Labb6_H20

Laboration 6 - ADT och testning

Mål

- Att kunna tillämpa grundläggande enhetstestning genom att använda basfunktioner i testramverket JUnit.
- Att kunna använda generiska typer i Java.
- Att kunna tillämpa datatypen Map för att implementera en ADT på flera sätt.
- Att använda implementationer av Map och Set effektivt genom att läsa och tolka dokumentationen i Java-API:n.

Instruktionen förutsätter att programmen körs i terminalfönster på skolans Unix-datorer genom att ni loggar in en Ubuntu-datorsal eller kopplar upp er till exempel mot **student-shell.sys.kth.se** med hjälp av egen dator och ssh (**information om installation/användning av ssh (https://kth.instructure.com/courses/20915/pages/ssh-pa-windows-10-och-mac-os-x)**). Det är absolut tillåtet att köra JUnit genom någon IDE men vi kan inte utlova handledning för det.

Här är Oracles sida med en tutorial om generiska typer: **Generic Types (https://docs.oracle.com/javase/tutorial/java/generics/types.html)** . Det räcker att läsa startsidan och något om **Bounded Type.** Se vänsterspalten i tutorial. Finns även exempel med generisk typning på bilderna till föreläsning 5. I samma föreläsning finns genomgång av Comparable som ska användas i uppgift 1.

Här finns en sida med tutorial för JUnit. (https://www.vogella.com/tutorials/JUnit/article.html)

Observera att endast Junit 4 finns på skolans datorer.

Uppgift 1 - numrerat objekt

Implementera en liten klass som heter NumberedItem där ett objekt av godtycklig typ kan lagras tillsammans med ett heltalsindex.

Krav på klassen:

- Två attribut, ett heltal (int) och ett av generiska typen T. Båda ska initieras i konstruktorn. Heltalet ska tolkas som ett index.

- Om heltalsparametern till konstruktorn (indexet) är negativ (alltså mindre än 0) så ska 0 lagras som indexattribut.
- Konstruktorns andra parameter ska vara av typen T
- Klassdeklarationen startar alltså med **class NumberedItem<T>**
- Objekt av NumberedItem ska kunna jämföras med andra objekt av samma typ, t.ex. för automatisk sortering inom Java-ramverket Collections. Endast attributet av typ int (det som säkert är ett heltal) ska användas vid jämförelsen. Låt jämförelsemetoden returnera skillnaden mellan indexen!
- toString() ska finnas som skriver ut t.ex. "Index: 42 Value: 1729".

Observera att T är namnet på en formell parameter för typen. T syftar inte på en klass som heter T.

När klassen går att kompilera utan fel, provkör följande JUnit-testklass på NumberedItem. Finns för nedladdning här: **NumberedItemTest0.java (https://kth.instructure.com/courses/20915/files/3226160/download?wrap=1)** ⇩ **(https://kth.instructure.com/courses/20915/files/3226160/download?download_frd=1)**

```
import static org.junit.Assert.*;
import org.junit.Test;

public class NumberedItemTest0 {

    @Test
    public void testCompare() {
        NumberedItem<Integer> i1 = new NumberedItem<>(-1,1);   // note the diamond syntax
        NumberedItem<Integer> i2 = new NumberedItem<>(0,17);
        NumberedItem<Integer> i3 = new NumberedItem<>(5,100);
        NumberedItem<Integer> i4 = new NumberedItem<>(10,378);
        assertEquals(0, i1.compareTo(i2));
        assertEquals(-5, i3.compareTo(i4));
        assertEquals(5, i4.compareTo(i3));
    }
}
```

Det aktuella värdet av typparametern T är Integer i testen ovan. Prova gärna själva med annat T.

När ni sparat NumberedItemTest0.java, kompilera den enligt följande (gäller på CSC:s Ubuntudatorer):

```
javac -classpath /usr/share/java/junit4.jar:.  NumberedItemTest0.java
```

Provkör nu testklassen med följande kommando:

```
java -classpath /usr/share/java/junit4.jar:.  org.junit.runner.JUnitCore  NumberedItemTest0
```

Om testen misslyckas, åtgärda NumberedItem och kör testen igen, upprepa tills testen går igenom. Använd uppåtpil för hitta samma kommando igen så ni inte behöver skriva om (eller kopiera om) den långa kommandotexten varje gång.

När testen ovan är klar, ladda ner följande lite mer omfattande test och kör den på samma sätt: **NumberedItemTest.java (https://kth.instructure.com/courses/20915/files/3226202/download?**

wrap=1) (https://kth.instructure.com/courses/20915/files/3226202/download?download_frd=1) .
Läs först testprogrammet och komplettera NumberedItem om ni ser att behövs. Iterera ändring av NumberedItem och testkörning tills testen lyckas.

Uppgift 2 - gles vektor

Klassen från uppgift 1 ska INTE användas här, även om det skulle gå och verkar lämpligt.

En klass som hanterar en gles vektor med nästan godtycklig elementtyp ska skrivas. I en vanlig array (eller vektor) används alla index från 0 upp till en övre gräns. I den aktuella uppgiften ska objekt lagras tillsammans med ett index men alla index behöver inte utnyttjas. Det ska t.ex. gå att lägga ett element på plats 3, sen ett på plats 14, därefter på plats 7 o.s.v. utan att det finns något på de icke utnyttjade indexen. I en ADT för ändamålet behöver man inte specificera hur tomma platser hanteras men implementationen här ska vara så smart att outnyttjade index inte tar någon plats alls, undantaget om man ber om en sådan representation via ett metodanrop. "Nästan godtycklig" ovan syftar på att vi ska begränsa elementtypen till att de ska gå att jämföra, de ska implementera interfacet Comparable. Detta uttrycks genom att vi anger en begränsning för typen, vi anväder en *bounded type parameter*. Här är ett gränssnitt för den glesa vektorn:

```
import java.util.*;
public interface SparseVec<E extends Comparable<E>> {
    public void add(E elem);           // add at lowest index >= 0 and not already occupied
    public void add(int pos, E elem);  // add elem at pos, if pos is occupied replace with elem, if po
s<0 use index = 0
    public int indexOf(E elem);        // find first (lowest index) occurence of elem, return its inde
x,
                                       // if not found, return -1
    public void removeAt(int pos);     // if index pos is not used, nothing happens
    public void removeElem(E elem);    // remove first occurence (lowest index) of elem
    public int size();
    public int minIndex();             // lowest index in use, if vector is empty return -1
    public int maxIndex();             // highest index in use, if vector is empty return -1
    public String toString();          // one line per element with index and value, sorted by index
    public E get(int pos);             // return null if not available
    public Object[] toArray();         // return the SparseVector as a regular array with value null a
t unused indexes
    public List<E> sortedValues();     // return a List of the values of the Vector in sorted order
}
```

Interfacet ovan finns för nedladdning här: SparseVec.java
(https://kth.instructure.com/courses/20915/files/3226201/download?wrap=1) (https://kth.instructure.com/courses/20915/files/3226201/download?download_frd=1) .

Ordet ***extends*** i "E extends Comparable<E>" ovan uttrycker en begränsning på parametern E. Den aktuella typen E måste vara en klass som implementerar Comparable (eller implementerar ett interface som ärver Comparable). ***extends*** har här en annan betydelse än den vi är vana vid.

Uppgift 2A - implementera en SparseVec som HAR en TreeMap

Skriv en implementation av gränssnittet ovan. Implementationen (klassen ni skriver) ska fortfarande ha en typparameter så att godtyckliga objekt (som är Comparable) kan lagras. För att lagra objekt och deras index, sorterat på index, låt klassen ha ett objekt av en implementation av Java-interfacet Map<K,V>, förslagsvis klassen TreeMap<K,V>. En hel del av den funktionalitet som krävs i SparseVec finns redan i TreeMap så läs dokumentationen i API:n av TreeMap<K,V> noga så att ni inte gör en massa jobb i onödan. Läs även dokumentationen för datatypen Set (särskilt SortedSet) då mängden nycklar hanteras som en sådan. Alla funktioner i SparseVec kan implementeras med bara en TreeMap som attribut. Några av metoderna kommer att kräva lokala variabler men det är en annan sak.

Tips för och krav på implementationen:

- **Syntaxtips:** Inled klassdeklarationen (efter importsats(er)) så här:
  class XXXXX <E extends Comparable<E>> implements SparseVec<E>
  där XXXXX är ert eget väl valda namn på klassen.
- **Tips:** Följande metoder kan implementeras på en eller två rader vardera genom att direkt utnyttja funktionalitet i TreeMap eller kombinera andra metoder i SparseVec: size(), minIndex(), maxIndex(), removeAt(pos), get(pos), removeElem(elem), add(pos,elem)
- Följande metoder kräver lite mer jobb: add(elem), indexOf(elem), toString(), toArray(), sortedValues()
- **Krav:** För toString(), använd klassen StringBuilder för att bygga upp texten, inte upprepade + mellan textvärden. Utskriften ska ha en rad per element där index och värde anges.

Skriv en liten testklass till klassen ovan.

Titta på förlagan från uppgift 1 för att få ett skelett till testklassen. Försök skriva och köra testklassen växelvis med vektorklassen.

Tips för och krav på testklass och testning:

- **Tips:** För testen måste ni välja ett värde på aktuell typparameter, alltså en klass som implementerar Comparable. Några enkla val för testning är Integer eller String.
- **Krav:** För en tom SparseVec, testa size(), minIndex(), maxIndex() som ju alla ska ge heltalsvärden. Testa get(k), där k är ett godtyckligt int-värde. Anropa toArray() och sortedValues() på tom vektor och gör lämpliga tester på resultaten.
- **Krav:** Lägg till element på specificerade positioner, några på samma index och testa size() efter varje insättning. Testa värdena på minIndex() och maxIndex().
- **Krav:** Lägg till element på ospecificerad position. Testa size() och get() så att det stämmer med förväntade värden. Kom ihåg att testa get() med ett index som inte används. Den ska gå ge null.
- **Krav:** Testa båda varianterna av remove och gör tester som visar att de fungerar.
- **Krav:** I SparseVec-implementationens main-metod, skapa en vektor, lägg in minst fem värden och testa toString(), toArray() och sortedValues() genom att anropa dem och skriva ut resultaten. Använd alltså inte JUnit och assert för dessa tester.
- **Tips:** Det räcker att använda assertEquals() och assertNull() i testerna. Men använd gärna fler varianter av assert om det passar. Gör gärna en testmetod för varje grupp av tester, t.ex.

testEmpty, testIndex, testRemove. Men det är OK att ha en enda metod.

Uppgift 2B - Implementera en SparseVec som ÄR en TreeMap

Gör en ny klass genom att ta en kopia av klassen från 2A. Låt klassen ärva från TreeMap och implementera SparseVec. Objekt av klassen kommer nu att VARA TreeMap. Tag bort TreeMap-attributet från klassen. Ytterligare ändringar bör göras. Beroende på hur ni har implementerat klassen i 2A kan ett litet (men lärorikt) fel uppstå.

Kopiera också testklassen från 2A och genomför testerna.

Att redovisa

1. Klassen NumberedItem.java samt körning av givet testfall NumberedItemTest.java
2. Klassen från 2A med tillhörande testklass enligt krav ovan och exekvering av testen.
3. Klassen från 2B med tillhörande testklass enligt krav ovan och exekvering av testen.