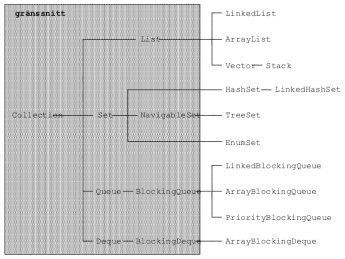
Containerklasser



Klassen Set (Mängder)

I en mängd får varje element finnas en - och endast en - gång.

Comparatorn används för att jämföra icke naturligt jämförbara typer med en extern jämförare.

TreeSet är navigerbara/sortera mängder.

Exempel
Plockar ut alla ord som förekommer i en given text.

Programmet använder sig av TreeSet, varför utskriften blir sorterad.

`NavigableSet<String> m = new TreeSet<String>(co);` använder sig av collatorn tidigare i koden (använd alltid PRIMARY för att få bokstavsordning) för att sortera.

Iterator

En iterator är en klass vilken är deklarerad inuti en samling. Den har koll på allt hokus pokus som försiggår inuti samlingen som skulle varit dolt för oss annars. Precis som scanner kan vi använda iteratorn för att leta i listor och liknande.

Iteratorns typ ska vara av samma som typen på containern.

I en loop behövs inte det tredje argumentet, varför det är tomt efter sista `;`

Avbildningstabeller

Har nycklar vilka är kopplade till värden. Typerna på dem är oberoende av varandra och är ett smidigt sätt att indexera saker. Det får inte finnas flera `Keys` med samma värde men självklart får det finnas flera `Value` av samma sort.

Det får bara finnas en Nisse, men många kan ha fått 3a på tentan.

Exempel
Den första tabellen görs i en TreeMap för vi bryr oss om ordningen.
Den andra tabellen görs i en HashMap för det går fortare.

De generiska typerna måste vara referenstyper, alltså inte enkla typer, varför vi måste skriva `new Integer(18).` Det hade kanske gått att skriva `18` och förlita sig på boxing, men Skansholm ville vara tydlig.

Skapa samlingar

```
Set<typ> h = new HashSet<>();
... // placera element i mängden h
List<typ> l = new LinkedList<>(h); // kopia av h

l.add(new String("en text"));
```

Mängder

```
Set<String> s0 = Collections.emptySet();
Set<String> s1 = Collections.singleton(new String("Ensam"));
```

```
TreeSet<E>()
TreeSet<E>(Comparator<? super E> comp)
TreeSet<E>(Collection<? extends E> coll)
TreeSet<E>(SortedSet<E> ss)
```

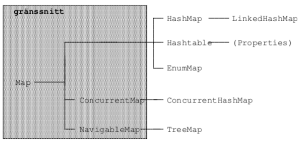
```
import java.util.*;
import java.text.*;
import java.io.*;
public class TextAnalys {
  public static void main(String[] arg) throws IOException {
    Collator co = Collator.getInstance();  // jämför texter
    co.setStrength(Collator.PRIMARY);
    NavigableSet<String> m = new TreeSet<String>(co);

    // koppla en scanner till filen
    Scanner sc = new Scanner(new File(arg[0]));

    // läs ett ord i taget och addera till mängden
    while(sc.hasNext())
      m.add(sc.next());

    // skriv ut alla orden
    for (String ord : m)
      System.out.println(ord);
  }
}
```

Iteratorer

```
List<Integer> li = new LinkedList<>();
for (Iterator<Integer> it=li.iterator(); it.hasNext(); )
  if (it.next() == 0)
    it.remove();


for (Integer i : li)
  System.out.println(i);
```

Samma som:

```
for (Iterator<Integer> it=li.iterator(); it.hasNext();) {
  Integer i = it.next();
  System.out.println(i);
}
```




```
Map<key,value> tab1 = new TreeMap<>();
... // lägg in avbildningar i tab1
Map<key,value> tab2 = new HashMap<>(tab1);
```

```
Map<String,Integer> pertab  = new TreeMap<>();
Map<String,Motorfordon> reg = new HashMap<>();
pertab.put("David", new Integer(18));
Motorfordon f = reg.get("ABC123");
```

```
TreeMap()
TreeMap(Comparator<? super K> comp)
TreeMap(Map<? extends K,? extends V> m)
TreeMap(SortedMap<K,? extends V> sm)
```

TextAnalys2.java

Tillskillnad från förra gången använder vi en `TreeMap` för att kunna nyttja tabellböset.

I while-loopen söker vi igenom inputen och lägger varje ord i en temp `String` för att sedan kolla upp om denna string finns i tabellen.

Vi ökar förekomsten av ordet med 1 och nya ord läggs till i tabellen.

Märk att vi använder `Integer` istället för `int` ty `tab.get(ord)` kan returnera *`null`*.

För att kunna skriva ut allting skapar vi en sorterad mängd.

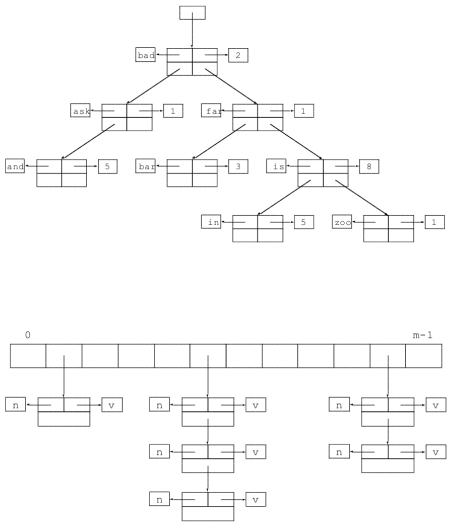
```
import java.util.*;
import java.text.*;
import java.io.*;

public class TextAnalys2  {
  public static void main(String[] arg) throws IOException {
    Collator co = Collator.getInstance();  // jämför texter
    co.setStrength(Collator.PRIMARY);
    // skapa avbildningstabellen
    NavigableMap<String,Integer> tab = new TreeMap<>(co);
    // koppla en scanner till filen
    Scanner sc = new Scanner(new File(arg[0]));
    // läs ett ord i taget och addera till mängden
    while(sc.hasNext()) {
      String ord = sc.next();
      Integer antal = tab.get(ord);  // slå upp ordet i tabellen
      if (antal == null)
        antal = new Integer(0);      // ordet fanns inte tidigare
      // öka antalet förekomster av ordet
      tab.put(ord, new Integer(antal+1));
    }
    // skapa en sorterad mängd med alla par av ord och antal
    Set<Map.Entry<String,Integer>> m = tab.entrySet();
    // skriv ut alla paren
    for (Map.Entry<String,Integer> a : m)
      System.out.println(a.getKey() + "  " + a.getValue());
  }
}
```

Hur fungerar Collections?

Bilden till höger är uppbyggnaden av `TreeMap` ovh är ett exempel på en sorterad Lista eller Avbildningstabell.

Det vänstra delträdet innehåller lägre värden (tidigare i alfabetet).

Detta är en `HashMap`.



14

Trådar/Processer

Trådar

- En tråd beskriver en separat aktivitet inom ett program.
- Ett program kan ha flera trådar vilka exekverar parallellt
  (verkligt om man har flera processorer eller pseudoparallellt på en processor).
- Varje tråd har en aktuell exekveringspunkt.
- Varje tråd i ett Javaprogram beskrivs av en instans av klassen `Thread`.

```
Thread t = new Thread(r);   // skapar en ny tråd
```

Där `r` refererar till ett objekt av en klass som implementerar gränssnittet `Runnable`.
Detta gränssnitt har bara en metod:

```
void run();
```

Tråden startas med anropet

```
t.start();
```

Då kommer koden i metoden `run` att exekveras.

Andra grundläggande metoder i klassen `Thread`:

`interrupt()`	ber tråden att avsluta sin exekvering
`interrupted()`	ger **`true`** om tråden blivit ombedd att sluta
`sleep(m)`	låter den tråden vänta i `m` ms (och `n` ns), ger
`sleep(m,n)`	`InterruptedException` om tråden blivit ombedd att sluta
`t.join()`	väntar tills tråden `t` har avslutats
`t.join(m)`	väntar tills tråden `t` har avslutats, väntar högst `m` ms
`t.join(m,n)`	väntar tills tråden `t` har avslutats, väntar högst `m` ms och `n` ns
`getPriority()`	ger prioriteten för tråden `t`
`setPriority(p)`	ändrar prioriteten för tråden `t`

2

HokusPokus.java

```
public class HokusPokus {
  public static void main (String[] arg) {
    Skrivare s1 = new Skrivare("Hokus", 5),
             s2 = new Skrivare("Pokus", 9);
    s1.aktivitet.start();
    s2.aktivitet.start();
    XThread.delay(60000);  // vänta en minut
    s1.aktivitet.interrupt();
    s2.aktivitet.interrupt();
  }
}
```

Skrivare.java

```
public class Skrivare implements Runnable {
  public Thread aktivitet = new Thread(this);
  private String text;
  private long intervall;

  public Skrivare(String txt, long tid) {  // konstruktor
    text=txt;
    intervall = tid*1000;
  }

  public void run() {
    while(!Thread.interrupted()) {
      try {
        Thread.sleep(intervall);
      }
      catch (InterruptedException e) {
        break; // avbryt while-satsen
      }
      System.out.print(text + " "); System.out.flush();
      }
 }
}
```

XThread.java

```
public class XThread extends Thread {
  public static boolean delay(long millis) {
    if (interrupted())
      return false;
    try {
        sleep(millis);
    }
    catch (InterruptedException e) {
        return false;
    }
    return true;  // tråden har inte blivit avbruten
  }
}
```

Thread Safe

`synchronized` gör så att bara en sak åt gången kan exekveras, detta gör att en metod blir **thread safe** om den inte redan är det.

Om tråden ska kunna avbrytas:

```
public void run() {
  while(!Thread.interrupted()) {
    try {
      Thread.sleep(intervall);
    }
    catch (InterruptedException e) {
      break; // avbryt while-satsen
    }
    System.out.print(text + " "); System.out.flush();
  }
}
```

- Programkod som är utformad så att flera parallella trådar samtidigt kan använda sig av den utan att det blir fel, kallas *trådsäker* (*thread safe*) kod.
- I Java kan man göra en klass trådsäker genom att ange att vissa metoder skall vara *synkroniserade*.
- Ett objekt blir låst för andra trådar så länge en synkroniserad metod exekveras.

```
class Konto {
  private double saldo;
  ...
  public synchronized void transaktion(double belopp) {
    if (belopp<0 && saldo+belopp<0
      System.out.println("Uttag ej möjligt!");
    else
      saldo = saldo+belopp;
  }
  ...
}
```

Synkronisering med `wait` och `notify`.

```
import java.util.*;
public class SimpleQueue {
  private List<Object> l = new Vector<>();
  public int size() {
    return l.size();
  }
  public synchronized void put(Object obj) {
    l.add(obj);
    notify();
  }
  public synchronized Object take() {
    while (l.isEmpty())
      try {
        wait();
      }
      catch (InterruptedException e) {
        return null;
      }
    Object obj = l.get(0);
    l.remove(0);
    return obj;
  }
}
```