javabog.dk − − Forord

Indholdsfortegnelse

Indholdsfortegnelse

Indholdsfortegnelse

Indholdsfortegnelse

Indholdsfortegnelse

Indholdsfortegnelse

Indholdsfortegnelse

0 Forord

Denne bog henvender sig til de, der allerede har kendskab til Java eller grundlæggende objektorienteret programmering på et vist niveau, men som ønsker at vide mere.

Bogen udspringer af noter fra kurset "Videregående Programmering", som jeg begyndte at undervise i i efteråret 2001 på IT−Diplomuddannelsen på Ingeniørhøjskolen i København. De er senere blevet udbygget under videre undervisning i kurset.

0.1 Formål og indhold

Det, du kan opnå med denne bog, er:

- Bedre kendskab til almindeligt anvendte datastrukturer/samlinger af data og algoritmer
- At kunne lave komponentbaseret udvikling
  (afgrænsede programdele, der kan genanvendes i flere sammenhænge)
- At få kendskab og kunne anvende forskellige hyppigt anvendte designmønstre (eng.: reusable design patterns), såsom Singleton, Adapter, Iterator, Fabrik, Proxy, ... osv.
- At få bedre forståelse for Javas standardbibliotek
  (hvor designmønstrene er vidt anvendte)
- At blive mere rutineret i programmering
- At få kendskab til forskellige konkrete teknologier i Java
  (javadoc, introspektion, Swing, Java2D, JNI, JDBC, J2SE, J2ME, J2EE, EJB, ...) osv.

0.1.1 Bogens opbygning og indhold

Bogen består af 2 dele. Hver del består af nogle emner, der behandles i hvert sit kapitel. Det er i stor udstrækning muligt at læse kapitlerne uafhængigt af hinanden.

Forrest i hvert kapitel beskrives forudsætningerne for at læse kapitlet, så her kan du vurdere, om der er et andet kapitel du eventuelt bør se i først.

Første del: Videregående Java

Første del giver en introduktion til nogle af de vigtigste teknologier inden for java.

Kapitel 1, Samlinger af data, beskriver de almindeligt anvendte datastrukturer og algoritmer (Collections−klasserne − stakke, lister, mængder, afbildninger, hashtabeller, søgetræer...)

Kapitel 2, Dokumentation, beskriver brugen af javadoc, der er standarden for dokumentation af javakode.

Kapitel 3, Rekursion, giver en introduktion til rekursive algoritmer (metoder der kalder sig selv).

Kapitel 4, Komponentbaseret programmering, handler om javabønner (eng.: JavaBeans) som er genbrugelige programkomponenter som kan manipuleres visuelt i et udviklingsværktøj.

Kapitel 5, Grafiktegning (Java2D), giver en introduktion til den avancerede 2D−grafik i Java.

Kapitel 6, Grafiske brugergrænseflader (Swing), behandler nogle af de mere avancerede aspekter af brugergrænseflader og Swing, bl.a. JTable og Model−View−Controller−arkitekturen brugt i Swing.

Kapitel 7, Internationalisering, diskuterer, hvordan man får sine programmer til at kunne køre på flere sprog.

Kapitel 8, Databaser (JDBC), uddyber JDBC og databaser, bl.a. m.h.t. optimering og metadata.

Kapitel 9, Optimering, giver en række råd og vejledning i, hvordan man kan optimere sit javaprogram, så det kører hurtigere.

Kapitel 10, Eksterne kald og JNI, beskriver, hvordan man kalder eksterne programmer, og hvordan man bruger JNI (Java Native Interface) til at kalde f.eks. C− eller C++−kode fra Java.

Kapitel 11, Introspektion, beskriver, hvordan man undersøger et vilkårligt objekts variabler og metoder under kørslen (også kaldet reflection), og hvordan man løbende kan generere nye klasser og indlæse dem under kørslen.

Kapitel 12, Standardbiblioteket (J2SE), giver et overblik over hele standardbiblioteket (kaldet J2SE − Java 2 Standard Edition).

Kapitel 13, Mobiltelefoner (J2ME), beskriver, hvordan man programmerer til J2ME (Java 2 Micro Edition) og midletter til små, indlejrede systemer (bl.a. mobiltelefoner).

Kapitel 14, Servere (J2EE & EJB), beskriver J2EE (Java 2 Enterprise Edition), der er beregnet til større serversystemer og webservere.

Anden del: Videregående objektorienteret programmering

Anden del beskriver en række af de mest anvendte designmønstre og strategier for, hvordan man skal designe sit program.

Kapitel 15, Videregående OOP, diskuterer nogle af de begreber og tankegange, der gør sig gældende inden for objektorienteret programmering, og introducerer designmønstre.

Kapitel 16, Skabende designmønstre, beskriver designmønstre til at mindske bindingen mellem den del af programmet som bruger nogle objekter, og den del af programmet, der bestemmer, hvordan disse objekter oprettes/fremskaffes.
Abstrakt Fabrik, Fabrik, Singleton, Bygmester, Prototype og Objektpulje gennemgås.

Kapitel 17, Hyppigt anvendte designmønstre, beskriver de designmønstre, der ofte ses anvendt i i standardbiblioteket og i lidt større programmer.
Proxy, Adapter, Iterator, Facade, Observatør/Lytter og Dynamisk Binding gennemgås.

Kapitel 18, Andre designmønstre, beskriver nogle lidt mere eksotiske designmønstre, der sjældnere ses anvendt, men som er velegnede i visse situationer.
Uforanderlig, Fluevægt, Filter, Lagdelt Initialisering, Komposit/Rekursiv komposition og Kommando gennemgås.

Kapitel 19, Model−View−Controller−arkitekturen, beskriver, hvordan man kan opdele programmer med en brugergrænseflade i en model, en præsentation og en kontrol−del.

0.2 Forudsætninger for at læse bogen

For at få fuld nytte af hele bogen skal du på forhånd have kendskab til nogle emner, som er kendte fra grundkurser i Java og objektorienteret programmering:

- Vector−klassen og andre almindeligt kendte objekter
- Arv og polymorfi

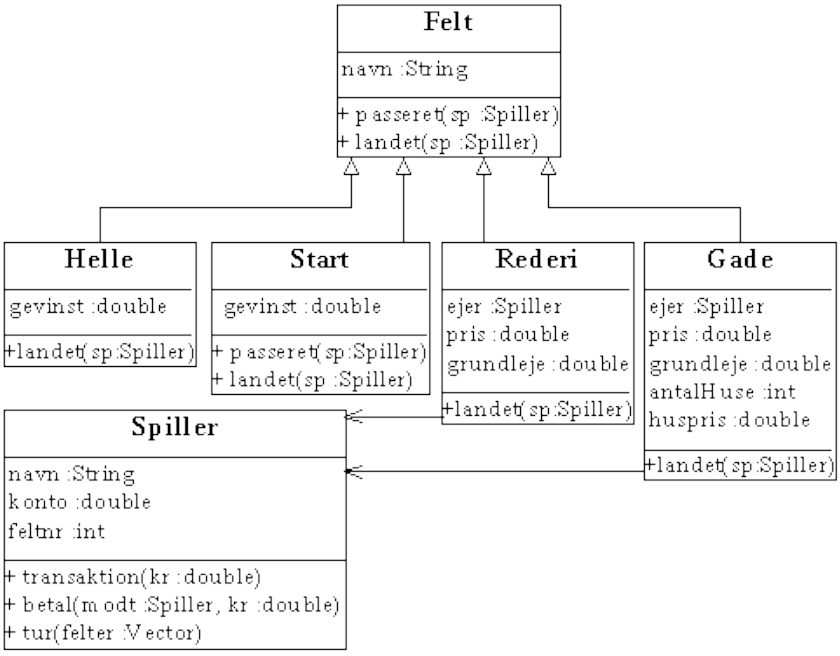
- Indkapsling (public/private)

- Lokale variabler, objektvariabler og klassevariabler og deres virkefelt (eng.: scope)
- Hændelser i grafiske brugergrænseflader
- Programmering af grafiske brugergrænseflader på et basalt niveau

Hvis du er usikker i et eller flere af emnerne, kan du læse om dem på http://javabog.dk.

0.2.1 UML: Klassediagrammer

Klassediagrammer forudsættes kendt. Vigtige relationer mellem klasser er *har*−relationen og *er−en*−relationen. Eksempel (et matadorspil):



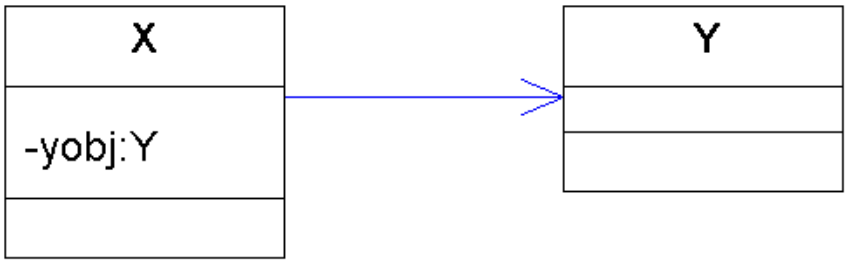
Rederi og Gade *har* en Spiller.

Helle, Start, Rederi og Gade *er−et* Felt.

Har−relationen

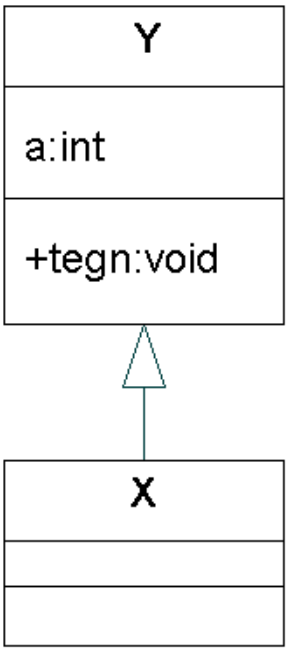
At X *har* Y betyder: i klassen X er defineret en variabel af type Y. Et X−objekt har derfor en reference til et Y−objekt, og derfor har X−objektet mulighed for at kalde metoder og bruge variabler fra Y−objektet gennem denne reference. Y−objekter kender ikke nødvendigvis noget til X−objekters eksistens.

Eksempel (det væsentlige i fed):




```
public class X
{
  private Y yobj = new Y();
  ...
}
```

Er−en−relationen



At X *er−en* Y betyder: i klassen X er defineret, at X arver fra klassen Y. Et X−objekt indeholder derfor (mindst) alle de metoder og variabler, som et Y−objekt indeholder.

Eksempel:

```
public class X extends Y
{
  ...
}
```

Det er også en er−en−relation, når man implementerer et interface.

0.3 Netudgaven af bogen

Størstedelen af bogen (opgaver og avancerede emner er skåret væk) findes også på adressen http://javabog.dk/VP, der frit kan bruges af enhver.

Her kan du også hente:

- Programeksemplerne fra bogen (http://javabog.dk/VP/kode)
- Undervisningsmateriale, såsom lektionsplaner, ugesedler, opgaver og transparenter
- Foredrag om udvalgte kapitler (som lydfiler)
- Rettelser og tilføjelser til bogen

Netudgaven, dvs. alt hvad der kan findes på http://javabog.dk/VP, er frigivet under Åben Dokumentlicens (ÅDL). Det betyder at du kan udskrive og kopiere netudgaven, som du lyster.

Den præcise ordlyd af Åben Dokumentlicens kan læses på næste side.

I netudgaven er avancerede afsnit, løsninger på opgaver samt kapitel 14, Servere (J2EE & EJB), og kapitel 18, Andre designmønstre, skåret væk.

Bidrag og rettelser til bogen og hjemmesiden er meget velkomne. Send dem til: nordfalk@mobilixnet.dk

0.4 Åben Dokumentlicens (ÅDL)

Version 1.0 (25. januar 2002)

1. Kopiering og distribution af værket i uændret form

Du må frit kopiere dette værk uændret og distribuere det på ethvert medium, forudsat at du sammen med hver kopi bibeholder denne licens og en henvisning til værkets kilde.

Du kan vælge at kræve penge for kopiering og distribution af indholdet, for undervisning i indholdet, for brugerstøtte og garantier i forbindelse med indholdet. Hvis du videregiver værket til andre, har du pligt til at se til, at du selv eller andre giver modtageren gratis adgang i minimum 3 år til en elektronisk udgave af indholdet − i et åbent redigérbart format − for eksempel via internettet, og at der i hvert eksemplar tydeligt angives, hvordan man kan få den elektroniske udgave.

2. Ændring af indholdet

Du må gerne ændre i dette værk eller dele af det (og dermed lave et nyt værk baseret på dette værk) og distribuere det som nævnt ovenfor, forudsat at du opfylder alle følgende krav:

a. Du skal sørge for, at det nye værk indeholder en tydelig anmærkning om, at du ændrede det, hvad du ændrede og datoen for ændringerne.
b. Det skal være omfattet af samme rettigheder og betingelser ("licens") som det oprindelige værk.

Alternativt kan det nye værk bære en anden licens eller almindelig ophavsretlig beskyttelse, forudsat at samtlige dele, der oprindeligt stammer fra dette værk, tydeligt er markeret (f.eks. i sidefoden på hver side) som værende under denne licens, og at licensen følges for disse dele af værket, herunder at der gives gratis elektronisk adgang som nævnt under afsnittet om kopiering og distribution.

3. Pligter

Enhver anvendelse, kopiering eller distribuering betragtes som en accept af denne licens og dens betingelser. Ved overtrædelse af licensbetingelserne, vil almindelig ophavsretlig beskyttelse være gældende for hele værket og vil dermed gøre person eller personer erstatningspligtige for enhver uautoriseret anvendelse.

4. Garanti

Indholdet leveres "som det er", uden nogen form for garanti for købs−, brugs− eller nytteværdi, medmindre der er givet en særskilt skriftlig garanti herom.

Denne bog er under ÅDL, med undtagelse af avancerede afsnit, løsninger på opgaver samt kapitel 14, Servere (J2EE & EJB), og kapitel 18, Andre designmønstre. Disse dele er under almindelig ophavsretlig beskyttelse.

0.5 Tak

Først og fremmest en speciel tak til Tina Modvig Frederiksen, først kursist i faget Videregående Programmering, senere medunderviser i det. Hun er kommet med mange vigtige bidrag til bogen, bl.a. afsnit 15.6, Ansvarsområder, og en meget stor del af kapitel 13, Mobiltelefoner (J2ME).

Tak til Troels Nordfalk og de mange andre kursister, der gennem tiden har påpeget fejl og mangler i noterne, bl.a. Kenn Thisted, Florin−Gabriel Badea, Jon Axelsson, Anita Mygind og Jan Christiansen.

Sidst men ikke mindst − tak til Linux−samfundet for at lave et styresystem, der styrer!

Denne bog er skrevet med OpenOffice.org under Mandrake Linux. Begge er projekter med åben kildekode (Open Source) og kan gratis hentes på henholdsvis www.openoffice.org og www.linux−mandrake.com.

Bonan plezuron legi la libron!
(det er esperanto og betyder "god fornøjelse med at læse bogen")

Jacob Nordfalk

Valby, maj 2003.

Første del: Videregående Java

1 Samlinger af data

En overfladisk forståelse af dette kapitel (især lister) forudsættes i det meste af bogen.

En samling (eng.: collection) er en datastruktur, der husker på en samling objekter.

Med JDK1.2 blev der tilføjet en række klasser til java.util−pakken, og de bliver under et bliver betegnet collections−klasserne. De kan meget mere end Vector og Hashtable, og det anbefales at bruge de nye klasser til programmer, der ikke behøver at kunne køre under JDK1.1.8 eller tidligere.

1.1 Lister og mængder

Den mest almindelige liste er ArrayList, en liste, der husker sine objekter, som lå de som perler på en snor lige som den gamle Vector−klasse. Man opretter en ArrayList med f.eks.:

```
ArrayList liste;
liste = new ArrayList();
```

Derefter kan man tilføje et objekt i enden af listen med **add**(objekt), f.eks.:

```
liste.add("æh");
```

tilføjer strengen "æh" sidst i listen.

Man kan sætte ind midt i listen med **add**(objekt, indeks), f.eks.:

```
liste.add("øh",0);
```

indsætter "øh" på plads nummer 0, sådan at listen nu indeholder først "øh" og så "æh". Alle elementerne fra og med det indeks, hvori man indsætter, får altså rykket deres indeks et frem. Ligeledes rykkes alle de efterfølgende elementer, hvis man fjerner et element med metoden **remove**(indeks).

Man henter elementerne ud igen med **get**(indeks).

Med liste.**size**() får man antallet af elementer i listen, i dette tilfælde 2.

Nogle af metoderne i List−klasserne (så som ArrayList og LinkedList)

Metoder

boolean **add**(objekt)
Føjer *objekt* til listen (sidst). Objektet kan være af en vilkårlig klasse. Simple typer (som int og double) tillades ikke (se afsnit 1.1.1). Returnerer altid true.

void **add**(objekt, int indeks)
Indsætter *objekt* i listen lige før plads nummer *indeks*.

void **remove**(int indeks)
Sletter objektet på plads nummer *indeks*.

boolean **remove**(objekt)
Sletter *objekt* fra listen, hvis det findes. Hvis det fandtes, returneres true, ellers false.

int **size**()
Returnerer antallet af elementer i listen.

Object **get**(int indeks)
Returnerer en reference til objektet på plads nummer *indeks*. Husk at lave en typekonvertering af referencen til den rigtige klasse, før resultatet lægges i en variabel. Listen selv bliver ikke ændret.

String **toString** ()
Returnerer listens indhold som en streng. Dette sker ved at konvertere hver af elementerne til en streng.

Ovenstående beskrivelse gælder ikke kun ArrayList, men er en beskrivelse af et interface, der gælder for alle List−objekter (lister), herunder ArrayList. Vi vil senere se på et andet List−objekt, nemlig LinkedList.

1.1.1 De simple typer og samlinger af data

Ligesom Vector kan lister (eller nogen af de andre nye samlinger) kun indeholde objekter, men ikke de simple typer som int og double:

```
liste.add( 5 );                 // forkert!
liste.add( 3.14 );              // forkert!
```

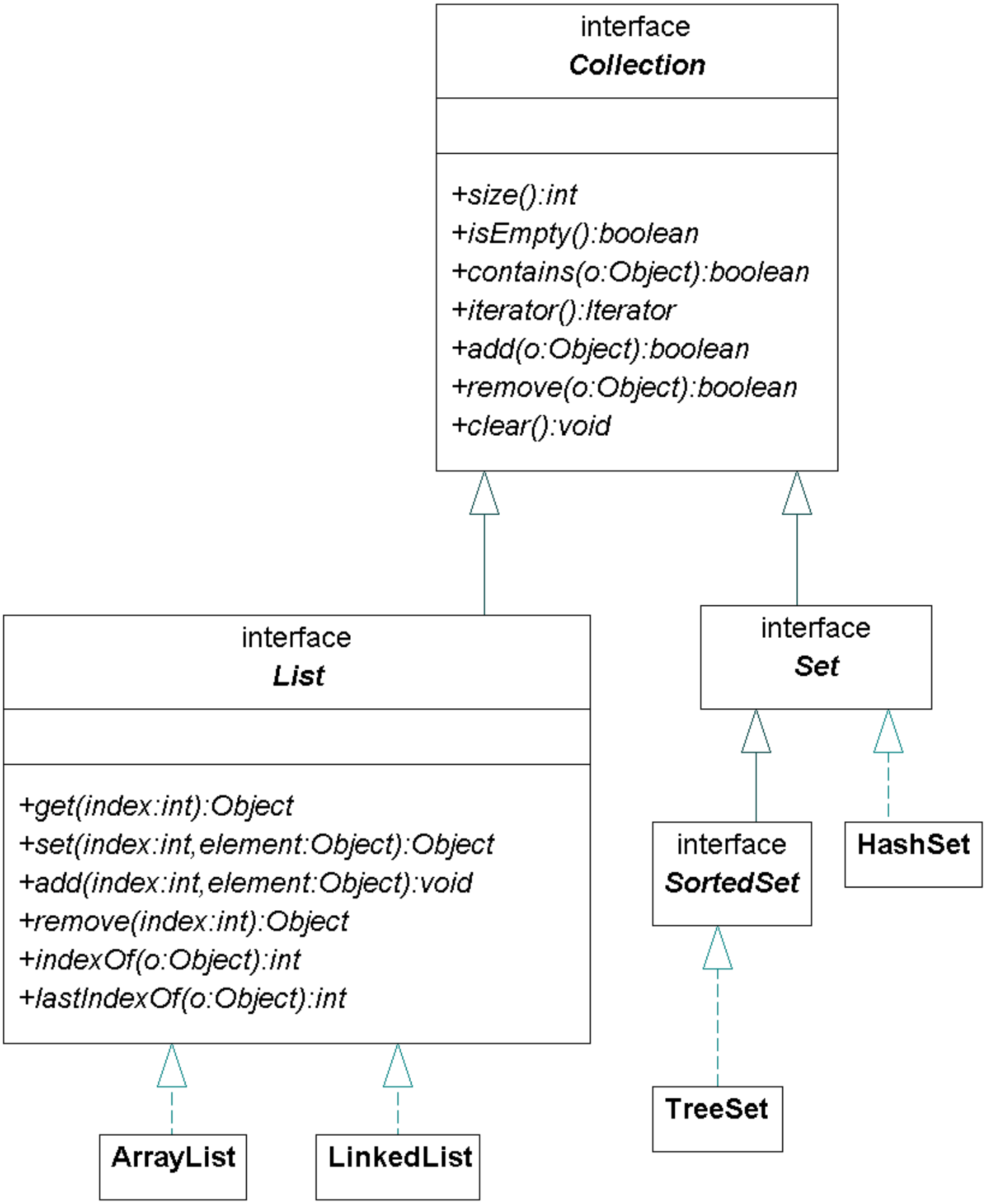
Man er nødt til at pakke dem ind i objekter, der svarer til de simple typer, f.eks.:

```
liste.add( new Integer(5) );      // rigtigt
liste.add( new Double(3.14) );    // rigtigt
```

1.1.2 Interfacene til lister og mængder

I stedet for bare at levere nogle klasser, som programmørerne så kan bruge (som med f.eks. Vector), er alle de væsentlige metoder beskrevet i interfaces, og der findes så flere forskellige implementationer af interfacene (klasser) til forskelligt brug.

Interfacene er vist herunder sammen med klasserne, der implementerer dem.



Interfacet Collection (samlinger af data)

Interfacet Collection (da.: samling) beskriver en samling data og indeholder metoder fælles for alle samlinger af data.

Der er ikke nogen klasser, der direkte implementerer Collection. I stedet implementerer de interfacene List, Set eller SortedSet, der udbygger (arver fra) Collection.

1.1.3 Lister

List beskriver en *ordnet* liste af elementer (dvs. elementerne har en rækkefølge og hvert element har en plads, ligesom med Vector).

Den indeholder derfor (ud over dem i Collection) metoder, der forudsætter at elementerne har et index, som f.eks. get(int index), Object remove(int index) og set(int index, Object element). Da hvert element identificeres med et indeks, kan der godt være dubletter (dvs. det samme element optræder flere gange i listen på forskellige indeks). Eksempel på brug:

```
List l = new ArrayList();      // variablen er bare af interfacets type
l.add("En");
l.add("snegl");
```

Læg mærke til, at man ofte bare lader variablen være af *interfacets* type. Det gør, at man nemt kunne ændre ovenstående kode til at bruge LinkedList eller en anden samling i stedet (der skal kun ændres fra "new ArrayList()" til "new LinkedList()".

Der findes to klasser, der implementerer List, nemlig ArrayList og LinkedList. Udefra bruges de på helt samme måde (og har de samme metoder), men de fungerer forskelligt indeni.

ArrayList

ArrayList bruger internt et array til at gemme sine data, og svarer til den gamle Vector−klasse (men ArrayList er dog lidt hurtigere end Vector, da dens metoder ikke er synkroniseret).

Den er derfor hurtig at finde elementer i, men langsom, hvis elementer skal indsættes et vilkårligt sted i listen (da de andre elementer i arrayet så skal rykkes). Se afsnit 1.6.2 for en nærmere diskussion af ArrayList's virkemåde.

LinkedList

LinkedList bruger internt en dobbelthægtet liste til at gemme sine data. I en dobbelthægtet liste gemmes data led i en kæde: hver indgang har en reference til ("er hægtet" med) forrige og næste indgang.

Skal man søge et bestemt index frem i en sådan liste, går det langsomt: Man må starte fra en ende og så løbe gennem hver indgang ("led i kæden"), indtil den ønskede indgang nås. Det er til gengæld meget hurtigt at føje nye elementer til listen (da de andre elementer ikke behøves at blive rykket). Se afsnit 1.6.3 for en nærmere diskussion af LinkedList's virkemåde.

1.1.4 Mængder

Interfacet Set

Set (da.: mængde) beskriver en uordnet mængde af elementer. I en mængde kan der højest være en af hvert element, og elementerne ligger "hulter til bulter" mellem hinanden, dvs. rækkefølgen, som elementerne blev sat ind, huskes ikke. Den indeholder ingen metoder, ud over metoderne fra Collection.

Klassen HashSet

Klassen HashSet implementerer Set. Den bruger en hashtabel til at holde styr på elementerne. En hashtabel er en opslagstabel, der laves ud fra objekternes hashkode (se afsnit 1.6.4). Hashtabeller er meget hurtige at søge og indsætte i. Eksempel på brug:

```
Set m = new HashSet();      // variablen er bare af interfacets type

m.addAll( l );              // tilføj alle elementer fra listen l
m.add("er");
m.add("tegn");
```

Se afsnit 1.6.4 for en diskussion af hashtabellers interne virkemåde.

Sorterede mængder (Interfacet SortedSet og klassen TreeSet)

SortedSet beskriver en ordnet mængde af elementer (dvs. elementerne har en rækkefølge, og der kan højest være én af hver).

Klassen TreeSet implementerer SortedSet. Den bruger et binært søgetræ til at holde styr på elementerne (se afsnit 1.6.5).

Ordningen (dvs. hvordan elementerne skal sorteres) kan bestemmes ved at angive et Comparator−objekt i konstruktøren til TreeSet (ellers skal elementerne være Comparable).

Eksempel på brug:

```
SortedSet sm = new TreeSet(); // variablen er bare af interfacets type

sm.addAll( m );               // tilføj alle elementer fra mængden m
sm.add("på");
sm.add("regn");
```

1.1.5 Iteratorer

En iterator er et lille objekt, der hjælper med at gennemløbe nogle data.

I stedet for at bruge en tællevariabel:

```
for (int i=0; i<liste.size(); i++) {                    // uden iterator
  String s = (String) liste.get(i);
  // gør noget med s
  System.out.println(s);
}
```

... kan man altså få et iterator−objekt ved at kalde metoden iterator() og bruge det:

```
for (Iterator iter=liste.iterator(); iter.hasNext(); ) {  // med iterator
  String s = (String) iter.next();
  // gør noget med s
  System.out.println(s);
}
```

Her er det samme skrevet om til at bruge en while−løkke (hvilket gør det lidt enklere):

```
Iterator iter=samling.iterator();                       // med iterator
while (iter.hasNext()) {
  String s = (String) iter.next();
  // gør noget med s
  System.out.println(s);
}
```

... der altså svarer til:

```
int i = 0;                                              // uden iterator
while (i<liste.size()) {
  String s = (String) liste.get(i);
  // gør noget med s
  System.out.println(s);
  i++;
}
```

Iterator−objekter har altså to[1] vigtige metoder:

boolean **hasNext**()
fortæller, om der er flere elementer i

Object **next**()
går til næste element og returnerer det samtidig.

Alle samlinger af data har metoden iterator(), der returnerer et Iterator−objekt. Iteratorer kan altså bruges på enhver samling af data, også på f.eks. mængder, hvor en almindelig tællevariabel kommer til kort.

1.2 Afbildninger (nøgleindekserede lister)

En afbildning (eng.: map) er noget, der afbilder nogle nøgler til værdier. Afbildninger er nyttige, når man vil indeksere nogle objekter ud fra f.eks. et navn i stedet for et nummer.

Afbildninger i forhold til lister

Før vi går videre, så husk, at lister *går fra heltal til objekter*, dvs. hvert element har en plads og et nummer. For eksempel har en liste metoderne:

void **add**(objekt, int indeks)
Indsætter *objekt* i listen lige før plads nummer *indeks.*

Object **get**(int indeks)
Returnerer en reference til objektet på plads nummer *indeks* i listen.

Man kan også sige, at elementerne i en liste indekseres (fremfindes) ud fra et tal.

1.2.1 Afbildninger (Interfacet Map)

Afbildninger *går fra objekter til objekter* på den måde, at til hvert element knyttes et nøgle−objekt. Elementerne kan derefter findes frem ud fra nøglerne.

Man kan altså sige, at elementerne (værdierne) i en afbildning indekseres (fremfindes) ud fra nøglerne, og at en afbildning består af nogle nøgle−værdi−par.

Interfacet Map beskriver metoderne, der kan kaldes på en afbildning.

Nogle af Map−klassernes metoder − afbildning eller nøgleindekseret liste af objekter (se appendiks for fuld liste)

void **put**(Object nøgle, Object værdi)
føjer objektet *værdi* til afbildningen under objektet *nøgle*. Objekterne kan være et vilkårligt objekt (men ikke en simpel type).

Object **get**(Object nøgle)
finder indgangen under *nøgle* og returnerer værdien. Husk at lave en typekonvertering af referencen til den rigtige klasse, før resultatet lægges i en variabel.

Object **remove**(Object nøgle)
sletter indgangen under *nøgle* og returnerer værdien.

boolean **isEmpty**()
returnerer sand, hvis afbildningen er tom (indeholder 0 indgange).

int **size**()
returnerer antallet af indgange (nøgle−værdi−par).

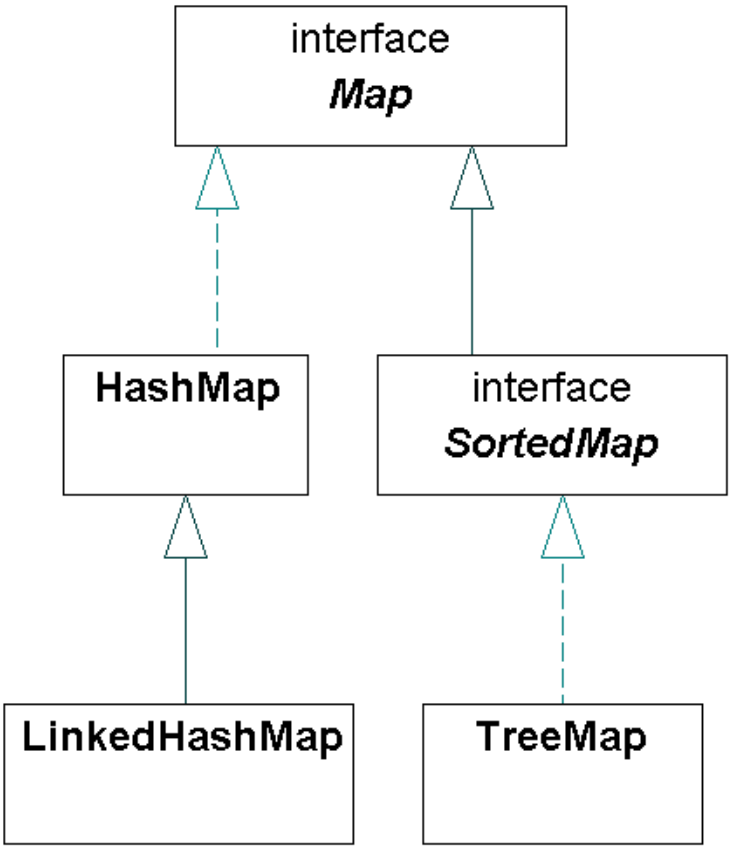
boolean **containsKey**(Object nøgle)
returnerer sand, hvis *nøgle* findes blandt nøglerne i afbildningen.

boolean **containsValue**(Object værdi)
returnerer sand, hvis *værdi* findes blandt værdierne i afbildningen.

Set **keySet**()
giver en datastruktur (en mængde) af afbildningens nøgler. Kan være nyttigt, hvis man vil gennemløbe alle indgangene i afbildningen.

String **toString** ()
Returnerer alle afbildningens indgange som én stor streng. Dette sker ved at konvertere hver af indgangenes nøgler og værdier til strenge.

Klassen HashMap er den eneste, der implementerer Map−interfacet direkte (indirekte er der to til, nemlig klassen LinkedHashMap, der kom til i JDK1.4, og klassen TreeMap).



1.2.2 Hashtabeller (klassen HashMap)

Man opretter en afbildning (i dette tilfælde en hashtabel) med f.eks.:

```
Map tabel;
tabel = new HashMap();
```

Derefter kan man lægge en indgang i tabellen med **put**(nøgle, værdi). F.eks.:

```
tabel.put("abc", "def");
```

husker strengen "def" under nøglen "abc". Vil vi finde strengen frem igen, skal vi slå op under "abc":

```
  String værdi = (String) tabel.get("abc");
```

HashMap afløser den gamle HashTable−klasse og kan nogenlunde det samme. Da hashtabeller oftest anvendes til at lave afbildninger med, bruger man i dagligdags sprogbrug mere ordet 'hashtabel' end ordet 'afbildning'.

1.2.3 Eksempel − ordbog

Vi kunne f.eks. bruge en afbildning (HashMap) til at lave en lille esperanto−dansk−ordbog med. Nøglerne er esperanto−ordene, og værdierne er de danske ord.

```
import java.util.*;

public class BenytHashMapOrdbog
{
  public static void main(String[] args)
  {
    HashMap ord = new HashMap();
    ord.put("granda", "stor");
    ord.put("longa", "lang");
    ord.put("hundo", "hund");
    ord.put("estas", "(det) er");

    String esperantotekst = "Estas longa granda hundo.";
    StringTokenizer st = new StringTokenizer(esperantotekst, " .,\t\n", true);
    while (st.hasMoreTokens()) {
      String da, eo;
      eo = st.nextToken().toLowerCase();
      da = (String) ord.get(eo);// slå esperantoordet op og få det danske ord
      if (da == null) da = eo;  // hvis intet fundet lader vi det være uoversat
      System.out.print( da );
    }
  }
}
```

```
(det) er lang stor hund.
```

Har vi en tekst på esperanto, kan vi nu oversætte teksten ord for ord til dansk. Hvert ord slås op i hashtabellen, og hvis det findes, erstattes det med det danske ord. Ord som ikke kan findes, efterlades uforandret.

1.2.4 Eksempel − fødselsdatoer

Nøglerne og værdierne behøver ikke være strenge. I eksemplet herunder opretter vi en hashtabel, der holder styr på fødselsdatoer for et antal personer ud fra deres fornavne.

```
import java.util.*;

public class BenytHashMapDatoer
{
  public static void main(String[] args)
  {
    HashMap hashtabel = new HashMap();
    Date dato;

    dato = new Date(71,0,1);               //  1. januar 1971
    hashtabel.put("Jacob",dato);

    dato = new Date(72,7,11);              // 11. august 1972
    hashtabel.put("Troels",dato);

    hashtabel.put("Ulrik",new Date(73,2,5));//  5. marts 1973
    hashtabel.put("Ulla",new Date(69,1,19));// 19. februar 1969
    hashtabel.put("Brian",new Date(70,3,1));//  1. april 1970
    hashtabel.put("Bo",new Date(76,6,9));// 9. juli 1976

    // Lav nogle opslag i tabellen under forskellige navne
    dato = (Date) hashtabel.get("Troels");
    System.out.println( "Opslag under Troels giver: "+dato);

    System.out.println( "Opslag under Jacob:     "+hashtabel.get("Jacob"));
    System.out.println( "Opslag under Kurtbørge: "+hashtabel.get("Kurtbørge"));
    System.out.println( "Opslag under Brian:     "+hashtabel.get("Brian"));

    // Gennemløb af alle elementer
    for (Iterator i = hashtabel.keySet().iterator() ; i.hasNext() ;) {
      String nøgle = (String) i.next();
      dato = (Date) hashtabel.get(nøgle);
      System.out.println(nøgle + " har fødselsåret: "+dato.getYear());
    }
  }
}
```

```
xxx udskrift genereres igen
Opslag under 'Troels' giver: Fri Aug 11 00:00:00 GMT+00:00 1972
.. og under Jacob: Fri Jan 01 00:00:00 GMT+00:00 1971
.. Kurtbørge: null
.. Eva: Mon Mar 05 00:00:00 GMT+00:00 1973
Jacob's fødselsår: 71
```

```
Ulla's fødselsår: 69
Eva's fødselsår: 73
Troels's fødselsår: 72
```

De sidste linjer i koden viser, hvordan man gennemløber alle indgangene i en afbildning. Vi kan ikke, som med en liste, bare lave en for−løkke, der løber fra 0 til antal elementer. I stedet skaffer vi mængden af alle nøgler med hashtabel.keySet() og kalder derefter .iterator() for at få et Iterator−objekt, der gennemløber nøglerne, hvorefter vi for hver nøgle fremfinder værdien[2].

Bemærk hvordan en HashMap ikke husker rækkefølgen af indgangene. Derfor er rækkefølgen, som elementerne bliver udskrevet i (Jacob, Ulla, Brian, Troels, Ulrik ogBo xxx tjek), ikke den samme som den rækkefølge, de blev sat ind i (Jacob, Troels, Ulrik, Ulla, Brian og Bo).

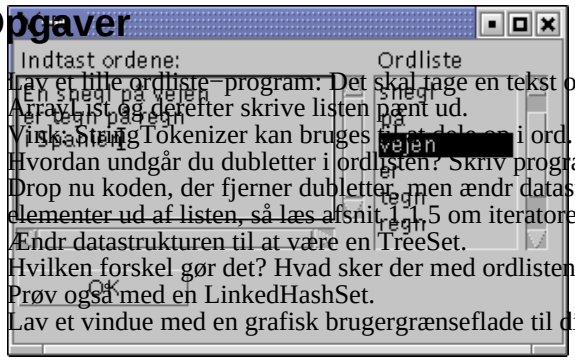
1.2.5 Avanceret: LinkedHashMap i JDK 1.4

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

1.2.6 Interfacet SortedMap (sorterede afbildninger) og klassen TreeMap

Dette er en afbildning, hvor nøglerne er sorteret på samme måde som SortedSet. Klassen TreeMap er den eneste, der implementerer SortedMap.

1.3 Opgaver



1. Lav et lille ordliste−program: Det skal tage en tekst og finde frem til de ord, der indgår i teksten, gemme dem i en ArrayList og derefter skrive listen pænt ud.
   Vink: StringTokenizer kan bruges til at dele en i ord.
2. Hvordan undgår du dubletter i ordlisten? Skriv programmet om så det fjerner dubletter.
3. Drop nu koden, der fjerner dubletter, men ændr datastrukturen til at være en HashSet (har du problemer med at hente elementer ud af listen, så læs afsnit 1.1.5 om iteratorer). Hvilken forskel gør det? Hvad sker der med ordlisten?
4. Ændr datastrukturen til at være en TreeSet.
   Hvilken forskel gør det? Hvad sker der med ordlistens rækkefølge?
   Prøv også med en LinkedHashSet.
5. Lav et vindue med en grafisk brugergrænseflade til dit ordliste−program:

   Når brugeren trykker på OK−knappen, skal ordlisten opdateres med de ord, den ikke allerede indeholder.
6. Læs kildeteksten til Vector, og prøv at forstå hvordan, den fungerer. Den kan findes i din java−installation (i filen src.zip) eller i afsnit 1.6.8. Studér især de metoder, du er fortrolig med, som elementAt(), addElement(), insertElementAt() og removeElementAt().

1.4 Videre læsning

En introduktion til samlinger af data kan findes på
http://developer.java.sun.com/developer/onlineTraining/collections/

En anden, mere opgave−orienteret vejledning kan findes på adressen
http://java.sun.com/docs/books/tutorial/collections/index.html

1.5 Appendiks

Her er en komplet oversigt over og forklaring på alle samlingerne af data og deres metoder.

1.5.1 Collection

*Interfacet **Collection** − fællesnævnerne for alle samlinger af data (lister og mængder)*

boolean **add**(Object) føjer et element til samlingen

boolean **remove**(Object) fjerner et element, hvis det findes i samlingen

boolean **addAll**(Collection) tilføjer elementerne fra en anden samling (foreningsmængde)

boolean **removeAll**(Collection) fjerner alle elementer, der findes i en anden samling

boolean **retainAll**(Collection) fjerner alt, der ikke også er i en anden samling (fællesmængde)

ovenstående metoder returnerer true, hvis data ændredes af operationen, false hvis samlingen var upåvirket

void **clear**() tømmer samlingen for alle elementer

int **size**() giver antal elementer

boolean **isEmpty**() returnerer true, hvis der er 0 elementer

boolean **contains**(Object) returnerer true, hvis elementet findes

boolean **containsAll**(Collection) returnerer true, hvis alle elementerne findes

Iterator **iterator**() giver en iterator (se afsnit 1.5.6)

Object[] **toArray**() giver et array med alle elementerne

Object[] **toArray**(Object[]) lægger alle elementer med en given type i et eksisterende array

boolean **equals**(Object) undersøger lighed (samme type samling med samme elementer)

1.5.2 Set

*Interfacet **Set** repræsenterer en mængde. Det arver fra Collection, men tilføjer ingen nye metoder.*

*Klassen **HashSet** implementerer Set (og dermed Collection), Cloneable og Serializable og har konstruktørerne:*

HashSet() opretter tom mængde med startkapacitet på ca. 10 elementer

HashSet(int startkapacitet) opretter tom mængde med specificeret startkapacitet

HashSet(int startkapacitet, float maxFyldningsfaktor) fyldningsfaktor =antal elementer/kapacitet

HashSet(Collection) opretter mængde med alle elementer fra en anden samling

Object **clone**() laver en overfladisk kopi (referencerne kopieres, ikke elementerne)

*Med JDK1.4 kom klassen **LinkedHashSet**, der tillader hurtigere gennemløb af nøglerne, men ellers ikke adskiller sig fra HashSet.*

1.5.3 SortedSet

*Interfacet **SortedSet** er en ordnet mængde. Det arver fra Set (og dermed fra Collection) og tilføjer metoderne:*

Object **first**() giver første element i mængden

Object **last**() giver sidste element i mængden

SortedSet **tailSet**(Object fra) giver alle elementer i mængden fra (og evt. med) elementet *fra*

SortedSet **headSet**(Object til) giver alle elementer i mængden, der kommer før elementet *til*

SortedSet **subSet**(Object fra, Object til) giver kombinationen af headSet() og tailSet()

Comparator **comparator**() giver ordningen (eller null, hvis standardordning benyttes)

*Klassen **TreeSet** implementerer SortedSet (og dermed Set), Cloneable og Serializable og har konstruktørerne:*

TreeSet() opretter tom mængde

TreeSet(Comparator) opretter tom mængde med specificeret ordning

TreeSet(Collection) opretter mængde med alle elementer fra en samling af data

TreeSet(SortedSet) opretter mængde med alle elementer fra en anden ordnet mængde

Object **clone**() laver en overfladisk kopi (referencerne kopieres, ikke elementerne)

1.5.4 List

*Interfacet **List** repræsenterer en liste (hvor dubletter er tilladt). Det arver fra Collection og tilføjer metoderne:*

void **add**(int, Object) indsætter element på bestemt plads i listen

Object **get**(int) aflæser, hvilket element der er på en bestemt plads i listen

Object **set**(int, Object) erstatter element på en bestemt plads (erstattet element returneres)

Object **remove**(int) fjerner element på en bestemt plads (fjernet element returneres)

boolean **addAll**(int, Collection) indsætter alle elementer fra en samling af data på en bestemt plads

int **indexOf**(Object) finder første position i listen, hvor et element optræder

int **lastIndexOf**(Object) finder sidste position i listen, hvor et element optræder

ListIterator **listIterator**() giver en iterator (se afsnit 1.5.6) med udvidede muligheder

ListIterator **listIterator**(int) giver en iterator (se afsnit 1.5.6), der starter fra en bestemt plads

List **subList**(int, int) giver en delliste (ændres i dellisten, ændres også i oprindelige liste)

*Klassen **ArrayList** implementerer List (og dermed Collection), Cloneable, Serializable og har derudover:*

ArrayList() opretter tom liste med startkapacitet på ca. 10 elementer

ArrayList(int) opretter tom liste med specificeret startkapacitet

ArrayList(Collection) opretter liste med alle elementer fra en anden samling af data

void **trimToSize**() trimmer listen, så kapaciteten lige netop kan rumme elementerne

void **ensureCapacity**(int kap) sikrer at kapacitet (antal elementer, listen kan rumme før den er nødt til at udvide sig) er mindst *kap*

Object **clone**() laver en overfladisk kopi (referencerne kopieres, ikke elementerne)

*Klassen **LinkedList** implementerer List (og dermed Collection), Cloneable, Serializable og har derudover:*

LinkedList() opretter en tom liste

LinkedList(Collection) opretter en liste med alle elementer fra en anden samling af data

void **addFirst**(Object) tilføjer først i listen

void **addLast**(Object) tilføjer sidst i listen

Object **getFirst**() giver første element

Object **getLast**() giver sidste element

Object **removeFirst**() fjerner første element (fjernet element returneres)

Object **removeLast**() fjerner sidste element (fjernet element returneres)

Object **clone**() laver en overfladisk kopi (referencerne kopieres, ikke elementerne)

1.5.5 Map

*Interfacet **Map** − fællesnævnerne for alle associative tabeller (afbildninger)*

Object **put**(Object n, Object v) tilføjer *v* under nøglen *n* (evt. erstattet værdi returneres)

Object **get**(Object nøgle) finder værdien under *nøgle* og returnerer værdien

Object **remove**(Object nøgle) fjerner værdien under *nøgle* (evt. fjernet værdi returneres)

boolean **containsKey**(Object n) returnerer sand, hvis nøglen *n* findes i afbildningen

boolean **containsValue**(Object v) returnerer sand, hvis værdi *v* findes i afbildningen

void **putAll**(Map) føjer alle indgange fra en anden afbildning til denne

int **size**() returnerer antallet af indgange

boolean **isEmpty**() er sand, hvis afbildningen er tom (indeholder 0 indgange)

void **clear**() tømmer afbildningen for alle elementer

boolean **equals**(Object) undersøger lighed (afbildning med de samme nøgle−værdi−par)

Set **keySet**() giver mængde med alle nøgler

Collection **values**() giver samlingen af værdierne (der kan være dubletter)

Set **entrySet**() giver mængde med nøgle−værdi−par (se Map.Entry nedenfor)

*Klassen **HashMap** implementerer Map, Cloneable, Serializable og har konstruktørerne:*

HashMap() opretter tom afbildning med startkapacitet på ca. 10 elementer

HashMap(int startkapacitet) opretter tom afbildning med specificeret startkapacitet

HashMap(int startkapacitet, float maxFyldfaktor) fyldfaktor =antal elementer/kapacitet

HashMap(Map) opretter afbildning med alle indgange fra en anden afbildning

Object **clone**() laver en overfladisk kopi (referencer kopieres, ikke elementer)

*Interfacet **SortedMap** er en afbildning, der er ordnet efter nøglerne. Det arver fra Set og tilføjer metoderne:*

Object **firstKey**() giver første nøgle i afbildningen

Object **lastKey**() giver sidste nøgle i afbildningen

SortedMap **tailMap**(Object fra) giver en afbildning med alle nøgler fra (og evt. med) *fra*

SortedMap **headMap**(Object til) giver en afbildning med alle nøgler der kommer før *til*

SortedMap **subMap**(Object fra, Object til) kombinationen af tailMap() og headMap()

Comparator **comparator**() giver ordningen (null hvis standardordning benyttes)

*Klassen **TreeMap** implementerer SortedMap (og dermed Map), Cloneable, Serializable og har konstruktørerne:*

TreeMap() opretter tom afbildning

TreeMap(Comparator) opretter tom afbildning med specificeret ordning

TreeMap(Map) opretter afbildning med alle indgange fra en anden afbildning

TreeMap(SortedMap) opretter afbildning fra en anden ordnet afbildning

Object **clone**() laver overfladisk kopi (referencerne kopieres, ikke elementerne)

*Metoden entrySet() giver mængden af alle indgangene i afbildningen, repræsenteret ved interfacet **Map.Entry**:*

Object **getKey**() giver nøglen

Object **getValue**() giver værdien

Object **setValue**(Object) sætter værdien (erstattet værdi returneres)

1.5.6 Iteratorer

En iterator er et lille objekt, der hjælper en med at gennemløbe nogle data (se afsnit 1.1.5). Man kan man få et iterator−objekt ved at kalde metoden iterator() på enhver samling af data:

```
Iterator iter = samling.iterator();
while (iter.hasNext())
{
  String s = (String) iter.next();
  // gør noget med s
}
```

*Interfacet **Iterator**:*

boolean **hasNext**() returnerer sand, hvis der er flere elementer

Object **next**() går til næste element og returnerer det

void **remove**() fjerner aktuelle element fra samlingen (ikke altid implementeret)

Til lister (af typen List) kan man få et udvidet iterator−objekt ved at kalde listIterator():

*Interfacet **ListIterator** arver fra Iterator og har derudover metoderne:*

boolean **hasPrevious**() returnerer sand, hvis der er tidligere elementer

Object **previous**() går til næste element og returnerer det

int **nextIndex**() giver indeks på element, der ville returneres af next()

int **previousIndex**() giver indeks på element, der ville returneres af previous()

void **set**(Object) erstatter det aktuelle element i listen med et andet

void **add**(Object) indsætter element lige før næste element i listen

Når en iterator bruges på en samling, er det ulovligt at ændre samlingen strukturelt (f.eks. indsætte eller slette elementer) direkte, da iteratoren derved kommer ud af trit med samlingen. Kommer en iterator ud af trit med sin samling, er den ubrugelig og vil kaste undtagelsen ConcurrentModificationException, hvis next() nogen sinde kaldes på den igen.

Ønsker man at ændre i samlingen samtidig med at man bruger en iterator, bør man derfor bruge iteratorens ændringsmetoder.

Enumeration

Da Iterator og ListIterator kom først til med JDK 1.2 bruges I standardbiblioteket ofte som iterator et Enumeration−objekt (på dansk 'opremsning') i stedet, f.eks.:

```
Enumeration enum = vektor.elements();
while (enum.hasMoreElements())
{
  String s = (String) enum.nextElement();
  // gør noget med s
}
```

*Interfacet **Enumeration** − har eksisteret længere end Iterator og findes derfor ofte brugt i standardbiblioteket*

boolean **hasMoreElements**() returnerer sand, hvis der er flere elementer

Object **nextElement**() går til næste element og returnerer det

1.5.7 Manipulation og sortering af data

Der findes en række metoder til at manipulere og sortere array af data i klassen Arrays.

*Klassen **Arrays** − alskens metoder til at arbejde med array.*

static void **sort**(double*[]) sorterer array af simpel type (også m. byte, int, ...)

static void **sort**(double*[], int fra, int til) sorterer arrayet fra og med plads nr. *fra* til plads nr. *til*

static void **sort**(Object[]) sorterer array af objekter

static void **sort**(Object[], int fra, int til) sorterer arrayet fra og med plads nr. *fra* til plads nr. *til*

static void **sort**(Object[], Comparator) sorterer array af objekter efter en bestemt ordning

static void **sort**(Object[], int, int, Comparator) sorterer arrayet fra og med plads nr. *fra* til plads nr. *til*

static int **binarySearch**(double*[], double*) søger (binært) i allerede sorteret array af simpel type

static int **binarySearch**(Object[], Object) søger (binært) i allerede sorteret array af objekter

static int **binarySearch**(Object[], Object, Comparator) søger i array sorteret med bestemt ordning

static boolean **equals**(double*[], double*[]) lighed af to array af simpel type (også m. byte, int, ...)

static boolean **equals**(Object[], Object[]) undersøger, om to array har de samme objekter

static void **fill**(double*[], double*) sætter alle pladser til en bestemt værdi (også m. int..)

static void **fill**(double*[], int, int, double*) sætter værdier fra og med plads nr. *fra* til plads nr. *til*

static void **fill**(Object[], Object) sætter alle pladser i arrayet til et bestemt objekt

static void **fill**(Object[], int, int, Object) fylder med objekt fra og med plads nr. *fra* til nr. *til*

static List **asList**(Object[]) laver en liste ud fra et array af objekter

**) betyder, at metoden findes både med parameter af typen boolean, byte, char, int, float og double*

Her er et eksempel på brug af nogle af funktionerne:

```
import java.util.*;

public class BenytArrays
{
```

```
  public static void main(String[] args)
  {
    int[] tal = new int[20];
    Arrays.fill(tal,42);        // sæt alle elementerne i arrayet til 42
    Arrays.fill(tal,5,15,44);   // sæt plads 5 til og med plads nr. 14 til 44
    tal[10] = 43;               // sæt plads nr 10 til 43
    Arrays.sort(tal);           // sortér array (for f.eks. at kunne søge i det)

    int pos44 = Arrays.binarySearch(tal,44); // hvad er nu index på tallet 44?
    System.out.println( "Tallet 44 er på plads nr: " + pos44 );
  }
}
```

```
Tallet 44 er på plads nr: 14
```

Bemærk, at vil man kopiere et array af objekter gøres det mest effektivt med metoden System.arraycopy().

Ligeledes findes metoder til at manipulere og sortere lister, mængder og afbildninger:

*Klassen **Collections** − alskens metoder til at arbejde med samlinger af data.*

static final Set **EMPTY_SET** repræsenterer en tom mængde
static final List **EMPTY_LIST** repræsenterer en tom liste
static final Map **EMPTY_MAP** repræsenterer en tom afbildning

static void **sort**(List) sorterer listen
static void **sort**(List, Comparator) sorterer listen efter en bestemt ordning

static int **binarySearch**(List, Object) giver indeks på et objekt ved binær søgning i en sorteret liste

static int **binarySearch**(List, Object, Comparator) søger i en sorteret (med bestemt ordning) liste

static void **reverse**(List) vender rækkefølgen af alle elementerne i listen om

static void **shuffle**(List) blander listen, så elementerne kommer i tilfældig rækkefølge

static void **shuffle**(List, Random) blander listen v.hj.a. et specificeret Random−objekt

static void **fill**(List, Object) erstatter alle pladser i listen med det samme element

static void **copy**(List til, List fra) kopierer liste. Til−listen skal have plads til elementerne.

static Object **min**(Collection) finder mindste element

static Object **min**(Collection, Comparator) finder mindste element (i forhold til en bestemt ordning)

static Object **max**(Collection) finder største element

static Object **max**(Collection, Comparator) finder største element (i forhold til en bestemt ordning)

static Collection **unmodifiableCollection**(Collection) laver en uforanderlig samling
static Set **unmodifiableSet**(Set) laver en uforanderlig mængde
static SortedSet **unmodifiableSortedSet**(SortedSet) laver en uforanderlig sorteret mængde
static List **unmodifiableList**(List) laver en uforanderlig liste
static Map **unmodifiableMap**(Map) laver en uforanderlig afbildning
static SortedMap **unmodifiableSortedMap**(SortedMap) laver uforanderlig sorteret afbildning

static Collection **synchronizedCollection**(Collection) laver en trådsikker samling
static Set **synchronizedSet**(Set) laver en trådsikker mængde
static SortedSet **synchronizedSortedSet**(SortedSet) laver en trådsikker sorteret mængde
static List **synchronizedList**(List) laver en trådsikker liste
static Map **synchronizedMap**(Map) laver en trådsikker afbildning
static SortedMap **synchronizedSortedMap**(SortedMap) laver en trådsikker sorteret afbildning

static Set **singleton**(Object) laver en uforanderlig mængde med ét element

static List **singletonList**(Object) laver en uforanderlig liste med ét element

static Map **singletonMap**(Object, Object) laver en uforanderlig afbildning med én indgang

static List **nCopies**(int n, Object elem) laver en uforanderlig liste med *n* kopier af *elem*

static Comparator **reverseOrder**() giver den omvendte ordning af den naturlige ordning

static Enumeration **enumeration**(Collection) giver en opremsning (Enumeration) af en samling

1.6 Avanceret

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

1.6.1 Store−O−notationen

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

1.6.2 Arraybaserede listers virkemåde

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

1.6.3 Hægtede listers virkemåde

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

1.6.4 Hashtabellers virkemåde

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

1.6.5 Søgetræers virkemåde

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

1.6.6 Uforanderlige samlinger

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

1.6.7 Trådsikre samlinger

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

1.6.8 Uddrag af kildeteksten til Vector−klassen

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

1Der er egentlig en tredje metode i Iterator, remove(), der fjerner det element, der sidst blev returneret af next(), men den bliver ikke behandlet her. Til lister (af typen List) kan man få et udvidet iterator−objekt ved at kalde listIterator(). Disse ting diskuteres i appendiks 1.5.6.

2Til udskrivning af hele indholdet af en hashtabel er det lidt hurtigere at hente mængden af alle indgange (med både nøgle og værdi) ved at kalde hashtabel.entrySet(). Se afsnit 1.5.5.

3En nærmere diskussion af, hvad en proxy og et uforanderligt objekt er, findes i hhv. afsnit 17.1 og afsnit 18.1.

2 Dokumentation

En overfladisk forståelse af dette kapitel er en fordel i resten af bogen.

Tak til Jonas Kongslund og Skåne Sjælland Linux−brugergruppe, der har doneret udgangspunktet til kapitlet, fra http://www.linuxbog.dk/java/bog/.

Dokumentation er et vigtigt aspekt af ethvert programudviklingsprojekt. Dokumentationen kan have mange former og modtagere. Det kan f.eks. være:

- Henvendt til slutbrugeren og vejlede i, hvordan programmet bruges (brugervejledning)
- Henvendt til driftsafdelingen/systemadministratoren og vejlede i:
- Udseende og fortolkning af logfiler
- Hvordan programmet skal bringes til at fungere igen, hvis det stopper
- Hvilke forudsætninger der skal gælde (f.eks. hvilke andre programmer der skal være installeret og i hvilke versioner), for at programmet fungerer
- Hvordan programmet installeres og konfigureres
- Henvendt til en anden programmør som dokumentation af et færdigt program, som programmøren skal vedligeholde:
- Testtilfælde (inddata og forventede uddata)
- Analyse− og designdokumenter, herunder klassediagrammer
- Beskrivelse af klasser, metoder og variabler (javadoc)
- Henvendt til en anden programmør som dokumentation af et API−programbibliotek som skal bruges fra et program:
- Klassediagrammer
- Beskrivelse af klasser, metoder og variabler (javadoc)

Dette kapitel beskriver hvordan, man kan udarbejde dokumentation med javadoc.

2.1 Javadoc

Javadoc er en beskrivelse af alle pakker, klasser, interfaces, metoder og variabler, samt eventuelt vejledning i, hvordan de bør bruges. Den kan genereres ud fra kommentarer i kildeteksten ved hjælp af et udviklingsværktøj eller ved at køre javadoc fra kommandolinjen (DOS−prompten).

2.1.1 API−dokumentation

Et API (Application Programming Interface) er et sæt klasser, som man kan bruge i sit program, og som letter programmørens arbejde. F.eks. giver JDBC−klasserne let adgang til forskellige databaser på en ensartet måde.

Dokumentationen af et API kan opfattes som en kontrakt mellem den programmør, der har implementeret (skrevet) API'et (leverandøren), og den programmør, der vil bruge API'et (klienten).

Hvis leverandøren f.eks. lover at metoden double sqr(int tal) returnerer kvadratroden af tallet, når det ikke er negativt, så ved klienten, at uanset hvordan metoden er implementeret, så vil den altid returnere kvadratroden med så stor præcision, som returtypen tillader.

Hvis returværdien viser sig at være forkert i nogle tilfælde, så har leverandøren brudt kontrakten, og metoden er implementeret forkert.

Hvis klienten kalder metoden med et negativt tal, er der også tale om kontraktbrud. Kontraktbrud kan enten resultere i en undtagelse, eller at resultatet ikke er veldefineret.

2.1.2 At dokumentere med javadoc

Kommentarer i kildeteksten, der starter med `/**` og slutter med `*/`, opfattes som specielle javadoc−kommentarer. Både interfaces, klasser, variabler, konstruktører og metoder kan dokumenteres ved at lave en javadoc−kommentar umiddelbart før det, der skal dokumenteres.

2.1.3 Simpelt eksempel

Her er en klasse med en metode. Begge er kommenteret med javadoc−kommentarer:

```
/**
 * Eksempel på en kommenteret klasse.
 */
public class EnKommenteretKlasse
{
  /**
   * Et eksempel på en metode. Metoden tjener
   * til at vise hvordan javadoc virker.
   *
   * @param enStreng strengen
   * @param etTal tallet
   *
   * @return strengen og tallet sat sammen
   */
  public String enMetode(String enStreng,
                         int etTal)
  {
    return enStreng+etTal;
  }
}
```

Bemærk, hvordan kommentaren, der beskriver klassen, står umiddelbart før erklæringen af klassen:

```
/**
 * Eksempel på en kommenteret klasse.
 */
public class EnKommenteretKlasse
{
...
```

Ligeledes står kommentaren, der beskriver metoden, umiddelbart før erklæringen af metoden:

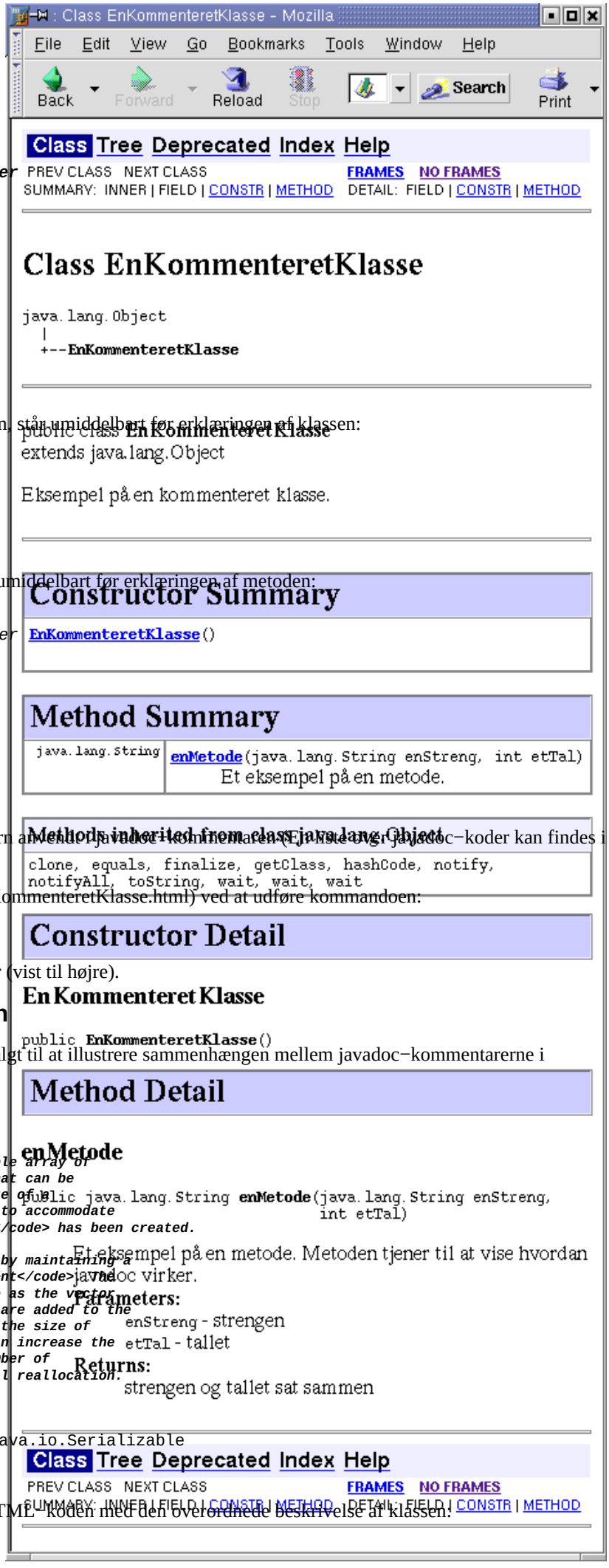
```
  /**
   * Et eksempel på en metode. Metoden tjener
   * til at vise hvordan javadoc virker.
   *
   * @param enStreng strengen
   * @param etTal tallet
   *
   * @return strengen og tallet sat sammen
   */
  public String enMetode(String enStreng,
                         int etTal)
  { ...
```

Her er de specielle javadoc−koder @param og @return anvendt i javadoc−kommentaren (En liste over javadoc−koder kan findes i afsnit 2.3.2).

Nu kan vi generere en HTML−fil (med filnavnet EnKommenteretKlasse.html) ved at udføre kommandoen:

```
javadoc EnKommenteretKlasse.java
```

Resultatet er den velkendte dokumentation for klasser (vist til højre).

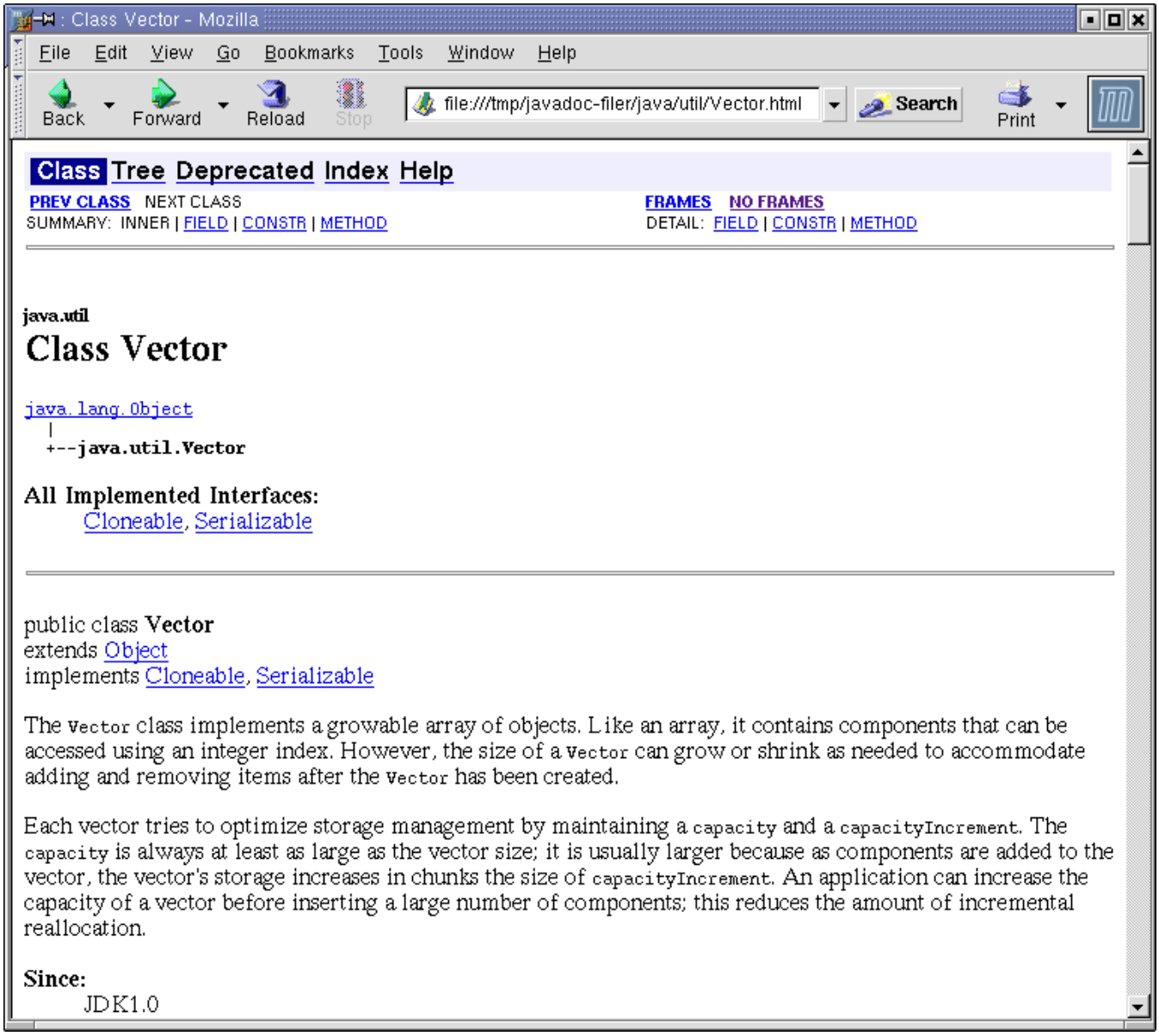


2.1.4 Større eksempel: Vector−klassen

Da Vector−klassen er velkendt for de fleste, er den valgt til at illustrere sammenhængen mellem javadoc−kommentarerne i kildetekst og den genererede HTML−dokumentation:

```
package java.util;

/**
 * The <code>Vector</code> class implements a growable array of
 * objects. Like an array, it contains components that can be
 * accessed using an integer index. However, the size of a
 * <code>Vector</code> can grow or shrink as needed to accommodate
 * adding and removing items after the <code>Vector</code> has been created.
 * <p>
 * Each vector tries to optimize storage management by maintaining a
 * <code>capacity</code> and a <code>capacityIncrement</code>. The
 * <code>capacity</code> is always at least as large as the vector
 * size; it is usually larger because as components are added to the
 * vector, the vector's storage increases in chunks the size of
 * <code>capacityIncrement</code>. An application can increase the
 * capacity of a vector before inserting a large number of
 * components; this reduces the amount of incremental reallocation.
 *
 * @since   JDK1.0
 */
public class Vector implements Cloneable, java.io.Serializable
{
  ... (nogle metoder og variabler)
```

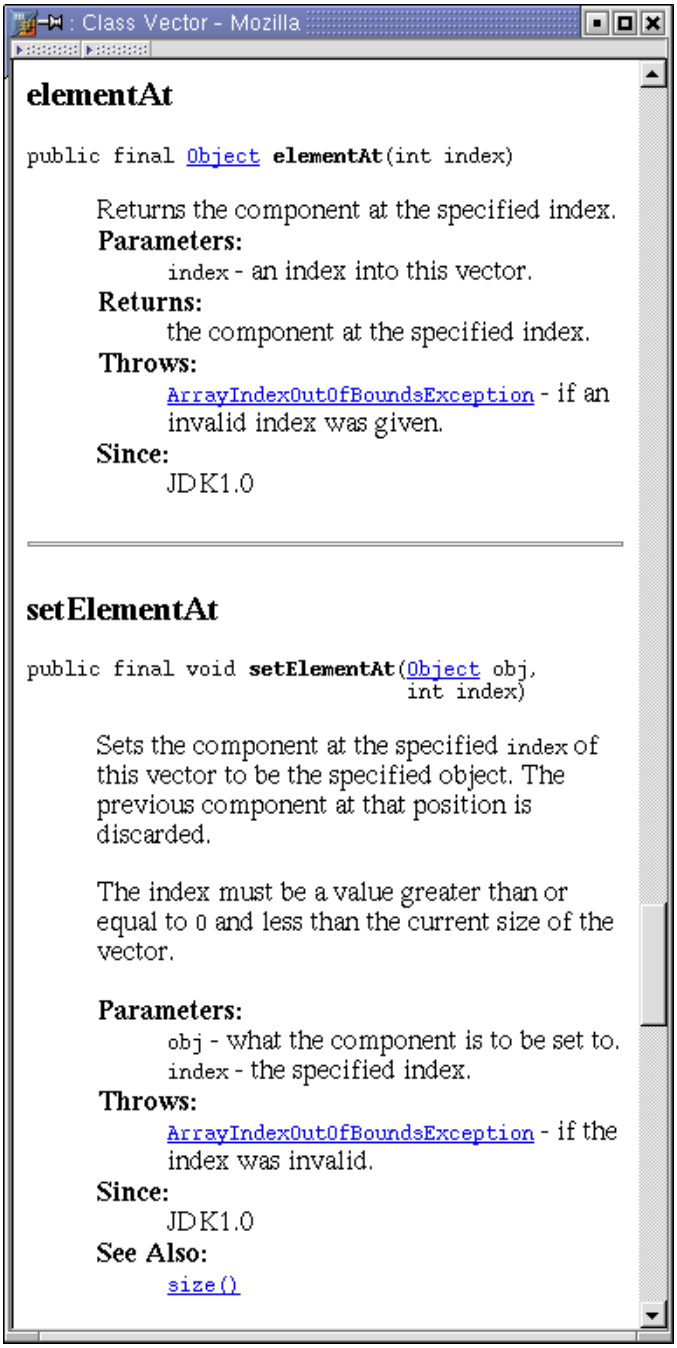
Ud fra denne del af kildeteksten genererer javadoc HTML−koden med den overordnede beskrivelse af klassen:



Bemærk, hvordan der bruges HTML−koder i kildeteksten til at fremhæve blandt andet klassenavne for at øge læsevenligheden af den genererede dokumentation.

Fortsætter vi i kildeteksten til Vector (der findes samlet i afsnit 1.6.8), kan vi se, hvordan javadoc genererer HTML−kode (vist til højre) ud fra beskrivelsen af metoderne og specielle javadoc−koder (beskrevet i afsnit 2.3.2) som @param, @return, @exception, @since og @see:

```
...
/**
```




```
 * Returns the component at the specified index.
 *
 * @param   index   an index into this vector.
 * @return the component at the specified index.
 * @exception ArrayIndexOutOfBoundsException if
 *              an invalid index was given.
 * @since    JDK1.0
 */
public final Object elementAt(int index) {
  if (index >= elementCount) {
    throw new ArrayIndexOutOfBoundsException(index + " >= " + elementCount);
  }
  try {
    return elementData[index];
  } catch (ArrayIndexOutOfBoundsException e) {
    throw new ArrayIndexOutOfBoundsException(index + " < 0");
  }
}

/**
 * Sets the component at the specified <code>index</code> of this
 * vector to be the specified object. The previous component at that
 * position is discarded.
 * <p>
 * The index must be a value greater than or equal to <code>0</code>
 * and less than the current size of the vector.
 *
 * @param   obj what the component is to be set to.
 * @param   index the specified index.
 * @exception ArrayIndexOutOfBoundsException if the index was invalid.
 * @see    java.util.Vector#size()
 * @since    JDK1.0
 */
```

```
public final void setElementAt(Object obj, int index) {
  if (index >= elementCount) {
    throw new ArrayIndexOutOfBoundsException(index + " >= " +
                elementCount);
  }
  elementData[index] = obj;
}

...
```

2.1.5 Oversigter

Mellem den overordnede klassebeskrivelse og denne (detaljerede) beskrivelse af metoderne findes en oversigt over variabler, konstruktører og metoder.

I oversigter står generelt *kun teksten indtil det første punktum i javadoc−kommentarerne*. F.eks. står der i oversigten over metoder ('method summary') for setElementAt() følgende:

void	**setElementAt**(Object obj, int index) Sets the component at the specified index of this vector to be the specified object.

Resten af javadoc−kommentaren (efter punktummet) er altså udeladt i oversigten (sammenlign med kommentaren til setElementAt() ovenfor).

Det er derfor smart at lade den første sætning i hver kommentar være beskrivende nok til, at man hurtigt får en idé om, hvad eksempelvis en metode gør.

2.2 Kørsel af javadoc

Javadoc kan køres fra kommandolinjen, men er også indbygget i de fleste java−udviklingsmiljøer (kunne f.eks. være under "Tools" / "Generate Javadoc").

I det følgende vil vi koncentrere os om kørsel fra kommandolinjen. Det forudsættes, at læseren er nogenlunde bekendt med kommandolinjen (f.eks. ved, hvordan man skifter katalog).

Får du fejlen "javadoc: Forkert kommando eller filnavn", når du skriver 'javadoc' på kommandolinjen, er det, fordi java−udviklingskittet (JDK'en) ikke er med i stien. Du kan da:

- Enten skrive den fulde sti, f.eks.:
  `/usr/local/jdk1.4/bin/javadoc` (UNIX/Linux) eller
  `\jdk1.4\bin\javadoc` (DOS/Windows)
- Eller ændre stien (PATH−variablen), f.eks.:
  `export PATH=/usr/local/jdk1.4/bin/:$PATH` (UNIX/Linux) eller
  `set PATH=\jdk1.4\bin\;%PATH%` (DOS/Windows)

2.2.1 Simpel brug

Det simpleste er at skifte katalog til der, hvor .java−filerne er og skrive:

```
javadoc *.java
```

Herefter vil HTML−filerne blive genereret og ligge i det aktuelle katalog (har man nogen pakker, vil HTML−filerne ligge i de tilsvarende underkataloger).

Ønsker man, at de genererede HTML−filer skal lægges andetsteds, kan det specificeres med parameteren −d og stien, f.eks.:

```
javadoc −d /tmp/javadoc−filer *.java
```

hvorefter HTML'en lægges i kataloget /tmp/javadoc−filer[1].

Parameteren '*.java' tager alle de .java−filer, der findes i det aktuelle katalog.

Hvis man arbejder med pakker, kan man også blot nævne pakkens navn. Har man f.eks. defineret pakken 'vp', kan man skrive (fra kilde−rodkataloget, dvs. overkataloget til 'vp')

```
javadoc vp
```

hvorefter der bliver genereret dokumentation for alle vp−pakkens klasser (alle .java−filer i det tilsvarende underkatalog) og eventuelle underpakker.

De vigtigste kommandolinje−parametre til javadoc er beskrevet i afsnit 2.3.1.

2.2.2 Henvise til den eksisterende javadokumentation

Ofte (faktisk altid) vil ens metoder returnere (eller have som parametre) nogle objekter fra standardbiblioteket. I eksemplet med EnKommenteretKlasse tager metoden en streng, men klassen java.lang.String har vi ikke selv lavet, og derfor er der ikke nogen

henvisning til dokumentation for den, da javadoc som udgangspunkt genererer HTML−dokumentation, der ikke henviser til eksterne kilder.

Det kan dog laves om ved at fortælle javadoc, hvor den eksterne dokumentation befinder sig, med parameteren −link, f.eks.:

```
javadoc −link http://java.sun.com/j2se/1.3/docs/api/ *.java
```

Herefter vil der blive henvist til javas standarddokumentation på Suns hjemmeside, hver gang javadoc møder returtyper, parametertyper, superklasser/implementerede interfaces og kastede undtagelser fra standardbiblioteket.

2.3 Appendiks

2.3.1 Parametre til javadoc

Den generelle brug af javadoc er:

```
javadoc [tilvalg] [pakkenavne] [kildetekstfiler] [klassenavne]
```

hvor de vigtigste tilvalg (flag) er beskrevet herunder:

Tilvalg til javadoc. En fuld liste over tilvalg fås ved at skrive 'javadoc' fra kommandolinjen.

−public Medtag kun ting (klasser, interfaces, variabler, metoder) erklæret public

−protected Medtag kun ting erklæret protected eller public (standardindstilling)

−package Medtag alt undtagen ting erklæret private

−private Medtag alt

−sourcepath <sti−liste> Angiv, hvor .java−filer med kildeteksten kan findes

−classpath <sti−liste> Angiv, hvor binære .class−filer i øvrigt kan findes

−d <directory> Destinationskatalog for de genererede HTML−filer

−use Medtag oversigter over, hvor klasser og pakker bruges henne

−version Medtag @version−afsnit

−author Medtag @author−afsnit

−link <url> Opret eksterne henvisninger til eksisterende javadoc på <url>

−nodeprecated Ignorér @deprecated (frarådede metoder/klasser)

−nosince Ignorer @since (hvornår noget blev indført i programmet)

−nodeprecatedlist Opret ikke liste over frarådede ting

−notree Opret ikke et træ over klassehierarkiet

−noindex Opret ikke en indholdsfortegnelse over klasserne

−nohelp Opret ikke en henvisning til hjælp

−nonavbar Opret ikke en navigations−bjælke

2.3.2 Koder tilgængelige i javadoc

I eksemplet er der knyttet Javadoc−kommentarer til metoder og klassevariabler samt klassen selv. I de fleste af kommentarerne anvendes der Javadoc−koder.

- @author angiver ophavsmanden til klassen. Det er muligt at angive flere @author−koder på separate linjer, såfremt der er flere ophavsmænd.
- @version, der er obligatorisk, angiver versionsnummeret og er for Javadoc−kommentarernes vedkommende beregnet til klasser og interfaces. Versionsnummeret har ikke nogen speciel betydning og kan derfor være hvad som helst.
- @since angiver, fra hvilken version tilføjelsen fandt sted, og kan bruges overalt. Versionsnummeret har ikke nogen speciel betydning og kan derfor være hvad som helst.
- @param beskriver en parameter og er beregnet til metoder og konstruktører. Først angives parameterens navn og dernæst beskrivelsen.
- @return beskriver returværdien af en metode.
- @exception gør det samme som @throws.
- @throws beskriver en undtagelse og er beregnet til metoder og konstruktører. Der kan være flere @throws−koder. Efter hver @throws skal stå klassenavnet på undtagelsen og en beskrivelse af hvornår denne undtagelse opstår.

- @see henviser til en pakke, interface, klasse, variabel, konstruktør eller metode. En af de mulige henvisningsformer er @see pakkenavn.klassenavn#medlem, hvor medlem kan være et metodenavn med parametertyper eller et variabelnavn. Pakkenavn og klassenavn kan udelades, hvis der henvises til en metode eller variabel indenfor samme klasse.
- @deprecated kan bruges overalt og angiver, at en metode (eller en variabel, klasse, interface, ...) ikke længere bør anvendes (og muligvis vil udgå i en senere version). Det er anbefalelsesværdigt at henvise til et alternativ ved hjælp af @see.
- @serial kan bruges til at komme med bemærkninger om serialiseringen af klassen.

2.3.3 Pakkekommentarer

En pakkekommentar laves ved at oprette en HTML−fil kaldet package.html, der placeres i pakkens katalog. Hvis pakken f.eks. hedder 'vp', så placeres filen i underkataloget 'vp' (der hvor .java−filerne også ligger), hvorefter javadoc sørger for at medtage filen.

Her er et eksempel på en pakkekommentar:

```
<html>
  <head><title>package</title></head>
  <body>
    Dette er et eksempel på en pakkekommentar.
    Den skal ligge i filen package.html sammen med .java−filerne den beskriver.

    @author Jacob Nordfalk
    @since 1.0
  </body>
</html>
```

Javadoc bruger kun det, der står mellem <body> og </body>, så titlen kan være hvad som helst. Det er muligt at anvende visse Javadoc−koder i pakkekommentarer, nemlig @author, @version, @since, @deprecated og @see.

Bemærk, at HTML−filen med pakkekommentarer ikke må indeholde /** ... */, og at en linje ikke må starte med *.

2.4 Videre læsning

- På UNIX/Linux−systemer giver 'man javadoc' en meget udførlig vejledning
- Javadoc−værktøjets hjemmeside, http://java.sun.com/j2se/javadoc/
- Råd og vejledning til god praksis omkring at skrive javadoc−kommentarer: http://java.sun.com/j2se/javadoc/writingdoccomments/

2.5 Opgaver

1. Kig i kildeteksten til Vector−klassen (den kan findes i afsnit 1.6.8 eller i din java−installation, ofte i filen src.zip), og studér metoderne insertElementAt() og toArray().
   Sammenlign med javadokumentationen (hjælpen) til Vector, og find ud af, hvad koderne @param, @exception, @see, @return og @since gør.
2. Hent klassen SenderActionEvent fra afsnit 4.3.5 ned til dig selv, og generér dens HTML−dokumentation.
3. Medtag henvisninger til standarddokumentationen med parameteren −link
   (beskrevet i afsnit 2.2.2, Henvise til den eksisterende javadokumentation).

2.6 Løsninger

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

1Under UNIX/Linux. Under DOS/Windows skriver man f.eks. c:\tmp\javadoc−filer

3 Rekursion

Forståelse af rekursion er en forudsætning for at forstå et eksempel i afsnit 4.7.3. Derudover forudsættes kapitlet ikke i resten af bogen.

Rekursion er velegnet, hvis en opgave kan deles op i mindre tilsvarende delopgaver.

3.1 Introduktion til rekursive algoritmer

Hvis en metode kalder sig selv, er der tale om rekursion. F.eks.:

```
public class Rekursion
{
  public static void main(String[] arg)
  {
    tælNed(3);
  }

  public static void tælNed(int tæller)
  {
    System.out.print(tæller+" ");
    if (tæller>0) tælNed(tæller-1); // tælNed() kalder sig selv !!
  }
}
```

```
3 2 1 0
```

Fidusen er, at tæller−parameteren eksisterer én gang for hver gang, tælNed() bliver kaldt. Så når tælNed() vender tilbage til kalderen, som også er tælNed(), er tællers værdi bevaret som før kaldet.

3.1.1 Folde en rekursion ud

Er man uvant med rekursion, kan det være svært at gennemskue, hvad der sker. Husk da, at et kald til en metode er uafhængigt af, om metoden eventuelt allerede "er i gang med" at blive kaldt. Ovenstående rekursion kunne "foldes ud" til følgende program:

```
public class RekursionUdfoldet
{
  public static void main(String[] arg)
  {
    tælNed(3);
  }

  public static void tælNed(int tæller)
  {
    System.out.print(tæller+" ");
    if (tæller>0) tælNedA(tæller-1);    // kald tælNedA(2)
  }

  public static void tælNedA(int tæller)
  {
    System.out.print(tæller+" ");
    if (tæller>0) tælNedB(tæller-1);    // kald tælNedB(1)
  }

  public static void tælNedB(int tæller)
  {
    System.out.print(tæller+" ");
    if (tæller>0) tælNedC(tæller-1);    // kald tælNedC(0)
  }

  public static void tælNedC(int tæller)
  {
    System.out.print(tæller+" ");
    if (tæller>0) tælNedC(tæller-1);    // kalder ikke videre, da tæller=0
  }
}
```

```
3 2 1 0
```

Det er klart, at rekursion nemt kan føre til uendelige løkker, hvis man ikke passer på.

3.1.2 Forudsætninger for rekursion

Man kan groft sagt være sikker på, at der ikke opstår en uendelig løkke, hvis man kan påvise, at alle videre kald sker med en *mindre* opgave end der blev kaldt med, og at en tilstrækkelig lille opgave bliver udført uden et videre kald.

Herover bliver opgaven mindre og mindre, fordi tæller bliver reduceret med 1 i hvert kald, og fordi tælNed() kun kalder sig selv, hvis tæller er større end 0.

Rekursion kan løse en slags opgave hvis:

1. Opgaven kan deles om i mindre bidder af samme karakter som den oprindelige opgave
2. En tilstrækkelig lille opgave altid kan løses

3.2 Rekursiv listning af filer

Her ses et andet eksempel på rekursion, hvor filer i det aktuelle katalog og alle underkataloger listes rekursivt.

Klassen File, der repræsenterer en fil eller et katalog, kan bruges til at navigere i filsystemet, slette eller omdøbe filer eller kataloger og aflæse eller sætte deres attributter.

Det følgende eksempel lister alle filerne i det aktuelle katalog. Er der nogle underkataloger, listes indholdet af dem også (rekursionen består i, at listKatalog() kalder sig selv).

```
import java.io.*;
public class ListFilerRekursivt
{
  public static void main(String[] arg)
  {
    File kat = new File(".");   // objekt der repræsenterer det aktuelle katalog
    listKatalog(kat);
  }

  private static void listKatalog(File kat)
  {
    File[] filer = kat.listFiles();

    for (int i=0; i<filer.length; i++)
    {
      File f = filer[i];

      if (f.isDirectory())
      {
        System.out.println("Katalog "+f+":");
        listKatalog(f);                          // kald listKatalog() rekursivt
        System.out.println("Katalog "+f+" slut.");
      } else {
        System.out.println(f);                   // udskriv filens navn og sti
      }
    }
  }
}
```

```
./ListFilerRekursivt.class
./ListFilerRekursivt.java
Katalog ./Desktop:
./Desktop/Home.desktop
./Desktop/MandrakeClub.desktop
Katalog ./Desktop/Removable media:
./Desktop/Removable media/Floppy
./Desktop/Removable media/CD-ROM
Katalog ./Desktop/Removable media slut.
Katalog ./Desktop/Affald:
Katalog ./Desktop/Affald slut.
Katalog ./Desktop slut.
Katalog ./tmp:
Katalog ./tmp slut.
```

Programmet er startet fra et katalog, der indeholdt ListFilerRekursivt.java og ListFilerRekursivt.class og underkatalogerne Desktop (der igen indeholdt bl.a. Home.desktop, MandrakeClub.desktop og underkatalogerne Removable media og Affald) og tmp.

3.3 Beregning af et matematisk udtryk

Det følgende program kan analysere en streng og udregne et matematisk udtryk med de fire regnearter, sinus−funktionen og et vilkårligt antal parenteser.

```
public class Formelberegning
{
  /**
   * Finder første position af en operator, f.eks. +, −, * eller /.
   * Går uden om de operatorer, der er inde i en parentes.
```

```
   * Simpel løsning, der ikke tager højde for parenteser: udtryk.indexOf(tegn)
   */
  private static int findUdenforParenteser(char tegn, String udtryk)
  {
    int par = 0;
    for (int i = 0; i<udtryk.length(); i++)
    {
      char t = udtryk.charAt(i);
      if (t == tegn && par==0) return i; // tegn fundet udenfor parenteser!
      else if (t == '(') par++;          // vi går ind i en parentes
      else if (t == ')') par--;          // vi går ud af en parentes
    }
    return -1; // tegnet blev ikke fundet (i hvert fald ikke udenfor parenteser)
  }


  /**
   * Beregner værdien at et udtryk.
   * @param udtryk En streng med udtrykket, der skal beregnes.
   * @return værdien af udtrykket.
   * @throws NumberFormatException hvis udtrykket ikke er gyldigt
   */
  public static double beregn(String udtryk)
  {
    udtryk = udtryk.trim();                  // fjern overflødige blanktegn
    for (int opNr = 0; opNr < 4; opNr++)     // løb gennem de fire regnearter
    {
      char op = "+-*/".charAt(opNr);         // op er nu '+', '-', '*' eller '/'
      int pos = findUdenforParenteser(op,udtryk);
      if (pos > 0)                           // findes op i udtrykket?
      {
        String vs = udtryk.substring(0,pos); // ja, find venstresiden
        String hs = udtryk.substring(pos+1); // find højresiden

        // metoden kalder nu sig selv og analyserer hvert element i strengen
        double vsr = beregn(vs);             // beregn højresidens værdi
        System.out.println("beregn("+vs+") = "+vsr);
        double hsr = beregn(hs);             // beregn venstresidens værdi
        System.out.println("beregn("+hs+") = "+hsr);

        if (op == '+') return vsr + hsr;     // beregn resultat og returnér
        if (op == '-') return vsr - hsr;
        if (op == '*') return vsr * hsr;
        return vsr / hsr;
      }
    }
    // Hvis vi kommer herned kunne der ikke dele op i flere operatorer
    if (udtryk.startsWith("(") && udtryk.endsWith(")"))   // parenteser omkring?
    {
      udtryk = udtryk.substring(1,udtryk.length()-1);     // fjern dem
      return beregn(udtryk);                              // beregn indmad
    }
    if (udtryk.startsWith("sin(") && udtryk.endsWith(")"))// sinus-funktion
    {
      udtryk = udtryk.substring(4,udtryk.length()-1);     // fjern 'sin(' og )
      double resultat = beregn(udtryk);                   // beregn parameteren
      System.out.println("beregn("+udtryk+") = "+resultat);
      return Math.sin(resultat);
    }
    // intet andet fundet - så må det være et tal!
    // (ellers kastes NumberFormatException)
    return Double.parseDouble(udtryk);
  }

  public static void main(String[] arg)
  {
    String formel = "(1+2)*3 - sin(4/5*(6-7))";
    double værdi = beregn(formel);
    System.out.println("Formlen "+formel+" er beregnet til "+værdi);
  }
}
```

Herunder ses udskriften fra programmet ved udregning af "(1+2)*3 − sin(4/5*(6−7))":

```
beregn(1) = 1.0
beregn(2) = 2.0
beregn((1+2)) = 3.0
beregn(3) = 3.0
beregn((1+2)*3 ) = 9.0
beregn(4) = 4.0
beregn(5) = 5.0
beregn(4/5) = 0.8
beregn(6) = 6.0
beregn(7) = 7.0
beregn((6-7)) = -1.0
beregn(4/5*(6-7)) = -0.8
beregn( sin(4/5*(6-7))) = -0.7173560908995228
Formlen (1+2)*3 - sin(4/5*(6-7)) er beregnet til 9.717356090899523
```

Metoden beregn() deler strengen op i mindre bidder, som den udregner værdien af ved at kalde sig selv rekursivt. For eksempel deles "(1+2)*3 − sin(4*5/(6−7))" op i delene "(1+2)*3" og "sin(4*5/(6−7))", der hver især udregnes ved at kalde beregn().

Bliver beregn() kaldt med en streng, der ikke kan opdeles yderligere, antages det, at strengen indeholder et tal, som bliver fundet med et kald til Double.parseDouble().

Rækkefølgen af kaldene i programmet er:

- beregn("(1+2)*3 − sin(4*5/(6−7))"), der kalder
- beregn("(1+2)*3"), der kalder

- beregn("(1+2)"), der kalder
- beregn("1+2"), der kalder
- beregn("1"), der giver 1
- beregn("2"), der giver 2
- returværdierne 1 og 2 lægges sammen og giver 3
- beregn("3"), der giver 3
- returværdierne 3 og 3 multipliceres og giver 9

- beregn("sin(4/5*(6−7))"), der kalder
- beregn("4/5*(6−7)"), der kalder
- beregn("4/5"), der kalder
- beregn("4"), der giver 4
- beregn("5"), der giver 5
- returværdierne 4 og 5 ganges sammen og giver 0.8
- beregn("(6−7)"), der kalder
- beregn("6−7"), der kalder
- beregn("6"), der giver 6
- beregn("7"), der giver 7
- returværdierne 4 og 5 trækkes fra hinanden og giver −1
- returværdierne 0.8 og −1 divideres med hinanden og giver −0.8
- sinus til returværdien −0.8 beregnes og giver −0.717
- returværdierne 9 og −0.717 trækkes fra hinanden og giver 9.717

Indrykningerne illustrerer dybden af rekursionen (hvor mange gange beregn() er i gang med at kalde sig selv).

Vi vil bygge videre på ovenstående eksempel i afsnit 4.7.3.

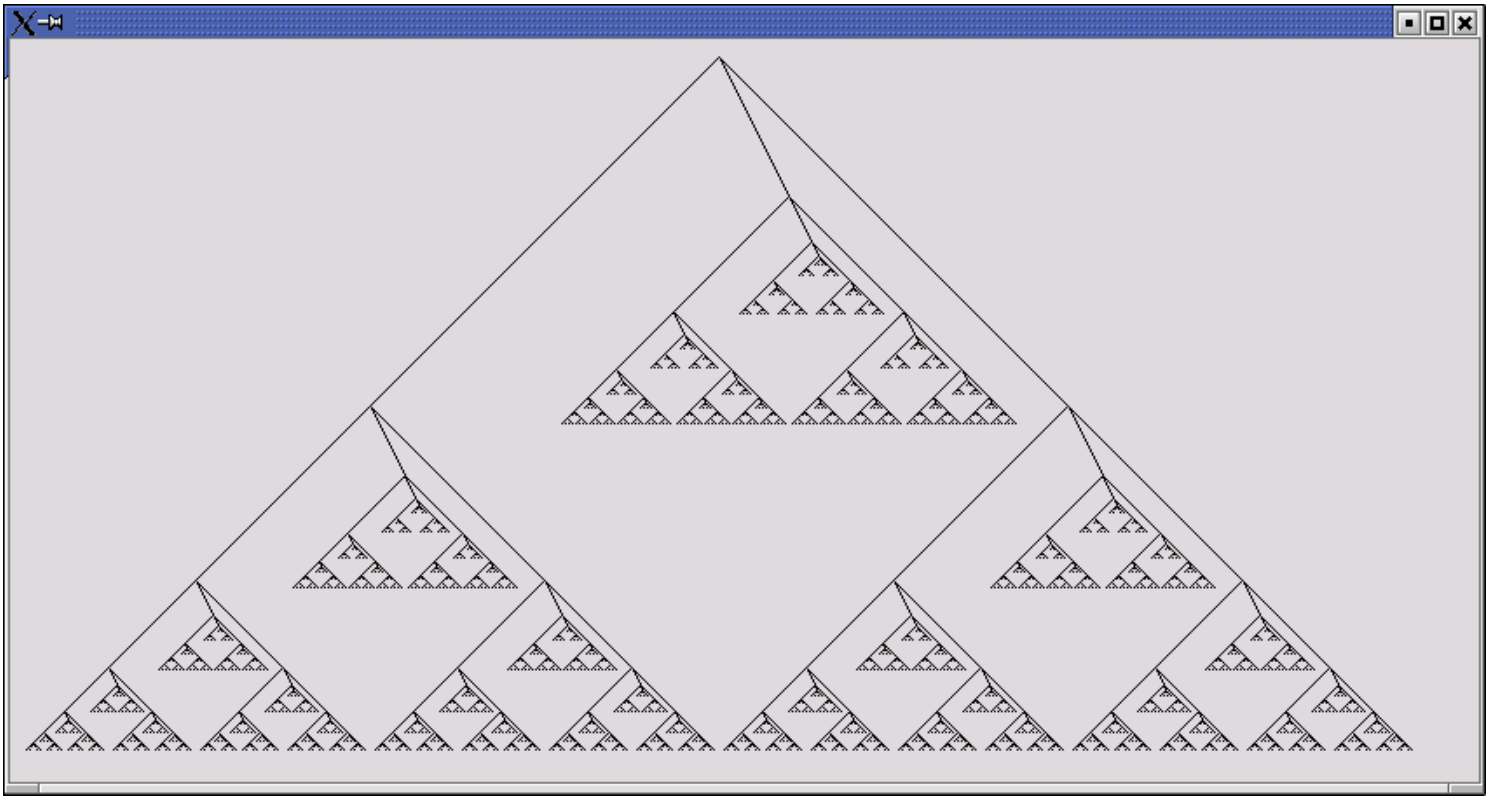
3.4 Tegning af fraktaler

Dette eksempel tegner et fraktalt træ. En fraktal er en struktur, hvor man, hvis man går tæt på en del af strukturen, opdager, at delen har lige så mange detaljer som helheden.

Her er rekursion velegnet, da man blot kan lave en metode tegnGren(), der tegner en gren i et bestemt størrelsesforhold ved at tegne "stammen" i grenen og derefter tegne mindre grene (med kald til tegnGren() med mindre størrelsesforhold).

```
import java.awt.*;
public class Fraktaltrae extends Frame
{
  /**
   * Tegner et fraktalt træ. Hver gren er i sig selv et træ.
   * @param x   x−koordinaten hvor træets rod skal tegnes
   * @param y   y−koordinaten hvor træets rod skal tegnes
   * @param dx  x−forskydning fra rod til træets første forgrening
   * @param dy  y−forskydning fra rod til træets første forgrening
   * @param str træets størrelse
   * @param g   Graphics−objektet
   */
  public void tegnGren(Graphics g, int x, int y, int dx, int dy, int str)
  {
    if (str < 1) return; // vi vil ikke tegne forsvindende små grene

    g.drawLine(x, y, x+dx, y+dy);                // tegn stammen

    tegnGren(g, x+dx, y+dy,−str/2, str/2, str/2); // tegn gren til venstre
    tegnGren(g, x+dx, y+dy, str/10,str/5, str/3); // lille gren lidt til højre
    tegnGren(g, x+dx, y+dy, str/2, str/2, str/2); // tegn gren til højre
  }

  public void paint(Graphics g)
  {
    tegnGren(g,410,30,0,0,400);
  }

  public static void main(String[] arg)
  {
    Fraktaltrae træ = new Fraktaltrae();
    træ.setSize(850, 450);
    træ.setVisible(true);
  }
}
```

På figuren kan man se, hvordan hver gren er en formindsket udgave af hele træet.



3.5 Opgaver

1. Prøv eksemplerne Rekursion, ListFilerRekursivt, Formelberegning og Fraktaltrae.
2. Hvad skriver det følgende program ud? Regn det ud uden at køre programmet.
   Prøv derefter at køre det, og følg med i variablernes værdier.

```
public class Rekursionsopgave
{
  public static String kaldRekursivt(String s)
  {
    if (s.length()<=1) return s;

    String førsteTegn = s.substring(0,1);
    String resten     = s.substring(1);
    String resten2    = kaldRekursivt( resten );
    String detHele    = resten2 + førsteTegn;
    return detHele;
  }

  public static void main(String[] arg)
  {
    String resultat = kaldRekursivt("Hej verden!");
    System.out.println(resultat);
  }
}
```

4 Komponentbaseret programmering

Eksemplet i afsnit 4.7.3 forudsætter kapitel 3, Rekursion.

4.1 Genbrugelige komponenter

Komponenter er programmørens byggeklodser: De kan bruges igen og igen i mange sammenhænge og sættes sammen på alle mulige måder.

En bred definition på 'komponent' kunne være 'afgrænset programdel, der kan genanvendes i flere sammenhænge'. Med denne definition er der dog ret meget, der er komponenter:

- Grafiske komponenter: Button, Label, TextField, Checkbox, ...

- Ikke−grafiske komponenter: CheckboxGroup, en FTP−komponent, ...
- Appletter, servletter, Enterprise JavaBeans (EJB), ...
- De fleste klasser kan bruges igen og igen, f.eks. String, Date, ...

For alle komponenter gælder, at deres omgivelser − beholderen eller containeren − også er vigtig. En komponent skal bruges på en bestemt måde, og den vil ikke fungere uden de rette omgivelser. F.eks. virker et TextField−objekt ikke, hvis dets paint()−metode ikke kaldes (det sørger java.awt.Container for). En applet skal indlæses i en netlæser/browser, en servlet skal udføres i en webserver. Et Date−objekt skal have kaldt de rette metoder, osv.

4.1.1 Javabønner

En mere snæver definition af begrebet komponent er: 'en afgrænset programdel, der kan genanvendes i flere sammenhænge, som kan bruges i et udviklingsværktøjs palette og som tillader brugeren at manipulere med dets egenskaber ved hjælp af værktøjet'.

Denne definition peger hen på Javabønner (eng.: JavaBeans) som f.eks. de grafiske komponenter Button, Label, TextField, Checkbox, ...

Javabønner har en standardiseret måde at undersøge, hvilke egenskaber (eng.: properties) de har (med get− og set−metoder, se senere), og de fleste udviklingsværktøjer ved derfor, hvordan disse komponenter kan konfigureres.

Der er derfor ingen kode i værktøjet beregnet specielt mod de enkelte javabønner. Nye bønner kan uden videre føjes til udviklingsværktøjets palette. Senere i kapitlet vil vi selv programmere nogle bønner og føje dem til paletten.

4.1.2 Eksempel: Bønnen TextField

Eksempelvis har TextField nogle egenskaber:

text angiver, hvad der står i feltet.

columns angiver, hvor bredt feltet skal være.

editable angiver, om brugeren kan redigere teksten i indtastningsfeltet.

echoChar bruges til felter, der skal skjule det indtastede, typisk adgangskoder.

Egenskab

Type

Sættes med metoden

Aflæses med metoden

text

String

setText(String t)

getText()

editable

boolean

setEditable(boolean rediger)

isEditable()

columns

int

setColumns(int bredde)

getColumns()

echoChar

char

setEchoChar(char tegn)

getEchoChar()

Abonnement på hændelser

Bønnen kan blandt andet sende Action−hændelser. Det betyder at, den har metoderne

```
  public void addActionListener(ActionListener l)
  public void removeActionListener(ActionListener l)
```

til at tilføje og fjerne en lytter på denne komponent.

Trykker man retur i indtastningsfeltet, oprettes et hændelses−objekt (af type ActionEvent), og alle lytterne får kaldt metoden actionPerformed() med dette objekt.

4.1.3 Bruge en javabønne

Lad os lave et program, der bruger bønnen TextField, sætter bønnens egenskab *text*, lytter efter, om brugeren trykker retur i indtastningsfeltet (abonnerer på Action−hændelser fra komponenten) og udskriver indholdet af tekstfeltet, når det sker.

Bruge en javabønne fra et udviklingsværktøj

Genereres koden med et udviklingsværktøj, kommer kildeteksten til at se nogenlunde således ud:

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

public class BenytBoenneMedVaerktoej extends Frame
{
  TextField textFieldNavn = new TextField();      // opret bønnen

  public BenytBoenneMedVaerktoej() {
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }

  private void jbInit() throws Exception {
    textFieldNavn.setText("Jacob");               // sæt egenskaben text

    // anonym indre klasse lytter på hændelser og kalder derpå videre til
    // metoden textFieldNavn_actionPerformed()
    textFieldNavn.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        textFieldNavn_actionPerformed(e);
      }
    });

    textFieldNavn.setBounds(new Rectangle(141, 61, 112, 29));
    this.setLayout(null);
    this.add(textFieldNavn, null);
  }

  void textFieldNavn_actionPerformed(ActionEvent e) {
    String navn = textFieldNavn.getText();        // aflæs egenskaben text
    System.out.println("Navnet er: "+navn);
  }

  public static void main(String[] arg)
  {
    BenytBoenneMedVaerktoej vindue = new BenytBoenneMedVaerktoej();
    vindue.setSize(350,100);                      // sæt vinduets størrelse
    vindue.setVisible(true);                      // åbn vinduet
  }
}
```

Der er oprettet en konstruktør, der kalder metoden jbInit(). Et udviklingsværktøj definerer gerne en separat metode, hvor den initialiserer komponenterne. I JBuilder og JDeveloper hedder den jbInit(), mens den hedder initComponents() i Netbeans og Sun ONE Studio.

I metoden jbInit() lægger værktøjet koden til at initialisere de grafiske komponenter og sørger for at placere dem korrekt i vinduet. Føjer du din egen kode til denne metode, så sørg for, at det ligner værktøjets egen kode, ellers kan værktøjet have svært ved at opretholde sammenhængen mellem kode og design.

Der er brugt en anonym indre klasse[1] til at lytte på hændelser (defineret i parameteren til textFieldNavn.addActionListener()). Når hændelsen sker, kalder lytter−objektet videre i metoden textFieldNavn_actionPerformed().

I princippet kunne der altså lige så godt have stået:

```
    textFieldNavn.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        String navn = textFieldNavn.getText();
        System.out.println("Navnet er: "+navn);
      }
    });
```

Bruge en javabønne uden et udviklingsværktøj

Skriver man koden selv uden hjælp fra et udviklingsværktøj, vil man sandsynligvis lægge initialiseringen direkte i konstruktøren og lade klassen selv være en ActionListener i stedet for at definere en indre klasse. Så kommer kildeteksten til at se nogenlunde således ud:

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

public class BenytBoenneSkrevetSelv extends Frame implements ActionListener
{
  TextField textFieldNavn = new TextField();

  public BenytBoenneSkrevetSelv() {
    textFieldNavn.setText("Jacob");

    // klassen selv lytter på hændelser
    textFieldNavn.addActionListener(this);

    textFieldNavn.setBounds(new Rectangle(141, 61, 112, 29));
    this.setLayout(null);
    this.add(textFieldNavn, null);
  }

  public void actionPerformed(ActionEvent e) {
    String navn = textFieldNavn.getText();
    System.out.println("Navnet er: "+navn);
  }

  public static void main(String[] arg) {
    BenytBoenneSkrevetSelv vindue = new BenytBoenneSkrevetSelv();
    vindue.setSize(350,100);
    vindue.setVisible(true);
  }
}
```

Vi har ladet vinduet selv implementere ActionListener, hvorfor vi blot skriver

```
    textFieldNavn.addActionListener(this);
```

og definerer actionPerformed() selv.

4.2 At definere javabønner

Lad os nu se på, hvordan vi selv definerer komponenter/javabønner.

Når man definerer sine egne javabønner, skal man holde tungen lige i munden og huske at skelne mellem tre roller:

- **Programmøren af bønnen (leverandøren)** er den person, der skriver bønnens kode (dvs. koden *inde i* klassen), men kan ikke antage så meget om, hvordan bønnen vil blive brugt. En bønne skal kunne benyttes helt uden at kende den kode, som leverandøren af bønnen har lavet, og være tilpas generel og genanvendelig til, at en anden programmør kan bruge den. Eksempelvis skulle leverandøren til TextField (en programmør ansat i Sun) kode den, så den ikke afhænger ret meget af sine omgivelser.
- **Programmøren, der anvender bønnen i sit program (klienten)**. I denne rolle (som er den mest almindelige) gør man brug af en bønne (klasse), som en anden programmør har skrevet (dvs. man skriver kode *uden for* klassen). Eksempel: At bruge TextField udefra, f.eks. ved i et udviklingsværktøj at føje den til den grafiske brugergrænseflade.
- **Slutbrugeren af programmet** behøver ikke, kende noget til, hvordan koden til hverken bønnen eller for den sags skyld resten af programmet ser ud.

4.2.1 En meget simpel bønne: GentagTekst

Herunder er en simpel grafisk bønne, der tegner en tekst tre gange skråt under hinanden.

```
package vp;
import java.awt.*;
public class GentagTekst extends Component
{
  private String tekstDerSkalVises = "gentag";

  public void setTekst(String t)
  {
    tekstDerSkalVises = t;
  }

  public String getTekst()
  {
    return tekstDerSkalVises;
  }

  public void paint(Graphics g)
  {
    g.drawString(tekstDerSkalVises, 0, 10);
    g.drawString(tekstDerSkalVises, 5, 15);
    g.drawString(tekstDerSkalVises, 10,20);
  }
```

```
}
```

Ud fra kildeteksten ses, at bønnen har egenskaben *tekst* (der bestemmer, hvilken tekst der skal vises).

> En javabønne siges at have en egenskab, hvis den har en tilsvarende get− og/eller set−metode

Bemærk at en javabønnes egenskaber ikke har nogen given relation til de objektvariabler, den måtte have. Således har bønnen egenskaben *tekst*, fordi den har metoden getTekst() og/eller setTekst(), og hvordan bønnen husker egenskaben internt (her sker det i den private variabel tekstDerSkalVises) er sagen uvedkommende, set udefra.

Visse udviklingsværktøjer − herunder JBuilder − kan kun håndtere bønner, der ligger i en pakke, så derfor er alle eksemplerne lagt i pakken 'vp' (husk, at de så også ligger i undermappen 'vp' i forhold til en kildetekstfil, der ikke lå i en pakke).

4.2.2 Brug af bønner fra et udviklingsværktøj

Så snart bønnen er oversat er den klar til brug og kan bruges fra udviklingsværktøjet.

I JBuilder kan det gøres ved at klikke på den lille kasse til venstre for komponenterne på design−fanen og vælge bønnen direkte.

Ellers skal du først definere et bibliotek, der indeholder bønnen:

- I JBuilder: vælg 'Tools | Configure Libraries', vælg 'New..' og 'Add...', og angiv stien (roden) til .class−filerne.
- I JDeveloper: vælg 'Project | Default project settings', under 'Libraries' vælg 'New...' og angiv stien (roden) til .class−filerne.

Nu kan du konfigurere paletten med 'Tools | Configure palette', vælg 'Add component', og vælg den fane, du ønsker komponenten skal vises på (f.eks. 'AWT').

Under 'Library' vælger du det bibliotek du lige har oprettet og derpå klassen i det.

Kode genereret af udviklingsværktøjet

Udviklingsværktøjet kan nu generere et program, der bruger bønnen.

Vælger man bønnens egenskaber, vil man se, at den har egenskaben *tekst*, ud over de, der er arvet fra Component (f.eks. forgrunds− og baggrundsfarve).

Den genererede kode ser således ud (for JBuilder og JDeveloper):

```
import vp.*;
import java.awt.*;

public class VindueMedGentagTekst extends Frame
{
  GentagTekst gentagtekst1 = new GentagTekst();

  public VindueMedGentagTekst() {
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }

  private void jbInit() throws Exception  {
    this.setSize(new Dimension(319, 247));
    this.setLayout(null);

    gentagtekst1.setTekst("ryst!");
    gentagtekst1.setBounds(new Rectangle(177, 190, 111, 28));
    this.add(gentagtekst1, null);
  }
}
```

4.2.3 Størrelsen af en grafisk komponent

Prøver man at bruge GentagTekst som en bønne, opdager man at den som udgangspunkt bliver meget lille (den får en bredde og højde på nul). Det hænger sammen med, at alle grafiske komponenter skal kunne fortælle deres omgivelser, hvor store de har behov for at være på skærmen for at kunne fungere korrekt.

Det sker ved, at omgivelserne kalder metoden getPreferredSize(), der skal returnere et Dimension−objekt med den foretrukne bredde og højde.

Herunder er endnu en grafisk komponent. Fordi vi har defineret getPreferredSize(), vil udviklingsværktøj og layout−managere så vidt muligt rette sig efter denne størrelse.

Layout−managere og deres virkemåde er beskrevet mere grundigt i afsnit 4.7.4.

```
package vp;
import java.awt.*;
```

```
public class BoenneMedForetrukkenStr extends Component
{
  Dimension foretrukneStørrelse = new Dimension(100,50);

  /** fortæl containeren hvad denne komponents foretrukne størrelse er */
  public Dimension getPreferredSize()
  {
    return foretrukneStørrelse;
  }

  public void paint(Graphics g)
  {
    Dimension str = getSize();  // faktisk størrelse (kan variere fra foretrukne)
    g.drawOval(0, 0, str.width, str.height);
  }
}
```

Ud over getPreferredSize() findes også getMinimumSize() og getMaximumSize():

```
  public Dimension getMinimumSize()
  public Dimension getMaximumSize()
```

Disse bruges (sammen med getPreferredSize()) af containerens layout−manager for at afgøre, hvordan containerens komponenter skal placeres indbyrdes. Det er altså, ligesom med pant()−metoden, ikke meningen, at man selv kalder dem, men at systemet (containerens layout−manager) kalder dem, når der er brug for det (f.eks. når vinduet ændrer størrelse).

4.2.4 Skelne mellem udviklings− og i kørselsfasen

Når man i udviklingsfasen bruger designværktøjet til at designe skærmbilleder og lægger en komponent (bønne) ind på skærmbilledet, sker der det, at værktøjet rent faktisk opretter bønnen (med new), sætter dens egenskaber tilsvarende (med kald af set−metoder) og viser bønnen på skærmen.

Nogle gange har en bønne imidlertid brug for at vide, om den lige nu er i gang med at blive brugt af et udviklingsværktøj under udviklingsfasen eller om den rent faktisk er en del af et kørende program. Det kunne f.eks. være relevant, hvis bønnen lagde beslag på nogle vigtige resurser eller hvis den havde et højt processor− eller hukommelsesforbrug.

En bønne kan undersøge, om den er i udviklingsfasen/designfasen (og altså kører i et udviklingsværktøj) eller i kørselsfasen ved at kalde metoden java.beans.Beans.isDesignTime():

```
package vp;
import java.awt.*;
public class BoenneGenkenderDesignfase extends Component
{
  Dimension foretrukneStørrelse = new Dimension(100,50);

  public Dimension getPreferredSize()
  {
    return foretrukneStørrelse;
  }

  public void paint(Graphics g)
  {
    if (java.beans.Beans.isDesignTime())
    {
      g.drawString("Designfase",5,15);
    } else {
      g.drawString("Under kørsel",5,15);
    }
  }
}
```

4.3 Karakteristika ved javabønner

Der er visse karakteristika ved javabønner, som gør det muligt at behandle dem som afgrænsede komponenter og konfigurere dem med et udviklingsværktøj. Disse er ridset op i det følgende.

4.3.1 Bønner skal have en parameterløs konstruktør

Bønner *skal* have en konstruktør uden parametre, dvs. de skal kunne oprettes som f.eks.:

```
  GentagTekst gentagtekst1 = new GentagTekst();
```

Dette er, for at udviklingsværktøjet kan oprette komponenterne på en ensartet måde.

Prøv, for at illustrere nødvendigheden af dette krav, at forestille dig, at kravet *ikke* fandtes: Så skulle udviklingsværktøjet kunne håndtere klasser med meget indviklede konstruktører, f.eks. en med 17 parametre, hvor parameter nummer 6 i øvrigt altid skal være større end parameter 7 og parameter 16 være et lige tal større end 38. Et menneske kunne selvfølgelig læse dokumentationen og forstå kravene, mens det er helt umuligt for værktøjet at imødekomme sådanne krav.

I stedet for at angive startværdierne i konstruktøren sættes de som egenskaber.

4.3.2 Bønner kan have egenskaber

Egenskaberne læses/sættes med de kendte get− og set−metoder.

Disse kan være simple som f.eks.:

```
public void setTekst(String t)
public String getTekst()
```

Egenskaberne bør være uafhængige, forstået på den måde at bønnen ikke kan antage at en bestemt egenskab bliver sat eller at egenskaberne bliver sat i en bestemt rækkefølge.

Arver man fra en bønne, vil de arvede egenskaber indgå på lige fod med de nye egenskaber.

Bønnens egenskaber kan være af en vilkårlig type, men værktøjer understøtter som regel kun egenskaber af de simple typer (int, double, ...) og objekter af type String, Color, Point, Dimension og et par stykker til. Ønsker man, at brugeren kan redigere egenskaber af andre typer må man selv programmere, hvordan det skal ske (se næste afsnit).

Indekserede egenskaber

Egenskaber kan også være indekserede (dvs. i et array) som f.eks.:

```
public void setTekst(String t[])
public String[] getTekst()
```

Ofte vil der da også være mulighed for at arbejde med en enkelt indgang, f.eks.:

```
public void setTekst(int indeks, String t)
public String getTekst(int indeks)
```

Bundne og begrænsede egenskaber

Nogen gange er det hensigtsmæssigt, at andre kan få at vide, når en egenskab bliver ændret. Til det formål kan egenskaber være bundne (eng.: bound) eller begrænsede (eng.: constrained).

Når *bundne egenskaber* ændres, afstedkommer det en hændelse (af typen PropertyChangeEvent). Disse hændelser kan man lytte efter (ved at kalde addPropertyChangeListener() på komponenten med en lytter af typen PropertyChangeListener som parameter). Man kan altså få at vide, når en bestemt egenskab ændres, ved at "abonnere" på den, og det foregår på præcis samme måde som med andre slags hændelser.

Begrænsede egenskaber er som bundne egenskaber, men hændelserne, der sendes, er i stedet af typen VetoableChangeEvent. Lytterne (der er af typen VetoableChangeListener) har lov til at kaste undtagelsen PropertyVetoException for at annullere ændringen af egenskaben.

4.3.3 Bønner kan have tilknyttet ekstra information

Ekstra information om en bønne ligger i en såkaldt BeanInfo−klasse[2]. F.eks. kunne GentagTekst.java have tilknyttet klassen GentagTekstBeanInfo.java til ekstra information.

Denne information er *kun* til hjælp for udviklingsværktøjet. Hvis der ikke er en BeanInfo−klasse, bruges introspektion (beskrevet i kapitel 11), dvs. inspicering af bønne−klassens metoder (egenskaber). Introspektion er oftest tilstrækkeligt, så BeanInfo−klasser er ikke så almindelige. De ekstra informationer, der kan specificeres i en BeanInfo−klasse, er:

- Hvilket ikon bønnen skal repræsenteres af i værktøjet
- Hvilke egenskaber der findes, og for hver egenskab en beskrivelse og hvordan den aflæses og sættes
- Hvordan de redigeres. Der kan tilknyttes en klasse, der bestemmer f.eks.:

- Hvilke værdier der er mulige
- Hvilken javakode der skal sættes ind i kildeteksten
- Om et skræddersyet redigeringsvindue skal dukke op

4.3.4 Bønner bør være afgrænsede og uafhængige

Det siger sig selv, at en komponent, der skal kunne bruges igen og igen i mange sammenhænge, skal være fuldstændig afgrænset og uafhængig af omgivelserne. Er den afhængig af dele af resten af programmet, kan den jo netop ikke bruges uden disse programdele.

4.3.5 Bønner kan understøtte hændelses−lyttere

Mange bønner har brug for at fortælle resten af programmet (selvom bønnen selvfølgelig ikke aner, hvilket program det måtte være), at der er sket noget.

Det går ikke, at bønnen kalder en metode i det omkringliggende program, for så ville bønnen ikke virke i andre sammenhænge.

I stedet skal det omgivende program registrere et lytter−objekt hos bønnen, som bønnen skal huske. Når den får brug for at fortælle resten af programmet, at "nu er der er sket noget" kalder den lytteren.

Det bedst kendte eksempel på dette er måske Button. For at få at vide, når der trykkes på den, skal vi registrere et ActionListener−objekt hos den. Når der trykkes på knappen, vil den kalde metoden actionPerformed() på lytteren med et ActionEvent−objekt som parameter, der beskriver hændelsen.

Hvis du nu skal til at programmere din første komponent nu, så bemærk, at du har skiftet rolle, jvf. afsnit 4.2: Før har du kun *anvendt* andre komponenter og i den forbindelse derfor måske *kaldt* f.eks. addActionListener().

Nu laver du selv komponenter og skal derfor *definere* f.eks. addActionListener(), hvis du vil have, at dine komponenter understøtter denne slags hændelser. Tilsvarende bør du definere en removeActionListener(), der fjerner en lytter fra din komponent, så at sige "annullerer abonnementet" på hændelserne.

Det vil også være en god idé internt at huske, hvilke lyttere der er registreret, og definere en intern metode, der kalder actionPerformed() på alle lytterne, når en hændelse skal affyres (man kunne passende kalde metoden for sendActionPerformedTilLytterne()).

Herunder et eksempel på en komponent, der sender Action−hændelser. De sendes, hver gang dets paint()−metode kaldes (dette er ikke specielt hensigtsmæssigt, men giver et simpelt eksempel[3]).

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

/** En komponent, der sender en hændelse hver gang den gentegnes.
 *  Du kan vælge at kopiere kildeteksten ind i dit eget program.
 *  Hændelsen, der sendes er ActionEvent. Denne hændelse indeholder bl.a.:
 *  <ul>
 *  <li> et objekt (kilden til hændelsen)
 *  <li> en streng (beskrivelse)
 *  <li> et tal (et ID)
 *  </ul>
 *  Du kan selvfølgelig selv bestemme hvad objektet, strengen og tallet er,
 *  hvis det er dine egne klasser der lytter efter hændelserne.
 */
public class SenderActionEvent extends Component
{
  // ---------------------------- kopier herfra ----------------------------
   /** Lyttere til denne bønne */
   private ArrayList lyttere = new ArrayList(2);

   /** Tilføjer en lytter. Lytteren vil få kaldt metoden actionPerformed() når
    *  der sker en hændelse.
    *  @param l Lytteren, der skal tilføjes denne komponent.
    */
   public synchronized void addActionListener(ActionListener l)
   {
     lyttere.add(l);
  }

   /** Fjerner en lytter */
   public synchronized void removeActionListener(ActionListener l)
   {
     lyttere.remove(l);
  }

   /** Sender en hændelse til lyttere. Lytterne er de, der tidligere er blevet
    *  tilføjet med kald til addActionListener()
    *  @see #addActionListener(ActionListener)
    *  @param hændelse Hændelsen med de data, der skal sendes til lytterne
    */
   protected void sendActionPerformedTilLytterne(ActionEvent hændelse)
   {
     for (Iterator i=lyttere.iterator(); i.hasNext(); )
     {
       ActionListener l = (ActionListener) i.next();
       l.actionPerformed(hændelse);
     }
  }
  // ---------------------------- kopier hertil ----------------------------

  public void paint(Graphics g)
  {
    // opret en Action−hændelse, der kommer fra denne komponent og send den
    // brug konstruktøren new ActionEvent( afstanderobjekt, id, beskrivelse)
    ActionEvent hændelse = new ActionEvent(this, 0, "paint() kaldt");
    sendActionPerformedTilLytterne(hændelse);
  }
}
```

Koden fra eksemplet kan kopieres og genbruges i dine egne komponenter.

Action−hændelser er den mest anvendte hændelsestype, men hvis du vil understøtte en anden type, skal du blot ændre alle de steder, hvor der står 'Action' (til f.eks. 'Mouse').

4.4 Ekstra eksempler

Her kommer nogle flere eksempler på komponenter. Alle kan lægges ind i et udviklingsværktøjs palette og manipuleres fuldstændigt som de forud installerede komponenter.

4.4.1 Rystetekst

Den følgende bønne viser en tekst, der ryster. Det gøres ved, at den starter en separat tråd, som 10 gange i sekundet kalder repaint() for at få komponenten gentegnet. I paint() tegnes en tekst med en tilfældig forskydning, og når komponenten gentegnes 10 gange i sekundet, ser det ud, som om teksten ryster.

Samtidig kan denne bønne fortælle containeren, hvad dens foretrukne størrelse er (metoden getPreferredSize() beskrevet i afsnit 4.2.3).

Det sker ved at undersøge den pågældende tekst og finde ud af, hvor bred og høj den er med den pågældende skrifttype−størrelse (font−metrik).

```
package vp;
import java.awt.*;
public class Rystetekst extends Component implements Runnable
{
  private String tekst = "rystetekst";
  public void setTekst(String t) { tekst = t; foretrukneStørrelse = null; }
  public String getTekst() { return tekst; }

  Dimension foretrukneStørrelse = null;

  public Dimension getPreferredSize() {
    if (foretrukneStørrelse == null) try {
      FontMetrics fm = getFontMetrics(getFont());
      int tbr = fm.stringWidth(tekst);                          // tekstbredde
      int thø = fm.getHeight();                                 // teksthøjde
      foretrukneStørrelse = new Dimension(tbr + 10, thø + 10); // lidt ekstra
    } catch (Exception e) {
      e.printStackTrace();
      foretrukneStørrelse = new Dimension(150,50);
    }
    return foretrukneStørrelse;
  }

  public Rystetekst()
  {
    // hvis ikke i designfase så start en tråd der tager sig af opdateringen
    if (!java.beans.Beans.isDesignTime())
    {
      Thread tråd = new Thread(this);
      tråd.setDaemon(true);    // systemet skal ikke vente på at tråden stopper
      tråd.start();            // ny vil ny tråd starte nede i run()−metoden
    }
  }

  /** sørger for at kalde repaint() regelmæssigt */
  public void run() {
    try {
      while (true)
      {
        Thread.sleep(100);     // vent 1/10 sekund
        repaint();             // gentegn komponenten
      }
    } catch (Exception e) {}
  }

  public void paint(Graphics g)
  {
    // tegn tekst på tilfældig x− og y−koordinat
    g.drawString(tekst,            (int)(Math.random()*10),
                       getHeight()−(int)(Math.random()*10));
  }
}
```

4.4.2 Rulletekst

Her er endnu en grafisk bønne, der lader en tekst rulle vandret hen over skærmen. Den har derfor egenskaberne *tekst* og *fart*. Derudover er der *opdateringstid*, der bestemmer, hvor hyppigt den helst skal gentegnes på skærmen.

```
package vp;
import java.awt.*;
import java.util.*;

public class Rulletekst extends Component implements Runnable
{
  /** fortæl containeren hvad denne komponents foretrukne størrelse er */
  public Dimension getPreferredSize() { return new Dimension(100,15); }

  // Egenskaber
  private String tekst = "rulletekst ";
```

```
  private int fart = 10;           // antal punkter der rykkes i sekundet
  private int opdateringstid = 50; // antal millisekunder mellem hver gentegning

  public void setTekst(String t) { tekst = t; klar = false; }
  public String getTekst() { return tekst; }

  public void setFart(int f) { fart = f; }
  public int getFart() { return fart; }

  public void setOpdateringstid(int f) {
    if (f>=10) opdateringstid = f; // tillad ikke under 10 msek
    else opdateringstid = 10;
  }

  public int getOpdateringstid() { return opdateringstid; }

  // Interne variabler
  private boolean klar = false;
  private int tbr, thø;            // tekstens bredde og højde i punkter
  private String tekstx;           // teksten i det nødvendige antal kopier

  private synchronized void gørKlar()
  {
    FontMetrics fm = getFontMetrics(getFont());
    tbr = fm.stringWidth(tekst);
    thø = fm.getHeight();
    Dimension d = getSize(); // komponentens størrelse
    int antalKopier = 2*d.width/tbr + 1;
    tekstx = tekst;
    while (antalKopier-- >= 1) tekstx = tekstx + tekst;

    Thread tråd = new Thread(this);
    tråd.setDaemon(true);
    tråd.start();
    klar = true;
  }

  /** sørger for at kalde repaint() regelmæssigt */
  public void run() {
    try {
      while (true) {
        Thread.sleep(opdateringstid);
        repaint();
      }
    } catch (Exception e) {}
  }

  public void paint(Graphics g)
  {
    if (!klar) gørKlar();

    int x = (int) (System.currentTimeMillis()*fart/1000);
    g.drawString(tekstx, x%tbr-tbr, thø);
  }
}
```

4.4.3 Simpel kryptering

Javabønner behøver ikke at være grafiske. Her er en ikke−grafisk bønne, der kan kode tekster med en gammel simpel metode (Cæsar−kodning): Man erstatter a med b, b med c, c med d, ... æ med å og å med a.

Ønsker man en anden rækkefølge af bogstaverne, ændrer man egenskaben *kodestreng*.

Egenskaben *hop* bruges til at specificere kodningens retning (1 svarer til indkodning, −1 til afkodning).

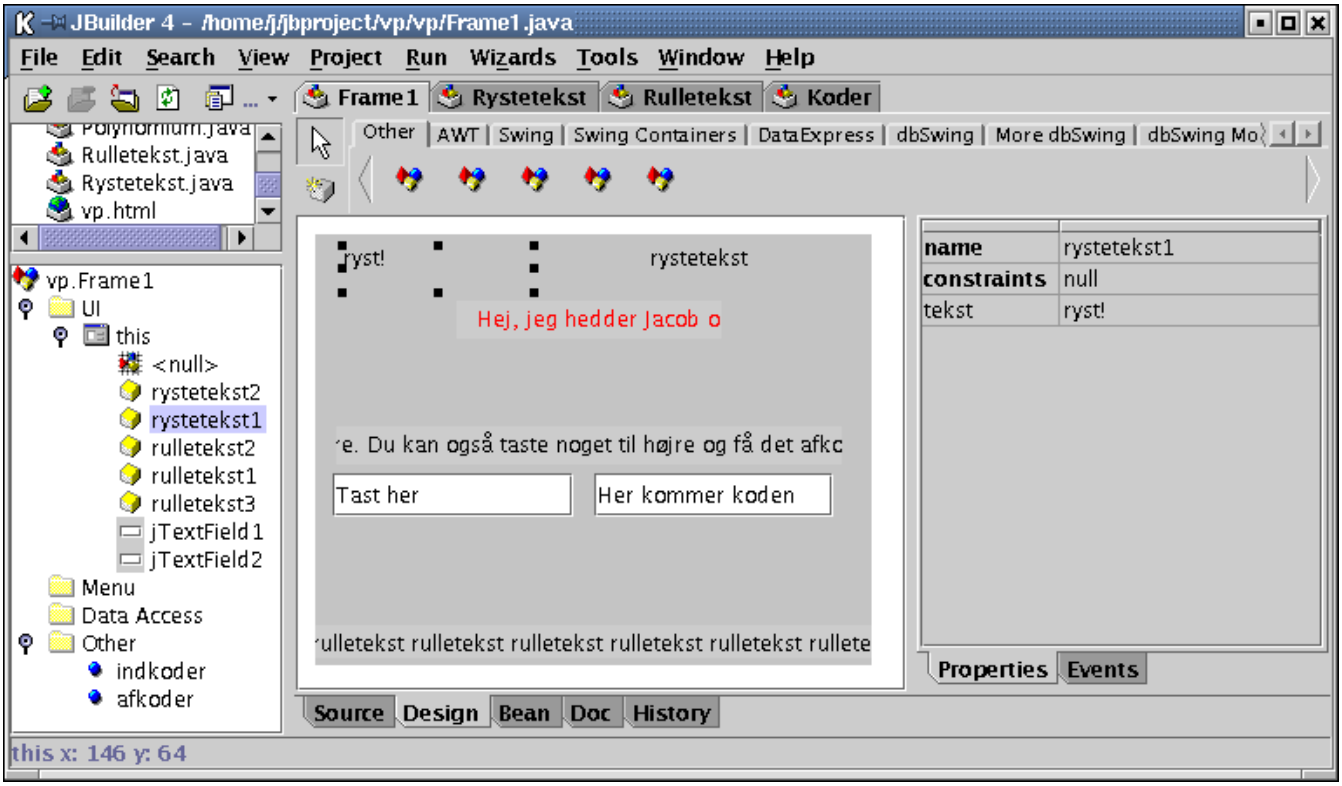
Metoden kod() udfører den faktiske kodning. Den svarer ikke til en egenskab (med get− og set−metoder), og programmøren, der anvender bønnen, må derfor selv sørge for at lave koden, der kalder metoden, når der er brug for det.

```
package vp;
public class Koder
{
  private String kodestreng = "abcdefghijklmnopqrstuvwxyzæøå";
  private int hop = 1;

  public String getKodestreng() { return kodestreng; }
  public void setKodestreng(String ny) { kodestreng = ny; }

  public void setHop(int ny) { hop = ny; }
  public int getHop() {  return hop; }

  public String kod(String s)
  {
    StringBuffer sb = new StringBuffer();
    for (int i=0; i<s.length(); i++)
    {
      int p = kodestreng.indexOf( s.charAt(i) );
      if (p>=0) {
        p = (p + hop + kodestreng.length()) % kodestreng.length();
        sb.append( kodestreng.charAt( p ));
```

```
      } else sb.append( s.charAt(i) );
    }
    return sb.toString();
  }
}
```

4.4.4 Eksempel på brug af bønnerne i et værktøj

På figuren herunder er vi i gang med at designe et lille program med bønnerne Rystetekst, Rulletekst og Koder (der er lagt ind i 'Other'−fanen i udviklingsværktøjet JBuilders palette). Vi har lige nu markeret bønnen Rystetekst, og værktøjet viser derfor egenskaben *tekst* til højre.



Da bønnen Koder ikke er grafisk, vises den ikke i værktøjets design−fane. I stedet må man vælge den i struktur−træet (nederst til venstre er instanserne indkoder og afkoder af Koder−bønnen vist).

Programmet viser nogle ryste− og rulletekster og to indtastningsfelter. Taster man noget i feltet til venstre og trykker retur, vil teksten blive kodet med en Koder−bønne og resultatet vist i feltet til højre.

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

public class BenytRystetekstRulletekstKoder extends Frame
{
  Rystetekst rystetekst1 = new Rystetekst();
  Rystetekst rystetekst2 = new Rystetekst();

  Rulletekst rulletekst1 = new Rulletekst();
  Rulletekst rulletekst2 = new Rulletekst();
  Rulletekst rulletekst3 = new Rulletekst();

  Koder indkoder = new Koder();
  Koder afkoder = new Koder();

  JTextField jTextField1 = new JTextField();
  JTextField jTextField2 = new JTextField();


  public BenytRystetekstRulletekstKoder() {
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }

  private void jbInit() throws Exception {
    rulletekst1.setTekst("Hej, jeg hedder Jacob Nordfalk.   ");
    rulletekst1.setForeground(Color.red);
    rulletekst1.setFart(-50);
    rulletekst1.setOpdateringstid(20);
    rulletekst2.setOpdateringstid(100);
```

```
    rulletekst3.setTekst("Indtast teksten i feltet til venstre herunder og " +
    "tryk retur. Så kommer den indkodede tekst til højre. Du kan også taste " +
    "noget til højre og få det afkodet til venstre.");
    rulletekst3.setFart(-20);

    afkoder.setHop(-1);

    jTextField1.setText("Tast her");
    jTextField1.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        jTextField1_actionPerformed(e);
      }
    });
    jTextField2.setText("Her kommer koden");
    jTextField2.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        jTextField2_actionPerformed(e);
      }
    });

    this.setLayout(null);
    this.setSize(new Dimension(319, 247));

    this.add(rystetekst1, null);                          // tilføj til container
    this.add(rystetekst2, null);
    this.add(rulletekst1, null);
    this.add(rulletekst2, null);
    this.add(rulletekst3, null);
    this.add(jTextField1, null);
    this.add(jTextField2, null);

    rystetekst1.setBounds(new Rectangle(15, 6, 111, 28)); // placering på skærm
    rystetekst2.setBounds(new Rectangle(183, 3, 112, 35));
    rulletekst1.setBounds(new Rectangle(81, 38, 152, 22));
    rulletekst2.setBounds(new Rectangle(0, 224, 320, 19));
    rulletekst3.setBounds(new Rectangle(12, 110, 290, 22));
    jTextField1.setBounds(new Rectangle(10, 137, 138, 25));
    jTextField2.setBounds(new Rectangle(160, 137, 137, 25));
  }

  void jTextField1_actionPerformed(ActionEvent e) {
    String s = e.getActionCommand();
    s = indkoder.kod(s);
    jTextField2.setText(s);
  }

  void jTextField2_actionPerformed(ActionEvent e) {
    String s = e.getActionCommand();
    s = afkoder.kod(s);
    jTextField1.setText(s);
  }

  public static void main(String[] arg) {
    BenytRystetekstRulletekstKoder v = new BenytRystetekstRulletekstKoder();
    v.setSize(400,400);
    v.setVisible(true);
  }
}
```

4.5 Opgaver

I alle opgaverne herunder er det underforstået, at du sørger for at afprøve dine ting ved at lave små programmer, der bruger komponenterne.

Vejledende løsninger på nogen af opgaverne findes i næste afsnit.

Grafiske komponenter

1. Ret i bønnen GentagTekst, og indfør egenskaben *antal*, der bestemmer, hvor mange gange teksten skal tegnes.
2. Ret bønnen, så den fortæller sin foretrukne størrelse til containeren (se afsnit 4.2.3, Størrelsen af en grafisk komponent).
3. Kig på bønnen Rystetekst i afsnit 4.4.1. Hvad hvis man skulle kunne styre bønnens opdateringstid, der bestemmer, hvor hurtigt teksten ryster (tidsrummet mellem to gentegninger)? Indfør egenskaben *opdateringstid* i bønnen, og brug den.
4. Blandt AWT−komponenterne mangler der en komponent, der kan vise et billede[4].
   Lav en, f.eks. med udgangspunkt i GentagTekst.java. Definér egenskaben
   filnavn en streng, der beskriver, hvor billedet er
   Billeder kan hentes med f.eks.:
   ```
   Image i = Toolkit.getDefaultToolkit().getImage("hej.jpg");
   g.drawImage(i, 10, 10, this);
   ```
   Her skal filen hej.jpg ligge samme sted, som programmet udføres (ellers prøv at kopiere billedet til nogle forskellige steder i filstrukturen, indtil programmet 'får fat' i det).
5. Lav en grafisk komponent, der viser en animation, dvs. et antal billeder vist i kort rækkefølge efter hinanden. Kig på Rulletekst.java for nogle idéer.
   Har du ikke lavet den forrige opgave, så tag udgangspunkt i Swing−komponenten JLabel (der ud over en tekst også kan vise et billede).

Webserver−komponent

1. Lav en (ikke−grafisk) webserver−bønne. Tag udgangspunkt klasserne FlertraadetHjemmesidevaert og Anmodning fra http://javabog.dk/OOP/eksempler/kapitel_17/ (beskrevet i http://javabog.dk/OOP/kapitel16.html og det efterfølgende kapitel).
   Den skal have egenskaberne:
   port angiver, hvilken port serveren skal lytte på (f.eks. 8080)
   aktiv om den er aktiv, dvs. om den venter på anmodninger (true/false)
   Lad den først bare svare med den samme tekst uafhængig af spørgsmålet.
   Opret et testprogram, der anvender bønnen og sætter dens egenskaber.
2. Lad den understøtte abonnement på Action−hændelser (se afsnit 4.3.5), på en sådan måde, at der sendes en hændelse, hver gang der kommer en anmodning.
   Ændr i testprogrammet så det abonnerer på hændelsen og skriver ud, når den indtræffer.
3. Lad bønnen kunne læse filer fra filsystemet og sende til brugeren. Definér egenskaben
   rod sti til rodkataloget, hvor HTML−siderne, der kan hentes, er (f.eks. C:\HTML).

4.6 Løsninger

4.6.1 Grafisk komponent: GentagTekst

4.6.2 Grafisk komponent: Billede

4.6.3 Webserver−komponent

4.7 Avanceret

4.7.1 En komponent til at tegne kurver

4.7.2 Repræsentation af funktioner

4.7.3 Fortolkning af strenge til funktioner

4.7.4 Layout−manageres virkemåde

4.7.5 Øvelse: Samspil med layout−manager

Jeg lover at anskaffe den i nær fremtid.

4.7.6 Opgave: Adresseindtastningskomponent

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

4.7.7 Løsning: Adresseindtastningskomponent

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

1En anonym klasse er en klasse uden navn, som der oprettes netop ét objekt ud fra der, hvor den defineres (altså en unavngiven lokal klasse). Mere information om anonyme klasser kan f.eks. findes i kapitlet 'Avancerede klasser' i 'Objektorienteret programmering i Java' af Jacob Nordfalk, som også kan læses på http://javabog.dk

2Dette skulle svare til et type−library i Windows' COM−verden.

3Normalt sendes der ikke hændelser, fordi paint() kaldes. Se klassen Kontomodel i afsnit 19.3.1 for et mere realistisk eksempel.

4Dette gælder ikke Swing, hvor man kan sætte billeder på næsten alle komponenterne. For eksempel kan JLabel ud over en tekst også vise et billede.

5Dette er et eksempel på Rekursiv Komposition (se afsnit 18.5).

5 Grafiktegning (Java2D)

Java2D giver mulighed for mere avanceret grafikmanipulering. Blandt faciliteterne er:

- Klasser til en række geometriske grundfigurer, såsom linjer, kurver, rektangler, ellipser og mekanismer til at tegne næsten enhver ønskelig geometrisk form.
- Transformationer (skalering, rotering, vridning) af figurer, tekster og billeder.
- Halvgennemsigtig tegning (hvor det, der var bag ved det tegnede, stadig kan skimtes).
- Trappeudjævning (udjævnede farveovergange, eng.: antialiasing).
- Mekanismer til at afgøre, om et punkt er inden eller uden for en vilkårlig geometrisk figur, tekst eller billede (f.eks. klik med musen).

5.1 Java2D kort fortalt

Java2D virker ved, at paint()−metoden får overført et Graphics−objekt, som i virkeligheden er et Graphics2D−objekt (Graphics2D arver fra Graphics).

Det typekonverterer man til Graphics2D og har så adgang til de ekstra funktioner i Java2D:

```
public void paint(Graphics g)
{
  Graphics2D g2 = (Graphics2D) g;
  ...
}
```

Man kan derefter tegne med de normale funktioner fra Graphics−klassen, men også med nogle af de mange nye metoder i Graphics2D−klassen.

De gamle tegnemetoder (som drawLine(), drawRect(), ...) er erstattet med en enkelt, draw(), der kan tegne alle mulige geometriske figurer repræsenteret af Shape−objekter:

```
Shape s = new Line2D.Float(0, 0, 100, 100);
g2.draw(s);
```

5.1.1 Geometriske figurer

Shape−objekter (der findes i pakken java.awt.geom) repræsenterer en eller anden geometrisk figur på skærmen. Der findes disse grundlæggende Shape−objekter:

- Rektangler − Rectangle2D og RoundRectangle2D
- Linjer − Line2D (en ret linje), CubicCurve2D (en linje, der er buet efter et ankerpunkt) og QuadCurve2D (en linje, der er buet efter to ankerpunkter)
- Ellipse2D og Arc2D (buestykke)

Klassen GeneralPath repræsenterer en vilkårlig figur, der er en kombination af ovenstående figurer (Shape−objekter):

```
GeneralPath figur = new GeneralPath();
figur.append( new Line2D.Float(0, 0, 100, 100), false );
figur.append( new CubicCurve2D.Float(0, 0, 80, 15, 10, 90, 100, 100), false );
figur.append( new Arc2D.Float(−30, 0, 100, 100, 60, −120, Arc2D.PIE), false );
```

Metoden append() tager et Shape−objekt og en boolean, der afgør om det nye Shape−objekt skal forbindes med den eksisterende figur eller ej.

GeneralPath har også metoderne moveTo(x, y), lineTo(x, y), quadTo(x,y,x2,y2) og curveTo(x, y, x1, y1, x2, y2) til at bygge en figur op med.

Man kan derefter tegne den resulterende figur med én kommando:

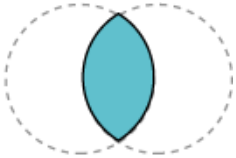
```
  g2.draw( figur );
```

Det samme GeneralPath−objekt kan med fordel bruges igen i hver gentegning for hurtigere kørselstid (hvis det oprettes et andet sted end i paint()−metoden).

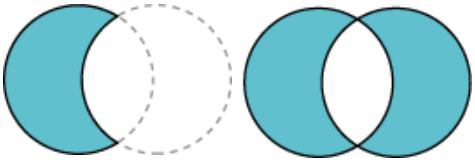
5.1.2 Geometriske operationer

Med klassen Area kan man hurtigt udføre en række handlinger med shapeobjekter uden at beskrive hvert linjesegment eller kurve, som f.eks.:

- **add** finder foreningsmængden af A og B
- **intersect** finder fællesmængden af A og B
- **subtract** trækker fællesmængden af A og B fra A
- **exclusiveOr** finder foreningsmængden af A og B, men fjerner fællesmængden



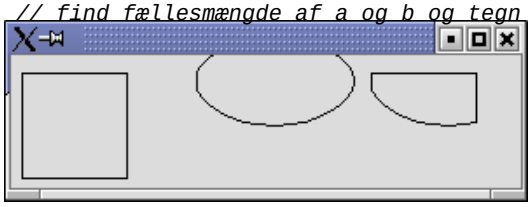
add intersect



subtract exclusiveOr

Nedenfor er vist et lille eksempel på brugen af klassen Area.

```
import java.awt.*;
import java.awt.geom.*;  // indeholder klasser der er specielle for Java2D

public class FigurLeg extends Frame
{
  public void paint(Graphics g)
  {
    Graphics2D g2 = (Graphics2D)g;
    Area a = new Area(new Rectangle2D.Double(10,30,60,60));
    g2.draw(a);

    Area b = new Area(new Ellipse2D.Double(10,10,90,50));
    g2.translate(100,0); // tegn i midten
    g2.draw(b);

    a.intersect(b);      // find fællesmængde af a og b og tegn yderst til højre
```



```
    g2.translate(100,0);
    g2.draw(a);
  }

  public static void main(String[] args) {
    FigurLeg fl = new FigurLeg();
    fl.setSize(300,100);
    fl.setVisible(true);
  }
}
```

Yderst til højre ses fællesmængden af firkanten og cirklen.

5.1.3 Transformationer

Klassen AffineTransform repræsenterer en lineær (affin) transformation af koordinatsystemet. Med den kan man lave skaleringer (større/mindre), drejninger (med/mod uret) og vridninger ("vælte" figuren).

Ved at kalde transform() på Graphics2D−objektet ændres dets koordinatsystem sådan, at alt det, der efterfølgende tegnes, bliver skaleret, drejet og vredet i henhold til transformationen. Hvis man ændrer transformationen, bør man huske den gamle (fås med getTransform()) og sætte den tilbage igen (med setTransform()), før paint() returnerer.

5.1.4 Linjetyper

Med setStroke() sætter man bredden af linjen, der skal tegnes, om linjerne skal have kantede eller afrundede endestykker og knæk, om de skal være punkterede (og hvordan).

5.1.5 Tegnekvalitet

Med setRenderingHint() sætter man forskellige vink til tegnealgoritmen: om der skal laves trappeudjævning, om hastighed eller kvalitet skal prioriteres og meget andet.

5.1.6 Videre læsning

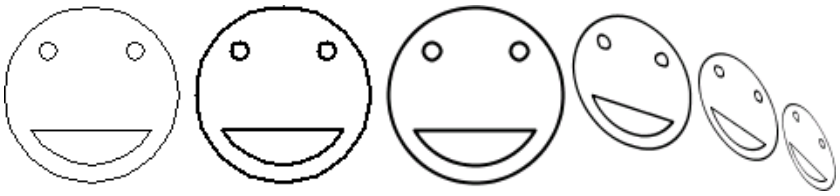
Afsnit 12.4.2 giver et overblik over de mange klasser, der er til rådighed i Java2D.

På http://java.sun.com/docs/books/tutorial/2d findes en god introduktion til Java2D.

5.2 Eksempel

I eksemplet herunder opretter vi en figur og tegner den, først almindeligt, derefter med en linjetykkelse på 2 punkter, derefter med trappeudjævning og til sidst med mere og mere rotation, skalering og vridning.

Resultatet bliver følgende figurer:



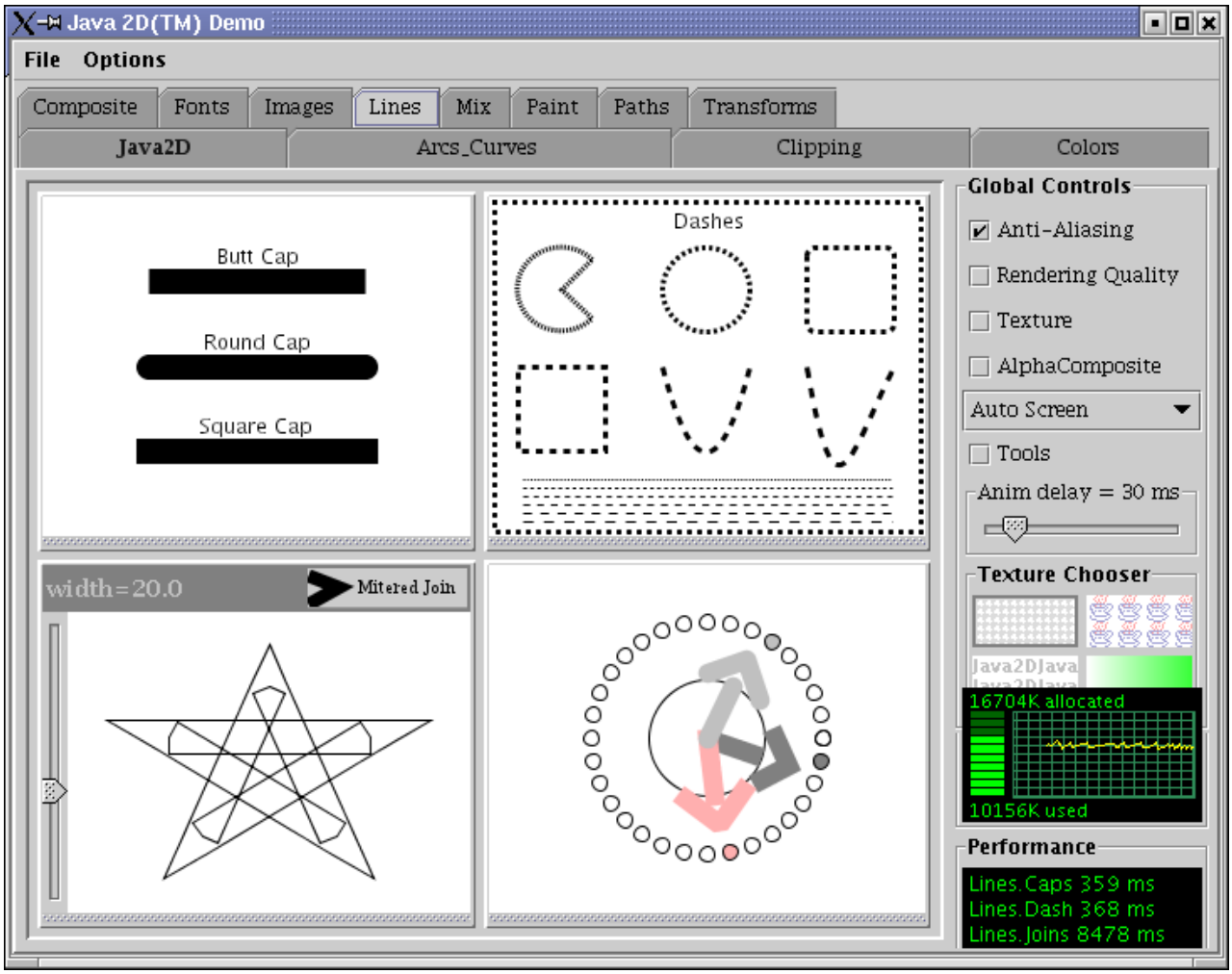
almindelig 2−pkt linjetykkelse trappeudjævning transformation x2 x3

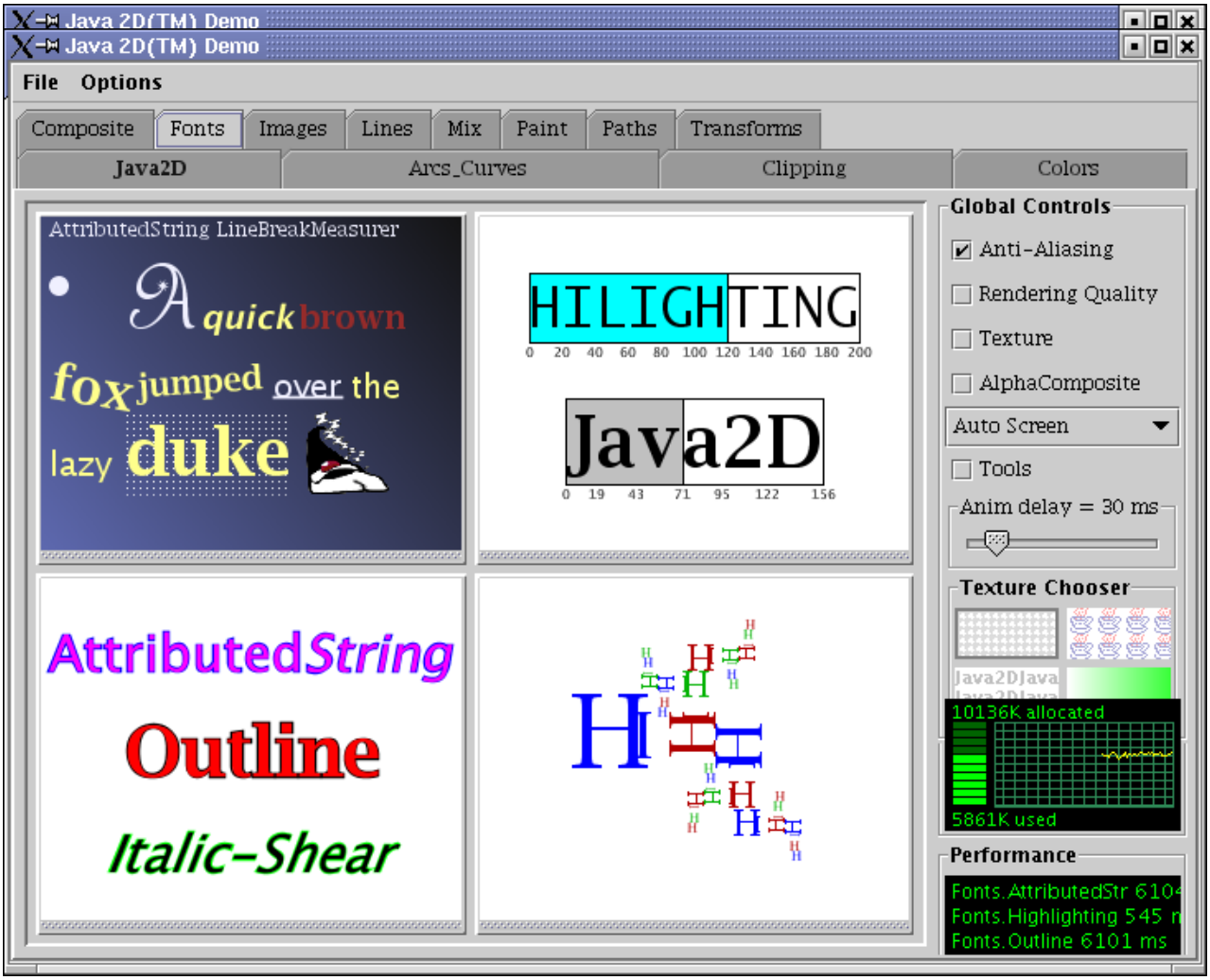
```
import java.awt.*;
import java.awt.geom.*;
public class Java2DDemo extends Frame
{
  GeneralPath fig;

  AffineTransform trans;

  BasicStroke stregtype = // 2−punktsstreg med kantede ender og runde knæk
    new BasicStroke(2, BasicStroke.CAP_SQUARE, BasicStroke.JOIN_ROUND);

  public Java2DDemo()
  {
    setBackground(Color.white);

    // Lav 'smiley' − cirkel, to øjne og glad åben mund
    fig = new GeneralPath( new Ellipse2D.Float(0, 0, 100, 100) );
    fig.append( new Ellipse2D.Float(20, 20, 10, 10), false );
    fig.append( new Ellipse2D.Float(70, 20, 10, 10), false );
    fig.append( new Arc2D.Float(10,10, 80,80, 330,−120,Arc2D.CHORD), false );

    trans = AffineTransform.getScaleInstance(0.7, 0.7); // formindsk
    trans.rotate(0.3);                                  // roter
    trans.shear(0.3,0);                                 // vrid
    trans.translate(160,−50);                           // flyt til siden
  }

  public void paint(Graphics g)
  {
    Graphics2D g2 = (Graphics2D) g;             // brug Java2D

    // koordinattransformation
    AffineTransform orgTrans = g2.getTransform();// husk original transformation
    g2.translate(10,30);

    g2.draw( fig );                             // tegn almindelig
    g2.translate( 110, 0 );

    g2.setStroke( stregtype );
    g2.draw( fig );                             // tegn med 2−pkt linjetykkelse
    g2.translate( 110, 0 );

    g2.setRenderingHint( // sæt tegnevink til trappeudjævning (antialias)
      RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
    g2.draw( fig );                             // tegn med trappeudjævning


    g2.transform( trans );
    g2.draw( fig );                             // tegn med transformation

    g2.transform( trans );
```

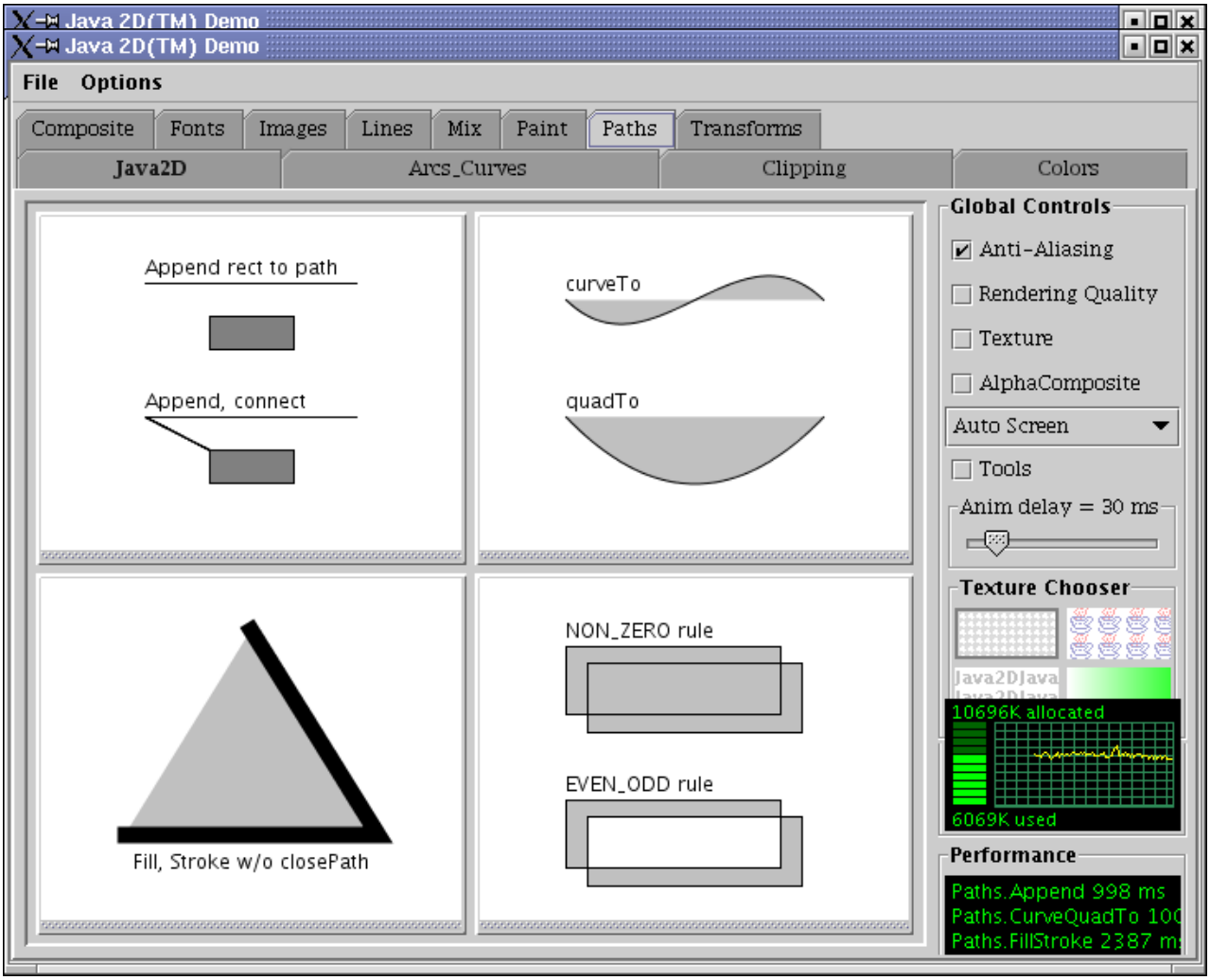
```
    g2.draw( fig );                          // tegn med transformation  x2

    g2.transform( trans );
    g2.draw( fig );                          // tegn med transformation  x3

    g2.setTransform( orgTrans );                // genskab orig. transformation
  }

  public static void main(String[] arg)
  {
    Java2DDemo vindue = new Java2DDemo();
    vindue.setSize(500,150);
    vindue.setVisible(true);
  }
}
```

5.3 Eksempel fra JDK'et

Sammen med Java følger et program, der illustrerer de fleste af mulighederne med Java2D. Det ligger i jdk1.4/demo/jfc/Java2D/Java2Demo.jar og startes ved at dobbeltklikke på jar−filen eller fra kommandolinjen skrive: java −jar Java2Demo.jar. Her ses nogle skærmbilleder:




Java 2D(TM) Demo
Java 2D(TM) Demo
File
Options
Composite
Fonts
Images
Lines
Mix
Paint
Paths
Transforms
Java2D
Arcs_Curves
Clipping
Colors
AttributedString LineBreakMeasurer
A quick brown
fox jumped over the
lazy duke
HILIGHTING
0 20 40 60 80 100 120 140 160 180 200
Java2D
0 19 43 71 95 122 156
AttributedString
Outline
Italic-Shear
Global Controls
Anti-Aliasing
Rendering Quality
Texture
AlphaComposite
Auto Screen
Tools
Anim delay = 30 ms
Texture Chooser
10136K allocated
5861K used
Performance
Fonts.AttributedStr 610
Fonts.Highlighting 545 n
Fonts.Outline 6101 ms


Java 2D(TM) Demo
Java 2D(TM) Demo
File
Options
Composite
Fonts
Images
Lines
Mix
Paint
Paths
Transforms
Java2D
Arcs_Curves
Clipping
Colors
Append rect to path
Append, connect
curveTo
quadTo
Fill, Stroke w/o closePath
NON_ZERO rule
EVEN_ODD rule
Global Controls
Anti-Aliasing
Rendering Quality
Texture
AlphaComposite
Auto Screen
Tools
Anim delay = 30 ms
Texture Chooser
10696K allocated
6069K used
Performance
Paths.Append 998 ms
Paths.CurveQuadTo 10
Paths.FillStroke 2387 m

 javabog.dk | << forrige | indhold | næste >> | programeksempler | om bogen

6 Grafiske brugergrænseflader (Swing)

Det er en fordel at have læst den første del af kapitel 19 for at have et kendskab til Model−View−Controller−arkitekturen.

Dette kapitel handler hovedsageligt om de ting i Swing, der er umiddelbart svære at forstå. Det behandler nogle af de mere avancerede aspekter af brugergrænseflader og Swing, som bl.a. JTable og MVC (Model−View−Controller−arkitekturen, beskrevet i dybden i kapitel 19).

En mere grundlæggende (om end lidt gammel) introduktion finder man i Kristian Hansens bog 'Avanceret Java−programmering', der kan hentes gratis på http://bog.ing.dk/.

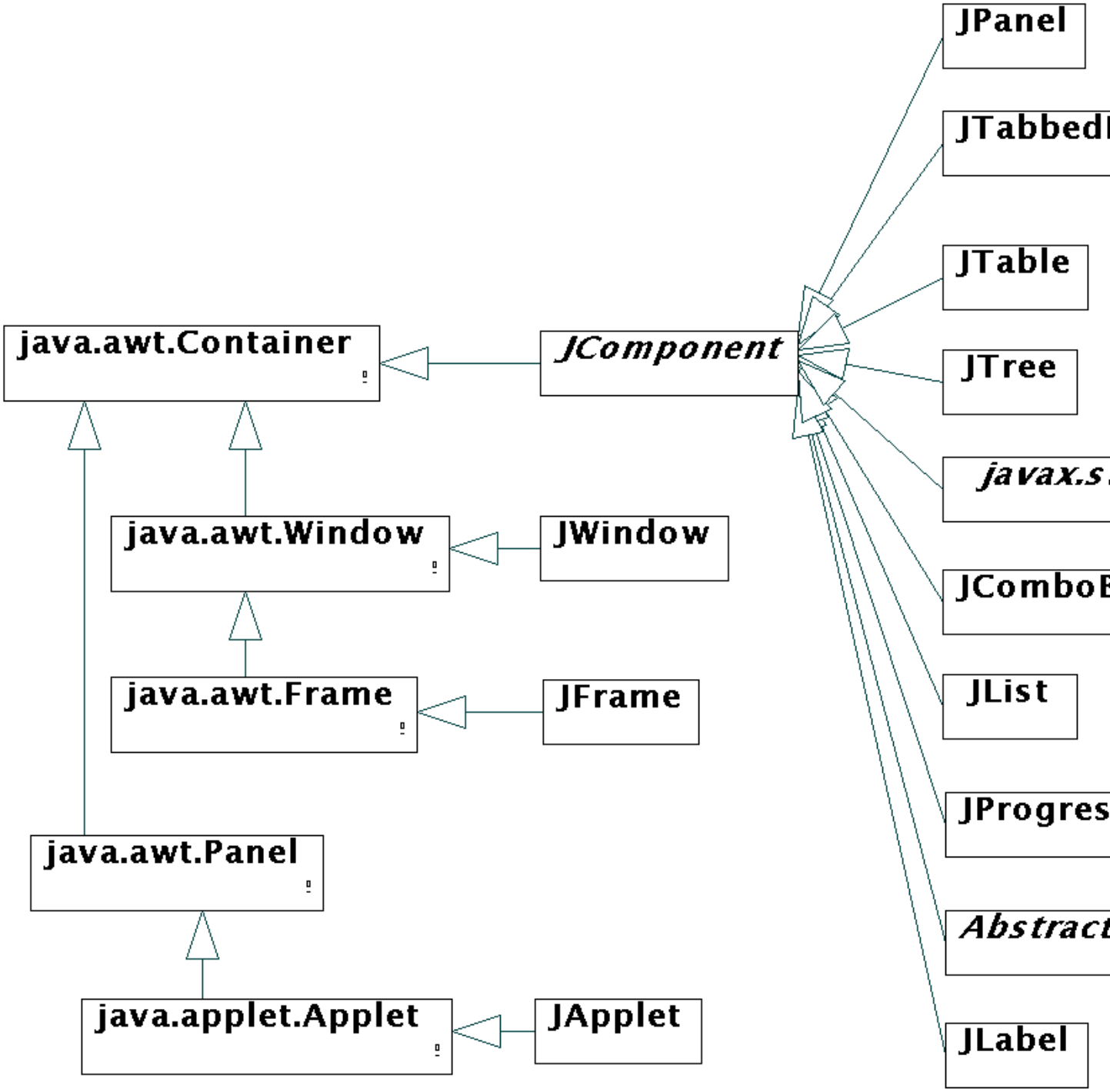
6.1 Introduktion til Swing

I Java 1.2 blev et nyt sæt komponenter føjet til standardbibliotekerne. De svarer meget til AWT−komponenterne, men de er letvægtkomponenter, dvs. de er programmeret helt i Java og er derfor mindre afhængige af det underliggende styresystem.

De kaldes Swing−komponenter og findes i pakken javax.swing, og de starter alle med et J i klassenavnet, f.eks. JButton, JCheckBox, JRadioButton, JLabel, JList, JTextField, JTextArea, JPanel, JApplet og JFrame.

Der er også flere komponenter, som ikke findes i java.awt: JComboBox, JToggleBotton, JTable (en regneark−lignende komponent til at vise og manipulere tabeldata), JTree (viser en træstruktur ligesom stifinderen i Windows) og JTextPane (kan vise og redigere tekst med formatering og indlejrede billeder, herunder HTML− og RTF−dokumenter).

Her er et klassediagram over de vigtigste Swing−komponenter:



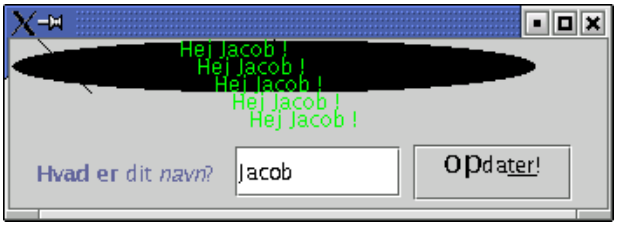
Yderst til venstre ses AWT−klasserne Container, Window, Frame, Panel og Applet. Resten er Swing−komponenter, og det ses, hvordan AWT−komponenterne alle har en pendant i Swing med et foranstillet J.

For at undgå blinkeri bruger alle Swing−containere som standard dobbelt tegnebuffer (eng.: double buffering, hvor komponenterne, når de skal gentegnes, ikke tegnes direkte på skærmen, men i en separat grafikbuffer, som derefter kopieres ind på skærmen.

6.1.1 Eksempel på forskelle mellem Swing og AWT

Dette eksempel illustrerer de væsentligste forskelle mellem Swing og AWT (forskellene er markeret med fed):

```
import java.awt.*;
import javax.swing.*;

public class SwingVindue extends JFrame
{
  JLabel labelHvadErDitNavn = new JLabel();
  JTextField textFieldNavn = new JTextField();
  JButton buttonOpdater = new JButton();
  JTextArea textAreaHilsen = new JTextArea();

  public void paint(Graphics g) {
    // vigtigt! Kald den oprindelige paint() så Swing-komponenter bliver tegnet
    super.paint(g);

    g.drawLine(0,0,50,50);
    g.fillOval(5,20,300,30);
    g.setColor(Color.green);
    String navn = textFieldNavn.getText();
    for (int i=0; i<50; i=i+10)
      g.drawString("Hej "+navn+" !",100+i,30+i);
  }

  public SwingVindue() {
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }

  private void jbInit() throws Exception {
    // Swing-komponenter tillader HTML-koder i deres tekst:
    labelHvadErDitNavn.setText("<html><b>Hvad er</b> dit <i>navn</i>?</html>");
    labelHvadErDitNavn.setBounds(new Rectangle(15, 69, 108, 15));
    textFieldNavn.setText("Jacob");
    textFieldNavn.setBounds(new Rectangle(129, 61, 95, 29));

    buttonOpdater.setText("<html><font size=+1>op</font>da<u>ter</u>!</html>");
    buttonOpdater.setBounds(new Rectangle(231, 60, 91, 32));
    textAreaHilsen.setText("Her kommer en tekst...");
    textAreaHilsen.setBounds(new Rectangle(6, 102, 316, 78));

    // vigtigt! Tilføj komponenterne til "content pane" på en Swing-container
    this.getContentPane().setLayout(null);
    this.getContentPane().add(labelHvadErDitNavn, null);
    this.getContentPane().add(textAreaHilsen, null);
    this.getContentPane().add(buttonOpdater, null);
    this.getContentPane().add(textFieldNavn, null);
  }

  public static void main(String[] arg) {
    SwingVindue vindue = new SwingVindue();
    vindue.setSize(350,120);
    vindue.setVisible(true);
  }
}
```



Som det ses af eksemplet, understøtter Swing−komponenter visning af HTML−kode, så man kan få knapper og etiketter, hvor dele af teksten er f.eks. fed, kursiv eller af en anden størrelse.

Bruger man Swing−komponenter og samtidig har defineret paint()−metoden, skal man huske at kalde den oprindelige paint()−metode, ellers "forsvinder" Swing−komponenterne:

```
  public void paint(Graphics g)
  {
    // vigtigt! Kald den oprindelige paint() så Swing-komponenter bliver tegnet
    super.paint(g);

    // herunder kommer vores egen tegningskode
  }
```

Det skyldes, at Swing−komponenter bliver tegnet fra containerens (superklassens) paint()−metode. Bliver paint() ikke kaldt i superklassen, bliver Swing−komponenterne ikke tegnet.

En anden vigtig forskel er, at man ikke føjer komponenter direkte til containeren, men til dens "content pane", med this.getContentPane().add(komponent).

I AWT skrev man blot this.add(komponent).

6.2 MVC i Swing− og AWT−komponenter

Selvom MVC (Model−View−Controller−arkitekturen beskrevet i kapitel 19) egentlig er beregnet på at adskille et programs logik fra dets brugergrænseflade, kan tankegangen også bruges internt i en grafisk komponent til at adskille de data, som komponenten repræsenterer, fra fremvisningen og ændringen af dem.

Lad os se på eksempelvis et indtastningsfelt (TextField eller JTextField):

- **Modellen** er selve teksten, brugeren redigerer i.
- **Præsentationen** er at vise teksten på skærmen.
- **Kontroldelen** er fortolkningen af tastetryk og at omsætte dem til tegn, der bliver sat ind i teksten.

6.2.1 AWT−komponenter

AWT−komponenterne, de oprindelige simple komponenter, som Java blev født med, fungerer som bro til platformens egne grafiske komponenter, sådan at når et program opretter f.eks. et Button−objekt, vil der faktisk blive oprettet en Windows−knap under Windows, en Linux−knap under Linux, en MacOS−knap på Macintosh osv.. Button−objektet vil så fungere som platformsuafhængig facade ned mod den meget platformsspecifikke knap.

Det er derfor ikke muligt at påvirke, hvordan en AWT−komponent virker, og model, præsentation og kontrol−del er uadskillelige. Det gør dem relativt simple at arbejde med, men også relativt ufleksible, hvis man skulle ønske at ændre dem, så de virker lidt anderledes.

F.eks. er det meget besværligt (hvis ikke umuligt) at have to tekstfelter, der redigerer i den samme tekst, eller at lave et tekstfelt, som kun viser store bogstaver.

6.2.2 Swing−komponenter

Swing−komponenter er implementeret i ren Java, og derfor har det været muligt at gøre dem langt mere fleksible end AWT−komponenterne.

I Swing−komponenterne er datamodellen skilt ud fra komponenten. Det vil sige, at programmøren har mulighed for at selv bestemme, hvilket objekt der skal bruges som datamodel. Præsentationen og kontroldelen har man også adgang til, omend mere begrænset.

6.3 Datamodellen i Swing−komponenter

Modellen i en komponent er de data, som komponenten repræsenterer. Hvilken slags model der anvendes afhænger meget af komponenten.

6.3.1 Tekstfelter (interfacet Document)

I JTextField, JTextArea og de andre tekstredigeringsfelter er datamodellen et objekt af typen Document, og læses hhv. sættes med metoderne getDocument() og setDocument():

```
public Document getDocument()
public void setDocument(Document doc)
```

Document er et interface, der er ret indviklet af finde rundt i, fordi det er beregnet til også at understøtte RTF− og HTML−formaterede dokumenter (JEditorPane og JTextPane), så det vil vi ikke beskrive i detaljer.

Eksempel: To tekstfelter, der deler det samme dokument

Har man to tekstfelter...

```
  JTextField jTextField1 = new JTextField();
  JTextField jTextField2 = new JTextField();
```

kan man få dem til at redigere *i den samme tekst* ved at sætte den ene til at bruge den andens model:

```
  Document datamodel = jTextField1.getDocument();
  jTextField2.setDocument( datamodel );
```

Eksempel: Et tekstfelt, der kun accepterer store bogstaver

Ved at lave sin egen datamodel kan man selv bestemme præcist hvad der kan ske med teksten.

I stedet for at lave en implementation af Document−interfacet fra grunden (det er ret indviklet) vælger man ofte at arve fra standardmodellen (der hedder PlainDocument) og bare tilsidesætte den metode, der kaldes for at indsætte tekst i dokumentet:

```
import javax.swing.text.*;
public class StortDokument extends PlainDocument
{
  public void insertString(int o, String s, AttributeSet a)
    throws BadLocationException
```

```
  {
    // kald den oprindelige metode med strengen med store bogstaver
    super.insertString(o, s.toUpperCase(), a);
    // log ændringen, så vi kan følge med i hvad der sker
    System.out.println("insertString("+o+", '"+s+"') kaldt.\n");
  }
}
```

Ovenstående klasse kan bruges som datamodel i et tekstfelt med koden:

```
  Document datamodel = new StortDokument();
  jTextField2.setDocument( datamodel );
```

Af andre klasser, der implementerer Document−interfacet, kan nævnes DefaultStyledDocument og HTMLDocument. I disse dokumenttyper har tegnene attributter såsom skriftstørrelse, fed, kursiv, understreget osv. (beregnet til JEditorPane og JTextPane).

6.3.2 Tabeller (interfacet TableModel)

Tabeller (JTable) bruger en TableModel som model og har metoder til at læse/sætte den:

```
public TableModel getModel()
public void setModel(TableModel dataModel)
```

JTable kalder modellens metoder for at vide, hvad den skal vise.

Modellen har interfacet:

```
package javax.swing.table;

public interface TableModel
{
  public int getRowCount();
  public int getColumnCount();
  public String getColumnName(int kollonneindeks);
  public Class getColumnClass(int kollonneindeks)
  public boolean isCellEditable(int rækkeindeks, int kollonneindeks);
  public Object getValueAt(int rækkeindeks, int kollonneindeks);
  public void setValueAt(Object værdi, int rækkeindeks, int kollonneindeks);
  public void addTableModelListener(TableModelListener lytter);
  public void removeTableModelListener(TableModelListener lytter);
}
```

En måde at lave en tabel på er altså at lave en klasse, der implementerer TableModel−interfacet, og så bruge den som datamodel. Det er dog lidt besværligt, fordi man selv skal holde styr på lyttere og implementere addTableModelListener() og removeTableModelListener().

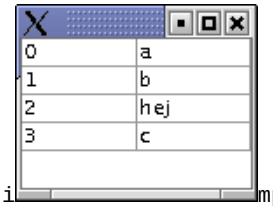
Man kan også vælge at arve fra AbstractTableModel (eller DefaultTableModel), der implementerer TableModel og selv sørger for at håndtere lyttere.

Her er en klasse, der får et endimensionalt array af strenge til at fungere som en TableModel, så den kan vises af JTable (sådan en hjælpeklasse, der får 'noget til at passe ind i noget', kaldes en adapter, se afsnit 17.2).

Tabellen kommer til at vise to kolonner, første kolonne med rækkenummeret og anden kolonne er indholdet af arrayet på det pågældende rækkenummer.

```
import javax.swing.table.*;

public class StrengArrayTableModel extends AbstractTableModel
{
  private String[] data;
  public  StrengArrayTableModel(String[] dat) { data = dat; }

  public int getColumnCount()    { return 2; }
  public int getRowCount()       { return data.length ;}
  public Object getValueAt(int ræk, int kol)
  {
    if (kol == 0) return new Integer(ræk); // første kolonne er indeks
    else return data[ræk];                 // anden kolonne er strengenes værdi
  }
}
```

Her er et lille program, der benytter adapteren til at vise et array af strenge:



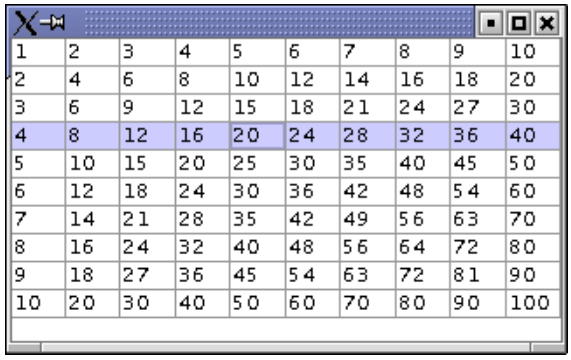

```
import javax.swing.*;

public class BenytStrengArrayTableModel
{
```

```
  public static void main(String[] args)
  {
    JFrame f = new JFrame();
    JTable t;
    f.setSize(200, 200);

    String[] arr = {"a","b","hej","c"}; // strenge
    t = new JTable(new StrengArrayTableModel(arr));

    f.getContentPane().add(t);
    f.setVisible(true);
  }
}
```

Man kan naturligvis også bare *lade*, som om man viser nogle data uden at hente dem fra nogen datastruktur. Eksempelvis er her en datamodel, der repræsenterer den lille tabel. Den har 10 rækker og 10 kolonner, og hver celle har samme værdi som rækkenummer+1 gange kolonnenummer+1.

```
import javax.swing.table.*;

public class DenLilleTabel extends AbstractTableModel
{
  public int getColumnCount()    { return 10; }
  public int getRowCount()       { return 10;}
  public Object getValueAt(int r, int k) { return new Integer((r+1)*(k+1)); }
}
```

Vist i en JTable giver klassen følgende data:



1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

En anden måde er at angive et todimensionalt array (eller en Vector af Vector−objekter) med startværdier i konstruktøren til JTable. Da vil den selv oprette en datamodel, der repræsenterer data.

6.3.3 Knapper (interfacet ButtonModel)

Alle knapper og lignende, dvs. JButton, JToggleButton, JMenuItem, JMenu, JCheckbox og JRadiobox bruger en ButtonModel som datamodel.

Modellen læses og sættes med metoderne:

```
ButtonModel getModel()
void setModel(ButtonModel nyModel)
```

Selve ButtonModel er et interface med metoderne

```
public interface ButtonModel extends ItemSelectable
{
  public void setArmed(boolean b);
  public void setSelected(boolean b);
  public void setEnabled(boolean b);
  public void setPressed(boolean b);
  public void setRollover(boolean b);
  public void setMnemonic(int key);
  public int  getMnemonic();
  public void setActionCommand(String s);
  public String getActionCommand();
  public void setGroup(ButtonGroup group);

  // ... flere metoder
}
```

6.3.4 Andre komponenter

I de fleste andre hedder metoderne til at aflæse/udskifte datamodellen hhv. getModel() og setModel().

JComboBox bruger modellen ComboBoxModel:

```
void setModel(ComboBoxModel nyModel)
ComboBoxModel getModel()
```

JList bruger modellen ListModel:

```
public ListModel getModel()
public void setModel(ListModel nyModel)
```

JProgressBar, JScrollBar og JSlider, dvs. komponenter, der på en eller anden måde grafisk skal vise en værdi i et interval, bruger modellen BoundedRangeModel:

```
public BoundedRangeModel getModel()
public void setModel(BoundedRangeModel nyModel)
```

JTree bruger modellen TreeModel:

```
public TreeModel getModel()
public void setModel(TreeModel nyModel)
```

... og så videre.

Enhver Swing−komponent har altså en datamodel, der beskriver komponentens tilstand.

6.3.5 Opgaver

1. Lav et program med med to tekstfelter, der deler samme dokument.
2. Tilføj et tekstfelt, der kun accepterer store bogstaver til programmet.
3. Prøv at lade to knapper dele datamodel. Hvad sker der så? Hvad med en menuindgang og en knap?

6.3.6 Løsning

6.4 Præsentationsdelen i Swing

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

6.4.1 Lister (ListCellRenderer)

For JComboBox og JList skal fremviser−objektet implementere ListCellRenderer:

```
package javax.swing;

public interface ListCellRenderer
{
  /**
   * Return a component that has been configured to display the specified
   * value. That component's <code>paint</code> method is then called to
   * "render" the cell.  If it is necessary to compute the dimensions
   * of a list because the list cells do not have a fixed size, this method
   * is called to generate a component on which <code>getPreferredSize</code>
   * can be invoked.
   *
   * @param list The JList we're painting.
   * @param value The value returned by list.getModel().getElementAt(index).
   * @param index The cells index.
   * @param isSelected True if the specified cell was selected.
   * @param cellHasFocus True if the specified cell has the focus.
   * @return A component whose paint() method will render the specified value.
   *
   * @see JList
   * @see ListSelectionModel
   * @see ListModel
   */
  Component getListCellRendererComponent(
    JList list,
    Object value, int index,
    boolean isSelected, boolean cellHasFocus);
}
```

Fremvisningen sættes med metoden setCellRenderer(ListCellRenderer cellRenderer).

Har man ikke angivet nogen fremviser, bruges klassen DefaultListCellRenderer, der kan vise ikoner (Icon) og tekster. Vil man lave sin egen fremvisning, anbefales det, at man arver fra denne og omdefinerer getListCellRendererComponent() eller paint().

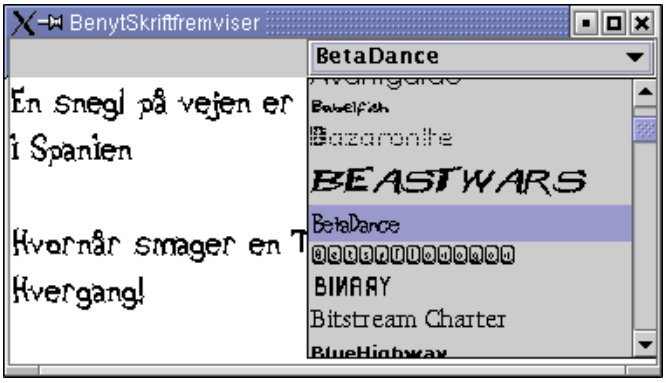
Eksempel på præsentationsdel af en liste

Her er et eksempel på en liste, der viser alle sine punkter overstreget. Det er gjort ved at arve fra DefaultListCellRenderer og omdefinere paint()−metoden til at tegne en skrå streg.

```
import java.awt.*;
import javax.swing.*;
public class OverstregedeCeller extends DefaultListCellRenderer {
  public void paint(Graphics g) {
    super.paint(g);                          // tegn cellen
    g.drawLine(0,0,getWidth(),getHeight());  // tegn en streg hen over den
  }
}
```

6.4.2 Eksempel: Præsentation af skrifter i en liste

Det følgende eksempel på en præsentationsdel til lister (ListCellRenderer) viser en liste over skrifttyper, hvor hver skrifttype bliver vist som den rent faktisk ser ud på skærmen:

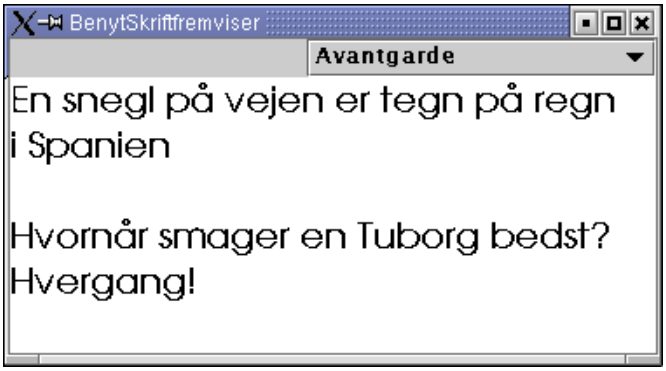


Fremvisningen sker ved at arve fra DefaultListCellRenderer og tilsidesætte metoden getListCellRendererComponent():

```
import java.awt.*;
import java.util.*;
import javax.swing.*;

public class Skriftfremviser extends DefaultListCellRenderer
{
  Map skrifter = new HashMap(); // afbildning fra skriftnavn til Font-objekt

  public Component getListCellRendererComponent(
    JList liste,
    Object værdi, int index,
    boolean valgt, boolean harFokus)
  {
    Component visningsKomponent =
      super.getListCellRendererComponent(liste, værdi, index, valgt, harFokus);

    System.out.println(index +" "+ værdi); // se hvor ofte metoden bliver kaldt

    if (værdi != null)
    {
      Font skrifttype = (Font) skrifter.get(værdi);
      if (skrifttype==null)
      {
        skrifttype = new Font( (String) værdi, Font.PLAIN, 14); // indlæs skrift
        skrifter.put(værdi, skrifttype);                        // ..og husk den
      }
      visningsKomponent.setFont(skrifttype);
    }

    return visningsKomponent;
  }
}
```

I getListCellRendererComponent() kalder vi først superklassens oprindelige metode for at få ordnet andre ting, såsom at afgøre hvilken tekst, der skal vises og om cellen (indgangen) skal have en speciel farve, fordi den er valgt/har fokus.

Derefter prøver vi at indlæse den pågældende skrifttype. For at undgå at oprette alt for mange Font−objekter laver vi en afbildning fra skriftnavne til Font−objekter og cacher Font−objekterne.



Her er et program, der benytter Skriftfremviser.

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class BenytSkriftfremviser extends JFrame implements ActionListener
{
  JComboBox jComboBox1 = new JComboBox();
  JTextArea jTextArea1 = new JTextArea();

  public BenytSkriftfremviser() {
    setTitle("BenytSkriftfremviser");

    String[] skriftnavne = GraphicsEnvironment.
      getLocalGraphicsEnvironment().getAvailableFontFamilyNames();

    jTextArea1.setLineWrap(true);
    jTextArea1.setText("En snegl på vejen er tegn på regn\ni Spanien\n\n"
      +"Hvornår smager en Tuborg bedst?\nHvergang!");

    jComboBox1.setModel(new DefaultComboBoxModel(skriftnavne));
    jComboBox1.setRenderer(new Skriftfremviser());
    jComboBox1.addActionListener(this);

    JPanel jPanel1 = new JPanel();
    jPanel1.setLayout(new BorderLayout());
    jPanel1.add(jComboBox1, BorderLayout.EAST);

    getContentPane().setLayout(new BorderLayout());
    getContentPane().add(jPanel1, BorderLayout.NORTH);
    getContentPane().add(jTextArea1, BorderLayout.CENTER);
  }

  public void actionPerformed(ActionEvent e) { // kaldes når valglisten ændres
    String skriftnavn = (String) jComboBox1.getSelectedItem();
    System.out.println(skriftnavn + " " + e);
    jTextArea1.setFont( new Font( skriftnavn, Font.PLAIN, 20) );
  }

  public static void main(String[] args) {
    BenytSkriftfremviser vindue = new BenytSkriftfremviser();
    vindue.pack();
    vindue.setVisible(true);
  }
}
```

6.4.3 Tabeller (TableCellRenderer)

For JTable kan man sætte fremvisningen på hver kolonne (hentet med getColumn() og navnet på kolonnen) med metoden: setRenderer(TableCellRenderer fremviser).

```
package javax.swing.table;

public interface TableCellRenderer
{
  /**
   *  Returns the component used for drawing the cell.  This method is
   *  used to configure the renderer appropriately before drawing.
   *
   * @param table     the <code>JTable</code> that is asking the
   *          renderer to draw; can be <code>null</code>
   * @param value     the value of the cell to be rendered.  It is
   *          up to the specific renderer to interpret
   *          and draw the value.  For example, if
   *          <code>value</code>
   *          is the string "true", it could be rendered as a
   *          string or it could be rendered as a check
   *          box that is checked.  <code>null</code> is a
   *          valid value
   * @param isSelected  true if the cell is to be rendered with the
   *          selection highlighted; otherwise false
```

```
   * @param hasFocus    if true, render cell appropriately.  For
   *          example, put a special border on the cell, if
   *          the cell can be edited, render in the color used
   *          to indicate editing
   * @param row     the row index of the cell being drawn.  When
   *          drawing the header, the value of
   *          <code>row</code> is -1
   * @param column    the column index of the cell being drawn
   */

  Component getTableCellRendererComponent(
    JTable table,
    Object value,
    boolean isSelected, boolean hasFocus,
    int row, int column);
}
```

Her er et eksempel på et præsentationsobjekt oprettet direkte ud fra interfacet. Det er beregnet til at vise Color−objekter. I getTableCellRendererComponent() konfigureres en JLabel derfor til at vise den pågældende farve, hvorefter denne JLabel returneres:

```
import javax.swing.*;
import javax.swing.table.*;
import java.awt.*;

public class Farvepraesentation implements TableCellRenderer
{
  JLabel jLabel = new JLabel();          // genbrug den samme komponent

  public Component getTableCellRendererComponent(JTable tabel, Object værdi,
          boolean erValgt, boolean harFokus, int række, int kolonne)
  {
    if (værdi instanceof Color)          // er værdien af type Double?
    {
      Color f = (Color) værdi;
      jLabel.setBackground(f);           // hele baggrunden viser farven
      jLabel.setForeground(f.darker());  // tekstfarven er lidt mørkere
      jLabel.setOpaque(true);            // uigennemsigtig så baggrunden kan ses
      jLabel.setText("farve");           // vis bare en eller anden tekst
    } else {
      jLabel.setBackground(Color.white); // er objektet ikke en farve, så vis
      jLabel.setForeground(Color.black); // det som toString() giver
      jLabel.setText(værdi.toString());
    }
    return jLabel;                       // returnér komponenten der skal vises
  }
}
```

I stedet for at implementere TableCellRenderer direkte tilrådes det dog, at man arver fra klassen DefaultTableCellRenderer og definerer metoden setValue(). Det følgende eksempel præsenterer tal (af typen Double) med sort skrift, hvis de er positive, og rød skrift, hvis de er negative:

```
import javax.swing.*;
import javax.swing.table.*;
import java.awt.*;

public class Talpraesentation extends DefaultTableCellRenderer
{
  public void setValue(Object værdi)
  {
    if (værdi instanceof Double) {
      if (((Double) værdi).doubleValue()<0) setForeground(Color.red);
      else setForeground(Color.blue);
    }
    else setForeground(Color.black);

    setText(værdi.toString());
  }
}
```

Som det ses, er det altså simplere at arve fra DefaultTableCellRenderer (som implementerer TableCellRenderer). Samtidigt er DefaultTableCellRenderer optimeret beregnet på den ret specielle brug, som JTable gør af den, så det giver også mere effektiv kode.

Begge klasser kan ses brugt i afsnit 6.6.2.

6.4.4 Træer (TreeCellRenderer)

Tilsvarende har JTree metoden setCellRenderer(TreeCellRenderer fremviser) og en DefaultTreeCellRenderer, der bruges, hvis en anden fremviser ikke specificeres.

6.4.5 Opgaver

1. Prøv Skriftfremviser−eksemplet.
2. Lav din egen Farvefremviser, der giver brugeren mulighed for at vælge mellem et antal farver.

6.5 Kontroldelen af Swing−komponenter

På tilsvarende måde som med Renderer findes der (for tabeller og træer) også klasser og metoder til at specificere, hvordan redigering af indholdet af komponenten skal ske .

Måden at arbejde med disse svarer ret meget til måden at arbejde med præsentations−delen. Kontrol−delene skal implementere interfacet CellEditor:

```
public interface CellEditor {
  public Object getCellEditorValue();
  public boolean isCellEditable(EventObject anEvent);
  public boolean shouldSelectCell(EventObject anEvent);
  public boolean stopCellEditing();
  public void cancelCellEditing();
  public void addCellEditorListener(CellEditorListener l);
  public void removeCellEditorListener(CellEditorListener l);
}
```

I stedet for at skrive sin egen editor anbefales det at anvende (eller eventuelt arve fra) klassen DefaultCellEditor (beskrevet nedenfor).

Det er der eksempler på i afsnit 6.6.2.

6.5.1 Tabeller (TableCellEditor)

Interfacet TableCellEditor (der udvider CellEditor) skal implementeres hvis man selv vil bestemme hvordan redigeringen af en tabel skal foregå. Det har metoden:

```
  Component getTableCellEditorComponent(
    JTable table,
    Object value,
    boolean isSelected,
    int row,
    int column
  );
```

som skal defineres til at returnere en (korrekt konfigureret) komponent, der står for redigeringen af den pågældende celle.

6.5.2 Træer (TreeCellEditor)

Interfacet TreeCellEditor (der udvider CellEditor) skal implementeres hvis man selv vil bestemme, hvordan redigeringen af et træ skal foregå. Det har metoden:

```
  Component getTreeCellEditorComponent(
    JTree tree,
    Object value,
    boolean isSelected,
    boolean expanded,
    boolean leaf,
    int row
  );
```

som skal defineres til at returnere en (korrekt konfigureret) komponent, der står for redigeringen af den pågældende gren af træet.

6.5.3 Standardredigering med DefaultCellEditor

DefaultCellEditor implementerer både TableCellEditor og TreeCellEditor. Den bruges af både JTable og JTree, hvis ingen anden editor er angivet. I dens konstruktør kan man angive en JTextField, JCheckBox eller JComboBox, som da vil blive brugt til redigeringen.

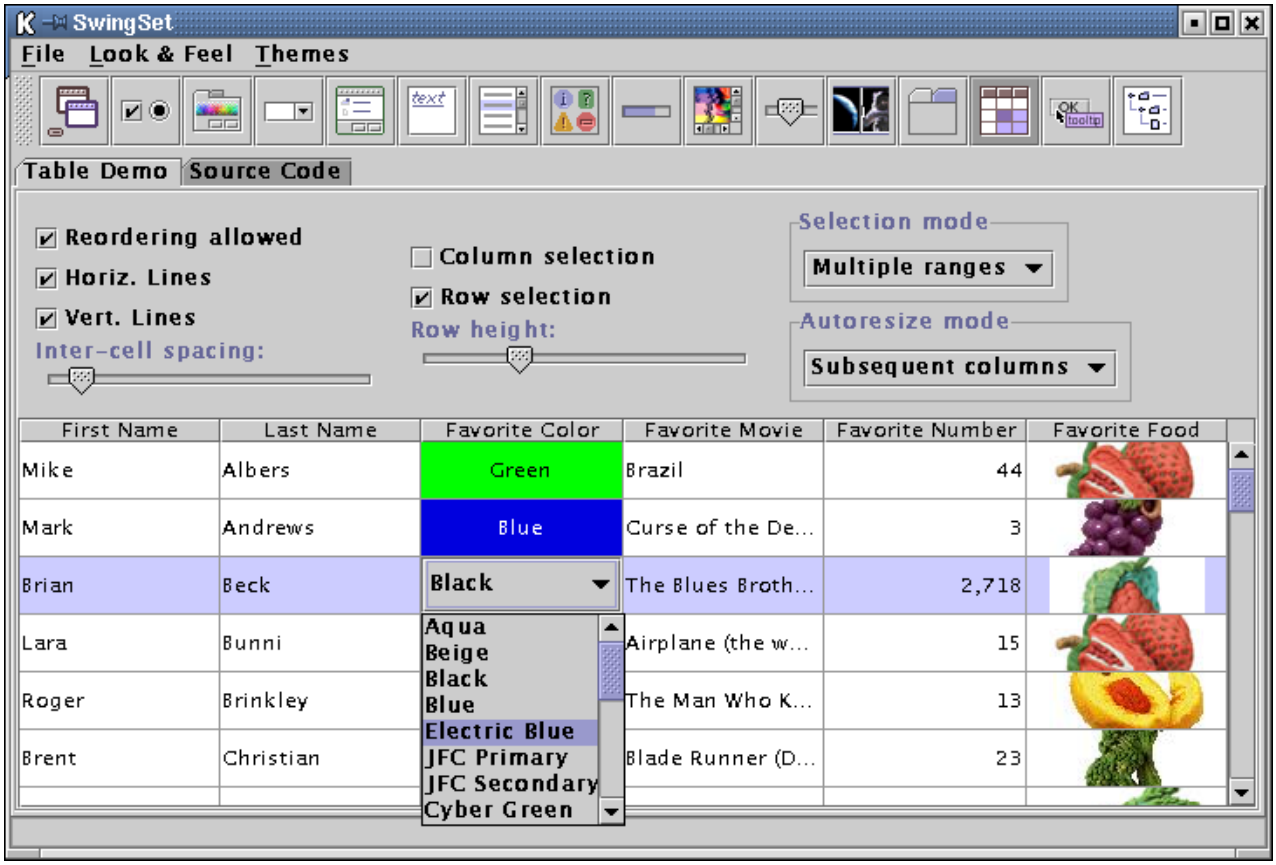
6.6 Eksempler

Sammen med JDK'en følger der en række eksempler (ligger i f.eks. jdk1.4/demo/). Det er en virkelig god idé at prøve nogle af disse eksempler, da de illustrerer de muligheder, man har som programmør, og samtidig følger kildeteksten med til alle eksemplerne, så man kan se, hvordan det er programmeret.

6.6.1 SwingSet2−demo af JTable

Et eksempel, der illustrerer de fleste af mulighederne med Swing−komponenter, hedder SwingSet2. Det ligger i jdk1.4/demo/jfc/SwingSet2 og startes ved at dobbeltklikke på jar−filen eller fra kommandolinjen skrive: java −jar SwingSet2.jar.

Herunder ses skærmbilledet af SwingSet2's demo af JTable:



Der er lavet kontroller til de fleste ting, kan "dreje på" for en JTable, bl.a.:

```
// kan brugeren bytte om på rækkefølgen kolonnerne vises i ved at trække i dem
jTable.getTableHeader().setReorderingAllowed(flag);

// skal der vises vandrette/lodrette linjer mellem cellerne
jTable.setShowHorizontalLines(flag);    jTable.setShowVerticalLines(flag);

// afstand mellem cellerne
jTable.setIntercellSpacing(new Dimension(bredde, højde));

// skal der vælges hele kolonner/rækker når brugeren klikker i en celle
jTable.setColumnSelectionAllowed(flag); jTable.setRowSelectionAllowed(flag);

// rækkehøjden
jTable.setRowHeight(height);
```

Kildeteksten til SwingSet2 er lidt svær at overskue, da den samtidig demonstrerer internationalisering (programmet kan også vises på kinesisk!), tastaturgenveje og en masse andet.

6.6.2 Eksempel med JTable

Herunder ses et simplere eksempel på brug af JTable (inspireret af SwingSet2).


```
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.table.*;

public class BenytJTable extends JFrame {
  JTable jTable1 = new JTable();
  JScrollPane jScrollPane1 = new JScrollPane();

  public static void main(String[] args)  {
```

```
  BenytJTable vindue = new BenytJTable(); // opret vinduet
  vindue.pack();                          // tilpas størrelsen til indholdet
  vindue.setVisible(true);                // vis vinduet
}

private void jbInit() throws Exception {
  this.getContentPane().add(jScrollPane1, BorderLayout.CENTER);
  jScrollPane1.getViewport().add(jTable1, null);
}

public BenytJTable() {
  try {
    jbInit();
  }
  catch(Exception e) {
    e.printStackTrace();
  }

  // Hjælpevariabler
  ImageIcon æble     = new ImageIcon("apple.jpg",     "æble");
  ImageIcon asparges = new ImageIcon("asparagus.jpg", "asparges");
  ImageIcon banan    = new ImageIcon("banana.jpg",    "banan");
  ImageIcon broccoli = new ImageIcon("broccoli.jpg",  "broccoli");
  ImageIcon gulerod  = new ImageIcon("carrot.jpg",    "gulerod");
  ImageIcon kiwi     = new ImageIcon("kiwi.jpg",      "kiwi");
  ImageIcon løg      = new ImageIcon("onion.jpg",     "løg");
  Boolean ja  = Boolean.TRUE;
  Boolean nej = Boolean.FALSE;

  ////////////////////////////////////////////////
  // Data
  ////////////////////////////////////////////////
  final String[] kolonner =
    {"Navn",   "Yndlingsfarve", "Yndlingstal","Yndlingsret","Enlig?","Sexet?"};

  final Object[][] data = {
    {"Hans Petersen",   Color.blue,     new Double(44),   æble,    ja,  nej},
    {"Jacob Sørensenn", Color.yellow,   new Double(42),   kiwi,    ja,  ja},
    {"Grethe Ibsen",    Color.blue,     new Double(-37),  løg,     nej, nej},
    {"Kurt Ibsen",      Color.darkGray, new Double(1),    banan,   nej, nej},
    {"Karin Ibsen",     Color.green,    new Double(10.5), gulerod, nej, ja}
  };

  jTable1.setRowHeight(2*æble.getIconHeight()/3); // sæt højden af cellerne

  ////////////////////////////////////////////////
  // Oprettelse af datamodellen
  ////////////////////////////////////////////////

  /* Man kan oprette en datamodel direkte ud fra interfacet...
  class DatamodelFraInterface implements TableModel
  {
    public int getColumnCount()              { return kolonner.length; }
    public int getRowCount()                 { return data.length; }
    public Object getValueAt(int r, int k)   { return data[r][k]; }
    public String getColumnName(int k)       { return kolonner[k]; }
    public Class getColumnClass(int k)       { return data[0][k].getClass();}
    public boolean isCellEditable(int r, int k)    { return k != 1; }
    public void setValueAt(Object v, int r, int k) { data[r][k]=v; }
    public void addTableModelListener(TableModelListener p0) {}    // hmm...
    public void removeTableModelListener(TableModelListener p0) {} // hmm...
  };
  jTable1.setModel(new DatamodelFraInterface());
  */

  // eller oprette fra en hjælpeklasse der sørger for bl.a. hændelser
  class DatamodelFraHjaelpeklasse extends AbstractTableModel
  {
    public int getColumnCount()               { return kolonner.length; }
    public int getRowCount()                  { return data.length; }
    public Object getValueAt(int r, int k)    { return data[r][k]; }
    public String getColumnName(int k)        { return kolonner[k]; }
    public Class getColumnClass(int k)        { return data[0][k].getClass();}
    public boolean isCellEditable(int r, int k)   { return k != 1; }
    public void setValueAt(Object v, int r, int k){ data[r][k]=v; }
  };
  jTable1.setModel(new DatamodelFraHjaelpeklasse());

  // ... eller bruge standardklassen DefaultTableModel der sørger for
  // det hele (men ikke altid helt som man vil ha' det)...
  //jTable1.setModel(new DefaultTableModel(data,kolonner));

  ////////////////////////////////////////////////
  // Oprettelese af præsentationsobjekter
  ////////////////////////////////////////////////

  // brug et Farvepraesentation-objekt til kolonnen "Yndlingsfarve"
  TableCellRenderer farvepræsentation = new Farvepraesentation();
  jTable1.getColumn("Yndlingsfarve").setCellRenderer( farvepræsentation );

  // brug et Talpraesentation-objekt til kolonnen "Yndlingstal"
  TableCellRenderer talpræsentation = new Talpraesentation();
```

```
    jTable1.getColumn("Yndlingstal").setCellRenderer( talpræsentation );

    ////////////////////////////////////////////////
    // Oprettelse af redigeringsobjekter
    ////////////////////////////////////////////////

    // Lav en valgliste til at vælge mellem retterne.
    JComboBox comboBox = new JComboBox();
    comboBox.addItem(æble);
    comboBox.addItem(asparges);
    comboBox.addItem(banan);
    comboBox.addItem(broccoli);
    comboBox.addItem(gulerod);
    comboBox.addItem(kiwi);
    comboBox.addItem(løg);
    jTable1.getColumn("Yndlingsret").setCellEditor(
                                          new DefaultCellEditor(comboBox));

    // Og en til vælge mellem enlig og ikke-enlig.
    comboBox = new JComboBox();
    comboBox.addItem( new Boolean(true));
    comboBox.addItem( new Boolean(false));
    jTable1.getColumn("Enlig?").setCellEditor(new DefaultCellEditor(comboBox));
  }
}
```

6.7 Avanceret: Udseendet af Swing

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

6.7.1 Swing-komponenters standardudseender

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

6.7.2 Andre udseender

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

6.7.3 Kunststoff-udseendet

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

6.7.4 Alloys-udseendet

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

6.7.5 Opgaver

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

7 Internationalisering

7.1 Internationale programmer

Når et program skal anvendes af flere kulturer og sprog, opstår behov for, at programtekster, beløb og dato angives i de pågældende landes sprog, og man må i gang med at internationalisere og lokaliseret programmet.

Internationalisering (eng.: Internationalization, også kaldet I18N) består i at gøre programmet sprogneutralt, ved at sørge for at al formatering og fortolkning af tal−, beløbs−, dato− og tidsangivelser sker afhængigt af sproget, og at al sproglig tekst er flyttet til resursefiler.

Lokalisering består i at oversætte resursefilerne til et bestemt sprog.

7.1.1 Formatering af tidspunkter

DateFormat formaterer Date−objekter til strenge (og den anden vej).

```
import java.text.*;
import java.util.*;
public class BenytDateFormat
{
  public static void main(String arg[])
  {
    DateFormat klformat, datoformat, dkf;
    klformat   = DateFormat.getTimeInstance(DateFormat.MEDIUM);
    datoformat = DateFormat.getDateInstance(DateFormat.FULL);
    dkf = DateFormat.getDateTimeInstance(DateFormat.MEDIUM,DateFormat.SHORT);

    Date tid = new Date();
    System.out.println( tid );
    System.out.println( "Kl   :"+ klformat.format(tid) );
    System.out.println( "Dato :"+ datoformat.format(tid) );
    System.out.println( "Tid  :"+ dkf.format(tid) );
  }
}
```

```
Wed Feb 05 14:23:46 GMT+00:00 2003
Kl   :14:23:46
Dato :5. februar 2003
Tid  :05−02−2003 14:23
```

Læg for det første mærke til, at toString() på Date ikke er lokaliseret. Den bør kun bruges til test−udskrifter og logning og ikke i tekst, som brugeren skal læse.

Ovenstående program er kørt med danske sprogindstillinger. Med amerikanske sprogindstillinger bliver uddata i stedet:

```
Mon Dec 03 13:27:57 GMT+01:00 2001
Kl:   1:27:57 PM
Dato: Monday, December 3, 2001
Tid:  Dec 3, 2001 1:27 PM
```

Det ses, at det afhænger en del af sproget, præcist hvordan tider bliver formateret (f.eks. er ugedag med i den amerikanske tekst, men ikke i den danske).

Ønsker man som programmør fuld kontrol over, hvordan teksten bliver formateret, må man selv specificere formatet med SimpleDateFormat:

```
import java.text.*;
import java.util.*;
public class BenytSimpleDateFormat
{
  public static void main(String arg[])
  {
    DateFormat df = new SimpleDateFormat("EEEE 'den' d. MMMM 'år' yyyy.");

    Date tid = new Date();
    System.out.println( df.format(tid) );
  }
}
```

```
mandag den 3. december år 2001.
```

Dermed bliver selve formaterings−strengen sprogspecifik (og den bør lægges ud i en resursefil − se senere). Køres den med amerikanske sprogindstillinger, bliver uddata:

```
Monday den 3. December år 2001.
```

7.1.2 Formatering af tal og beløb

På samme måde som med tidsangivelser formateres/fortolkes tal ved at bede om et formateringsobjekt, der klarer netop denne form for tal:

```
import java.text.*;
public class BenytNumberFormat {
  public static void main(String arg[]) {
    NumberFormat fmt1 = NumberFormat.getInstance();
    NumberFormat fmt2 = NumberFormat.getCurrencyInstance();
    NumberFormat fmt3 = NumberFormat.getPercentInstance();
    double tal = 1234.5678;
    System.out.println( fmt1.format(tal) );
    System.out.println( fmt2.format(tal) );
    System.out.println( fmt3.format(tal) );
  }
}
```

```
1.234,568
kr 1.234,57
123.457%
```

Med amerikanske sprogindstillinger bliver uddata:

```
1,234.568
$1,234.57
123,457%
```

7.1.3 Tekstindhold i resursefiler

Programmet BenytDateFormat er halvt internationaliseret, da tidsformateringen korrekt skifter afhængigt af sproget. Der mangler kun strengene "Kl: ", "Dato: " og "Tid: ".

Lad os nu fuldføre internationaliseringen ved at lægge tekstindholdet (strengene) ud i et resursebundt (eng.: resource bundle) med navnet Tekster. Disse kan tilgås fra programmet således:

```
import java.text.*;
import java.util.*;
public class BenytDateFormatMedResurser
{
  public static void main(String arg[])
  {
    ResourceBundle res = ResourceBundle.getBundle("Tekster");
    DateFormat klformat, datoformat, dkf;
    klformat   = DateFormat.getTimeInstance(DateFormat.MEDIUM);
    datoformat = DateFormat.getDateInstance(DateFormat.FULL);
    dkf = DateFormat.getDateTimeInstance(DateFormat.MEDIUM,DateFormat.SHORT);

    Date tid = new Date();
    System.out.println( res.getString("Kl_")+ klformat.format(tid) );
    System.out.println( res.getString("Dato_")+ datoformat.format(tid) );
    System.out.println( res.getString("Tid_")+ dkf.format(tid) );
  }
}
```

```
Kl  : 14:23:50
Dato: 5. februar 2003
Tid : 05−02−2003 14:23
```

Resursebundtet, der skal ligge i filen Tekster.properties, indeholder teksten i nøgle−værdi−par. Disse vil blive brugt, hvis der ikke findes resursefiler for det pågældende sprog:

```
Tid_=Tid  \:
Kl_=Kl   \:
Dato_=Dato \:
```

Køres programmet (på et styresystem indstillet til dansk), fås samme udskrift som før.

Lad os nu lokalisere programmet til engelsk. Det består i, at vi opretter Tekster**_en**.properties med de engelske tekster:

```
Tid_=Time       \:
Kl_=Time of day\:
Dato_=Date       \:
```

Starter vi derefter programmet med engelske sprogindstillinger, får vi uddata:

```
Time of day : 3:07:28 PM
Date        : Monday, December 3, 2001
Time        : Dec 3, 2001 3:07 PM
```

Hvordan der søges efter resurser

Når programmet skal finde en programtekst ud fra sprogindstillingerne, sker det ved først at kigge i den mest specifikke resursefil. Hvis programteksten ikke findes der, kigges i en mere generel resursefil og til sidst i den mest generelle.

For eksempel vil resurser til sproget fr_CA (canadisk fransk) blive søgt

1. først i Tekster_fr_CA.properties
2. dernæst i Tekster_fr.properties
3. og sidst i Tekster.properties

7.1.4 Avanceret: Binære resursefiler

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

7.1.5 Avanceret tekstformatering

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

7.2 Selv bestemme sproget

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

7.2.1 Styre sproget fra kommandolinjen

Ved opstart af et program opretter Java et Locale−objekt, der svarer til sprogindstillingerne på maskinen.

På et UNIX−system gøres det ud fra miljøvariablen LC_CTYPE eller LC_ALL. Den amerikanske udskrift til BenytDateFormat er f.eks. fundet med UNIX−kommandoen:

```
  LC_CTYPE=en java BenytDateFormat
```

7.2.2 De mulige Locale−objekter

Man kan undersøge, hvilke Locale−objekter der er tilgængelige, ved at kalde klassemetoden Locale.getAvailableLocales():

```
import java.text.*;
import java.util.*;
public class MuligeSprog
{
  public static void main(String arg[])
  {
    Locale[] l = Locale.getAvailableLocales();
    for (int i=0; i<l.length; i++) System.out.print(l[i]+" ");
    System.out.println( );
  }
}
```

```
ar ar_AE ar_BH ar_DZ ar_EG ar_IQ ar_JO ar_KW ar_LB ar_LY ar_MA ar_OM ar_QA ar_SA
ar_SD ar_SY ar_TN ar_YE be be_BY bg bg_BG ca ca_ES cs cs_CZ da da_DK de de_AT
de_CH de_DE de_LU el el_GR en_AU en_CA en_GB en_IE en_IN en_NZ en_ZA es es_BO
es_AR es_CL es_CO es_CR es_DO es_EC es_ES es_GT es_HN es_MX es_NI es_PA es_PE
es_PR es_PY es_SV es_UY es_VE et et_EE fi fi_FI fr fr_BE fr_CA fr_CH fr_FR fr_LU
hr hi_IN hr_HR hu hu_HU is is_IS it it_CH it_IT iw iw_IL ja ja_JP ko ko_KR lt
lt_LT lv lv_LV mk mk_MK nl nl_BE nl_NL no no_NO no_NO_NY pl pl_PL pt pt_BR pt_PT
ro ro_RO ru ru_RU sh sh_YU sk sk_SK sl sl_SI sq sq_AL sr sr_YU sv sv_SE th th_TH
th_TH_TH tr tr_TR uk uk_UA zh zh_CN zh_HK zh_TW en en_US
```

Localet består af tre dele:

- Første del er sprogkoden, f.eks. da, sv, no, en, fr.
- En valgfri anden del er landekoden, f.eks. DK, GB, DE, FR
- En valgfri tredje del er varianten inden for sprogområdet (f.eks. om valutaen er i euro).

Eksempler:

fr_BE: Fransk i Belgien

fr_BE_EURO: Fransk i Belgien med euro−valuta

fr_CA: Fransk i Canada

fr_FR: Fransk i Frankrig

fr_LU: Fransk i Luxembourg

7.2.3 Bruge Locale−objekter

Ønsker man finkornet kontrol over, hvilket sprog der anvendes, kan man oprette et Locale−objekt selv, og dette objekt kan så bruges til at fremskaffe formateringsobjekter til det pågældende sprog:

```
import java.text.*;
import java.util.*;
public class BenytDateFormat2
{
  public static void main(String arg[])
  {
    DateFormat klformat, datoformat;

    Locale fransk = new Locale("fr","FR");
    klformat   = DateFormat.getTimeInstance(DateFormat.SHORT,fransk);
    datoformat = DateFormat.getDateInstance(DateFormat.LONG, fransk);

    Date tid = new Date();
    System.out.println( "Kl:   "+ klformat.format(tid) );
    System.out.println( "Dato: "+ datoformat.format(tid) );
  }
}
```

```
Kl:   14:23
Dato: 5 février 2003
```

7.2.4 Avanceret: Sætte standardsprog

8 Databaser (JDBC)

Dette kapitel forudsætter lidt kendskab til databaser og databasesproget SQL (Structured Query Language), og at man har en fungerende database, som man ønsker adgang til fra Java.

8.1 Basisfunktioner i JDBC

Et program, der bruger JDBC, skal basalt set udføre følgende handlinger: Indlæse driveren, etablere forbindelsen og dernæst kommunikere med databasen.

8.1.1 Indlæse driveren

Det gøres ved at indlæse en driver−klasse:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Denne klasse sørger for at registrere sig hos java.sql.DriverManager.

Med Java under Windows følger en standard JDBC−ODBC−bro med, så man kan kontakte alle datakilder, der er defineret under ODBC.

Er det en anden database, skal man have en jar−fil (et Java−ARkiv, en samling klasser pakket i zip−formatet) med en driver fra producenten, og man skal indlæse en anden driver−klasse fra jar−filen (der skal ligge i CLASSPATH).

De nyeste drivere kan findes på http://java.sun.com/jdbc, men følger ofte med når man anskaffer sig en database.

For at få indlæst en driver til en Oracle−database skriver man:

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

Driver−filen, der følger med Oracles produkter, hedder typisk classes12.zip.

For at få indlæst en driver til en MySQL−database skriver man:

```
Class.forName("com.mysql.jdbc.Driver")
```

Driver−filen til MySQL (kaldet Connector/J) kan hentes på http://mysql.com.

8.1.2 Etablere forbindelsen

Herefter kan man oprette forbindelsen med (for en ODBC−driver):

```
  Connection forb = DriverManager.getConnection("jdbc:odbc:datakilde1");
```

Parameteren er en URL til databasen, som driver−klassen genkender. Husk, at datakildens navn (her "datakilde1") først skal være defineret i Windows' Kontrolpanel under ODBC.

Er det en Oracle−database, kunne man skrive:

```
  Connection forb = DriverManager.getConnection(
              "jdbc:oracle:thin:@ora.javabog.dk:1521:student","jacob","jacob");
```

Hvorefter man skulle have oprettet forbindelsen til databasen 'student' på maskinen 'ora.javabog.dk' port 1521 med brugernavn 'jacob' og adgangskode 'jacob'.

Er det en MySQL−database, kunne man skrive:

```
  Connection c = DriverManager.getConnection("jdbc:mysql:///jacob","root","xyz");
```

Hvorefter man skulle have oprettet forbindelsen til databasen 'jacob' på den lokale maskine med brugernavn 'root' og adgangskode 'xyz'.

8.1.3 Kommunikere med databasen

Når man har oprettet en forbindelse, kan man oprette et Statement−objekt, som man kan sende kommandoer og forespørgsler til databasen med (svarer til en SQL−kommandolinje):

```
  Statement stmt = forb.createStatement();
```

Her opretter vi f.eks. tabellen KUNDER og indsætter et par rækker:

```
import java.sql.*;
public class Databasekommunikation
{
  public static void main(String[] arg) throws Exception
  {
    Class.forName("oracle.jdbc.driver.OracleDriver");
    System.out.println("Driver indlæst");

    Connection forb = DriverManager.getConnection(
              "jdbc:oracle:thin:@ora.javabog.dk:1521:student","jacob","jacob");
    System.out.println("Forbindelse oprettet");

    Statement stmt = forb.createStatement();

    // forsøg at slette tabel − hvis den ikke findes opstår en fejl som fanges
    try { stmt.executeUpdate("drop table KUNDER"); } catch (Exception e) {}

    // opret tabel
    stmt.executeUpdate("create table KUNDER (NAVN varchar(32), KREDIT number)");
    System.out.println("Tabel oprettet");

    // tilføj data
    stmt.executeUpdate("insert into KUNDER values('Jacob', −1799)");

    // brug helst navngivne kolonner. Hvis man senere skulle finde på at
    // udvide tabellen med flere kolonner, vil SQL−kommandoen stadig virke!
    stmt.executeUpdate("insert into KUNDER(NAVN,KREDIT) values('Brian', 0)");

    // indsæt data fra variabler
    String navn = "Hans";
    double kredit = 500;
    stmt.executeUpdate(
              "insert into KUNDER(NAVN,KREDIT) values('"+navn+"', "+kredit+")");

    // forespørgsler
    ResultSet rs = stmt.executeQuery("select NAVN, KREDIT from KUNDER");
    while (rs.next())
    {
      navn = rs.getString("NAVN");
      kredit = rs.getDouble("KREDIT");
      System.out.println(navn+" "+kredit);
    }
  }
}
```

```
Driver indlæst
Forbindelse oprettet
Tabel oprettet
Jacob −1799.0
Brian 0.0
Hans 500.0
```

Se også afsnit 12.8 for en beskrivelse af alle klasserne, der er til rådighed til kommunikation med en database.

8.2 Forpligtigende eller ej? (commit)

Normalt er alle SQL−kommandoer gennem JDBC *automatisk forpligtigende* (eng. auto−committing), dvs. de kan ikke annulleres, hvis en senere SQL−kommando går galt.

Vil man slå automatisk forpligtigelse fra, kan det gøres ved at kalde setAutoCommit() på forbindelsen. Derefter vil transaktioner (SQL−kommandoer) ikke være forpligtigende, og de kan annulleres ved at kalde rollback() på forbindelsen. Når man ønsker, at transaktionerne skal træde i kraft, kalder man commit(), og først herefter er transaktionerne endeligt udført på databasen og dermed forpligtigende.

Oftest anvendes denne facilitet i forbindelse med flere transaktioner, der enten alle skal gennemføres eller alle annulleres, f.eks.:

```
import java.sql.*;
public class UdenAutocommit
{
  public static void main(String[] arg) throws Exception
  {
    Class.forName("oracle.jdbc.driver.OracleDriver");
    Connection forb = DriverManager.getConnection(
               "jdbc:oracle:thin:@ora.javabog.dk:1521:student","jacob","jacob");

    try {
      forb.setAutoCommit(false);
      Statement stmt = forb.createStatement();

      stmt.executeUpdate("insert into KUNDER(NAVN,KREDIT) values('Jacob',−17)");
      stmt.executeUpdate("insert into KUNDER(NAVN,KREDIT) values('Brian', 0)");

      // flere transaktioner ...
      System.err.println("Alt gik godt, gør ændringerne forpligtigende");
      forb.commit();
    }
    catch (Exception e)
    {
      e.printStackTrace();
      System.err.println("Noget gik galt! Annullerer ændringerne...");
      forb.rollback();
    }
    finally
    {
      // Husk at sætte auto−commit tilbage, af hensyn til andre transaktioner
      forb.setAutoCommit(true);
    }
  }
}
```

```
Alt gik godt, gør ændringerne forpligtigende
```

8.3 Optimering

Ønsker man hurtigere kommunikation med databasen, er der en række ting, man kan gøre.

8.3.1 Bruge den rigtige databasedriver

JDBC−drivere findes i fire typer:

- Type 1: JDBC−ODBC−broen. Dette er den langsomste, da den kører gennem ODBC, og den er dermed platformsspecifik (hovedsageligt til Windows).
- Type 2: Drivere, hvor JDBC−laget kalder funktioner skrevet til i f.eks. maskinkode, C eller C++ til den specifikke platform (normalt den hurtigste).
- Type 3: Platformsuafhængig (ren Java−) driver, der benytter en databaseuafhængig kommunikationsprotokol til at kommunikere med en server, som så kommunikerer videre med den specifikke database. Denne type driver giver stor fleksibilitet.
- Type 4: Platformsuafhængig (ren Java−) driver, der er skrevet til at kommunikere via en kommunikationsprotokol med en specifik database.

De hurtigste drivere er type 2, men ofte kan en type 4 driver blive næsten lige så hurtig. En liste med over hundrede tilgængelige drivere kan findes på http://java.sun.com/jdbc.

8.3.2 Lægge opdateringer i kø (batch−opdateringer)

Skal man lave mange opdateringer, kan det være en fordel at sende dem af sted som én samlet enhed (eng.: batch) i stedet for som normalt at vente på, at hver opdatering skal gennemføres, før den næste kan sendes.

```
import java.sql.*;
public class Batchopdateringer
{
  public static void main(String[] arg) throws Exception
  {
    Class.forName("oracle.jdbc.driver.OracleDriver");
    Connection forb = DriverManager.getConnection(
               "jdbc:oracle:thin:@ora.javabog.dk:1521:student","jacob","jacob");
```

```
    Statement stmt = forb.createStatement();
    stmt.addBatch("insert into KUNDER(NAVN,KREDIT) values('Jacob', -1799)");
    stmt.addBatch("insert into KUNDER(NAVN,KREDIT) values('Brian', 0)");

    // send først nu ændringer til databasen
    stmt.executeBatch();
  }
}
```

8.3.3 På forhånd forberedt SQL

Det er klart, at der i koden vist ovenfor bruges noget tid på at fortolke SQL−kommandoen hver gang kaldet foretages.

I stedet for at oprette en SQL−kommandolinje med createStatement(), kan man bruge metoden prepareStatement(), hvor man angiver SQL−kommandoen én gang under opstart af programmet, og derefter kan udføre kommandoen flere gange.

```
import java.sql.*;
public class ForberedtSQL
{
  public static void main(String[] arg) throws Exception
  {
    Class.forName("oracle.jdbc.driver.OracleDriver");
    Connection forb = DriverManager.getConnection(
               "jdbc:oracle:thin:@ora.javabog.dk:1521:student","jacob","jacob");

    // Forbered kommandoerne til databasen i starten af programmet:

    PreparedStatement indsæt = forb.prepareStatement(
                                "insert into KUNDER(NAVN,KREDIT) values(?, ?)");

    PreparedStatement hent = forb.prepareStatement(
                                           "select NAVN, KREDIT from KUNDER");

    // under programudførelsen kan forberedte kommandoer udføres mange gange:
    indsæt.setString(1, "Jacob");
    indsæt.setInt(2, -1799);
    indsæt.execute();

    indsæt.setString(1, "Brian");
    indsæt.setInt(2, 0);
    indsæt.execute();

    // som før kan opdateringerne også lægges i kø:
    indsæt.setString(1, "Hans");
    indsæt.setInt(2, 142);
    indsæt.addBatch();

    indsæt.setString(1, "Grethe");
    indsæt.setInt(2, 242);
    indsæt.addBatch();

    // send ændringer til databasen
    indsæt.executeBatch();

    ResultSet rs = hent.executeQuery();
    // man løber igennem svaret som man plejer
    while (rs.next())
    {
      String navn = rs.getString(1);
      double kredit = rs.getDouble(2);
      System.out.println(navn+" "+kredit);
    }
  }
}
```

```
Jacob -1799.0
Brian 0.0
Hans 142.0
Grethe 242.0
```

8.3.4 Kalde gemte procedurer i databasen

Større databaser understøtter 'stored procedures' − procedurer gemt i databasen. Disse procedurer kan udføres hurtigere, da databasen på forhånd kan optimere, hvordan SQL−kaldene skal foregå.

En gemt procedure kan kaldes med et CallableStatement (her forestiller vi os, at der på forhånd er oprettet procedurerne **indsaetkunde** og **hentkunder** i databasen):

```
  CallableStatement indsætP =  forb.prepareCall("call indsaetkunde(?, ?)");
  CallableStatement hentP =  forb.prepareStatement("?= hentkunder");
```

Resten af arbejdet foregår som med PreparedStatement:

```
  indsætP.setString(1, "Jacob");
  indsætP.setInt(2, -1799);
  indsætP.execute();
```

```
indsætP.setString(1, "Brian");
indsætP.setInt(2, 0);
indsætP.execute();

ResultSet rs = hentP.executeQuery();
...
```

8.4 Avanceret

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

8.4.1 Opdatering og navigering i ResultSet−objekter

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

8.4.2 Metadata

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

8.4.3 Metadata om databasen

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

8.4.4 Metadata om svaret på en forespørgsel

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

8.4.5 Eksempel − udskrive en vilkårlig tabel

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

8.4.6 Persistering af objekter − JDO

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

9 Optimering

9.1 Optimering

Et programs ydeevne begrænses næsten altid af højst to−tre faktorer (flaskehalse).

Inden du læser videre, så bemærk lige en generel regel omkring optimering af ydelse:

> Under programmeringen er det i de fleste tilfælde spild af tid at tænke på optimering − det er bedre at koncentrere sig om funktionaliteten og, når programmet er skrevet næsten helt færdigt, identificere flaskehalsene og optimere disse dele af koden

Optimering af kørselstiden bør dog ske tidligere, hvis:

- Det forventes at forårsage større strukturelle ændringer i programmet. Optimeringer der forventes at indvirke på programmets struktur, bør ske i design−fasen, inden programmeringen påbegyndes.
- Det forventes at give kortere udviklingstid (fordi de løbende afprøvninger af programmet kan udføres hurtigere).

9.2 Ting, man skal undgå (hvis man kan)

Man skal så vidt muligt undgå at:

9.2.1 Oprette mange objekter

Det tager både tid at oprette dem, og det tager også tid for den virtuelle maskine at finde og frigive objekter, der ikke mere er i brug. Genbrug, objekterne hvis det er muligt.

Eksempel: Streng−manipulation

Skal du sætte mange strenge sammen, bør du bruge en StringBuffer, da du så undgår at oprette de mange midlertidige String−objekter, der ellers ville opstå.

Følgende program demonstrerer den enorme hastighedsforskel, en StringBuffer kan gøre.

```
// Demonstrerer hastighedsforskellen mellem String og StringBuffer
// ved sammensætning af mange strenge
public class HastighedsforskelMellemStringOgStringBuffer
{
  public static void main (String[] arg)
  {
    long tid1 = System.currentTimeMillis();

    String s = "";
    for (int i=0; i<10000; i++) s = s + "x";   // her oprettes 10000 objekter

    long tid2 = System.currentTimeMillis();
    System.out.println("Antal sekunder med String: "+ (tid2−tid1)*0.001 );

    StringBuffer sb = new StringBuffer(10000); // reservér plads til 10000 tegn
    for (int i=0; i<10000; i++) sb.append("x");// her ændres i det samme objekt
    String s2 = sb.toString();

    long tid3 = System.currentTimeMillis();
    System.out.println("Antal sek med StringBuffer: "+ (tid3−tid2)*0.001 );
  }
}
```

```
Antal sekunder med String: 3.432
Antal sek med StringBuffer: 0.021
```

Her sparer man altså over faktor 150 i kørselstid (0.02 i stedet for 3.4 sekunder).

Besparelsen skyldes, at der oprettes færre objekter (1 StringBuffer i stedet for 10000 strenge), og oprettelse (og oprydning af) objekter er en forholdsvis tidskrævende operation.

9.2.2 Oprette mange tråde

Tråde tager beslag på en del systemresurser, så opret ikke alt for mange, og genbrug dem, hvis det er muligt (se afsnit 16.7.3, Genbrug af tråde).

9.2.3 Kaste og fange mange undtagelser

Undtagelser er netop undtagelser, og den virtuelle maskine udfører programmet hurtigst, hvis de ikke opstår.

9.2.4 Gå ind og ud af synkroniserede blokke/metoder

Hver gang en tråd går ind i en blok/metode mærket med synchronized, skal den virtuelle maskine tjekke, om der allerede er en tråd aktiv i denne blok. Den får altså noget ekstra arbejde, som tager tid.

9.2.5 Bruge mange (evt. anonyme) klasser

Ved opstart indlæses alle klasserne, som programmet består af eller har brug for, og det tager selvfølgelig tid.

Overraskende nok (for nogen) ligger indre klasser og anonyme klasser *ikke* inde i den ydre klasse på filsystemet, men skal indlæses separat.

Bruger man derfor mange indre klasser eller anonyme klasser, f.eks. i forbindelse med programmering af grafiske brugergrænseflader, bør man overveje, om man kunne slå nogen af dem sammen.

Hvis det drejer sig om hændelses−lyttere, kunne man overveje at lade den ydre klasse lytte efter alle de forskellige hændelser og droppe alle de anonyme klasser som et GUI−udviklingsværktøj typisk laver.

Eksempel

I afsnit 4.1.3 Bruge en javabønne er der et eksempel på hvordan. Først er der et eksempel med en anonym ActionListener−klasse (der er altså 2 klasser, der skal indlæses):

```
import java.awt.*;

public class BenytBoenneMedVaerktoej extends Frame
{
  ...

  public BenytBoenneMedVaerktoej()
  {
    // anonym indre klasse lytter på hændelser og kalder derpå videre til
    // metoden textFieldNavn_actionPerformed()
    textFieldNavn.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        textFieldNavn_actionPerformed(e);
      }
    });
    ...
  }

  void textFieldNavn_actionPerformed(ActionEvent e) {
    String navn = textFieldNavn.getText();
    System.out.println("Navnet er: "+navn);
  }
}
```

Efterfølgende blev den anonyme klasse sparet væk ved at lade vinduet selv implementere ActionListener (der er altså kun 1 klasse, der skal indlæses):

```
import java.awt.*;

public class BenytBoenneSkrevetSelv extends Frame implements ActionListener
{
  ...

  public BenytBoenneSkrevetSelv() {
    // klassen selv lytter på hændelser
    textFieldNavn.addActionListener(this);
    ...
  }

  void actionPerformed(ActionEvent e) {
    String navn = textFieldNavn.getText();
```

```
    System.out.println("Navnet er: "+navn);
  }
}
```

9.3 Er C++ hurtigere end Java?

Det er en udbredt opfattelse, at C++ er hurtigere end Java, og at C++ derfor er mere velegnet til programmer, der skal foretage mange beregninger.

Det er imidlertid en myte, der slet ikke gælder for de nyere udgave af den virtuelle maskine (JDK 1.2 og derefter).

> Et javaprogram laver beregninger nogenlunde lige så hurtigt (eller hurtigere) som et fuldt optimeret C++−program

Myten stammer fra 1990−erne, før de virtuelle maskiner fik implementeret JIT (Just In Time)−oversættelse. JIT virker ved at oversætte bytekoden til maskinkode, efterhånden som den skal udføres, sådan at der bruges noget tid på JIT−oversættelse første gang, noget bytekode udføres, derefter går det lige så hurtigt som maskinkode.

Almindelige C/C++−programmer (og Fortran− og Pascal−programmer for den sags skyld) bliver oversat til maskinkode én gang for alle, hvorefter de kan køres igen og igen.

Javaprogrammer bliver oversat til maskinkode af den virtuelle maskine under kørslen, hvilket naturligvis tager mere tid første gang, men i modsætning til C/C++−programmer kan JIT−oversætteren vælge at lave maskinkoden afhængig af, hvordan kørslen 'normalt' foregår.

9.3.1 Et testprogram i Java

Dette lille program finder alle primtal mellem 50000 og 100000. Der bruges ingen objekter:

```
public class Primtal
{
  public static void main(String[] args) {
    int antalPrimtal = 0;

    int tal;
    int faktor;

    for (tal = 50000; tal<100000; tal++)
    {
      faktor = 2;

      while (tal % faktor > 0) faktor++;

      if (faktor == tal)
      {
        System.out.print(tal + " er et primtal.\n");
        antalPrimtal = antalPrimtal + 1;
      }
    }
    System.out.println("Antal primtal i alt: " + antalPrimtal);
  }
}
```

Under Linux kan man undersøge, hvor lang tid en kommando tager med 'time'. For at tage tiden på ovenstående program skriver man:

```
time java Primtal
```

... hvorefter programmet kører:

```
50021 er et primtal.
50023 er et primtal.
...
99961 er et primtal.
99971 er et primtal.
99989 er et primtal.
99991 er et primtal.
Antal primtal i alt: 4459
32.16user 0.19system 0:36.13elapsed 89%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (5674major+2738minor)pagefaults 0swaps
```

Sidst udskrives kørselstiden. Dette program tager 32.16 sekunders CPU (plus 0.19 sekunders CPU−tid til systemkald).

9.3.2 Testprogrammet i C++

Lad os nu skrive det tilsvarende program i C++:

```
#include <iostream.h>

int main() {
  int antalPrimtal = 0;

  int tal;
```

```
  int faktor;

  for (tal = 50000; tal<100000; tal++)
  {
    faktor = 2;

    while (tal % faktor > 0) faktor++;

    if (faktor == tal)
    {
      cout<<tal<<" er et primtal.\n";
      antalPrimtal = antalPrimtal + 1;
    }
  }

  cout<<"Antal primtal i alt: "<<antalPrimtal<<"\n";
}
```

Lad sige, at kildeteksten ligger i filen prim.c++. Så oversætter man og kører programmet med:

```
g++ -O3 prim.c++
time a.out
```

Parameteren −O3 fortæller oversætteren, at den skal forsøge at optimere programmet så meget som overhovedet muligt (mildere indstillinger er −O og −O2). Oversætteren generer filen a.out med det eksekverbare program, som udføres og times samtidig:

```
time a.out
```

Uddata fra programmet er:

```
...
99929 er et primtal.
99961 er et primtal.
99971 er et primtal.
99989 er et primtal.
99991 er et primtal.
Antal primtal i alt: 4459
32.71user 0.08system 0:35.96elapsed 91%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (200major+27minor)pagefaults 0swaps
```

Selv med den mest ekstreme optimering af C++−programmet, klarer det sig altså lidt dårligere end Javaprogrammet (dog ikke ret meget − men uden optimering tager C++−programmet 37 sekunder at køre).

Forklaringen på, at Java i dette tilfælde er hurtigere end C++, skal nok findes i JIT'ens behandling af while−løkken og if−sætningen:

```
      while (tal % faktor > 0) faktor++;
      if (faktor == tal)
```

Eftersom C++−oversætteren under oversættelsen og optimeringen ikke kan vide, om betingelserne som regel er opfyldt eller ej, må den prøve at gætte − og den gætter sandsynligvis ofte forkert.

JIT'en kan derimod observere kørslen nogle gange og konstatere, at while−betingelsen oftest er opfyldt, mens if−sætningen meget sjældent er det, og generere maskinkode, der er optimeret derefter.

9.4 Videre læsning

- På hjemmesiden http://www.javaperformancetuning.com/ af Jack Shirazi findes tusindvis af tip til, hvordan man kan optimere sit program.
- Samme forfatter har udgivet bogen "Java Performance Tuning": på http://www.oreilly.com/catalog/javapt/.
- I næste kapitel er beskrevet, hvordan man kalder et eksternt program eller lænker noget maskinkode fra et andet programmeringssprog ind i ens javaprogram.

10 Eksterne kald og JNI

Undertiden får man brug for at kalde noget kode, der ikke er skrevet i Java. Man kan stå i to situationer:

- Man ønsker at aktivere en kommando eller et eksternt program fra Java.
- Man ønsker af den ene eller anden grund at skrive noget af sit program i et andet programmeringssprog end Java, f.eks. for at bruge et programbibliotek, der ikke findes til Java.

Det første er enklere end det sidste, hvorfor man bør overveje, om ens problemstilling kan løses ved at lave et eksternt program og kalde det.

I begge tilfælde skal man være opmærksom på, at man mister platformsuafhængigheden, dvs. låser brugeren fast på en bestemt platform, hvis man kalder et program eller et programbibliotek, som kun findes på en enkelt platform (hvis man f.eks. vil bruge Windows' win32−API−kald, har man låst brugeren af programmet fast på Windows−platformen).

10.1 Kalde eksterne programmer

Hvis man vil aktivere et eksternt program fra Java, kan man gøre dette med metoden exec(), der findes i Runtime−klassen. I parameteren til exec() kan man skrive kommandoen, ligesom hvis programmet var aktiveret fra kommandolinjen.

Her er et program, der kalder det eksterne program 'sort' (der er valgt, fordi det findes under både UNIX/Linux og DOS/Windows):

```
import java.io.*;

public class KaldEksterntProgram
{
  public static void main(String[] args) throws Exception
  {
    // Start programmet 'sort', der sorterer linerne i en fil
    Process proces = Runtime.getRuntime().exec("sort");

    // sort kan læse data fra standard input og skriver dem på standard output
    PrintWriter s = new PrintWriter(proces.getOutputStream());
    BufferedReader l = new BufferedReader(
                       new InputStreamReader(proces.getInputStream()));
    s.println("En");
    s.println("snegl");
    s.println("på");
    s.println("vejen");
    s.println("er");
    s.println("tegn");
    s.println("på");
    s.println("regn");
    s.println("i");
    s.println("Spanien");
    s.close(); // luk datastrømmen (sort sorterer først når den har alle data)

    // Læs resultatet fra programmets standard output og udskriv det
    String lin;
    while ((lin = l.readLine()) != null) System.out.println("Fra sort: "+lin);
  }
}
```

```
Fra sort: En
Fra sort: er
Fra sort: i
Fra sort: på
Fra sort: på
Fra sort: regn
Fra sort: snegl
```

```
Fra sort: Spanien
Fra sort: tegn
Fra sort: vejen
```

10.1.1 Kommunikation med det eksterne program

Programmet 'sort' læser data fra standard input (normalt fra tastaturet eller omdirigeret fra en fil) og skriver dem på standard output (normalt til skærmen eller omdirigeret til en fil).

I eksemplet får vi fat i processens standard input med getOutputStream() (det, vi skriver til programmet, bliver jo programmets input) og standard output med getInputStream().

Man skal være opmærksom på, at hvis ens javaprogram forsøger at læse noget fra et eksternt programs standard output, inden det har skrevet noget, vil læsekaldet 'hænge', indtil der kommer data fra det eksterne program (det kan undertiden forvirre den mindre erfarne programmør).

Vil man blot vente på, at det eksterne program er kørt færdigt, kan man bruge kaldet:

```
    proces.waitFor();                       // vent på at processen er færdig
```

Det er der et eksempel på i afsnit 11.4.1, Kalde oversætteren som eksternt program. Samme sted kan man se, hvordan man kan overføre parametre til det eksterne program ved simpelt hen at skrive dem efter programnavnet:

```
    Process p = r.exec("javac UndersoegKlasse.java");
```

I dette tilfælde får javac parameteren 'UndersoegKlasse.java'.

10.2 Kald til andre programmeringssprog

Hvis man af den ene eller anden grund ønsker at skrive noget af sit program i et andet programmeringssprog end Java, f.eks. for at bruge et programbibliotek, der ikke findes til Java, skal man bruge JNI (Java Native Interface).

10.2.1 JNI − Java Native Interface

JNI giver mulighed for at:

- Kalde andre sprog fra Java, dvs. at kalde maskinkode fra Java. Denne maskinkode kan være genereret ud fra kildetekst skrevet i andre programmeringssprog, f.eks. C, C++ eller Pascal.
- Kalde Java fra andre sprog, dvs. starte Javas virtuelle maskine op, indlæse klasser og kalde metoder i dem fra kildetekst skrevet i andre programmeringssprog.

I det følgende vil kun den første mulighed (kalde andre sprog fra Java) blive behandlet overfladisk. Ønsker man at vide mere om JNI, findes der nogle henvisninger i afsnit 10.2.7.

10.2.2 Skrive Java−klasser med maskinkode

Man kan implementere metoder i en klasse i f.eks. C, C++ eller direkte i maskinkode ved at markere dem med nøgleordet **native** i javakoden:

```
package vp;
public class HejVerdenFraCKode
{
  public native void hejVerden();
}
```

Her er et program, der benytter klassen HejVerdenFraCKode efter at have indlæst den nødvendige maskinkode:

```
package vp;
public class BenytHejVerdenFraCKode
{
  public static void main(String[] args)
  {
    System.out.println("Indlæser maskinkoden.");
    // Indlæs maskinekodebiblioteket libHej.so (UNUX) eller hej.dll (Windows)
    try {
      System.loadLibrary("Hej");
    } catch (Error e) {
      System.err.println("Fejl ved indlæsning af maskinkodeblok.");
      e.printStackTrace();
      System.exit(1);      // afslut programmet
    }

    System.out.println("Opretter objekt.");
    HejVerdenFraCKode objekt = new HejVerdenFraCKode();

    System.out.println("Kalder metode implementeret i maskinkode.");
    objekt.hejVerden();
  }
}
```

```
Indlæser maskinkoden.
```

```
Opretter objekt.
Kalder metode implementeret i maskinkode.
Hej Verden fra C++ !
```

10.2.3 Skrive en metode i C++

Efter at have skrevet klassen skal den oversættes til bytekode, dvs. fra .java−fil til .class−fil. Det kan gøres i et udviklingsværktøj eller fra kommandolinjen med kommandoen:

```
  javac vp/HejVerdenFraCKode.java
```

Husk, at da klassen ligger i pakken 'vp', skal den også ligge i underkataloget 'vp'.

Nu har vi filen vp/HejVerdenFraCKode.class. Så skal der genereres en C−headerfil med:

```
  javah −jni vp.HejVerdenFraCKode
```

Denne headerfil (der kommer til at hedde vp_HejVerdenFraCKode.h) indeholder kun navnet på metoden hejVerden(), der i C−koden bliver omdøbt til at indeholde pakkenavnet og klassenavnet (Java_vp_HejVerdenFraCKode_hejVerden).

I den genererede headerfil står:

```
/* DO NOT EDIT THIS FILE − it is machine generated */
#include <jni.h>
/* Header for class vp_HejVerdenFraCKode */

#ifndef _Included_vp_HejVerdenFraCKode
#define _Included_vp_HejVerdenFraCKode
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     vp_HejVerdenFraCKode
 * Method:    hejVerden
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_vp_HejVerdenFraCKode_hejVerden (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Denne metode skal vi selv implementere i f.eks. C++. Vi opretter derfor en fil kaldet vp_HejVerdenFraCKode.cpp og skriver C++−koden, der implementerer metoden:

```
#include "vp_HejVerdenFraCKode.h"
#include <stdio.h>

JNIEXPORT void JNICALL Java_vp_HejVerdenFraCKode_hejVerden (JNIEnv *e, jobject o)
{
  printf("Hej Verden fra C++ !\n");
}
```

Denne kode skal oversættes til et bibliotek med maskinkode (eng.: shared library), der kan indlæses fra Java. Præcist hvordan dette gøres afhænger af styresystemet.

10.2.4 Oversætte til maskinkode

Fra UNIX/Linux kunne det gøres ved at kalde standard C++−oversætteren:

```
 g++ −shared −g −I/usr/include/java vp_HejVerdenFraCKode.cpp −o libHej.so
```

Man skal huske, at '/usr/include/java' skal pege hen på, hvor ens Java−headerfiler findes. Det kan også være, at man er nødt til at inkludere flere kataloger med yderligere headerfiler, f.eks.:

```
 g++ −shared −g −I/usr/local/java/include  −I/usr/local/java/include/linux \
  vp_HejVerdenFraCKode.cpp −o libHej.so
```

Under UNIX/Linux genereres en .so−fil (som her hedder libHej.so). Under Windows skal der genereres en .DLL−fil (dynamic link library), der tilsvarende ville hedde Hej.DLL (præcis hvordan man gør afhænger af, hvilken C++−oversætter man bruger).

10.2.5 Køre programmet

Når kommandoen System.loadLibrary("Hej") kaldes i eksemplets main()−metode ovenfor bliver filen libHej.so eller Hej.DLL (afhængig af styresystemet) indlæst af den virtuelle maskine og lænket sammen med resten af programmet.

Vi starter programmet med:

```
 java vp.HejVerdenFraCKode
```

Hvis den genererede maskinkode ikke ligger der hvor systemet i øvrigt lægger sine maskinkodebiblioteker (dvs. i /usr/lib/ under UNIX og c:\win\system\ under Windows), får man fejlen:

```
Indlæser maskinkoden.
Fejl ved indlæsning af maskinkodeblok.
java.lang.UnsatisfiedLinkError: no Hej in java.library.path
        at java.lang.ClassLoader.loadLibrary(ClassLoader.java:1403)
        at java.lang.Runtime.loadLibrary0(Runtime.java:788)
        at java.lang.System.loadLibrary(System.java:832)
        at vp.HejVerdenFraCKode.main(HejVerdenFraCKode.java:11)
```

Da kan man fortælle den virtuelle maskine, at den skal lede efter maskinkode i det aktuelle katalog ('.') med:

```
 java −Djava.library.path=. vp.HejVerdenFraCKode
```

Nu udskriver programmet:

```
Indlæser maskinkoden.
Opretter objekt.
Kalder metode implementeret i maskinkode.
Hej Verden fra C++ !
```

Den sidste linje bliver udskrevet fra C++, så det lykkedes os at bruge JNI.

10.2.6 Kommunikation fra C−koden til Java

Ser man nærmere på erklæringen af metoden opdager man, at selvom java−metoden ikke havde nogle parametre, så har C−koden to parametre:

```
JNIEXPORT void JNICALL Java_vp_HejVerdenFraCKode_hejVerden (JNIEnv *e, jobject o)
```

Den første er en pointer til den virtuelle maskine. Den skal bruges, hvis man ønsker at kommunikere med den virtuelle maskine, f.eks. hvis man vil oprette nye Java−objekter fra C−koden.

Den anden variabel er en pointer til objektet (instansen af HejVerdenFraCKode), som metoden blev kaldt på (eller klassen, hvis metoden hejVerden havde været en klassemetode).

Disse to pointere bruges i udstrakt grad i JNI til at få adgang til metoder eller variabler i objektet eller klassen, til at oprette objekter etc. etc.

Man skal være opmærksom på, at den virtuelle maskine kan finde på at flytte rundt på objekterne i hukommelsen, når den rydder op i ubrugte objekter (garbage collection). Derfor skal man fra C−koden fortælle den virtuelle maskine, hvilke objekter man er i gang med at arbejde i, så den ikke flytter dem samtidig.

10.2.7 Henvisninger

- Suns dokumentation af JNI findes på http://java.sun.com/products/jdk/1.3/docs/guide/jni/
- Ønsker du for alvor at bruge JNI har Kristian Hansens bog 'Avanceret Java−programmering', der kan hentes gratis på http://bog.ing.dk/, en meget grundig behandling af emnet.

11 Introspektion

Introspektion (eng.: Introspection eller reflection) handler om, hvordan man under kørslen af programmet kan inspicere et *vilkårligt objekt*, finde dets klasse, finde ud af, hvilke metoder og variabler klassen har, kalde metoderne og aflæse/sætte variablerne, oprette nye objekter fra klassen etc. etc.

Introspektion bruges sjældent i almindelige programmer, men det kan være nyttigt at kende til mulighederne for introspektion for at forstå, hvordan andre programmer, bl.a. udviklingsværktøj, fungerer.

11.1 Læse klasseinformation for et objekt

Stamklassen java.lang.Object, som alle objekter direkte eller indirekte arver fra, indeholder metoden getClass() − og ethvert objekt har derfor getClass(). Denne metode returnerer en repræsentation af objektets *klasse* (af typen java.lang.Class).

I eksemplet herunder opretter vi et Frame−objekt og henter dets klasse. Vi kigger derefter på superklasserne og udskriver dem.

```
import java.lang.reflect.*;
import java.awt.*;

public class UndersoegKlasse
{
  public static void main(String[] args)
  {
    Object o = new Frame();

    // Find klassen
    Class klasse = o.getClass();
    System.out.println("Klassen navn er: "+klasse.getName());

    // Find superklasserne
    Class superklasse = klasse.getSuperclass();
    while (superklasse != null)
    {
      System.out.println("... og den har superklasse: "+superklasse.getName());
      superklasse = superklasse.getSuperclass();
    }
  }
}
```

```
Klassen navn er: java.awt.Frame
... og den har superklasse: java.awt.Container
... og den har superklasse: java.awt.Component
... og den har superklasse: java.lang.Object
```

Man ser, at Frame er en Container, der igen arver fra Component, som arver fra Object.

11.1.1 Oversigt over pakken java.lang.reflect

Class−objekter har metoder til at spørge om alt, hvad der er værd at vide om klassen, herunder variabler, konstruktører og metoder (der bruges hjælpeklasserne Field, Constructor og Method). De vigtigste er opremset herunder:

Vigtigste metoder i Class (der repræsenterer en klasse)

String getName() giver en streng med klassens navn (og pakkenavn)

Class getSuperclass() giver Class−objektet, der repræsenterer superklassen

Class[] getInterfaces() giver et array med de interfaces, klassen implementerer

Field[] getFields() giver et array med de variabler, der er erklæret public i klassen

Constructor[] getConstructors() giver et array med de konstruktører, der er erklæret public

Method[] getMethods() giver et array med de metoder, der er erklæret public i klassen

Class[] getClasses() giver et array af de indre klasser (og interfaces), der er public i klassen

Ved normal introspektion af en klasse ses kun variabler, metoder og konstruktører, der er erklæret public. Vil man også se protected, pakke og private variabler/metoder/konstruktører på klassen skal man bruge nogle tilsvarende metoder, der starter med *getDeclared*, f.eks. getDeclaredFields(), getDeclaredConstructors() og getDeclaredMethods(). Disse metoder giver alle data, men der sker først et sikkerhedstjek for, om den kaldende tråd har tilladelse til at få disse oplysninger (det har f.eks. en applet ikke).

11.1.2 Opremse metoderne i en klasse

I eksemplet herunder får vi med getMethods() et array af alle metoder i klassen, der er erklæret public. Dette gennemløbes, og for hver metode udskrives retur− og parametertyper.

```
import java.lang.reflect.*;
import java.awt.*;

public class FindMetoder
{
  public static void main(String[] args)
  {
    Object o = new Button();

    // Find klassen
    Class klasse = o.getClass();
    System.out.println("Klassen navn er: "+klasse.getName());

    Method[] metoder = klasse.getMethods();
    for (int i=0; i<metoder.length; i++)
    {
      Method m = metoder[i];
      System.out.print("Metode "+m.getName());
      System.out.print(" har returtype: "+m.getReturnType().getName());
      Class[] parametertyper = m.getParameterTypes();
      System.out.print(" og parametertyper:");
      for (int j=0; j<parametertyper.length; j++)
        System.out.print(" " + parametertyper[j].getName());
    }
    System.out.println();
  }
}
```

```
Klassen navn er: java.awt.Button
...
Metode: notify har returtype: void
Metode: notifyAll har returtype: void
Metode: toString har returtype: java.lang.String
...
Metode: getLabel har returtype: java.lang.String
Metode: setLabel har returtype: void og parametertyper: java.lang.String
...
```

Programudskriften er ret lang, idet også superklassernes metoder udskrives.

11.2 Arbejde med objekter

Ud over at inspicere klasserne er der også mulighed for at arbejde aktivt med dem:

- Class−objekter har metoden newInstance(), der opretter et objekt ved at bruge standardkonstruktøren (er der ikke adgang til standardkonstruktøren, kastes en undtagelse).
- Ønsker man at bruge en anden konstruktør, får man arrayet af andre konstruktører ved at kalde getConstructors(). Hvert Constructor−objekt har metoden newInstance() til at oprette et objekt med netop denne konstruktør.
- Hver Field (variabel) har metoderne get() og set() til at arbejde med variablens værdi.
- Hvert Method−objekt har metoden invoke() til at kalde metoden på objektet (der er et eksempel på brug herunder).

11.3 Introspektion på javabønner

Netop til javabønner findes der nogle klasser i pakken java.beans, der er specielt velegnede til at undersøge og manipulere med egenskaber (dvs. get− og set−metoder, se kapitel 4, Komponentbaseret programmering).

Man kan f.eks. opremse en bønnes egenskaber, udskrive en beskrivelse af hver egenskab og aflæse dens værdi:

```
import java.lang.reflect.*;
import javax.swing.*;
import java.beans.*;

public class Boenneintrospektion
{
```

```
  public static void main(String[] args) throws Exception
  {
    Object objekt = new JButton();
    Class klasse = objekt.getClass();
    BeanInfo bønneinfo = Introspector.getBeanInfo(klasse);
    PropertyDescriptor egenskaber[] = bønneinfo.getPropertyDescriptors();

    for (int i=0; i<egenskaber.length; i++)
    {
      PropertyDescriptor e = egenskaber[i];

      System.out.print(e.getName()+": "+e.getShortDescription());

      Method læsemetode = e.getReadMethod();
      if (læsemetode != null)
      {
        Object[] tomParameterliste = {};
        Object værdi = læsemetode.invoke(objekt,tomParameterliste);
        System.out.print(" (værdi="+værdi+")");
      }

      // sæt egenskaben til true, hvis den kan sættes og er af type boolean
      Method skrivemetode = e.getWriteMethod();
      if (skrivemetode!=null && e.getPropertyType()==java.lang.Boolean.TYPE)
      {
        Boolean[] parameterlisteMedTRUE = { Boolean.TRUE };
        skrivemetode.invoke(objekt, parameterlisteMedTRUE ); // sæt egenskab
      }
      System.out.println();
    }
  }
}
```

Herunder ses, hvad programmet skriver ud (nogle linjer er fjernet):

```
icon: The button's default icon (værdi=null)
inputMap: inputMap (værdi=javax.swing.InputMap@6e3d60)
insets: insets (værdi=java.awt.Insets[top=5,left=17,bottom=5,right=17])
alignmentY: The preferred vertical alignment of the component. (værdi=0.5)
alignmentX: The preferred horizontal alignment of the component. (værdi=0.0)
toolTipText: The text to display in a tool tip. (værdi=null)
mnemonic: the keyboard character mnemonic (værdi=0)
verticalAlignment: The vertical alignment of the icon and text. (værdi=0)
defaultButton: Whether or not this button is the default button (værdi=false)
rolloverEnabled: Whether rollover effects should be enabled. (værdi=false)
horizontalAlignment: The horizontal alignment of the icon and text. (værdi=0)
borderPainted: Whether the border should be painted. (værdi=true)
```

Metoden getShortDescription() henter en beskrivelse af egenskaben fra bønnens BeanInfo−klasse (ekstra information, der er beregnet til udviklingsværktøj, se afsnit 4.3.3). Kører du selv eksemplet, kan det være, at den ekstra information ikke er tilgængelig, og da giver getShortDescription() blot det korte navn.

11.4 Generere nye klasser og indlæse dem

Et spørgsmål relateret til introspektion er:

- Hvordan kan et program lave nye klasser og indlæse dem?
- Hvordan kan man oversætte en .java−fil (kildetekst) til en .class−fil (bytekode)?

11.4.1 Kalde oversætteren som eksternt program

En mulighed for at oversætte en .java−fil til en .class−fil er at kalde oversætteren javac som et eksternt program, ligesom det gøres fra kommandolinjen, f.eks.:

```
public class KaldJavacEksternt
{
  public static void main(String[] args) throws Exception
  {
    Runtime r = Runtime.getRuntime();
    Process p = r.exec("javac UndersoegKlasse.java");
    p.waitFor();                                // vent på at processen er færdig
    System.out.println("færdig");
  }
}
```

```
færdig
```

Man skal være opmærksom på, at kommandoen 'javac' skal være tilgængelig fra kommandolinjen. Det kan f.eks. ske ved at ændre i stien (PATH−miljøvariablen).

11.4.2 Bruge oversætteren internt

Oversætteren er faktisk programmeret i Java og findes i klassen com.sun.tools.javac.Main. En anden mulighed er derfor at bruge denne klasse direkte:

```
public class KaldJavacInternt
{
  public static void main(String[] args)
  {
    com.sun.tools.javac.Main oversætter = new com.sun.tools.javac.Main();
    String[] filer = { "UndersoegKlasse.java" };
    oversætter.compile( filer );
    System.out.println("færdig");
  }
}
```

```
færdig
```

Skal man oversætte klasser mange gange, er denne måde langt hurtigere, idet oversætteren kører i den allerede eksisterende virtuelle maskine i stedet for at blive startet som et nyt program.

Man skal være opmærksom på, at filen lib/tools.jar fra ens Java−installation skal inkluderes i klassestien (CLASSPATH), sådan at programmet skal oversættes og køres f.eks. således:

```
javac −classpath /usr/local/jdk1.4/lib/tools.jar:. KaldJavacInternt.java
java −classpath /usr/local/jdk1.4/lib/tools.jar:. KaldJavacInternt
```

11.4.3 Indlæse klasser fra filsystemet

Hvis man vil indlæse klasser fra et andet sted end der, hvor systemet plejer at lede (f.eks. over netværket), skal man definere sin egen ClassLoader.

Eksemplet herunder er en ClassLoader, der kan indlæse klasser fra et bestemt katalog. Det får stien til kataloget overført i konstruktøren, hvorefter det opbygger en liste over de tilgængelige .class−filer. Denne liste kan hentes udefra ved at kalde tilgængeligeKlasser().

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

public class ClassLoaderFraKatalog extends ClassLoader {
  private Map klasser = new HashMap();

  public Set tilgængeligeKlasser()
  {
    return klasser.keySet();
  }

  public ClassLoaderFraKatalog(String sti)
  {
    File katalog = new File(sti);
    File[] filer = katalog.listFiles();

    for (int i=0; i<filer.length; i++)
    {
      File f = filer[i];
      System.out.println(f);
      String fn = f.getName();

      if (fn.endsWith(".class"))
      {
        String klassenavn = fn.substring(0,fn.length()−6); // fjern .class
        klasser.put(klassenavn,f);
      }
    }
  }

  public Class findClass(String navn) throws ClassNotFoundException
  {
    //System.out.println("findClass("+navn);
    try {
      File f = (File) klasser.get(navn);
      if (f==null) throw new IllegalArgumentException("Ukendt klasse: "+navn);

      byte[] b = new byte[(int) f.length()];
      FileInputStream fis = new FileInputStream(f);
      fis.read(b);
      return defineClass(navn, b, 0, b.length);
    } catch (Exception e) {
      System.out.println("findClass() fejl:"+e.getMessage());
      throw new ClassNotFoundException(e.getMessage());
    }
  }
}
```

Herunder et program, der benytter ClassLoaderFraKatalog:

```
import java.util.*;

public class BenytClassLoaderFraKatalog
{
  public static void main(String[] args) throws Exception
  {
```

```
    // indlæs klasser fra det aktuelle katalog (.)
    ClassLoaderFraKatalog classLoader = new ClassLoaderFraKatalog(".");

    Set tilgængeligeKlasser = classLoader.tilgængeligeKlasser();
    System.out.println("tilgængeligeKlasser="+tilgængeligeKlasser);

    for (Iterator i=tilgængeligeKlasser.iterator(); i.hasNext(); )
    {
      try {
        String klassenavn = (String) i.next();
        Class klasse = classLoader.loadClass(klassenavn);
        Object objekt = klasse.newInstance();
        System.out.println("obj = " + objekt );
      } catch (Throwable e) {
        e.printStackTrace();
      }
    }
  }
}
```

```
./FindMetoder.java
./ClassLoaderFraKatalog.java
./Boenneintrospektion.java
./FindMetoder.class
./KaldJavacEksternt.class
./KaldJavacInternt.java
./KaldJavacEksternt.java
./UndersoegKlasse.class
./BenytClassLoaderFraKatalog.class
./UndersoegKlasse.java
./BenytClassLoaderFraKatalog.java
./ClassLoaderFraKatalog.class
./Boenneintrospektion.class
./KaldJavacInternt.class
tilgængeligeKlasser=[FindMetoder, ClassLoaderFraKatalog, UndersoegKlasse, BenytClassLoaderFraKatalog, KaldJavacE
obj = FindMetoder@53c015
java.lang.InstantiationException: ClassLoaderFraKatalog
        at java.lang.Class.newInstance0(Native Method)
        at java.lang.Class.newInstance(Class.java:232)
        at BenytClassLoaderFraKatalog.main(BenytClassLoaderFraKatalog.java:18)
obj = UndersoegKlasse@680a59
obj = BenytClassLoaderFraKatalog@7f5ea7
obj = KaldJavacEksternt@13fac
obj = Boenneintrospektion@4672d0
obj = KaldJavacInternt@4abc9
```

Det ses, at det lykkes os at indlæse klasserne og oprette objekter fra dem.

Eneste undtagelse er indlæsningen af klassen ClassLoaderFraKatalog, hvor kaldet til newInstance() fejler, da klassen ikke har en konstruktør uden parametre.

11.5 Videre læsning

For en grundigere indføring end den herover se

- Kristian Hansens bog 'Avanceret Java−programmering', der kan hentes gratis på http://bog.ing.dk/.
- Suns dokumentation http://java.sun.com/docs/books/tutorial/reflect/.

12 Standardbiblioteket (J2SE)

Dette kapitel giver en oversigt over de mest brugte dele af standardbiblioteket (kaldet J2SE − Java 2 Standard Edition). Pakkerne og klasserne er ikke, som i den normale javadokumentation, sorteret alfabetisk, men i stedet ordnet efter nytteværdi og emne.

Kapitlet findes også i netudgaven af bogen, http://javabog.dk/VP, hvor hver klasse henviser til Suns javadokumentation, så du ved at klikke på klassen kan læse mere om den.

Klasser er skrevet med almindelig skrift, mens interfaces er skrevet i kursiv. Undtagelser er ikke medtaget. Ligeledes er frarådede klasser (markeret med deprecated) og klasser og pakker kun beregnet til internt brug ikke medtaget.

12.1 Grundlæggende klasser (java.lang)

Object Class `Object` is the root of the class hierarchy.

String The `String` class represents character strings.

StringBuffer A string buffer implements a mutable sequence of characters.

System The `System` class contains several useful class fields and methods.

Runtime Every Java application has a single instance of class `Runtime` that allows the application to interface with the environment in which the application is running.

RuntimePermission This class is for runtime permissions.

Process The `Runtime.exec` methods create a native process and return an instance of a subclass of `Process` that can be used to control the process and obtain information about it.

SecurityManager The security manager allows applications to implement a security policy.

Math The class `Math` contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

StrictMath The class `StrictMath` contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

Cloneable A class implements the `Cloneable` interface to indicate to the `Object.clone()` method that it is legal for that method to make a field−for−field copy of instances of that class.

Comparable This interface imposes a total ordering on the objects of each class that implements it.

12.1.1 Klasser, der svarer til de simple typer

Boolean The Boolean class wraps a value of the primitive type `boolean` in an object.

Byte The Byte class is the standard wrapper for byte values.

Character The Character class wraps a value of the primitive type `char` in an object.

Character.Subset Instances of this class represent particular subsets of the Unicode character set.

Character.UnicodeBlock A family of character subsets representing the character blocks defined by the Unicode 2.0 specification.

Double The Double class wraps a value of the primitive type `double` in an object.

Float The Float class wraps a value of primitive type `float` in an object.

Integer The Integer class wraps a value of the primitive type `int` in an object.

Long The Long class wraps a value of the primitive type `long` in an object.

Number The abstract class `Number` is the superclass of classes `Byte`, `Double`, `Float`, `Integer`, `Long`, and `Short`.

Short The Short class is the standard wrapper for short values.

12.1.2 Tråde

Runnable The `Runnable` interface should be implemented by any class whose instances are intended to be executed by a thread.

Thread A *thread* is a thread of execution in a program.

ThreadGroup A thread group represents a set of threads.

ThreadLocal This class provides ThreadLocal variables.

InheritableThreadLocal This class extends ThreadLocal to provide inheritance of values from parent Thread to child Thread: when a child thread is created, the child receives initial values for all InheritableThreadLocals for which the parent has values.

Hvis en metode skal kaldes efter et stykke tid eller med regelmæssige tidsintervaller: Se også java.util.Timer og java.util.TimerTask for et alternativ til at arbejde direkte med tråde.

Hvis du bruger Swing: Bemærk, at efter et vindue er blevet synligt, bør du ikke kalde metoder i grafiske komponenter fra andre tråde end GUI−tråden. Den kan du aktivere med metoderne invokeLater() og invokeAndWait() på javax.swing.SwingUtilities.

12.1.3 Introspektion og dynamisk klasseindlæsning

Se også kapitel 11.

Class Instances of the class `Class` represent classes and interfaces in a running Java application.

ClassLoader The class `ClassLoader` is an abstract class.

Compiler The `Compiler` class is provided to support Java−to−native−code compilers and related services.

Package `Package` objects contain version information about the implementation and specification of a Java package.

Void The Void class is an uninstantiable placeholder class to hold a reference to the Class object representing the primitive Java type void.

Pakken java.lang.reflect har hjælpeklasser til introspektion:

Member Member is an interface that reflects identifying information about a single member (a field or a method) or a constructor.

AccessibleObject This is the base class for Field, Method and Constructor objects.

Field A `Field` provides information about, and dynamic access to, a single field of a class or an interface.

Method A `Method` provides information about, and access to, a single method on a class or interface.

Constructor `Constructor` provides information about, and access to, a single constructor for a class.

Array The `Array` class provides static methods to dynamically create and access Java arrays.

Modifier The Modifier class provides static methods and constants to decode class and member access modifiers.

InvocationHandler `InvocationHandler` is the interface implemented by the *invocation handler* of a proxy instance.

Proxy `Proxy` provides static methods for creating dynamic proxy classes and instances, and it is also the superclass of all dynamic proxy classes created by those methods.

ReflectPermission The Permission class for reflective operations.

I pakken java.beans findes en række hjælpeklasser til introspektion af javabønner.

12.1.4 Svage referencer (java.lang.ref)

Normalt kan et objekt kun blive smidt væk (frigivet fra hukommelsen), hvis der ikke er nogen referencer til det.

Svage referencer (eng.: weak references) er specielle objektreferencer, hvor det refererede objekt godt (i tilfælde af hukommelsesmangel) kan blive frigivet. Bruger man svage referencer, har man altså mulighed for at have objekter der bliver smidt væk i tilfælde af hukommelsesmangel.

Reference Abstract base class for reference objects.

WeakReference Weak reference objects, which do not prevent their referents from being made finalizable, finalized, and then reclaimed.

SoftReference Soft reference objects, which are cleared at the discretion of the garbage collector in response to memory demand.

PhantomReference Phantom reference objects, which are enqueued after the collector determines that their referents may otherwise be reclaimed.

ReferenceQueue Reference queues, to which registered reference objects are appended by the garbage collector after the appropriate reachability changes are detected.

12.1.5 Matematik med vilkårlig præcision (java.math)

Denne pakke (ikke at forveksle med den hyppigst brugte matematikklasse java.lang.Math) er beregnet til regning med vilkårligt antal decimaler. Den indeholder kun to klasser:

BigDecimal Immutable, arbitrary−precision signed decimal numbers.

BigInteger Immutable arbitrary−precision integers.

12.2 Værktøjsklasser (java.util)

Indeholder en række værktøjsklasser, dato− og tidsfaciliteter, internationalisering og en masse datastrukturer.

Random An instance of this class is used to generate a stream of pseudorandom numbers.

StringTokenizer The string tokenizer class allows an application to break a string into tokens.

Timer A facility for threads to schedule tasks for future execution in a background thread.

TimerTask A task that can be scheduled for one−time or repeated execution by a Timer.

Date The class `Date` represents a specific instant in time, with millisecond precision.

Calendar `Calendar` is an abstract base class for converting between a `Date` object and a set of integer fields such as `YEAR`, `MONTH`, `DAY`, `HOUR`, and so on.

GregorianCalendar `GregorianCalendar` is a concrete subclass of `Calendar` and provides the standard calendar used by most of the world.

TimeZone `TimeZone` represents a time zone offset, and also figures out daylight savings.

SimpleTimeZone `SimpleTimeZone` is a concrete subclass of `TimeZone` that represents a time zone for use with a Gregorian calendar.

Locale A `Locale` object represents a specific geographical, political, or cultural region.

ResourceBundle Resource bundles contain locale−specific objects.

PropertyResourceBundle `PropertyResourceBundle` is a concrete subclass of `ResourceBundle` that manages resources for a locale using a set of static strings from a property file.

ListResourceBundle `ListResourceBundle` is a abstract subclass of `ResourceBundle` that manages resources for a locale in a convenient and easy to use list.

Properties The `Properties` class represents a persistent set of properties.

PropertyPermission This class is for property permissions.

EventObject The root class from which all event state objects shall be derived.

EventListener A tagging interface that all event listener interfaces must extend.

Observable This class represents an observable object, or "data" in the model−view paradigm.

Observer A class can implement the `Observer` interface when it wants to be informed of changes in observable objects.

12.2.1 De "klassiske" datastrukturer i java.util

Disse datastrukturer har eksisteret siden JDK 1.0:

Comparator A comparison function, which imposes a *total ordering* on some collection of objects.

Enumeration An object that implements the Enumeration interface generates a series of elements, one at a time.

BitSet This class implements a vector of bits that grows as needed.

Dictionary The `Dictionary` class is the abstract parent of any class, such as `Hashtable`, which maps keys to values.

Hashtable This class implements a hashtable, which maps keys to values.

Stack The `Stack` class represents a last−in−first−out (LIFO) stack of objects.

Vector The `Vector` class implements a growable array of objects.

12.2.2 Collections−klasserne i java.util

Disse datastrukturer kom til i JDK 1.2. Mange af dem er beskrevet i kapitel 1.

Collections This class consists exclusively of static methods that operate on or return collections.

Arrays This class contains various methods for manipulating arrays (such as sorting and searching).

Collection The root interface in the *collection hierarchy*.

AbstractCollection This class provides a skeletal implementation of the `Collection` interface, to minimize the effort required to implement this interface.

Iterator An iterator over a collection.

ListIterator An iterator for lists that allows the programmer to traverse the list in either direction and modify the list during iteration.

Lister (datastrukturer hvor rækkefølgen af elementerne huskes):

List An ordered collection (also known as a *sequence*).

AbstractList This class provides a skeletal implementation of the `List` interface to minimize the effort required to implement this interface backed by a "random access" data store (such as an array).

ArrayList Resizable−array implementation of the `List` interface.

AbstractSequentialList This class provides a skeletal implementation of the `List` interface to minimize the effort required to implement this interface backed by a "sequential access" data store (such as a linked list).

LinkedList Linked list implementation of the `List` interface.

Mængder (datastrukturer hvor rækkefølgen af elementerne ikke huskes)

Set A collection that contains no duplicate elements.

AbstractSet This class provides a skeletal implementation of the `Set` interface to minimize the effort required to implement this interface.

HashSet This class implements the `Set` interface, backed by a hash table (actually a `HashMap` instance).

SortedSet A set that further guarantees that its iterator will traverse the set in ascending element order, sorted according to the *natural ordering* of its elements (see Comparable), or by a Comparator provided at sorted set creation time.

TreeSet This class implements the `Set` interface, backed by a `TreeMap` instance.

Afbildninger (datastrukturer hvor nøglen er et objekt)

Map An object that maps keys to values.

Map.Entry A map entry (key−value pair).

AbstractMap This class provides a skeletal implementation of the `Map` interface, to minimize the effort required to implement this interface.

HashMap Hash table based implementation of the `Map` interface.

WeakHashMap A hashtable−based `Map` implementation with *weak keys*.

SortedMap A map that further guarantees that it will be in ascending key order, sorted according to the *natural ordering* of its keys (see the `Comparable` interface), or by a comparator provided at sorted map creation time.

TreeMap Red−Black tree based implementation of the `SortedMap` interface.

12.3 Behandling af tekst (java.text)

Formatering og parsning af tekst, datoer og tal.

Format `Format` is an abstract base class for formatting locale−sensitive information such as dates, messages, and numbers.

FieldPosition `FieldPosition` is a simple class used by `Format` and its subclasses to identify fields in formatted output.

ParsePosition `ParsePosition` is a simple class used by `Format` and its subclasses to keep track of the current position during parsing.

ChoiceFormat A `ChoiceFormat` allows you to attach a format to a range of numbers.

DateFormat DateFormat is an abstract class for date/time formatting subclasses which formats and parses dates or time in a language−independent manner.

SimpleDateFormat `SimpleDateFormat` is a concrete class for formatting and parsing dates in a locale−sensitive manner.

DateFormatSymbols `DateFormatSymbols` is a public class for encapsulating localizable date−time formatting data, such as the names of the months, the names of the days of the week, and the time zone data.

DecimalFormat `DecimalFormat` is a concrete subclass of `NumberFormat` that formats decimal numbers.

DecimalFormatSymbols This class represents the set of symbols (such as the decimal separator, the grouping separator, and so on) needed by `DecimalFormat` to format numbers.

MessageFormat `MessageFormat` provides a means to produce concatenated messages in language−neutral way.

NumberFormat `NumberFormat` is the abstract base class for all number formats.

Sortering af strenge (afhængigt af sprog)

Collator The `Collator` class performs locale−sensitive `String` comparison.

CollationKey A `CollationKey` represents a `String` under the rules of a specific `Collator` object.

RuleBasedCollator The `RuleBasedCollator` class is a concrete subclass of `Collator` that provides a simple, data−driven, table collator.

Klasser til at gennemløbe tekst (afhængigt af sprog)

CharacterIterator This interface defines a protocol for bidirectional iteration over text.

StringCharacterIterator `StringCharacterIterator` implements the `CharacterIterater` protocol for a `String`.

BreakIterator The `BreakIterator` class implements methods for finding the location of boundaries in text.

AttributedCharacterIterator An AttributedCharacterIterator allows iteration through both text and related attribute information.

AttributedCharacterIterator.Attribute Defines attribute keys that are used to identify text attributes.

Annotation An Annotation object is used as a wrapper for a text attribute value if the attribute has annotation characteristics.

AttributedString An AttributedString holds text and related attribute information.

CollationElementIterator The `CollationElementIterator` class is used as an iterator to walk through each character of an international string.

12.4 Grafiktegning (java.awt del 1)

12.4.1 Generelt

Graphics The `Graphics` class is the abstract base class for all graphics contexts that allow an application to draw onto components that are realized on various devices, as well as onto off−screen images.

Color The `Color` class is used encapsulate colors in the default sRGB color space or colors in arbitrary color spaces identified by a `ColorSpace`.

Point A point representing a location in (x, y) coordinate space, specified in integer precision.

Dimension The `Dimension` class encapsulates the width and height of a component (in integer precision) in a single object.

Rectangle A `Rectangle` specifies an area in a coordinate space that is enclosed by the `Rectangle` object's top−left point (*x*, *y*) in the coordinate space, its width, and its height.

Polygon The `Polygon` class encapsulates a description of a closed, two−dimensional region within a coordinate space.

Font The `Font` class represents a font.

FontTMetrics The Font`Metrics` class defines a font metrics object, which encapsulates information about the rendering of a particular font on a particular screen.

Image The abstract class `Image` is the superclass of all classes that represent graphical images.

MediaTracker The `MediaTracker` class is a utility class to track the status of a number of media objects.

Cursor A class to encapsulate the bitmap representation of the mouse cursor.

12.4.2 Grafiktegning (Java2D)

De følgende klasser til avanceret 2D−grafik kaldes under et 'Java2D'. Se også kapitel 5.

Graphics2D This `Graphics2D` class extends the `Graphics` class to provide more sophisticated control over geometry, coordinate transformations, color management, and text layout.

Composite The `Composite` interface, along with `CompositeContext` , defines the methods to compose a draw primitive with the underlying graphics area.

CompositeContext The `CompositeContext` interface defines the encapsulated and optimized environment for a compositing operation.

AlphaComposite This `AlphaComposite` class implements the basic alpha compositing rules for combining source and destination pixels to achieve blending and transparency effects with graphics and images.

Shape The `Shape` interface provides definitions for objects that represent some form of geometric shape.

Stroke The `Stroke` interface allows a `Graphics2D` object to obtain a `Shape` that is the decorated outline, or stylistic representation of the outline, of the specified `Shape`.

BasicStroke The `BasicStroke` class defines a basic set of rendering attributes for the outlines of graphics primitives.

Transparency The `Transparency` interface defines the common transparency modes for implementing classes.

Paint This `Paint` interface defines how color patterns can be generated for `Graphics2D` operations.

PaintContext The `PaintContext` interface defines the encapsulated and optimized environment to generate color patterns in device space for fill or stroke operations on a `Graphics2D`.

GradientPaint The `GradientPaint` class provides a way to fill a `Shape` with a linear color gradient pattern.

TexturePaint The `TexturePaint` class provides a way to fill a `Shape` with a texture that is specified as a `BufferedImage`.

GraphicsConfiguration The `GraphicsConfiguration` class describes the characteristics of a graphics destination such as a printer or monitor.

GraphicsConfigTemplate The `GraphicsConfigTemplate` class is used to obtain a valid `GraphicsConfiguration`.

GraphicsDevice The `GraphicsDevice` class describes the graphics devices that might be available in a particular graphics environment.

GraphicsEnvironment The `GraphicsEnvironment` class describes the collection of `GraphicsDevice` objects and `Font` objects available to a Java(tm) application on a particular platform.

RenderingHints The `RenderingHints` class contains rendering hints that can be used by the `Graphics2D` class, and classes that implement `BufferedImageOp` and `Raster`.

RenderingHints.Key Defines the base type of all keys used to control various aspects of the rendering and imaging pipelines.

Pakken java.awt.font indeholder forskellige klasser og interfaces til nøjere undersøgelse og manipulation af skrifter.

Pakken java.awt.image indeholder funktioner til at oprette og manipulere med billeder.

Pakken java.awt.geom indeholder klasser og interfaces til undersøgelse og manipulation af geometriske former:

AffineTransform The `AffineTransform` class represents a 2D affine transform that performs a linear mapping from 2D coordinates to other 2D coordinates that preserves the "straightness" and "parallelness" of lines.

Arc2D The abstract superclass for all objects that store a 2D arc defined by a bounding rectangle, start angle, angular extent (length of the arc), and a closure type (`OPEN`, `CHORD`, or `PIE`).

Arc2D.Double An arc specified in double precision,

Arc2D.Float An arc specified in float precision,

Area The `Area` class is a device−independent specification of an arbitrarily−shaped area.

CubicCurve2D This class defines a cubic parametric curve segment in (x, y) coordinate space.

CubicCurve2D.Double A cubic parametric curve segment specified with double coordinates.

CubicCurve2D.Float A cubic parametric curve segment specified with float coordinates.

Dimension2D The `Dimension2D` class is to encapsulate a width and a height dimension.

Ellipse2D The `Ellipse2D` class describes an ellipse that is defined by a bounding rectangle.

Ellipse2D.Double The `Double` class defines an ellipse specified in `double` precision.

Ellipse2D.Float The `Float` class defines an ellipse specified in `float` precision.

FlatteningPathIterator This class returns a flattened view of another `PathIterator` object.

GeneralPath The `GeneralPath` class represents a geometric path constructed from straight lines, and quadratic and cubic (Bézier) curves.

Line2D This `Line2D` represents a line segment in (x, y) coordinate space.

Line2D.Double A line segment specified with double coordinates.

Line2D.Float A line segment specified with float coordinates.

Point2D The `Point2D` class defines a point representing a location in (x, y) coordinate space.

Point2D.Double The `Double` class defines a point specified in `double` precision.

Point2D.Float The `Float` class defines a point specified in float precision.

QuadCurve2D The `QuadCurve2D` class defines a quadratic parametric curve segment in (x, y) coordinate space.

QuadCurve2D.Double A quadratic parametric curve segment specified with `double` coordinates.

QuadCurve2D.Float A quadratic parametric curve segment specified with `float` coordinates.

Rectangle2D The `Rectangle2D` class describes a rectangle defined by a location (x, y) and dimension (w x h).

Rectangle2D.Double The `Double` class defines a rectangle specified in double coordinates.

Rectangle2D.Float The `Float` class defines a rectangle specified in float coordinates.

RectangularShape `RectangularShape` is the base class for a number of `Shape` objects whose geometry is defined by a rectangular frame.

RoundRectangle2D The `RoundRectangle2D` class defines a rectangle with rounded corners defined by a location (x, y), a dimension (w x h), and the width and height of an arc with which to round the corners.

RoundRectangle2D.Double The `Double` class defines a rectangle with rounded corners all specified in `double` coordinates.

RoundRectangle2D.Float The `Float` class defines a rectangle with rounded corners all specified in `float` coordinates.

12.4.3 Udskrivning

Pakken java.awt indeholder følgende klasser til håndtering af udskrivning:

PrintJob An abstract class which initiates and executes a print job.

JobAttributes A set of attributes which control a print job.

JobAttributes.DefaultSelectionType A type−safe enumeration of possible default selection states.

JobAttributes.DestinationType A type−safe enumeration of possible job destinations.

JobAttributes.DialogType A type−safe enumeration of possible dialogs to display to the user.

JobAttributes.MultipleDocumentHandlingType A type−safe enumeration of possible multiple document handling states.

JobAttributes.SidesType A type−safe enumeration of possible multi−page impositions.

PageAttributes A set of attributes which control the output of a printed page.

PageAttributes.ColorType A type−safe enumeration of possible color states.

PageAttributes.MediaType A type−safe enumeration of possible paper sizes.

PageAttributes.OrientationRequestedType A type−safe enumeration of possible orientations.

PageAttributes.OriginType A type−safe enumeration of possible origins.

PageAttributes.PrintQualityType A type−safe enumeration of possible print qualities.

Pakken java.awt.print, der kom til med JDK 1.2, indeholder:

Pageable The `Pageable` implementation represents a set of pages to be printed.

Printable The `Printable` interface is implemented by the `print` methods of the current page painter, which is called by the printing system to render a page.

PrinterGraphics The `PrinterGraphics` interface is implemented by `Graphics` objects that are passed to `Printable` objects to render a page.

Book The `Book` class provides a representation of a document in which pages may have different page formats and page painters.

PageFormat The `PageFormat` class describes the size and orientation of a page to be printed.

Paper The `Paper` class describes the physical characteristics of a piece of paper.

PrinterJob The `PrinterJob` class is the principal class that controls printing.

12.5 Grafiske brugergrænseflader (java.awt del 2)

De avancerede grafiske brugergrænseflader man kan lave med Swing−klasserne (pakken javax.swing) er af pladshensyn ikke beskrevet her.

Toolkit This class is the abstract superclass of all actual implementations of the Abstract Window Toolkit.

SystemColor A class to encapsulate symbolic colors representing the color of GUI objects on a system.

Robot This class is used to generate native system input events for the purposes of test automation, self−running demos, and other applications where control of the mouse and keyboard is needed.

Bemærk at klassen Robot også fange et udsnit af skærmen og gemme i et billede (fungere som screen grabber).

12.5.1 Grafiske komponenter i java.awt

Component A *component* is an object having a graphical representation that can be displayed on the screen and that can interact with the user.

De følgende arver alle fra Component

Button This class creates a labeled button.

Canvas A `Canvas` component represents a blank rectangular area of the screen onto which the application can draw or from which the application can trap input events from the user.

Checkbox A check box is a graphical component that can be in either an "on" (`true`) or "off" (`false`) state.

CheckboxGroup The `CheckboxGroup` class is used to group together a set of `Checkbox` buttons.

CheckboxMenuItem This class represents a check box that can be included in a menu.

Choice The `Choice` class presents a pop−up menu of choices.

Scrollbar The `Scrollbar` class embodies a scroll bar, a familiar user−interface object.

TextComponent The `TextComponent` class is the superclass of any component that allows the editing of some text.

TextArea A `TextArea` object is a multi−line region that displays text.

TextField A `TextField` object is a text component that allows for the editing of a single line of text.

Label A `Label` object is a component for placing text in a container.

List The `List` component presents the user with a scrolling list of text items.

12.5.2 Containere i java.awt

Container A generic Abstract Window Toolkit(AWT) container object is a component that can contain other AWT components.

Panel `Panel` is the simplest container class.

ScrollPane A container class which implements automatic horizontal and/or vertical scrolling for a single child component.

Window A `Window` object is a top−level window with no borders and no menubar.

Frame A Frame is a top−level window with a title and a border.

Dialog A Dialog is a top−level window with a title and a border that is typically used to take some form of input from the user.

FileDialog The `FileDialog` class displays a dialog window from which the user can select a file.

12.5.3 Appletter (pakken java.applet)

Denne pakke indeholder klassen Applet og et par hjælpeklasser.

12.5.4 Layout−managere i java.awt

BorderLayout A border layout lays out a container, arranging and resizing its components to fit in five regions: north, south, east, west, and center.

CardLayout A `CardLayout` object is a layout manager for a container.

FlowLayout A flow layout arranges components in a left−to−right flow, much like lines of text in a paragraph.

GridLayout The `GridLayout` class is a layout manager that lays out a container's components in a rectangular grid.

GridBagLayout The `GridBagLayout` class is a flexible layout manager that aligns components vertically and horizontally, without requiring that the components be of the same size.

GridBagConstraints The `GridBagConstraints` class specifies constraints for components that are laid out using the `GridBagLayout` class.

Insets An `Insets` object is a representation of the borders of a container.

12.5.5 Hændelser i java.awt

AWTEvent The root event class for all AWT events.

Pakken java.awt.event

De fleste hændelser og lyttere findes i pakken java.awt.event:

ActionListener The listener interface for receiving action events.

AdjustmentListener The listener interface for receiving adjustment events.

AWTEventListener The listener interface for receiving notification of events dispatched to objects that are instances of Component or MenuComponent or their subclasses.

ComponentListener The listener interface for receiving component events.

ContainerListener The listener interface for receiving container events.

FocusListener The listener interface for receiving keyboard focus events on a component.

HierarchyBoundsListener The listener interface for receiving ancestor moved and resized events.

HierarchyListener The listener interface for receiving hierarchy changed events.

InputMethodListener The listener interface for receiving input method events.

ItemListener The listener interface for receiving item events.

KeyListener The listener interface for receiving keyboard events (keystrokes).

MouseListener The listener interface for receiving "interesting" mouse events (press, release, click, enter, and exit) on a component.

MouseMotionListener The listener interface for receiving mouse motion events on a component.

TextListener The listener interface for receiving text events.

WindowListener The listener interface for receiving window events.

ActionEvent A semantic event which indicates that a component−defined action occured.

AdjustmentEvent The adjustment event emitted by Adjustable objects.

ComponentAdapter An abstract adapter class for receiving component events.

ComponentEvent A low−level event which indicates that a component moved, changed size, or changed visibility (also, the root class for the other component−level events).

ContainerAdapter An abstract adapter class for receiving container events.

ContainerEvent A low−level event which indicates that a container's contents changed because a component was added or removed.

FocusAdapter An abstract adapter class for receiving keyboard focus events.

FocusEvent A low−level event which indicates that a component has gained or lost the keyboard focus.

HierarchyBoundsAdapter An abstract adapter class for receiving ancestor moved and resized events.

HierarchyEvent An event which indicates a change to the Component hierarchy to which a Component belongs.

InputEvent The root event class for all component−level input events.

InputMethodEvent Input method events contain information about text that is being composed using an input method.

InvocationEvent An event which executes the `run()` method on a `Runnable` when dispatched by the AWT event dispatcher thread.

ItemEvent A semantic event which indicates that an item was selected or deselected.

KeyAdapter An abstract adapter class for receiving keyboard events.

KeyEvent An event which indicates that a keystroke occurred in a component.

MouseAdapter An abstract adapter class for receiving mouse events.

MouseEvent An event which indicates that a mouse action occurred in a component.

MouseMotionAdapter An abstract adapter class for receiving mouse motion events.

PaintEvent The component−level paint event.

TextEvent A semantic event which indicates that an object's text changed.

WindowAdapter An abstract adapter class for receiving window events.

WindowEvent A low−level event which indicates that a window has changed its status.

12.5.6 Menuer i java.awt

Menu A `Menu` object is a pull−down menu component that is deployed from a menu bar.

MenuBar The `MenuBar` class encapsulates the platform's concept of a menu bar bound to a frame.

MenuComponent The abstract class `MenuComponent` is the superclass of all menu−related components.

MenuItem All items in a menu must belong to the class `MenuItem`, or one of its subclasses.

MenuShortcut A class which represents a keyboard accelerator for a MenuItem.

PopupMenu A class that implements a menu which can be dynamically popped up at a specified position within a component.

12.5.7 Kopiering, udklipsholder og træk−og−slip

java.awt.datatransfer

Pakken java.awt.datatransfer med bl.a. klassen Clipboard arbejder med udklipsholderen og overførsel af data til/fra andre programmer.

java.awt.dnd

Pakken java.awt.dnd sørger for funktionalitet til, at brugeren kan trække data fra et sted til et andet (Drag 'N' Drop).

12.6 Lyd

12.6.1 WAV/indspillet lyd (javax.sound.sampled)

Denne pakke tillader at optage, behandle og afspille samplet lyd (f.eks. WAV−filer).

Clip The `Clip` interface represents a special kind of data line whose audio data can be loaded prior to playback, instead of being streamed in real time.

DataLine `DataLine` adds media−related functionality to its superinterface, `Line`.

Line The `Line` interface represents a mono or multi−channel audio feed.

LineListener Instances of classes that implement the `LineListener` interface can register to receive events when a line's status changes.

Mixer A mixer is an audio device with one or more lines.

Port Ports are simple lines for input or output of audio to or from audio devices.

SourceDataLine A source data line is a data line to which data may be written.

TargetDataLine A target data line is a type of `DataLine` from which audio data can be read.

AudioFileFormat An instance of the `AudioFileFormat` class describes an audio file, including the file type, the file's length in bytes, the length in sample frames of the audio data contained in the file, and the format of the audio data.

AudioFileFormat.Type An instance of the `Type` class represents one of the standard types of audio file.

AudioFormat `AudioFormat` is the class that specifies a particular arrangement of data in a sound stream.

AudioFormat.Encoding The `Encoding` class names the specific type of data representation used for an audio stream.

AudioInputStream An audio input stream is an input stream with a specified audio format and length.

AudioPermission The `AudioPermission` class represents access rights to the audio system resources.

AudioSystem The `AudioSystem` class acts as the entry point to the sampled−audio system resources.

BooleanControl A `BooleanControl` provides the ability to switch between two possible settings that affect a line's audio.

BooleanControl.Type An instance of the `BooleanControl.Type` class identifies one kind of boolean control.

CompoundControl A `CompoundControl`, such as a graphic equalizer, provides control over two or more related properties, each of which is itself represented as a `Control`.

CompoundControl.Type An instance of the `CompoundControl.Type` inner class identifies one kind of compound control.

Control `Lines` often have a set of controls, such as gain and pan, that affect the audio signal passing through the line.

Control.Type An instance of the `Type` class represents the type of the control.

DataLine.Info Besides the class information inherited from its superclass, `DataLine.Info` provides additional information specific to data lines.

EnumControl A `EnumControl` provides control over a set of discrete possible values, each represented by an object.

EnumControl.Type An instance of the `EnumControl.Type` inner class identifies one kind of enumerated control.

FloatControl A `FloatControl` object provides control over a range of floating−point values.

FloatControl.Type An instance of the `FloatControl.Type` inner class identifies one kind of float control.

Line.Info A `Line.Info` object contains information about a line.

LineEvent The `LineEvent` class encapsulates information that a line sends its listeners whenever the line opens, closes, starts, or stops.

LineEvent.Type The LineEvent.Type inner class identifies what kind of event occurred on a line.

Mixer.Info The `Mixer.Info` class represents information about an audio mixer, including the product's name, version, and vendor, along with a textual description.

Port.Info The `Port.Info` class extends `Line.Info` with additional information specific to ports, including the port's name and whether it is a source or a target for its mixer.

ReverbType The `ReverbType` class provides methods for accessing various reverberation settings to be applied to an audio signal.

12.6.2 MIDI/nodebaseret musik (javax.sound.midi)

Denne pakke giver mulighed for læsning, skrivning, sequencing og syntese af MIDI−data (Musical Instrument Digital Interface).

ControllerEventListener The `ControllerEventListener` interface should be implemented by classes whose instances need to be notified when a `Sequencer` has processed a requested type of MIDI control−change event.

MetaEventListener The `MetaEventListener` interface should be implemented by classes whose instances need to be notified when a `Sequencer` has processed a `MetaMessage`.

MidiChannel A `MidiChannel` object represents a single MIDI channel.

MidiDevice `MidiDevice` is the base interface for all MIDI devices.

Receiver A `Receiver` receives `MidiEvent` objects and typically does something useful in response, such as interpreting them to generate sound or raw MIDI output.

Sequencer A hardware or software device that plays back a MIDI `sequence` is known as a *sequencer*.

Soundbank A `Soundbank` contains a set of `Instruments` that can be loaded into a `Synthesizer`.

Synthesizer A `Synthesizer` generates sound.

Transmitter A `Transmitter` sends `MidiEvent` objects to one or more `Receivers`.

Instrument An instrument is a sound−synthesis algorithm with certain parameter settings, usually designed to emulate a specific real−world musical instrument or to achieve a specific sort of sound effect.

MetaMessage A `MetaMessage` is a `MidiMessage` that is not meaningful to synthesizers, but that can be stored in a MIDI file and interpreted by a sequencer program.

MidiDevice.Info A `MidiDevice.Info` object contains assorted data about a `MidiDevice`, including its name, the company who created it, and descriptive text.

MidiEvent MIDI events contain a MIDI message and a corresponding time−stamp expressed in ticks, and can represent the MIDI event information stored in a MIDI file or a `Sequence` object.

MidiFileFormat A `MidiFileFormat` object encapsulates a MIDI file's type, as well as its length and timing information.

MidiMessage `MidiMessage` is the base class for MIDI messages.

MidiSystem The `MidiSystem` class provides access to the installed MIDI system resources, including devices such as synthesizers, sequencers, and MIDI input and output ports.

Patch A `Patch` object represents a location, on a MIDI synthesizer, into which a single instrument is stored (loaded).

Sequence A `Sequence` is a data structure containing musical information (often an entire song or composition) that can be played back by a `Sequencer` object.

Sequencer.SyncMode A `SyncMode` object represents one of the ways in which a MIDI sequencer's notion of time can be synchronized with a master or slave device.

ShortMessage A `ShortMessage` contains a MIDI message that has at most two data bytes following its status byte.

SoundbankResource A `SoundbankResource` represents any audio resource stored in a `Soundbank`.

SysexMessage A `SysexMessage` object represents a MIDI system exclusive message.

Track A MIDI track is an independent stream of MIDI events (time−stamped MIDI data) that can be stored along with other tracks in a standard MIDI file.

VoiceStatus A `VoiceStatus` object contains information about the current status of one of the voices produced by a `Synthesizer`.

12.7 Netværkskommunikation (java.net)

InetAddress This class represents an Internet Protocol (IP) address.

Socket This class implements client sockets (also called just "sockets").

ServerSocket This class implements server sockets.

SocketImpl The abstract class `SocketImpl` is a common superclass of all classes that actually implement sockets.

SocketImplFactory This interface defines a factory for socket implementations.

SocketOptions Interface of methods to get/set socket options.

SocketPermission This class represents access to a network via sockets.

DatagramSocket This class represents a socket for sending and receiving datagram packets.

DatagramSocketImpl Abstract datagram and multicast socket implementation base class.

DatagramSocketImplFactory This interface defines a factory for datagram socket implementations.

MulticastSocket The multicast datagram socket class is useful for sending and receiving IP multicast packets.

DatagramPacket This class represents a datagram packet.

URL Class `URL` represents a Uniform Resource Locator, a pointer to a "resource" on the World Wide Web.

URLClassLoader This class loader is used to load classes and resources from a search path of URLs referring to both JAR files and directories.

URLConnection The abstract class `URLConnection` is the superclass of all classes that represent a communications link between the application and a URL.

HttpURLConnection A URLConnection with support for HTTP−specific features.

JarURLConnection A URL Connection to a Java ARchive (JAR) file or an entry in a JAR file.

FileNameMap A simple interface which provides a mechanism to map between a file name and a MIME type string.

URLDecoder The class contains a utility method for converting from a MIME format called "`x−www−form−urlencoded`" to a `String`

URLEncoder The class contains a utility method for converting a `String` into a MIME format called "`x−www−form−urlencoded`" format.

URLStreamHandler The abstract class `URLStreamHandler` is the common superclass for all stream protocol handlers.

URLStreamHandlerFactory This interface defines a factory for `URL` stream protocol handlers.

ContentHandler The abstract class `ContentHandler` is the superclass of all classes that read an `Object` from a `URLConnection`.

ContentHandlerFactory This interface defines a factory for content handlers.

Authenticator The class Authenticator represents an object that knows how to obtain authentication for a network connection.

PasswordAuthentication The class PasswordAuthentication is a data holder that is used by Authenticator.

NetPermission This class is for various network permissions.

12.8 Databasekommunikation (java.sql)

Programmeringsgrænsefladen til databaser (JDBC). Se også kapitel 8.

Making a connection with a data source:

DriverManager The basic service for managing a set of JDBC drivers.
Driver The interface that every driver class must implement.

DriverPropertyInfo Driver properties for making a connection.

Connection A connection (session) with a specific database.

DatabaseMetaData Comprehensive information about the database as a whole.

SQLPermission The permission for which the `SecurityManager` will check when code that is running in an applet calls one of the `setLogWriter` methods.

Sending SQL statements to a database:

Statement The object used for executing a static SQL statement and obtaining the results produced by it.

PreparedStatement An object that represents a precompiled SQL statement.

CallableStatement The interface used to execute SQL stored procedures.

ResultSet A table of data representing a database result set, which is usually generated by executing a statement that queries the database.

ResultSetMetaData An object that can be used to get information about the types and properties of the columns in a `ResultSet` object.

Mapping an SQL value to the standard mapping in the Java programming language:

Array The mapping in the Java programming language for the SQL type `ARRAY`.

Blob The representation (mapping) in the Java™ programming language of an SQL `BLOB` value.

Clob The mapping in the Java™ programming language for the SQL `CLOB` type.

Date A thin wrapper around a millisecond value that allows JDBC to identify this as a SQL DATE.

Ref The mapping in the Java programming language of an SQL `REF` value, which is a reference to an SQL structured type value in the database.

Struct The standard mapping in the Java programming language for an SQL structured type.

Time A thin wrapper around `java.util.Date` that allows JDBC to identify this as a SQL `TIME` value.

Timestamp A thin wrapper around `java.util.Date` that allows the JDBC API to identify this as an SQL `TIMESTAMP` value.

Types The class that defines the constants that are used to identify generic SQL types, called JDBC types.

Custom mapping an SQL user−defined type to a class in the Java programming language:

SQLData The interface used for the custom mapping of SQL user−defined types.

SQLInput An input stream that contains a stream of values representing an instance of an SQL structured or distinct type.

SQLOutput The output stream for writing the attributes of a user−defined type back to the database.

12.9 Fjernkald af metoder (java.rmi)

RMI (Remote Method Invocation) er en måde at arbejde med objekter, der eksisterer i en anden Java virtuel maskine (ofte på en anden fysisk maskine), som om de var lokale objekter.

Remote The `Remote` interface serves to identify interfaces whose methods may be invoked from a non−local virtual machine.

Naming The `Naming` class provides methods for storing and obtaining references to remote objects in the remote object registry.

MarshalledObject A `MarshalledObject` contains a byte stream with the serialized representation of an object given to its constructor.

RMISecurityManager `RMISecurityManager` provides an example security manager for use by RMI applications that use downloaded code.

Pakken java.rmi.server

Denne pakke har klasser og interfaces til værtssiden af RMI.

RemoteObject The `RemoteObject` class implements the `java.lang.Object` behavior for remote objects.

RemoteServer The `RemoteServer` class is the common superclass to server implementations and provides the framework to support a wide range of remote reference semantics.

UnicastRemoteObject The UnicastRemoteObject class defines a non−replicated remote object whose references are valid only while the server process is alive.

RemoteStub The `RemoteStub` class is the common superclass to client stubs and provides the framework to support a wide range of remote reference semantics.

RemoteRef `RemoteRef` represents the handle for a remote object.

ServerRef A ServerRef represents the server−side handle for a remote object implementation.

Unreferenced A remote object implementation should implement the `Unreferenced` interface to receive notification when there are no more clients that reference that remote object.

RMISocketFactory An `RMISocketFactory` instance is used by the RMI runtime in order to obtain client and server sockets for RMI calls.

RMIClientSocketFactory An `RMIClientSocketFactory` instance is used by the RMI runtime in order to obtain client sockets for RMI calls.

RMIServerSocketFactory An `RMIServerSocketFactory` instance is used by the RMI runtime in order to obtain server sockets for RMI calls.

RMIFailureHandler An `RMIFailureHandler` can be registered via the `RMISocketFactory.setFailureHandler` call.

RMIClassLoader `RMIClassLoader` provides static methods for loading classes from a network location (one or more URLs) and obtaining the location from which an existing class can be loaded.

ObjID An `ObjID` is used to identify remote objects uniquely in a VM over time.

UID Abstraction for creating identifiers that are unique with respect to the the host on which it is generated.

Pakken java.rmi.activation

Denne pakke giver mulighed for objektaktivering (eng.: RMI Object Activation). Et fjernobjekts reference kan laves "persistent" og senere aktiveres til et "levende" objekt igen.

Pakken java.rmi.dgc

Denne pakke indeholder klasser til distribueret garbage−collection (DGC).

Pakken java.rmi.registry

Lille pakke med ekstra klasser til RMI's registreringsværktøj (rmiregistry).

Pakken javax.naming (navnetjenester og JNDI)

Pakker og klasser til Java Naming and Directory Interface (JNDI), der understøtter navneopslag og registrering i forskellige navnetjenester, bl.a. LDAP.

Pakken javax.rmi, javax.transaction og org.omg.CORBA (CORBA)

Pakker og klasser til CORBA (fjernkald af metoder på objekter, der er programmeret i andre programmeringssprog end Java, f.eks. C og C++) og RMI over IIOP (Internet Inter−Orb Protokollen).

12.10 Sikkerhed, kryptering, adgangskontrol (java.security)

Pakken java.security og dens underpakker understøtter en række systemer til checksum, kryptering, certifikater, digitale signaturer og nøgler, såsom SHA, MD2, MD5, DES, og RSA.

Med underpakkerne er der i alt omkring 100 klasser.

javabog.dk | << forrige | indhold | næste >> | programeksempler | om bogen

13 Mobiltelefoner (J2ME)

J2ME (Java 2 Micro Edition) er Suns Java−udviklingsplatform til at lave applikationer, der er beregnet til apparater med begrænset hukommelse.

Det er muligt at sammensætte bestanddelene i J2ME, sådan at man får de klasser, der er brug for til en bestemt kategori af apparater.

Der, hvor J2ME har vundet størst udbredelse, er inden for udviklingen af små programmer beregnet til mobiltelefoner (kaldet midletter), og det er det, vi vil beskæftige os med her. Andre anvendelsesmuligheder beskrives kort i afsnit 13.9.

For kortheds skyld vil vi benytte benævnelsen 'telefon' i det følgende, og det er op til læseren selv at læse 'telefon' som 'mobiltelefoner og en lang række andre små håndholdte computere og apparater'.

13.1 Introduktion til midletter

Midletter er små programmer, som er beregnet til at køre på en mobiltelefon. Ordet 'midlet' kommer fra MIDP (Mobile Information Device Profile) og skal opfattes analogt med ordet 'applet'. Ligesom appletter skal arve fra klassen Applet, skal midletter arve fra klassen MIDlet (der ligger i pakken javax.microedition.midlet).

13.1.1 Eksempel − en simpel midlet

Lad os se nærmere på, hvad man skal gøre, hvis man vil skrive sine egne midletter. Nedenfor er et simpelt eksempel på en midlet, der viser en (fiktiv) vejrudsigt:


```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class Vejrmidlet extends MIDlet
{
   // systemet starter midletten
  public void startApp()
  {
    // opret et skærmbillede (en liste)
    List sb = new List("Vejret", List.IMPLICIT);
    sb.append("Det bliver let skyet og blæsende", null);
    sb.append("Temperatur mellem 17 og 22 grader", null);

    // vis skærmbilledet
    Display.getDisplay(this).setCurrent( sb );
  }

   // systemet standser midletten
  public void pauseApp() {}

   // systemet smider midletten væk
  public void destroyApp(boolean unconditional) {}
}
```

I dette eksempel bruger vi en List (en valgliste) og tilføjer den tekst vi ønsker skal vises ud for hvert af listens elementer.

Når startApp() kaldes af systemet skal midletten bestemme, hvad der vises på skærmen. Det gøres med

```
  Display.getDisplay(this).setCurrent( skærmbillede )
```

hvor 'skærmbillede' kan være f.eks. en List, TextBox, Form eller Canvas (se senere).

13.1.2 Prøvekøre en midlet

Før du kan oversætte og køre en midlet skal du have fat i et udviklingsværktøj til mobiltelefonudvikling. Udvilkingsværktøjer til J2ME behandles i afsnit 13.8.

Under udviklingen af en midlet vil man normalt også prøve at køre den, før man overfører programmet til en rigtig telefon. Til dette formål har de fleste udvilkingsværktøjer en emulator indbygget, der kan udføre midletten, som om den kørte på en rigtig telefon. Figuren til højre for programudskriften ovenfor viser en sådan emulator (kaldet DefaultColorPhone).

Hvis ens program består af flere klasser, er det nødvendigt at pakke dem i en JAR−fil, for at de kan hentes ned af brugeren. Derudover skal man lave en manifestfil, der indeholder information om indholdet i JAR−filen. Man kan oprette disse filer med de fleste udviklingsværktøj til midletter.

13.1.3 Midletters livscyklus

En midlets livscyklus styres af et program i mobiltelefonen, der hedder Application Management Software (AMS).

Når brugeren ønsker at køre en midlet, henter AMS midletten ned fra en server og starter midletten:

- Først kaldes midlettens **konstruktør**. Mens konstruktøren udføres, er midletten standset, og den kan ikke modtage brugerinput eller vise noget på skærmen.
- Når konstruktøren er færdig med at blive udført, kalder AMS metoden **startApp()**, hvorved midletten gøres aktiv og venter på input fra brugeren (eller fra en server).
- AMS kan kalde **pauseApp()** for at standse midlettens programudførsel og give kontrollen til et andet program, for eksempel et telefonopkald.
  Senere, når det andet program er afsluttet, genoptager AMS midlettens programudførelse ved at kalde startApp() igen.
- Når AMS vil frigive hukommelsen til et andet program, kalder den **destroyApp()** og smider midletten væk.

Metoderne startApp(), pauseApp() og destroyApp() er abstrakte og skal derfor altid implementeres.

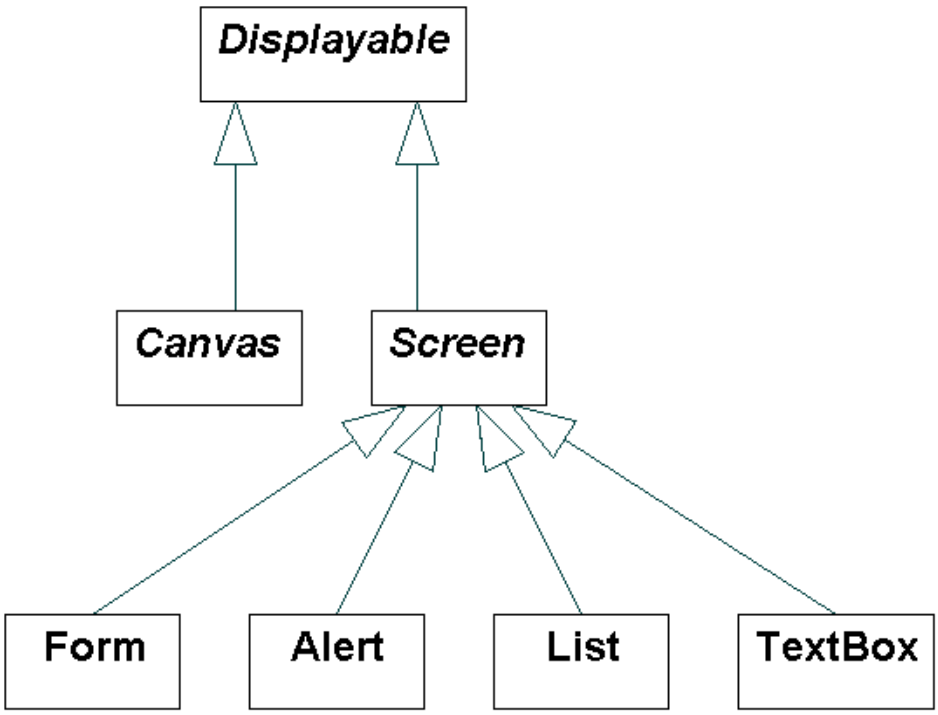
Kode, der skal udføres ved initialiseringen af midletten, bør egentlig være i midlettens konstruktør fremfor i startApp().

I pauseApp() kan man placere kode til at gemme data indtil startApp() kaldes igen.

13.2 Brugergrænseflader i midletter

De muligheder, man har for at lave brugergrænseflader til en midlet, er defineret i pakken javax.microedition.lcdui.

Det centrale element, når man skal designe brugergrænseflader, er et skærmbillede (en klasse, der arver fra Displayable). Et skærmbillede er et objekt, som viser grafik og tager imod indtastninger fra brugeren. Der kan kun være ét skærmbillede synligt ad gangen.



Klassen Canvas er beregnet til at lave direkte grafiktegning (til f.eks. spil). Bruger man denne klasse, må man selv tage højde for skærmstørrelsen og farvedybden. Denne klasse er nærmere beskrevet i afsnit 13.3, Direkte grafiktegning og spil.

Der er 4 klasser til at lave grafiske brugergrænseflader v.hj.a. standardkomponenter, nemlig Form, Alert, List og TextBox. Når man bruge disse klasser behøver man ikke tænke på hvilken skærmstørrelse og farvedybde telefonen har (de er programmeret til at ligne det øvrige menusystem på den pågældende telefon). Disse klasser er nærmere beskrevet i afsnit 13.4 Grafiske standardkomponenter.

13.2.1 Klassen Display (den fysiske skærm)

Klassen Display repræsenterer den fysiske skærm og har bl.a. metoderne:

static Display getDisplay(midlet) skaffer Display−objektet

boolean isColor() om telefonen har farveskærm

int numColors() giver antallet af farver eller gråtoner på telefonen

void setCurrent(Displayable skærmbillede) beder telefonen vise et bestemt skærmbillede

Displayable getCurrent() giver det aktuelle skærmbillede, der vises

boolean flashBacklight(int varighed) blinker med telefonens baggrundsbelysning

boolean vibrate(int varighed) vibrerer med telefonen

Midletten kan skifte mellem de forskellige skærmbilleder ved at kalde metoden setCurrent(Displayable) på Display−objektet.

Klassen Display er implementeret som en singleton, så der kan kun være et skærmbillede synligt ad gangen. Man får fat i Display−objektet ved at kalde Display.getDisplay() med midletten som parameter.

13.2.2 Klassen Displayable (skærmbilleder)

Klassen Displayable − metoder fælles for alle skærmbilleder

String getTitle() giver titlen på skærmbilledet

void setTitle(String titel) sætter titlen

Ticker getTicker() giver rulleteksten på skærmbilledet

void setTicker(Ticker rulletekst) sætter en rulletekst (i stedet for den gamle)

boolean isShown() tjekker, om skærmbilledet er synligt

void addCommand(Command k) føjer en kommando til skærmen

void removeCommand(Command k) sletter en kommando fra skærmen

void setCommandListener(CommandListener l) sæt lytter til kommandoer for dette skærmbillede

int getWidth() giver bredden af skærmbilledet

int getHeight() giver højden af skærmbilledet

13.2.3 Klassen Ticker (rulletekst på et skærmbillede)

Man kan få en lille tekst til at rulle hen over skærmen (en rulletekst − eng.: ticker). Det gøres med f.eks.:

```
Ticker rulletekst = new Ticker("Dette er en tekst der ruller hen over skærmen");
Displayable skærmbillede = new List("Vejret", List.IMPLICIT);
skærmbillede.setTicker( rulletekst );
```

Klassen Ticker (rulletekst på et skærmbillede)

String getString() aflæser rulleteksten

void setString(String str) sætter rulleteksten

13.2.4 Klassen Image (billeder)

Et Image−objekt bruges til at tegne grafikbilleder på skærmen (med Canvas) eller i forskellige grafiske standardkomponenter såsom Alert, Choice, Form og ImageItem.

Klassen Image

static Image createImage(Image b) opretter et billede ud fra det eksisterende billede b

static Image createImage(Image b, int x, y, br, hø, transf) ditto, men med transformation og klipning

static Image createImage(byte[] data, int afs, int lgd) opretter billede fra array af byte

static Image createImage(String navnPåResurse) opretter billede fra navngiven resurse

static Image createImage(InputStream) opretter billede fra datastrøm

static Image createRGBImage(int[] rgb, int br, hø, boolean alfa) opretter billede fra farvedata

static Image createImage(int bredde, int højde) opretter et **foranderligt** billede (der kan tegnes på)

Graphics getGraphics() grafiktegning på et (foranderligt) billede

boolean isMutable() sand, hvis foranderligt, falsk, hvis uforanderligt

int getWidth() giver bredden af billedet

int getHeight() giver højden af billedet

void getRGB(int[] rgb, int afs, scLgd, x, y, br, hø) aflæser billeddata og gemmer i arrayet rgb

Et billede er, afhængigt af hvordan det er oprettet, enten uforanderligt (dvs. umuligt at ændre i, når det først er oprettet), eller foranderligt (dvs. der kan ændres i det, efter at det er blevet oprettet).

Uforanderlige billeder oprettes ved at kalde createImage() med et array af byte, en datastrøm fra en fil eller netværket.

Foranderlige (eng.: mutable) billeder kan man ændre i via et Graphics objekt. De oprettes ved at kalde createImage() med en bredde og højde og starter med at være helt hvide.

Bruges et foranderligt billede i grafiske standardkomponenter, såsom Alert, Choice, Form og ImageItem, vil der blive taget en kopi af billedet.

13.2.5 Klassen Font (skrifttyper)

Man kan bede om en given skrifttype ved hjælp af metoden Font.getFont() og angive skriftstil, størrelse og type som parametre. Metoden vil returnere den skrifttype, som kommer tættest på det ønskede, ud fra de muligheder den givne telefon har.

Klassen Font

static Font getDefaultFont() giver systemets standardskrifttype

static Font getFont(int skriftnr) giver standardskriften for enten:
static int FONT_STATIC_TEXT eller FONT_INPUT_TEXT

static Font getFont(int type, int stil, int størrelse) giver skrifttype tættest på de angivne parametre
static int STYLE_PLAIN, STYLE_BOLD, STYLE_ITALIC, STYLE_UNDERLINED skriftstil
static int SIZE_SMALL, SIZE_MEDIUM, SIZE_LARGE skriftstørrelse
static int FACE_SYSTEM, FACE_MONOSPACE, FACE_PROPORTIONAL skrifttype

int getStyle() returnerer skriftstilen (fed, kursiv, understreget)

int getSize() giver skriftstørrelsen

int getFace() giver skrifttypen

boolean isPlain() giver sand, hvis skriftstilen er 'plain' (ikke fed, kursiv, ...)

boolean isBold() giver sand, hvis skriftstilen er sat til fed skrift

boolean isItalic() giver sand, hvis skriftstilen er sat til kursiv

boolean isUnderlined() giver sand, hvis skriftstilen er sat til understreget

int getHeight() giver standardhøjden af en tekstlinje

int getBaselinePosition() giver afstand i punkter fra top af teksten til grundlinjen

int charWidth(char tegn) giver bredden af et tegn i skærmpunkter

int charsWidth(char[], int afs, int lgd) giver bredden af et array af tegn

int stringWidth(String streng) giver bredden af en streng

int substringWidth(String str, int afs, int lgd) giver bredden af en delstreng

13.3 Direkte grafiktegning og spil

Hvis man vil lave sin egen grafik, skal man arve fra klassen Canvas og definere dens paint()−metode. Canvas er også beregnet til at lave spil og har metoder til at opfange brugerens tastetryk og evt. brug af et pegeredskab.

Her er et eksempel, der tegner en tekst og en firkant, der kan styres med piletasterne.

Først skærmbilledet, der viser grafikken:


```
import javax.microedition.lcdui.*;

public class Canvasgrafik extends Canvas
{
  private int x,y;

  public Canvasgrafik()
  {
    x = getWidth()/2;
    y = getHeight()/2;
  }

  public void paint(Graphics g)
  {
    // slet baggrunden
    g.setColor( 0x00ffffff ); // hvid
    g.fillRect(0, 0, getWidth(), getHeight());

    g.setColor( 0x00000000 ); // sort
    g.drawString("Brug piletasterne",0,0,
      Graphics.TOP|Graphics.LEFT);
    g.fillRect(x,y,3,3);
  }

  protected void keyPressed(int tastkode)
  {
    switch (getGameAction(tastkode)) {
      case UP:   y--; break;
      case DOWN: y++; break;
      case LEFT: x--; break;
      case RIGHT:x++; break;
    }
    repaint();  // bed systemet kalde paint
  }
}
```

Vi har defineret paint() for at kunne tegne noget på skærmen. Graphics−objektet, der overføres, minder meget om det Graphics−objekt, man kender fra java.awt.

Vi har også defineret keyPressed() for at få at vide, når brugeren trykker på en tast. For at få tastkoden omsat til en handling i 'spillet' på en platformsuafhængig måde kalder vi getGameAction().

Her er midletten, der viser Canvas−objektet:

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class VisCanvasgrafik extends MIDlet
{
   Canvasgrafik grafik = new Canvasgrafik();

   public void startApp()
   {
      Display.getDisplay(this).setCurrent(grafik);
   }

   public void pauseApp() {}
   public void destroyApp(boolean unconditional) {}
}
```

13.3.1 Klassen Canvas

Klassen Canvas (der arver fra Displayable) er beregnet til at nedarve fra.

Metoder i klassen Canvas, der kan kaldes af programmøren

boolean isDoubleBuffered() om telefonen bruger dobbelt tegnebuffer til jævn grafik

boolean hasPointerEvents() om telefonen har en pegeenhed (tryk og slip−hændelser)

boolean hasPointerMotionEvents() om telefonen har en pegeenhed, der har træk−hændelser

boolean hasRepeatEvents() om tasterne kan repetere (se keyRepeated() senere)

String getKeyName(int tastkode) oversætter en tastkode til en streng, der beskriver tasten

int getGameAction(int tastkode) oversætter en tastkode til handling (f.eks. i et spil)

int getKeyCode(int handling) oversætter en handling (f.eks. i et spil) til en tastkode
tastkode>0 svarer til unikode−tegnet, så tegn=(char)tastkode
tastkoder på telefonen: KEY_NUM0, ... , KEY_NUM9, KEY_STAR, KEY_POUND
handlinger (spilknapper): UP, DOWN, LEFT, RIGHT, FIRE, GAME_A, ... , GAME_D

void setFullScreenMode(boolean) sætter fuldskærmstilstand (uden titel, menuer etc.)

void repaint() beder systemet om at gentegne hele skærmen

void repaint(x, y, br, hø) beder systemet om at gentegne et område af skærmen

void serviceRepaints() gennemtvinger evt. ventende gentegninger omgående

Metoder på Canvas, som systemet kalder

De følgende metoder kan du definere i nedarvingen. Systemet kalder dem på bestemte tidspunkter. De kan altså tilsidesættes for at få kaldt noget kode i bestemte tilfælde:

Tomme metoder som systemet kalder i Canvas (beregnet til, at programmøren definerer dem)

```
  // skal defineres. Kaldes, når skærmbilledet skal gentegnes
  protected void paint(Graphics g)

  // Kaldes med en tastkode, når brugeren ...
  protected void keyPressed(int tast)   // trykker en tast ned
  protected void keyRepeated(int tast)  // holder en tast nede, så den repeterer
  protected void keyReleased(int tast)  // når brugeren slipper tasten igen

  // Kaldes når en evt. pegeenhed (f.eks. en mus) ...
  protected void pointerPressed(x,y)    // trykkes ned (a la mousePressed)
  protected void pointerDragged(x,y)    // trækkes nedtrykket (a la mouseDragged)
  protected void pointerReleased(x,y)   // slippes (a la mouseReleased)

  // Kaldes lige før skærmbilledet bliver vist (bliver synligt på skærmen)
  protected void showNotify()
  // Kaldes lige efter at skærmbilledet er blevet skjult (ikke mere er synligt)
  protected void hideNotify()

  // Kaldtes når skærmbilledet skifter størrelse (f.eks. til fuld skærm)
  protected void sizeChanged(int nyBredde, int nyHøjde)
```

13.3.2 Klassen GameCanvas og lagdelt grafik

Klassen GameCanvas, der findes pakken javax.microedition.lcdui.game, er endnu mere specialiseret mod spil. Den udvider Canvas med faciliteter som dobbeltbuffer (at grafikken først tegnes i en separat tegnebuffer og derefter vises på skærmen for at opnå mere flydende grafik) og tastestatus (om en tast er holdt nede eller ej).

I samme pakke findes også understøttelse for lagdelt grafik. Det er interessant, når man skal tegne f.eks. en bane og nogle bevægelige figurer på banen (kaldet sprites).

13.3.3 Klassen Graphics

Klassen Graphics

final int HCENTER, LEFT, RIGHT horisontal justering

final int VCENTER, TOP, BOTTOM, BASELINE vertikal justering

void translate(int x, int y) sæt forskydning

int getTranslateX(), getTranslateY() aflæs forskydning

void setColor(int farve) sæt farve

void setColor(int r, int g, int b)

void setGrayScale(int farve)

int getColor() aflæs farve

int getRedComponent(), getGreenComponent(), getBlueComponent(), getGrayScale()

int getDisplayColor(int farve) giver en farves faktiske tegnefarve på skærm

final int SOLID, DOTTED linjetyper (ubrudt eller stiplet linje)

void setStrokeStyle(int linjetype) sætter linjetypen

int getStrokeStyle() aflæser linjetype

void setFont(Font) skrifttypen

Font getFont()

void setClip(int x, int y, int br, int hø) sæt klipning (hvor der skal tegnes)

int getClipX(), getClipY(), getClipWidth(), getClipHeight()

void clipRect(int x, int y, int b, int h) tilføj klipnings−rektangel

void drawLine(int x, int y, int x2, int y2)

void fillRect(int x, int y, int bredde, int højde)

void drawRect(int x, int y, int bredde, int højde)

void drawRoundRect(int x, int y, int bredde, int højde, int buebredde, int buehøjde)

void fillRoundRect(int x, int y, int bredde, int højde, int buebredde, int buehøjde)

void fillArc(int x, int y, int bredde, int højde, int startvinkel, int buevinkel)

void drawArc(int x, int y, int bredde, int højde, int startvinkel, int buevinkel)

void drawString(String tekst, int x, int y, int ankerpunkt)

void drawSubstring(String tekst, int afsæt, int længde, int x, int y, int ankerpunkt)

void drawChar(char tekst, int x, int y, int ankerpunkt)

void drawChars(char[], tekst, int afsæt, int længde, int x, int y, int ankerpunkt)

void drawImage(Image billede, int x, int y, int ankerpunkt)

void drawRegion(Image bil, int xBil, int yBil, int br, int hø, int trans, int x, int y, int anker)

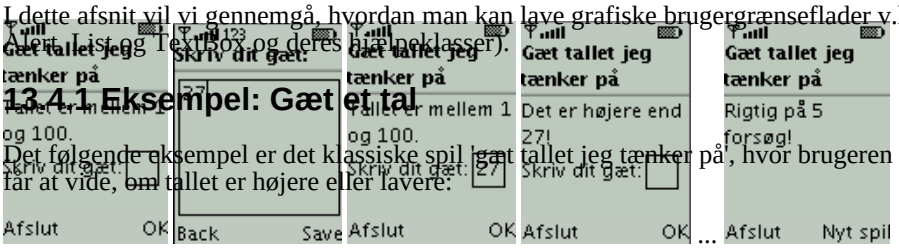
void copyArea(int xFra, int yFra, int br, int hø, int trans, int x, int y, int anker)

void fillTriangle(int x1, int y1, int x2, int y2, int x3, int y3)

void drawRGB(int[] farver, int afs, int lgd, int x, int y, int br, int hø, boolean gennemsigtig)

13.4 Grafiske standardkomponenter

I dette afsnit vil vi gennemgå, hvordan man kan lave grafiske brugergrænseflader v.hj.a. standardkomponenter (klasserne Form, Alert, List og TextBox og deres hjælpeklasser).



13.4.1 Eksempel: Gæt et tal

Det følgende eksempel er det klassiske spil gæt tallet jeg tænker på, hvor brugeren skal gætte et tal mellem 1 og 100 og hele tiden får at vide, om tallet er højere eller lavere:

Det fungerer ved at bruge en formular (Form) med med to elementer, en streng (StringItem) og et indtastningsfelt (TextField).

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;

public class GaetEtTalMidletMIDlet extends MIDlet implements CommandListener
{
  private Display display = Display.getDisplay(this);

  private Form f        = new Form("Gæt tallet jeg tænker på");
  private StringItem si = new StringItem(null, null);
  private TextField tf  = new TextField(null, "", 2, TextField.NUMERIC);

  private Command afslutCommand = new Command("Afslut", Command.SCREEN, 2);
  private Command okCommand = new Command("OK", Command.SCREEN, 1);
  private Command nytSpilCommand = new Command("Nyt spil", Command.SCREEN, 1);

  private Random tilf = new Random();
  private int tallet;
  private int forsøg;


  public GaetEtTalMidletMIDlet() {
    f.addCommand(afslutCommand);
    f.addCommand(okCommand);

    // tilføj klassen (implementerer CommandListener) som lytter til formularen
    f.setCommandListener(this);

    f.append(si);  // tilføj strengen til formularen
    f.append(tf);  // tilføj indtastningsfeltet
  }

  private void nytSpil() {
    tallet = Math.abs(tilf.nextInt()) % 100 + 1; // et tal mellem 1 og 100
    forsøg = 0;
    si.setText("Tallet er mellem 1 og 100.");
    tf.setLabel("Skriv dit gæt:");
  }

  public void startApp() {    // kaldes lige efter konstruktøren
    nytSpil();
    display.setCurrent(f);
  }

  public void pauseApp() { }

  public void destroyApp(boolean unconditional) { }

  /**
   * Reagér på kommandoerne ok, afslut og nyt spil.
   * Metode skal defineres fordi klassen implementerer CommandListener-interfacet
   * og den bliver kaldt fordi formularen har fået tilføjet klassen som lytter.
   * Ved afslut ryddes op og midletten giver besked på at den skal smides væk.
   */
  public void commandAction(Command c, Displayable s) {
    if (c == afslutCommand) {
      destroyApp(false);
      notifyDestroyed();
    }
```

```
    else if (c == okCommand) {
      int gæt = Integer.parseInt( tf.getString() );
      tf.setString("");
      forsøg++;
      if (gæt < tallet) si.setText("Det er højere end "+gæt+"!");
      else if (gæt > tallet) si.setText("Det er lavere end "+gæt+"!");
      else {
        si.setText("Rigtig på "+forsøg+" forsøg!");
        f.delete(1);      // Fjern indtastningsfeltet fra formularen
        f.removeCommand(okCommand);
        f.addCommand(nytSpilCommand);
      }
    }
    else if (c == nytSpilCommand) {
      f.removeCommand(nytSpilCommand);
      f.addCommand(okCommand);
      f.append(tf);       // Tilføj indtastningsfeltet til formularen igen
      nytSpil();
    }
  }
}
```

13.4.2 Kommandoer og hændelser i midletter

Hver af brugerens mulige handlinger repræsenteres af et Command−objekt. I 'Gæt et tal'−eksemplet ovenfor var de mulige handlinger 'Afslut', 'OK' og 'Nyt spil'.

Command−objektet har et navn (kort tekst, der vises på skærmen), en valgfri længere tekst, en type (der kan indvirke på hvordan kommandoen vises) og en prioritet.

Kommandoernes indbyrdes placering afgøres af deres prioritet. Afhængig af telefonens udformning kan det ske at den ikke kan vise alle kommandoerne. Første prioritet vises altid, mens kommandoer med anden− og tredieprioritet måske kun kan vises i en undermenu.

Klassen Command repræsenterer en mulig handling brugeren kan foretage

static int BACK, CANCEL, EXIT, HELP, ITEM, OK, SCREEN, STOP kommandotyper

Command(String navn, int type, int prioritet) konstruktør

Command(String navn, String langtNavn, int kommandoType, int prioritet) konstruktør

int getCommandType() giver typen af kommandoen

String getLabel() giver navnet på kommandoen

String getLongLabel() giver det lange navn på kommandoen

int getPriority() giver prioriteten i forhold til andre kommandoer (1 er højest)

Kommandoer skal føjes til et skærmbillede, hvorefter de vises nederst som mulige handlinger, brugeren kan gøre:

```
skærmbillede.addCommand(okCommand);
```

Man skal sætte en lytter (der skal implementere CommandListener) på skærmbilledet

```
skærmbillede.setCommandListener( lytter );
```

Interfacet CommandListener har metoden commandAction(), der bliver kaldt, hvis brugeren vælger handlingen, som Command−objektet repræsenterer.

13.4.3 TextBox

En TextBox giver brugeren mulighed for at indtaste og redigere tekst.

Klassen TextBox (arver fra Displayable)

TextBox(String titel,String tekst,int maxstørrelse,int type) opretter en TextBox, hvor type kan være:
ANY, EMAILADDR, NUMERIC, PHONENUMBER, URL, DECIMAL, PASSWORD,
SENSITIVE, UNEDITABLE, NON_PREDICTIVE,
INITIAL_CAPS_WORD, INITIAL_CAPS_SENTENCE

String getString() giver TextBoxens indhold som en streng

void setString(String nytIndhold) sætter indholdet

int getChars(char[]) giver TextBoxens indhold som et array af char

void setChars(char[] nytIndhold, int afs, int lgd) erstatter teksten med indholdet nytIndhold

void insert(String tekst, int position) sætter en tekst ind lige før den angivne position

void insert(char[] data, int afs, int lgd, int position) ditto.

void delete(int afs, int lgd) sletter *lgd* tegn startende fra *afs*

int getMaxSize() giver maksimale antal tegn, der er plads til

int setMaxSize(int) sætter det maksimale antal tegn

int size() giver antallet af tegn, der er i tekstboksen lige nu

int getCaretPosition() returnerer markørens aktuelle position

void setConstraints(int) sætter type (ANY, EMAILADDR, NUMERIC, ...)

int getConstraints() giver den aktuelle type

void setInitialInputMode(String måde) vink til inputmåde (f.eks. store bogstaver)

13.4.4 Alert

En Alert er en klasse, der viser data til brugeren, og venter et bestemt tidsrum, før det næste skærmbillede vises. Et Alert−skærmbillede kan vise tekst og billede og kan modtage brugerinput som andre skærmbilleder.

Klassen Alert (arver fra Displayable)

Alert(String titel) opretter en Alert med den givne titel
Alert(String titel,String tekst,Image billede,AlertType alerttype) ditto

static Command DISMISS_COMMAND kommando, der angiver, at brugeren har lukket alerten

int getDefaultTimeout() returnerer standardtiden, alerten vises

void setTimeout(int tid i millisek eller FOREVER) sætter, hvor længe alerten vises

int getTimeout() returnerer, hvor længe alerten vises

void setType(AlertType) sætter typen af alerten

AlertType getType() returnerer typen af alerten (typen kan være `INFO`, `WARNING`, `ERROR`, `ALARM`, og `CONFIRMATION`)

void setString(String) sætter teksten

String getString() returnerer teksten i alerten

void setImage(Image) sætter alerten til også at vise et billedet

Image getImage() returnerer et evt. tidligere sat billede

void setIndicator(Gauge) sætter alerten til også at vise en "progress bar"

Gauge getIndicator() returnerer en evt. tidligere sat "progress bar"

13.4.5 List

En List er et skærmbillede med en liste af valgmuligheder, som brugeren kan navigere op og ned i.

Lister kan fungere på 3 måder:

- EXCLUSIVE: Listens elementer vises som radioknapper (hvor højst ét element kan være valgt ad gangen).
- MULTIPLE: Listens elementer vises som afkrydsningsfelter (flere elementer kan være valgt ad gangen).
- IMPLICIT: Listens elementer vises som en valgliste. List.IMPLICIT er speciel ved, at den har en foruddefineret Command, som hedder SELECT_COMMAND.

Programmøren kan selv, vælge hvilke handlinger brugeren kan lave med hvert element, ved at definere nogle Command−objekter (se afsnit 13.4.2) og tilføje dem med addCommand(Command c) på listen. Derefter skal metoden setCommandListener(CommandListener lytter) kaldes med en lytter, der skal håndtere, hvis brugeren vælger at udføre en handling på et af elementerne i listen.

Klassen List (arver fra Displayable)

static int EXCLUSIVE, MULTIPLE, IMPLICIT listens type

static Command SELECT_COMMAND standardkommando for IMPLICIT−lister

List(String titel, int type) konstruktør til ny liste

List(String titel ,int type,String[] tekster, Image[] billeder) ny liste med tekst og billeder

String getString(int elementnr) giver strengen, der står på *elementnr* i listen

Image getImage(int elementnr) giver billedet på plads nr. *elementnr*

void set(int elementnr, String tekst, Image billede) sætter indholdet af listen på plads nr. *elementnr*

int append(String tekst, Image billede) tilføjer et nyt element i enden af listen

void insert(int elementnr, String tekst, Image billede) indsætter et nyt element lige før *elementnr*

void delete(int elementnr) sletter elementet på elementnr

void deleteAll() sletter alle elementer i listen

int size() returnerer antallet af elementer

void setFitPolicy(int politik) bestemmer, hvordan elementerne organiseres.
mulighederne er: Choice.TEXT_WRAP_DEFAULT, TEXT_WRAP_ON og TEXT_WRAP_OFF

int getFitPolicy() giver den aktuelle organisering af elementer

void setFont(int, Font) sætter skrifttypen

Font getFont(int) giver den aktuelle skrifttype

boolean isSelected(int) returnerer sand, hvis elementet er valgt

int getSelectedIndex() returnerer index for det valgte element

int getSelectedFlags(boolean[] valgte) giver antal valgte elementer og sætter valg.
(arrayet skal mindst have et antal pladser svarende til antallet af elementer i listen)

void setSelectedIndex(int elementnr, boolean valgt) sæt elementnr til at være valgt

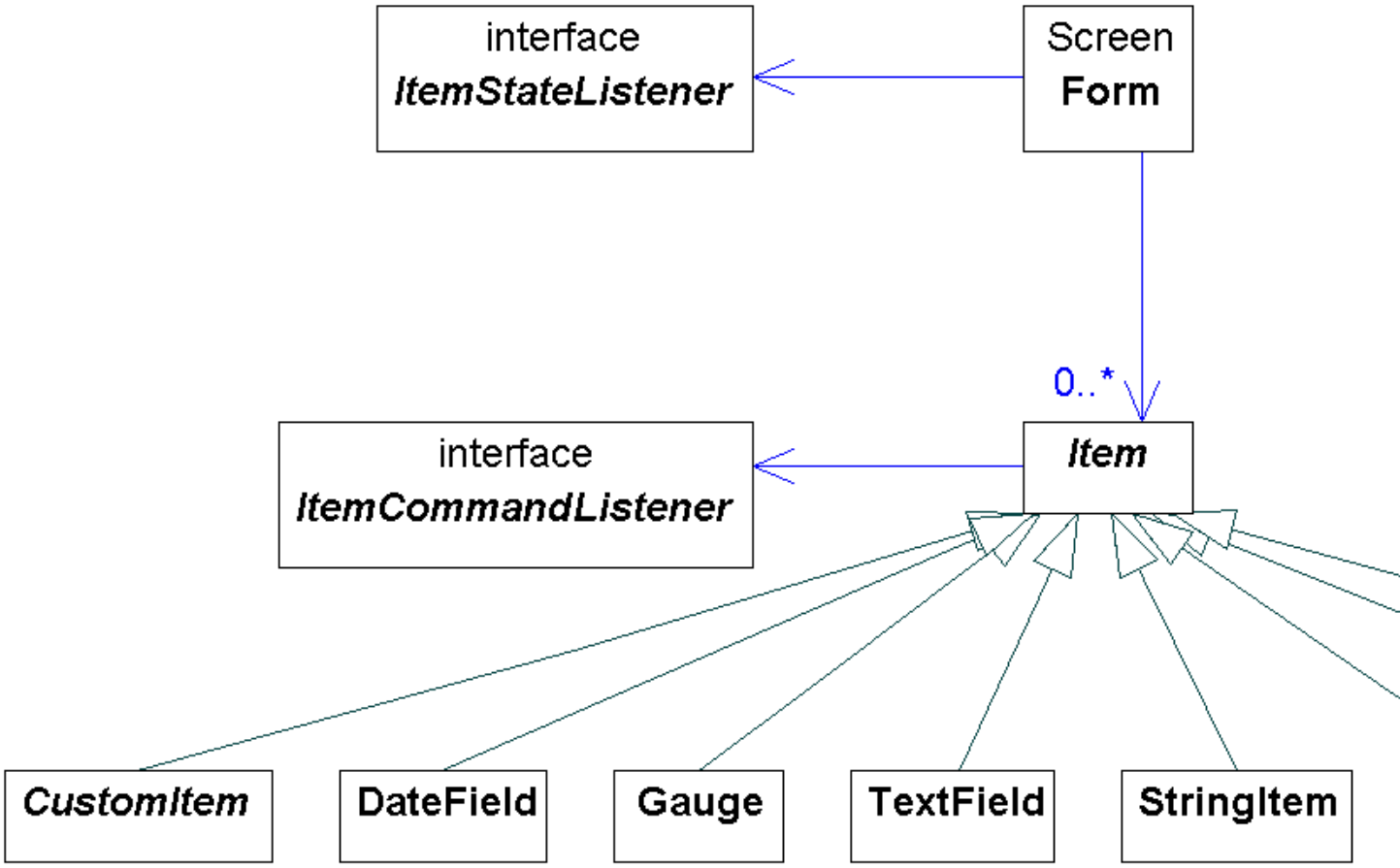
void setSelectedFlags(boolean[] valgteelementer) sætter de valgte elementer i listen

void setSelectCommand(Command) tilsidesætter List.SELECT_COMMAND med en anden kommando (kun for type=List.IMPLICIT)

13.4.6 Form

Med en Form kan man sammensætte billeder, tekstfelter og lister, som man lyster.

En Form er en indtastningsformular, der kan indeholde forskellige felter: (Item−objekter), såsom: ImageItem, StringItem, CustomItem (som man kan arve fra og lave sine egne slags formular−elementer), ChoiceGroup, TextField, DateField, Gauge og Spacer.



En Form kan have et ItemStateListener−objekt tilknyttet, der lytter efter ændringer i formularens felter, og et ItemCommandListener, der lytter efter, om brugeren forsøger at udføre en kommando på et af formularens elementer.

Klassen Form (arver fra Displayable)

Form(String titel) opretter en ny Form med den angivne titel

Form(String,Item[]) opretter en ny Form med den angivne titel og Item−objekter

int append(Item) tilføjer et nyt Item til Form−objektet

int append(String) tilføjer en streng til formularen

int append(Image) tilføjer et billede til formularen

void insert(int itemnr, Item) Indsætter et Item−objekt på pladsen specificeret ved *Itemnr*

void delete(int itemnr) sletter Item−objektet på den angivne plads

void deleteAll() sletter alle Item−objekter i Formen

void set(int itemnr, Item nytItem) Udskifter Item på plads nr *itemnr* med det nye Item

Item get(int itemnr) returnerer elementet på plads nr. *itemnr*

void setItemStateListener(ItemStateListener l) sætter den nye lytter, hvorved den gamle slettes.

int size() returnerer antallet af Items i Formen

int getWidth() giver bredden i punkter af det areal, der er til rådighed i formularen

int getHeight() giver højden i punkter af det areal, der er til rådighed i formularen

13.5 Tilgængelige klasser fra J2SE

De følgende pakker og klasser fra Javas standardbibliotek (J2SE) er også tilgængelige for midletter.

13.5.1 Pakken java.lang

I pakken java.lang findes stort set alle de kendte klasser, man forventer fra J2SE.

Det er, ud over forskellige klasser til at håndtere undtagelser (nedarvinger fra Exception eller Throwable), klasserne:

- Object, Class, Thread, Runnable, System, Runtime og Math
- String og StringBuffer
- Klasserne, der repræsenterer de simple typer: Boolean, Short, Byte, Character, Long og Integer

13.5.2 Pakken java.util

I pakken java.util findes klasserne kendt fra J2SE (undtaget er dog klasserne til samlinger af data, Collections−klasserne, såsom List, ArrayList, Set, Map, ...):

- Vector, Hashtable, Stack og Enumeration
- Date, TimeZone og Calendar
- TimerTask og Timer
- Random

13.5.3 Pakken java.io

I pakken java.io findes de almindelige klasser til IO−håndtering, dog uden dem, der er beregnet til filhåndtering (da man ikke kan gemme filer på en mobiltelefon).

Disse er (excl. klasser til at håndtere undtagelser):

- InputStream, ByteArrayInputStream, DataInput og DataInputStream
- Reader, InputStreamReader
- OutputStream, ByteArrayOutputStream, DataOutput og DataOutputStream
- Writer, OutputStreamWriter, PrintStream

I afsnit 13.6.1 beskrives de ekstra IO−klasser til netværkskommunikation, som er tilgængelige på en mobiltelefon.

Har man brug for at gemme data i telefonen kan man bruge RMS−systemet, der er beskrevet i afsnit 13.7.

13.6 Netværkskommunikation

Midletter kan kommunikere over netværket med en lang række protokoller (præcist hvilke der understøttes afhænger af apparatet), hvoraf den mest udbredte er HTTP−protokollen.

13.6.1 Pakken javax.microedition.io

I pakken javax.microedition.io ligger en række klasser og interfaces til netværkskommunikation, der er specielt rettet mod små apparater som telefoner.

Helt central er klassen Connector, gennem hvilken man kan åbne en række forskellige slags forbindelser, f.eks. en HTTP−forbindelse:

```
HttpConnection c = (HttpConnection) Connector.open("http://javabog.dk");
```

Man får altså et Connection−objekt ud, som man derefter selv må sørge for at typekonvertere til den rigtige slags forbindelse.

Klassen Connector − alle metoderne kan kaste IOException

static int READ, WRITE, READ_WRITE læse/skrive−flag

static Connection open(String url) åbner forbindelse

static Connection open(String url, int læsSkriv) do, med læse/skrive−flag

static Connection open(String url, int læsSkriv, boolean udløb) do, men tjekker for udløbstid

static DataInputStream openDataInputStream(String url)

static DataOutputStream openDataOutputStream(String url)

static InputStream openInputStream(String url)

static OutputStream openOutputStream(String url)

Når Connector.open() kaldes, er det parameteren url, der afgør, hvilken slags forbindelse der bliver returneret:

- HttpConnection − til HTTP−kommunikation
  f.eks.: Connector.open("http://javabog.dk")

- HttpsConnection − til sikker HTTP−kommunikation
  f.eks.: Connector.open("https://javabog.dk")
- SocketConnection − til at åbne en TCP−forbindelse
  f.eks.: Connector.open("socket://host.com:79")
- ServerSocketConnection − til at lytte på en port efter indkommende forbindelser
  f.eks.: Connector.open("socket://:79")
- SecureConnection − til SSL−forbindelser (Secure Socket Layer)
  f.eks.: Connector.open("ssl://maskinnavn.dk:79")
- DatagramConnection
  f.eks.: Connector.open("datagram://123.456.789.12:1234") for klientforbindelse
  eller: Connector.open("datagram://:1234") for serverforbindelse, der lytter på en port
- CommConnection − til eventuelle serielle porte
  f.eks.: Connector.open("comm:1")

Connector opretter forbindelser dynamisk ved at slå en protokolklasse op, hvis navn består af navnet på platformen, midletten køres fra, og protokolnavnet (fremfindes ud fra URL'en).

Derudover kan man også angive, om man ønsker at læse, skrive eller både læse og skrive til serveren i open()−metoden:

```
HttpConnection http = (HttpConnection) Connector.open(URL, Connector.READ_WRITE);
```

Når man har oprettet forbindelsen, kan man åbne en DataOutputStream. Den tillader, at man skriver en tekststreng direkte til den ved hjælp af metoden writeChars():

```
DataOutputStream out = http.openDataOutputStream();
out.writeChars("En tekst");
```

Man kunne også få en binær OutputStream og skrive til den, men så ville man i dette tilfælde være nødt til at pakke den ind i en PrintWriter(), som konverterer tekst til binært format.

13.6.2 Eksempel: Kommunikation med webserver

Nedenstående eksempel illustrerer, hvordan man kan sende og modtage tekst fra en server.

Brugeren kommunikerer med serveren ved at skrive en tekst, som bliver sendt til serveren. På serveren aktiveres JSP−filen modtag_besked.jsp (Java Server Pages, se afsnit 14.2):

```
<%@ page language = "java" import="java.io.*"%>
<%
  BufferedReader in = request.getReader();
  String besked = in.readLine(); // læs beskeden
  out.print(besked);      // ... og send den tilbage igen!
//<title>modtag besked</title>
%>
```

JSP−siden på serveren kvitterer for modtagelsen, ved at sende teksten tilbage igen.


```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class BeskedMidlet extends MIDlet
{
  private static BeskedMidlet instance;
  private SendBesked displayable = new SendBesked(this);

  public BeskedMidlet() {
    instance = this;
  }

  public void startApp() {
```

```
    Display.getDisplay(this).setCurrent(displayable);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }

  public static void quitApp() {
    instance.destroyApp(true);
    instance.notifyDestroyed();
    instance = null;
  }
}
```

Midletten bruger klassen SendBesked (en TextBox), der står for kommunikation mellem bruger og server:

```
import javax.microedition.lcdui.*;
import java.io.*;
import javax.microedition.io.*;

public class SendBesked extends TextBox implements CommandListener {
  private BeskedMidlet midletten;
  private Alert a;
  public SendBesked(BeskedMidlet midletten)  {
    super("Skriv din besked", "", 50, TextField.ANY);
    this.midletten=midletten;
    setCommandListener(this);
    addCommand(new Command("Exit", Command.EXIT, 1));
    addCommand(new Command("Send", Command.OK,1));
  }

  public void commandAction(Command command, Displayable displayable) {
    if(command.getCommandType() == Command.EXIT)
      BeskedMidlet.quitApp();
    else if(command.getCommandType() == Command.OK) try {
      sendData(getString(),"http://localhost:8080/modtag_besked.jsp");
    } catch(IOException e){System.out.println(e);}
  }

  /**
   * Afsender en tekststreng til den angivne URL.
   * Efter afsendelsen af data kaldes metoden læsData(), som læser svaret fra
   * serveren.
   * @param data den tekst der skal sendes til serveren
   * @param URL URLén til den servlet eller JSP-side, der håndterer anmodningen
   */
  private void sendData(String data, String URL)throws IOException {
    HttpConnection http = null;
    try {
      http = (HttpConnection) Connector.open(URL, Connector.READ_WRITE);
      http.setRequestMethod(HttpConnection.POST);
      DataOutputStream out = http.openDataOutputStream();
      out.writeChars(data);
      out.flush();
      læsData(http.openDataInputStream());
    }
    catch (Exception ex) {
      System.out.println(ex);
    }
    finally {
      http.close();  //husk altid at lukke forbindelsen!
    }
  }

 /**
  * Læser en tekststreng fra den angivne InputStream.
  * Herefter vises en alert med indholdet af tekststrengen
  * @param is Indeholder de data, der kommer fra serveren
  */
  private void læsData(InputStream is) throws IOException {
    StringBuffer besked = new StringBuffer();
    try {
      int bogstav = is.read();
      while ( bogstav != -1)
      {
        besked.append ((char) bogstav);
        bogstav = is.read();
      }
      setString("");
      a = new Alert("Besked afsendt!",besked.toString(),null,AlertType.INFO);
      a.setTimeout(5000); //Vis Alerten i 5 sek
      Display.getDisplay(midletten).setCurrent(a);
    }
    catch(Exception e) {
      System.out.println(e);
    } finally {
      if(is!=null) is.close();
    }
  }
```

```
}
```

13.7 Gemme data i telefonen

Vil man gemme data i mobiltelefonen, skal man bruge klassen RecordStore fra pakken javax.microedition.rms (Record Management System). Man kan tænke på det som en begrænset mulighed for at gemme filer i telefonen.

Her er et eksempel på brug:

```
// Åbn en 'fil' i telefonen. Opret den hvis den ikke allerede findes.
RecordStore database = RecordStore.openRecordStore("minFil", true);

// Skaf data i form af et array af byte
String strengDerSkalGemmes = "Hej Verden";
byte[] data =  strengDerSkalGemmes.getBytes();

// gem data
database.addRecord( data, 0, data.length );

// luk 'filen'
database.closeRecordStore();
```

Klassen RecordStore − næsten alle metoderne kan kaste RecordStoreException (ikke vist)

static String[] listRecordStores() giver navnene på gemte 'filer' for den aktuelle midlet

static void deleteRecordStore(String navn) sletter en 'fil'

static RecordStore openRecordStore(String navn, boolean opret) åbner og evt. opretter en 'fil'

static RecordStore openRecordStore(String navn, String leverandør, String midletnavn)

static RecordStore openRecordStore(String navn, boolean opret, int adgangsflag, boolean skrivbar)

static int AUTHMODE_PRIVATE, AUTHMODE_ANY adgangsflag

void setMode(int adgangsflag, boolean skrivbar)

String getName() giver navnet på 'filen'

int getVersion() giver versionsnummeret (ændres, når 'filen' ændres)

int getNumRecords() giver antallet af poster i 'filen' (Recordstore−objektet)

int getSize() giver antallet af byte, som 'filen' optager

int getSizeAvailable() giver, hvor meget ledig 'diskplads' der er i telefonen

long getLastModified() giver, hvornår 'filen' sidst var ændret i (i millisekunder)

int getNextRecordID() giver ID for den næste post

int addRecord(byte[] data, int afs, int lgd) tilføjer post med (del af) array af byte, giver et ID retur

void deleteRecord(int ID) sletter post med bestemt ID

int getRecordSize(int ID) giver størrelsen i byte for den givne post

int getRecord(int ID, byte[] data, int afs) gemmer post i et array af byte (afs bestemmer hvor)

byte[] getRecord(int ID) giver bytearray med en kopi af data fra den givne post

void setRecord(int ID, byte[] nyedata, int afs, int lgd) sætter dataene i den givne post

void addRecordListener(RecordListener) tilføjer et objekt, der lytter efter ændringer i 'filen'

void removeRecordListener(RecordListener) fjerner et lytter−objekt

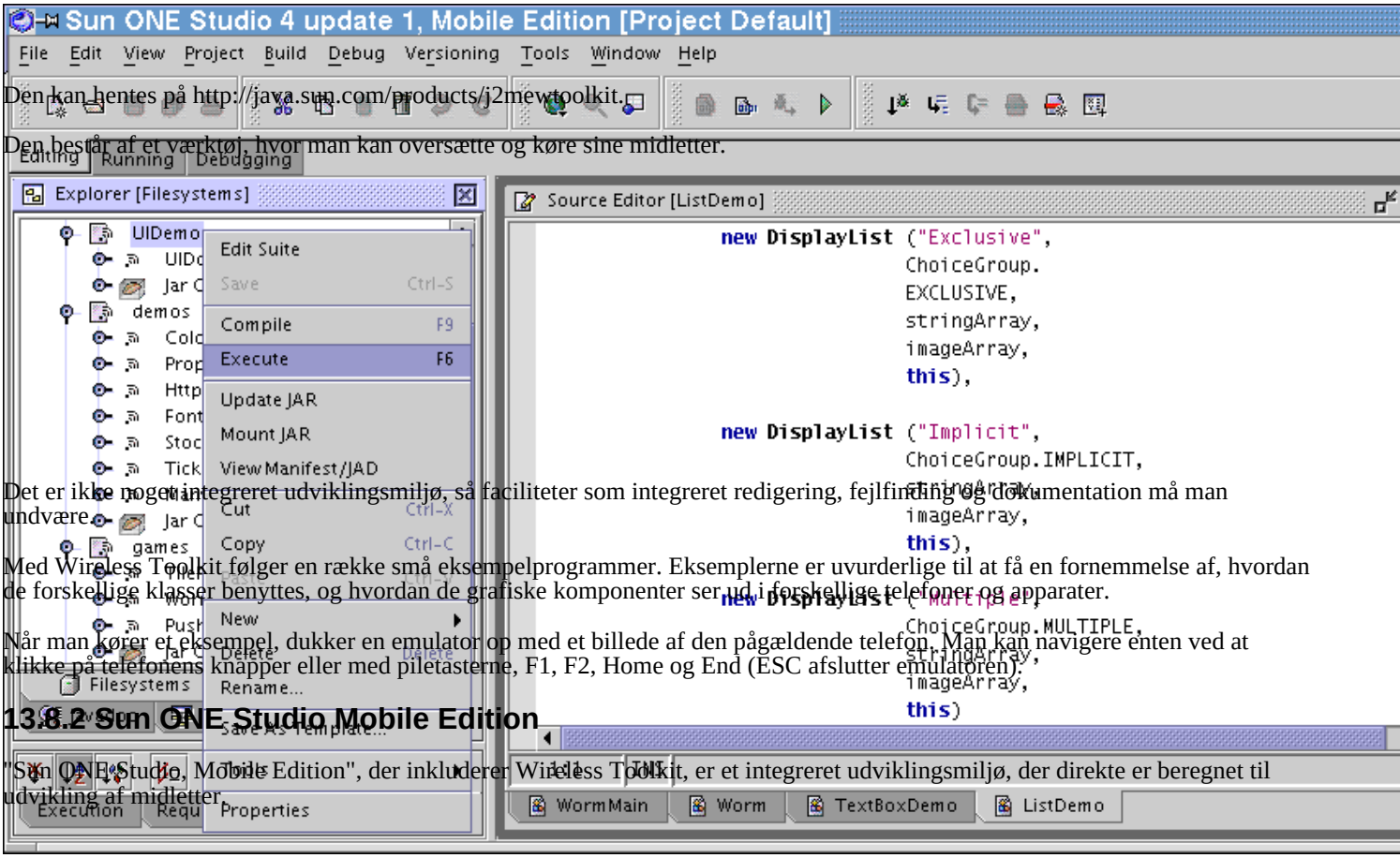
RecordEnumeration enumerateRecords(RecordFilter filter, RecordComparator, boolean synkron)

void closeRecordStore() lukker 'filen'

13.8 Udviklingsværktøjer til midletter

13.8.1 Wireless Toolkit

Sun har udgivet en referenceimplementation af, hvordan midletter skal fungere i forskellige mobiltelefoner og håndholdte apparater. Denne implementation hedder J2ME Wireless Toolkit og kører under Windows, Linux og Sun Solaris.



Den kan hentes på http://java.sun.com/products/j2mewtoolkit.

Den består af et værktøj, hvor man kan oversætte og køre sine midletter.

Det er ikke noget integreret udviklingsmiljø, så faciliteter som integreret redigering, fejlfinding og dokumentation må man undvære.

Med Wireless Toolkit følger en række små eksempelprogrammer. Eksemplerne er uvurderlige til at få en fornemmelse af, hvordan de forskellige klasser benyttes, og hvordan de grafiske komponenter ser ud i forskellige telefoner og apparater.

Når man kører et eksempel, dukker en emulator op med et billede af den pågældende telefon. Man kan navigere enten ved at klikke på telefonens knapper eller med piletasterne, F1, F2, Home og End (ESC afslutter emulatoren).

13.8.2 Sun ONE Studio Mobile Edition

"Sun ONE Studio, Mobile Edition", der inkluderer Wireless Toolkit, er et integreret udviklingsmiljø, der direkte er beregnet til udvikling af midletter.

er relativt nemt at installere, kører under Windows, Linux og Sun Solaris og kan hentes gratis på adressen: http://wwws.sun.com/software/sundev/jde.

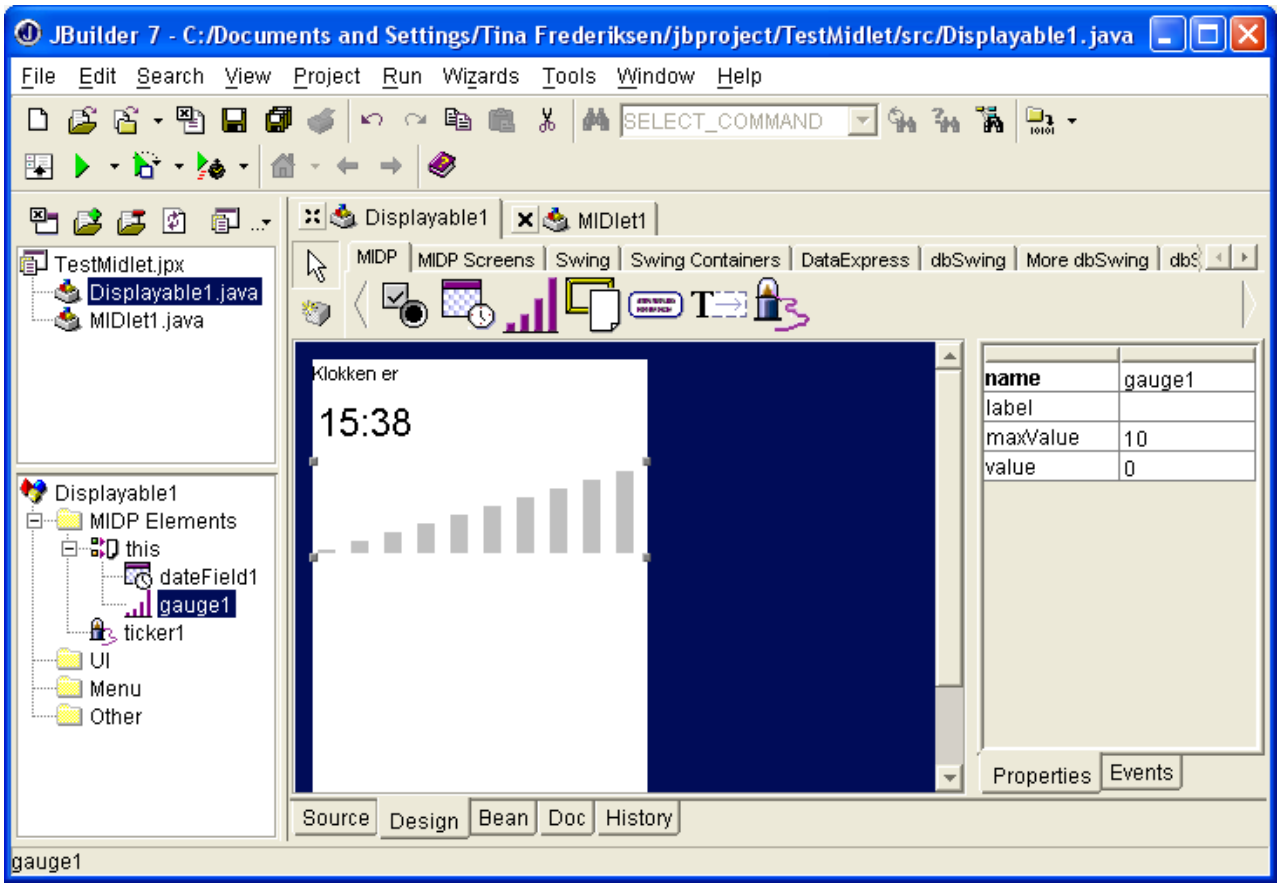
Når udviklingsmiljøet startes, dukker en række eksempler på midletter op, parat til at køre og med kildetekst, så man kan se hvordan forskellige ting programmeres. Højreklik f.eks. på den, der hedder 'UIDemo', og vælg 'Execute' (som vist på figuren ovenfor), og en telefon dukker op med midletten kørende inden i.

Når du vil lave dine egne midletter, skal du huske at have dem med i en "MIDlet suite", der også skal kende alle supplerende klasser.

13.8.3 Borland JBuilder MobileSet

Et andet udviklingsværktøj til midletter er JBuilder MobileSet. Det kan, som navnet antyder, integreres med JBuilder. JBuilder MobileSet inkluderer Wireless Toolkit.

Efter installationen har man alle de funktionaliteter, som er kendt fra JBuilder. Man kan umiddelbart oversætte og køre sine midletter.



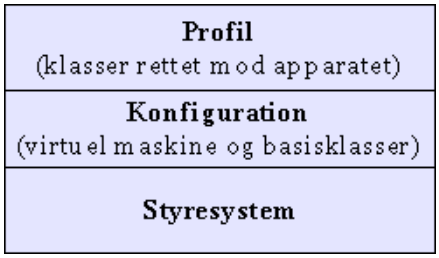
Sammen med JBuilder MobileSet ligger også en pdf−fil, som bl. a. giver en nærmere beskrivelse af, hvordan installationen foregår. Husk, at du skal ind og ændre i stien til JDK'et til der, hvor du har Wireless Toolkit liggende, for at få lov til at lave en midlet.

Når du så skal til at køre dine almindelige programmer igen, skal du sætte stien tilbage til dit almindelige JDK (gratisudgaven af JBuilder tillader kun ét JDK, så her er man nødt til at erstatte det almindelige JDK med J2MEs JDK).

13.9 Opbygningen af J2ME

J2ME består af en lagdelt struktur. Det betyder, at leverandøren af et indlejret system kan vælge den virtuelle maskine og de klasser, der skal være til rådighed for programmøren af en given slags apparat (f.eks. en vaskemaskine eller et fjernsyn), og stadig bruge mindst mulig hukommelse.

På figuren nedenfor er vist opbygningen af J2ME:



Oven på styresystemet haves en virtuel maskine af valgfri størrelse og en konfiguration med et udvalg af klasser fra standardbiblioteket. Oven på dette haves en profil, der afhænger af, hvilken type apparat man har med at gøre.

Den udgave af J2ME, der har afgjort størst interesse og udbredelse, er den, der kan køre i nyere mobiltelefoner. Dens konfiguration kaldes CLDC (Connected Limited Device Configuration) og har en profil der hedder MIDP (MIDlet Profile). Det er den vi har beskæftiget os med i dette kapitel.

En af idéerne i J2ME er, som med J2SE (normal Java) og J2EE (Java til serversystemer), at give mulighed for platformsuafhængighed, og programmer skrevet til MIDP−profilen på CLDC−konfigurationen kører da også umodificeret på

mange mobiltelefoner og en lang række andre lignende apparater.

13.9.1 Konfigurationer

En J2ME−konfiguration bestemmer den minimale platform for en bestemt kategori af apparater. Konfigurationen består af en virtuel maskine, et minimalt sæt af biblioteker, klasser og API'er (Application Programming Interface).

Der findes i øjeblikket 2 konfigurationer af J2ME:

- Connected Limited Device Configuration (CLDC) − til apparater med 60 − 512 Kb hukommelse.
- Connected Device Configuration (CDC) − til apparater med 512Kb − 2Mb hukommelse.

CDLC benytter sig af en virtuel maskine, der hedder KVM. KVM kan køres af små apparater, der er beregnet til at være koblet på et netværk som f.eks. mobiltelefoner og små PDA'er.

CDC benytter den almindelige JVM eller evt. CVM (en virtuel maskine i mellemstørrelse) og er beregnet til større PDA'er, der f.eks. kan lave ruteplanlægning eller lignende.

13.9.2 Profiler

En profil er en slags overbygning på konfigurationen, og den giver mulighed for, at man kan tilpasse udviklingsmiljøet til en bestemt type apparater. F.eks. findes der 'Handheld profile' og 'MID profile' (til midletter).

Profilen indeholder yderligere klasser og metoder, som man skal bruge for at udvikle til en bestemt slags apparater. Derudover indeholder profilen oplysninger om konfigurationen af J2ME−platformen.

Det betyder, at hver producent kan vælge at lave sin egen profil til en given slags apparat, som man kan bruge til at lave programmer, der udnytter netop denne slags apparaters specielle muligheder.

13.10 Yderligere læsning

For yderligere information om J2ME og midletter end den, der er givet her, henvises til:

- Dokumentationen til Wireless Toolkit, der kan læses på:
  http://wireless.java.sun.com.
  Her kan du også hente den nyeste udgave af Wireless Toolkit.
- Hvis du er mere interesseret i, hvordan man bruger CDC, så kig på:
  http://java.sun.com/products/cdc.

14 Servere (J2EE & EJB)

14.1 J2EE−platformens dele

14.2 Webservere (servletter og JSP)

14.2.1 JSP−sider

14.2.2 HTML−formularer

14.3 RMI − objekter over netværk

14.3.1 Principper i kald over et netværk

14.3.2 Fjerninterfacet til serverobjektet

14.3.3 Implementationen af serverobjektet

14.3.4 Klientsiden − brug af fjerninterfacet

14.4 EJB−bønner

14.4.1 Kildeteksten i en bønne

14.4.2 Eksempel: Veksler

14.4.3 EJB−Containerens information om Veksler

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

14.5 Bruge en EJB−bønne

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

14.5.1 JNDI (Java Naming and Directory Interface)

14.5.2 Brug af Veksler−bønnen

14.5.3 Brug af Veksler fra JSP

14.6 Typer af EJB−bønner

14.6.1 Sessionsbønner

14.6.2 Entitetsbønner

14.6.3 Meddelelses−drevne bønner

14.6.4 Tilstandsfuld sessionsbønne: Brugeradgang

14.6.5 Opgaver

14.7 Entitetsbønner

14.7.1 Eksempel på en entitetsbønne

14.7.2 Fremfindingsmetoder

14.7.3 Idriftsætte bønnen

14.7.4 Bruge bønnen

14.7.5 Fremfinde entitetsbønner i EJB 1.1

14.7.6 EJB 2.0 og EJB Query language (EJBQL)

14.7.7 Opgaver

14.8 Transaktioner

14.8.1 JTS (Java Transaction Service)

14.8.2 Transaktioner i JDBC (resumé)

14.8.3 Transaktioner i en EJB−container

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

14.8.4 Deklarativ transaktionsstyring af metodekald

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

---

15 Videregående OOP

I dette kapitel vil vi diskutere nogle af de begreber og tankegange, der kan være inspirerende, når man har tilegnet sig en grundlæggende viden og erfaring inden for objektorienteret programmering. Kapitlet er struktureret i emner, som kan læses nogenlunde uafhængigt af hinanden. Emnerne er beregnet som inspiration til eftertanke snarere end faste anvisninger, der altid bør følges.

Hvor det er relevant, beskrives nogle teknikker, der er specifikke i Java, og hvad man i stedet ville gøre i C++.

15.1 Arv

15.1.1 Holdninger til arv

Ved nedarving fra en klasse til en anden vil nedarvingen overtage ("arve") alle variabler og metoder fra superklassen. Der er dog flere mulige holdninger til, hvordan arv skal bruges.

Den kodenære, "amerikanske"[1]: *Arv bruges til at genbruge variabler og metoder*. Hvis to klasser har fælles variabler eller metoder, bør man lave en fælles superklasse, der tager vare på de ting, der er fælles.

Den analytiske, "skandinaviske": *Arv repræsenterer en er−en−relation*, og nedarving bør ske mellem klasser, når de begreber, som de står for, har en 'er−en'−relation til hinanden.

Den kodenære holdning, at arv bliver betragtet som en måde at spare kode på, er en 'nedefra−og−op'−strategi, hvor der først og fremmest programmeres og hvor klasserelationer som arv senere dukker frem som 'konsekvenser' af programmeringen. I de mere

udbredte programmeringssprog, som C og Pascal, var denne holdning dominerende i 1980'erne, lige da klasser og objekter var blevet indført i disse sprog.

Senere, da discipliner som objektorienteret analyse og design voksede frem, blev den analytiske 'oppefra−og−ned'−måde at anskue tingene på mere populær.

Ofte vil disse tankegange give de samme nedarvingsrelationer, så det kan være svært umiddelbart at forstå, hvilken forskel det skulle gøre.

15.1.2 Modstrid mellem holdningerne

For at tage et klassisk eksempel, hvor disse to tankegange kommer i indbyrdes modstrid, så betragt et tegneprogram med forskellige figurer: Punkter, linjer, rektangler, trekanter, firkanter, polygoner, cirkler, ovaler osv. I første omgang defineres følgende klasser:

Klasse

variabler

metoder

Punkt

int x, y

void tegn()
void flytTil(x,y)

Linje

int x, y, dx, dy

void tegn()
void flytTil(x,y)

Rektangel

int x, y, dx, dy
boolean udfyldt

void tegn()
void flytTil(x,y)
double areal()

Trekant

int x, y, dx, dy, dx2, dy2
boolean udfyldt

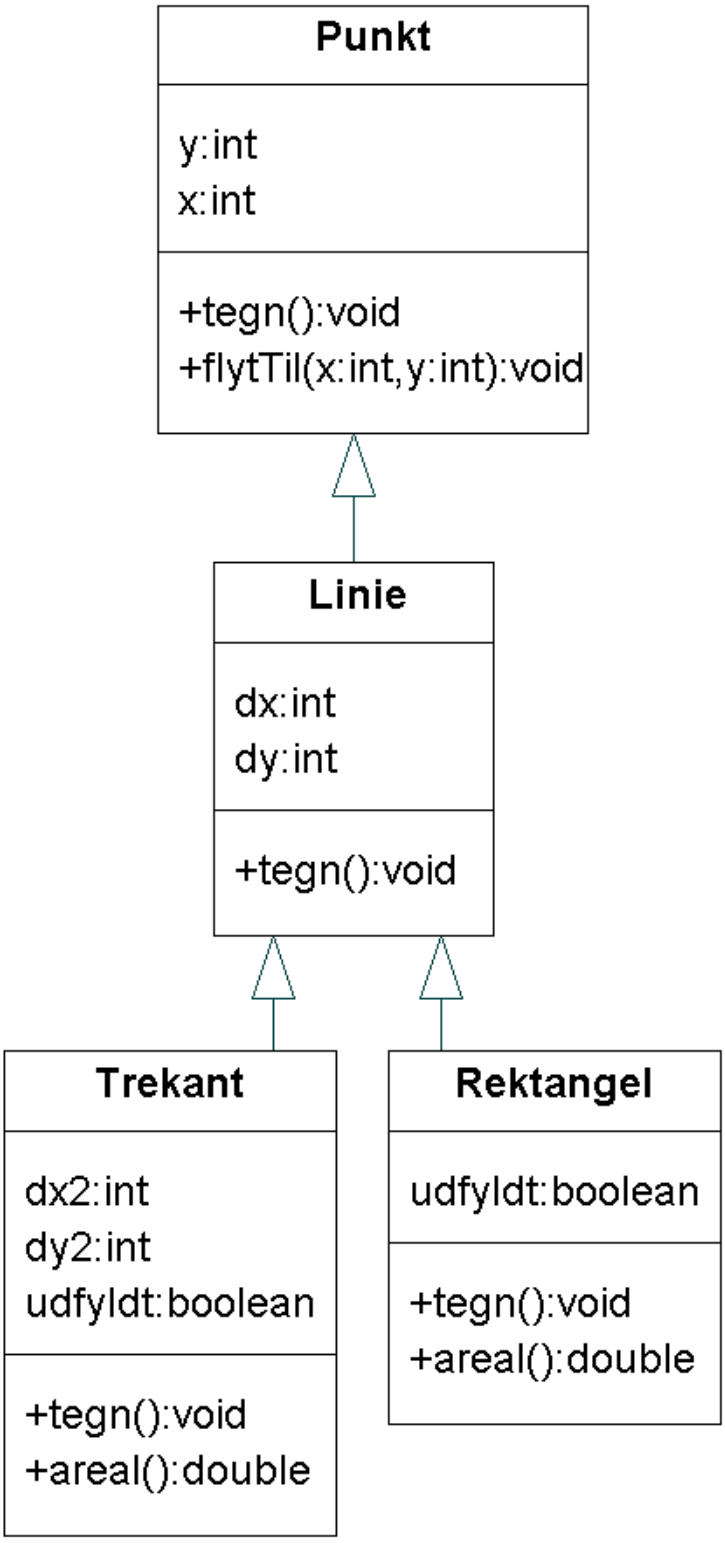
void tegn()
void flytTil(x,y)
double areal()

Det er oplagt, at disse klasser har meget til fælles, og at nedarving bør komme på tale. Men hvordan skal arvehierakiet være?

Arvehierakiet i den kodenære, "amerikanske" tankegang

I den kodenære tankegang er sagen klar:

- Punkt er superklassen,
- Linje arver fra Punkt, tilføjer to variabler og omdefinerer tegn(),
- Rektangel arver fra Linje og omdefinerer tegn().
- Trekant kan arve fra Linje og genbruge tegn() til at tegne den første streg (en anden mulighed var at arve fra Rektangel, så man får variablen 'udfyldt' med).



Arvehierakiet i den analytiske, "skandinaviske" tankegang

I den analytiske tankegang stiller sagen sig helt anderledes: Linje kan ikke arve fra Punkt, da linjer ikke er punkter!

Faktisk er der ingen af de ovenfor nævnte klasser, der kan arve fra hinanden. Man må opfinde nogle ekstra, mere abstrakte klasser som Figur og UdstraktFigur (en Figur med et areal):

Klasse

variabler

metoder

Figur

int x, y;

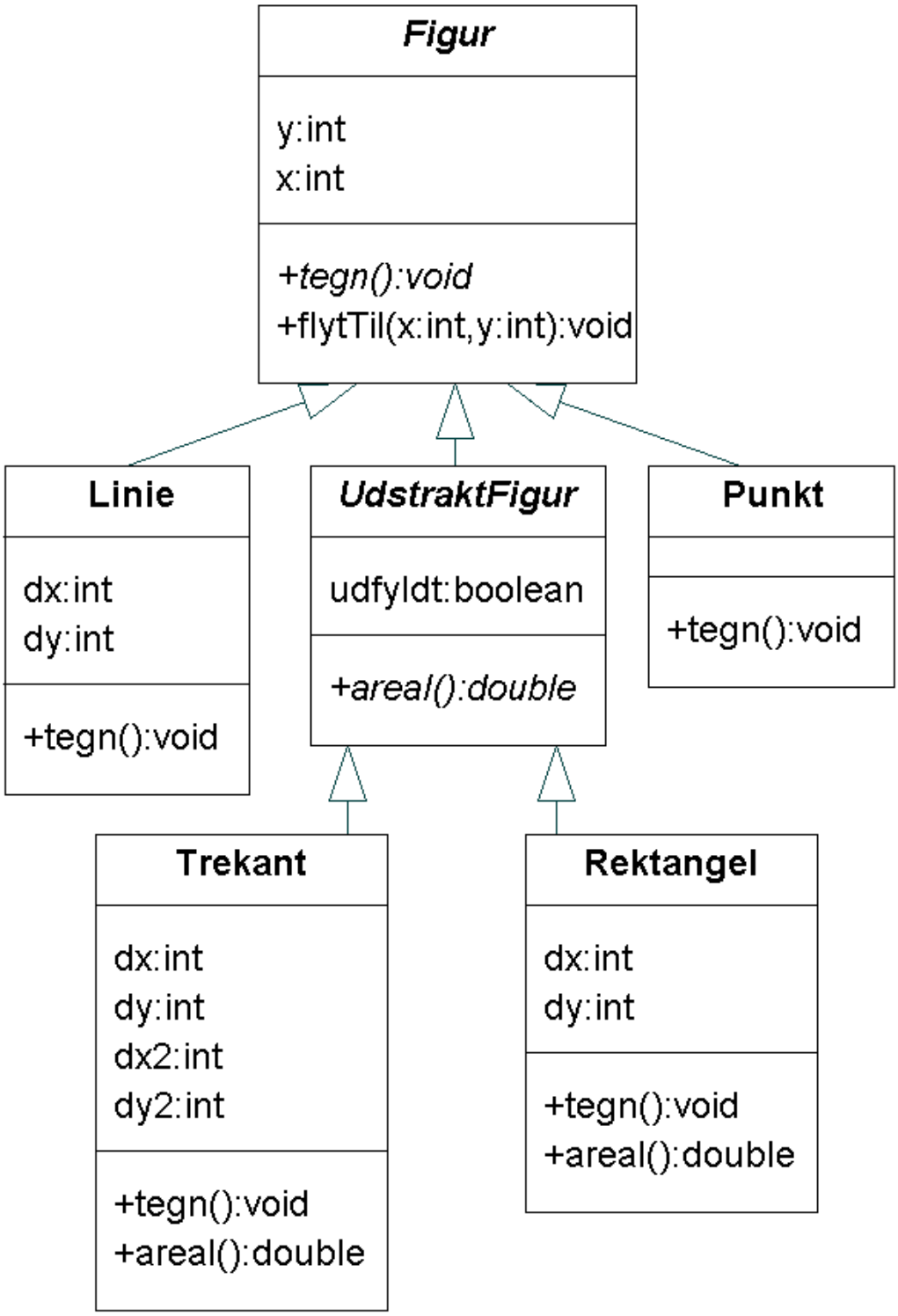
void tegn(); (evt. abstrakt)
void flytTil(x,y);

UdstraktFigur

int x, y;
boolean udfyldt;

void tegn(); (evt. abstrakt)
void flytTil(x,y);
double areal(); (evt. abstrakt)

Nu kan Figur være superklassen, Punkt er−en Figur og arver derfor fra Figur og definerer tegn(), UdstraktFigur er−en Figur med en ekstra variabel og metode, Rektangel er−en UdstraktFigur udvidet med variablerne dx og dy og ligeledes med Trekant, der er−en UdstraktFigur udvidet med variablerne dx, dy, dx2 og dy2.



15.1.3 Øvelse

1. Kig på klassediagrammet for den kodenære tankegang. Hvor skulle cirkler placeres? ellipser? firkanter? polygoner? Tegn dem ind.
2. Kig på klassediagrammet for den analytiske tankegang. Hvor skulle cirkler placeres? ellipser? firkanter? polygoner? Tegn dem ind.
3. Hvordan adskiller diagrammerne sig? Hvilken tankegang kan du bedst lide? Hvorfor? Hvor er der mest kode? Hvilket klassediagram tror du er nemmest at udvide senere?

15.1.4 Nedarvinger, hvor funktionaliteten ikke udvides, men begrænses

Et dilemma i den analytiske tankegang er, hvad man skal stille op, når der konceptuelt er tale om en specialisering (som altså burde medføre arv), men hvor funktionaliteten *begrænses* i stedet for (som normalt) at blive udvidet.

Tænk f.eks. på cirkler og ellipser. Det er klart, at Ellipse skal have to variabler til at angive radius (højde og bredde), mens Cirkel kun har brug for én radius.

På den anden side er der ingen tvivl om, at en cirkel er en slags ellipse (en hel rund ellipse), så konceptuelt gælder relationen Cirkel 'er−en' Ellipse, selvom Ellipse−klassen har flere data end Cirkel.

15.2 Delegering (i stedet for arv)

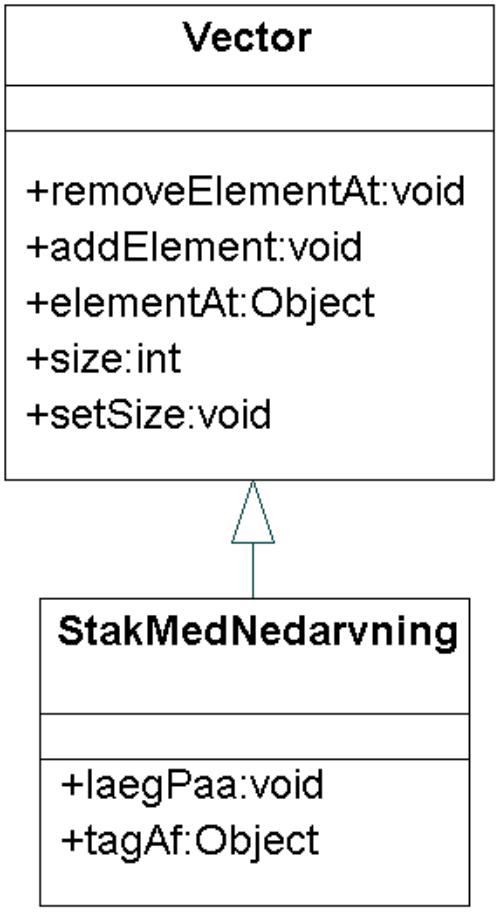
Delegering: At et objekt har et andet objekt, som det 'uddelegerer' nogle opgaver til

Ofte har man brug for at udvide og genbruge en klasses funktionalitet. Den sædvanlige måde at gøre det på er, som bekendt, ved at lave en nedarving, men det kan være, at det (af den ene eller anden grund) er mere hensigtsmæssigt at bruge delegering i stedet.

Delegering går ud på at skrive klassen med ekstra funktionalitet, sådan at den *bruger* den oprindelige klasse i stedet for at arve fra den.

15.2.1 At skjule nogle metoder

Eksempelvis kan det være, man ønsker at 'arve' en klasses data, og nogle men ikke alle, dens metoder. Så vil delegering sandsynligvis være mere hensigtsmæssigt end nedarving.



Arv kan ikke skjule metoder, men det kan delegering

Man vil lave datastrukturen 'stak', som er en liste, hvor man kun kan tilføje og fjerne elementer fra den ene ende (lægge på og tage af stakken). Man *kunne* arve fra Vector:

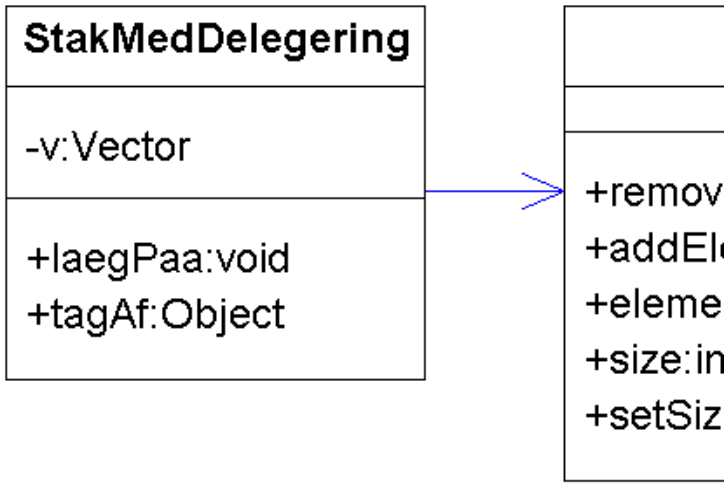
```
public class StakMedNedarving extends java.util.Vector
{
  public void lægPå(Object o)
  {
    addElement(o);
  }

  public Object tagAf()
  {
    Object o = lastElement();
    setSize( size() - 1 );
    return o;
  }
}
```

... men hvordan kan man nu være sikker på, at de, der bruger StakMedNedarving, bruger de nye metoder?

De kunne godt kalde elementAt() eller nogen af de andre metoder som StakMedNedarving arver fra Vector, og der er ikke nogen elegant måde at forhindre det på.

Det er mere hensigtsmæssigt at *delegere* opgaven med at holde styr på listen til Vector i stedet for at arve fra Vector:

```
public class StakMedDelegering
{
```




```
  private java.util.Vector v = new java.util.Vector();

  public void lægPå(Object o)
  {
    v.addElement(o);
  }

  public Object tagAf()
  {
    Object o = v.lastElement();
    v.setSize( v.size() - 1 );
    return o;
  }
}
```

Nu er vektoren indkapslet i stakken, og det er umuligt at kalde andre metoder end dem, der er defineret i StakMedDelegering.

Det er sjældent en god idé at arve fra 'værktøjsklasser' såsom Vector: Man har nemlig ikke mulighed for at forhindre brugeren af den nye klasse i at kalde nogle metoder, det ikke var meningen, han skulle kalde.

StakMedNedarving er et eksempel på den 'kodenære' tankegang beskrevet i afsnit 15.1.1 og illustrerer et af problemerne med denne måde at tænke på.

I C++ kan problemet med at skjule superklassens metoder løses ved at bruge såkaldt "privat nedarving", hvor kun underklassen har adgang til de nedarvede metoder (så det virker, som om metoderne er erklæret private i den nedarvende klasse).

Forhindre metodekald ved at kaste undtagelser

En anden mulighed kunne være at tilsidesætte alle de nedarvede metoder, der var uønskede, med nogle nye, der konsekvent kaster en undtagelse (eng.: exception), hvis de bliver kaldt. Dette er naturligvis ikke specielt elegant, men fra tid til anden kan man blive tvunget ud i den løsning (et andet eksempel ses i afsnit 17.1.3, Gøre data uforanderlige vha. Proxy).

```
public class StakMedNedarvingKastUndtagelser extends java.util.Vector
{
  public void lægPå(Object o)
```

```
  {
    super.addElement(o);
  }

  public Object tagAf()
  {
    Object o = lastElement();
    setSize( size() - 1 );
    return o;
  }

  public void addElement(Object obj) {
    throw new UnsupportedOperationException("Ikke tilladt på en stak");
  }

  public void setSize(int newSize)   {
    throw new UnsupportedOperationException("Ikke tilladt på en stak");
  }

  public Object elementAt(int index) {
    throw new UnsupportedOperationException("Ikke tilladt på en stak");
  }

  public void setElementAt(Object obj, int index) {
    throw new UnsupportedOperationException("Ikke tilladt på en stak");
  }

  public void insertElementAt(Object obj, int index) {
    throw new UnsupportedOperationException("Ikke tilladt på en stak");
  }
  // osv...
}
```

En stor ulempe ved denne teknik er, at fejl først fanges på kørselstidspunktet i stedet for under oversættertidspunktet.

En anden ulempe er, at hver gang superklassen får defineret en ny metode, der ikke skal kunne kaldes i nedarvingen, skal man huske at opdatere nedarvingen tilsvarende. Vector er således blevet udvidet med et antal metoder fra JDK1.1 til JDK1.2 (f.eks. get(), add() og de andre metoder i interfacet List − se kapitel 1, Samlinger af data), og koden ovenfor skulle derfor udvides med disse metoder, da man skiftede til JDK1.2 (i øvrigt er de fleste af Vectors metoder erklæret final og kan derfor faktisk overhovedet ikke tilsidesættes i en nedarving).

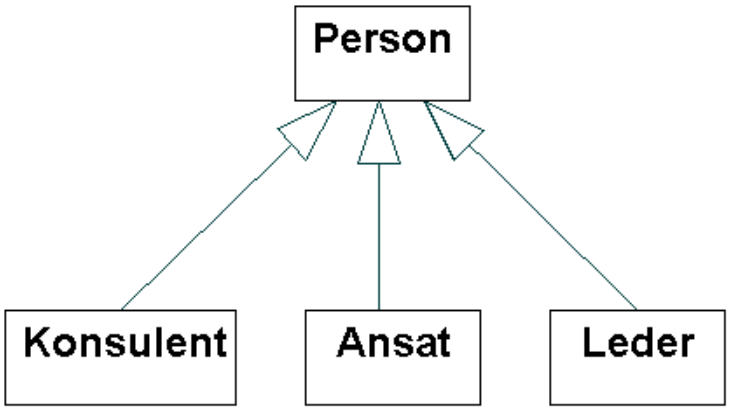
15.2.2 Modellering af roller

Et andet tilfælde, hvor delegering er en bedre idé end arv, er, hvor objekterne kan have flere roller eller skifte rolle under udførelsen af programmet.

> Arv er kun velegnet til "er−en"−relationer, der aldrig ændrer sig under kørslen

Lad os se på et eksempel inden for en virksomhed. Her er personer, der kan være ledere, ansatte og konsulenter (man kunne f.eks. forestille sig, at de har hver sin lønberegnings−metode, forskellige beføjelser, osv. ...).

Man kunne modellere rollerne med nedarving som vist i figuren herunder:



Rollerne modelleret med nedarving

Der er en fælles superklasse...

```
public class Person
{
  String fornavn;
  String efternavn;
  String cpr;

  public String hentFornavn() { return cpr; }
```

```
  // flere metoder og variabler...
}
```

... og nogle nedarvinger af Person, bl.a. lederen:

```
public class LederArv extends Person
{
  java.util.List ansatte;

  // opret en leder
  public LederArv() { }

  // opret en leder-rolle med en eksisteren person (ikke så elegant...)
  public LederArv(Person p)
  {
    this.fornavn   = p.fornavn;
    this.efternavn = p.efternavn;
    this.cpr       = p.cpr;
    // kopiér resten af Persons variabler her...
  }

  public java.util.List hentAnsatte() { return ansatte; }

  // flere metoder og variabler...
}
```

Problemerne med idéen om at bruge nedarving kan være at:

1. Personer kan i virkeligheden udfylde flere roller samtidigt.
2. Personer kan i virkeligheden skifte rolle.

Hvis objektet kan udfylde flere roller

Hvis kombinationer af rollerne kan opstå, sådan at f.eks. en person samtidig er konsulent (aflønningsmæssigt) og leder (m.h.t. Beføjelser), vil ingen af klasserne passe.

Skulle rollerne alligevel modelleres med roller, ville man blive nødsaget til at lave nye nedarvinger for kombinationsrollerne, f.eks. LederOgKonsulent og AnsatOgKonsulent.

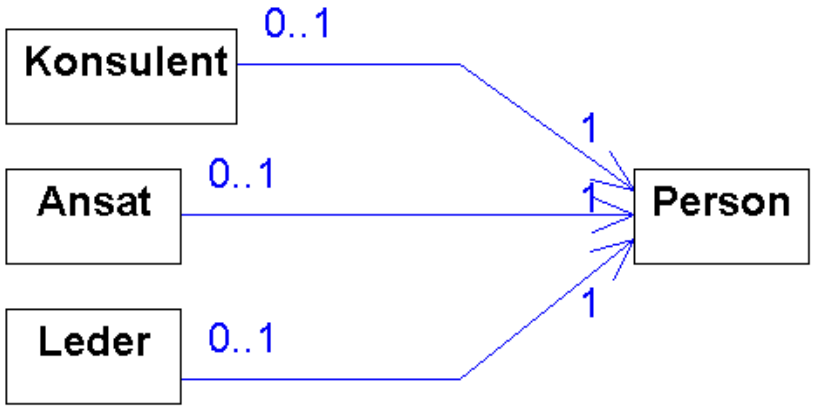
Hvis objektet kan skifte rolle

Værre ser det ud, hvis objektet kan skifte rolle (f.eks. fra Ansat til Leder). Da man ikke kan lave om på et objekts type (klasse), når det først er oprettet, er der intet at gøre andet end at oprette et objekt med den nye rolle og kopiere alle fælles data (dem, der stammer fra Person−klassen) fra det oprindelige objekt til det nye.

Skal vi lave en Ansat om til en Leder, kan det altså kun ske ved at oprette en ny Leder og smide det gamle Ansat−objekt væk. Vi må så kode Leder's konstruktør til at kopiere de fælles data (og i øvrigt håbe på, at der ikke er en anden del af programmet, der ændrer i data, efter at vi har kopieret dem).

Situationen med delegering i stedet for arv

I stedet for arv kunne vi modellere rollerne med delegering som vist nedenfor. Nu har Leder, Ansat og Konsulent en reference til et Person−objekt, og alle metodekald, der har med personen at gøre, delegeres videre til Person−objektet.



Rollerne modelleret med delegering

```
public class LederDelegering
{
  private Person person;
  private java.util.List ansatte;

  // opret en leder
  public LederDelegering()
```

```
  {
    person = new Person();
  }

  // opret en leder-rolle med en eksisteren person (mere elegant)
  public LederDelegering(Person p)
  {
    person = p;
  }

  public String hentFornavn() { return person.hentFornavn(); } // delegér

  public java.util.List hentAnsatte() { return ansatte; }

  // flere metoder og variabler...
}
```

Det er let at lade en Person have flere roller samtidig: Så deles et Leder−objekt og et Konsulent−objekt om et fælles Person−objekt.

Det er også let at lade en Person skifte rolle: Skal vi lave en Ansat om til en Leder, kan det ske ved at oprette en ny Leder og lægge det eksisterende Person−objekt ind i det.

Bemærk, at Leder, Ansat og Konsulent nu ikke mere er af typen Person. Dette er ikke noget problem, hvis man (som eksemplet forudsætter) har separate lister af ledere, ansatte og konsulenter. Har man brug for en fælles superklasse, kunne man definere et interface (eller en abstrakt klasse) PersonIF, som de alle implementerede, og som havde de metoder der var fælles for ledere, ansatte og konsulenter.

15.2.3 Resumé og konklusion

15.3 Specificér funktionalitet i et interface

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

15.3.1 Repetition af interfaces

I generel sprogbrug er et interface (da.: snitflade) en form for grænseflade, som man gør noget gennem. F.eks. er en grafisk brugergrænseflade de vinduer med knapper, indtastningsfelter og kontroller, som brugeren har til interaktion med programmet.

Vi minder om, at en klasse er definitionen af en type objekter. Her kunne man opdele i

1. *Grænsefladen* − hvordan objekterne kan bruges udefra.
   Dette udgøres af navnene[3] på metoderne, der kan ses udefra.
2. *Implementationen* − hvordan objekterne virker indeni.
   Dette udgøres af variabler og programkoden i metodekroppene.

Et 'interface' svarer til punkt 1): En definition af, hvordan objekter bruges udefra. Man kan sige, at et interface er en "halv" klasse.

> Et interface er en samling navne på metoder (uden krop)

Et interface kan implementeres af en klasse − det vil sige, at klassen definerer alle interfacets metoder sammen med programkoden, der beskriver, hvad der skal ske, når metoderne kaldes.

15.3.2 Eksempel: Stak

Grænsefladen til stakken (beskrevet i afsnit 15.2) kunne være defineret i et interface:

```
public interface Stak
{
  public void lægPå(Object o);
  public Object tagAf();
}
```

Der kan være mange klasser, der implementerer Stak, f.eks. de to fra forrige afsnit:

```
public class StakMedNedarving2 extends java.util.Vector implements Stak
{
  public void lægPå(Object o)
  {
    addElement(o);
  }

  public Object tagAf()
  {
    Object o = lastElement();
```

```
    setSize( size() - 1 );
    return o;
  }
}
```

og:

```
public class StakMedDelegering2 implements Stak
{
  private java.util.Vector v = new java.util.Vector();

  public void lægPå(Object o)
  {
    v.addElement(o);
  }

  public Object tagAf()
  {
    Object o = v.lastElement();
    v.setSize( v.size() - 1 );
    return o;
  }
}
```

Et enkelt sted i programmet er det nødvendigt at beslutte, hvilken implementation af Stak der anvendes, nemlig når objektet oprettes:

```
  Stak s = new StakMedNedarving2();
```

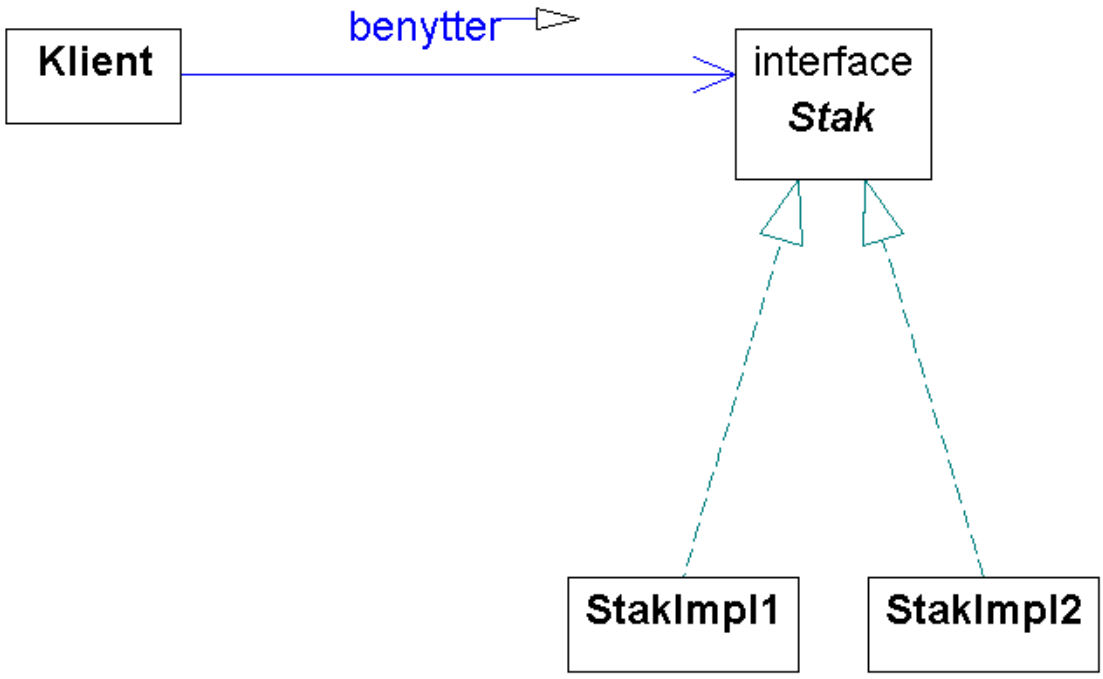
Resten af programmet arbejder bare med objekter af typen Stak *uden* at vide noget om, hvordan de er implementeret, f.eks.:

```
  ...
  System.out.print( s.tagAf() );
  System.out.print( s.tagAf() );
  fyldStakkenOp(s);
  ...
```

Også metoder, f.eks. fyldStakkenOp(), kan arbejde på ethvert objekt, der implementerer Stak−interfacet:

```
  public void fyldStakkenOp(Stak s)
  {
    s.lægPå("Hej");
    s.lægPå("med");
    s.lægPå("dig");
  }
```

Her er et klassediagram, der illustrerer, hvordan en klient (en klasse eller et program) benytter noget funktionalitet beskrevet i Stak. Metoder er ikke vist i diagrammet.



Pilen fra Klient til Stak viser, at klienten har en instans af Stak (en variabel af type Stak, der jo kan pege på alle objekter, der implementerer Stak−interfacet).

Stakimpl1 og Stakimpl2 er to klasser, der implementerer Stak−interfacet[4] (nedarvingspilen er tegnet med stiplet linje for at markere, at der er tale om implementation af interface, ikke nedarving), f.eks.: StakMedNedarving og StakMedDelegering.

15.3.3 Eksempel: Collections−klasserne

Funktionaliteten i datastrukturerne i Collections−klasserne er alle specificerede i interfaces, og brugeren opfordres til at henvise så lidt til implementationerne (som f.eks. ArrayList og LinkedList) som muligt og så meget til interfacene (som f.eks. List) som muligt:

```
  List l = new ArrayList();

  // ... resten af programmet arbejder bare med en List−variabel
  //     uden at kende til den konkrete implementation af listen
```

Det gør det lettere senere at udskifte implementationen af datastrukturen med en anden med samme funktionalitet, men anderledes indre struktur, f.eks. udskifte ArrayList med LinkedList:

```
  List l = new LinkedList();

  // ... resten af programmet arbejder bare med en List−variabel
  //     uden at kende til den konkrete implementation af listen
```

Se også afsnit 1.1.2, Interfacene til lister og mængder.

15.4 Konstanterklæringer i et interface

Det følgende er et lille fif i Java, der kan gøre programkoden mere læselig (i C++ har man mulighed for at erklære globale konstanter, så her er fiffet ikke relevant):

Ved at erklære konstanter, der skal bruges fra flere klasser i et interface, kan klasser, der har brug for disse konstanter, implementere interfacet og derefter bruge konstanterne *uden* at skulle skrive et klassenavn foran.

Eksempelvis kunne vi definere et interface med nogle tilstande:

```
public interface Tilstandsliste
{
  int IKKE_STARTET = 0;
  int STARTER = 1;
  int I_GANG = 2;
  int STOPPER = 3;
  int STOPPET = 4;
}
```

Bemærk, at variabler i et interface automatisk erklæres public static final (klassevariabel, der kan aflæses af alle, men ikke ændres).

Nu kunne vi selvfølgelig bruge konstanterne, som man plejer, ved at skrive klassen foran:

```
public class BenytTilstande1
{
  public static void main(String[] args)
  {
    int tilstand = Tilstandsliste.IKKE_STARTET;

    while (tilstand != Tilstandsliste.STOPPET) {
      System.out.println("tilstand = "+tilstand);
      tilstand++;
    }
  }
}
```

... men i klasser, der implementerer Tilstandsliste, kan det skrives kortere:

```
public class BenytTilstande2 implements Tilstandsliste
{
  public static void main(String[] args)
  {
    int tilstand = IKKE_STARTET;

    while (tilstand != STOPPET) {
      System.out.println("tilstand = "+tilstand);
      tilstand++;
    }
  }
}
```

15.5 Markeringsinterface

Det følgende bliver brugt en del i Javas standardbibliotek og er derfor værd at nævne (de defineres sjældent uden for standardbiblioteket).

> Et markeringsinterface erklærer ingen metoder, men markerer et eller andet.
> En klasse mærkes ved at implementere markeringsinterfacet.

Et markeringsinterface (eng.: marker interface) er et interface uden nogen metoder (eller variabler). Det tjener kun til at mærke klasser, sådan at deres objekter kan genkendes og behandles særskilt af et eller andet system.

Vigtigste eksempel fra standardbiblioteket på et markeringsinterface er Serializable, der tjener til at markere, om en klasse må serialiseres.

Et andet er Cloneable (der markerer, om det er lovligt at kopiere et objekt ved at kalde dets clone()−metode).

Et tredje er Remote i pakken java.rmi, der markerer, om et objekt er et 'fjernobjekt' i RMI (og dermed, om det skal forblive på serveren, og klienten skal have en fjernreference til det, eller det skal serialiseres, og indholdet transporteres over til klienten, som da får en kopi af objektet).

15.6 Ansvarsområder

En grundidé inden for objektorienteret design er, at hvert objekt skal have et eller flere ansvarsområder.

Ansvarsområder for et objekt kan være:

- At oprette nye objekter eller udføre en beregning
- At foretage en handling i andre objekter
- At kontrollere og koordinere aktiviteter i andre objekter
- At kende private data
- At kende relaterede objekter
- At kende ting, som objektet kan beregne

15.6.1 Eksempel på tildeling af ansvarsområder

Et eksempel på, hvordan man tildeler objekter ansvarsområder, er vist i neden for (fra bogen 'Objektorienteret programmering i Java', http://javabog.dk, kapitel 4):

```
public class Terning
{
  public int værdi;

  public Terning()
  {
    kast();
  }

  public void kast()
  {
    double tilfældigtTal = Math.random();
    værdi = (int) (tilfældigtTal * 6 + 1);
  }

  public String toString()
  {
    String svar = ""+værdi;
    return svar;
  }
}
```

Et terning−objekt har ansvaret for det, som vedrører den enkelte terning, dvs. ansvar for at vise den enkelte ternings værdi og at kaste den enkelte terning.

Et raflebæger−objekt har ansvar for alt det, som vedrører alle terningerne, dvs. oprettelsen af terningerne, kaste terningerne, kende summen af deres værdier og beskrive bægerets indhold.

Man kunne godt nøjes med én klasse og så have al koden i den, men en opdeling i forskellige klasser og dermed objekter med hvert sit ansvarsområde gør programmet lettere at overskue.

```
import java.util.*;

public class Raflebaeger
{
  public Vector terninger;            // Raflebaeger har en liste af terninger

  public Raflebaeger(int antalTerninger)
  {
    terninger = new Vector();
    for (int i=0;i<antalTerninger;i++)
    {
      Terning t;
      t = new Terning();
      tilføj(t);
    }
  }

  public void tilføj(Terning t)       // Læg en terning i bægeret
  {
    terninger.addElement(t);
  }
```

```
  public void ryst()                // Kast alle terningerne
  {
    for (int i=0;i<terninger.size();i++)
    {
      Terning t;
      t = (Terning) terninger.elementAt(i);
      t.kast();
    }
  }

  public int sum()                      // Summen af alle terningers værdier
  {
    int resultat;
    resultat=0;
    for (int i=0;i<terninger.size();i++)
    {
      Terning t;
      t = (Terning) terninger.elementAt(i);
      resultat = resultat + t.værdi;
    }
    return resultat;
  }

  public int antalDerViser(int værdi) // Antal terninger med en bestemt værdi
  {
    int resultat;
    resultat = 0;
    for (int i=0;i<terninger.size();i++)
    {
      Terning t;
      t = (Terning) terninger.elementAt(i);
      if (t.værdi==værdi)
      {
        resultat = resultat + 1;
      }
    }
    return resultat;
  }

  public String toString ()           // Beskriv bægerets indhold
  {
    // (vektorens toString() kalder toString() på hver terning)
    return terninger.toString();
  }
}
```

15.6.2 Ekspert

> Hvilke ansvarsområder skal de enkelte klasser have?

Tildel den klasse, der har den nødvendige information til at udføre handlingen, ansvaret for handlingen.

I raflebæger−eksemplet har Terning−objekterne ansvar for de ting, der har med den enkelte terning at gøre, og Raflebægeret har ansvar for de ting, der vedrører alle terningerne, da bægeret kender terningerne.

15.6.3 Skaber

> Hvem skal være ansvarlig for at oprette nye instanser (objekter) af en given klasse?

En klasse A skal være ansvarlig for at oprette nye instanser af en anden klasse B, hvis en eller flere af de følgende ting gælder:

- A indeholder objekter af typen B
- A består af objekter af typen B
- A kender de data, som B skal initialiseres med
- A bruger objekter af typen B meget

I raflebæger−eksemplet skal et objekt af typen Raflebaeger være ansvarlig for at oprette Terning−objekterne, da et raflebæger indeholder terninger, og raflebægeret bruger Terning−objekterne meget.

Kapitel 16, Skabende designmønstre, handler i øvrigt om oprettelse af objekter.

15.7 Lav kobling og høj kohæsion

Når man designer sit program, må man gøre sig klart, hvilke dele af programmet der vil være hyppige udvidelser og omstruktureringer i fremtiden.

Disse dele skal gerne være kendetegnet af lav kobling udadtil og høj kohæsion indadtil.

15.7.1 Lav kobling

> Hvordan opnås det, at klasserne i et program bliver lette at udskifte, og eventuelt genbruge?

Koblingen (bindingen) mellem klasser er et mål for, hvor afhængige klasserne er af hinanden. Programmer med høj kobling er kendetegnet ved, at ændringer i en klasse har stor indflydelse på de øvrige dele af programmet, og at koden i den enkelte klasse er svær at forstå isoleret set.

Ved at sørge for, at klasserne har så lille afhængighed af hinanden som muligt, kan overskueligheden af programmet forøges, og det bliver lettere at udskifte dele af programmet.

Det er især vigtigt, hvis man overvejer at genbruge noget af programkoden i en anden sammenhæng, at man sørger for, at bindingerne mellem disse to dele er så lav som mulig.

Lav kobling giver ofte flere fordele ud over genanvendelighed:

- Fleksibilitet − den anvendte del kan lettere ændres eller udskiftes med noget andet.
- Overskuelighed − delene kan forstås uafhængigt af hinanden.

Måder at få lav kobling

- Lad klasser have referencer til så få andre klasser som muligt.
- Specificér vigtig funktionalitet i interfaces, og lad variabler (og parametre) være af interfacets type for at undgå at lægge dig fast på en bestemt implementation (dette er beskrevet i afsnit 15.3).
- Tildel den enkelte klasse et let forståeligt og ensartet ansvarsområde (høj kohæsion − se afsnit 15.7.2).

Høj kobling

Det modsatte, høj kobling, betyder, at programdelene er afhængige af hinanden, så man har et fastlåst program, hvor delene ikke kan isoleres fra helheden og anvendes i en anden sammenhæng.

Høj kobling opstår typisk, hvis ansvarsområderne ikke bliver ordentligt klarlagt. Et kendetegn på udflydende ansvarsområder er, at der er mange klasser involveret i at varetage en bestemt opgave (spaghetti−programmering). Koden vil ofte være svært forståelig, fordi man skal sætte sig ind i alle de forskellige klasser, som tager del i udførelsen af opgaven.

Høj kobling kan også opstå ved meget genbrug (arv er meget høj kobling) eller hvis man har mange små metoder, der kalder hinanden meget.

15.7.2 Høj kohæsion

Høj kohæsion (sammenhæng − eng.: high cohesion) vil sige, at et objekt har et overskueligt og let forståeligt ansvarsområde, eller eventuelt flere ansvarsområder, der er tæt relaterede til hinanden. Jo flere forskellige ansvarsområder et objekt/en klasse har, jo sværere bliver det at genbruge objektet/klassen.

15.7.3 Indkapsling

Den nemmeste måde at sikre lav kobling er ved at gøre metoder og variabler, man ikke ønsker brugt udefra, utilgængelige udefra.

Vi minder om at:

- Variabler og metoder erklæret **public** altid er tilgængelige inden for og uden for klassen.
- Variabler og metoder erklæret **protected** er tilgængelige for alle klasser inden for samme pakke. Klasser i andre pakker kan kun få adgang, hvis de er nedarvinger.
- Skriver man **ingenting**, er det kun klasser i samme pakke, der har adgang til variablen eller metoden.
- Hvis en variabel eller metode er erklæret **private**, kan den kun benyttes inden for samme klasse (og kan derfor ikke tilsidesættes i en nedarving). Det er det mest restriktive.

Adgangen kan sættes på skemaform:

Adgang

public

protected

(ingenting)

private

i samme klasse

ja

ja

ja

ja

klasse i samme pakke

ja

ja

ja

nej

nedarving i en anden pakke

ja

ja

nej

nej

ej nedarving og i en anden pakke

ja

nej

nej

nej

Holder man sig inden for samme pakke, er der altså *ingen* forskel mellem public, protected og ingenting.

15.7.4 Indkapsling og pakker

Ud af de ovenstående regler kan man konkludere, at adgangskontrol ud over public/private først bliver interessant, når programmet spænder over flere pakker.

Så kan klasserne inden for samme pakke virke nært sammen (f.eks. ændre i hinandens interne variabler), mens adgangen fra klasserne i de andre pakker er begrænset.

> For at indkapsle en gruppe klasser, sådan at de kan tilgå hinandens metoder, mens disse metoder ikke er synlige udefra, er man nødt til at lægge dem i en pakke for sig

Eksempel

Kigger man nærmere på Funktion−klassen i afsnit 4.7.2, Repræsentation af funktioner, ser man, at dens forskellige nedarvinger (X, Konst, Sin etc.) ikke er erklæret public, men ingenting. Disse klasser er derfor kun tilgængelige for klasser, der ligger i den samme pakke (pakken vp.funktion), såsom klassen Funktionsfortolker beskrevet i afsnit 4.7.3, Fortolkning af strenge til funktioner. Samtidig er konstruktøren til Funktion erklæret på pakkeniveau, hvilket gør, at kun klasser i samme pakke kan arve fra Funktion.

Alt i alt er al logik omkring funktioner indkapslet i pakken vp.funktion.

Klassen Kurvetegner (afsnit 4.7.1, En komponent til at tegne kurver) ligger i pakken vp og har derfor ikke adgang til nedarvingerne. Den kan derfor ikke selv kombinere Funktion−objekter, men må bruge Funktionsfortolker.

15.8 GRASP

GRASP er nogle grundlæggende tommelfingerregler for, hvordan man skal designe sine klasser. GRASP står for: General Responsibility Assignment Software Patterns (og selve navnet GRASP angiver samtidig, at det er noget, der er lige til at tage og bruge).

De forrige afsnit har beskrevet en del af disse tommelfingerregler, mens andre bliver beskrevet senere i kapitlerne omkring designmønstre.

De 4 af i alt 9 GRASP−regler, der er beskrevet i denne bog, er:

- Ekspert (eng.: Information Expert) − beskrevet i afsnit 15.6.2.
- Skaber (eng.: Creator) − beskrevet i afsnit 15.6.3.
- Lav kobling − beskrevet i afsnit 15.7.1.
- Høj kohæsion (sammenhæng) − beskrevet i afsnit 15.7.2.

15.9 Introduktion til designmønstre

Et designmønster (eng.: design pattern) er en navngiven beskrivelse af, hvordan et givent problem kan løses og konsekvenserne af denne måde at løse problemet på.

Idéen er, at man måske senere i et andet program kan genbruge det samme designmønster til at løse et lignende problem (selve begrebet mønster angiver jo, at det er noget, som gentages).

Designmønstre hjælper med:

- at give idéer til godt design, sådan at når man står over for en problemstilling ikke skal "opfinde den dybe tallerken" igen og kan undgå de mest almindelige faldgruber.
- at give programmører en klar, fælles begrebsramme, der gør det lettere at forklare/dokumentere hvad man har lavet.

Ofte vil designmønstre give idéer til, hvordan man kan formindske graden af bindinger (kobling) mellem forskellige dele af programmet. Programkoden deles ofte op i to dele:

- Den del, der **anvender** en bestemt anden del. Denne del kaldes kaldes 'klienten'.

Den del, der **anvendes**. Denne del er måske generel anvendelig, måske endda en del af et standardbibliotek.

15.9.1 Designmønstre berørt i de følgende kapitler

I kapitel 16, Skabende designmønstre, behandles:

- Fabrikeringsmetode − en metode, der fabrikerer objekter for klienten (i stedet for at klienten selv opretter dem med new)
- Fabrik − et objekt med en fabrikeringsmetode
- Singleton − sikring af, at der kun eksisterer ét objekt af en bestemt slags
- Abstrakt Fabrik/Toolkit − er en Fabrik med nedarvinger, der sørger for objektoprettelsen. Hvilken nedarving der anvendes, bestemmes af en fabrikeringsmetode
- Bygmester − simplificerer oprettelsen af nogle relaterede objekter ved at oprette og konfigurere objekterne (evt. trinvist) for klienten
- Prototype − objekter oprettes ud fra eksisterende skabelon−objekter
- Objektpulje − genbrug de samme objekter igen og igen ved at huske dem i en pulje

I kapitel 17, Hyppigt anvendte designmønstre, behandles:

- Proxy − få metodekald til at gå gennem et mellem−objekt (proxyen), der modtager metodekald på vegne af (fungerer som en erstatning) for det rigtige objekt og kalder videre i det rigtige objekt
- Herunder Virtuel Proxy/Doven Initialisering − udskyde oprettelsen af det rigtige objekt til første gang, der er brug for det
- Adapter − et hjælpeobjekt, der får et objekt til at passe ind i et system ved at fungere som omformer mellem objektet og systemet
- Iterator − hjælper med at gennemløbe nogle data
- Facade − forenkler brugen af et sæt objekter ved at give en simplificeret grænseflade til dem
- Observatør/Lytter − at objekter kan 'abonnere' på, at en ting (hændelse) sker
- Dynamisk Binding − at understøtte "plugin"−klasser, der kan indlæses under kørslen og dermed udvide programmet, efter at det er skrevet

I kapitel 18, Andre designmønstre, behandles:

- Uforanderlig − objektet kan ikke ændres, når det først er oprettet
- Fluevægt − begræns antallet af objekter ved at sørge for, at der ikke bliver oprettet objekter med de samme data. I stedet er der mange referencer til de samme (unikke) objekter
- Filter − objekter, der "filtrerer" en strøm af data. Filtrene kan kombineres vilkårligt
- Lagdelt Initialisering − klienten opretter et objekt direkte, og dette objekt opretter eller fremskaffer det, der i virkeligheden skal bruges og videredelegerer det meste af arbejdet til det
- Komposit/Rekursiv Komposition − skabe et meget fleksibelt objekthierarki, ved at definere objekter, der kan indeholde andre objekter inkl. sin egen slags
- Kommando − registrér brugeren ændringer af data, sådan at de kan fortrydes igen.

I kapitel 19, Model−View−Controller−arkitekturen, behandles et designmønster beregnet til programmer med en brugergrænseflade, der anbefaler at man opdeler programmet i en datamodel, som repræsenterer data, forretningslogikken og de bagvedliggende beregninger, en præsentation af data over for brugeren, og en kontrol−del, der giver brugeren mulighed for at ændre i data.

1Ordene "amerikansk" og "skandinavisk" tankegang bruges bl.a. i et citat af Erik Ernst i artiklen "The Origins of Object Orientation", som kan findes på http://ootips.org/history.html

2Se f.eks. kapitel 1, Samlinger af data, hvor ArrayList og LinkedList let kan skiftes ud med hinanden, fordi deres fælles funktionalitet er defineret i interfacet List.

3Egentlig signaturen, dvs. metodenavn og antal og type af parametre.

4Det kunne f.eks. være StakMedNedarving og StakMedDelegering.

16 Skabende designmønstre

Det er en god idé at kigge i afsnit 1.1, Lister og mængder, kapitel 8, Databaser (JDBC) og afsnit 15.9, Introduktion til designmønstre, før man læser kapitlet.

16.1 Overordnet idé

Den overordnede idé i de skabende designmønstre er at give idéer til, hvordan man afkobler (dvs. mindsker graden af bindinger) mellem den del af programmet, som *bruger* nogle objekter (kaldet klienten), og den del af programmet, der bestemmer hvordan disse objekter bliver *oprettet*.

Kig på følgende eksempel, hvor klienten både opretter og bruger objektet af type Hjælp:

```
Hjælp h = new Hjælp();        // høj kobling − klient opretter et Hjælp−objekt
...
```

```
h.metode1();
h.metode2();
```

Ovenstående kan *ikke* udvides til, at klienten (uden at vide det) f.eks. benytter flere forskellige typer (nedarvinger) af Hjælp−objekter, eller at Hjælp−objektet oprettes med nogle andre parametre i konstruktøren. At oprette et objekt kræver jo, at man angiver præcist, hvilken klasse og hvilken konstruktør der skal anvendes.

Den eneste måde at gøre det på er rent faktisk at ændre i klientens kode − dvs. der er høj kobling (binding).

16.1.1 Fabrikeringsmetoder

En fabrikeringsmetode (eng.: Factory Method) er en metode, der opretter et objekt for klienten. Med en fabrikeringsmetode har man fjernet opgaven med at oprette objektet fra klienten, sådan at klienten i stedet kalder fabrikeringsmetoden.

Eksempelvis kunne Hjælp−klassen have defineret metoden opretHjælp():

```
Hjælp h = Hjælp.opretHjælp(); // fabrikeringsmetode leverer objekt til klienten

...

h.metode1();
h.metode2();
```

Fabrikeringsmetoder kan være en måde at opnå, at nogle objekter arver fra en fælles superklasse og anvendes ensartet udefra, men internt fungerer forskelligt (polymorfi) uden klientens vidende. Næsten alle de følgende designmønstre gør derfor brug af fabrikeringsmetoder.

Inde i metoden opretHjælp() oprettes objekterne muligvis nøjagtigt lige som før (med new Hjælp()), men det kunne også være at:

- det var en nedarving af Hjælp, der blev oprettet (polymorfi)
- det blev oprettet med nogle bestemte parametre i konstruktøren
- det samme objekt blev brugt af alle klienter (en Singleton, se afsnit 16.3)
- et eksisterende Hjælp−objekt blev genbrugt (en Objektpulje, se afsnit 16.7)

16.2 Fabrik

Problem: Klienten kan/skal ikke bestemme præcist, hvordan nogle objekter oprettes.

Løsning: Lad en Fabrik (eng.: Factory) med en fabrikeringsmetode varetage oprettelsen.

> En Fabrik er et objekt, der opretter objekter for klienten

En Fabrik er et objekt, der "fabrikerer" objekter for kalderen (klienten), og altså et objekt, der har en fabrikeringsmetode. Den kan lade klienten angive indhold og type. Klienten kender ikke til detaljerne omkring oprettelsen.

16.2.1 Eksempel: Image

Når man vil oprette et billede (af typen Image), sker det ikke med

```
Image i = new Image("billede.gif"); // forkert!!
```

Image−objekter kan være forskelligt repræsenteret afhængig af typen (f.eks. GIF, JPG eller PNG) og derudover bruges Image−objekter også til andre ting, bl.a. filtrerede billeder. Derudover kan et sort/hvid−billede med fordel repræsenteres med en bit per punkt, mens et GIF−billede kan have op til 256 farver og derfor med fordel repræsenteres med en byte per punkt. JPG−billeder kan have op til 16.7 millioner farver. Endelig kan et GIF−billede indeholde gennemsigtige punkter, der ikke tegnes på skærmen, mens et JPG−billede ikke kan være gennemsigtigt.

Det er bedre at lade systemet afgøre præcist, hvilken nedarving af Image der skal oprettes og hvordan. Man kalder derfor i stedet fabrikeringsmetoden getImage() på en grafisk komponent (arving fra java.awt.Component, f.eks. en applet eller et panel):

```
Image i = this.getImage("billede.gif"); // i en applet eller panel
```

Man siger, at komponenten fungerer som en *fabrik*, og at den *fabrikerer* Image−objekter.

Image er en abstrakt klasse, der repræsenterer et billede, der kan tegnes på skærmen. Ved hjælp af parameteren afgør metoden getImage(), hvordan billedet skal oprettes: Er parameteren et JPG−billede, vil getImage() vælge en nedarving af Image, der er velegnet til at repræsentere JPG−billeder. Er parameteren et GIF−billede, vil en nedarving velegnet til at repræsentere GIF−billeder blive valgt.

16.3 Singleton

Problem: Klienten må ikke have flere objekter af en bestemt type, men skal altid bruge det samme objekt.

Løsning: Programmér sådan, at der aldrig kan oprettes mere end ét eksemplar af det pågældende objekt.

> En Singleton er en klasse, som der må være én og kun én instans (objekt) af

Med en Singleton sørger man altså for, at der eksisterer højst ét objekt. Dette udføres som regel i koden ved at gøre konstruktøren privat og lade klassen selv håndtere oprettelsen af objektet (med en klassemetode, der kan fabrikere objektet).

En Singleton bruges ofte til at repræsentere resurser som i deres natur kun skal eksistere én gang. Det kan for eksempel være en forbindelse til en database, en resurse med begrænset adgang, en systemresurse (f.eks. afspilning af lyd), ...

16.3.1 Eksempel: java.lang.Runtime

Runtime er et eksempel på en Singleton.

Ethvert kørende javaprogram har et objekt af type Runtime, der giver programmet adgang til de omgivelser, det kører i. Man kan f.eks. kalde eksterne programmer, stoppe den virtuelle maskine, undersøge ledig hukommelse, køre garbage collector, indlæse flere klasser og et par andre ting.

Man får fat i Runtime−objektet ved at kalde metoden Runtime.getRuntime().

```
  Runtime rt = Runtime.getRuntime();

  // eksempler på brug at Runtime−objektet
  System.out.println("Hukommelse reserveret til Java: "+rt.totalMemory());
  System.out.println("Heraf ledigt: "+rt.freeMemory());
  rt.gc(); // kør garbage collector
  System.out.println("Nu ledigt: "+rt.freeMemory());
```

16.3.2 Eksempel: java.awt.Toolkit

Toolkit−klassen, der repræsenterer de konkrete implementationer af AWT (Abstract Window Toolkit), er et andet eksempel på en singleton. Det er klart at der skal eksistere ét og kun ét grafisk system samtidigt.

Man får fat i Toolkit−objektet med metoden Toolkit.getDefaultToolkit():

```
  Toolkit tk = Toolkit.getDefaultToolkit();
```

Fabrikeringsmetoden getDefaultToolkit() returnerer altid det samme objekt (ellers var det ikke en Singleton).

```
  System.out.println("Skærmstørrelse (punkter): " + tk.getScreenSize());
  tk.beep(); // computeren siger bip
  Image i = tk.getImage("billede.gif");
```

I den sidste linje fungerer Toolkit−objektet også som fabrik (der fabrikerer billeder).

16.3.3 Eksempel: Dataforbindelse

En forbindelse til en database kan med fordel implementeres som en Singleton, hvis man ønsker, at alle forespørgsler skal gå gennem det samme objekt. Det kunne være for at cache forespørgslerne eller for at sikre konsistens i data.

```
import java.util.*;

public class Dataforbindelse1
{
  /**
   * Instansen (forekomsten, objektet) af dataforbindelsen.
   * Dette er en klassevariabel, så den oprettes når klassen indlæses.
   */
  private static Dataforbindelse1 instans = new Dataforbindelse1();

  /**
   * Giver instansen af Dataforbindelse.
   */
  public static Dataforbindelse1 hentForbindelse()
  {
    return instans;
  }

  /**
   * Privat konstruktør sikrer at der ikke kan oprettes objekter udenfor klassen.
   */
  private Dataforbindelse1()
  {
    alle = new ArrayList();
  }

  private List alle;

  public void sletAlleData() { alle.clear(); }
  public void indsæt(Kunde k) { alle.add(k); }
  public List hentAlle() { return alle; }
}
```

Her er det sikret, at der ikke oprettes flere objekter ved at gøre konstruktøren privat.

Dataforbindelsen bruges fra ens program ved at bede om instansen i stedet for at oprette et nyt objekt:

```
public class BenytDataforbindelse1
{
  public static void main(String[] args)
  {
    Dataforbindelse1 dbf = Dataforbindelse1.hentForbindelse();
    dbf.indsæt( new Kunde("Kurt",1000) );
    // ...
  }
}
```

16.3.4 Singleton med sen initialisering

I ovenstående eksempel blev Dataforbindelse−objektet oprettet ved opstart af programmet, når klassen blev indlæst. Har singletonen brug for, at andre ting er initialiseret, før den oprettes, eller er det slet ikke sikkert, at singletonen overhovedet bliver brugt, kan det være hensigtsmæssigt at udskyde oprettelsen af objektet, indtil første gang der er brug for det.

```
import java.util.*;

public class Dataforbindelse2
{
  /**
   * Instansen (forekomsten, objektet) af dataforbindesen.
   * Dette er en klassevariabel, så den oprettes når klassen indlæses.
   */
  private static Dataforbindelse2 instans = null;

  /**
   * Giver instansen af Dataforbindelse2.
   * Instansen bliver oprettet første gang denne metode kaldes.
   */
  public static synchronized Dataforbindelse2 hentForbindelse()
  {
    if (instans != null) return instans;
    instans = new Dataforbindelse2();
    return instans;
  }

  /**
   * En privat konstruktør sikrer at der ikke kan oprettes objekter uden
   * for klassen.
   */
  private Dataforbindelse2()
  {
    alle = new ArrayList();
  }

  private List alle;

  public void sletAlleData() { alle.clear(); }
  public void indsæt(Kunde k) { alle.add(k); }
  public List hentAlle() { return alle; }
}
```

Bemærk, at da man kunne være så uheldig, at to tråde samtidig kalder hentForbindelse(), er det nødvendigt, at den er synkroniseret, så trådene kører en ad gangen i metoden.

At kalde en metode, der er synkroniseret, går lidt langsommere, da den virtuelle maskine skal lave noget arbejde for at sikre enkeltradet gennemløb af metoden.

16.3.5 Singleton uden privat konstruktør

Den mest almindelige måde at gennemtvinge en Singleton på er at lave konstruktøren privat. Den har dog det problem, at den forhindrer nedarvinger (f.eks. specialiseringer i forskellige slags datalagre). Det kunne dog løses ved at erklære konstruktøren protected og lægge klassen og dens nedarvinger i en separat pakke (se afsnit 15.7.4, Indkapsling og pakker).

En anden mulighed for at sikre, at der kun er ét objekt (der evt. er en nedarving), er at kaste en undtagelse fra konstruktøren (altså på køretidspunktet at nægte at oprette objektet), hvis objektet allerede eksisterer.

Nedenfor er vist en Dataforbindelse−klasse der, første gang hentForbindelse() bliver kaldt, beslutter om det skal være en Dataforbindelse eller nedarvingen SpecialiseretDataforbindelse, der skal bruges.

```
import java.util.*;

public class Dataforbindelse3
{
  /**
   * Instansen (forekomsten, objektet) af dataforbindelsen.
   * Dette er en klassevariabel, så den oprettes når klassen indlæses.
   */
  private static Dataforbindelse3 instans;

  /**
   * Giver instansen af Dataforbindelse.
   */
  public static Dataforbindelse3 hentForbindelse()
  {
```

```
    if (instans != null) return instans;

    if ( ...normal brug... ) return new Dataforbindelse3();
    else return new SpecialiseretDataforbindelse();
  }

  /**
   * Konstruktøren sikrer at der ikke kan oprettes flere instanser.
   * Da den ikke er erklæret public er det under alle omstændigheder kun
   * muligt at oprette objekter inden for samme pakke
   * @throws IllegalAccessException hvis instansen allerede eksisterer.
   */
  protected Dataforbindelse3()
  {
    if (instans != null) throw new IllegalAccessException("Objekt eksisterer");
    alle = new ArrayList();
    instans = this;
  }

  private List alle;

  public void sletAlleData() { alle.clear(); }
  public void indsæt(Kunde k) { alle.add(k); }
  public List hentAlle() { return alle; }
}
```

16.4 Abstrakt Fabrik

Problem: En Fabrik bliver uforholdsmæssigt kompliceret, fordi nogle omstændigheder har stor indflydelse på, hvordan oprettelsen skal foregå.

Løsning: Lav en Abstrakt Fabrik (eng.: Abstract Factory) med en nedarving (Fabrik) for hver omstændighed.

> En Abstrakt Fabrik er en Fabrik med nedarvinger, og disse nedarvinger sørger for den egentlige objektoprettelse

En Abstrakt Fabrik kaldes også undertiden Kit eller Toolkit.

Forestil dig, at du har en Fabrik, men at du står i den situation, at der er nogle ydre omstændigheder, der gør, at der nogen gange skal oprettes en anden slags objekter.

Man kunne selvfølgelig klare det med nogle if−sætninger i fabrikeringsmetod(erne), men det bliver bøvlet, hvis der på denne måde kommer et stort antal if−sætninger i hver fabrikeringsmetode, eller hvis der er et stort antal fabrikeringsmetoder.

En smartere løsning er at lade fabrik−klassen være abstrakt og lave en nedarving for hver omstændighed (som man ellers klarede med en if−sætning).

Som regel er det altid den samme nedarvede Fabrik, der skal bruges hver gang. Så bruger man ofte Singleton−designmønstret på superklassen ved at give den en metode til at fremskaffe nedarvingen.

16.4.1 Eksempel: AdresseFabrik

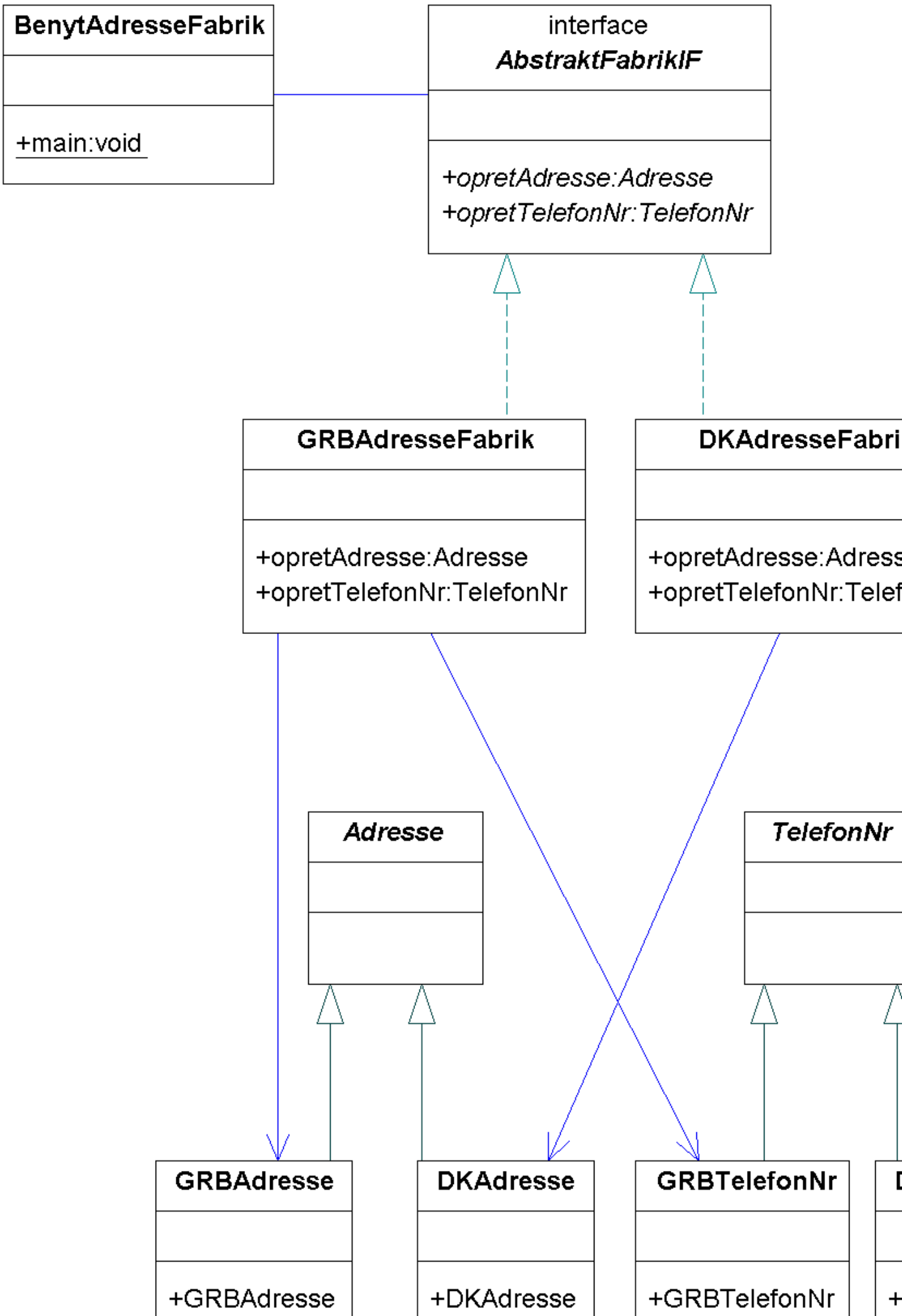
Forestil dig, at du har et program til at lagre internationale telefonnumre og adresser. For hvert land er udformningen af telefonnumre og adresser lidt forskellige. Man kunne lave en fabrik, som indeholdt en masse if−sætninger afhængigt af, hvilket land der var tale om, men det ville give en hel del kode i fabrik−klassen, og hver gang man skal tilføje adresse og telefonnummer for et nyt land, skal man ind og rette i fabrikken.

Så er det bedre at lave Fabrik−klasser, der arver fra en overordnet Fabrik, og som sørger for den specifikke oprettelse af adresse og telefonnummer klasser for hvert land, og så evt. lade den kode, som er generel for alle adresseklasserne, ligge i superklassen.

```
public class BenytAdresseFabrik {
  public static void main(String[] args)
  {
     AbstraktFabrikIF adresseFabrikRef;

     adresseFabrikRef = new DKAdresseFabrik();
     adresseFabrikRef.opretAdresse();
     adresseFabrikRef.opretTelefonNr();

     // i dette eksempel vælger klienten selv hvilken nedarving den vil bruge.
     adresseFabrikRef = new GRBAdresseFabrik();
     adresseFabrikRef.opretAdresse();
     adresseFabrikRef.opretTelefonNr();
  }
}
```

Den overordnede Fabrik skal der normalt ikke laves instanser af (den kunne altså erklæres abstrakt − deraf navnet Abstrakt Fabrik). I dette eksempel har vi derfor valgt at lade det være et interface:

```
public interface AbstraktFabrikIF
{
   public Adresse opretAdresse();
   public TelefonNr opretTelefonNr();
}
```


BenytAdresseFabrik
+main:void
interface
AbstraktFabrikIF
+opretAdresse:Adresse
+opretTelefonNr:TelefonNr
GRBAdresseFabrik
+opretAdresse:Adresse
+opretTelefonNr:TelefonNr
DKAdresseFabri
+opretAdresse:Adress
+opretTelefonNr:Telef
Adresse
TelefonNr
GRBAdresse
+GRBAdresse
DKAdresse
+DKAdresse
GRBTelefonNr
+GRBTelefonNr

```
public class DKAdresseFabrik implements AbstraktFabrikIF
{
   public Adresse opretAdresse() {
      return new DKAdresse();
   }

   public TelefonNr opretTelefonNr() {
      return new DKTelefonNr();
   }
}

public class GRBAdresseFabrik implements AbstraktFabrikIF
{
   public Adresse opretAdresse() {
      return new GRBAdresse();
   }
   public TelefonNr opretTelefonNr() {
      return new GRBTelefonNr();
   }
}
```

Adresse−klassen skulle have de metoder og variabler, der forventes at findes i alle adresser:

```
public abstract class Adresse
{
  // mangler: metoder/variabler der findes i alle adresser
}

public class DKAdresse extends Adresse
{
  public DKAdresse() {
     System.out.println("DKAdresse oprettet");
  }
}

public class GRBAdresse extends Adresse
{
  public GRBAdresse() {
     System.out.println("GRBAdresse oprettet");
  }
}
```

Ligeledes med telefonnumre

```
public abstract class TelefonNr
{
  // mangler: metoder/variabler der findes i alle telefonnumre
}

public class GRBTelefonNr extends TelefonNr
{
  public GRBTelefonNr() {
     System.out.println("GRBTelefonNr oprettet");
  }
}

public class DKTelefonNr extends TelefonNr
{
  public DKTelefonNr() {
     System.out.println("DKTelefonNr oprettet");
  }
}
```

16.4.2 Eksempel fra standardklasserne: AWT

Klassen Toolkit er et eksempel på en Abstrakt Fabrik.

Da AWT−komponenterne skal virke på flere platforme (Windows, Linux, Solaris, MacOS), fungerer de sådan, at de har en platformsspecifik del (et såkaldt *peer*) tilknyttet, der sørger for at tegne dem på skærmen.

F.eks. har klassen Button et ButtonPeer−objekt (beskrevet i pakken java.awt.peer), som det delegerer nogle opgaver ud til. Da disse peers er implementeret forskelligt fra platform til platform, er der tilsvarende nedarvinger af ButtonPeer for de forskellige platforme, f.eks. WindowsButtonPeer, LinuxToolkit, SolarisToolkit og MacOSToolkit.

Disse peer−objekter fabrikeres af klassen Toolkit, der har f.eks. metoden createButton til at oprette et ButtonPeer−objekt.

I stedet for, at Toolkit direkte kender til de forskellige styresystemer, er klassen erklæret abstrakt, og der er en nedarving af Toolkit for hvert styresystem (f.eks. WindowsToolkit, LinuxToolkit, SolarisToolkit og MacOSToolkit).

Metoden Toolkit.getDefaultToolkit() giver den konkrete nedarving, der passer til styresystemet.

16.5 Bygmester

Problem: Der skal oprettes flere objekter, som hænger sammen.

Eller problem: De nødvendige informationer til oprettelse af nogle objekter kommer ikke alle på én gang, eller det er uoverskueligt at angive alle informationerne i ét kald.

Løsning: Brug en Bygmester (eng.: Builder) til at specificere informationerne skridt for skridt og til sidst fabrikere objekterne.

> Simplificér oprettelsen af nogle relaterede objekter ved at lave en Bygmester−klasse, der opretter og konfigurerer objekterne for klienten

En Bygmester er en Fabrik, der, evt. trinvist, opretter et netværk af objekter. En Bygmester minder også om en Facade (se afsnit 17.4 om Facade), men i stedet for at være en simplificering af *brug* af nogle objekter er det en simplificering af *oprettelsen* af nogle objekter.

Ligesom en Fabrik kan en Bygmester lade klienter oprette objekter ved at angive indhold og type. Klienten kender ikke til detaljerne omkring oprettelsen.

16.5.1 Trinvis konstruktion

Bygmesteren kan have metoder til at konfigurere objektet skridt for skridt. Til sidst kaldes en fabrikeringsmetode på bygmesteren, som returnerer det færdige objekt.

Dette er forskelligt fra en Fabrik, hvor konstruktion sker med et enkelt metodekald.

Hvornår bruge trinvis konstruktion?

Trinvis oprettelse kan være berettiget i forskellige situationer:

- For at opnå større læsbarhed − i stedet for store indviklede konstruktører
- Oplysninger er ikke alle til stede når oprettelsen påbegyndes, f.eks.:
- På grund af måden, et filformat er organiseret på (ved indlæsning af data)
- Brugeren er ved at indtaste nogle data. Efterhånden som data indtastes gives de videre til Bygmesteren, der registrerer og validerer dem (f.eks. en aftale i en kalender).
- Beslutningen om, præcis hvilken klasse der skal oprettes, kan først tages, når alle oplysningerne er til stede (dette er f.eks. ikke muligt ved først at skabe objektet med en Fabrik og derefter lægge data ind i det resulterende objekt).

16.5.2 Eksempel på trinvis konstruktion: E−post

Forestil dig et system, der kan afsende e−post.

Uden en Bygmester skulle klienten kalde en konstruktør med alle parametrene. Konstruktøren kunne f.eks. se således ud:

```
public Meddelelse(
  Adresse afsender,
  Adresse[] modtagere,
  Date tidspunkt,
  String[] supplerendeData,
  String hovedtekst,
  Vedhaeftning[] vedhæftedeData
)
```

Så ville oprettelsen f.eks. se således ud:

```
Meddelelse m = new Meddelelse(
  new Adresse("nordfalk@mobilixnet.dk"),
  new Adresse[] { new Adresse("bo@hansen.dk"), new Adresse("hans@hansen.dk") }
  new Date(),
  null,
  "En hilsen fra Jacob",
  null
);
```

Med en Bygmester sker oprettelsen skridt for skridt, og det er umiddelbart nemmere at forstå, hvad der foregår:

```
Meddelelsesbygger mb = new Meddelelsesbygger(); // eller en fabrikeringsmetode

mb.sætAfsender("nordfalk@mobilixnet.dk");
mb.tilføjModtager("bo@hansen.dk");
mb.tilføjModtager("hans@hansen.dk")
mb.sætHovedtekst("En hilsen fra Jacob");

Meddelelse m = mb.byg();
```

16.5.3 Eksempel på netværk af objekter: Funktioner

I eksemplet afsnit 4.7.1, En komponent til at tegne kurver, har klassen Funktionsfortolker i afsnit 4.7.3 rollen som Bygmester, idet den opretter og konfigurerer Funktion−objekterne til at passe med en bestemt formel.

16.6 Prototype

Problem: Klienten ved ikke, hvad der skal oprettes, men kan dog angive et andet objekt, som ligner det, der skal oprettes.

Løsning: Brug det andet objekt som Prototype, og opret objektet ud fra prototypen.

> Tag et eksisterende objekt (prototypen), lav en kopi, og ret kopien til efter behov

Med Prototype kan man lade et objekt oprette en kopi af sig selv på bestilling udefra. Klienten, der bestiller oprettelsen, kan så arbejde med objektet og kopien uden at vide eksakt hvad objektet indeholder eller detaljerne i, hvordan objektet oprettes.

Prototype−designmønstret består altså i at oprette objekter ud fra et allerede eksisterende objekt, hvorefter kopien tilrettes sådan, at den passer til formålet.

Det kan også ske, at klienten beder et andet objekt (en Fabrik eller Bygmester) om at oprette objektet. Klienten angiver så, hvad der skal oprettes, og fabrikken finder så den rette Prototype, kopierer den og retter den til.

Prototype−designmønstret har flere fordele:

- Klienten behøver ikke at kende den præcise måde, objektet skal oprettes på.
- Der kan være data i objektet, som klienten ikke kender til.
- Klienten kan have en liste af prototyper at oprette objekter ud fra. Denne liste kan udvides, hvorved klienten uden videre får et større repertoire af objekter, den kan oprette.

16.6.1 Interfacet Cloneable

I Java er Prototype−designmønstret specielt let at implementere, da ethvert objekt har metoden clone() (arvet fra superklassen Object), der returnerer en kopi af objektet. Metoden kan dog kun kaldes på objekter, der tillader det. En klasse kan give tilladelse til at blive klonet ved at den (eller dens superklasse) implementerer interfacet Cloneable.

Ligesom Serializable markerer, at et objekt godt må serialiseres, markerer Cloneable, at et objekt godt må klones (kopieres ved at kalde clone()). Kaldes clone() på et objekt, der ikke implementerer Cloneable, opstår undtagelsen CloneNotSupportedException.

16.6.2 Overfladisk og dyb kopiering

Metoden clone() kopierer objektvariablerne, uanset om det er simple typer eller referencer til andre objekter. Hvis f.eks. et objekt husker en liste af strenge i et Vector−objekt, og vi kalder clone() på objektet for at få en kopi, vil kopien bruge det samme Vector−objekt (da det kun var objektreferencen der blev kopieret), og hvis "den enes" liste ændres, vil "den andens" liste også blive ændret. Dette kaldes en *overfladisk kopi* (eng.: shallow copy), da kopien har referencer til de samme objekter som originalen.

En *dyb kopi* (eng.: deep copy) er en kopiering, hvor alle de refererede objekter også kopieres, sådan at originalen og kopien er helt uafhængige og ikke deler objekter. Ønsker man at lave en dyb kopi af et objekt, må man selv skrive koden[1].

16.6.3 Eksempel: Et tegneprogram

Forestil dig et program, der kan tegne forskellige former og figurer på skærmen. Programmet har en palette af figurer. Brugeren vælger en figur fra paletten og klikker derefter på skærmen der, hvor den skal være.

Programmet skal altså registrere, hvilken figur brugeren har valgt, og derefter oprette et figur−objekt af den rette type og tegne det på skærmen med de rette koordinater.

En (ikke særlig raffineret) måde at løse denne opgave på er med en række if−sætninger. Hvis der blev klikket på figur 1, så opret et Figur1−objekt, hvis der blev klikket på figur 2, så opret et Figur2−objekt, osv.

I en mere raffineret løsning ville der være en liste af figur−prototyper, og paletten viser alle elementerne i denne liste. Når der klikkes på paletten, findes frem til det pågældende element, og det anvendes som Prototype til at oprette objektet, der skal tegnes på skærmen.

Denne løsning har den fordel, at paletten løbende kan udvides med nye figurer (og derfor er den mere fleksibel end f.eks. en Abstrakt Fabrik). Brugeren kan også tage en eksisterende figur og ændre den (f.eks. farven og om den er udfyldt) og lægge den ind i paletten.

Brugeren kunne måske endda kombinere nogle af figurobjekterne, der tilsammen udgjorde f.eks. et hus eller en mand, i et samlet Figur−objekt (se afsnit 18.5, Komposit/Rekursiv komposition), og udvide paletten med denne figur.

Her er et udkast til koden til eksemplet.

Superklassen Figur implementerer Cloneable og har metoden kopi():

```
public abstract class Figur implements Cloneable
{
  int x;
  int y;

  public Figur(int x1, int y1)
  {
    x=x1; y=y1;
  }

  /** Tegner figuren på skærmen */
  public abstract void tegn(java.awt.Graphics g);
```

```
  /** Giver en overfladisk kopi af dette objekt */
  public Figur kopi()
  {
    try {
      Figur kopien = (Figur) this.clone();
      return kopien;
    } catch (Exception e) { throw new InternalError("Kunne ikke klone"); }
  }
}
```

Cirkel har en radius:

```
public class Cirkel extends Figur
{
  int radius;

  public Cirkel(int x1, int y1, int radius1)
  {
    super(x1,y1);
    radius = radius1;
  }

  public void tegn(java.awt.Graphics g)
  {
    g.drawOval(x,y,radius,radius);
  }
}
```

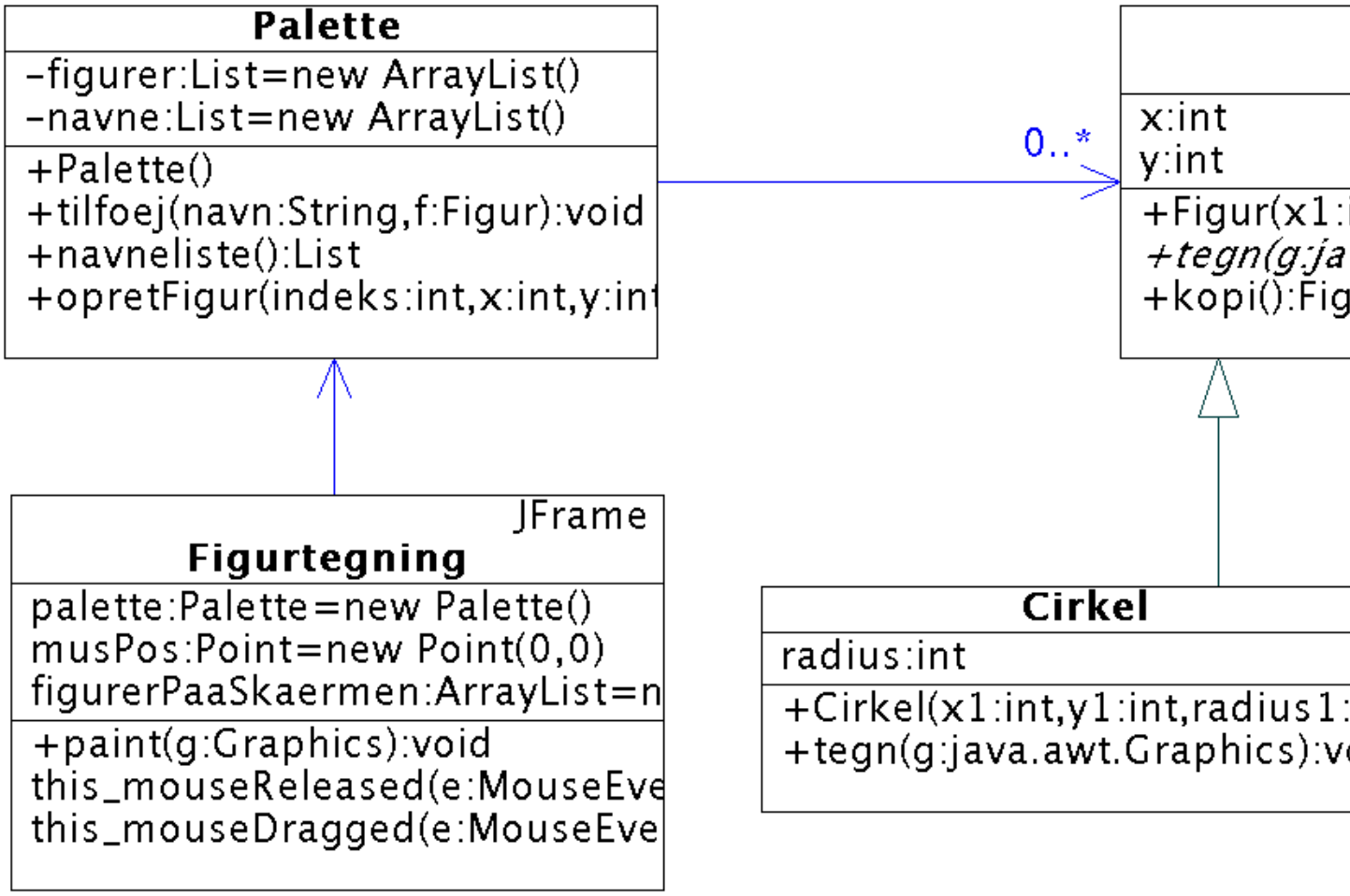
Firkant har en bredde og højde:

```
public class Firkant extends Figur
{
  int bredde;
  int højde;

  public Firkant(int x1, int y1, int bredde1, int højde1)
  {
    super(x1,y1);
    bredde = bredde1;
    højde  = højde1;
  }

  public void tegn(java.awt.Graphics g)
  {
    g.drawRect(x,y,bredde,højde);
  }
}
```

Paletten indeholder figurerne og deres navne (til f.eks. at lade brugeren vælge mellem dem i en valgliste). Paletten fungerer som bygmester med metoden opretFigur(), der opretter en figur ud fra prototype og derefter giver kopien de rigtige koordinater.

Metoden tilføj() kan bruges udefra til at føje nye figurer til paletten.

```
import java.util.*;
public class Palette
{
  /**
   * @associates <{Figur}>
   * @supplierCardinality 0..*
   */
  private List figurer = new ArrayList();
  private List navne = new ArrayList();

  public Palette()
  {
    tilføj("Cirkel", new Cirkel(10,10,10));
    tilføj("Firkant", new Firkant(0,0,20,20));
  }

  public void tilføj(String navn, Figur f)
  {
    navne.add(navn);
    figurer.add(f);
  }

  public List navneliste()
  {
    return navne;
  }

  public Figur opretFigur(int indeks, int x, int y)
  {
    Figur f = (Figur) figurer.get(indeks);
    f = f.kopi();
    f.x = x;
    f.y = y;
    return f;
  }
}
```



xxx Figurtegning fra

/home/j/dokumenter/vp/dm_eks_prototype_komposit_kommando/Figurtegning.java

ind

16.6.4 Opgave

Definér endnu en figur kaldet xxx

16.7 Objektpulje

Problem: Der er et begrænset antal resurser, som skal fordeles.

Problem: Der oprettes for mange objekter. Programmet er langsomt eller kører ujævnt, fordi der oprettes så mange objekter, der løbende smides væk igen. Objekterne kunne egentligt godt genbruges i stedet for at blive smidt væk, men oprettelserne sker spredt rundt i programmet, så det er svært at koordinere.

Løsning: Lad en Objektpulje (eng.: Object Pool) varetage resurserne/objekterne, og lad klienterne reservere og frigive objekter fra objektpuljen.

Idéen er at genbruge allerede eksisterende objekter, ved at have en pulje med ledige objekter, og på den måde undgå eller begrænse oprettelsen af nye objekter.

> Genbrug de samme objekter igen og igen ved at lave en Objektpulje, der husker objekterne

Objektpuljen har en fabrikeringsmetode, der fjerner et objekt fra puljen af ledige objekter og returnerer den. Hvis puljen er tom, oprettes et nyt objekt. Der skal også være en metode til at lægge et objekt, der ikke mere er i brug, tilbage i puljen.

En Objektpulje kan også bruges til at repræsentere begrænsede resurser. I dette tilfælde vil fabrikeringsmetoden, i stedet for at oprette nye objekter, vente, indtil et objekt bliver ledigt, før metoden returnerer (den kaldende tråd bliver altså stoppet).

16.7.1 Eksempel: Begrænsede resurser

Følgende klasse er en meget simpel objektpulje til begrænsede resurser. Hvis puljen løber tør for objekter, kastes en undtagelse.

```
import java.util.*;
```

```
/**
 * En pulje af objekter.
 * Objekterne skal først føjes til puljen udefra med kald til sætInd()
 */
public class Objektpulje
{
  private ArrayList ledige = new ArrayList();

  /**
   * Læg et ledigt objekt ind i puljen.
   * Bruges både til at initialisere puljen lige efter dens oprettelse,
   * og løbende, når et fjernet objekt bliver ledigt igen.
   */
  public synchronized void sætInd(Object obj)
  {
    ledige.add(obj);
  }

  /**
   * Tag et ledigt objekt ud af puljen.
   * @throws RuntimeException hvis puljen er løbet tør for objekter.
   */
  public synchronized Object tagUd()
  {
    if (ledige.isEmpty()) throw new RuntimeException("Ikke flere objekter");
    Object obj = ledige.remove(ledige.size()-1); // tag objekt ud af puljen
    return obj;
  }
}
```

16.7.2 Variation: Klient 'hænger', hvis puljen løber tør

I nogle tilfælde er det mere hensigtsmæssigt, at objektpuljen lader kalderen vente på, at der bliver et objekt ledigt. Det følgende eksempel er en objektpulje til begrænsede resurser, hvor den kaldende tråd, hvis puljen er tom, 'hænger', indtil der kommer et ledigt objekt.

```
import java.util.*;
/**
 * En pulje af objekter.
 * Objekterne skal tilføjes til puljen udefra.
 */
public class ObjektpuljeKlientHaenger
{
  private ArrayList ledige = new ArrayList();

  /**
   * Læg et ledigt objekt ind i puljen.
   * Bruges både til at initialisere puljen lige efter dens oprettelse,
   * og løbende, når et fjernet objekt bliver ledigt igen.
   */
  public synchronized void sætInd(Object obj)
  {
    ledige.add(obj);
    this.notify(); // væk eventuelle ventende tråde
  }

  /**
   * Tag et ledigt objekt ud af puljen.
   * Er der ikke flere objekter tilbage 'hænger' kaldet, indtil
   * et objekt bliver ledigt.
   */
  public synchronized Object tagUd()
  {
    try {
      while (ledige.isEmpty()) // så længe der ikke er ledige objekter...
      {
        System.out.println("Ikke flere objekter i puljen, venter...");
        this.wait();           // .... vent på at blive vækket
      }
      Object obj = ledige.remove(ledige.size()-1); // tag objekt
      return obj;
    } catch (InterruptedException e) {
      e.printStackTrace();
      return null;
    }
  }
}
```

16.7.3 Genbrug af tråde

At oprette en tråd er dyrt i kørselstid. Ønsker man derfor et hurtigt system, bør man genbruge tråde i stedet for at oprette og nedlægge dem alt for ofte.

Det følgende eksempel viser, hvordan en pulje af tråde kan implementeres. En tråd aktiveres ved at kalde metoden startOpgave() med et objekt, der så vil få kaldt sin run()−metode i en separat arbejds−tråd (er der ingen ledige tråde, oprettes en ny).

```
import java.util.*;
```

```
public class Traadpulje
{
  private List ledige = new ArrayList(); // Listen over ledige arbejdstråde

  /** Læg en opgave i kø til en arbejdstråd */
  public synchronized void startOpgave(Runnable opgave)
  {
    Arbejder a;

    if (ledige.isEmpty())
    {
      a = new Arbejder(); // ingen arbejdstråde ledige, en ny oprettes
      a.setDaemon(true);  // tillad systemet at lukke ned selvom tråden er aktiv
      System.out.println("Ny arbejdstråd oprettet.");
      a.start();
    } else synchronized(ledige) {
      a = (Arbejder) ledige.remove(ledige.size()-1);// tag arbejdstråd fra liste
    }

    synchronized(a)
    {
      if (a.opgave != null) throw new InternalError("Tråden kører allerede");
      a.opgave = opgave;
      a.notify();         // væk arbejdstråden der venter i wait()
    }
  }

  /**
   * Arbejds-tråden.
   * Den er stærkt bundet til puljen så den er lagt som en privat indre klasse.
   */
  private class Arbejder extends Thread
  {
    private Runnable opgave = null;

    public final synchronized void run()
    {
      while (true) try
      {
        if (opgave != null)
        {
          System.out.println(this+" virker nu på "+opgave);
          opgave.run();      // udfør opgaven
          opgave = null;     // ... og glem den (!)
          synchronized(ledige) {ledige.add(this);} // læg tråd tilbage i listen
        }
        System.out.println(this+" venter på opgave.");
        this.wait();         // vent på at blive vækket med notify()
      } catch (Exception e) {
        System.err.println(this+": Fejl opstod i opgave "+opgave);
        e.printStackTrace();
      }
    } // slut på run()
  }  // slut på den indre klasse
}
```

Trådpuljen kunne for eksempel udnyttes i en webserver (her er den anvendt på eksemplet FlertraadetHjemmesidevaert fra kapitel 17 i http://javabog.dk):

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

public class FlertraadetHjemmesidevaertMedTraadpulje
{
  public static void main(String arg[])
  {
    try {
      ServerSocket værtssokkel = new ServerSocket(8001);

      Traadpulje trådpulje = new Traadpulje();          // nyt
      while (true)
      {
        Socket forbindelse = værtssokkel.accept();
        Anmodning a = new Anmodning(forbindelse);

        trådpulje.startOpgave(a);                       // nyt
        // før: new Thread(a).start();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}

class Anmodning implements Runnable
{
  private Socket forbindelse;

  Anmodning(Socket forbindelse)
  {
```

```
    this.forbindelse = forbindelse;
  }

  public void run()
  {
    try {
      PrintWriter ud = new PrintWriter(forbindelse.getOutputStream());
      BufferedReader ind = new BufferedReader(
        new InputStreamReader(forbindelse.getInputStream()));

      String anmodning = ind.readLine();
      System.out.println("start "+new Date()+" "+anmodning);

      ud.println("HTTP/0.9 200 OK");
      ud.println();
      ud.println("<html><head><title>Svar</title></head>");
      ud.println("<body><h1>Svar</h1>");
      ud.println("Tænker over "+anmodning+"<br>");
      for (int i=0; i<100; i++)
      {
        ud.print(".<br>");
        ud.flush();
        Thread.sleep(100);
      }
      ud.println("Nu har jeg tænkt færdig!</body></html>");
      ud.flush();
      forbindelse.close();
      System.out.println("slut  "+new Date()+" "+anmodning);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

16.7.4 Opgave: Objektpulje med objektfabrik

Udvid Objektpulje, sådan at den kan få overført en Objektfabrik i konstruktøren. Klassen Objektfabrik har metoden opretObjekt(), der skaber nye objekter til puljen. Ændr puljen til at bruge fabrikken, sådan at der oprettes nye objekter, hvis puljen løber tør.

16.8 Større eksempel: Dataforbindelse

En forbindelse til en database kan med fordel implementeres som en Singleton, hvis man ønsker, at alle forespørgsler skal gå gennem det samme objekt. Det kunne være for at cache forespørgslerne eller for at sikre konsistens i data.

Det følgende er også et eksempel på indkapsling og abstraktion − al SQL−specifik kode er pakket ind i en klasse for sig, i et Dataforbindelse−objekt.

For at abstraktionen skal holde, skal de data (her drejer det sig om kunder), der overføres mellem dataforbindelsen og klientprogrammet, også repræsenteres på en måde, så de ikke er afhængige af, at kommunikationen skal foregå med JDBC til en SQL−database:

```
import java.io.*;

public class Kunde implements Serializable
{
  public String navn;
  public double kredit;

  public Kunde(String n, double k)
  {
    navn = n;
    kredit = k;
  }

  public String toString() { return navn+": "+kredit+" kr."; }
}
```

Klient−programmet kan så abstrahere fra, hvordan data er lagret:

```
import java.util.*;

public class BenytDataforbindelse
{
  public static void main(String arg[])
  {
    try {
      Dataforbindelse dbf = Dataforbindelse.hentForbindelse();

      List liste = dbf.hentAlle();
      System.out.println("Alle data: "+ liste);
      dbf.sletAlleData();

      System.out.println("Alle data nu: "+dbf.hentAlle());

      dbf.indsæt( new Kunde("Kurt",1000) );
      dbf.indsæt( new Kunde("kunde indsat fra BenytDataforbindelse", 2) );
      System.out.println("Alle data nu: "+dbf.hentAlle());
```

```
    } catch(Exception e) {
      System.out.println("Problem med dataforbindelse: "+e);
      e.printStackTrace();
    }
  }
}
```

```
Alle data: [Kurt: 1000.0 kr., kunde indsat fra BenytDataforbindelse: 2.0 kr.]
Alle data nu: []
Alle data nu: [Kurt: 1000.0 kr., kunde indsat fra BenytDataforbindelse: 2.0 kr.]
```

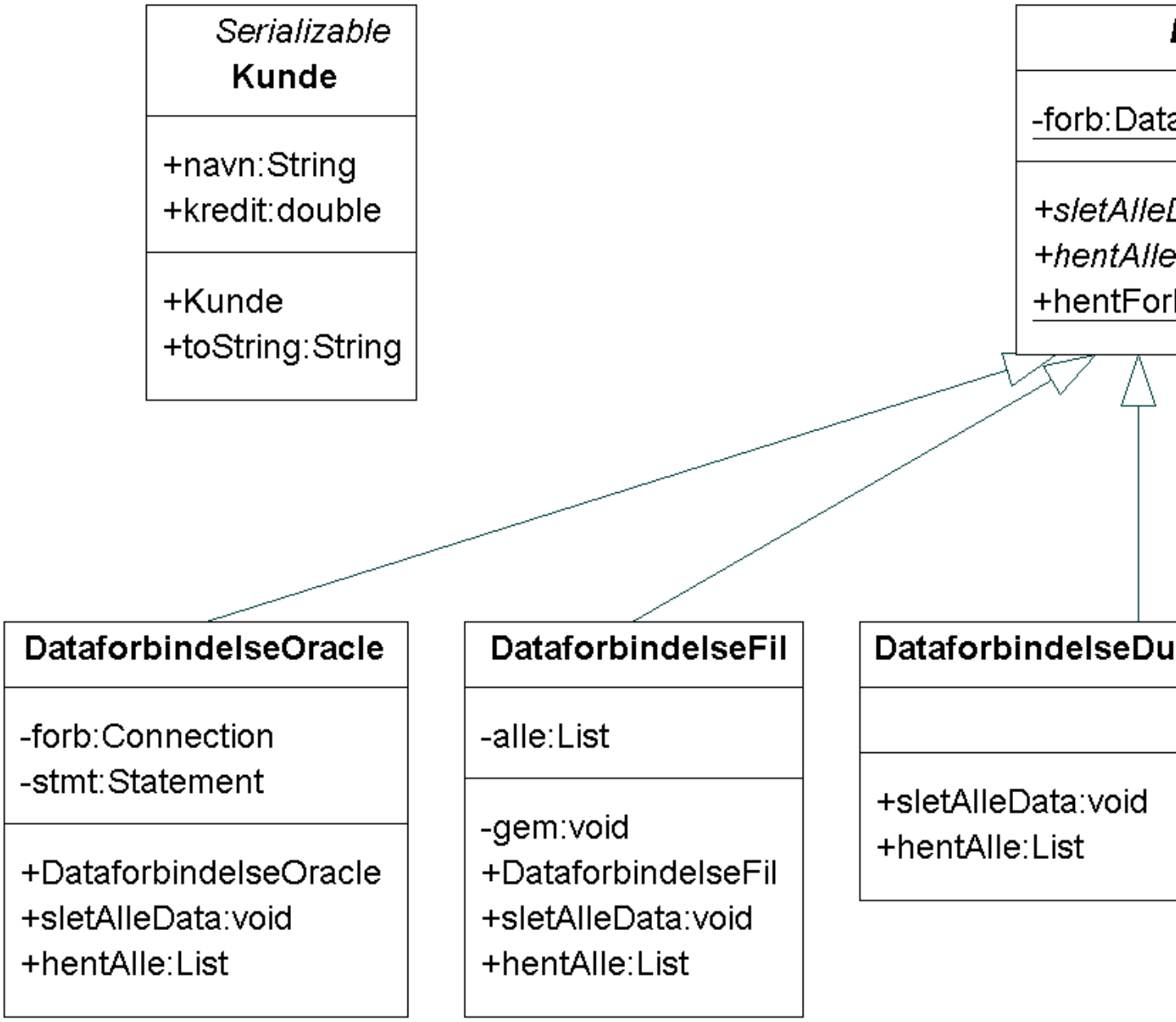
Herunder er Dataforbindelse programmeret som en Singleton (noget af koden er skåret væk).

```
import java.util.*;

public abstract class Dataforbindelse
{
  abstract public void sletAlleData() throws Exception;
  abstract public void indsæt(Kunde k) throws Exception;
  abstract public List hentAlle() throws Exception;

  // fabrikeringsmetode
  synchronized public static Dataforbindelse hentForbindelse() throws Exception
  {
    // − her implementeret som en Singleton

    if (forb != null) return forb;
    // ... her skal kode ind der skaffer en forbindelse

    forb = new DataforbindelseDummy();    // lige nu
    //forb = new DataforbindelseOracle(); // senere
    //forb = new DataforbindelseOverNetvaerketTilEnServlet(); // endnu senere!!

    return forb;
  }
  private static Dataforbindelse forb;
}
```

Fabrikeringsmetoden Dataforbindelse.hentForbindelse() kan senere ændres til at skaffe andre slags forbindelser, uden at der skal rettes i klient−programmet.

16.8.1 Specialiseringer af Dataforbindelse

Der kan være forskellige specialiseringer af Dataforbindelse til forskellige situationer:



I starten af programmeringen udvikles nok en DataforbindelseDummy (eller DataforbindelseStub), der kun er beregnet til test. Den lader, som om den er en forbindelse, men gør reelt ingen verdens ting:

```
import java.util.*;

public class DataforbindelseDummy extends Dataforbindelse
{
  public void sletAlleData()  { System.out.println("sletAlleData() kaldt"); }
  public void indsæt(Kunde k) { System.out.println("indsæt("+k+") kaldt"); }

  public List hentAlle()
  {
    List alle = new ArrayList();
    Kunde k = new Kunde( "Jacob", -1722);
    alle.add(k);
    return alle;
  }
}
```

I denne udgave har vi ikke en gang gidet at huske data (selvom det ville være nemt nok i dette tilfælde).

Senere laves en, der husker data, men serialiseret i en fil:

```
import java.util.*;
import java.io.*;
public class DataforbindelseFil extends Dataforbindelse {
  private List alle;

  private void gem() {
    try {
```

```
      ObjectOutputStream p = new ObjectOutputStream(
        new FileOutputStream("data.ser"));
      p.writeObject(alle);
      p.close();
    } catch (Exception e) { e.printStackTrace(); }
  }

  public DataforbindelseFil() {
    try {
      ObjectInputStream p = new ObjectInputStream(
        new FileInputStream("data.ser"));
      alle = (List) p.readObject();
      p.close();
    } catch (Exception e) {
      alle = new ArrayList();
    }
  }

  public void sletAlleData() {
    alle = new ArrayList();
    gem();
  }

  public void indsæt(Kunde k) {
    alle.add(k);
    gem();
  }

  public List hentAlle() { return alle; }
}
```

Senere, når de rigtige tabeller osv. er blevet oprettet i databasen, laves en rigtig forbindelse:

```
import java.sql.*;
import java.util.*;
public class DataforbindelseOracle extends Dataforbindelse {
  private Connection forb;
  private Statement stmt;

  public DataforbindelseOracle() throws Exception {
    Class.forName("oracle.jdbc.driver.OracleDriver");
    Connection forb = DriverManager.getConnection(
      "jdbc:oracle:thin:@ora.javabog.dk:1521:student","stuk1001","hemli'");
    stmt = forb.createStatement();
  }

  public void sletAlleData() throws SQLException {
    stmt.execute("truncate table kunder");
  }

  public void indsæt(Kunde k) throws SQLException {
    stmt.executeUpdate(
      "INSERT INTO kunder VALUES('" + k.navn + "', " + k.kredit + ")");
  }

  public List hentAlle() throws SQLException {
    List alle = new ArrayList();
    ResultSet rs = stmt.executeQuery("SELECT navn, kredit FROM kunder");
    while (rs.next())
    {
      Kunde k = new Kunde( rs.getString("navn"), rs.getDouble("kredit"));
      alle.add(k);
    }
    return alle;
  }
}
```

16.8.2 Dataforbindelse over netværk

Endnu senere kan det være, at programmet er udvidet med en forbindelse, der kontakter en servlet på en webserver et andet sted på netværket. Altså et objekt, der 'lader som om', det er det rigtige objekt (en Dataforbindelse), men i virkeligheden sender kaldene videre til det rigtige Dataforbindelse−objekt (der i dette tilfælde ligger på en anden maskine). Dette kaldes også en Proxy − se afsnit 17.1.

Herunder er sådan en forbindelse implementeret. Den sender en GET−anmodning med HTTP−protokollen v.hj.a. URL−klassen. Med anmodningen sendes parameteren kommando, der beskriver, hvad der skal gøres. Eventuelle data indkodes med URLEncoder−klassen (i metoden indsæt()). For eksempel kunne en anmodning om at slette alle data se således ud:

```
http://localhost:8080/servlet/DataforbindelseServlet?kommando=sletAlleData
```

De fleste kommandoer forventer ikke noget svar. Eneste undtagelse er kommando=hentAlle, der forventer, at servletten sender binære data tilbage i form af serialiserede objekter.

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

/**
```

```
 * Dataforbindelse over netværket til en servlet.
 * @see DataforbindelseServlet
 */
public class DataforbindelseOverNetvaerketTilEnServlet extends Dataforbindelse
{
  private String basisUrl;

  private InputStream spørg(String spm) throws IOException
  {
    System.out.println("Spørger på "+ basisUrl+"?"+spm);
    URL u = new URL(basisUrl+"?"+spm);// opret URL
    u.openConnection().connect();     // send forespørgslen
    return u.openStream();            // returner datastrøm med svaret
  }

  public DataforbindelseOverNetvaerketTilEnServlet(String urlPåServlet)
  {
    basisUrl = urlPåServlet;
  }

  public void sletAlleData() throws IOException
  {
    spørg("kommando=sletAlleData");
  }

  public void indsæt(Kunde k) throws IOException
  {
    spørg("kommando=indsæt"
      +"&navn="   +URLEncoder.encode(k.navn)        // indkod navn
      +"&kredit=" +URLEncoder.encode(""+k.kredit)); // indkode kredit
  }

  public List hentAlle() throws Exception
  {
    InputStream is = spørg("kommando=hentAlle");
    ObjectInputStream p = new ObjectInputStream(is);
    List alle = (List) p.readObject();             // deserialisér liste-objekt
    p.close();

    return alle;
  }
}
```

Herunder er den tilsvarende servlet. Den bruger et dataforbindelses−objekt på *værten* til at afgøre, hvad den skal svare. Her er det vigtigt at denne Dataforbindelse på værtsmaskinen ikke selv forsøger at få data fra en servlet (data skal jo komme et sted fra i sidste ende).

Derfor giver den Dataforbindelse et vink (med Dataforbindelse.sætForbindelsesvink()) om hvilken forbindelsestype den ønsker.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class DataforbindelseServlet extends HttpServlet {

  public void init(ServletConfig config) throws ServletException {
    super.init(config);

    // Det er vigtigt at dataforbindelsen på værten IKKE forsøger
    // at få forbindelse med en servlet, så vi sætter et vink om
    // hvilken forbindelse vi ønsker
    try {
      Dataforbindelse.sætForbindelsesvink("fil");
    } catch (Exception e) { e.printStackTrace(); }
  }

  public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    try {
      String kommando = request.getParameter("kommando");
      Dataforbindelse dbforb = Dataforbindelse.hentForbindelse();

      System.out.println("kommando="+kommando);
      // Test − for at sikre os at det rent faktisk kommer gennem servletten
      // sender vi en ekstra "kunde" med
      // dbf.indsæt( new Kunde( "Servlet kommando = "+kommando, 0) );

      if ("sletAlleData".equals(kommando)) dbforb.sletAlleData();
      else if ("hentAlle".equals(kommando))
      {
        // sæt indholdstypen til noget binært (bare noget andet end text/html
        response.setContentType("application/x−serialiserede−kunder");

        // serialisér svaret og send det til klienten
        List alle = dbforb.hentAlle();
        ObjectOutputStream p=new ObjectOutputStream(
          response.getOutputStream());
        p.writeObject(alle);
        p.close();
        return;
```

```
      }
      else if ("indsæt".equals(kommando)) {
        Kunde k = new Kunde (
          request.getParameter("navn"),
          Double.parseDouble( request.getParameter("kredit") )
        );
        dbforb.indsæt(k);
      } else throw new IllegalArgumentException("ukendt kommando: "+kommando);

    } catch (Exception e) {
      e.printStackTrace();
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      System.out.println();
      out.println("<html>");
      out.println("<head><title>DataforbindelseServlet − fejl</title></head>");
      out.println("<body><h1>Fejl: "+e+"</h1>");
      out.println("<p>Der opstod en fejl i servletten:</p>");
      e.printStackTrace(out);
      out.println("</body></html>");
    }
  }
}
```

Teknikken med at serialisere objekter og sende dem over netværket over HTTP anvendes ofte til at sende objekter til en applet.

16.8.3 Dataforbindelse, der cacher forespørgsler

Efterhånden som programmet bliver mere og mere udviklet, opdager vi måske, at nogle dele af programmet, der laver hyppige dataforespørgsler, kører for langsomt.

De fleste af forespørgslerne er endda overflødige, da data ikke har ændret sig.

Nu kunne vi selvfølgelig kode DataforbindelseOverNetvaerketTilEnServlet om til også at cache forespørgslerne, men det ville give en lavere kohæsion (diskuteret i afsnit 15.7.2, Høj kohæsion), da klassen så ville have to ansvarsområder (forespørgsler over netværket og caching).

En mere elegant løsning ville være at lave en klasse, der kun tog sig af caching. Den kunne lade som om, den var en Dataforbindelse, men i virkeligheden kalder den videre i en anden Dataforbindelse, hvis den ikke har svaret (dette kaldes også en Proxy − se afsnit 17.1).

```
import java.util.*;

public class DataforbindelseCache extends Dataforbindelse
{
  private Dataforbindelse df;
  private List cache;

  public DataforbindelseCache(Dataforbindelse forb) {
    df = forb;
  }

  public void sletAlleData() throws Exception
  {
    cache = null;            // der er sket en ændring − nulstil cache
    df.sletAlleData();
  }

  public void indsæt(Kunde k) throws Exception
  {
    cache = null;            // der er sket en ændring − nulstil cache
    df.indsæt(k);
  }

  public List hentAlle() throws Exception
  {
    if (cache != null)
    {
      // Vi har listen i cachen − returnér den, uden at spørge videre
      return cache;
    }
    else
    {
      // Øv − vi har ikke listen i cachen − vi er nødt til at spørge videre
      cache = df.hentAlle(); // husk listen til en anden gang
      return cache;
    }
  }
}
```

16.8.4 Endelig udgave af Dataforbindelse

Her er Dataforbindelse.java, som den ser ud sidst i programmeringsforløbet:

```
import java.util.*;

public abstract class Dataforbindelse
```

```
{
  /**
   * Sletter alle data
   */
  abstract public void sletAlleData() throws Exception;

  /**
   * Indsætter en kunde
   * @param kunden
   */
  abstract public void indsæt(Kunde k) throws Exception;

  /**
   * Henter alle kunderne
   * @return en liste af Kunde-objekter
   */
  abstract public List hentAlle() throws Exception;

  private static Dataforbindelse forb;
  /**
   * Fabrikeringsmetode til at skaffe en forbindelse
   * @return forbindelsen
   */
  synchronized public static Dataforbindelse hentForbindelse() throws Exception
  {
    // implementeret som en Singleton, så der må kun eksistere ét objekt
    if (forb != null) return forb;

    // ... herunder skal kode ind der skaffer en forbindelse
    //forb = new DataforbindelseDummy();  // før
    //forb = new DataforbindelseFil();    // senere
    forb = new DataforbindelseOracle();
    // endnu senere !!!
    // forb = new DataforbindelseOverNetvaerketTilEnServlet(
    //   "http://localhost:8080/servlet/DataforbindelseServlet");

    return forb;
  }

  /**
   * Lavet sådan at man kan angive hvilken slags forbindelse man foretrækker.
   * Dette er rart til afprøvning og fejlfinding.
   * Skal kaldes som <b>før</b> første kald til <code>hentForbindelse()</code>.
   * Eksempler på brug:<br>
   * <code>Dataforbindelse.sætForbindelsesvink("dummy");</code><br>
   * <code>Dataforbindelse.sætForbindelsesvink("fil");</code><br>
   * <code>Dataforbindelse.sætForbindelsesvink(
   *   "http://localhost:8080/servlet/DataforbindelseServlet");</code>
   *
   * @param vink Vink til hvilken type forbindelse der foretrækkes
   * @see #hentForbindelse()
   */
  synchronized public static void sætForbindelsesvink(String v) throws Exception
  {
    if (forb != null) return; // ignorer vink når forbindelse først er oprettet.
    if (v.equals("dummy"))         forb = new DataforbindelseDummy();
    else if (v.startsWith("fil"))  forb = new DataforbindelseFil();
    else if (v.startsWith("http")) forb = new DataforbindelseCache(
              new DataforbindelseOverNetvaerketTilEnServlet(v));    // brug cache
    else forb = new DataforbindelseOracle();
  }
}
```

Bemærk, at vi *ikke* smider de tidlige udgaver, f.eks. DataforbindelseDummy, ud. De kan måske være nyttige senere, f.eks. er det ikke utænkeligt, at man senere skal videreudvikle en anden del af programmet, men af den ene eller anden grund ikke kan/vil koble op til database.

1En mulighed er dog at bruge serialisering − serialisere objektet til en datastrøm (en fil eller et array af byte) og deserialisere det igen. Dette er dog ikke nogen særlig effektiv måde at lave kopier af objekter, så den anbefales ikke, hvis programmet skal køre hurtigt, eller der skal kopieres mange objekter.

17 Hyppigt anvendte designmønstre

Det er en god idé at have kigget i afsnit 1.1, Lister og mængder, kapitel 8, Databaser (JDBC), afsnit 15.9, Introduktion til designmønstre og kapitel 16, Skabende designmønstre, før man læser dette kapitel.

I dette kapitel vil vi beskrive nogle af de designmønstre, der ofte ses anvendt i i standardbiblioteket og i lidt større programmer.

Den overordnede idé i mange af designmønstrene er, som diskuteret i afsnit 15.9, at give idéer til, hvordan man afkobler (dvs. mindsker graden af bindinger) mellem den del af programmet, som *bruger* nogle objekter (kaldet klienten), og den del af programmet (dvs. de klasser), der bruges.

17.1 Proxy

Problem: Et objekt, der bliver brugt af klienten, skal nogen gange bruges lidt anderledes, men ikke altid, så det er uhensigtsmæssigt at ændre i klassen eller i klienten.

Løsning: Lav en Proxy−klasse, der *lader* som om, den er det rigtige objekt, og kalder videre i det rigtige objekt.

> Få metodekald til et objekt til at gå igennem et Proxy−objekt (mellem−objekt), der modtager metodekald på vegne af (fungerer som en erstatning) for det rigtige objekt

Proxy kunne på dansk hedde "stråmand" eller "mellemmand". Ordet brugtes oprindeligt i banksektoren, men de fleste kender i dag kun ordet i forbindelse med internettet: Har man ikke direkte forbindelse til internettet kan det være nødvendigt at konfigurere proxy−indstillingerne i sin netlæser, sådan at den sender forespørgslerne til en proxy−server, der spørger videre ud på internettet.

Oftest ved klienten ikke at den bruger en proxy. Når proxyen bliver kaldt, vil den som regel delegere kaldet videre til det andet objekt, men den kan også vælge f.eks.:

- at returnere med det samme og udføre kaldet i baggrunden
- at afvise kaldet (f.eks. ved at kaste en undtagelse)
- at udføre kaldet på en anden måde (f.eks. anderledes parametre)

17.1.1 Simpelt eksempel: En stak, der logger kaldene

I stak−eksemplet fra afsnit 15.3.2 kunne det måske være rart under programudviklingen at logge de metodekald, der bliver foretaget på stakken.

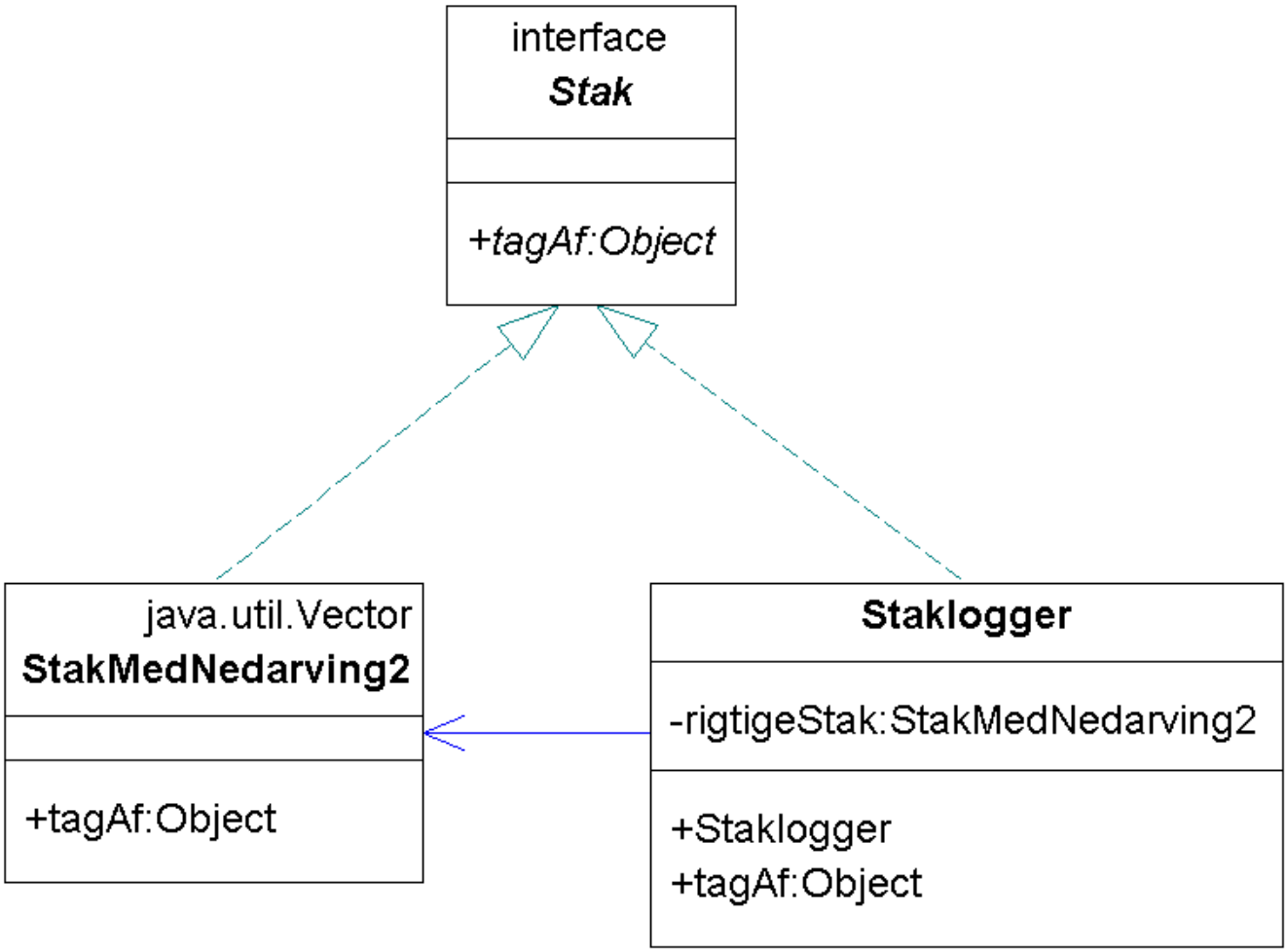
Vi kunne da lave en Proxy til stakken, der udskriver kaldene:

```
public class Staklogger implements Stak
{
  private StakMedNedarving2 rigtigeStak;

  public Staklogger(StakMedNedarving2 s) {  rigtigeStak = s; }

  public void lægPå(Object o)
  {
    System.out.print("Staklogger: lægPå("+o+")");
    rigtigeStak.lægPå(o);
  }

  public Object tagAf()
  {
    Object o = rigtigeStak.tagAf();
    System.out.print("Staklogger: tagAf() gav: "+o);
    return o;
  }
}
```

Der, hvor vi opretter objektet, pakker vi det rigtige Stak−objekt ind i vores Staklogger:

```
  Stak s = new Staklogger( new StakMedNedarving2() );
```

Staklogger vil nu blive brugt på vegne af (i stedet for) StakMedNedarving2, uden at klienten (resten af programmet) ved det.



Staklogger er en Stak, der delegerer videre til en anden Stak (en StakMedNedarving2)

Bemærk, at i dette simple eksempel refererer StakLogger til en StakMedNedarving2−klasse. Man kan gøre StakLogger mere generelt anvendeligt ved at referere til Stak−interfacet i stedet (jvf. afsnit 15.3).

17.1.2 Variationer af designmønstret Proxy

Der findes nogle almindelige variationer af Proxy−designmønstret:

1. Fjernproxy − bruges, når man har brug for en lokal repræsentation af et objekt, der ligger på en anden maskine. Afsnit 16.8.2 Dataforbindelse over netværk er et eksempel på dette. RMI (Remote Method Invocation) beskrevet i afsnit 14.3 anvender også dette princip.
2. Cache − fungerer som proxy for et objekt med nogle omkostningsfulde metodekald. I de tilfælde hvor en tidligere cachet returværdi fra metodekaldet kan bruges, foretages kaldet ikke, men den cachede værdi returneres i stedet (se afsnit 16.8.3, Dataforbindelse, der cacher forespørgsler).
3. Adgangssproxy − bestemmer, hvad klienten kan gøre med det virkelige objekt. Et eksempel kan findes i næste afsnit.
4. Virtuel Proxy − udskyder oprettelsen af omkostningsfulde objekter, indtil der er brug for dem. Et eksempel kan findes i afsnit 17.1.5.

17.1.3 Eksempel: Gøre data uforanderlige vha. Proxy

Dette eksempel viser, hvordan en samling af data (af type Collection) kan gøres uforanderlig (dvs. at data i objektet ikke kan ændres, efter at objektet er oprettet − se afsnit 18.1 for en nærmere diskussion) ved hjælp af en Proxy.

Klassen bruges ved at pakke den oprindelige samling ind i proxy−klassen (klassen UforanderligSamling vist herunder) og derefter kun huske referencen til proxyen:

```
Collection d = new ArrayList();
d.add("Hej");
d.add("med");
d.add("dig");
...

d = new UforanderligSamling(d);
// herefter kan dataene ikke mere ændres gennem d
```

UforanderligSamling delegerer alle forespørgsler videre til den oprindelige samling, mens alle ændringer afvises ved at kaste undtagelsen UnsupportedOperationException.

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

public class UforanderligSamling implements Collection, Serializable
{
  private Collection c; // til videredelegering

  UforanderligSamling(Collection c) {
    if (c==null) throw new NullPointerException();
    this.c = c;
  }

  // videredelegering af kald, der ikke ændrer samlingen c
  public int size()                          { return c.size(); }
  public boolean isEmpty()                   { return c.isEmpty(); }
  public boolean contains(Object o)          { return c.contains(o); }
  public boolean containsAll(Collection coll) { return c.containsAll(coll); }
  public Object[] toArray()                  { return c.toArray(); }
  public Object[] toArray(Object[] a)        { return c.toArray(a); }
  public String toString()                   { return c.toString(); }

  private static void fejl() {
    throw new UnsupportedOperationException("Denne samling kan ikke ændres");
  }

  // afvisning af kald, der ændrer samlingen
  public void clear()                    { fejl(); }
  public boolean add(Object o)           { fejl(); return false; }
  public boolean remove(Object o)        { fejl(); return false; }
  public boolean addAll(Collection c)    { fejl(); return false; }
  public boolean removeAll(Collection c) { fejl(); return false; }
  public boolean retainAll(Collection c) { fejl(); return false; }

  // iteratorer skal afvise ændringer, men ellers fungere som c's iterator
  public Iterator iterator()
  {
    return new Iterator() {       // anonym klasse, der implementerer Iterator
      Iterator i = c.iterator();  // til videredelegering til c's iterator
      public boolean hasNext() { return i.hasNext(); }
      public Object next()     { return i.next(); }
      public void remove()     { fejl(); }
    };
  }
}
```

I afsnit 1.6.6, Uforanderlige samlinger, vises, hvad der sker, når man prøver at ændre i en uforanderlig samling. Ovenstående svarer nemlig til det Proxy−objekt man får hvis man kalder Collections.unmodifiableCollection().

17.1.4 Doven Initialisering/Virtuel Proxy

Bruges en Proxy, kan oprettelsen af det andet objekt egentlig godt udskydes, *indtil første gang der er brug for det*. Så kalder man proxyen en Virtuel Proxy. Første gang proxyen får brug for at kalde videre i det andet objekt, oprettes dette.

> En Virtuel Proxy modtager metodekald på vegne af et andet objekt, som det først opretter, når der er brug for det

Omkostningen ved at programmere en Virtuel Proxy er, at hver gang objektet skal bruges, skal det først tjekkes, om objektet er blevet oprettet.

Der kan være flere grunde til at bruge en Virtuel Proxy:

- Det rigtige objekt kan ikke oprettes endnu, f.eks. fordi det afhænger af andre objekter, der ikke er klar endnu på oprettelsestidspunktet.
- At oprette det rigtige objekt er dyrt i hukommelses− eller CPU−forbrug, og det er måske slet ikke sikkert, at programmet kommer til at bruge objektet, så det er en fordel at udskyde oprettelsen.

17.1.5 Eksempel på Virtuel Proxy: En stak der først oprettes, når den skal bruges

Her er en Virtuel Proxy til stakken fra afsnit 15.3.2:

```
public class VirtuelStak implements Stak
{
  private Stak rigtigeStak;

  public void lægPå(Object o)
  {
    if (rigtigeStak==null) rigtigeStak = new StakMedNedarving2();
    rigtigeStak.lægPå(o);
  }

  public Object tagAf()
  {
    if (rigtigeStak==null) rigtigeStak = new StakMedNedarving2();
    return rigtigeStak.tagAf();
  }
}
```

Der, hvor vi opretter objektet, bruger vi den virtuelle stak:

```
  Stak s = new VirtuelStak();
```

Den virtuelle stak er nu oprettet, men ikke den rigtige stak.

Den oprettes første gang lægPå() kaldes:

```
  ...
  s.lægPå("Hej");  // først her oprettes den rigtige stak
  s.lægPå("med");
  s.lægPå("dig");
```

Der er egentlig ikke nogen specielt god grund til at give en stak en Virtuel Proxy for noget så simpelt som en stak − eksemplet er valgt, fordi det er simpelt og illustrativt.

17.2 Adapter

Problem: Et system forventer et objekt af en bestemt type (der implementerer et bestemt interface eller arver fra en bestemt klasse), men det objekt, man ønsker at give til systemet, har ikke denne type.

Løsning: Definér et Adapter−objekt af den type, som systemet forventer, og lad Adapter−objektet delegere kaldene videre til det rigtige objekt.

> Få et objekt til at passe ind i et system ved at bruge et Adapter−objekt, der passer ind i systemet, og som kalder videre i det rigtige objekt
>
> En Adapter fungerer som omformer mellem nogle klasser

I almindeligt sprogbrug er en adapter en lille omformer, der gør det muligt at forbinde et stik og en fatning, der ellers ikke ville passe sammen. Som designmønster er en Adapter er en klasse, der fungerer som 'lim' mellem nogle klasser og får dem til at fungere sammen, selvom de ikke umiddelbart er beregnet til at spille sammen.

Man kunne f.eks. i en virksomhed stå i den situation, at man har lavet en del af et program selv, men så ønsker at udbygge programmet med nogle klasser, der er lavet af en ekstern udvikler, som ikke kender noget til resten af programmet. Virksomhedens egne klasser implementerer et givet interface, men den del af programmet, der er lavet af den eksterne udvikler, implementerer ikke dette interface. Hvad gør man så?

- En mulighed er at omskrive den del af programmet, der er lavet eksternt, sådan at klasserne kommer til at implementere virksomhedens eget interface. Det kan være en besværlig og tidskrævende proces.
- En anden mulighed er at lave en adapterklasse, som implementerer virksomhedens interface, og som så sørger for at kalde videre i de klasser, der er lavet eksternt.

Typisk implementerer en Adapter altså et interface, der er kendt af klienten, og formidler adgang til en klasse, der ikke er kendt af klienten.

17.2.1 Simpelt eksempel

Lad os sige, at vi har nogle opgaver, der skal udføres, og at vi har defineret klassen Opgave, der tager sig af disse opgaver:

```
public class Opgave
{
  public void udfør() {
    // noget kode her til at udføre opgaven
    // ...
    System.out.println("Opgave udført.");
  }
}
```

Nu viser det sig senere, at vi får brug for at køre opgaven i en separat tråd. For at køre noget i en separat tråd skal interfacet Runnable (der specificerer run()−metoden) implementeres.

Nu kunne vi selvfølgelig ændre klassen Opgave, så den implementerede Runnable, men vi kunne også vælge at lave en Adapter, der får Opgave til at passe ind i Runnable:

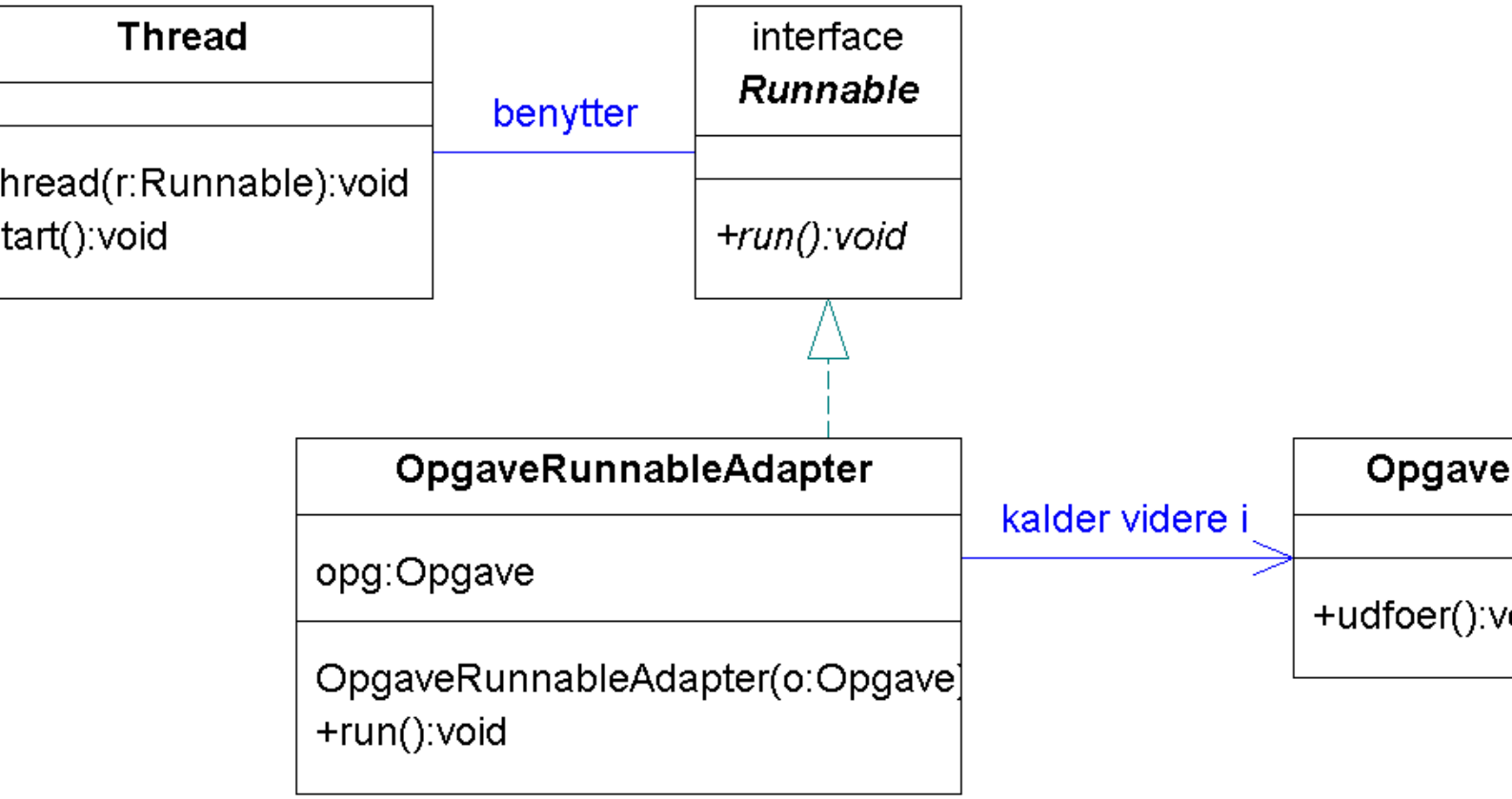
```
public class OpgaveRunnableAdapter implements Runnable
{
  Opgave opg;
  OpgaveRunnableAdapter(Opgave o) { opg = o; }
  public void run() {
    opg.udfør();              // Oversæt kald af run() til kald af udfør()
  }
}
```

Derefter kan vi passe et Opgave−objekt ind i en tråd:

```
public class BenytOpgaveRunnableAdapter
{
  public static void main(String[] args) {
    Opgave opgave = new Opgave();
```

```
    Runnable r = new OpgaveRunnableAdapter(opgave);
    Thread t = new Thread(r);
    t.start();
  }
}
```

```
Opgave udført.
```



Vi ønsker Opgave kørt af en tråd, men Thread kræver, at det implementerer Runnable, så vi laver en adapter til Opgave.

17.2.2 Anonyme klasser som adaptere

En anonym klasse er en klasse uden navn, som der oprettes et objekt ud fra der, hvor den defineres. F.eks.:

```
public class KlasseMedAnonymKlasse
{
  public void metode()
  {
    // ... programkode for metode

    X objektAfAnonymKlasse = new X()
    {
      void metodeIAnonymKlasse()
      {
        // programkode
      }
      // flere metoder og variabler i anonym klasse
    };

    // mere programkode for metode
  }
}
```

Lige efter new angives det, hvad den anonyme klasse arver fra, eller et interface, der implementeres (i dette tilfælde X). Man kan ikke definere en konstruktør til en anonym klasse (den har altid standardkonstruktøren). Angiver man nogle parametre ved new X(), er det parametre til superklassens konstruktør.

Fordelen ved anonyme klasser er, at det tillades på en nem måde at definere et specialiseret objekt præcis, hvor det er nødvendigt − det kan være meget arbejdsbesparende.

Da adapterklasser er så små og ofte kun skal bruges et enkelt sted, definerer man dem ofte som anonyme klasser.

Eksempel: Anonym RunnableAdapter

Det følgende eksempel gør det samme, men i stedet for at bruge OpgaveRunnableAdapter anvendes en anonym klasse, der implementerer Runnable og kalder videre i Opgave.

```
public class BenytAnonymAdapter
{
  public static void main(String[] args)
  {
    final Opgave opgave = new Opgave();

    Runnable r = new Runnable()
    {
      public void run()                      // kræves af Runnable
      {
        opgave.udfør();
      }
    };
    Thread t = new Thread(r);
    t.start();
  }
}
```

17.2.3 Anonyme adaptere til at lytte efter hændelser

Vi har brugt masser af en bestemt slags adapter−klasser i vores programmering: Til at lytte efter hændelser og udføre noget bestemt, når hændelsen skete. F.eks.:

```
  private TextArea t1, t2;
  private Button kopierKnap;
  ...

  kopierKnap.addActionListener(new java.awt.event.ActionListener() {

    public void actionPerformed(ActionEvent e) {
      String s = t1.getText();
      t2.setText(s);
    }
  });
```

Her er den anonyme klasse en Adapter, der implementerer interfacet ActionListener, der er kendt af klienten (kopierKnap), og formidler adgang til klasser, der ikke er kendt af klienten (TextArea t1 og t2).

17.2.4 Eksempel: Få data til at passe ind i en JTable

Eksempelvis kunne det være, at vi havde en liste af Kunde−objekter (defineret i afsnit 16.8), som vi ønskede at vise på skærmen i en JTable (beskrevet i afsnit 6.3.2).

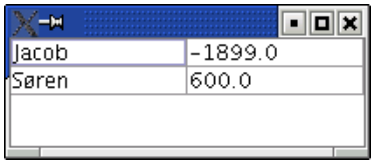
JTable ved selvfølgelig ikke, hvordan den skal vise Kunde−objekter − de "passer" ikke umiddelbart i en JTable. Nu kunne vi naturligvis lave programmet om, sådan at det var bedre indrettet til JTable, eller vi kunne kopiere indholdet af Kunde−objekterne over i en datastruktur som JTable kunne genkende.

En smartere løsning ville være at lave en adapterklasse, der "passer" ind i JTable (implementerer TableModel eller arver fra AbstractTableModel eller DefaultTableModel), og som bruger den eksisterende Kunde−liste.

```
import java.util.*;
import javax.swing.table.*;

public class KundelisteTableModelAdapter extends AbstractTableModel
{
  private List liste;

  public KundelisteTableModelAdapter(List liste1) { liste = liste1; }

  public int getRowCount() { return liste.size(); }

  public int getColumnCount() { return 2; }  // navn og kredit

  public String getColumnName(int kol)
  {
    return kol==0 ? "Navn" : "Kredit";
  }

  public Object getValueAt(int række, int kol)
  {
    Kunde k = (Kunde) liste.get(række);
    return kol==0 ? k.navn : ""+k.kredit;
  }
}
```

KundelisteTableModelAdapter er en Adapter, der implementerer interfacet TableModel (gennem klassen AbstractTableModel), der er kendt af klienten (JTable), og formidler adgang til klasser, der ikke er kendt af klienten (listen af Kunde−objekter).

Her er et eksempel på brug af KundelisteTableModelAdapter:




```
import java.util.*;
import javax.swing.*;


public class BenytKundelisteTableModelAdapter
{
  public static void main(String arg[])
  {
    // Opret liste
    List liste = new ArrayList();
    liste.add( new Kunde( "Jacob", -1899) );
    liste.add( new Kunde( "Søren",   600) );

    // Opret vindue med tabel
    JFrame vindue = new JFrame();
    JTable tabel = new JTable();
    vindue.getContentPane().add(tabel);

    // Lad tabel vise liste v.hj.a. adapteren
    tabel.setModel( new KundelisteTableModelAdapter( liste ));

    // vis vindue
    vindue.setSize(200,100);
    vindue.setVisible(true);
  }
}
```

17.2.5 Ikke−eksempel: Adapter−klasserne

Uheldigvis er der i Java også nogle klasser, der hedder adapter−klasserne. *Ingen af disse klasser er eksempler på designmønstret Adapter*!

Disse klasser tjener i stedet til at lette programmørens arbejde, når han skal implementere et hændelseslytter−interface, der har mere end en metode. I stedet for at implementere interfacet direkte arver man fra en såkaldt adapter−klasse, der har tomme implementationer for alle metoderne.

Eksempelvis, i stedet for at implementere MouseListener:

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

public class Linjelytter implements MouseListener
{
  public void mousePressed(MouseEvent hændelse)  // kræves af MouseListener
  {
    System.out.println("Der blev trykkket med musen!");
  }
  //-------------------------------------------------------------------
  //  Ubrugte hændelser (skal defineres for at implementere MouseListener)
  //-------------------------------------------------------------------
  public void mouseReleased(MouseEvent hændelse){} // kræves af MouseListener
  public void mouseClicked(MouseEvent event) {}  // kræves af MouseListener
  public void mouseEntered (MouseEvent event) {} // kræves af MouseListener
  public void mouseExited (MouseEvent event) {}  // kræves af MouseListener
}
```

kan man arve fra MouseAdapter (der implementerer MouseListener med tomme metoder):

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

public class Linjelytter2 extends MouseAdapter
{
  public void mousePressed(MouseEvent hændelse)
  {
    System.out.println("Der blev trykkket med musen!");
  }
}
```

Tilsvarende med de andre klasser i java.awt.event, der ender på Adapter (ComponentAdapter, FocusAdapter, KeyAdapter, MouseAdapter, MouseMotionAdapter og WindowAdapter): Ingen af dem er en Adapter i designmønster−henseende.

17.3 Iterator

Problem: Du er i gang med at lave et system, som andre (klienter) skal anvende, hvor de skal kunne gennemløbe dine data. Du ønsker ikke, at de skal kende noget til, hvordan data er repræsenteret i dit system (f.eks. antal elementer eller deres placering i forhold til hinanden).

Løsning: Definér et hjælpeobjekt (en Iterator), som klienten kan bruge til at gennemløbe data i dit system.

> En Iterator er et hjælpeobjekt beregnet til at gennemløbe data

En Iterator har som minimum:

- en metode til at spørge, om der er flere elementer, og
- en metode til at hente næste element

En Iterator bruges i stedet for en tællevariabel. Fordelen ved at definere en Iterator er, at klienten *ikke behøver at vide noget om strukturen af de data, der gennemløbes*.

17.3.1 Iteratorer i Collections−klasserne

Iteratorer er flittigt brugt i Collections−klasserne, idet alle datastrukturerne ligefrem har metoden iterator(), der giver et objekt, der itererer gennem elementerne.

Interfacet Iterator ser således ud:

```
package java.util;

public interface Iterator {
  boolean hasNext();  // om der er flere elementer
  Object next();      // hent næste element
  void remove();
}
```

Man kan f.eks. gennemløbe en Vector (eller en anden af Collections−klasserne) med

```
Vector samlingAfData;

// indsæt nogle strenge i v
...

Iterator i = samlingAfData.iterator();
while (i.hasNext())
{
  String s = (String) i.next();
  ...
}
```

eller med en for−løkke:

```
for (Iterator i = v.iterator(); i.hasNext(); )
{
  String s = (String) i.next();
  ...
}
```

Fordi vi bruger en Iterator, er vi afskærmet fra strukturen af de data, der gennemløbes. Data behøver f.eks. ikke have et bestemt indeks eller rækkefølge som i en Vector. Ovenstående eksempler virker lige så godt med en List eller Set (en mængde).

Bemærk, at klassen Iterator, som den er defineret i Collections−klasserne, har noget, man normalt ikke forbinder med en iterator, nemlig metoden remove(), der fjerner det aktuelle element fra den underliggende samling af data.

17.3.2 Definere sin egen form for iterator

Selvom forskellige former for iteratorer bruges flittigt i standardbiblioteket, vil man nok sjældent komme ud i at definere sin egen form for Iterator, da interfacet java.util.Iterator dækker langt de flestes behov.

> Man vil meget sjældent selv definere en ny form for Iterator, med mindre man er i gang med at programmere et programbibliotek

17.3.3 Iteratorer i JDBC

Når vi behandler svaret på en forespørgsel til en database, sker det ved at iterere gennem svar−tabellen række for række.

Det gøres med et ResultSet−objekt. Interfacet ResultSet ser således ud:

```
package java.sql;

public interface ResultSet {
  boolean next() throws SQLException;
  boolean previous() throws SQLException;

  boolean isFirst() throws SQLException;
  boolean isLast() throws SQLException;
  boolean first() throws SQLException;
  boolean last() throws SQLException;

  int getRow() throws SQLException;
  boolean absolute( int row ) throws SQLException;
  boolean relative( int rows ) throws SQLException;
```

```
  ... mange andre metoder til bl.a. at aflæse/opdatere rækker
}
```

Eksempel på brug:

```
// forespørgsler

ResultSet rs = stmt.executeQuery("SELECT navn, kredit FROM kunder");
while (rs.next())
{
  String navn = rs.getString("navn");
  double kredit = rs.getDouble("kredit");
  System.out.println(navn+" "+kredit);
}
```

Her er metoderne til at spørge, om der er flere elementer, og til at hente næste element slået sammen til én metode, nemlig next().

Vi (klienten) behøver ikke at vide noget om, hvordan data er repræsenteret, og heller ikke om alle data hentes på én gang eller lidt efter lidt efterhånden som vi kalder next().

17.3.4 Iterator til at gennemløbe geometriske figurer

De funktioner, Java har til at manipulere med todimensionale geometriske figurer (Java2D − beskrevet i kapitel 5) i pakken java.awt.geom, baserer sig kraftigt på iteratorer.

Der findes disse grundlæggende geometriske figurer:

- Rektangler − Rectangle2D og RoundRectangle2D
- Linjer − Line2D (en ret linje), CubicCurve2D (en linje, der er buet efter et ankerpunkt) og QuadCurve2D (en linje, der er buet efter to ankerpunkter)
- Ellipse2D og Arc2D (buestykke)

Fælles for dem alle er, at de består af (buede eller rette) linjestykker (en firkant består f.eks. af fire rette linjestykker), og alle figurerne kan, selvom de er ret forskelligt repræsenteret indeni, returnere en Iterator til at gennemløbe linjestykkerne i figuren.

Denne iterator ser således ud:

```
package java.awt.geom;

public interface PathIterator
{
  // om iterationen er nået gennem alle linjestykkerne
  public boolean isDone();

  // gå til næste linjesegment
  public void next();

  // lægger data får det aktuelle linjestykke ind i variablen segment
  public int currentSegment(double[] segment);

  ... flere metoder
}
```

Når Java2D skal kombinere flere geometriske figurer til en ny figur, sker det ved, at den gennemløber figurernes linjesegmenter v.hj.a. iteratoren og derpå opbygger det kombinerede geometriske objekt.

Det sker f.eks. i klassen GeneralPath, der repræsenterer en vilkårlig geometrisk figur, der f.eks. kan bygges op ved at kombinere andre geometriske figurer:

```
  GeneralPath figur = new GeneralPath();
  figur.append( new Line2D.Float(0, 0, 100, 100), false );
  figur.append( new CubicCurve2D.Float(0, 0, 80, 15, 10, 90, 100, 100), false );
  figur.append( new Arc2D.Float(−30, 0, 100, 100, 60, −120, Arc2D.PIE), false );
```

Hver gang append() bliver kaldt med en figur, findes først en PathIterator på figuren, denne gennemløbes derefter, og linjestykkerne føjes til GeneralPath−objektet.

17.4 Facade

Problem: Et sæt af beslægtede objekter er indviklede at bruge, og der er brug for en simpel grænseflade til dem.

Løsning: Definér et hjælpeobjekt, en *Facade*, der gør objekterne lettere at bruge.

> En Facade giver en simplificeret grænseflade til en gruppe delsystemer eller til et indviklet system

En Facade er altså et objekt, der giver en "brugergrænseflade" til nogle andre objekter og dermed forenkler brugen af disse objekter.

17.4.1 Eksempel: URL

I pakken java.net er klassen URL en Facade for en række andre klasser, der kan håndtere en lang række protokoller, bl.a. HTTP, FTP, e−post og lokale filer (se eksempler i afsnit 18.4.3).

Men for klienten er URL−klassen ekstrem nem at bruge, f.eks. kan klienten hente en hjemmeside ned i en datastrøm med:

```
  URL u = new URL("http://java.sun.com/");
  InputStream is = u.openStream();
  ...
```

Facaden skjuler hele processen for os og letter os dermed fra byrden med at forstå, hvordan klasserne og kommunikationen fungerer inde bagved: Internt bruger URL et InetAddress−objekt til at repræsentere værtsmaskinens IP−adresse, og den bruger et URLStreamHandler−objekt til at håndtere protokollen (i dette tilfælde HTTP−protokollen). Dette URLStreamHandler−objekt fabrikerer en URLConnection, og URL kalder videre i dette URLConnection−objekt, når vi kalder openStream().

Se afsnit 12.7 for en beskrivelse af netværksklasserne i standardbiblioteket.

17.4.2 Eksempel: Socket og ServerSocket

Læser man dokumentationen til klasserne Socket (der repræsenterer en forbindelse til en bestemt maskine over netværket på en bestemt port og som klienter bruger til at forbinde sig til værtsmaskiner med) og ServerSocket (der repræsenterer en port på en værtsmaskine der er åben for indkommende forbindelser), finder man ud af, at de begge faktisk er facader til klassen SocketImpl.

Denne konstruktion skyldes, at det på styresystemets niveau er næsten de samme kald, der skal ske for en Socket og en ServerSocket, og de varetages derfor af den samme klasse (SocketImpl). Socket og en ServerSocket bruger altså begge SocketImpl til at varetage den egentlige netværkskommunikation.

Designerne til Javas standardbibliotek har anvendt designmønstret Facade for at gøre det simplere at lave netværkskommunikation: De har delt funktionaliteten i SocketImpl op i to letforståelige klasser (Socket og en ServerSocket) til de to måder, den kan bruges på.

17.5 Observatør/Lytter

Problem: Et objekt skal kunne underrette nogle andre objekter om en eller anden ændring eller hændelse, men det er ikke hensigtsmæssigt, at objektet kender direkte til de andre objekter.

Løsning: Lad lytterne (observatørerne) implementere et fælles interface (eller arve fra en fælles superklasse) og registrere sig hos det observable (observerbare) objekt. Det observable objekt kan herefter underrette lytterne gennem interfacet, når der er brug for det.

Designmønstret Observatør (eng.: Observer) kaldes også Abonnent (eng.: Publisher−Subscriber) eller lytter (eng.: Listener).

17.5.1 Eksempel: Hændelser

Det mest kendte eksempel på Observatør−designmønstret er hændelseshåndteringen i grafiske brugergrænseflader.

Når man vil lytte efter musehændelser, opretter man en klasse, der implementerer MouseListener−interfacet (observatøren):

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

public class Muselytter implements MouseListener
{
  public void mousePressed(MouseEvent hændelse)  // kræves af MouseListener
  {
    Point trykpunkt = hændelse.getPoint();
    System.out.println("Mus trykket ned i "+trykpunkt);
  }

  public void mouseReleased(MouseEvent hændelse)  // kræves af MouseListener
  {
    Point slippunkt = hændelse.getPoint();
    System.out.println("Mus sluppet i "+slippunkt);
  }

  public void mouseClicked(MouseEvent hændelse)  // kræves af MouseListener
  {
    System.out.println("Mus klikket i "+hændelse.getPoint());
  }

  //-----------------------------------------------------------------
  //  Ubrugte hændelser (skal defineres for at implementere MouseListener)
  //-----------------------------------------------------------------
  public void mouseEntered (MouseEvent event) {}  // kræves af MouseListener
  public void mouseExited (MouseEvent event) {}  // kræves af MouseListener
}
```

Man skal registrere lytteren (observatøren) ved at kalde metoden addMouseListener(lytter) på den grafiske komponent, der sender hændelserne (den observable):

```
import java.applet.*;
public class LytTilMusen extends Applet
{
  public LytTilMusen()
  {
    Muselytter lytter = new Muselytter();
    this.addMouseListener(lytter);  // tilføj lytteren til er appletten selv
  }
}
```

Når man senere klikker med musen, bliver metoder i lytteren kaldt.

17.5.2 Eksempel: Kalender

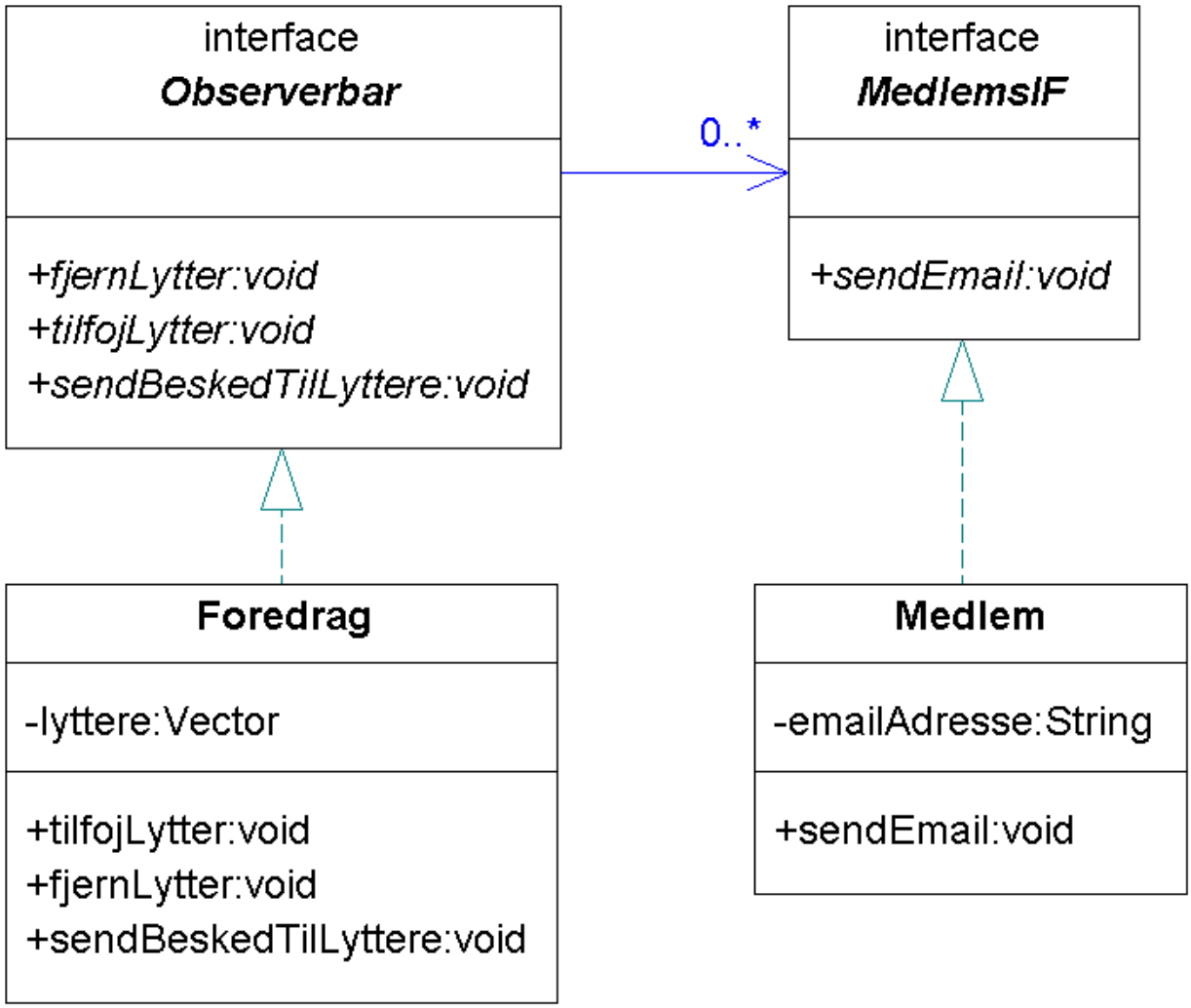
Forestil dig et program, der skal fungere som en fælles kalender for en forening, der udbyder forskellige foredrag.

Kalenderen indeholder en liste over foredragene, og medlemmerne af foreningen kan tilmelde sig de forskellige foredrag efter ønske.

Hvis der sker ændringer i planen, skal kun de interesserede medlemmer have besked, dvs. de, som har tilmeldt sig et givent foredrag.

Det kan implementeres ved at lave et foredragsobjekt, som indeholder en liste af registrerede lyttere. Objektet har metoder til at tilføje og fjerne lyttere og til at løbe listen igennem og sende beskeder, når det er nødvendigt.

Her er en skitse til et klassediagram:



17.6 Dynamisk Binding

Problem: Programmet skal senere kunne udvides til at bruge nogle flere klasser, uden at programmet skal skrives om.

Løsning: Definér et fælles interface (eller superklasse) for klasserne, og søg efter egnede klasser på køretidspunktet, indlæs dem og brug dem.

Indlæs klasser dynamisk under kørslen

Dynamisk Binding (eller Dynamisk Lænkning − eng.: Dynamic Linkage) går ud på at indlæse klasser dynamisk, *efter* at programmet er startet. Det er en meget kraftfuld mekanisme der tillader "plug−ins", dvs. at programdele kan føjes til programmet, efterhånden som der er behov for dem, og som måske er produceret af nogle helt andre end dem, der oprindeligt skrev programmet.

I Java indlæses en klasse dynamisk med et kald til Class.forName(), der tager en streng med et klassenavn som parameter.

F.eks. indlæser man Vector−klassen og opretter et objekt med:

```
  Class klassen = Class.forName("java.util.Vector");
  Object objektet = klassen.newInstance();
```

Dette forudsætter, at klasserne findes der, hvor systemet plejer at lede (ellers kan man definere sin egen ClassLoader som beskrevet i afsnit 11.4.3, Indlæse klasser fra filsystemet).

Herunder er beskrevet to konkrete eksempler på Dynamisk Binding. Et tredje eksempel (URL−klassen) diskuteres i afsnit 18.4.5.

17.6.1 JDBC og Dynamisk Binding

JDBC anvender Dynamisk Binding til at håndtere drivere for de forskellige databaser. Det tillader enhver databaseleverandør at levere drivere, og de vil umiddelbart passe ind i JDBC.

Kigger man i pakken java.sql (se javadokumentationen), kan man få en idé om, hvordan det gøres. Herunder er en forsimplet udgave af implementationen af JDBC, der illustrerer princippet (eksemplet kan ikke køres, det skal mere ses som en illustration af JDBC's brug af Dynamisk Binding).

DriverManager har en intern liste af drivere. Når DriverManager.getConnection() kaldes, kalder den en metode i hver af sine indlæste drivere for at finde en, der passer. Drivere, der ikke passer til den URL, der beskriver forbindelsen, signalerer dette ved at returnere null.

Driverne har på forhånd kaldt DriverManager.registerDriver() og registreret sig selv.

```
import java.sql.*;
import java.util.*;

public class DriverManager
{
  private static Vector drivere = new Vector();

  public static Connection getConnection(String url) throws SQLException
  {
    // Gå gennem alle de indlæste drivere og forsøg at lave en forbindelse

    for (int i = 0; i < drivere.size(); i++)
    {
      Driver d = (Driver) drivere.elementAt(i);

      Connection result = d.connect(url);
      if (result != null) {
        // Success!
        return result;
      }
    }

    // kommer hertil var der ingen drivere der kunne klare forbindelsen
    throw new SQLException("No suitable driver");
  }

/**
 * Drivere kalder denne metode når de bliver indlæst, for at registrere sig selv
 */
  public static void registerDriver(Driver driver)
  {
    drivere.addElement(driver);
  }

  // ... flere metoder
}
```

Alle drivere skal implementere interfacet Driver.

```
import java.sql.*;

public interface Driver
{
/**
 * Når DriverManager.getConnection() kaldes, kalder den denne metode i hver
 * af sine indlæste drivere, for at finde en driver der passer.
 * @return Forbindelse til databasen, eller null hvis denne driver ikke passer
 * @param url Adressen på databasen, f.eks. "jdbc:odbc:datakilde1"
 * eller "jdbc:oracle:thin:@ora.javabog.dk:1521:student"
 */

  Connection connect(String url) throws SQLException;
```

```
  // ... flere metoder
}
```

Driver−klasserne skal også sørge for, når de bliver indlæst, at registrere sig hos DriverManager. Lad os forestille os, at vi vil skrive en driver til databasen Xyz, som skal genkende URL'er på formen "jdbc:xyz:...". Hvis den får en passende URL, returnerer den et XyzConnection−objekt, ellers null.

```
import java.sql.*;

public class XyzDriver implements Driver
{
  // Klasseinitialiseringsblok − køres én gang når klassen indlæses
  static
  {
    Driver drv = new XyzDriver();
    DriverManager.registerDriver( drv );
  }

  public Connection connect(String url) throws SQLException
  {
    // passer URLen til min driver?
    if (url.startsWith("jdbc:xyz:"))
    {
      // kode til at oprette et passende Connection−objekt
      return new XyzConnection(url);
    }

    // hvis URL'en ikke startede med "jdbc:xyz:" så returner null
    // og så vil en anden driver blive forsøgt.
    return null;
  }

  // ... flere metoder
}
```

17.6.2 Eksempel: Fortolkning af matematikfunktioner

Vi forestiller os, at et program til at tegne kurver dynamisk skal kunne udvides med flere funktioner. Funktioner, implementerer alle interfacet Funktion:

```
public interface Funktion
{
  public double beregn(double x);
}
```

Alle klasserne har en navngivning, der tillader dem at blive dynamisk indlæst: De hedder alle Funktion_ og så navnet, f.eks. repræsenterer klassen Funktion_sin funktionen sinus:

```
public class Funktion_sin implements Funktion
{
  public double beregn(double x)
  {
    return Math.sin(x);
  }
}
```

Funktions−fortolkeren indlæser klasser, som implementerer Funktion−interfacet ud fra funktionens navn dynamisk:

```
public class FunktionsfortolkerDynBind
{
  public Funktion findFunktion(String navn)
  {
    try
    {
      // Prøv at indlæse en klasse der hedder f.eks. Funktion_sin
      Class klasse =   Class.forName("Funktion_"+navn);

      // Opret et objekt fra klassen
      Funktion f = (Funktion) klasse.newInstance();

      return f;
    }
    catch (Exception ex)
    {
      ex.printStackTrace();
      throw new IllegalArgumentException("ukendt funktion: "+navn);
    }
  }

  public Funktion fortolk(String udtryk)
  {
    // endnu ikke implementeret − returner bare noget.
    return findFunktion("sin");
  }
}
```

Klientprogrammer kalder analyser() med en streng og får en tilsvarende Funktion ud. Programmet kan senere udvides med f.eks. Funktion_cos, Funktion_tan o.s.v.[1].

Her er et eksempel på brug:

```
public class BenytFunktionsfortolkerDynBind
{
  public static void main(String arg[])
  {
    FunktionsfortolkerDynBind analysator = new FunktionsfortolkerDynBind();
    Funktion f = analysator.fortolk("sin(5*cos(x))");
    System.out.println("f(1)=" + f.beregn(1) );
  }
}
```

17.7 Opgaver

Proxy

Læs afsnit 16.8.3, Dataforbindelse, der cacher forespørgsler, og lav med, udgangspunkt i eksemplet, proxy−klassen Dataforbindelseslogger, der skriver ud til skærmen hver gang der kaldes en metode på dataforbindelsen.

Dynamisk Binding

1. Metoden findFunktion() i FunktionsfortolkerDynBind kan gøres mere effektiv ved at huske de allerede indlæste Funktion−klasser, sådan at en funktion kun bliver indlæst én gang.
   Udvid fortolkeren med en afbildning (en HashMap), der afbilder allerede indlæste funktionsnavne (strenge) over i de tilsvarende klasser.
2. Udvid Funktionsfortolker fra afsnit 4.7.2 til at bruge dynamisk binding.

Designmønstre i JDBC

Kig i afsnit 8.1, Basisfunktioner i JDBC. Hvilke designmønstre kan du se, der er anvendt i JDBC−biblioteket, ud fra beskrivelsen?

1. Nævn 2 fabrikeringsmetoder.
2. Nævn mindst 2 andre designmønstre anvendt i JDBC, og beskriv dem.

Svarene findes i næste afsnit.

17.8 Løsninger

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

17.8.1 Dataforbindelseslogger

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

17.8.2 Designmønstre i JDBC

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

1Dette eksempel kan dog ikke tage højde for sammensatte funktioner som f.eks. sin(5*x+1), da Funktion−objekter ikke kan kombineres. Se afsnit 4.7.2 for et eksempel, der tager højde for sammensatte funktioner.

18 Andre designmønstre

18.1 Uforanderligt objekt (Immutable)

18.1.1 Uforanderlige objekter i standardbiblioteket

18.1.2 Eksempel − en uforanderlig punkt−klasse

18.1.3 Opgave

18.2 Fluevægt

18.2.1 Eksempel: Strenge

18.2.2 Eksempel: Dele UforanderligtPunkt−objekter

18.3 Filter

18.3.1 Eksempel: Kunde−filtre

18.3.2 Eksempel: Streng−filtre

18.3.3 Opgave

18.3.4 Eksempel: Output− og InputStream−klasserne

18.3.5 Avanceret: Læse vs. skrive data

18.4 Lagdelt Initialisering

18.4.1 Simpelt eksempel

18.4.2 Lagdelt initialisering vs. fabrikeringsmetode

18.4.3 Eksempel: URL−klassen

18.4.4 Avanceret lagdelt initialisering

18.4.5 Avanceret lagdelt initialisering og dynamisk binding i URL−klassen

18.5 Komposit/Rekursiv komposition

18.5.1 Analogi til byggeindustrien

18.5.2 Eksempel: Component/Container

18.5.3 Eksempel: Funktioner

18.5.4 Eksempel: Gruppering af figurer

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

18.6 Kommando

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

18.6.1 Eksempel

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

18.6.2 Avanceret: Variationer af designmønstret

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

18.6.3 Opgaver

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

18.7 Opgave: Designmønstre i standardbibliotekerne

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

18.7.1 Løsning: Designmønstre i standardbiblioteket

Dette afsnit er ikke omfattet af Åben Dokumentslicens.
Du skal købe bogen for at måtte læse dette afsnit. Jeg erklærer, at jeg allerede har købt bogen
Jeg lover at anskaffe den i nær fremtid.

1Variablerne s1 og s3 peger to forskellige steder hen i hukommelsen, så s1!=s3, men de to strengobjekter, s1 og s3 peger på, indeholder den samme tekst, så udtrykket s1.equals(s3) er sandt.

2Det er dog ikke så relevant i dette eksempel, da beton nok aldrig skal ændres til at bestå af andre materialer end vand, sten, sand og cement. I et program, der skulle kunne håndtere flere kompositmaterialer, ville man nok omdøbe klassen til at hedde SammensatMateriale og så lave en nedarving, der hed Beton.

19 Model−View−Controller−arkitekturen

Det er en god idé at kigge i dette kapitel før man læser kapitel 6, Grafiske brugergrænseflader (Swing).

De fleste programmer med en brugergrænseflade kan inddeles i tre dele, nemlig:

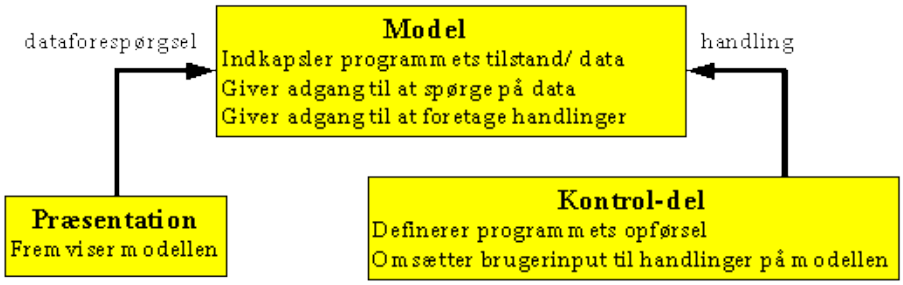
1. *datamodellen*, som repræsenterer data og de bagvedliggende beregninger
2. *præsentationen* af data over for brugeren
3. brugerens mulighed for at *ændre* i disse data gennem forskellige handlinger.

Ofte præsenteres brugeren ikke for alle data, måske kan han ikke ændre dem frit, og måske er der konsekvenser for andre data i det samme eller i andre skærmbilleder.

19.1 De tre dele af MVC

Model−View−Controller−arkitekturen (forkortet MVC) er et designmønster beregnet til programmer med en brugergrænseflade.

Den anbefaler at man opdeler programmet (i hvert fald mentalt) i tre dele: En model, en præsentation, og en kontrol−del:



Model−View−Controller−arkitekturen i dens grundform. Pilene viser, hvilke dele der kender til hverandre (der kan være flere − se afsnit 19.2.2 og frem).

19.1.1 Modellen

Datamodellen *indeholder data og registrerer, hvilken tilstand den pågældende del af programmet er i*. Oftest er data indkapslet sådan, at konsistens sikres. I så fald er der kun adgang til at spørge og ændre på data gennem metodekald.

Modellen bør være uafhængig af, hvordan data præsenteres over for brugeren, og er der flere programmer, der arbejder med de samme slags data, kan de i princippet have den samme datamodel, selvom de i øvrigt er helt forskellige.

Eksempel: En bankkonto har navn på ejer, kontonummer, kort−ID, saldo, bevægelser, renteoplysninger etc. Saldoen kan ikke ændres direkte, men med handlingerne overførsel, udbetaling og indbetaling kan saldoen påvirkes (se eksempelvis klassen Kontomodel i afsnit 19.3.1).

Bemærk, hvordan modellen for en bankkonto er universel. Modellen kunne f.eks. anvendes både i et program til en pengeautomat, i et netbank−system og i programmet, som ekspeditionsmedarbejderen anvender ved skranken.

19.1.2 Præsentationen

Præsentationen (eng.: View) *henter relevante data fra modellen og viser dem for brugeren i en passende form*. Selvom to præsentationer deler model (viser data fra samme model), kan de være meget forskellige, da de er beregnet på en bestemt brugergrænseflade.

Eksempel: Bankkontoen præsenteres meget forskelligt. I en pengeautomat vises ingen personlige oplysninger overhovedet. I et netbank−system kan saldo og bevægelser ses (det kunne være en webløsning i HTML, f.eks. en servlet eller JSP−side). Ved skranken kan medarbejderen se endnu mere, f.eks. filial og kontaktperson i banken (det kunne være implementeret som en grafisk applikation, der kører hos brugeren).

19.1.3 Kontrol−delen

Kontroldelen (eng.: controller) definerer, hvad programmet kan. Den *omsætter brugerens indtastninger, museklik mv. til handlinger, der skal udføres på modellen*.

Eksempel: I pengeautomat kan man kun hæve penge. I et netbank−system kan brugeren måske lave visse former for overførsel fra sin egen konto. Ved skranken kan medarbejderen derudover foretage ind− og udbetalinger.

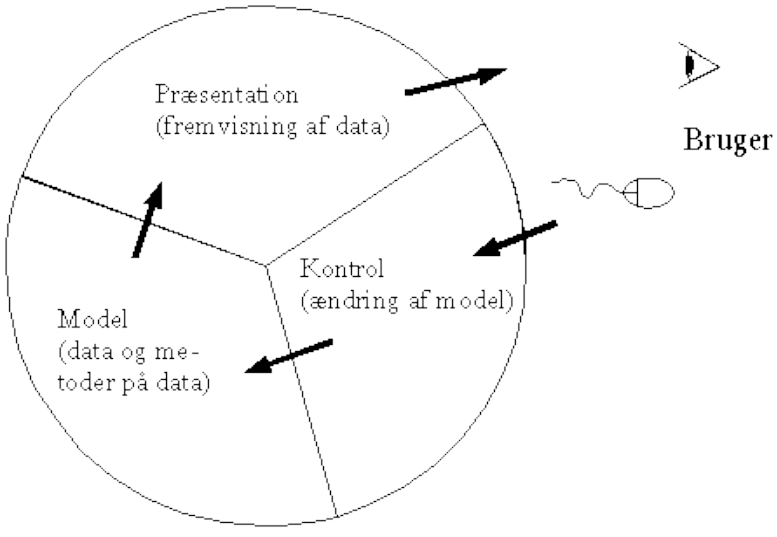
19.2 Relationer mellem delene

Forestil dig, at modellen, præsentationen og kontroldelen udgøres af hver sin klasse. Hvad er så relationerne mellem klasserne?

Det er klart, at præsentationen og kontroldelen, for at kunne fremvise hhv. manipulere med modellen, skal kende til modellen og dens metoder. Hvilke andre bindinger er der?

19.2.1 Informationsstrøm gennem MVC

Figuren herunder illustrerer, hvordan strømmen af information går fra modellen via præsentationen til brugeren (symboliseret ved et øje). Brugeren foretager nogle handlinger (symboliseret ved musen), som via kontrol−delen fortolkes som nogle ændringer, der foretages på modellen, hvorefter de nye data vises for brugeren.



19.2.2 Opdatering af præsentationen − tre muligheder

Præsentationen er normalt[1] nødt til på en eller anden måde at få at vide, når der er sket en ændring i modellens data, så den kan opdatere skærmbilledet.

Det kunne ske på tre måder: Enten må præsentationen regelmæssigt undersøge modellen for at opdage ændringer, eller også må kontroldelen eller modellen fortælle præsentationen, at noget er ændret. Lad os se på alle tre muligheder.

19.2.3 A − Præsentationer undersøger modellen

Hvis præsentationen regelmæssigt skal undersøge model−klassen for at opdage ændringerne, vil ændringerne naturligvis dukke op på brugerens skærm med en vis forsinkelse, der kan virke forvirrende for brugeren. Forsinkelsen kan naturligvis mindskes ved at

undersøge modellen meget ofte (dette kaldes polling), men det er en ineffektiv løsning, der kræver meget processortid.

Denne mulighed er dog velegnet i de tilfælde, hvor der skal ske opdateringer af skærmen så ofte som overhovedet muligt (f.eks. i et computerspil, hvor animationerne skal være så flydende som muligt). Her vil præsentationen konstant undersøge modellen (for at tegne skærmbilledet), og ændringer vil derfor blive synlige næsten omgående.

I andre tilfælde ligger det i sagens natur, at der *altid* sker en fremvisning lige efter en opdatering. Det gælder f.eks. webløsninger som servletter/JSP beskrevet i afsnit 14.2, Webservere (servletter og JSP), hvor netlæseren altid foretager en anmodning (der indeholder formulardata, som kontroldelen omsætter til kald til modellen) og får HTML−koden til et nyt skærmbillede tilbage som svar (genereret af præsentationen ud fra modellen).

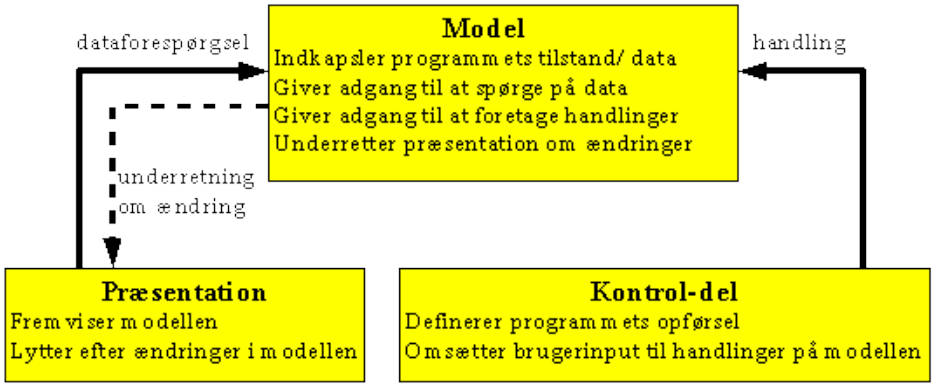
19.2.4 B − Kontroldel underretter præsentationer

En oplagt mulighed er, at kontroldelen, efter hver ændring på modellen, underretter præsentationen om, at noget er ændret (og dermed opdaterer skærmen).

Det er en fin løsning til mindre systemer, men hvad nu hvis der er flere præsentationer (og kontroldele) af den samme model? For at holde visningen over for brugeren korrekt skal hver kontroldel holde styr på samtlige præsentationer. Hver gang der kommer en ny præsentation til, skal kontroldelene opdateres til at medtage den. I et lidt større scenario kan det kan give et ret uoverskueligt program.

19.2.5 C − Modellen underretter præsentationer

En anden, mere avanceret (men i længden mere enkel) løsning er at lægge opdateringsopgaven hos modellen, sådan at den fortæller det til præsentationen (og andre interessenter), når den ændres:



MVC med præsentationer, som observerer modellen − den oftest brugte variant af MVC.

Da modellen skal være uafhængig af præsentationerne (f.eks. for at modellen kan genbruges med en anden præsentation i et andet program), kan den nødvendigvis ikke vide noget om præsentationerne direkte.

I stedet observerer præsentationerne modellen på samme måde som i Javas hændelsessystem: Præsentationerne registrerer sig som lyttere hos modellen, og modellen sender en hændelse til alle registrerede lytter når den ændres. Den stiplede linje (underretning om ændring) illustrerer dette.

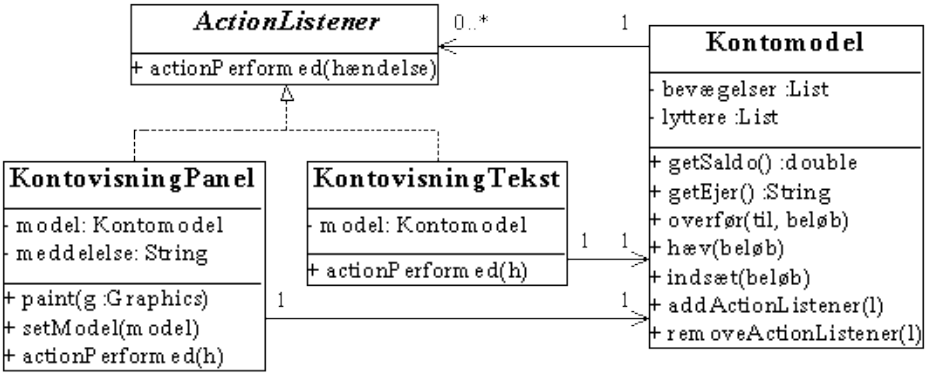
Dette er designmønstret Observatør (beskrevet i afsnit 17.5) og kan også udtrykkes i dette designmønsters ordvalg: Præsentationerne er observatører, der observerer modellen. Modellen underretter sine observatører, når den ændres.

19.3 Eksempel − bankkonti

Da den mere avancerede mulighed C stemmer overens med Javas hændelsesmodel og er den mest generelle, er det den, der oftest tages i anvendelse. Mulighed B er velegnet til simplere systemer.

Her følger et tænkt eksempel på en bankkonto. Hver konto har en ejer, og man kan indsætte, hæve og overføre penge til en anden konto.

Mulighed C er valgt her, dvs. præsentationer skal registrere sig hos modellen (ActionEvent er anvendt som hændelse, så det vil sige, at lyttere skal implementere ActionListener−interfacet).



19.3.1 Modellen

Modellen har metoder, der svarer til forretningslogikken i programmet, såsom overfør(), hæv() og indsæt().

Derudover har den metoderne addActionListener() og removeActionListener() til at registrere lyttere (observatører) på den.

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

public class Kontomodel
{
  private String ejer;
  private double saldo;
  private List bevægelser = new ArrayList();         // til historik

  public Kontomodel(String ejer1) { ejer = ejer1; }

  public double getSaldo() { return saldo; }
  public String getEjer() { return ejer; }

  public String toString() { return ejer + ": "+saldo+" kr"; }

  public void overfør(Kontomodel til, double beløb)
  {
    if (beløb<0) throw new IllegalArgumentException(
                                    "Beløb kan ikke være negativt eller nul.");

    saldo = saldo - beløb;
    til.saldo = til.saldo + beløb;// privat variabel kan ses i samme klasse

    String ændring = "Overført "+beløb+" fra "+ejer+" til "+til.ejer;
    bevægelser.add(ændring);
    til.bevægelser.add(ændring);

    fortælLyttere(ændring);       // besked til alle visninger af denne konto
    til.fortælLyttere(ændring);   // besked til alle visninger af beløbsmodtager
  }

  public void hæv(double beløb)
  {
    if (beløb<0) throw new IllegalArgumentException(
                                    "Beløb kan ikke være negativt eller nul.");
    saldo = saldo - beløb;

    String ændring = "Hævet "+beløb;
    bevægelser.add(ændring);
    fortælLyttere(ændring);       // send besked til alle visninger
  }

  public void indsæt(double beløb)
  {
    if (beløb<0) throw new IllegalArgumentException(
                                    "Beløb kan ikke være negativt eller nul.");
    saldo = saldo + beløb;

    String ændring = "Indsat "+beløb;
    bevægelser.add(ændring);
    fortælLyttere(ændring);       // Send besked til alle visninger
  }

  //
  // Underetning af hændelses-lyttere.
  //

  /** Lyttere til denne model */
   private List lyttere = new ArrayList(2);

  /** Tilføjer en lytter */
  public synchronized void addActionListener(ActionListener l)
  {
    lyttere.add(l);
```

```
  }

  /** Fjerner en lytter */
  public synchronized void removeActionListener(ActionListener l)
  {
    lyttere.remove(l);
  }

  /** Fortæller alle lytter om en ændring i modellen */
  private void fortælLyttere(String ændring)
  {
    // opret hændelse, der beskriver ændringen
    ActionEvent hændelse = new ActionEvent(this, 0, ændring);
    for (Iterator i=lyttere.iterator(); i.hasNext(); )
    {
      ActionListener l = (ActionListener) i.next();
      l.actionPerformed(hændelse);// underret l om hændelsen
    }
  }
}
```

19.3.2 Præsentationer

Herunder to eksempler på præsentationer af modellen. Den første er ret enkel, da den 'viser' modellen ved at udskrive på System.out, hver gang der sker en ændring:

```
import java.awt.event.*;
public class KontovisningTekst implements ActionListener
{
  private Kontomodel model;

  public KontovisningTekst(Kontomodel model1)
  {
    model = model1;
    model.addActionListener(this);        // registrér som lytter på modellen
  }

  public void actionPerformed(ActionEvent hændels)
  {
    // getActionCommand() giver beskrivelsen af hændelsen
    System.out.println("Konto "+model.getEjer()+": "+hændels.getActionCommand());
    System.out.println("Konto "+model.getEjer()+": Saldo er: "+model.getSaldo());
  }
}
```

Uddata fra denne klasse kunne være:

```
Konto Jacob: Indsat 20.0
Konto Jacob: Saldo er nu: 20.0
Konto Jacob: Indsat 20.0
Konto Jacob: Saldo er nu: 40.0
Konto Jacob: Overført 50.0 fra Jacob til Brian
Konto Jacob: Saldo er nu: −10.0
```

Den anden er et grafisk panel (i figuren til højre er det vist i et applet−vindue). Ved hver ændring i modellen bliver panelet gentegnet.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class KontovisningPanel extends JPanel implements ActionListener
{
```


```
  private Kontomodel model;
  private String meddelelse;

  public void paint(Graphics g)
  {
    super.paint(g);
    if (model == null) return;
    g.drawString("Konto "+model.getEjer(),10,10);

    if (meddelelse != null)
    {
      g.drawString(meddelelse,10,25);
      // næste gang der gentegnes skal meddelelsen ikke vises
      meddelelse = null;
    }

    if (model.getSaldo()<0) g.setColor(Color.red);
    else g.setColor(Color.darkGray);
    g.drawString("saldo: "+model.getSaldo(),10,40);
  }
```

```
  public void setModel(Kontomodel model1)
  {
    if (model != null) model.removeActionListener(this);
    model = model1;
    if (model != null) model.addActionListener(this);     // lytter på modellen
  }

  public void actionPerformed(ActionEvent hændelse)
  {
    meddelelse = hændelse.getActionCommand();
    repaint();
  }
}
```

19.3.3 Kontroldel

Herunder har vi lavet en applet, der opretter en Jacob og en Brian−kontomodel. I appletten er der et antal KontovisningPanel'er, der viser kontiene, og nogle knapper, der kan ændre på kontiene (disse knapper udgør kontrol−delen af programmet).


```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class KontokontrolApplet extends JApplet
{
  Kontomodel jacobsKonto = new Kontomodel("Jacob");
  Kontomodel briansKonto = new Kontomodel("Brian");

  KontovisningPanel panelJacob = new KontovisningPanel();
  KontovisningPanel panelBrian = new KontovisningPanel();
  KontovisningPanel panelJacobAndenVisning = new KontovisningPanel();

  JButton buttonJtilB50kr = new JButton();
  JButton buttonJ20krind = new JButton();
  JButton buttonB30krud = new JButton();

  public void init() {
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
    panelJacob.setModel(jacobsKonto);       // panelJacob lytter på jacobsKonto
    panelJacobAndenVisning.setModel(jacobsKonto); // ditto
    panelBrian.setModel(briansKonto);       // panelBrian lytter på briansKonto

    new KontovisningTekst(jacobsKonto);     // tekstvisning på jacobsKonto
    new KontovisningTekst(briansKonto);     // tekstvisning på briansKonto

    JFrame f = new JFrame("Brian");         // lav også separat vindue til Brian
    KontovisningPanel panelBrianAndenVisning = new KontovisningPanel();
    panelBrianAndenVisning.setModel(briansKonto);

    f.getContentPane().add(panelBrianAndenVisning);
    f.setSize(150,100);
    f.validate();
    f.setVisible(true);
  }

  private void jbInit() throws Exception {
    this.getContentPane().setLayout(null);
    panelJacob.setBounds(new Rectangle(0, 0, 139, 102));
    panelBrian.setBounds(new Rectangle(250, 0, 150, 101));
    panelJacobAndenVisning.setBounds(new Rectangle(130, 105, 140, 84));
    buttonJtilB50kr.setText("50 kr ->");
    buttonJtilB50kr.setBounds(new Rectangle(140, 33, 100, 25));

    // når der affyres en hændelse fra buttonJtilB50kr, så kald
    // metoden buttonJtilB50kr_actionPerformed, defineret nedenfor
    buttonJtilB50kr.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        buttonJtilB50kr_actionPerformed(e);
```

```
      }
    });

    buttonJ20krind.setText("Indsæt 20 kr");
    buttonJ20krind.setBounds(new Rectangle(6, 104, 124, 25));
    buttonJ20krind.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        buttonJ20krind_actionPerformed(e);
      }
    });

    buttonB30krud.setText("Hæv 30 kr");
    buttonB30krud.setBounds(new Rectangle(272, 105, 124, 25));
    buttonB30krud.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        buttonB30krud_actionPerformed(e);
      }
    });
    this.getContentPane().add(panelBrian, null);
    this.getContentPane().add(panelJacob, null);
    this.getContentPane().add(buttonJtilB50kr, null);
    this.getContentPane().add(buttonJ20krind, null);
    this.getContentPane().add(buttonB30krud, null);
    this.getContentPane().add(panelJacobAndenVisning, null);
  }

  // kontroldelen af programmet - ændr i modellen efter brugerens handlinger
  void buttonJ20krind_actionPerformed(ActionEvent e) {
    jacobsKonto.indsæt(20);
  }

  void buttonB30krud_actionPerformed(ActionEvent e) {
    briansKonto.hæv(30);
  }

  void buttonJtilB50kr_actionPerformed(ActionEvent e) {
    jacobsKonto.overfør(briansKonto,50);
  }
}
```

19.4 Model−View − "den lille MVC"

Adskillelsen mellem præsentation og kontroldel kan være besværlig eller uhensigtsmæssig at gennemføre i praksis.

Det gælder for eksempel, når man programmerer grafiske brugergrænseflader i Java: Her er det hensigtsmæssigt at lægge præsentation (paint()−metode og grafiske komponenter) og kontrol−del (hændelseslyttere på komponenterne) i samme klasse.

19.4.1 Adskillelse af præsentation og programlogik

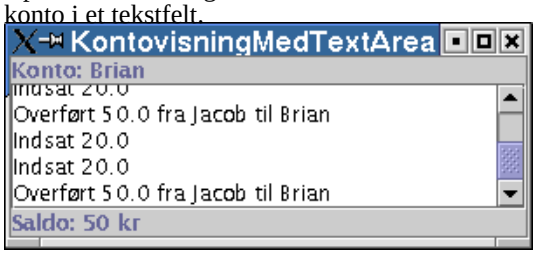
Derfor begrænses MVC undertiden til "Model−View"−arkitekturen, hvor præsentation og kontrol−del er slået sammen.

Model−View−arkitekturen er idéen om at adskille præsentation og programlogik, sådan at programlogikken (modellen) kan genbruges i andre sammenhænge.

I denne begrænsede udgave af MVC vil der ofte kun være én præsentation af data, og man kan derfor se bort fra diskussionen i afsnit 19.2.2 om opdateringen af præsentationen.

19.5 Opgaver

1. Hent Konto−eksemplet fra http://javabog.dk/VP/kode/. Prøv at køre det.
   Hvordan kan det være, at visningerne af Kontomodel bliver opdateret?
2. Opret KontovisningMedTekstArea, der er en klasse (JPanel eller JFrame), der viser (og opdaterer) en beskrivelse af en konto i et tekstfelt.



19.5.1 Løsning

1I nogle tilfælde bliver præsentationen *altid* kaldt efter en opdatering. Det gælder f.eks. JSP−sider og servletter.

Ønsker du at se de sidste 29% af dette værk (362838 tegn) skal du købe bogen. Så får du pæne figurer og layout, stikordsregister og en trykt bog med i købet.