Inlämningsuppgift 2

Collection, Listor, datastrukturen ArrayList, iteratorer, generics, Comparable och Comparator.

Uppgift 1, Specifik arraybaserad datastruktur

Implementera och testa den enklaste möjliga datastrukturen. Det är klassen SimpleDataStructure som enbart kan ”lägga till”, ”ta bort” och ”leta efter” information. Informationen i datastrukturen är text strängar. Se koden nedan. Kopiera och spara den i en fil. Kompilera och exekvera. Följ instruktionerna nedan och förbättra datastrukturen.

```
import java.util.*;

public class SimpleDataStructure
{

 private String [] friends;
 private int counter;

public SimpleDataStructure()
{
  friends= new String[5];
  counter=0;
}

/*Appends the other friend name to the end of this list.*/

public boolean add(String other){
   friends[counter] = other;
   counter++;

   return true;
}


/** returns the name at the specified index*/
public String get(int index){
   return friends[index];
```

```
}

/** removes the first occurrence of the specified element in
this list if the list contains the name*/
 publicboolean remove( String name){

       for(int i=0;i<counter; i++)
       {
         if( friends[i].equals(name)){
          friends[i]=null;

          return true;

          }
       }

      return false;
}


/** prints all names in the array friends*/
public void printFriends()
{
 for(int i=0;i<friends.length; i++)
   System.out.print(friends[i]+ " " );
 System.out.println();
}

public static void main( String [] arg)
{
  SimpleDataStructure myfriends = new SimpleDataStructure();
  myfriends.add("Kalle");
  myfriends.printFriends();

}

}
```

1. Ändra i metoden printFriends() så att den aldrig skriver ut ”null” platserna.

2. Anropa metoden add() 5 gånger med fem olika kompis namn. Kör programmet. Troligen kraschar ditt program. Gör ändringar i metoden add() så att den aldrig kraschar oavsett hur många gånger den anropas. Dubblera arrayen när den är full.

3. Anropa metoden get() på följande sätt myfriends.get(23)
Troligen kraschar programmet igen.
Varför? Ändrar metoden get() så att den aldrig kraschar
Anropa metoden remove() med några kompis namn som du vet finns.
Skriv ut datastrukturen igen.
Nu plötsligt finns det massa "null" bland dina kompisar ☺.
Det vill man inte ha. Ändra metoden remove() så att den inte lämnar ”null” platsen kvar när kompisen tas bort. Dvs, metoden måste flytta alla element ett steg framme.

4. Sist skall du lägga till datastrukturen en metod som heter addSort(String namn). Metoden skall se till att arrayen *friends* hålls sorterad med namnen i alfabetisk ordning. Använd metoden compareTo() från klassen String för att jämföra 2 strängar.

Uppgift 2 -Generisk och Generiska arraybaserad datastrukturer

a) Skriv en generisk (statisk) klassmetod print som skriver ut elementen i en objektsamling:

```
public static <T> void print(Collection<T> l)
```

Om samlingen innehåller elementen a, b och c så skall utskriften ha formen [a,b,c] och [] om samlingen är tom. Placera metoden i klassen CollectionOps och skriv testfall för den i main.
Du får anta att samlingens elementtyp har en toString-metod, men *inte* att hela samlingen har en sådan. Studera först typen java.util.Collection i Java API. *Tips*: Samlingar är iterbara

b) Skriv en generisk klassmetod som vänder elementen i en lista:

```
public static <T> List<T> reverse(List<T> l)
```

Exempel: Givet följande tre listor, med innehåll inom parentes

`List<Integer> heltal` [1, 2, 3, 4, 5]
`List<Double> flyttal` [1.25, 3.14, 9.7]
`List<String> campusLindholmen` ["Saga", "Svea", "Jupiter"]
så skall metodanropen

```
CollectionOps.print(CollectionOps.reverse(heltal));
```

```
CollectionOps.print(CollectionOps.reverse(flyttal));
CollectionOps.print(CollectionOps.reverse(campusLindholmen));
```

resultera i att listornas innehåll förändras till [5,4,3,2,1], [9.7,3.14,1.25], resp.
[”Jupiter”,”Svea”,”Saga”], ingen ny lista skall skapas, men en referens till den förändrade listan skall returneras, så att man t.ex. kan anropa metoden som ovan.

OBS! Algoritmen får ej skapa temporära listor, enkla variabler räcker.

c) Ett **ArrayList**-objekt liknar en array genom att den innehåller en ändlig följd av värden. Men, arrayer är statiska datastrukturer i den meningen att en array har en viss storlek- dvs. antalet platser för värdena bestäms när den skapas och därefter inte kan ändras. En ArrayList-objekt däremot kan växa och krympa, man behöver inte ange en storlek från början.

För att förstå i djupet hur en generisk datastruktur fungerar är det bäst att du själv skriver en sådan datastruktur.

a) Din första uppgift är att implementera en klass MyArrayList (kallat så för att undvika namnkollision med Javas egen klass ArrayList). MyArrayList liknar klassen ArrayList från java-biblioteket. Skeletten för klassen finns länkad. Ladda ner filen MyArrayList.java och undersök klassen.

Titta på API för klassen ArrayList som förtydligar metoderna.

Arbetsgång:

1. Skriv ett par metoder i taget. Testa metoden genom att anropa dessa i en enkel main() program.

När du är klar med klassen och det verkar som att metoderna fungerar skall du göra en mer professionellt test.

Testa metoderna i en testklass som autogenereras av verktyget JUnit (detta visar jag på föreläsningen). Tänk vad som gäller för varje metod och vad som skall testas. Skriv först på papper med egna ord ett testprotokoll (vad som skall testas och hur).

2. Iteratorer är mycket speciella objekt. De används för att kunna iterera bland alla typer av "Collection" objekt.

Spara klassen ArrayListIterator.java i samma mapp. Undersök klassen. Gå till Java API och undersök interfacen Iterable och Iterator.

3. Komplettera klassen MyArrayList med metoden public Iterator iterator() som returnerar ett ArrayListIterator objekt.

4. Skapa en JUnitTest fil för din MyArrayList klass. Kopiera koden från min fil MyArrayListTest och klistra in den i din fil som heter troligen dessa.(Obs! Det går inte att bara ladda ner filen). Nu skall du köra din fil MyArrayListTest mot din klass. Om det fungerar då är din klass MyArrayList korrekt och du kan gå vidare. Om du får fel, undersök vad mina tester gör och ändra i koden i MyArrayList till din kod klarar av testet.

5. Dokumentera i en textfil det fel du hittade när du körde testfilen samt hur du åtgärdade dessa fel. Detta fil skall du visa när du redovisar labben.

Uppgift 3

Comparable och Comparator är 2 interface som används ofta när vi arbetar med samlingar av objekt som skall rangordnas. Undersök dessa 2 interface (gränssnitt).

1. Skriv om klassen Account så att den implementerar interfacen Comparable. Ett konto som har mindre pengar är ***minst***. Gör main () och testa genom att skapa 3 Account objekt. Lägg de i en ArrayList. Sortera de med en av metoderna från Collections.sort().

2. Bevisa att de är sorterade utan att skriva ut listan och titta med ögat.

3. Gör nu ett litet program som använder din egen MyArrayList och Comparable. Använd klassen MyArrayList i ett enkelt program som gör följande:

- skapar ett MyArrayList-objekt kallad databas

- skapar 4 Account-objekt och lägg objekten i ”databasen”.

- använd Iterator- objektet, och iterera genom lista samt anropa metoden deposit() med ett slumpat värde mellan 1000-5000

- skriv metoden *public static Comparable* ***findmin****(MyArrayList<Comparable> list)* som returnerar den minsta värde i en lista av Comparable object.

- Anropa med här metoden med listan av Account från förra uppgiften. Fungerar det ? Om ja ”Grattis” om inte gör om.

4. Ett riktigt program nu. Många applikationer behöver olika uppsättningar av data, dvs. samma data med rangordnat utifrån olika kriterier. För att detta skall göras effektivt används ”komparatorer”. Se exemplet från föreläsningen.

Nu skall du utveckla en klass **Land** som beskriver ett land. Följande data skall lagras i ett **Land- objekt:** landetsnamn, huvudstad, invånare, yta. Klassen behöver en konstruktor som instansierar klassens variabler som metoden toString() som skriver ut objektet. Andra metoder om du önskar.

För klassen Land skall du utveckla tre **Comparator** klasser. En som jämför länder utifrån deras namn (Belgien<Danmark <Sverige) alltså länderna jämförs utifrån deras alfabetiska ordning, en som jämför utifrån ländernas antal invånare och en för ländernas yta.

När detta är klart skall du använda dessa klasser i ett program som läser länder från en fil (europa, finns bifogad) och lägger de i en lista av typen ArrayList. Därefter skall programmet sortera listan (se färdiga metoder för sortering i java Collections.sort()) och skriva ut länderna en gång i alfabetisk ordning ,en gång i ordning utifrån deras yta, en gång efter deras invånare. OBS! Använd dina ”Komparator” –objekt. När du skriver ut listorna skall du använda Iterator.

b) Nu skall du ändra dit program så att den instället skriver dessa länder till olika filer. Lägg till en liten meny där användaren får välja hur ländernas skall sorteras och låt användaren välja. Producera sedan en fil som innehåller informationen sorterad.

Uppgift 4

1. Vad är skillnaden mellan en klass (datastruktur) implementerad som "generic" och en som "vanligt". När vill man ha klasser (datastrukturer) som är implementerade som " Generic type"?

2. Vad är en Interface? Vad menas med att en klass implementerar en interface. När och varför skall klasser implementera interfacen Comparable eller Comparator?

3. Vad är likheten respektive skillnaden mellan dessa två interface Collection och List. Vilka av metoderna som du har implementerad i MyArrayList tillhör interfacen List och vilka tillhör Collection.

4. I vilka fall är det lämpligt att du skall låta dina klasser implementera interfacen List? Vilka fördelar har du?

5. Hitta i java biblioteket klasser som implementerar någon av dessa interfacen. Ge exempel.

6. Vad har du lärt dig från denna uppgift.