Mittuniversitetet
MID SWEDEN UNIVERSITY

Lektion 5 – Collections

Java för C++-programmerare

Lektionen behandlar Javas samlingsklasser och iteratorer.

Att läsa: Skansholm kap. 10.10 och 17

Att jämföra objekt

I många sammanhang ställs man inför problemet att jämföra objekt för att konstatera om två objekt är lika eller inte. Ibland vill man också kunna avgöra om ett objekt är ”större än” eller ”mindre än” ett annat objekt av samma typ.
I samband med olika kontainerklasser är detta vanligt förekommande. I C++ kan vi hantera problemet genom att överlagra operatorerna ==, < och > sedan använda olika STL-klasser och generiska algoritmer. I Java finns inte operatoröverlagring utan här används istället:

- omdefiniering av metoden equals(Object) som är ärvd från Object. I sin ursprungliga version är det objekt-referenser som jämförs, dvs. ett objekt är endast lika med sig själv. equals returnerar en boolean.

- Interfacet Comparable. En klass blir ”comparable” genom att implementera interfacet Comparable.

Observera att jämförelse med operatorn == alltid jämför objektreferenser.

Interfacet Comparable<T> deklarerar en metod:

```
int compareTo(T object);
```

Vi noterar att i deklarationen av interfacet Comparable återfinns samma typ av syntax som används för mallar i C++ . I Java kallas typparametriserade klasser och interface för *generics*. Detta infördes från och med version 1.5. Numera är de flesta kontainerklasser och många interface i Java generiska.

Om vi exekverar

```
int result = obj1.compareTo(obj2);
```

så måste compareTo vara implementerad för den aktuella klassen så att resultatet blir

- 0 om obj1 och obj2 är ”lika”
- mindre än 0 om obj1 är ”mindre” än obj2
- större än 0 om obj1 är ”större” än obj2.

En klass som implementerar interfacet Comparable sägs ha *naturligt jämförbara* objekt.

Som exempel tar vi klassen Student.

Exempel

```
      class Student implements Comparable<Student> {

      private String iName;
      private String iPnr;

      public Student(String aName, String aPnr) {
            this.iName=aName;
            this.iPnr=aPnr;
      }

      public String getPnr(){return iPnr;}


      public String toString() {
            return iName+", "+iPnr;
      }

      public boolean equals(Object o) {
            Student s=(Student)o;
            boolean b=iPnr.equals(s.getPnr());
            return (b);
      }

      public int compareTo(Student s) {
            int i=iPnr.compareTo(s.getPnr());
            return (i);
      }
}
```

Personnumret får alltså vara grunden för jämförelsen.
Vi ser att eftersom klassen implementerar interfacet Comparable<Student> så blir parametern till metoden compareTo av typen Student.

I Javas variant av generics finns inget som motsvarar funktionsmallar i C++. Detta ser vi t.ex. metoden equals som i alla klasser har Object som parametertyp. Konsekvensen blir att varje klass som omdefinierar equals måste gör ett typecast från Object till den aktuella klassen:

```
Student s=(Student)o;
```

Varje typecast kan vara en källa till misstag och buggar och vi måste vara noggranna när vi gör ett sådant. Före införandet av generics i Java 1.5 var man tvungen att göra denna typ av typecast även i alla kontainerklasser och interface. Detta mindre önskvärda sätt att lösa problemet kan man fortfarande se i äldre kod.

Vi testar studentklassen genom att göra parvisa jämförelser mellan några studenter:

```
class StudentTest {

  public void compareAndPrint(Student stud1, Student stud2)
  {
    if(stud1.compareTo(stud2)==0)
      System.out.println(stud1 + " lika med " + stud2);
    else if(stud1.compareTo(stud2)<0)
      System.out.println(stud1 + " mindre än " + stud2);
    else
      System.out.println(stud1 + " större än " + stud2);
  }

  public static void main (String args[]) {

      Student s1 = new Student("Pelle Plugg","890413-8213");
      Student s2 = new Student("Nisse Nörd","900313-8117");
      Student s3 = new Student("Sandra Skarp","870202-8243");
      Student s4 = new Student("Pelle Plugg","890413-8213");

      StudentTest test = new StudentTest();
      test.compareAndPrint(s1,s2);
      test.compareAndPrint(s1,s3);
      test.compareAndPrint(s1,s4);
      test.compareAndPrint(s2,s3);
      test.compareAndPrint(s2,s4);
      test.compareAndPrint(s3,s4);
  }
}
```

Utskrift:

```
Pelle Plugg, 890413-8213 mindre än Nisse Nörd, 900313-8117
Pelle Plugg, 890413-8213 större än Sandra Skarp, 870202-8243
Pelle Plugg, 890413-8213 lika med Pelle Plugg, 890413-8213
Nisse Nörd, 900313-8117 större än Sandra Skarp, 870202-8243
Nisse Nörd, 900313-8117 större än Pelle Plugg, 890413-8213
Sandra Skarp, 870202-8243 mindre än Pelle Plugg, 890413-8213
```

I kap 10.10 ges också exempel på *extern jämförelse* av objekt som inte är naturligt jämförbara och därför inte implementerar Comparable. En klass för extern jämförelse implementerar interfacet Comparator<T> som deklarerar metoden int compare(T object1, T object2) . För att jämföra två object av en viss klass kan vi alltså använda ett speciellt anpassat Comparator-objekt som tar de två objekten som ska jämföras som argument till sin compare-metod.

Collections (samlingar)

Interfacet java.util.Collection<E>

Alla kontainerklasser i Java implementerar det generiska gränssnittet java.util.Collection<E>. Där deklareras grundläggande metoder för alla kontainerklasser (Skansholm 612).

Metoder i java.util.Collection:

`add( e)`	sätter in e i samlingen
`addAll(s)`	sätter in alla element i samling s i den aktuella samlingen
`clear()`	tar bort alla element ur samlingen
`contains(o)`	finns objektet o i samlingen?
`conatinsAll(s)`	finns alla element i samlingen s i den aktuella samlingen?
`equals(s)`	är samlingen s lika med den aktuella samlingen? Obs, definition av ”lika med” varierar!
`isEmpty()`	saknar element?
`iterator()`	returnerar en iterator av klassen `Iterator<T>` för iteration genom samlingen.
`remove(o)`	tar bort objektet 0 från samlingen
`removeAll(s)`	tar bort alla element i samlingen s från den aktuella samlingen.
`retianAll(s)`	tar bort alla element *utom* de som finns i samlingen s från den aktuella samlingen.
`size()`	returnerar antalet element i samlingen
`toArray()`	returnerar en array med alla objekt i den aktuella samlingen.

Från Collection finns olika sub-interface deriverade:
List, Set och Queue som utöver metoderna i Collection deklarerar egna metoder.

Interfacet java.util.List<E>

Intefacet List implementeras av generiska kontainerklasser som är alla är linjära och ordnade till sin struktur, dvs. varje element utom det första och sista har en unik föregångare och efterföljare.

Konkreta kontainerklasser som implementerar List<E> är

- LinkedList<E>
  Dubbellänkad lista, ingen direkt access till interna element. Bäst på att lägga till/tabort element i början/slutet. Sämre när det gäller att modifiera element i det inre av listan

- ArrayList<E>
- Vector<E>
  Både ArrayList och Vector har ”random access” till sina element. Ett element kan alltså nås direkt via sitt index.
  Båda är bra på att modifiera element inne i kontainern men ineffektiva när de gäller att sätta in nya element t.ex. först eller inne i listan.
  Skillnaden mellan ArrayList och Vector är främst att Vector har *synkroniserade* operationer vilket tillåter flera olika trådar att parallellt utför operationer på listan, medan ArrayList är osynkroniserad.

    - Stack<E> (deriverad från Vector<E>)
      ”Last-In-First-Out”- klass implementerar de vanliga LIFO-operationerna

Metoder i java.util.**List** (utöver metoderna i Collection):

`add(k,e)`	lägger in e på plats k
`addAll(k,s)`	lägger in hela samlingen s på plats k
`get(k)`	returnerar elementet på plats k
`indexOf(o)`	returnerar index för o
`lastIndexOf(o)`	returnerar index för o, söker bakifrån
`listIterator()`	returnerar en listIterator till element på plats 0
`listIterator(k)`	returnerar en listIterator till element på plats k
`remove(k)`	tar bort element på plats k
`set(k,e)`	ersätter element på plats k med e
`subList(i,j)`	returnerar en dellista med elementen fr.o.m. i t.o.m. j-1

Alla konkreta Collection-klasser definierar naturligtvis sina egna metoder utöver de som deklareras i de olika interface som de implementerar.

T.ex. definierar klassen LinkedList metoderna

`addFirst(o)`	lägger in o först i listan
`addLast(o)`	lägger in o sist i listan
`getFirst()`	returnerar första objektet i listan
`getLast()`	returnerar sista objektet i listan
`removeFirst()`	tar bort första objektet i listan
`removeLast()`	tar bort sista elementet i listan
`clone()`	returnerar en kopia av listan (med referenserna till objekten i listan kopierade, inte själva objekten)

Iteratorer

Begreppet iterator är välkänt från C++. Alla kontainerklasser i Java har metoden iterator() som returnerar en iterator till den aktuella samlingen. Semantiken för iteratorerna för de olika kontainerklasserna skiljer sig naturligtvis åt beroende på den interna strukturen i kontainerklassen. Alla kontainerklasser har metoden iterator() som returnerar en referens till ett objekt av typen Iterator<E> där E är den aktuella elementypen.

Alla iteratorer implementerar interfacet java.util.iterator med följande metoder:

`next()`	ger nästa element i samlingen vid iteration framlänges
`hasNext()`	finns det ett element till i samlingen?
`remove()`	tar bort elementet som iteratorn just nu refererar till ur samlingen

Klassen ListIterator definierar dessutom

`previous()`	ger nästa element i samlingen vid iteration baklänges
`hasPrevious()`	finns det ett element till (baklänges) i samlingen?
`nextIndex()`	ger index för nästa element (framlänges)
`previousIndex()`	ger index för nästa element (baklänges)
`add(e)`	skjuter in e i aktuell position i listan
`set(e)`	ersätter elementet i den aktuella positionen med e

Exempel - utskrift av alla element i en Vector<Integer>

```
Vector<Integer>  vec = new Vector<Integer>();
... // Fill it
Iterator<Integer> it =  vec.iterator(); // Get an iterator to first
element

while(it.hasNext()) {  // For every element…

      Integer value = it.next(); // ...get next element
      System.out.println(value+" ");
}
```

Eftersom en Vector är en indexerad samling så kan vi skriva ut värdena i vec genom att loopa igenom dem i en vanlig for-sats och utnyttja indexeringen

```
for(int i=0; i<vec.size(); ++i)
      System.out.println(vec.elementAt(i)+" ");
```

Operatorn [ ] finns inte definierad för Vector, istället använder vi metoden elementAt(index).

Samma sak kan också uttryckas med Javas förenklade for-sats:

```
for(Integer i:vec)
      System.out.println(i+" ");
```

Interfacet java.util.Set<E>

Interfacet java.util.Set definerar egenskaper för mängder i matematisk mening. Det innebär främst att två element i en mängd inte kan ha samma värde. Dessutom finns det inget krav på att en mängd skall vara ordnad, dvs. det finns ingen bestämd föregångare/efterträdare till ett element i en mängd. Inga nya metoder deklareras i interfacet med semantiken för metoden add(e) är ändrad så en dubblett till ett element i mängden inte kan läggas till.

Interfacet java.util.**SortedSet** <E> är ett sub-interface till Set<E> som deklarerar metoder för sorterade mängder. En klass som implementerar detta interface är **TreeSet<E>**
Förutsättningen för en sorterad mängd är att elementen är naturligt jämförbara, dvs. att de implementerar interfacet Comparable genom metoden compareTo. Om elementen inte implementerar compareTo måste ett lämpligt Comparator-objekt skickas som parameter till konstruktorn. Metoder som tillförs i interface SortedSet<E> är

`first()`	returnerar mängdens minsta element
`last()`	returnerar mängdens största element
`headSet(toE)`	returnerar en sorterad delmängd med alla element < toE
`tailSet(fromE)`	returnerar en sorterad delmängd med alla element >= fromE
`subSet(fromE,toE)`	returnerar unionen av headSet(toE) och tailSet(fromE)
`comparator()`	returnerar den comparator som används

Klasser som direkt inplementerar Set<E> är

- HashSet<E>
  - LinkedHashSet<E> deriverad från HashSet<E>
    Dessa två skiljer sig främst åt när det gäller prestanda beroende på implementationen. HashSet använder en sk. Hash-tabell medan LinkedHashSet dessutom har en länkad lista som behåller inläggningsordningen för

elementen.Om man ofta ska genomlöpa alla elementen är en LinkedHashSet effektivast medan en HashSet är snabbare på att söka upp, lägga till och ta bort element.

- EnumSet<E>
  Designad för att skapa mängder där elementen är av uppräkningstyp (enum). Använder en bit-vektor för att effektivt mappa ett element mot en bit med värdet 1 om elementet ingår i mängden annars 0.

Avbildningstabeller (maps)

En avbildningstabell är en associativ kontainer där en unik söknyckel (”key”) associeras med, eller avbildas på, ett visst värde (”value”). Dessa ”key-value”-par är kärnan i en avbildningsmap. Andra vanliga namn på denna typ av struktur är *map*, *associativ array* och *dictionary*.

Interfacet java.util.Map<E>

java.util.**Map<E>** deklarerar de metoder som är gemensamma för all avbildningstabeller i Java:

`put(key,value)`	lägger in ett nytt key/value-par, ev . ersätter ett gammalt
`putAll(tab)`	lägger in avbildningar från en annan avbildningstabell
`remove(key)`	tar bort avbildningen med key som nyckel
`clear()`	tar bort alla avbildningar från tabelle
`get(key)`	returnerar värdet för key, eller null
`containsKey(key)`	finns nyckeln key i tabellen
`containsValue(v)`	finns värdet v i tabellen?
`isEmpty()`	är tabellen tom?
`size()`	returnerar antalet avbildningar i tabellen
`equals(tab)`	har tabellen samma avbilningar som tab?
`keySet()`	returnerar en mängd med alla nycklar
`values()`	returnerar en samling med alla värden
`entrySet()`	returnerar en mängd med alla avbildningar
`Map.Entry<K,V>`	generisk klass som representerar en avbildning

Likheten med klassmallen `map` i C++ är uppenbar. Klassen `Map.Entry<K,V>` motsvarar

```
template <class T1, class T2>
class pair ;
```

Den implementation som mest överensstämmer med `map` i C++ är den generiska klassen TreeMap som även implementerar sub-interfacet java.util.**SortedMap.**

Metoder som tillförs i interfacet SortedMap:

`firstKey()`	returnerar minsta nyckel
`lastKey)`	returnerar största nyckel
`headMap(toK)`	returnerar en tabell med alla avbildningar där nyckeln < toK
`tailMap(fromK)`	returnerar en tabell med alla avbildningar där nyckeln >= fromK
`subMap(fromK,toK)`	returnerar det kombinerade resultatet av `headMap(toK)` och `tailMap(fromK)`.
`comparator()`	returnerar den comparator som används

Interfacet Map implementeras även av andra klasser där key/value-paren inte lagras i sorteringsordning, t.ex. HashMap.

Om en Map ska användas parallellt från flera trådar krävs synkronisering av anropen. Detta finns implementerat i klassen ConcurrentHashMap som implementerar sub-interfacet ConcurrentMap, deriverat från Map.

Klassen java.util.Collection**s**

Java har i senare versioner sökt att implementera samma typ av funktionalitet som återfinns i de generiska algoritmerna i C++ standardbibliotek. Resultatet har blivit en klass, Collections, med uteslutande klassmetoder (static) som ska användas tillsammans de olika kontainertyperna.
Här hittar vi bl.a. klassmetoder för

- sortering: `sort`
- sökning: `binarySearch`
- `max` och `min`
- reversering och rotering: `reverse`, `rotate`
- slumpmassig omkastning: `shuffle`
- fylla på lista: `fill`
- platsbyte: `swap`

samt mycket annat.