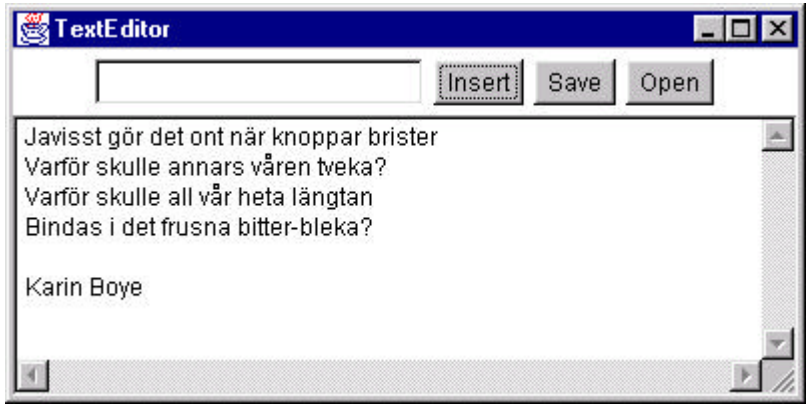
Programuppbyggnad från komponenter (en liten orientering)

Exempel från tre områden:

- grafiska användargränssnitt
- sparande på filer
- datasamlingsklasser

Texten till labb 8 finns sist i häftet, den finns även på
http://dsv.su.se/courses/pm2/introit/index.html
med klickbara hänvisningar till dokumentation

Grafiska användargränssnitt - en (konstig) texteditor i 5 steg



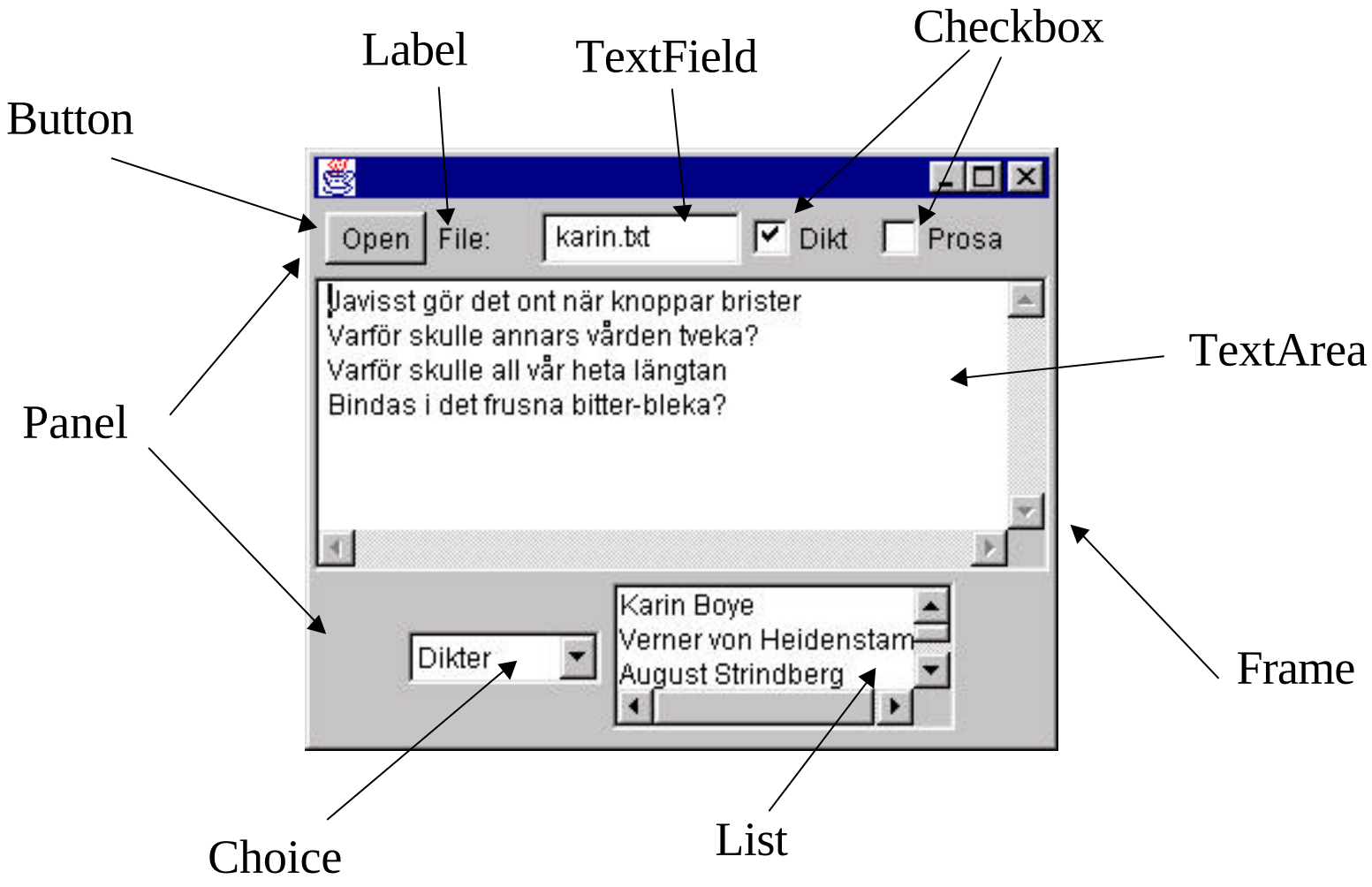
Editerbar text. I textfältet kan man mata in text, klickar man på ”Insert” så läggs texten in sist i textarean, klickar man på ”Save” så sparas texten från arean på en fil med namnet från textfältet och klickar man på Open så hämtas text från en fil med namnet från textfältet.

Fullständig programkod finns i häftet efter OH-bilderna

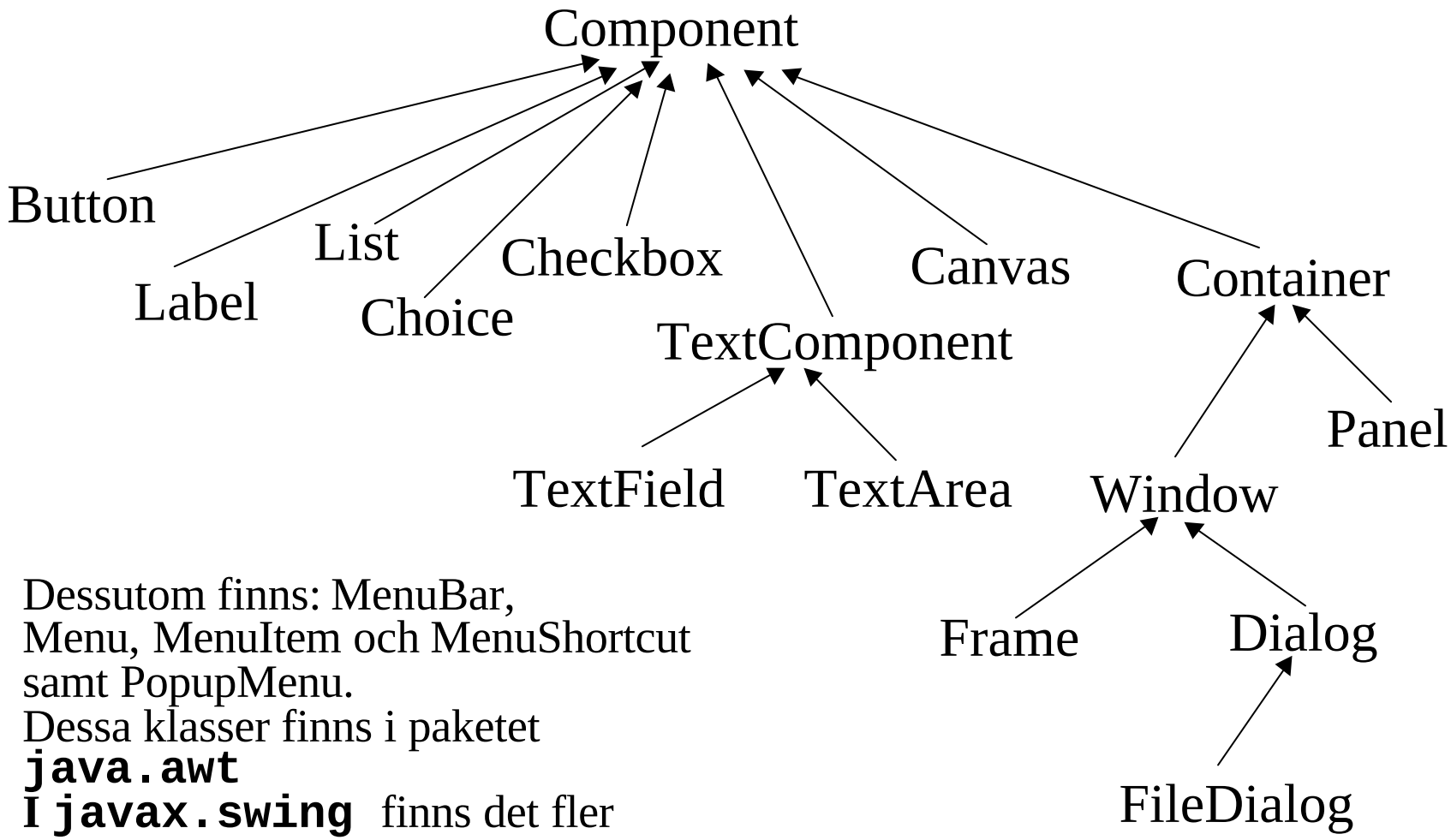
Första steget

```
import java.awt.*;

class TextEd1{
    Frame win = new Frame("TextEditor");
    TextArea ta = new TextArea();

    TextEd1(){
      win.add(ta, "Center");
      win.setSize(400, 200);
      win.show();
    } // Konstruktor

    public static void main(String[] args){
      new TextEd1();
    } // main

} // TextEd1
```

AWT-komponenter (ett urval)



AWT komponenthierarkin



Dessutom finns: MenuBar,
Menu, MenuItem och MenuShortcut
samt PopupMenu.
Dessa klasser finns i paketet
`java.awt`
I **`javax.swing`** finns det fler
komponenter samt alternativ till
AWT-komponenterna

Exempel på viktiga metoder

Alla komponenter

`void setBackground(Color color)`
`void setForeground(Color color)`

Färgen anges med ett objekt av klassen **`Color`** med Red, Green, Blue-värden (mellan 0 och 255) eller med en **`Color`**-konstant, t.ex.

```
ta.setBackground(new Color(130, 17, 255));
ta.setForeground(Color.red);
```

TextArea och TextField

`String getText()` - returnerar textinnehållet som sträng
`void setText(String str)` - sätter textinnehållet
Exempel: `String s = ta.getText();`
`ta.setText(””); tf.setText(”Hej hopp!”);`

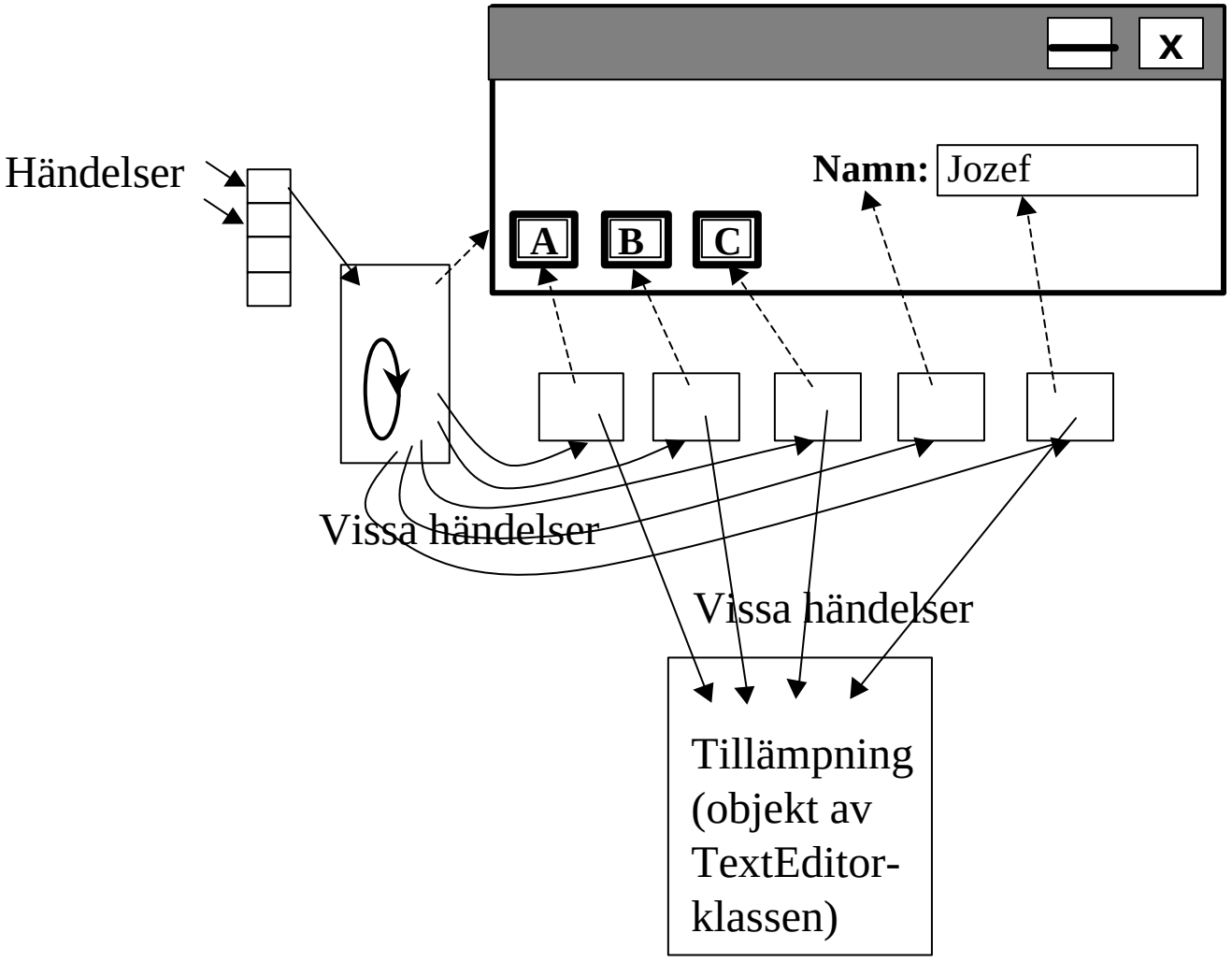
TextArea (bara)

`void append(String str)` - adderar strängen på slutet

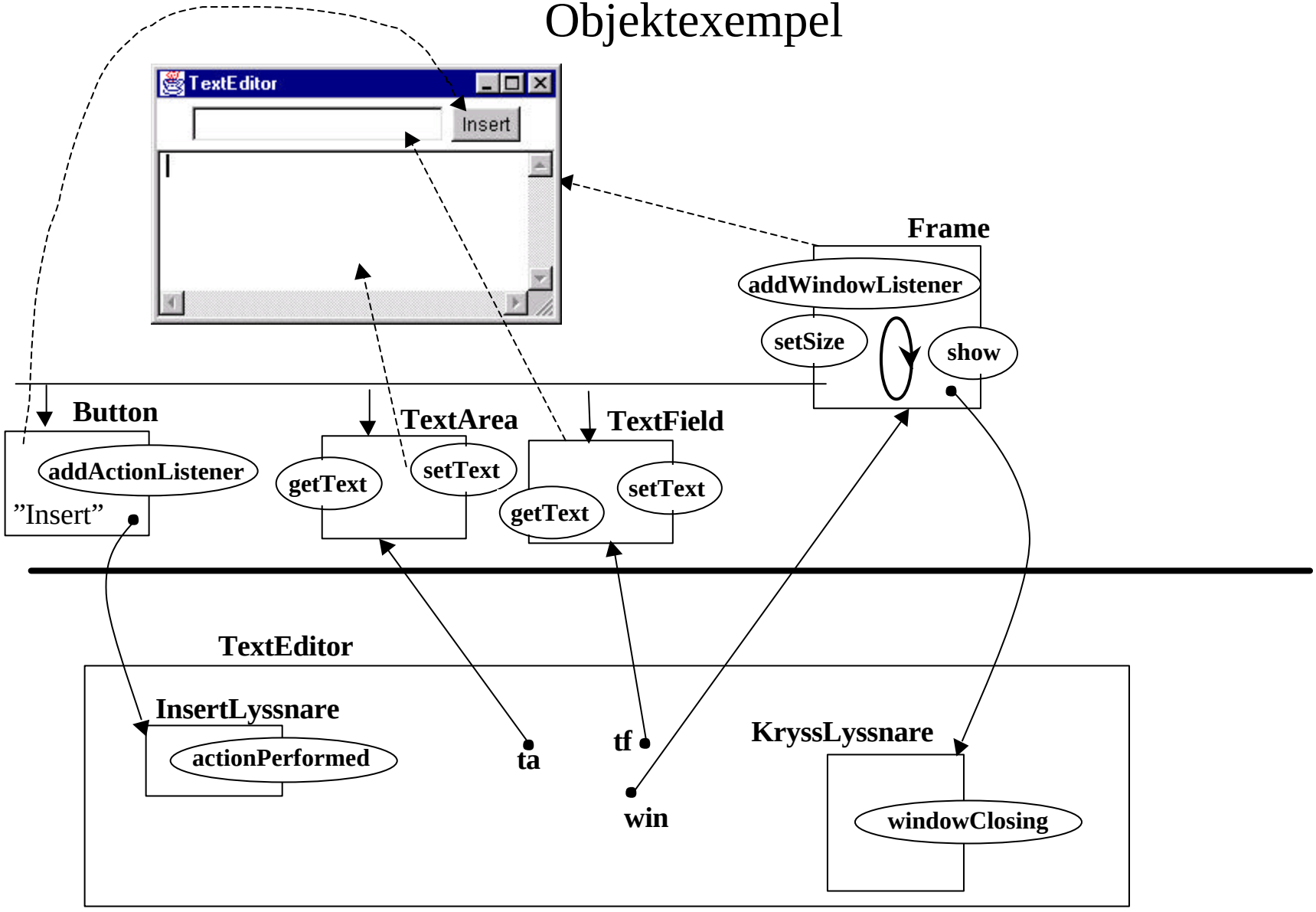
Button

`void addActionListener(ActionListener l)` -
”action” framkallas av att användaren klickar på knappen

Kommunikation fönster/komponenter/tillämpning

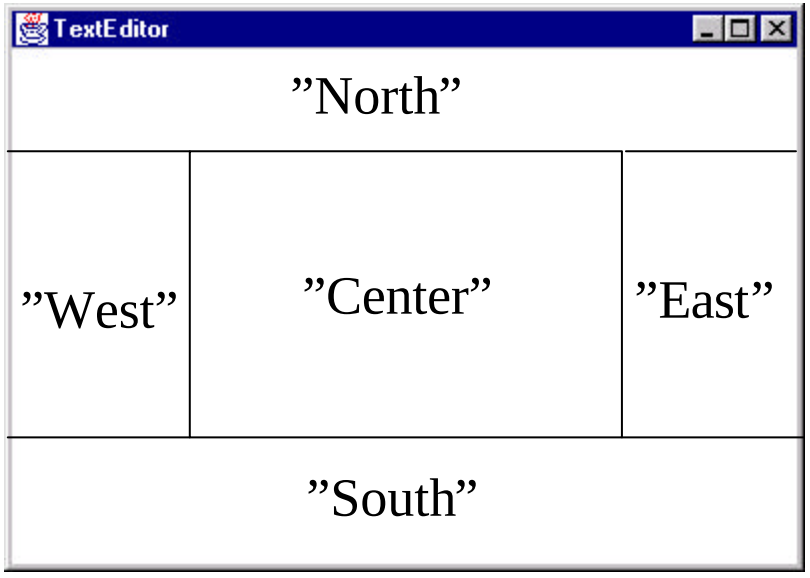


Objektexempel



```
import java.awt.*;
import java.awt.event.*;

class TextEd2{
    Frame win = new Frame("TextEditor");
    TextArea ta = new TextArea();

    TextEd2(){
      win.add(ta, "Center");
      win.addWindowListener(new KryssLyssnare());
      win.setSize(300, 200);
      win.show();
    } // Konstruktor

    public static void main(String[] args){
      new TextEd2();
    } // main

    class KryssLyssnare extends WindowAdapter{
      public void windowClosing(WindowEvent wev){
          System.exit(0);
      } // windowClosing
    } // KryssLyssnare

} // TextEd2
```

Fönsterlayout



```
... import som förut

class TextEd3{
    ... som förut
    TextField tf = new TextField(20);
    Button ib = new Button("Insert");

    TextEditor(){
      win.add(ta, "Center");
      Panel north=new Panel();
      win.add(north, "North");
      north.add(tf);
      north.add(ib);
      ib.addActionListener(new InsertLyssnare());
      ... som förut
    } // Konstruktor
    ... main och KryssLyssnare som förut

    class InsertLyssnare implements ActionListener{
      public void actionPerformed(ActionEvent ave){
          String str = tf.getText();
          if (str.length() > 0){
            ta.append(str);
            tf.setText("");
          } // if
      } // actionPerformed
    } // InsertLyssnare

} // TextEd3
```

```
import som förut ...
import java.io.*;

class TextEd4{
    ... som förut
    Button sb = new Button("Save");

    TextEd4(){
      ... som förut
      north.add(sb);
      sb.addActionListener(new SaveLyssnare());
      ... som förut
    } // Konstruktor

    ... main, KryssLyssnare och InsertLyssnare som förut

    class SaveLyssnare implements ActionListener{
      public void actionPerformed(ActionEvent ave){

         ... se nästa bild

      } // SaveLyssnare
} // TextEd4
```

```
class SaveLyssnare implements ActionListener{
  public void actionPerformed(ActionEvent ave){
    String filnamn=null;
    try{
      filnamn=tf.getText();
      ObjectOutputStream oos=new ObjectOutputStream(
                              new FileOutputStream(filnamn));
      String str = ta.getText();
      oos.writeObject(str);
    }catch(IOException e){
        System.out.println("Kan inte skriva på "+filnamn);
    } // catch
  } // actionPerformed
} // SaveLyssnare
```

```
import som förut ...

class TextEditor{
    ... som förut
    Button ob = new Button("Open");

    TextEditor(){
      ... som förut
      north.add(ob);
      ob.addActionListener(new OpenLyssnare());
      ... som förut
    } // Konstruktor

    ... main och lyssnarklasser som förut

    class OpenLyssnare implements ActionListener{
      public void actionPerformed(ActionEvent ave){
         ... se nästa bild
      } // actionPerformed
    } // OpenLyssnare

} // TextEditor
```

```
class OpenLyssnare implements ActionListener{
    public void actionPerformed(ActionEvent ave){
        String filnamn=null;
        try{

          filnamn=tf.getText();
          ObjectInputStream ois = new ObjectInputStream(
              new FileInputStream(filnamn));
          String str = (String)ois.readObject();
          ta.setText(str);

        }catch(IOException e){
          System.out.println("Kan inte läsa "+filnamn);
        }catch(ClassNotFoundException e){
          System.out.println("Kan inte läsa "+filnamn);
        } // catch

    } // actionPerformed
  } // OpenLyssnare
```

Viktiga metoder i klassen List

List

`void add(String item) -` adderar till slutet på listan

`String getSelectedItem() -` returnerar markerad sträng

`void select(int index) -` markerar strängen med detta nr

`void deselect(int index) -` avmarkerar strängen med detta nr

`void remove(String item) -` tar bort strängen från listan

`void remove(int index) -` tar bort strängen med detta nr

`void removeAll() -` tömmer listan

`void addActionListener(ActionListener l) -`

”action” framkallas av att användaren dubbelklickar på en sträng

Exempel:

```
class ListLyssnare implements ActionListener{
   public void actionPerformed(ActionEvent ave){
      String item = ls.getSelectedItem();
      ta.append(item);
   } // actionPerformed
} // ListLyssnare
```

Klassen **FileDialog**

FileDialog ger systemets vanliga fildialog. Vid skapande måste den ges referens till huvudfönstret (av klassen **Frame**), den kan även ges en titel och en av konstanterna **FileDialog.LOAD** eller **FileDialog.SAVE** (så att den anpassas till öppningsdialog resp. sparandedialog), t.ex.

```
FileDialog fd = new FileDialog(win);
```

eller

```
FileDialog fd = new FileDialog(win, ”Save as?”,
                                    FileDialog.SAVE);
```

Dialogen visas efter anrop av metoden **show()**, t.ex.:

```
fd.show();
```

Innan anropet av **show()** kan man sätta in ett förslag på filnamn, t.ex.:

```
fd.setFile(”minfil.txt”);
```

Dialogen avslutas när användaren har valt en fil eller tryckt på en av knapparna. Därefter kan man hämta filnamnet och sökvägen till katalogen, t.ex. för att öppna en fil med detta namn. Om användaren har tryckt på ”Cancel”-knappen blir filnamnet **null**. Exempel:

```
String fileName = fd.getFile();
if (fileName != null) {
   String path = fd.getDirectory();
   ObjectInputStream oos = new ObjectInputStream(
       new FileInputStream(path + fileName);
```

Datasamlingsklasser

De allra flesta program behöver hantera samlingar av data

Man kan använda arrayer för datasamlingar, men då måste man själv implementera de vanligaste operationerna:

- utöka arrayen vid behov - skapa ny array och kopiera de gamla värdena
- lägga till nya värden i slutet, hålla reda på antalet inlagda värden
- lägga till nya värden i mitten, flytta fram övriga värden för att ge plats
- ta bort något värde, flytta tillbaka övriga värden ett steg
- kontrollera om ett visst värde finns i arrayen
- lägga till alla värden som finns i en array till en annan array
- osv

I Javas klassbibliotek finns därför klassen **`ArrayList`** som innehåller en sådan array och implementerar sådana operationer. Det finns även några andra klasser för datasamlingar, samtliga i paketet **`java.util`**

ArrayList - exempel med objekt av egen klass

```
import java.util.*;

class Person{
   String namn;
   int pengar;
   Person(String n, int p) { namn=n; pengar=p;}
   int getPengar() { return pengar; }
} // Person

class Test{
   public static void main(String[] args){

      ArrayList lärare = new ArrayList();
      ArrayList studenter = new ArrayList();

      Person p1 = new Person(”Jozef”, 5);
      Person p2 = new Person(”Linda”, 100);

      lärare.add(p1);     // Jozef-objektet till lärarna
      studenter.add(p2);  // Linda-objektet till studenterna

      studenter.remove(p2); // Flytta Linda till lärarna
      lärare.add(p2);

      ArrayList fest=new ArrayList();
      fest.addAll(lärare);      // Ta med alla lärare
      fest.addAll(studenter);   // ...och även studenterna
      if (fest.contains(p1))    // Fast om Jozef är med ...
          fest.remove(p2);      // ...så kan man inte bjuda Linda
```

Object-referenser i **ArrayList**

ArrayList (liksom andra datasamlingsklasser) måste kunna hantera data av godtycklig typ. Hur kan då den interna arrayen vara deklarerad?
Av vilken typ får argument till metoderna (t.ex. **add**, **remove**, **contains**) vara?

I Java finns klassen **Object** som alla klasser är subklasser till (även arrayer).
En **Object**-referens kan därför peka ut objekt av alla klasser.

Den interna arrayens element och argumenten till metoderna är alltså **Object**-referenser!

Fyra viktiga konsekvenser:

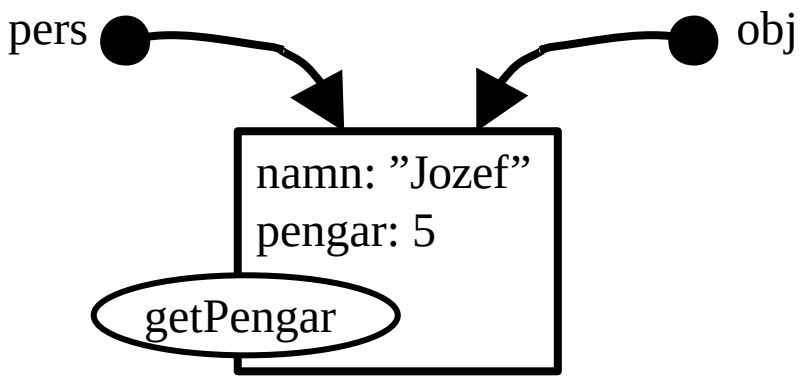
- objekt av alla klasser kan hanteras i en **ArrayList**
- endast klassobjekt och arrayer kan hanteras, inga värden av primitiva datatyper!
  Lösning: omslagsklasser: **Integer**, **Double** osv:

```
ArrayList minaTal = new ArrayList();
minaTal.add(new Integer(13));
```

- objekt av olika typer kan blandas i samma **ArrayList** (både för- och nackdel)
- när man hämtar ett objekt från en **ArrayList** vet man inte vad det är!

Attributsynlighet vid **`Object`**-referenser

```
Person pers = new Person(”Jozef”, 5);
Object obj = pers;
```




```
int x = pers.getPengar(); // Går bra
int y = obj.getPengar();  // Kompileringsfel!!!
```

Kompilatorn ”vet inte” att **`obj`** pekar ut ett Personobjekt, så den vägrar att acceptera anropet till **`getPengar`** (**`namn`**, **`pengar`** och **`getPengar`** ”syns inte”)

Man kan dock ”tvinga” kompilatorn genom explicit konvertering:

```
int y = ((Person)obj).getPengar();   // Går bra!
```

Konsekvens för datasamlingar

När man hämtar en objektreferens från en datasamling måste referensen konverteras till objektets förväntade typ innan man kan göra operationer på objektet.

Det går att testa typen med **`instanceof`**-operatorn, men det fallar utanför ramen för denna föreläsning.

Exempel:

`ArrayList` har metoden

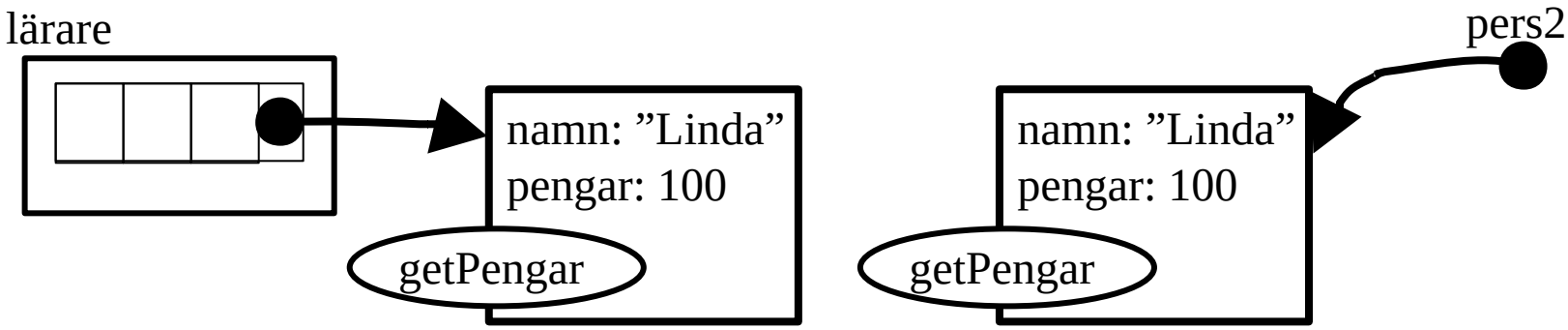
```
Object get(int index)
```

som returnerar referensen till objektet på plats nr **`index`**.
Man kan använda denna metod för att skriva ut vare lärares pengar:

```
for(int i=0; i<lärare.size(); i++) {  // Obs! metoden size()
  Person p = (Person)lärare.get(i);   // Obs! konvertering
  System.out.println(p.namn+" har "+p.getPengar()+" kronor");
} // for
```

Likhet mellan objekt: klass utan omdefinierad **`equals`**

```
ArrayList lärare=new ArrayList();
Person pers1=new Person(”Linda”, 100);
lärare.add(pers1);
Person pers2=new Person(”Linda”, 100);
```

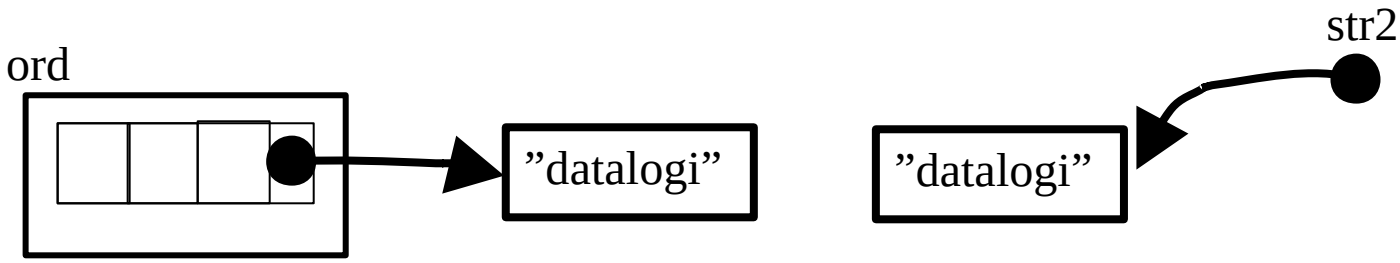



```
if (lärare.contains(pers2))          // Ger false!
   ...
```

Två Personobjekt är inte ”samma” bara för att de innehåller lika värden.

Likhet mellan objekt: klass med omdefinierad **`equals`**

```
ArrayList ord=new ArrayList();
String str1=”datalogi”;
ord.add(str1);
String str2=”datalogi”;
```



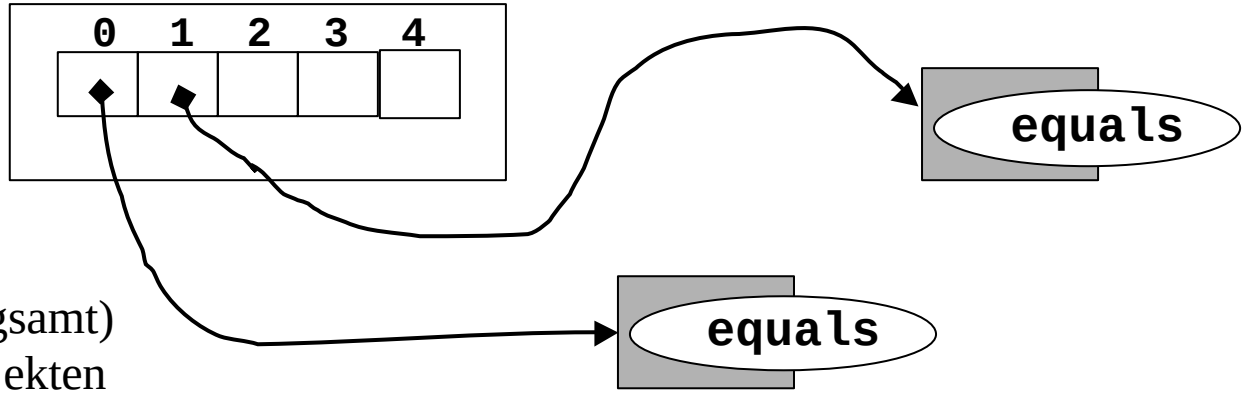

```
if (ord.contains(str2))                    // Ger true!
   ...
```

Två strängar är ”samma” om de har samma värde!

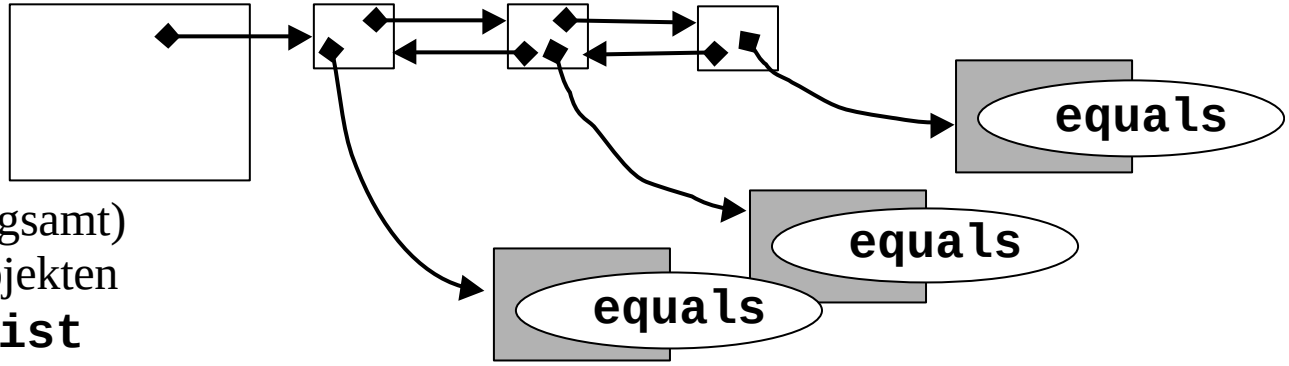
`contains` använder objektens **`equals`**-metod för att avgöra om två objekt är lika. **`String`**-klassen definierar om **`equals`**-metoden så att den jämför värden i objekten. På detta sätt kan samma datasamlingsklasser fungera olika för olika elementtyper (anpassas till behov).
Motsvarande gäller för datasamlingsklasser som kräver jämförelse för mindre än eller större än (för att kunna sortera elementen).

Något om implementeringstekniker I

Array (utökningsbar)
Snabb indexering,
snabba tillägg/borttag i slutet,
långsamma tillägg/borttag i
början och mitten.
Sökning genom iterering (långsamt)
Kräver endast **`equals`** av objekten
Används i klassen **`ArrayList`**



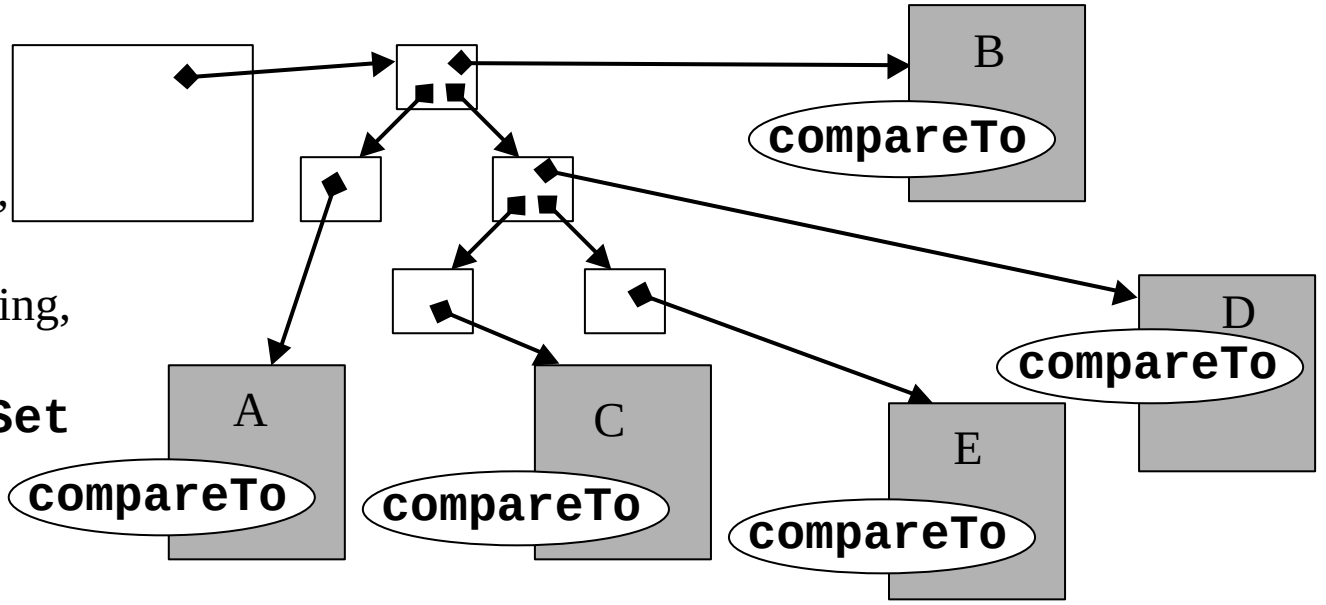
Länkad lista
Långsam indexering,
snabba tillägg/borttag.
Sökning genom iterering (långsamt)
Kräver endast **`equals`** av objekten
Används i klassen **`LinkedList`**



Något om implementeringstekniker II

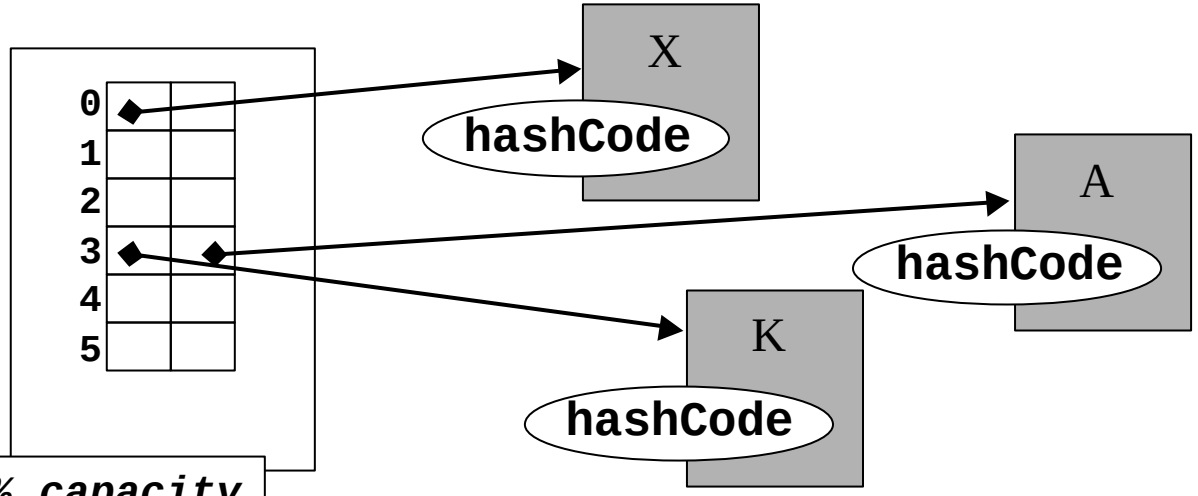
Balanserat binärt sökträd

Ganska snabba sökningar, tillägg och borttag.
Objekten i sorteringsordning, måste kunna jämföras.
Används i klassen **`TreeSet`** och i klassen **`TreeMap`**



Hashtabell

Använder **`hashCode()`*** som index i en tabell.
Snabb sökning, tillägg och borttag. Oordnad.
Används i **`HashSet`** och **`HashMap`**



* *Egentligen* ***`hashCode() % capacity`***

Javas datasamlingsklasser

`ArrayList` - arrayimplementering, tillåter dubletter,
har positionsoperationer (**`get(int index)`**,
`add(int index, object ny)`, **`remove(int index)`** osv)

`LinkedList` - länkad implementering, tillåter dubletter, har positionsoperationer

`HashSet` - hashtabell-implementering, tillåter inte dubletter,
har inga positionsoperationer, håller elementen osorterade

`TreeSet` - träd-implementering, tillåter inte dubletter,
har inga positionsoperationer, håller elementen sorterade,
kräver att elementen skall kunna jämföras med **`compareTo`**

`HashMap` - identifierar elementen med nycklvärden, implementerad med
hashtabell, håller elementen osorterade

`TreeMap` - identifierar elementen med nycklvärden, implementerad med
trädstruktur, håller elementen sorterade efter nyckelvärden,
kräver att nyckelvärden skall kunna jämföras med **`compareTo`**

Implementationsoberoende genomgång

Vid genomgång av alla element i en datasamling blir operationen ”ta fram nästa element” olika för olika implementeringar.

För att man skall kunna hantera alla datasamlingsklasser (utom **`Map`**-klasser) på samma sätt kan man hos ett objekt av en datasamlingsklass begära att få ett ”iterator”-objekt. Iteratorobjektet har främst två metoder:

`boolean hasNext()` - returnerar **`true`** om det finns ett nästa element
`Object next()` - returnerar referensen till nästa element

För objekt av klasserna **`ArrayList`**, **`LinkedList`**, **`HashSet`** och **`TreeSet`** kan genomgång av alla element göras på samma sätt, enligt följande exempel:
skriv ut alla lärares namn och pengar (igen):

```
Iterator iter = lärare.iterator();
while (iter.hasNext()){
   Person p = (Person)iter.next();
   System.out.println(p.namn+”: ”+p.getPengar()+” kr”);
} // while
```

Map-klasserna

Klasserna **`HashMap`** och **`TreeMap`** upprätthåller en avbildning av nycklar (som kan vara godtyckliga objekt) på värden (som också kan vara godtyckliga objekt).
Den viktigaste skillnaden mellan dem är att **`HashMap`** håller nyckel-värde-paren osorterade medan **`TreeMap`** håller dem sorterade i nyckel-ordning.

De viktigaste metoderna i Map-klasserna är

`Object put(Object key, Object value)` - stoppar in objektet **`value`** identifierat med objektet **`key`**. Om **`key`** fanns redan så bytes värdeobjektet, det gamla värdeobjektet returneras

`Object get(Object key)` - returnerar referensen till värdeobjektet för denna nyckel eller **`null`** om nyckeln inte fanns

`Object remove(Object key)` - tar bort nyckeln och värdet och returnerar värdet (eller **`null`** om nyckeln inte fanns)

TreeMap - användningsexempel

```
import java.util.*;

class Lexicon{
    public static void main(String[] args){
      TreeMap lexicon=new TreeMap();

      lexicon.put("hund", "dog");
      lexicon.put("katt", "cat");
      lexicon.put("dator", "computer");
      lexicon.put("grunka", "gadget");

      String svenska=LasIn.läsSträng("Uppslagsord: ");

      while (svenska.length() > 0) {
          String engelska = (String)lexicon.get(svenska);

          if (engelska == null)
            System.out.println("Finns inte i ordboken!");
          else
            System.out.println("Engelska: " + engelska);

          svenska = LasIn.läsSträng("Uppslagsord: ");
      } // while
```

Map-klasser: genomgång av nycklar eller värden

Man kan inte direkt gå igenom alla nyckel-värde-paren, alla nycklar eller alla värden i ett objekt av klasserna **`HashMap`** eller **`TreeMap`**.

Istället har dessa klasser metoder som returnerar datasamlingar med referenser till dessa objekt:

`keySet()` - returnerar en datasamling med referenser till alla nycklar
`values()` - returnerar en datasamling med referenser till alla värden
`entrySet()` - returnerar en datasamling med referenser till alla nyckel-värde-par

Man kan sedan gå igenom den returnerade datasamling med hjälp av en iterator, t.ex.: skriv ut alla svenska uppslagsord ur lexikonet:

```
Iterator iter = lexicon.keySet().iterator();
while (iter.hasNext()){
   String svord = (String)iter.next();
   System.out.println(svord);
} // while
```

För en **`HashMap`** skulle genomgången bli i slumpmässig ordning, för en **`TreeMap`** i sorteringsordning (eftersom nycklar här är strängar blir det alfabetisk ordning).