om designen med användning av mönstret *Template method* för att eliminera duplicerad kod. Lösningen redovisas som Java-kod.

13.

Följande interface och klass är givna:

```
public interface Set<T> {
    public void add(T e);
    public void delete(T e);
    public int size();
    public boolean has(T e);
} // Set

public class NewSet<T> {
    public void insert(T e) { ... }
    public void remove(T e) { ... }
    public int size() { ... }
    public boolean contains(T e) { ... }
} // NewSet
```

Skriv en klass `NewSetAdapter` som implementerar designmönstret *Adapter* för att anpassa gränssnittet för typen `NewSet` till typen `Set`.

14.

a) Vad är syftet med designmönstret *Decorator*?

b) Betrakta nedanstående kod:

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

```
public class B extends A {
    public void f() {
        super.f();
        System.out.println("B");
    }
} //B
```

```
public class C extends A {
    public void f() {
        super.f();
        System.out.println("C");
    }
} //C
```

```
public class BC extends A {
    public void f() {
        super.f();
        System.out.println("B");
        System.out.println("C");
    }
} //BC
```

```
public class CB extends A {
    public void f() {
        super.f();
        System.out.println("C");
        System.out.println("B");
    }
} //CB
```

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

Skriv om koden genom att använda designmönstret *Decorator*.

15.

Antag att klassen *B* är intresserad av alla tillståndsförändringar som sker i *A*. När instansvariabeln *value* i *A* ändras skall *B* skriva ut det nya värdet på konsolen. Komplettera koden så att designmönstret *Observer* realiseras.

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

```
public class B {
    private A theAObject;
    public B(A anAObject) {
        theAObject = anAObject;
    }
} //B
```

Interfacet *Observer* och klassen *Observable* har följande utseende:

<<interface>> Observer
+update(o : Observable; arg: Object):void

Observable
+addObserver(o : Observer): void +deleteObserver(o :Observer): void #setChanged(): void +notifyObservers(): void