Introduksjon til databaseprogrammering med Java

Kompendium for kurs i objektorientert programmering

Bjørn Kristoffersen

Avdeling for allmenne fag
Institutt for økonomi og informatikk

Høgskolen i Telemark
November 2004

HiT skrift nr 7/2004

ISBN 82-7206-238-0 (trykt)
ISBN 82-7206-239-9 (online)

ISSN 1501-8539 (trykt)
ISSN 1503-3767 (online)

Serietittel: *HiT skrift* eller *HiT Publication*

Høgskolen i Telemark
Postboks 203
3901 Porsgrunn

Telefon 35 57 50 00
Telefaks 35 57 50 01
http://www.hit.no/

Trykk: Kopisenteret. HiT-Bø

Sammendrag

Relasjonsdatabaser og objektorientert programmering er viktige teknologier for utvikling av dagens informasjonssystemer. Typisk så blir brukergrensesnitt og beregningsregler programmert med objektorienterte språk, mens permanent lagring av data gjøres i relasjonsdatabaser. Populært kan vi si at filosofien bak objektorientert programmering er å bygge systemer som en samling av ”kommuniserende” objekter. I relasjonsdatabaser blir data lagret som relasjoner. Det blir dermed vesentlig å forstå hvordan objekter og relasjoner kan brukes sammen.

Kompendiet tar utgangspunkt i datastrukturer som det er naturlig å bruke for å representere relasjoner i et objektorientert programmeringsspråk. Deretter blir det forklart hvordan slike datastrukturer kan lagres til og leses fra sekvensielle tekstfiler. Denne delen gir nødvendige kunnskaper for å programmere enkle applikasjoner som kan lagre strukturerte data på ytre lager *uten* bruk av en relasjonsdatabase.

Siste del av kompendiet tar for seg databaseprogrammering i Java ved bruk av klassebiblioteket Java DataBase Connectivity (JDBC). Ved hjelp av ferdiglagede metoder i JDBC og det standardiserte spørrespråket SQL blir det vist hvordan man kan utvikle enkle Java-applikasjoner som jobber mot relasjonsdatabaser i henhold til en klient/tjener-arkitektur. Det blir også kort gjort rede for hvordan man kan utvikle web-løsninger basert på databaser og Java.

Kompendiet egner seg som støttelitteratur til kurs i objektorientert programmering. Det er forutsatt at studentene har kjennskap til relasjonsdatabaser og SQL, samt at de har tilegnet seg grunnleggende ferdigheter i programmering med Java.

Nettsiden home.hit.no/~kristoff inneholder komplett programkode til eksemplene som blir omtalt.

Takk til Tor Lønnestad og Roy M. Istad som har lest utkast til kompendiet og sørget for en rekke forbedringer.

Bø, 25. november 2004

Bjørn Kristoffersen

Nøkkelord: objektorientert programmering, Java, relasjonsdatabaser, SQL, datastrukturer, filbehandling, databaseprogrammering, JDBC.

Innholdsfortegnelse

Introduksjon

Relasjonsdatabaser og *objektorientert programmering* er viktige teknologier for utvikling av dagens informasjonssystemer. Typisk så blir brukergrensesnitt og beregningsregler programmert med objektorienterte språk, som for eksempel *Java*, mens permanent lagring av data gjøres i relasjonsdatabaser. Populært kan vi si at filosofien bak objektorientert programmering er å bygge systemer som en samling av "kommuniserende" *objekter*. I relasjonsdatabaser blir data lagret som *relasjoner*[1], eller "databasetabeller" om man vil. Det blir dermed vesentlig å forstå hvordan objekter og relasjoner kan brukes sammen.

De fleste applikasjoner har behov for å lagre data over tid, fra en utførelse til den neste. I praksis innebærer det at data blir lagret på én av to måter:

- Enkeltstående filer
- Databasesystemer

Dette kompendiet forklarer hvordan Java-programmer kan jobbe med data på ytre lager, først data lagret på fil og deretter i database.

For å behandle, og eventuelt oppdatere data på ytre lager, må data først bli lest inn i internhukommelsen, og organisert i en passende datastruktur. Hvis datastrukturen blir oppdatert må den skrives ut igjen til ytre lager før programmet avslutter.

Et program som jobber mot en enkelt fil kan være bygget opp slik:

1. Les inn data fra filen og bygg opp en intern datastruktur.
2. Utfør beregninger som avleser og eventuelt oppdaterer datastrukturen.
3. Skriv oppdatert datastruktur tilbake til filen.

Det kan være nødvendig å gjenta innlesing og lagring flere ganger i løpet av en programutførelse. I et databasesystem blir data kontinuerlig skyflet fram og tilbake mellom internhukommelsen og ytre lager.

Alle data i en relasjonsdatabase blir logisk sett organisert i relasjoner. Vi fokuserer dermed på slike strukturer, og viser hvordan en relasjon kan representeres som en samling Java-objekter i henhold til et bestemt mønster.

Pakken java.io inneholder en rekke klasser for programmering mot filer. Vi tar kun for oss enkle tekstfiler som blir lest og skrevet sekvensielt, linje for linje. Filene vi bruker som eksempler kan betraktes som fysiske relasjoner, i den forstand at et databasesystem i prinsippet kan lagre data på denne måten.

Med *databaseprogrammering* mener vi de teknikkene man bruker for å lagre, endre og gjenfinne data i en database. Databaseprogrammering bygger på at programmeringsspråket, som her er Java, kan kommunisere med databasesystemet. *SQL* er et standardisert *spørrespråk* som de aller fleste databasesystemer støtter. En typisk kommunikasjonsform vil være at Java-programmet sender en *SQL-spørring* til databasesystemet, som utfører spørringen og sender resultatet tilbake til Java-programmet for videre beregninger og presentasjon. Program-biblioteket *JDBC* (Java DataBase Connectivity) inneholder klasser med tilhørende metoder for å gjøre denne formen for programmering enklere. JDBC oversetter mellom måten data blir representert i relasjonsdatabaser og i Java.

---

[1] For å unngå sammenblanding av begreper bruker vi ordet *relasjon* for databasetabell, og reserverer *tabell* for Java-tabeller.

Den første delen av kompendiet demonstrerer hvordan man kan lage sin egen database ved hjelp av tekstfiler og standard Java datastrukturer, og er skrevet som et bakteppe for å forstå databaseprogrammering med JDBC.

Kompendiet blir avsluttet med noen tips for videre studier innen databaseprogrammering med Java, med referanser til relevant litteratur.

Relasjoner og objekter

Java-bibliotekene inneholder flere klasser for å representere såkalte *objektsamlinger*, se for eksempel pakken java.util. Vi skal imidlertid fokusere på bruk av *tabeller* (arrays). Datastrukturene som blir beskrevet kan brukes for å representere både relasjoner og resultatet av SQL utvalgspørringer (SELECT-setninger).

En *tabell* består av en sekvens *elementer* av samme datatype. Elementene aksesseres ved deres *posisjon*, eller *indeks*, i tabellen. Elementene kan være verdier av primitive datatyper som int og boolean, men også objekter av ferdigdefinerte eller egendefinerte klasser.

Tabeller og objekter kan kombineres på forskjellige måter. For eksempel kan vi bygge opp en tabell av objekter, der hvert objekt igjen inneholder en heltallstabell.

Tabell 1 viser en relasjon Ansatt med fire *attributter* AnsattNr, Fornavn, Etternavn og Telefon. Den inneholder data om fem ansatte i en bedrift. Hver ansatt er representert ved et *tuppel*, eller en *rad*, som tilordner verdier til hvert attributt i relasjonen.

Hvis vi velger Java-tabeller som datastruktur kan relasjoner representeres på to forskjellige måter:

- Som et antall ”attributt-tabeller”: Relasjonen Ansatt kan representeres i fire tabeller, der elementene inneholder enkle verdier (tall og tekst).
- Som en tabell av ”rad-objekter”: Dette vil for relasjonen Ansatt gi én tabell, der hvert element blir et ”sammensatt” objekt som representerer en ansatt med fire attributter.

Vi skal kort gjøre rede for begge teknikker.

AnsattNr	Fornavn	Etternavn	Telefon
101	Bjørn	Krogfoss	37 94
113	Hans Petter	Fosen	37 98
117	Andrea	Børsheim	37 54
122	Lene	Nilssen	37 55
125	Fredrik	Wilhelmsen	37 89

Tabell 1. Relasjonen Ansatt

Merk for øvrig at det *ikke* er hensiktsmessig å representere relasjoner som *flerdimensjonale Java-tabeller*, fordi alle elementene i en (flerdimensjonal) tabell må være av samme datatype.

Verditabeller og referansetabeller

I en *verditabell* er elementene verdier[2] av de primitive datatypene som int, boolean og float. Attributtet AnsattNr kan representeres ved en heltallstabell, slik:

[2] Strengt tatt burde vi skrive *primitive* verdier, og ikke bare verdier, ettersom objektreferanser også er verdier i Java.

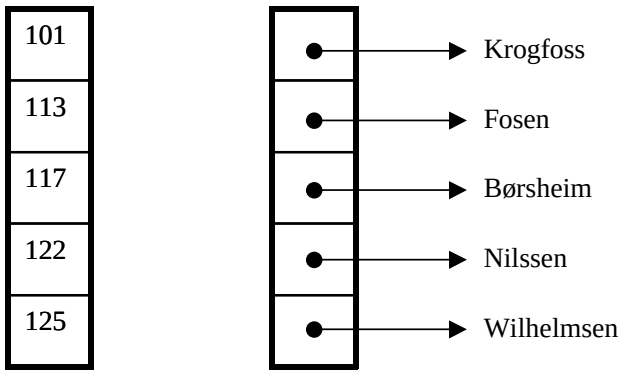
```
int[] ansattNrTab = { 101, 113, 117, 122, 125 };
```

I en *referansetabell* er elementene *objektreferanser*. Med referanser menes at tabellen ikke inneholder selve objektene, men adressene til der objektene er lagret i hukommelsen[3].

En tekststreng er egentlig et objekt av klassen String. Attributtet Etternavn kan dermed representeres som en referansetabell:

```
String[] etternavnTab =
  {"Krogfoss", "Fosen", "Børsheim", "Nilssen", "Wilhelmsen"};
```

Forskjellen på verditabeller og referansetabeller er illustrert i Figur 1, der pilene antyder objektreferanser.



Figur 1. Verditabeller og referansetabeller

Representasjon av relasjoner

Den første strategien nevnt over innebærer å opprette en tabell for hvert attributt. For relasjonen Ansatt får vi følgende klasse:

```
public class AnsattSamling extends Object
{
  private int[]    ansattNrTab;
  private String[] fornavnTab;
  private String[] etternavnTab;
  private String[] telefonTab;

  // Definisjon av metoder er utelatt.
}
```

Tanken er at informasjon om en bestemt ansatt legges i én og samme posisjon i alle fire tabeller. Hvis ansattNrTab[2] inneholder 117, så bør altså fornavnTab[2] inneholde referanse til ”Andrea”, etternavnTab[2] inneholde referanse til ”Børsheim” og telefonTab[2] inneholde referanse til ”37 54”.

Relasjoner kan altså representeres ved en samling ”attributt-tabeller”. Det er mulig å programmere på denne måten, men vi skal se at det blir mer elegant å bruke en tabell med

[3] Generelt er det mulig å tenke seg både tabeller som inneholder referanser til primitive verdier og tabeller som inneholder objekter, men i Java er det kun mulig å definere tabeller som inneholder primitive verdier og tabeller som inneholder objektreferanser.

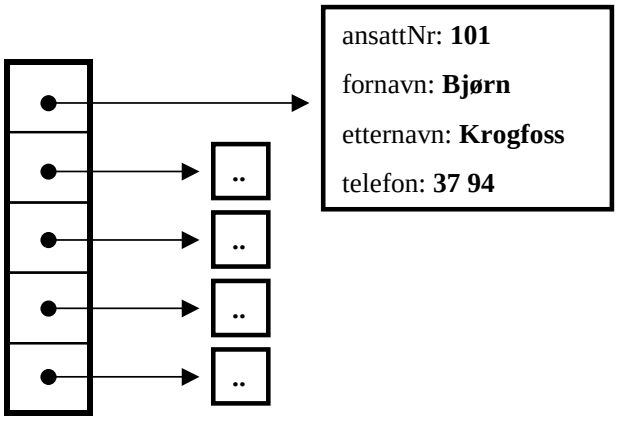
referanser til rad-objekter i stedet. For å få til dette må vi først finne en måte å representere en enkelt rad på. I vårt eksempel svarer en rad til en bestemt ansatt. Vi lager dermed en klasse Ansatt med én instansvariabel for hvert attributt:

```
public class Ansatt extends Object
{
  private int ansattNr;
  private String fornavn;
  private String etternavn;
  private String telefon;

  public Ansatt( int a, String f, String e, String t )
  {
    ansattNr = a;
    fornavn = f;
    etternavn = e;
    telefon = t;
  }

  // Øvrige metoder er utelatt.
}
```

Objekter av klassen Ansatt kan opprettes på denne måten:

```
Ansatt enAnsatt =
  new Ansatt( 101, "Bjørn", "Krogfoss", "37 94" );
```

Hele relasjonen kan deretter representeres som en tabell med referanser til Ansatt-objekter. Figur 2 illustrerer denne datastrukturen. Kun det første rad-objektet er tegnet komplett med innhold.



Figur 2. Tabell med referanser til rad-objekter

Vi vil få behov for et antall operasjoner på denne relasjonen, for eksempel for å sette inn og søke etter ansatte. Det er hensiktsmessig å samle koden i en egen klasse med selve tabellen som en instansvariabel, slik:

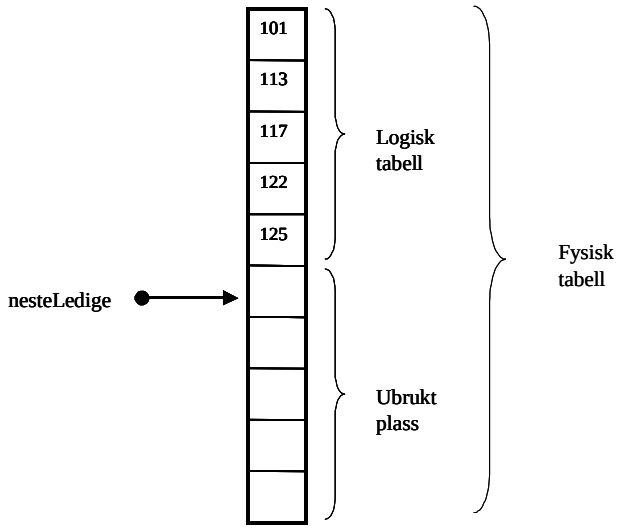
```
public class AnsattSamling extends Object
{
  private Ansatt[] tabell;

  // Metoder og øvrige instansvariable er utelatt.
}
```

Fysisk og logisk tabell

Når vi oppretter en Java-tabell må vi spesifisere størrelsen i form av antall elementer. Under utførelse vil det bli satt av plass i hukommelsen til tabellen, og selv før vi eksplisitt har satt inn data må tabellen nødvendigvis inneholde *noe*. Hva dette er varierer med hva slags datatype elementene i tabellen har. I verditabeller vil alle elementene bli satt til å inneholde en standardverdi. For eksempel så vil samtlige elementer i en heltallstabell inneholde 0. Elementene i en referansetabell vil derimot inneholde null.

Det er ofte hensiktsmessig å la tabeller inneholde ledig plass. Tenk på en applikasjon der brukeren registrerer data om ansatte i et interaktivt skjermbilde, og der klassen AnsattSamling blir brukt som datastruktur. Ved oppstart kan tabellen være "tom", men etter hvert blir den fylt med referanser til Ansatt-objekter. På et gitt tidspunkt vil noen elementer i tabellen peke på objekter, og resten av tabellen ikke være i bruk. Den delen av tabellen som er i bruk kaller vi for *den logiske tabellen*, og vil generelt utgjøre bare en del av *den fysiske tabellen*.

Den fysiske tabellen vil inneholde et fast antall elementer. Den logiske tabellen, derimot, vil øke og avta i størrelse i takt med innsettinger og slettinger. Verdiene utenfor den logiske tabellen skal ikke brukes av programmet. Vi må dermed holde rede på grensene for den logiske tabellen, og innfører en instansvariabel nesteLedige som viser til neste ledige plass i tabellen. I konstruktøren blir nesteLedige satt til 0, som betyr at den logiske tabellen er tom. Elementene fra og med posisjon 0 til og med posisjon nesteLedige-1 avgrenser den logiske tabellen. Forholdet mellom logisk og fysisk tabell er illustrert for en heltallstabell i Figur 3.



Figur 3. Fysisk tabell og logisk tabell

Merk at Figur 3 forutsetter at vi jobber med *pakkede* tabeller, der sletting medfører reorganisering av elementene. Uten reorganisering vil det oppstå "hull" i den logiske tabellen. For referansetabeller er det naturlig å sette inn *null* for slike elementer. For verditabeller er det ikke opplagt hvordan man skal representere fravær av verdier. Vi skal kun jobbe med pakkede tabeller, og lar dermed dette problemet ligge.

Klassen AnsattSamling utvidet med instansvariabelen nesteLedige og konstruktør blir slik:

```
public class AnsattSamling extends Object
{
  private Ansatt[] tabell;
  private int nesteLedige;

  public AnsattSamling( int antall )
  {
    tabell = new Ansatt[antall];
    nesteLedige = 0;
  }

  // Øvrige metoder er utelatt.
}
```

I konstruktøren for AnsattSamling blir det allokert plass til tabellen. Vi kan styre maksimalt antall elementer med en parameter. Følgende setning (skrevet i en annen klasse) fører til at det blir opprettet en tabell med plass til 100 objektreferanser:

```
AnsattSamling ansatte = new AnsattSamling( 100 );
```

En generell ”oppskrift” for å representere relasjoner i Java blir dermed:

1. Lag en klasse for å representere en enkelt rad.
2. Lag en klasse for å representere en samling av rad-objekter.

Java-tabeller er *statiske* datastrukturer i den forstand at antall elementer er bestemt når vi oppretter en tabell. Det betyr at vi får et problem ved forsøk på innsetting hvis tabellen allerede er full (når logisk tabell er lik fysisk tabell). Det finnes en rekke måter å løse dette på, blant annet så finnes det *dynamiske* datastrukturer, som ikke har en fast øvre grense for antall elementer. Det vil imidlertid føre for langt å gjøre rede for slike teknikker her. Vi skal nøye oss med å oppdage at tabellen er full, og sørge for at programmet kan gi brukeren beskjed.

Operasjoner på datastrukturene

Så langt har vi kun antydet en datastruktur for representasjon av relasjoner. For ajourhold og søk i slike datastrukturer er det nødvendig å innføre metoder både til de enkelte rad-objektene og til objektet som representerer hele relasjonen.

Hent- og sett-metoder

Innkapsling er et grunnleggende prinsipp i objektorientert programmering. Et objekt inneholder instansvariable (data) og metoder (operasjoner). I tråd med prinsippet om innkapsling bør instansvariable bli deklarert private. Det medfører at man fra andre klasser ikke kan avlese eller modifisere slike variable direkte. All aksess må i stedet gjøres ved å kalle metoder.

For å tilby objekter av andre klasser kontrollert tilgang til instansvariable er det vanlig å innføre såkalte *hent*- og *sett-metoder*. En hent-metode returnerer verdien til en instansvariabel. En sett-metode tar en ny verdi som parameter og tilordner verdien til tilhørende instansvariabel. Under er dette vist for instansvariabelen telefon.

```
public class Ansatt extends Object
{
  private String telefon;
  // Øvrige instansvariable er utelatt.

  public String hentTelefon()
  {
    return telefon;
  }

  public void settTelefon( String nyTelefon )
  {
    telefon = nyTelefon;
  }

  // Øvrige metoder er utelatt.
}
```

Merk at returdatatypen til hentTelefon og også datatypen til den formelle parameteren til settTelefon samsvarer med datatypen til instansvariabelen telefon. Dette vil normalt gjelde for alle sett- og hent-metoder.

Følgende kode viser hvordan metodene kan brukes utenfor klassen Ansatt:

```
Ansatt enAnsatt =
  new Ansatt( 101, "Bjørn", "Krogfoss", "37 94" );
enAnsatt.settTelefon( "35 99" );
System.out.println( "Nytt telefonnr: " +
                    enAnsatt.hentTelefon() );
```

Mange utviklingsverktøy vil generere slike metoder basert på deklarasjonen av instansvariablene, men vil typisk innføre engelske prefikser[4] til metodenavnene: getTelefon og setTelefon.

Nyttige databaseoperasjoner

De ”klassiske” databaseoperasjonene er *innsetting*, *oppdatering* og *sletting*, samt ulike typer av *søk*. Vi skal programmere eksempler på slike operasjoner som metoder til klassen AnsattSamling.

Et grunnleggende designvalg for AnsattSamling er om tabellen skal være sortert (etter et bestemt kriterium), eller ikke. For å holde tabellen sortert får vi litt mer arbeid ved innsetting, oppdatering og sletting, men kan til gjengjeld utføre søk mer effektivt. Vi velger her en enkel løsning uten sortering, men gir noen tips til hvordan mer effektive løsninger kan programmeres.

Håndtering av feilsituasjoner er et annet viktig designvalg. Det er neppe hensiktsmessig at klassen AnsattSamling skriver ut feilmeldinger. Dette er en oppgave for brukergrensesnittet til et program. Vi velger dermed å la noen av metodene returnere en boolsk verdi som viser om operasjonen gikk bra eller ikke.

Alle relasjoner skal ha en *primærnøkkel*. En primærnøkkel er ett eller flere attributter som identifiserer en rad i relasjonen. Det skal altså ikke eksistere to rader med samme verdi i primærnøkkelen. For relasjonen Ansatt er AnsattNr et naturlig valg som primærnøkkel. Metoder som modifiserer datastrukturen i klassen AnsattSamling bør sjekke at det ikke oppstår repetisjoner i slike attributter. Det betyr blant annet at vi ved innsetting av nye ansatte må sjekke at ansattnummeret ikke finnes allerede.

Hjelpemetode for søk

Det viser seg at flere av metodene vi skal programmere må utføre søk i tabellen. Både for å slette og oppdatere et Ansatt-objekt må vi opplagt først finne det. Også innsetting krever faktisk søk i vårt eksempel fordi vi må sjekke at det ikke oppstår repetisjoner i primærnøkkelen. Det viser seg at vi kan gjenbruke koden for søk ved å innføre en hjelpemetode finnPos som finner posisjonen til et rad-objekt, gitt en søkeverdi. Metoden blir definert som privat, og kan altså ikke brukes utenfor klassen AnsattSamling.

```
// Finner posisjonen til et Ansatt-objekt, gitt
// ansattnummer. Returnerer -1 hvis objektet ikke finnes.
private int finnPos( int ansNr )
{
  boolean funnet = false;
  int pos = 0;
  while ( pos<nesteLedige && !funnet )
  {
    if ( ansNr == tabell[pos].hentAnsattNr() )
      funnet=true;
    else
      pos = pos+1;
  }
```

[4] En *Java-bønne* (JavaBean) er en Java-klasse som kan spille rollen som en ”komponent”. Egenskapene til komponenten er definert ved set- og get-metodene. Man må altså bruke engelske prefikser for å lage Java-bønner.

```
    if ( funnet )
      return pos;
    else
      return -1;
  }
```

Legg merke til at while-løkka blir stoppet på en av to måter: enten ved at vi har lest forbi det siste elementet i tabellen, eller ved at vi har funnet det vi leter etter. I det første tilfellet returnerer vi -1, som er en ”kode” som betyr at søket var mislykket. Hvis søket lykkes inneholder variabelen pos posisjonen til elementet med ansattnummer lik parameteren ansNr. Flere av de andre metodene i klassen vil starte med å kalle metoden finnPos.

Generelt finnes det mange forskjellige måter å søke i tabeller på. Metoden finnPos implementerer det vi kaller for et *sekvensielt søk*, som rett og slett gjennomløper tabellen fra start til vi enten finner det vi leter etter, eller til vi er kommet til slutten av (den logiske) tabellen. Sekvensielt søk er lett å programmere, men svært ineffektivt sammenlignet med andre teknikker. Forutsatt at vi alltid finner det vi leter etter må vi i gjennomsnitt sammenligne søkeparameteren (ansNr) med n/2 verdier, der n er antall elementer i tabellen.

Hvis tabellen hadde vært sortert med hensyn på ansattnummer, kunne vi brukt en teknikk som kalles for *binærsøk*. Tanken er her at vi sammenligner søkeparameteren ansNr med ansattnummeret til det midterste elementet i tabellen. Hvis ansNr er mindre fortsetter vi med samme søkemetode i den delen av tabellen som befinner seg til "venstre" for midten, og hvis ansNr er større fortsetter vi med den "høyre" delen. Ved likhet har vi funnet det vi leter etter. Dette vil halvere *søkerommet* i hvert steg, og det vil være tilstrekkelig å sammenligne ansNr med $\log_2 n$ verdier. For en tabell med 1024 elementer så vil sekvensielt søk i gjennomsnitt kreve 512 sammenligninger, mens binærsøk ikke vil kreve mer enn 10 sammenligninger (fordi $2^{10}=1024$). Tips for programmering av binærsøk: Innfør to hjelpevariable som avgrenser den delen av tabellen som blir undersøkt. Lag en løkke som i hvert steg flytter enten nedre grense eller øvre grense til midten, og fortsetter helt til de to variablene "møtes", eller at vi har funnet det vi leter etter.

Søk

Fordi det ikke finnes to like verdier i en primærnøkkel, vil søk etter en bestemt verdi i AnsattNr returnere maksimalt ett objekt. Vi kan dermed lage en metode finn med Ansatt som returdatatype. Metoden kan returnere null hvis det ikke finnes en ansatt med det ansattnummeret vi søker etter. Det er naturlig å bruke hjelpemetoden finnPos for å utføre søket. Det eneste vi må gjøre i tillegg er dermed å sjekke om søket gikk bra og hente ut selve objektet, eller eventuelt returnere null.

```
// Finner ansatt, gitt ansattnummer.
// Returnerer null hvis objektet ikke finnes.
public Ansatt finn( int ansNr )
{
  int pos = finnPos( ansNr );
  if ( pos >= 0 )
    return tabell[pos];
  else
    return null;
}
```

Det vil være ønskelig å tilby søk i andre attributter enn primærnøkkelen. Generelt vil slike metoder kunne gi flere treff. Det kan for eksempel eksistere flere ansatte med samme etternavn. Programmeringsteknisk betyr det at søkemetodene må returnere en datastruktur som kan inneholde mange objekter. En elegant løsning tilpasset vårt eksempel vil være å returnere et objekt av klassen AnsattSamling. Nøyaktig den samme teknikken blir brukt i klassebiblioteket JDBC som vi skal omtale seinere, der klassen ResultSet brukes for å representere både tabeller og spørreresultater.

Innsetting

En metode for innsetting bør ta det nye rad-objektet som parameter. Det er naturlig å la metoden returnere en boolsk verdi som forteller om operasjonen lot seg gjennomføre eller ikke. Det kan jo tenkes at tabellen allerede er full, og at vi av den grunn må avvise nye innsettinger. Fordi vi skal representere en relasjon må vi dessuten sørge for at kravet til primærnøkler er oppfylt. Sjekk på unike primærnøkler gjøres ved kall på hjelpemetoden finnPos.

Det er mulig å tenke seg flere strategier for hvor i tabellen nye elementer skal plasseres. Det aller enkleste er å organisere rad-objektene i den rekkefølgen de settes inn, som betyr at nye elementer blir satt inn i posisjonen som variabelen nesteLedige inneholder. Hver innsetting må øke instansvariabelen nesteLedige med 1.

```
// Setter inn en ansatt bakerst. Returnerer false hvis det
// oppstår repetisjoner i primærnøkkelen, eller det ikke
// er ledig plass, og true ellers.
public boolean settInn( Ansatt a )
{
  if ( nesteLedige < tabell.length &&
       finnPos( a.hentAnsattNr() ) == -1 )
  {
    tabell[nesteLedige] = a;
    nesteLedige++;
    return true;
  }
  else
    return false;
}
```

Hvis vi ønsker å holde tabellen sortert til enhver tid må nye elementer settes inn på riktig posisjon, noe som medfører at vi først må "rydde plass". En grei måte å få til dette på er å starte bakerst (i posisjon nesteLedige-1), og flytte hvert element en posisjon bakover. Vi bør nok likevel starte med et kall på finnPos for å sikre at vi unngår repetisjoner i primærnøkkelen (før vi rydder plass).

Oppdatering

For å oppdatere innholdet i et rad-objekt må vi først finne det. Når vi har funnet riktig objekt kan vi bruke sett-metoder i klassen Ansatt for å modifisere det. Følgende kode er tenkt benyttet utenfor klassen Ansatt, der variabelen ansatte refererer et objekt av klassen AnsattSamling:

```
Ansatt enAnsatt = ansatte.finn( 125 );
if ( enAnsatt != null )
  enAnsatt.settTelefon( "35 99" );
```

Kallet på metoden finn vil returnere null hvis det ikke finnes en ansatt med ansattnummer 125. Vi må derfor teste for denne muligheten før vi oppdaterer telefonnummeret.

Det er kanskje ikke helt opplagt at oppdateringen vi gjør ved kall på metoden settTelefon har noen effekt på datastrukturen ansatte. Objektet er jo først "hentet ut" med metoden finn, og blir ikke "satt inn" igjen etterpå. Husk da at det som blir returnert fra metoden finn er en *referanse* til det samme objektet som hele tiden er en del av objektsamlingen.

Merk for øvrig at denne måten å oppdatere data om ansatte overlater kontrollen med unike primærnøkler til kode utenfor klassen AnsattSamling. Det blir dermed teknisk mulig å hente ut en ansatt, og så endre ansattnummeret til ett som allerede finnes. En løsning på dette problemet er å la ansattnummer være en egenskap som ikke lar seg endre, noe vi oppnår ved å fjerne metoden settAnsattNr.

Vi kan alternativt utvide klassen AnsattSamling med egne oppdateringsmetoder, for eksempel en metode for å endre telefonnummeret til en bestemt ansatt:

```
  // Endrer telefonnummeret til en ansatt, gitt ansattnummer
  // og nytt telefonnummer. Returnerer false hvis objektet
  // ikke finnes.
  public boolean endreTelefon( int ansNr, String nyTelefon )
  {
    Ansatt a = finn( ansNr );
    if ( a != null )
    {
      a.settTelefon( nyTelefon );
      return true;
    }
    else
      return false;
  }
```

Metoder for å oppdatere andre attributter følger det samme mønsteret.

Sletting

Også for å slette en ansatt må vi først finne objektet. Hvis vi som antydet over setter inn nye objekter bakerst bør vi ved sletting sørge for at det ikke blir "hull" i den logiske tabellen. En effektiv måte å få til dette på er å flytte det *siste* elementet til posisjonen der vi skal slette. Vi må dessuten telle ned instansvariabelen nesteLedige med 1.

Etter at vi har utført flytting og talt ned nesteLedige med 1 så vil elementet i posisjon nesteLedige ikke lenger være en del av den logiske tabellen, men fortsatt peke på det samme objektet. Logisk sett er ikke det et problem, men av effektivitetshensyn setter vi likevel elementet til å peke på null, fordi kjøresystemet[5] da kan gjenbruke plassen til noe annet og mer nyttig.

[5] Java har det vi kaller for automatisk *søppeltømming* (garbage collection), som innebærer at det under programutførelse av og til blir startet en systemprosess som finner objekter som ikke lenger er i bruk (les: blir referert), og merker plassen som "ledig", slik at den kan gjenbrukes til nye objekter.

```
// Sletter en ansatt, gitt ansattnummer.
// Returnerer false hvis objektet ikke finnes, true ellers.
public boolean slett( int ansNr )
{
  int pos = finnPos( ansNr );
  if ( pos >= 0 )
  {
    nesteLedige--;
    tabell[pos] = tabell[nesteLedige];
    tabell[nesteLedige] = null;
    return true;
  }
  else
    return false;
}
```

Merk at teknikken med å ”tette hull” i tabellen ved å flytte det siste elementet ikke kan brukes på en tabell som skal være sortert. I så fall må vi flytte samtlige elementer, posisjonert etter det elementet vi sletter, en posisjon framover i tabellen. Sammenlign for øvrig med tilsvarende diskusjon i avsnittet om innsetting.

Generaliseringer

Iteratorer

Vi har ofte behov for å behandle alle objektene i en objektsamling etter tur. Kanskje vi skal lage en rapport, eller oppdatere data om hvert enkelt objekt på en systematisk måte. I klassen AnsattSamling er det lett å definere metoder som behandler samtlige Ansatt-objekter ved en løkke som gjennomløper den logiske tabellen. Utenfor klassen AnsattSamling, derimot, er det ikke opplagt at vi kan få til det samme. Tabellen med Ansatt-objekter er jo deklarert som privat, og man kan argumentere for at det heller ikke bør lages hent- og sett-metoder for tabellen[6].

En *iterator* er en mekanisme som gjør det mulig å behandle alle objektene i en objektsamling, uten å ha tilgang til måten objektene er organisert. Iteratorer kan lettest forstås som metoder av typen "gå til første" og "gå til neste". Metodene må vedlikeholde det vi kan kalle for "nåværende posisjon". I koden under er det heltallsvariabelen iterPos som brukes til dette formålet.

```
public class AnsattSamling extends Object
{
  // Deklarasjon av andre instansvariable og metoder som før

  private int iterPos;   // Nåværende posisjon

  // Gå til første rad
  public void gåTilFørste()
  {
    iterPos = 0;
  }

  // Gå til neste rad
  public void gåTilNeste()
  {
    iterPos++;
  }

  // Returnerer false hvis vi har beveget oss utenfor
  // logisk tabell, og true ellers.
  public boolean flere()
  {
    return ( iterPos < nesteLedige );
  }
```

[6] Instansvariabelen tabell i klassen AnsattSamling har på sett og vis ingen mening utenfor klassen AnsattSamling, fordi den hører sammen med instansvariabelen nesteLedige. Til sammen danner de to variablene datastrukturen som representerer den logiske tabellen, og som bare klassen AnsattSamling bør kjenne til og få manipulere.

```
  // Returner inneværende ansatt, eller null hvis
  // vi har beveget oss forbi slutten.
  public Ansatt denne()
  {
    if ( flere() )
      return tabell[iterPos];
    else
      return null;
  }
}
```

Nå kan vi i en metode *utenfor* klassen skrive følgende, der variabelen ansatte refererer et objekt av klassen AnsattSamling:

```
ansatte.gåTilFørste();
while ( ansatte.flere() )
{
  Ansatt a = ansatte.denne();
  // Sett inn kode for å behandle a her.
  ansatte.gåTilNeste();
}
```

Gjennomløp av objektsamlinger med iteratorer blir brukt flere steder i Java-bibliotekene, blant annet i JDBC, som vi skal se på seinere.

Iteratormetodene vist over tillater kun behandling av Ansatt-objekter i den rekkefølgen de er lagret, fra start til slutt. Det er imidlertid enkelt å utvide med mulighet for navigering begge veier ved å innføre metoder gåTilSiste og gåTilForrige. Merk at vi da også må sjekke at vi ikke beveger oss forbi starten av tabellen ved kall på gåTilForrige.

Polymorfi

Vi har så langt jobbet med en relasjon som inneholder data om ansatte. Det er en god øvelse å modifisere eksempelkoden til å håndtere andre relasjoner. Selv om det er nødvendig å endre noen datatyper og noen kall på metoder i klassen som representerer rad-objektene, så vil koden i hovedsak følge det samme mønsteret. Det er naturlig å spørre seg om det er mulig å lage mer generelle datastrukturer, og på den måten slippe å programmere metoder for innsetting, søk og sletting mer enn én gang.

En *polymorf* metode er en metode som kan anvendes på verdier av mange datatyper. Java har forhåndsdefinerte polymorfe metoder. Tenk på metoden length for å finne antall elementer i en tabell. Den fungerer for alle slags tabeller.

I klassen AnsattSamling baserte vi både søk, innsetting og sletting på primærnøkkelen AnsattNr. Fordi det ikke finnes to ansatte med samme ansattnummer kan vi definere likhet på Ansatt-objekter som likhet i dette attributtet. I og med at enhver relasjon har en primærnøkkel er dette en teknikk som lar seg generalisere.

Første steg blir å innføre en generell måte å sammenligne rad-objekter. Selv om forskjellige relasjoner vil bruke forskjellige navn og datatyper på primærnøkkelen, kan alle primærnøkler konverteres til tekst. Vi skal dermed kreve at ethvert rad-objekt har en metode med navn nøkkel, som returnerer verdien til primærnøkkelen som en tekststreng. Dette gir oss en standard måte for å avgjøre om ett rad-objekt er likt et annet.

Et *grensesnitt* (interface) er en måte å beskrive krav til en klasse på. Syntaktisk er et grensesnitt bygget opp som en klasse, men ordet class er byttet ut med interface, og metodene inneholder ikke kode. Et grensesnitt som spesifiserer krav til rad-objekter kan skrives på denne måten:

```
public interface RadObjekt
{
  public String nøkkel();
}
```

For at en klasse skal oppfylle kravene i grensesnittet RadObjekt må den inneholde en definisjon av metoden nøkkel. Vi sier i så fall at klassen *implementerer* grensesnittet. For klassen Ansatt programmerer vi metoden nøkkel slik at den returnerer ansattnummeret konvertert til en tekststreng. Vi må også gi systemet beskjed om at klassen Ansatt implementerer grensesnittet RadObjekt. Det gjør vi med konstruksjonen implements i toppen av klassen.

```
public class Ansatt extends Object implements RadObjekt
{
  // Deklarasjon av instansvariable og
  // andre metoder er utelatt.

  public String nøkkel()
  {
    return Integer.toString( ansattNr );
  }
}
```

Fordi grensesnitt ikke inneholder konkrete metodedefinisjoner er det ikke mulig å lage objekter fra dem, slik vi kan fra klasser. Vi kan imidlertid bruke grensesnitt som datatyper på samme måte som klasser. Grensesnitt kan brukes som datatype til både variable, parametere og returverdier. Dette kan i første omgang virke meningsløst, men er vesentlig for å kunne lage generelle datastrukturer. Vi kan bruke datatypen RadObjekt og metoden nøkkel for å programmere en generell klasse for representasjon av relasjoner. Når denne klasssen blir instansiert og tatt i bruk vil de faktiske rad-objektene være av en klasse som implementerer grensesnittet RadObjekt, slik som Ansatt. Under følger en skisse av klassen Relasjon.

```
public class Relasjon extends Object
{
  private RadObjekt[] tabell;
  private int nesteLedige;

  public Relasjon( int antall ) { … }
  public boolean settInn( RadObjekt r ) { … }
  public RadObjekt finn( String nøkkel ) { … }
  public boolean slett( String nøkkel ) { … }

  private int finnPos( String nøkkel ) { … }
}
```

Metodene vil være nesten identiske med tilsvarende metoder i AnsattSamling. Direkte sammenligning av ansattnummer erstattes med sammenligning av returverdien fra metoden

nøkkel. For eksempel blir testen i finnPos som følger, der søkeparameteren nøkkel sammenlignes med primærnøkkelen til elementet i posisjon pos i tabellen tabell:

```
if ( nøkkel.equals( tabell[pos].nøkkel() ) )
  funnet=true;
```

Fordi klassen Ansatt implementerer grensesnittet RadObjekt, kan Ansatt-objekter brukes som elementer i klassen Relasjon. Koden under viser innsetting av to objekter, etterfulgt av et søk som henter ut det første objektet.

```
Relasjon ansatte = new Relasjon( 50 );
Ansatt enAnsatt =
  new Ansatt( 101, "Bjørn", "Krogfoss", "37 94" );
boolean ok = ansatte.settInn( enAnsatt );
enAnsatt = new Ansatt( 113, "Hans Petter", "Fosen", "37 98" );
ok = ansatte.settInn( enAnsatt );
enAnsatt = ( Ansatt ) ansatte.finn( "101" );
System.out.println( "Etternavn for 101: " +
                    enAnsatt.hentEtternavn() );
```

Merk at når vi henter ut Ansatt-objekter må vi gjøre eksplisitt *typeomforming* (casting). Returtypen til metoden finn er definert til å være RadObjekt. Som programmerere vet vi at det vi henter ut er et Ansatt-objekt, men dette må vi også fortelle kompilatoren for å få lov til å bruke Ansatt-metoder på objektet. Merk for øvrig at vi burde sjekket returverdien fra kall på settInn.

Vi har nå en generell klasse for å representere objektsamlinger. For hver nye relasjon vi ønsker å jobbe med er det tilstrekkelig å definere en klasse for å representere en enkelt rad i relasjonen. Denne klassen må implementere grensesnittet RadObjekt, og altså inneholde en definisjon av metoden nøkkel. Det er imidlertid alt som skal til. Det er ikke nødvendig å programmere metodene for innsetting, sletting og søk på nytt.

Klassen Relasjon er et eksempel på det vi kaller en Collection-klasse, det vil si en klasse for å representere en objektsamling. Java-biblioteket inneholder en rekke ferdiglagede klasser av denne typen i pakken java.util. Vector, ArrayList, TreeMap og TreeSet er fire eksempler. Klassene er forskjellige når det gjelder hvilke metoder de tilbyr og hvordan elementene blir organisert internt. Noen krever at elementklassen må implementere et grensesnitt.

For *sorterte* objektsamlinger må man være i stand til å avgjøre om et element er mindre enn et annet. Følgende grensesnitt beskriver krav til elementene i en sortert objektsamling:

```
public interface Sammenlignbar
{
  public boolean mindreEnn( Sammenlignbar obj );
}
```

Klassen Ansatt kan implementere Sammenlignbar ved å sammenligne ansattnumre i metoden mindreEnn.

```
public class Ansatt extends Object
  implements RadObjekt, Sammenlignbar
{
  // Deklarasjon av instansvariable og
  // andre metoder er utelatt.
```

```
  public boolean mindreEnn( Sammenlignbar obj )
  {
    return ansattNr < ((Ansatt) obj).ansattNr;
  }
}
```

Merk at typeomforming til Ansatt er nødvendig før sammenligning fordi Sammenlignbar ikke inneholder ansattnumre!

Java-biblioteket inneholder for øvrig et grensesnitt Comparable som bør brukes i stedet for Sammenlignbar.

Relasjoner med fremmednøkler

En relasjonsdatabase består som regel av flere relasjoner. Logiske forbindelser mellom relasjonene blir representert ved *fremmednøkler*. En fremmednøkkel inneholder verdier som forekommer i primærnøkkelen til en annen relasjon, og representerer dermed et *forhold* mellom tupler i de to relasjonene[7]. Eksempel: Utvid ansatt-databasen med en relasjon Avdeling. Ved hver avdeling jobber det et antall ansatte, mens hver ansatt jobber ved nøyaktig én avdeling. Det betyr at avdelinger og ansatte hører sammen i et såkalt en-til-mange forhold. Denne koblingen kan bli representert ved at hvert ansatt-tuppel blir utvidet med primærnøkkelverdien til avdelingen den ansatte jobber ved. På samme måte kan avdelingsledelse, som er et en-til-en forhold mellom de to samme relasjonene, bli representert ved at ansattnummeret til avdelingslederen blir lagret i tuppelet som beskriver avdelingen. Tabellstrukturen blir i så fall som følger, der primærnøkler er understreket og fremmednøkler er merket med ei stjerne:

- Ansatt(AnsattNr, Fornavn, Etternavn, Telefon, AvdNavn*)
- Avdeling(AvdNavn, TlfLinje, AnsattNrLeder*)

Spørsmålet blir så hvordan flere relasjoner med fremmednøkler bør bli organisert som datastrukturer i et Java-program. Et par-tre strategier peker seg ut. Den første strategien innebærer å behandle fremmednøkler på samme måte som andre attributter. Klassen Avdeling vil i så fall bli definert som følger:

```
public class Avdeling extends Object
{
  private String avdNavn;                    // primærnøkkel
  private String tlfLinje;
  private int ansNrLeder;                    // fremmednøkkel

  // Definisjon av metoder er utelatt.
}
```

Klassen Ansatt blir utvidet med et nytt attributt avdNavn:

[7] Et "forhold mellom to relasjoner" kan virke underlig og kanskje meningsløst, men husk da at en relasjon er en databasetabell og at informasjonen som er lagret i én databasetabell kan høre sammen med informasjon lagret i en annen databasetabell.

```
public class Ansatt extends Object
{
  private int ansattNr;                      // primærnøkkel
  private String fornavn, etternavn, telefon;
  private String avdNavn;                    // fremmednøkkel

  // Definisjon av metoder er utelatt.
}
```

Klassene AnsattSamling og AvdelingSamling følger mønsteret vi allerede har beskrevet med hensyn til selve datastrukturen. Det nye blir å programmere inn kontroll med fremmednøkler. Hver gang vi setter inn et nytt objekt av klassen Avdeling må vi sjekke at verdien i instansvariabelen ansNrLeder eksisterer som ansattnummer, det vil si at det finnes et Ansatt-objekt med den samme verdien i instansvariabelen ansattNr. Av samme grunn kan vi ikke slette et objekt av klassen Ansatt hvis det finnes referanser til objektet fra et objekt av klassen Avdeling. Den samme typen kontroll må også programmeres for fremmednøkkelen fra Ansatt mot Avdeling. Merk at koden for fremmednøkkelkontroll må ligge i klassene AvdelingSamling og AnsattSamling, og forutsetter at vi ser objekter av disse klassene i sammenheng[8].

Vi kan alternativt velge å representere forhold ved *referanser*[9]. I stedet for at et objekt av klassen Ansatt inneholder et avdelingsnavn kan det inneholde en referanse til et avdelingsobjekt, og i stedet for at et objekt av klassen Avdeling inneholder et ansattnummer kan det inneholde en referanse til et Ansatt-objekt, slik:

```
public class Avdeling extends Object
{
  private String avdNavn;                    // primærnøkkel
  private String tlfLinje;
  private Ansatt avdLeder;                   // referanse

  // Definisjon av metoder er utelatt.
}

public class Ansatt extends Object
{
  private int ansattNr;                      // primærnøkkel
  private String fornavn, etternavn, telefon;
  private Avdeling avd;                      // referanse

  // Definisjon av metoder er utelatt.
}
```

Referanser effektiviserer visse typer av søk. Anta at variabelen enAnsatt refererer et Ansatt-objekt. Da kan vi finne avdelingens telefonlinje som følger, der hentAvd og hentTlfLinje er standard hent-metoder i henholdsvis klassen Ansatt og Avdeling:

```
String tlf = enAnsatt.hentAvd().hentTlfLinje();
```

[8] I en database har vi kun én relasjon Ansatt og én relasjon Avdeling. I et Java-program kan vi lage mange objekter av både AnsattSamling og AvdelingSamling. Fremmednøkkelkontrollen må skje relativt til et bestemt par av slik objekter.

[9] Noen vil foretrekke begrepet *peker*. I begge tilfeller er det snakk om adresser til objekter.

Med den første strategien måtte vi ha gjennomført et søk i et AvdelingSamling-objekt, med avdelingsnavnet som parameter.

En tredje strategi, som kan brukes for å representere en-til-mange forhold, er basert på at objekter kan inneholde referanser til objektsamlinger. Ved en avdeling jobber det mange ansatte. I stedet for at hvert Ansatt-objekt inneholder en referanse til sin avdeling, kan avdelingsobjektet inneholde et objekt av klassen AnsattSamling:

```
public class Avdeling extends Object
{
  private String avdNavn;                    // primærnøkkel
  private String tlfLinje;
  private Ansatt avdLeder;                   // referanse
  private AnsattSamling ansatte;             // objektsamling

  // Definisjon av metoder er utelatt.
}
```

Objekter av klassen AvdelingSamling kan vi nå kalle for *multitabeller*. De inneholder en tabell med referanser til Avdeling-objekter, som hver inneholder (en referanse til et objekt som inneholder) en tabell med referanser til Ansatt-objekter.

Filbehandling

Lagring av relasjoner i tekstfiler

En relasjon er en logisk struktur som kan lagres i forskjellige typer av filer og på flere måter. Vi skal bruke tekstfiler der hver rad legges på en egen linje. Innenfor en linje blir verdiene for hvert attributt separert med et skilletegn. I Figur 4 er relasjonen Ansatt lagret på denne måten med semikolon som skilletegn. For at dette skal fungere kan ikke semikolon forekomme som del av verdiene.

```
101;Bjørn;Krogfoss;3794
113;Hans Petter;Fosen;3798
117;Andrea;Børsheim;3754
122;Lene;Nilssen;3755
125;Fredrik;Wilhelmsen;3789
```

Figur 4. Ansatt-relasjon lagret i semikolonseparert tekstfil

I tråd med objektorientert tankegang bør kode for innlesing fra fil og utskrift til fil plasseres inne i klassene Ansatt og AnsattSamling.

Unntak

Unntak (exceptions) brukes for å avbryte normal programutførelse. Kall på metoder som kan generere, eller "kaste" unntak må omsluttes av en *unntakshåndterer*:

```
try
{
  <Kall på metode som kan generere unntak>
}
catch ( <Unntaksklasse> <navn> )
{
  <Håndter feilen>
}
```

Det er mye som kan gå galt når et Java-program skal manipulere data lagret på fil. For eksempel kan filen vi prøver å lese være slettet. Metoder vi bruker for å åpne og lukke filer, lese fra filer og skrive til filer kan alle generere unntak som må "fanges". Kode for å åpne en fil for skriving blir slik:

```
try
{
  utfil = new PrintWriter( new FileWriter("fil.txt"), true );
}
catch ( IOException e )
{
  System.out.println( "Feil ved åpne fil: " + e.toString() );
  System.exit( -1 ); // Avslutter programmet pga. fatal feil
}
```

Her er *unntaksklassen* IOException (Input/Output-feil) og selve unntaket er gitt navnet e. Tenk på e som en slags formell parameter. Hvis noe går galt skriver vi ut en feilmelding, der e.toString() inneholder en detaljert forklaring til feilen.

Hvis vi skal utføre flere operasjoner som alle kan generere unntak er det ofte hensiktsmessig å samle alle i én try/catch-blokk:

```
try
{
  <Utfør operasjon 1>
  <Utfør operasjon 2>
  <Utfør operasjon 3>
}
catch ( <Unntaksklasse> <navn> )
{
  <Håndter mulige feil>
}
```

Uansett hvilken av operasjonene som genererer et unntak kommer vi til den samme unntakshåndtereren. Det er også mulig å ha flere catch-blokker i samme unntakshåndterer for å spesialbehandle de forskjellige feilsituasjonene.

Innlesing fra fil

Klassen AnsattSamling skal nå utvides med en metode lesFraFil som tar navn på datafilen som parameter. Metoden leser filen linje for linje, genererer ett rad-objekt for hver linje og legger referanser til objektene inn i tabellen.

I tråd med tidligere designvalg lar vi metoden returnere en boolsk verdi som forteller om operasjonen gikk bra eller ikke, og overlater til andre deler av programmet å skrive ut eventuelle feilmeldinger til brukeren.

```
  // Leser ansattdata inn fra fil og
  // setter Ansatt-objekter inn i tabellen.
  public boolean lesFraFil( String filnavn )
  {
    boolean ok = true;
    try
    {
      BufferedReader innfil =
        new BufferedReader( new FileReader( filnavn ) );

      String linje = innfil.readLine();
      while ( linje != null )
      {
        Ansatt a = new Ansatt( linje );
        if ( !settInn( a ) )
          ok = false;
        linje = innfil.readLine();
      }
      innfil.close();
    }
```

```
    catch ( IOException e )
    {
      ok = false;
    }
    return ok;
  }
```

For at koden over skal fungere må klassen Ansatt utvides med en ny konstruktør som tar en tekststreng som parameter. Teksten er hentet fra en linje på filen, og består altså av verdiene for hvert attributt separert med semikolon. Den forhåndsdefinerte klassen java.util.StringTokenizer kan brukes for å trekke ut verdiene fra teksten slik at de kan tilordnes instansvariablene.

For å gjøre det enkelt å bytte til et annet skilletegn er det innført en statisk instansvariabel i klassen Ansatt som holder rede på hvilket skilletegn som brukes.

```
  private static final String SKILLETEGN = ";";
```

Ved innlesing fra fil er det ofte nødvendig å konvertere fra String til datatyper som int og float for lagring i instansvariable. I dette tilfellet må ansattnumre gjøres om til heltall med metoden parseInt.

```
  // Konstruktør som bygger opp objektet fra en tekststreng
  // med attributtverdier separert med et skilletegn.
  public Ansatt( String linje )
  {
    StringTokenizer ord =
      new StringTokenizer( linje, SKILLETEGN );
    ansattNr = Integer.parseInt( ord.nextToken() );
    fornavn = ord.nextToken();
    etternavn = ord.nextToken();
    telefon = ord.nextToken();
  }
```

Vi antar her at filen alltid er på riktig format. Det kan være akseptabelt hvis vi har kontroll også med lagring til fil, men vi bør nok som hovedregel legge inn forskjellige typer av tester: at hver linje består av fire dataelementer, at det første alltid er et tall, at det siste har form av et lovlig telefonnummer og så videre.

Lagring til fil

Vi skal programmere lagring til fil ved en metode i klassen Ansatt og en metode i klassen AnsattSamling. Vi lar begge få navnet skrivTilFil. Dette kalles for *overlasting*[10] og er altså lovlig i Java.

Metoden Ansatt.skrivTilFil har som oppgave å skrive ett enkelt Ansatt-objekt til fil. Vi antar her at filen er åpnet, og lar dermed metoden få et PrintWriter-objekt som parameter. Metoden skriver attributtene med skilletegn til en ny linje på filen. Merk at filformatet som brukes ved lagring til fil må stemme overens med det formatet vi bygger på ved innlesing.

[10] Noen reserverer begrepet overlasting om det tilfellet at to metoder med samme navn blir definert i *samme klasse*.

```
// Skriver Ansatt-objekt til en linje på en utfil,
// der verdiene er separert med skilletegn.
public void skrivTilFil( PrintWriter utfil )
{
  utfil.println( ansattNr + SKILLETEGN +
                 fornavn + SKILLETEGN +
                 etternavn + SKILLETEGN +
                 telefon );
}
```

Metoden AnsattSamling.skrivTilFil tar navnet på datafilen som parameter, løper gjennom den logiske tabellen element for element, og skriver hvert objekt til en linje på filen ved kall på Ansatt.skrivTilFil. Også denne metoden har returdatatype boolean, der vi returnerer false hvis noe går galt ved lagring:

```
// Skriver ansattdata til fil.
public boolean skrivTilFil( String filnavn )
{
  try
  {
    PrintWriter utfil =
      new PrintWriter( new FileWriter( filnavn ), true );

    for ( int i=0; i<nesteLedige; i++ )
      tabell[i].skrivTilFil( utfil );

    utfil.close();
    return true;
  }
  catch ( IOException e )
  {
    return false;
  }
}
```

Konsollapplikasjoner

Vi skal nå lage et komplett program som bruker datastrukturen og metodene vi har definert. Programmet skal være en selvstendig applikasjon som konverterer samtlige telefonnumre i ansatt-filen til 8 siffer (nå er kun internnummer lagret), og dessuten skriver ut en kvittering om hvor mange ansatte som ble berørt av oppdateringen. Brukeren angir filnavnet ved en kommandolinjeparameter. Pseudokode:

1. Les inn ansatt-data fra fil og bygg opp intern datastruktur.
2. Gå gjennom alle Ansatt-objekter (med iteratormetoder):
    a. Endre attributtet Telefon slik at det inneholder fullstendig telefonnummer (8 siffer).
3. Skriv ut antall ansatte som er oppdatert.
4. Skriv oppdatert datastruktur tilbake til fil.

Programmet blir slik:

```
public class Telefon extends Object
{
  public static void main( String[] args )
  {
    final String TLFPREFIKS = "35 95 ";
    final int MAX = 100;

    String filnavn = " ";
    AnsattSamling ansatte;
    int antall = 0;

    // Hent filnavn fra kommandolinjen
    if ( args.length == 1 )
      filnavn = args[0];
    else
      Logg.avslutt( "Skriv: java Telefon <innfil>" );

    // Bygg datastruktur fra fil
    ansatte = new AnsattSamling( MAX );
    if ( !ansatte.lesFraFil( filnavn ) )
      Logg.avslutt( "Feil ved innlesing av fil: " + filnavn );

    // Oppdater datastrukturen og tell opp antall ansatte
    ansatte.gåTilFørste();
    while ( ansatte.flere() )
    {
      Ansatt enAnsatt = ansatte.denne();
      antall++;
      enAnsatt.settTelefon( TLFPREFIKS +
                            enAnsatt.hentTelefon() );
      ansatte.gåTilNeste();
    }

    // Skriv kvittering til konsollet
    Logg.melding( "Antall ansatte oppdatert: " + antall );

    // Skriv datastruktur til fil
    if ( !ansatte.skrivTilFil( filnavn ) )
      Logg.avslutt( "Feil ved lagring av fil: " + filnavn );
  }
}
```

Her og i andre eksempler som følger antar vi tilgang på statiske metoder melding og avslutt i en hjelpeklasse Logg. Begge viser en melding på skjermen. Sistnevnte avslutter dessuten programmet med metoden System.exit.

Applikasjoner med grafiske brukergrensesnitt

Vi skal nå skissere hvordan man kan lage et program med grafisk brukergrensesnitt for ajourhold av ansattdata. Programmet vil til enhver tid vise data om én ansatt i fire tekstbokser (ansattnummer, fornavn, etternavn og telefon). Brukeren skal kunne navigere med knapper merket "første", "forrige", "neste" og "siste". Brukeren skal dessuten kunne slette, oppdatere og registrere nye ansatte med egne knapper.

Programmet er skrevet som en selvstendig applikasjon der filnavnet blir oppgitt som kommandolinjeparameter. Ved oppstart blir data lest inn fra fil, og datastrukturen blir bygget opp som et objekt av klassen AnsattSamling.

Navigasjonsknappene er implementert ved kall på iteratormetoder. Oppdatering, innsetting og framvisning av inneværende ansatt gjøres ved egne hjelpemetoder.

Kode for initialisering av brukergrensesnittet gjøres i en hjelpemetode initGUI. Definisjonen av denne metoden er utelatt her.

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

public class AnsattReg extends Frame
  implements WindowListener, ActionListener
{
  private final int MAX = 100;

  private TextField txtAnsNr, txtFNavn, txtENavn, txtTlf;
  private Button btnFørste, btnSiste, btnForrige, btnNeste,
                 btnNy, btnOppdat, btnSlett;

  private AnsattSamling ansatte;
  private String filnavn;

  public static void main( String[] args )
  {
    AnsattReg skjema = new AnsattReg( args[0] );
  }

  public AnsattReg( String filnavn )
  {
    initGUI(); // Initialiserer brukergrensesnittet

    // Bygger opp datastruktur fra fil
    this.filnavn = filnavn;
    ansatte = new AnsattSamling( MAX );
    if ( !ansatte.lesFraFil( filnavn ) )
      Logg.avslutt( "Klarte ikke å lese filen " + filnavn );

    ansatte.gåTilFørste();
    vis( ansatte.denne() );
  }
```

```
public void actionPerformed( ActionEvent e )
{
  if ( e.getSource() == btnFørste )
    ansatte.gåTilFørste();
  else if ( e.getSource() == btnForrige )
    ansatte.gåTilForrige();
  if ( e.getSource() == btnNeste )
    ansatte.gåTilNeste();
  else if ( e.getSource() == btnSiste )
    ansatte.gåTilSiste();
  else if ( e.getSource() == btnNy )
    ansatte.settInn( lagNy() );
  else if ( e.getSource() == btnOppdat )
    oppdater ( ansatte.denne() );
  else if ( e.getSource() == btnSlett )
    ansatte.slett( Integer.parseInt( txtAnsNr.getText() ) );

  vis( ansatte.denne() );
}

// Skriver oppdatert datastruktur til fil og
// avslutter programmet
public void windowClosing( WindowEvent e )
{
  if ( ansatte.skrivTilFil( filnavn ) )
    Logg.avslutt( "Data er lagret." );
  else
    Logg.avslutt( "Klarte ikke å lagre data." );
}

// Viser ansattdata i tekstboksene
private void vis( Ansatt a )
{
  if ( a != null )
  {
    txtAnsNr.setText( Integer.toString(a.hentAnsattNr()) );
    txtFNavn.setText( a.hentFornavn() );
    txtENavn.setText( a.hentEtternavn() );
    txtTlf.setText( a.hentTelefon() );
  }
}
```

```
  // Oppretter et nytt Ansatt-objekt med data fra tekstboksene
  private Ansatt lagNy()
  {
    Ansatt a =
      new Ansatt( Integer.parseInt( txtAnsNr.getText() ),
                  txtFNavn.getText(),
                  txtENavn.getText(),
                  txtTlf.getText() );
    return a;
  }

  // Oppdater et gitt Ansatt-objekt med data fra tekstboksene
  private void oppdater( Ansatt a )
  {
    if ( a != null )
    {
      a.settAnsattNr( Integer.parseInt(txtAnsNr.getText()) );
      a.settFornavn( txtFNavn.getText() );
      a.settEtternavn( txtENavn.getText() );
      a.settTelefon( txtTlf.getText() );
    }
  }

  // Øvrige hendelsesmetoder og hjelpemetoder er utelatt.
}
```

Databaseprogrammering med JDBC

Vi skal nå forklare hvordan man kan utvikle enkle databaseapplikasjoner i henhold til en klient/tjener-arkitektur, der data er lagret i et relasjonsdatabasesystem og brukergrensesnittet er kodet i Java. For å få til dette må Java og databasesystemet kunne utveksle data. Heldigvis er mesteparten av jobben gjort for oss. *JDBC* (Java DataBase Connectivity) er et ferdiglaget klassebibliotek vi kan bruke for å aksessere databaser fra Java. Tanken er enkel:

- Java-programmet sender en SQL-spørring til databasesystemet. Konkret skjer det ved å kalle på metoder i JDBC-biblioteket med spørringen som en tekstparameter.
- Databasesystemet utfører SQL-spørringen og sender resultatet tilbake. I Java-koden får vi tak i spørreresultatet som returverdi fra metoden som fikk spørringen som parameter.

Tenk gjerne på JDBC som et verktøy for å pakke inn relasjoner som objekter; en slags objektorientert "brille" mot en relasjonsdatabase. For å få tilgang til JDBC-biblioteket må vi legge til følgende importsetning:

```
import java.sql.*;
```

Programmer som bruker JDBC må som regel gjennom følgende fire faser:

- Opprett forbindelse med databasen
- Send en spørring til databasen
- Behandle spørreresultatet
- Lukk forbindelsen med databasen

Bruk av JDBC forutsetter at databasen og nødvendige systemprogrammer er installert. Vi beskriver dette før vi ser nærmere på hver enkelt fase.

JDBC-drivere

I tillegg til klassebiblioteket JDBC må en passende *JDBC-driver* være installert på maskinen. Slike drivere er systemprogrammer som håndterer dataoverføringen mellom Java-programmet og databasesystemet.

Det finnes flere typer av JDBC-drivere. *JDBC/ODBC-broen* er inkludert i Java 2, og kan også lastes ned gratis fra internett. Driveren kan brukes mot alle databaser det finnes en ODBC-driver[11] for, og krever at tilhørende ODBC-driver er installert på maskinen der Java-programmet skal bli *utført*. Dette er dermed en grei løsning hvis man bare skal eksperimentere med JDBC og kan lagre alt på en og samme maskin.

Anta nå at JDBC/ODBC-broen er installert. Det som da gjenstår av forberedelser er å legge databasen som skal brukes inn i listen over ODBC datakilder, som i vårt eksempel er en Microsoft Access database. Dette gjøres ved hjelp av *ODBC-administratoren*, som er et systemprogram som i Windows kan startes fra kontrollpanelet. I norsk versjon vil man legge inn en ny *brukerdatakilde*, angi Access-databasen som programmet skal jobbe med, og gi denne et logisk navn. I eksemplene som følger antar vi at databasen er gitt logisk navn *ansatt*.

[11] ODBC står for Open DataBase Connectivity og er en standard som på samme måte som JDBC definerer et grensesnitt for å aksessere forskjellige databasesystemer. JDBC/ODBC-broen var en enkel måte å få JDBC "på lufta" ved at den er bygget på ODBC.

Opprette forbindelse med databasen

For å koble seg opp mot en lokal database er det tilstrekkelig å oppgi *datakildenavnet* til databasen, og eventuelt *brukernavn* og *passord*. Følgende kodebit oppretter en forbindelse mot ansatt-databasen:

```
String URL = "jdbc:odbc:ansatt";
String brukernavn = "ola";
String passord = "hemmelig";
Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
Connection forbindelse =
  DriverManager.getConnection( URL, brukernavn, passord );
```

I metodekallene på Class.forName og DriverManager.getConnection blir korrekt JDBC-driver valgt og forbindelsen til databasen etablert, og det skjer en god del ting bak kulissene. Det er imidlertid ikke nødvendig å forstå nøyaktig hvordan oppkoblingen gjøres. Du må imidlertid vite hvordan du tilpasser koden til ditt formål. Hvis du har gitt databasen navnet minDatabase i ODBC-administratoren setter du variabelen URL til "jdbc:odbc:minDatabase". Hvis du ikke trenger brukernavn og passord for å aksessere databasen bruker du tomme tekststrenger.

I koden som følger skal vi jobbe videre med variabelen forbindelse. Det er denne som er vår “inngangsport” til databasen.

Utføre spørringer

Vi utfører SQL-spørringer ved å kalle på bestemte metoder i JDBC-biblioteket. I utgangspunktet er det ingen restriksjoner på hvilke deler av SQL vi kan bruke. Både utvalgsspørringer (SELECT), innsetting (INSERT), oppdatering (UPDATE), sletting (DELETE) og datadefinisjon (CREATE TABLE, CREATE INDEX) er tilgjengelige.

Vi starter alltid med å lage et objekt av klassen Statement. Denne klassen har en metode executeQuery for å utføre utvalgsspørringer. Metoden tar SQL-koden som en tekstparameter og returnerer spørreresultatet som et objekt av en annen ferdiglaget klasse ResultSet:

```
String spørring =
  "SELECT AnsattNr, Etternavn FROM Ansatt ORDER BY Etternavn";
Statement setning = forbindelse.createStatement();
ResultSet resultat = setning.executeQuery( spørring );
```

I koden over blir spørreresultatet tilordnet variabelen resultat. Slike ResultSet-objekter ligner en god del på objekter av klassen AnsattSamling. De holder begge orden på en samling rad-objekter som kan gjennomløpes med iteratormetoder.

Som programmerer bør man være bevisst at koden over inneholder både Java og SQL. Normalt vil Java-kompilatoren gi beskjed om syntaksfeil og typefeil. Feil i SQL-koden vil imidlertid ikke bli oppdaget før man utfører programmet, og det er databasesystemet som vil oppdage feilen.

Metoden executeQuery kan kun brukes for utvalgsspørringer (SELECT). Skal du sette inn, slette eller endre rader i en tabell bruker du executeUpdate i stedet. Her får vi ikke en objektsamling tilbake, men bare et tall som forteller hvor mange rader som ble berørt av operasjonen. Antallet kan tilordnes en variabel, eller skrives ut på skjermen.

```
String spørring =
  "UPDATE Ansatt SET Telefon = '35 95 ' & Telefon";
Statement setning = forbindelse.createStatement();
int antallRader = setning.executeUpdate( spørring );
```

Merk for øvrig bruken av enkle apostrofer i SQL-setningen. '35 95 ' er en tekstkonstant som hører til SQL-setningen og dermed blir tolket av databasesystemet.

Det er faktisk mulig å la brukeren skrive inn en SQL-setning, som deretter kan bli utført. Da vet man (som programmerer) ikke om det er riktig å bruke executeQuery eller executeUpdate. Løsningen i slike situasjoner er å bruke den mer generelle metoden execute:

```
String spørring =
  JOptionPane.showInputDialog( "Skriv SQL: " );
Statement setning = forbindelse.createStatement();
boolean spørretype = setning.execute( spørring );
```

Her får vi en boolsk verdi som forteller hva slags type spørring som ble utført: true betyr utvalgsspørring og false oppdateringsspørring. Dernest kan vi bruke metodene getResultSet eller getUpdateCount i klassen Statement for å få tak i henholdsvis returnert objektsamling eller antall berørte rader.

Behandle spørreresultater

Følgende løkke løper gjennom resultatet av en utvalgsspørring rad for rad, og skriver ut verdiene til attributtene AnsattNr og Etternavn:

```
while ( resultat.next() )
{
  String ansNr = resultat.getString( "AnsattNr" );
  String etternavn = resultat.getString( "Etternavn" );
  System.out.println( ansNr + " " + etternavn );
}
```

Metoden next er en iteratormetode, men implementert litt annerledes enn slik vi gjorde det i avsnittet om iteratorer. Når vi mottar spørreresultatet er nåværende posisjon satt til en tenkt rad før den første raden. Posisjonen økes med 1 for hvert kall på next, som gjør at vi ved første gjennomløp av løkka får behandlet den første raden i tabellen.

I tillegg til å forflytte posisjonen til neste rad sjekker metoden next om vi er kommet til slutten. Metoden returnerer false hvis vi har lest forbi siste rad. Metoden brukes altså til tre ulike oppgaver: "Gå til første", "gå til neste" og "sjekk om det er flere rader".

Metoden getString tar et attributtnavn som parameter og returnerer tilhørende verdi som en tekst. Hvis verdien skal behandles som et tall må vi etterpå utføre en konvertering. Det finnes metoder som gjør avlesing og konvertering i ett steg, for eksempel kan man bruke getInt på attributter som man vet inneholder heltall.

Klassen ResultSet tilbyr for øvrig atskillig flere metoder enn next, getString og getInt.

Lukke forbindelsen til databasen

Man skal alltid rydde opp etter seg. Metoden close kan brukes på objekter av klassene ResultSet, Statement og Connection. Med vårt valg av variabelnavn over kan vi dermed skrive:

```
resultat.close();
setning.close();
forbindelse.close();
```

Når vi lukker et Statement-objekt lukker vi også tilhørende spørreresultat, og når vi lukker et Connection-objekt lukker vi også tilhørende setninger. I vårt eksempel vil dermed den siste setningen være tilstrekkelig.

Et testprogram

Kodebitene forklart over vil nå bli satt sammen til et komplett program. Følgende program viser etternavnene til alle ansatte i stigende rekkefølge:

```
import java.sql.*;
public class AnsattRapport extends Object
{
  public static void main( String[] args )
  {
    try
    {
      String url = "jdbc:odbc:ansatt";
      String spørring =
        "SELECT * FROM Ansatt ORDER BY Etternavn";
      Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
      Connection c = DriverManager.getConnection( url,"","" );
      Statement s = c.createStatement();
      ResultSet r = s.executeQuery( spørring );
      while ( r.next() )
        Logg.melding( r.getString( "Etternavn" ) );
      c.close();
    }
    catch ( Exception e )
    {
      Logg.avslutt( "Feil: " + e.toString() );
    }
  }
}
```

Merk at operasjonene mot databasen er omsluttet av en felles try/catch-blokk. Enten det oppstår en feil ved oppkobling til databasen, eller ved utførelse av SQL-spørringen, havner kontrollen i unntakshåndtereren der vi skriver ut en feilmelding.

Grafisk brukergrensesnitt

Vi skal nå skissere et innsynsprogram med grafisk brukergrensesnitt mot ansatt-databasen. Brukeren vil se data om én ansatt av gangen, vist fram i tekstbokser. Vi trenger dermed en tekstboks for hvert attributt i tabellen. En kommandoknapp brukes for å navigere til neste ansatt.

Brukeren kan ikke gå til forrige rad, noe som er en sterk begrensning ved programmet. Brukeren kan heller ikke sette inn, endre eller slette data. Vi kommenterer mulige utvidelser av funksjonaliteten med hensyn på navigering og oppdatering om litt.

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

public class AnsattGUI extends Frame
  implements WindowListener, ActionListener
{
  private Connection c;
  private Statement s;
  private ResultSet resultat;

  private Button btnNeste;
  private TextField txtAnsNr, txtFNavn, txtENavn, txtTelefon;

  public static void main( String[] args )
  {
    AnsattGUI skjema = new AnsattGUI();
  }

  public AnsattGUI()
  {
    initGUI();
    gåTilFørste();
  }

  public void actionPerformed( ActionEvent e )
  {
    if ( e.getSource() == btnNeste )
      gåTilNeste();
  }

  public void windowClosing( WindowEvent e )
  {
    lukk();
    Logg.avslutt();
  }
```

```
// Viser inneværende rad på skjermen
private void vis()
{
  try
  {
    txtAnsNr.setText( resultat.getString( "AnsattNr" ) );
    txtFNavn.setText( resultat.getString( "Fornavn" ) );
    txtENavn.setText( resultat.getString( "Etternavn" ) );
    txtTelefon.setText( resultat.getString( "Telefon" ) );
  }
  catch  ( Exception e )
  {
    Logg.melding( "Feil ved avlesing av spørreresultat: " +
                  e.toString() );
  }
}

// Oppretter forbindelse til databasen og viser første rad
private void gåTilFørste()
{
  try
  {
    // Oppretter forbindelse til databasen
    String url = "jdbc:odbc:ansatt";
    String sql = "SELECT * FROM Ansatt";
    Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
    c = DriverManager.getConnection( url,"","" );

    // Utfører spørringen
    s = c.createStatement();
    resultat = s.executeQuery( sql );

    // Går til første rad og viser data på skjermen
    if ( resultat.next() )
      vis();
  }
  catch ( Exception e )
  {
    Logg.avslutt( "Feil ved oppkobling til databasen: " +
                  e.toString() );
  }
}

// Går til neste rad og viser data på skjermen
private void gåTilNeste()
{
  try
  {
    if ( resultat.next() )
      vis();
  }
```

```
    catch  ( Exception e )
    {
      Logg.avslutt( "Feil ved navigering til neste rad: " +
                    e.toString() );
    }
  }

  // Definisjon av noen hendelsesmetoder,
  // samt initGUI og lukk er utelatt.
}
```

Legg merke til hvordan koden for aksess av databasen er fordelt utover flere av metodene i programmet:

- I metoden gåTilFørste oppretter vi forbindelsen til databasen, utfører spørringen og posisjonerer oss på første rad i spørreresultatet.
- I metoden gåTilNeste forsøker vi å flytte oss til neste rad. Dette lykkes ikke hvis vi allerede er på siste rad.
- I metoden vis viser vi fram data om nåværende ansatt.
- I metoden lukk (definisjonen er utelatt) lukker vi forbindelsen til databasen.

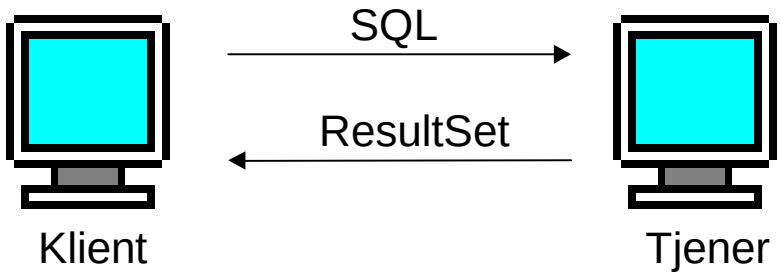
Brukeren vil opplagt ønske seg mulighet for å gå til både forrige rad og til neste rad. Det er to måter å få til dette på. Hvis JDBC-driveren og det konkrete spørreresultatet tillater det, noe ikke alle gjør, kan man rett og slett legge til en ny kommandoknapp og kalle på en metode previous. I motsatt fall må man programmere denne funksjonen selv, for eksempel ved å løpe gjennom spørreresultatet fra begynnelse til slutt, kopiere data over i en egen datastruktur, og så kalle på iteratormetoder mot denne strukturen i stedet.

I vårt program kunne ikke brukeren gjøre endringer. En slik utvidelse kan også gjøres på to måter, enten ved bruk av SQL-kommandoene INSERT, UPDATE og DELETE, eller ved egne JDBC-metoder for innsetting, oppdatering og sletting.

Klient/tjener-arkitektur og Web-løsninger

JDBC er altså et klassebibliotek som brukes for å få Java-programmer og databaser til å kommunisere på en enkel måte. Generelt vil databasesystemet og Java-applikasjonen bli utført på to forskjellige maskiner. Ofte vil mange brukere kunne aksessere den samme databasen over et lokalnett. Hver enkelt bruker vil i så fall utføre Java-applikasjonen på sin maskin.

Java-programmet og databasen kommuniserer da i henhold til en såkalt *klient/tjener-arkitektur*, se Figur 5. Et system bygget opp i henhold til dette mønsteret består av en *tjener* og en eller flere *klienter*. Kommunikasjon foregår ved at klienten sender en *forespørsel* til tjeneren. Ofte vil forespørselen medføre arbeidsoppgaver for tjeneren. Tjeneren utfører oppgavene og sender en *respons* tilbake. For oss vil Java-programmet (utført på en brukermaskin) spille rollen som klient, mens databasesystemet vil være tjeneren. Forespørsler vil være kodet som SQL-spørringer, og svar fra databasesystemet kan være et ResultSet (for utvalgsspørringer). Teknisk vil forespørselen være en parameter til en metode i JDBC-biblioteket, og responsen være returverdien fra den samme metoden.



Figur 5. Klient/tjener-arkitektur med JDBC

Klient/tjener-arkitektur blir for øvrig brukt i mange forskjellige sammenhenger, og er altså ikke begrenset til Java og databaser.

Denne formen for kommunikasjon fungerer i prinsippet like godt enten Java-programmet og databasesystemet ligger på samme maskin, eller på to maskiner lokalisert på hver sin side av kloden. Alt Java-programmet må ”vite” er hvor databasen befinner seg, og alt databasesystemet må ”kunne” er SQL. En *URL* (Uniform Resource Locator) er en adresseringsmekanisme til bruk på internett, som beskriver hvor ulike ressurser finnes og hva slags protokoll som gjelder for å bruke dem. I denne sammenhengen er en *ressurs* en database og en *protokoll* en bestemt JDBC-driver. I en URL på formen jdbc:odbc:ansatt vil ansatt henvise til en database i et lokalnett som blir aksessert via JDBC/ODBC-broen.

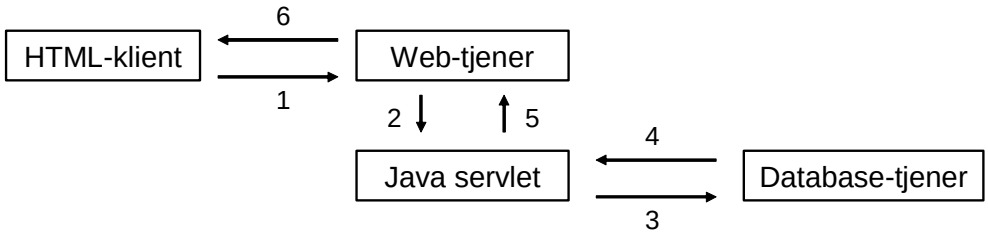
En *applet* er et Java-program som blir lastet ned fra internett og utført på klient-maskinen som del av en nettside. JDBC/ODBC-broen forutsetter at det er installert ODBC-drivere på klient-maskinen, og er dermed lite egnet for bruk i en applet, der man ikke har kontroll med hva slags programvare som er installert på klientmaskinen. Det finnes imidlertid andre typer drivere som kun forutsetter tilgang på en nettleser med Java-støtte. En URL for å opprette forbindelsen til en database fra en applet kan se ut som følger:

```
String URL = "jdbc:mindb://www.hit.no:7537/ansatt";
```

I dette tilfellet er jdbc:mindb protokollen som brukes, www.hit.no adressen til *web-tjeneren*, 7537 et *portnummer* og ansatt navnet på databasen. En applet og et databasesystem kan dermed kobles sammen i en klient/tjener-arkitektur og kommunisere over internett, som vist i Figur 5. Konkret syntaks for oppkobling av databaser vil variere avhengig av hvilken JDBC-driver man bruker. På hjemmesiden til Java, *java.sun.com*, finnes det en liste over tilgjengelige JDBC-drivere. Listen inneholder i skrivende stund mer enn 200 produkter. Det har for øvrig ingen hensikt å teste ut koden over. Databasen ansatt finnes ikke, og navnet på protokollen jdbc:mindb er funnet på for anledningen.

I utgangspunktet utgjør all nedlastet programkode en sikkerhetsrisiko for brukeren. Tenk på et program som prøver å slette alle filene på maskinen den blir lastet ned til. Enhver applet er imidlertid underlagt visse restriksjoner som gjør at dette ikke skal være mulig. Programmene blir utført i en såkalt *sandkasse*, som er en omgivelse der kun trygge operasjoner (for klienten) er lovlige. En applet kan blant annet ikke lese fra eller skrive til filer og databaser lagret på klienten. Derimot får en applet lov til å jobbe med filer og databaser fra maskinen den ble *lastet ned fra*.

I en *web-applikasjon* kan man altså kode brukergrensesnittet ved hjelp av en applet. Java blir imidlertid oftere benyttet på tjenersiden enn på klientsiden. En *servlet* er et Java-program som blir utført på en web-tjener. Man kan bygge web-løsninger fra HTML-klienter og servlet-teknologi. Figur 6 viser hvordan kommunikasjonen kan foregå.



Figur 6 Web-løsning med HTML-klient og servlet

Tallene i figuren viser hvordan en bruker kan utføre en forespørsel mot databasen. Anta at databasen inneholder data om varelageret til en nettbutikk, og at brukeren vil se alle varene i en bestemt kategori som koster mindre enn et bestemt beløp. Brukeren skriver inn kategori og prisgrense i et HTML-skjema. Deretter foregår kommunikasjonen i henhold til tallene i figuren:

1. HTML-klienten, det vil si nettleseren, overfører skjemadata (kategori og prisgrense) til web-tjeneren. Navn på Java-programmet blir også overført (navnet var allerede bakt inn i HTML-koden lastet ned fra web-tjeneren).
2. Web-tjeneren starter riktig servlet og videresender skjemadata som parametere.
3. Java-programmet kobler seg opp til databasen ved hjelp av JDBC, bygger opp en SQL utvalgsspørring, og overfører denne til databasen.
4. Spørreresultatet blir sendt tilbake til Java-programmet etter at databasesystemet har utført SQL-koden.
5. Java-programmet genererer en HTML-side som beskriver spørreresultatet, og gir dette til web-tjeneren.
6. Web-tjeneren videresender HTML-siden til nettleseren, som viser resultatet på skjermen til brukeren.

Fysisk kan dette systemet bestå av fire maskiner. Nettleseren er koblet sammen med Web-tjeneren via internett. Java-programmet kan være lagret på web-tjeneren, men trenger ikke å være det. Det er også mulig å plassere det på en egen maskin, en såkalt *applikasjonstjener*. Java-programmet og databasesystemet kan være koblet sammen i et lokalnett. Merk at web-tjener og servlet er å betrakte som en tjener mot nettleseren, men som en klient mot databasesystemet.

Ved å kode brukergrensesnittet i HTML får vi det som ofte kalles for *tynne klienter*, i motsetning til *tykke* Java-klienter. Servlet-løsningen med HTML-klienter har den fordelen framfor applet-klienter at den ikke krever installasjon av spesiell programvare og vil også være raskere å laste. Det eneste brukeren trenger er en nettleser. Motsatt så er fordelen med kraftigere Java-klienter økt funksjonalitet, for eksempel i form av mer avanserte brukergrensesnitt og mulighet for å utføre mer kompliserte beregninger lokalt.

Videre studier

Vi avslutter med noen tips til videre studier av databaseprogrammering med Java. Kompendiet bygger som nevnt innledningsvis på grunnleggende kunnskaper om relasjonsdatabaser og objektorientert programmering i Java. [Connolly, Begg] og [Lervik, Havdal] er to gode lærebøker innenfor hvert av disse emnene, og begge tar også for seg mer avanserte emner i skjæringspunktet mellom databaser og programmering. Hjemmesiden til Java, *java.sun.com*, og særlig underkatalogen */products/jdbc*, inneholder også mye relevant informasjon om Java og databaser.

JDBC, se [Fisher m.fl.] og [Reese], inneholder langt mer enn det som er beskrevet i dette kompendiet. For eksempel gir klassene MetaData og ResultSetMetaData tilgang til *systemkatalogen* i en database. Det gjør det mulig å lage et program som kan koble seg opp til en database, og hente ut navn på relasjoner og attributter. Slike teknikker er grunnlaget for å bygge generelle spørreverktøy, som må fungere mot en vilkårlig database. Videre så har vi kun sett på de aller enkleste teknikkene for å behandle spørringer. Eksempler på mer avanserte teknikker: toveis navigering i et ResultSet, transaksjonshåndtering, satsvis oppdatering av store datamengder, og gjenbruk av forbindelser for økt effektivitet (connection pooling).

SQLJ, se [Price 2001], er en måte å kombinere Java og SQL på mot Oracle-databaser, der vi i stedet for å kalle på metoder som i JDBC, skriver SQL rett inn i Java-koden med en spesiell syntaks. Slik programkode blir sendt gjennom en *preprosessor* som oversetter koden til et Java-program som inneholder kall på JDBC-metoder. *JDO* (Java Data Objects), se [Tyagi m.fl.], er nok en måte å kombinere Java og databaser, men vil i større grad enn JDBC og SQLJ skjule SQL og selve databasen. Tanken er at programmereren kun spesifiserer hvilke klasser som skal være *persistente*, noe som betyr at objektene automatisk blir skrevet til ytre lager.

Objektdatabaser kan betraktes som en utvidelse av objektorientert programmering med persistente objekter og databaseteknologi som gir støtte for transaksjoner, samtidige brukere og spørrespråk. Relasjonsdatabaser er mindre egnet for anvendelser som krever kompliserte datastrukturer, slik som dataassistert konstruksjon og produksjon (DAK/DAP), multimedia og geografiske informasjonssystemer (GIS). *Objektrelasjonelle databaser* er hybridsystemer som utvider relasjonsdatabaser med egendefinerte datatyper og avanserte datastrukturer fra objektorientering. [Connolly, Begg] inneholder en fyldig beskrivelse av slike systemer. Se [Melton] for en mer spesialisert bok.

Lagrede prosedyrer, se [Price 2004] og [Shah], er delprogrammer som blir lagret og utført på databasetjeneren. Bruk av lagrede prosedyrer kan blant annet redusere nettverkstrafikken i et distribuert system. I noen databasesystemer er det mulig å bruke Java for å utvikle slike mekanismer.

JSP (JavaServer Pages), se [Brown m.fl.], er et skriptspråk og en anvendelse av servlet-teknologi som gjør det mulig å skille beskrivelse av brukergrensesnittet, det vil si HTML-klienten, fra beregningsregler programmert i Java. JSP blir ofte kombinert med såkalte *Enterprise Java Beans* (EJB), som er komponenter for å bygge distribuerte databaseløsninger på en objektorientert måte. [Bodoff m.fl.] tar for seg EJB.

XML (eXtensible Markup Language), se [Goldfarb, Prescod], er et dokumentformat som syntaktisk ligner på HTML. Til forskjell fra HTML inneholder XML ingen predefinerte elementer, men til gjengjeld er det mulig å innføre egendefinerte elementer, noe som gjør det

mer generelt. Ett og samme XML-dokument kan bli presentert på flere ulike måter ved bruk av *stilark*. For eksempel kan vi la en servlet produsere XML i stedet for HTML, og heller gjøre det presentasjonstekniske med stilark. XML er definert ved en internasjonal standard, og har blitt en viktig teknologi, blant annet knyttet til *web-databaser*. XML kan brukes som overføringsformat mellom databasesystemer. Java har ferdige klasser for å manipulere XML-data. Nettsiden *www.w3.org* er en god startside for å finne ut mer om HTML, XML og andre teknologier relatert til internett og web.

RMI (Remote Method Invocation) gjør det mulig å kalle metoder i programmer som utføres på andre maskiner. RMI kan brukes for å lage distribuerte løsninger, der klientene er programmert i Java. *SOAP* (Simple Object Access Protocol) er en standard der XML brukes for å implementere RMI-lignende funksjonalitet ved at metodekall og returverdier blir kodet i XML. SOAP kan brukes for å bygge *web-tjenester* som fungerer på tvers av *brannmurer*. *JWSDP* (Java Web Services Development Pack) er en samlepakke som består av Apache web-tjener (*www.apache.org*), Tomcat servlet-maskin (*jakarta.apache.org/tomcat*), og ulike Java-verktøy for XML og SOAP. [Nagappan m.fl.] og [Singh m.fl.] behandler flere teknologier og biblioteker for å bygge web-tjenester med Java.

Litteratur

Bodoff S, Armstrong E, Ball J, Bode Carson D (2004) **The J2EE Tutorial**. 2. utgave, Addison-Wesley.

Brown S, Burdick R, Falkner J, Galbraith B, Johnson R, Kim L, Kochmer C, Kristmundsson T, Li S (2001) **Professional JSP**. 2. utgave, Wrox Press.

Connolly T, Begg C (2005) **Database Systems: A Practical Approach to Design, Implementation, and Management**. 4. utgave, Addison-Wesley.

Fisher M, Ellis J, Bruce J (2003) **JDBC API, Tutorial, and Reference**. 3. utgave, Addison-Wesley.

Goldfarb CF, Prescod P (2003) **Charles F. Goldfarb's XML Handbook**. 5. utgave, Prentice Hall.

Lervik E, Havdal VB (2003) **Programmering i Java**. 2. utgave, Gyldendal Norsk Forlag.

Melton J (2003) **Advanced SQL:1999. Understanding Object-Relational and Other Advanced Features**. Morgan Kaufmann.

Nagappan R, Skoczylas R, Patel Sriganesh R (2002) **Developing Java Web Services: Architecting and Developing Secure Web Services Using Java**. Wiley.

Price J (2001) **Java Programming with Oracle SQLJ**. O'Reilly.

Price J (2004) **Oracle Database 10g SQL**. McGraw-Hill/Osborne.

Reese G (2000) **Database Programming with JDBC and Java**. 2. utgave, O'Reilly.

Shah N (2005) **Database Systems Using Oracle: A Simplified Guide to SQL and PL/SQL**. 2. utgave, Prentice Hall.

Singh I, Brydon S, Murray G, Ramachandran V, Violleau T, Stearns B (2004) **Designing Web Services with the J2EE(TM) 1.4 Platform: JAX-RPC, SOAP, and XML Technologies**. Addison-Wesley, Java Series.

Tyagi S, Vorburger M, McCammon K, Bobzin H, McCannon K (2004) **Core Java Data Objects**. Prentice Hall.

HiT skrift / HiT Publication

Bjørn Kristoffersen Introduksjon til databaseprogrammering med Java. 33 s.	7/2004	kr. 100
Inger M. Oellingrath Kosthold, kroppslig selvbilde og spiseproblemer blant ungdom i Porsgrunn. 45 s.	6/2004	kr. 110
Svein Roald Moen Knud Lyne Rahbeks Dansk Læsebog og Exempelsamling til de forandrede lærde Skolers Brug. 491 s.	5/2004	kr. 450
Jan Heggenes Fører kraftutbygging til økt genetisk mangfold hos ørret? 25 s.	4/2004	kr. 90
Tangen, Jan Ove, red. Kyststien – tre perspektiver. 27 s.	3/2004	kr. 90
Jan Ove Tangen Idrettsanlegg og anleggsbrukere-tause forventninger og taus kunnskap. 59 s.	2/2004	kr. 100
Greta Hekneby Fonologisk bevissthet og lesing. 43 s.	1/2004	kr. 85
Ingunn Fjørtoft og Tone Reiten Barn og unges relasjoner til natur og friluftsliv. 83 s.	10/2003	kr. 140
Else Marie Halvorsen Teachers' understanding of culture and of transference of culture. 40 s.	9/2003	kr. 80
P.G. Rathnasiri and Magnar Ottøy Oxygen transfer and transport resistance across Silicone tubular membranes. 31 s.	8/2003	kr. 170
Else Marie Halvorsen Den estetiske dimensjonen og kunstfeltet - ulike tilnærminger. 17 s.	7/2003	kr. 70
Else Marie Halvorsen Estetisk erfaring. En fenomenlogisk tilnærming i Roman Ingardens perspektiv. 12 s.	6/2003	kr. 60
Steinar Kjosavik Fra forming til kunst og håndverk, fagutvikling og skolepolitikk 1974-1997. 48 s.	5/2003	kr. 90
Olav Solberg, Herleik Baklid, Peter Fjågesund (red.) Tekst og tradisjon. M. B. Landstad 1802-2002. 106 s.	4/2003	kr. 126
Ella Melbye Hovedfagsoppgaver i forming Notodden 1976-1999. Faglig innhold sett i lys av det å forme. 129 s. 1 CD-rom	3/2003	kr. 210
Olav Rosef m.fl. *Escherichia coli*-bakterien som alle har –men som noen blir syke av – en oversikt. 22 s.	2/2003	kr. 66
Olav Rosef m.fl. Forekomsten *av E.coli* O157 ("hamburgerbakterien") hos storfe i Telemark og i kjøttdeig fra Trøndelag. 25 s.	1/2003	kr. 70
Roy Istad Oppretting av polygon. 24 s.	3/2002	kr. 30
Ella Melbye (red.) Hovedfagsstudium i forming 25 år. 81 s.	2/2002	kr. 70
Vassli, Idar Wayne A. Grudems profetiforståelse. 27 s.	1/2002	kr. 40
Olav Rosef m.fl. Hjorten (Cervus elaphus atlanticus) i Telemark. 29 s.	1/2001	kr. 100
Else Marie Halvorsen Kulturforståelse hos lærere i Telemark anno 2000. 51 s.	4/2000	kr. 50
Norvald Fimreite, Bjarne Nenseter and Bjørn Steen Cadmium concentrations in limed and partly reacidified lakes in Telemark, Norway. 16 s.	3/2000	kr. 20

Tåle Bjørnvold	2/2000	kr. 55
Minimering av omstillingstider ved produksjon av høvellast. 65 s.		
Sunil R. de Silva (ed.)	1/2000	kr. 1200
International Symposium. Reliable Flow of Particulate Solids III. Proceedings.11.- 13. August 1999, Porsgrunn, Norway. Vol. 1-2		

HiTskrift kan bestilles fra Høgskolen i Telemark, kopisenteret i Bø: Kopisenter@bo.hit.no, tlf. 35952834 eller på internett: http://www.hit.no/main/content/view/full/1201

HiT notat / HiT Working Paper

Roy M. Istad: Tettere studentoppfølging? Undervegsrapport fra et HiT-internt prosjekt. 15 s.	1/2004	kr. 70
Eli Thorbergsen m.fl. "Kunnskapens tre har røtter…" Praksisfortellinger fra barnehagen. En FOU-rapport. 42 s.	5/2003	kr. 130
Per Arne Åsheim (ed.) Science didactic. Challenges in a period of time with focus on learning processes and new technology. 54 s.	4/2003	kr. 140
Roald Kommedal and Rune Bakke Modeling Pseudomonas aeruginosa biofilm detachment. 29 s.	3/2003	kr. 130
Elisabeth Aase Ledelse i undervisningssykehjem. 27 s., vedlegg	2/2003	kr. 90
John Heggenes og Knut H. Røed Genetisk undersøkelse av stamfisk av ørret fra Måna, Tinnsjø. 10 s.	1/2003	kr. 70
Erik Halvorsen (red.) Bruk av Hypermedia og Web-basert informasjon i naturfagundervisningen. Presentasjon og kritisk analyse. 69 s.	2/2002	kr. 150
Harald Klempe Overvåking av grunnvannsforurensning fra Revdalen kommunale avfallsfylling, Bø i Telemark. Årsrapport 2000. 24 s.	1/2002	kr. 30
Jan Ove Tangen Kompetanse og kompetansebehov i norske golfklubber. 12 s.	6/2001	kr. 20
Øyvind Risa Evaluering av Musikk 1. 5 vekttall. Desember 2000. Høgskolen i Telemark, Allmennlærerutdanninga på Notodden. 39 s.	5/2001	kr. 40
Harald Klempe Overvåking av grunnvannsforurensning fra Revdalen kommunale avfallsfylling, Bø i Telemark. Årsrapport 1999. 22. s.	4/2001	kr. 30
Harald Klempe Overvåking av grunnvannsforurensning fra Revdalen kommunale avfallsfylling, Bø i Telemark. Årsrapport 1998. 22 s.	3/2001	kr. 30
Sigrun Hvalvik Tolking av historisk tekst – et hermeneutisk perspektiv. Et vitenskapsteoretisk essay. 28 s.	2/2001	kr. 30
Sigrun Hvalvik Georg Henrik von Wright: Explanation of the human action : an analysis of von Wright's assumptions form the perspective of theory development in nursing history. 27 s.	1/2001	kr. 30
Arne Lande og Ralph Stålberg (red.) Bruken av Hardangervidda – ressurser, potensiale, konflikter. Bø i Telemark 8.-9. april 1999. Seminarrappport. 57 s.	3/2000	kr. 50
Nils Per Hovland Studentar i oppdrag : ein rapport som oppsummerer utført arbeid og røynsler frå prosjektet "Nyskaping som samarbeidsprosess mellom SMB og HiT", 1998-2000. 24 s.	2/2000	kr. 30
Jan Heggenes Undersøkelser av gyteplasser til ørret i Tinnelvas utløp fra Tinnsjø (Tinnoset), Notodden i Telemark, 1998. 7 s.	1/2000	kr. 20

HiT notat kan bestilles fra Høgskolen i Telemark, kopisenteret i Bø: Kopisenter@bo.hit.no, tlf. 35952834 eller på internett: http://www.hit.no/main/content/view/full/1201