Läsanvisningar

Börja med att läsa kapitlen i kursboken, därefter resurserna på Internet och till sist själva lektionsmaterialet.

Kursboken (5:e upplagan): Kapitel 17 och 10.10
Kursboken (6-7:e upplagan): Kapitel 17 och 10.11
Kursboken (8:e upplagan): Kapitel 17 (ej 17.8 och framåt) och 10.11

Internet: The Java Tutorial, Trail: Learning the Java Language - Generics
(http://docs.oracle.com/javase/tutorial/java/generics/index.html)

Internet: The Java Tutorial, Trail: Collections - Introduction to Collections
(http://docs.oracle.com/javase/tutorial/collections/intro/index.html)

Internet: The Java Tutorial, Trail: Collections - Interfaces
(http://docs.oracle.com/javase/tutorial/collections/interfaces/index.html)

Internet: The Java Tutorial, Trail: Collections - Implementations
(http://docs.oracle.com/javase/tutorial/collections/implementations/index.html)

Internet: The Java Tutorial, Trail: Collections - Algorithms
(http://docs.oracle.com/javase/tutorial/collections/algorithms/index.html)

Generiska klasser

Generiska klasser är klasser vars typ egentligen är odefinierad. Om du tittar på exemplet i kapitel 17.1.1 i kursboken ser du hur en klass deklarerar två strängar för att lagra information. Men om man nu skulle vilja göra samma sak fast med heltal istället för strängar, måste man skriva en ny klass med ändrad typ på instansvariablerna. Samma gäller om man skulle vilja ändra till double eller boolean och så vidare. För att lösa detta på ett enklare sätt har man från version 1.5 av Java infört något som heter generiska klasser. Exemplet på sida 640 i kursboken (600 i upplaga 5) visar hur detta kan se ut.

```
public class Pair <T1, T2> {
  private T1 fst;
  private T2 snd;

  public Pair() { }

  public Pair(T1 x1, T2 x2) {
    fst = x1;
    snd = x2;
  }

  public void setFirst(T1 x) {
    fst = x;
  }

  public void setSecond(T2 x) {
    snd = x;
  }
```

```
    public T1 first() {
      return fst;
    }

    public T2 second() {
      return snd;
    }
  }
```

Här kan T1 och T2 vara helt olika typer. Vilken typ det blir vid bestäms i samband med deklarationen.

```
Pair<String, Integer> p1 = new Pair<String, Integer>();
// vi skulle kunna göra något i stil med
p1.setFirst(new String("Test"));
p1.setSecond(new Integer(20));
```

Genomgående i kursboken och i referenser på nätet gällande samlingsklasserna refererar de till <E>, där E är ett element i en lista.

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 samlingsklasser är detta vanligt förekommande.

Att använda operatorn == för att jämföra objekt mot varandra är fel eftersom == alltid jämför objektreferenser. Det vill säga om de båda refererar till samma objekt. I Java får vi istället:

- överskugga metoden equals(Object) som ärvs från klassen Object. I sin ursprungliga version är det objekt-referenser som jämförs, det vill säga ett objekt är endast lika med sig själv. Metoden equals returnerar en boolean som är true om de båda objekten är lika.

eller

- göra klassen ”jämförbar” genom att implementera gränssnittet Comparable.

Interfacet Comparable<T> innehåller en metod som måste finnas med i de klasser som väljer att implementera gränssnittet:

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

Om vi exekverar

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

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.

```
public class Student implements Comparable<Student> {
  private String name;
  private String pnr;

  public Student(String name, String pnr) {
    this.name = name;
    this.pnr = pnr;
  }

  public String getPnr() {return pnr;}

  public String toString() {
    return name + ", " + pnr;
  }

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

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

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

Metoden equals i klassen Object har funnit sedan första versionen av Java och därmed innan generics infördes. Den måste därför ha 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 samlingsklasser och interface. Detta mindre önskvärda sätt att lösa problemet kan man fortfarande se i äldre kod.

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

```
public 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 kapitel 10.11.2 i kursboken (10.10.2 i upplaga 5) ges också exempel på extern jämförelse av objekt som inte är naturligt jämförbara och därför inte implementerar Comparable. En klass för extern jämförelse implementerar gränssnittet Comparator<T> som deklarerar metoden int compare(T object1, T object2). För att jämföra två objekt 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. Vill vi alltså utföra en jämförelse som behöver specialanpassas är Comparator vad vi behöver använda. Låt säga att vi vill jämföra studenter utifrån vilken månad de är födda. Vi kan då skriva en klass som implementerar gränssnittet Comparator<Student>.

```
public class CompareStudentMonth implements Comparator<Student> {
  public int compare(Student s1, Student s2) {
    // get month from pnr
    int month1 = Integer.parseInt(s1.getPnr().substring(2, 4));
    int month2 = Integer.parseInt(s2.getPnr().substring(2, 4));

    if (month1 < month2) {
      return -1;
    }
    else if (month1 > month2) {
      return 1;
    }
    return 0;
  }
}
```

I vår testklass lägger vi till en överlagrad variant av compareAndPrint som tar ett CompareStudentMonth-objekt som argument med vilken jämförelsen sen utförs med.

```
public void compareAndPrint(CompareStudentMonth csm, Student s1, Student s2)
{
  if(csm.compare(s1, s2) == 0) {
```

```
        System.out.println(s1 + " lika med " + s2);
    }
    else if(csm.compare(s1, s2) < 0) {
        System.out.println(s1 + " mindre än " + s2);
    }
    else {
        System.out.println(s1 + " större än " + s2);
    }
}
```

I main-metoden skapar vi ett objekt av CompareStudentMonth.

```
…
Student s4 = new Student("Pelle Plugg","890413-8213");
StudentTest test = new StudentTest();
CompareStudentMonth csm = new CompareStudentMonth();
test.compareAndPrint(csm, s1,s2);
…
```

Varpå utskriften blir.

```
Pelle Plugg, 890413-8213 större 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 mindre än Pelle Plugg, 890413-8213
Sandra Skarp, 870202-8243 mindre än Pelle Plugg, 890413-8213
```

Samlingsklasser – gränssnittet Collection

Med samlingsklasser menas de klasser som rör listor och mängder. Listor är information som är grupperat i någon typ av ordning och mängder är information som enbart finns en gång. Exempel på listor kan vara en telefonkatalog, medan mängder kan vara en grupp personer som alla är unika.

Det gränssnitt som beskriver listor och mängder är java.util.Collection<E>. Där deklareras grundläggande metoder för alla samlingsklasser. Se sida 652 för en lista av metoder i Collection (sida 612 i upplaga 5).

Samlingsklasserna delas in i fyra undergrupper: List, Set och Queue som utöver metoderna i Collection deklarerar egna metoder. Figur 17.1 visar denna uppdelning. Gränssnitten List och Set implementerar även gränssnittet Serializable, vilket leder till att hela listor eller mängder går att serialisera. I Java innebär serialisering att skriva/läsa data till/från en ström, vanligtvis en fil på användarens dator. Det är med andra ord en relativt enkel sak att ladda/spara stora mängder med information i Java. Mer om detta i lektion 8.

Iteratorer

För gå igenom alla element i samlingsklass kan man till exempel använda en for-loop och starta på element 0 och gå till sista elementet i listan (size-1). Men i vissa samlingar går det inte att använda en for-loop eller så är det kanske inte det mest effektiva sättet att gå igenom elementen. I stället finns det något som kallas iteratorer. I gränssnittet Collection<E> finns en metod, iterator(), som returnerar en iterator för just den samlingen. Eftersom metoden deklarerats i gränssnittet Collection>E> har alla samlingsklasser i Java har metoden iterator(). Typen som returneras av

metoden blir Iterator<E> där E är den aktuella elementypen.

Alla iteratorer implementerar interfacet java.util.iterator med metoderna next, som ger nästa element i samlingen, hasNext, som returnerar true så länge det finns fler element, och remove, som tar bort elementet iteratorn just nu refererar till. Förutom denna ”vanliga” iterator, finns det även mer specialanpassade iteratorer. ListIterator är en av dessa. Den har ett antal fler metoder för att bland annat kunna gå baklänges i samlingen och lägga till element på vissa positioner.

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 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)+" ");
}
```

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

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

java.util.List<E>

Gränssnittet List implementeras av generiska samlingsklasser som är alla är linjära och ordnade till sin struktur, det vill säga varje element utom det första och sista har en unik föregångare och efterföljare.

Konkreta samlingsklasser som implementerar List<E> är

- LinkedList<E>
  Dubbellänkad lista, ingen direkt access till interna element. Bäst på att lägga till och ta bort 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 samlingen men ineffektiva när de gäller att sätta in nya element till exempel 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. Du behöver inte förstå detta nu då trådar är något som kommer i Java III.
- Stack<E> (ärver från Vector<E>)

Stack är en subklass till Vector och använder sig av ett kö-system. Ett så kallat LIFO-system (last in first out). Man kan likna det med en tallrikshög. Den tallrik man lade högst upp på högen är den man tar först ur högen. Ett användningsområde skulle kunna vara ett felhanteringssystem där alla fel läggs på stacken och sedan tar man upp ett och ett och behandlar dessa, eller vad man nu väljer att göra.

I kursen är det i första hand denna typ av samlingsklasser som kommer att användas och närmare bestämt Vecotr och ArrayList. Du rekommenderas därför att studera dessa klasser extra noga.

java.util.Set<E>

Gränssnittet java.util.Set definierar 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. Betydelsen av metoden add(e) är ändrad så en dubblett till ett element i mängden inte kan läggas till.

Gränssnittet java.util.SortedSet <E> är en subklass till Set<E> och deklarerar metoder för sorterade mängder. En klass som implementerar detta gränssnitt är TreeSet<E>. Förutsättningen för en sorterad mängd är att elementen är naturligt jämförbara, dvs. att de implementerar gränssnittet Comparable genom metoden compareTo. Om elementen inte implementerar compareTo måste ett lämpligt Comparator-objekt skickas som parameter till konstruktorn.

Klasser som direkt implementerar Set<E> är

- HashSet<E>
- LinkedHashSet<E> ärver från HashSet<E>
  Dessa två skiljer sig främst åt när det gäller prestanda beroende på implementationen. HashSet använder en så kallad 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.

Kursen kommer inte att ha så mycket att göra med mängder så det räcker därför att du skummar igenom detta. Viktig att känna till att mängder finns, men du behöver inte kunna varenda klass och metod.

Avbildningstabeller (maps)

En avbildningstabell är en associativ samling 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. Detta är också en del vi inte kommer att beröra speciellt mycket. Vi kommer inte att begära att ni ska kunna alla olika varianter av dessa, men det är alltid bra att känna till att de existerar.

java.util.Collections

Förutom gränssnittet Collection finns även en klass som heter Collections, notera s'et på slutet. Denna klass har en hel del generiska (icke typ bundna) klassmetoder (static) som ska användas tillsammans med de olika samlingsklasserna. 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.

För att sort ska fungera så måste den typ som sorteras implementera gränssnittet Comparable alternativt att en separat Comparator-klass tillhandahålls. Det finns ett litet stycke om det i kapitel 10, om ni inte läst det, gör så nu. På sida 657 i kursboken (653 i upplaga 6-7 och sida 613 i upplaga 5) finns en lista över olika metoder som Collections tillhandahåller. Dessa kan vara bra att tittar över. Framför allt att sortera med Collections är smidigt.