Kapittel 3

Bruk av objekter

I dette kapitlet vil vi opprette objekter fra ferdiglagde klasser, og håndtere dem ved hjelp av referansevariabler. Strenger og tabeller brukes mye i programmering, og Java tilbyr støtte for å håndtere slike objekter. Vi vil også se på flere måter å gjenta utføring av setninger ved hjelp av løkker på, og en ny flervalgsetning. Til slutt ser vi på generering av slumptall. Dette kan brukes i flere sammenhenger, for eksempel i spill og simuleringer.

3.1 Innføring i objektmodellen

Alle fagfelt har terminologi som må beherskes dersom man skal sette seg inn i fagfeltet. Dette gjelder også programmering. Målet med dette avsnittet er å innføre noen viktige begreper uten å gå i detaljer. Detaljene kommer etter hvert som vi bygger opp begrepsapparatet.

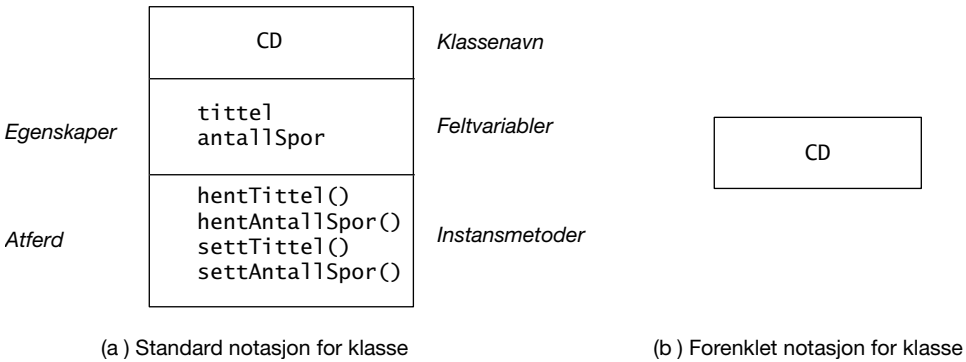
Abstraksjoner, klasser og objekter

Vi mennesker bruker abstraksjoner for å håndtere mangfoldet som møter oss i hverdagen. En *abstraksjon* omfatter relevante egenskaper ved et objekt, slik at vi kan skille det fra andre typer objekter. Både en Volvo S40 og en Toyota Avensis vil oppfattes som kjøretøy. Vår evne til å lage en abstraksjon av hva vi oppfatter som kjøretøy vil hjelpe oss med det. Fargen eller modellen er ikke vesentlig, det avgjørende er at det kan kjøres.

Når vi lager programmer som bearbeider informasjon om abstraksjoner fra virkeligheten, er det nyttig å representere *egenskapene* og *atferden* til disse abstraksjonene i datamaskinen. I Java kan abstraksjoner nedfelles som klasser. En *klasse* beskriver objekter av en bestemt type - hva de inneholder, og hva man kan gjøre med dem, mao. den spesifiserer egenskaper og atferd til disse objektene.

La oss illustrere forholdet mellom en klasse og dens objekter ved å spesifisere en enkel klasse for musikk-CD-er. En slik CD har visse egenskaper: en tittel og et visst antall spor. La oss anta at navnet på artisten eller på sporene ikke er viktige i denne sammenhengen. For en CD må det være mulig å finne ut hva tittelen er, og hvor mange spor det er på den. En abstraksjon der vi bruker en datamaskin, trenger ikke å være en eksakt avbildning av virkeligheten. I vår abstraksjon av en CD vil vi tillate å endre tittel og antall spor på CD-en, mens dette ikke er vanlig i virkeligheten. For å gjøre eksemplet enkelt er vi heller ikke opptatt av vanlig atferd til en CD, for eksempel å spille den.

Figur 3.1 *Klassenotasjon i UML*



Figur 3.2 *Klassedeklarasjon i Java*

```
    Klassenavn
class CD {
  // Deklarasjoner av feltvariabler                          Egenskaper
  String tittel;
  int    antallSpor;
  // Deklarasjoner av instansmetoder                         Atferd
  String hentTittel()     { return tittel; }
  int    hentAntallSpor() { return antallSpor; }
  void   settTittel(String nyTittel) { tittel = nyTittel; }
  void   settAntallSpor(int nSpor)   { antallSpor = nSpor; }
}
```

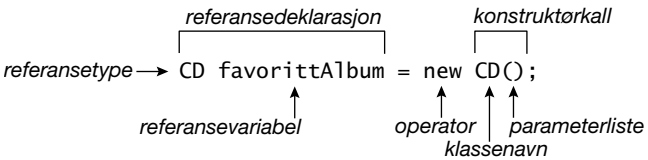
En *klassedeklarasjon* inneholder en rekke *deklarasjoner* som definerer egenskaper og atferd til klassens objekter. Figur 3.1 viser klassen `CD` i UML-notasjon (se vedlegg G). Tilsvarende klassedeklarasjon som Java-kode er vist i Figur 3.2. En klassedeklarasjon spesifiseres alltid med nøkkelordet `class` sammen med navnet på klassen. Deklarasjoner i klassen er alltid omsluttet av klammeparenteser, `{ }`.

Egenskapene til klassens objekter spesifiseres ved hjelp av *feltvariabler* (ofte bare kalt *felt*, eng. *field*). Deklarasjonen til klassen `CD` angir to feltvariabler, `tittel` og `antallSpor`, som lagrer opplysninger om henholdvis tittel og antall spor på en CD. Atferden til klassens objekter er angitt ved *instansmetoder*. Klassedeklarasjonen angir fire metoder, som har navnene `hentTittel`, `hentAntallSpor`, `settTittel` og `settAntallSpor`. Disse metodene kan brukes til å hente og endre opplysninger om tittel og antall spor på en CD. En metode inneholder handlinger som utøver ønsket atferd når metoden blir utført.

Objekter, referanseverdier og referansevariabler

En klasse er en «arbeidstegning» som kan brukes til å opprette objekter med egenskaper og atferd definert av klassen. I litteraturen brukes også begrepet *instans* (eng. *instance*) som synonym for et objekt. Vi vil alltid ha én `CD`-klasse, men vi kan opprette flere `CD`-objekter. Når vi oppretter et objekt fra en klasse, får vi en *referanseverdi* til det nyopprettede objektet. Hvert objekt av en klasse er unikt (dvs. har entydig identitet gitt ved referanseverdien), selv om alle objektene til klassen vil ha samme egenskaper og atferd.

Figur 3.3 *Objektopprettelse*



En *referansevariabel* (ofte forkortet til *referanse*) er en variabel som kan lagre en *referanseverdi* til et *objekt*. Analogt med variabler av primitive datatyper som lagrer verdier av primitive datatyper, lagrer referanser referanseverdier til objekter. Vi sier at referansevariabelen *refererer til* dette objektet. Vi får tak i objekter via referanser.

En *referansedeklarasjon* brukes til å deklarere en referansevariabel. Den angir navn på referanse og *referansetype*. Klassenavn er en referansetype. Referansen kan bare referere til objekter som har denne typen. Følgende referansedeklarasjon vil føre til at minne blir tildelt for referansen `favorittAlbum`, som kan lagre referanseverdi til objekter av klassen `CD`:

```
CD favorittAlbum;
```

Legg merke til at det *ikke* blir opprettet noe objekt som følge av deklarasjonen ovenfor. Følgende syntaks er ansvarlig for opprettelse av et objekt av klassen `CD`:

```
new CD();
```

Denne språkkonstruksjonen består av to ledd: operatoren `new` og et *konstruktørkall*, `CD()`. Operatoren `new` oppretter minnet for et objekt og returnerer referanseverdien til det nye objektet. Hvert objekt som blir opprettet, får egne kopier av feltvariabler (jf. Figur 3.2). Konstruktørkallet angir klassen og en *parameterliste* (tom i dette tilfellet) med opplysninger som kan brukes blant annet til å sette spesifikke verdier i feltvariabler til det nye objektet opprettet fra den angitte klassen.

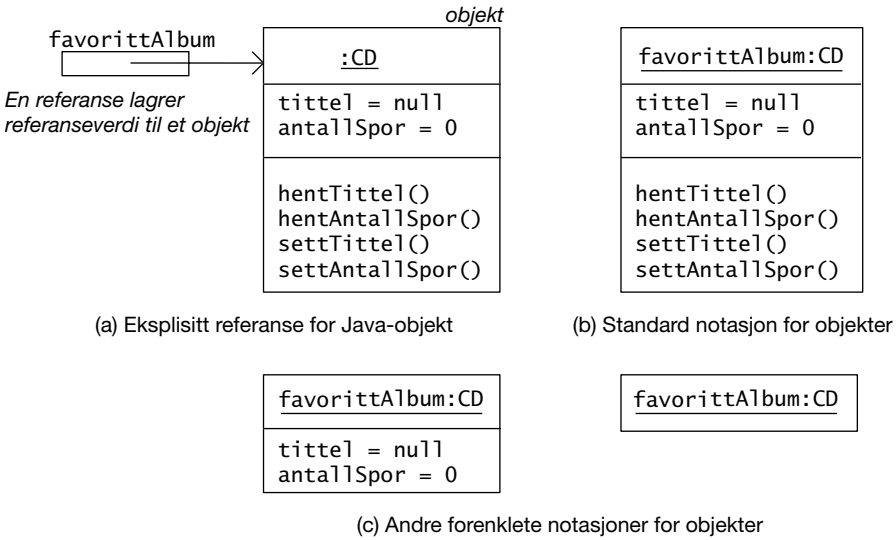
Ofte kombinerer vi både deklarasjon av referansen *og* opprettelse av et objekt, slik det er gjort i Figur 3.3. Referanseverdien til det nyopprettede objektet blir tilordnet referansen:

```
CD favorittAlbum = new CD();
```

Figur 3.4a viser resultatet av utføringen av kodelinjen ovenfor i UML-notasjon med eksplisitt angivelse av en referanse til det nyopprettede objektet. Vi ser at feltvariablene `tittel` og `antallSpor` har henholdvis verdiene `null` og 0. Dette vil alltid være tilfellet for hvert `CD`-objekt vi oppretter med `new`-operatoren, som er basert på vår deklarasjon av klassen `CD`.

Notasjonen i Figur 3.4a kan virke tungvint, slik at vi ofte bruker alternative notasjoner, vist i Figur 3.4b og Figur 3.4c. I UML merkes typisk objekter med både referansenavn og klassenavn på formen *referansenavn*:*Klassenavn*.

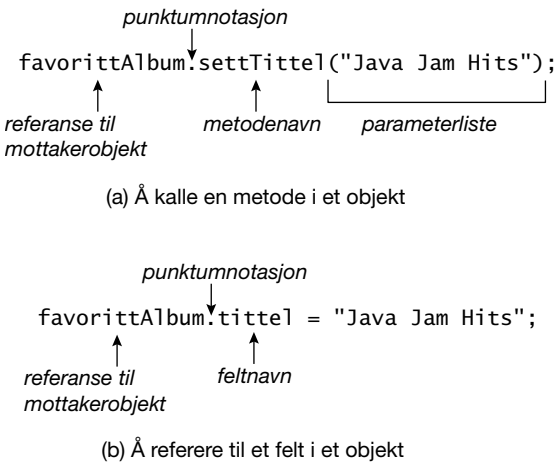
Figur 3.4 *Objektnotasjon i UML*



Å bruke objekter

Etter at et objekt er opprettet, kan en referanse til objektet brukes til å sende *meldinger* til objektet. Meldinger tar form av et *metodekall* i Java (se Figur 3.5a).

Figur 3.5 *Punktumnotasjon*



Et metodekall til et objekt angir

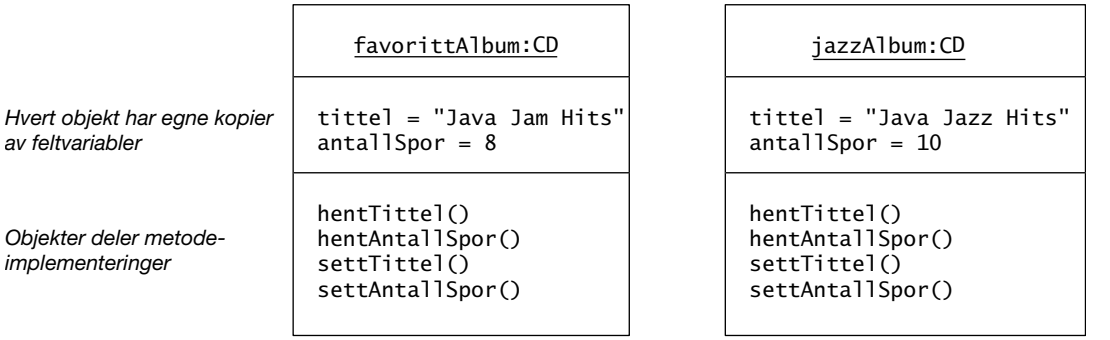
- referansen til mottakerobjektet
- navnet på metoden som skal utføres
- eventuelle opplysninger (parametere) som metoden trenger for å utføre sine handlinger (se delavsnittet «Metodekall og aktuelle parameteruttrykk» på side 118)

Legg merke til punktumtegnet (`.`) mellom referansen og metodenavnet. Klassen til det refererte objektet må definere metoden som blir kalt. I Figur 3.5a vil metodekallet føre til utføring av metoden `settTittel()` fra klassen `CD`. Fra Figur 3.2 ser vi at denne metoden tilordner verdien den får som parameter til feltet `tittel`, altså vil feltet `tittel` i objektet referert ved referansen `favorittAlbum` bli tilordnet referanseverdien til `String`-objektet `"Java Jam Hits"`.

Vi kan også bruke punktumnotasjonen sammen med en referanse for å referere til feltvariabler i et objekt. Et eksempel på dette er vist i Figur 3.5b. Kodelinjen i Figur 3.5b vil gi samme resultat som metodekallet i Figur 3.5a, nemlig at feltet `tittel` i objektet referert ved referansen `favorittAlbum` blir satt til å referere til `String`-objektet `"Java Jam Hits"`.

Felt til forskjellige objekter av klassen `CD` kan ha forskjellige verdier, slik vi ser i Figur 3.6. Hvert objekt har egne kopier av feltvariabler. Verdiene til objektets felt utgjør til enhver tid objektets *tilstand*. Atferden til et objekt er gitt ved instansmetoder. Kodingen av en metode, dvs. koden som utgjør en metodedeklarasjon, kalles en *metodeimplementering* (eng. *method implementation*), og objekter av samme klasse deler metodeimplementeringer (mer detaljer om det i avsnitt 4.3).

Figur 3.6 *Objekttilstand*



Program 3.1 definerer en klasse (`CDSampler`) som bruker objekter av klassen `CD`. Det er en nyttig øvelse å identifisere begrep som er innført hittil i dette kapitlet, i denne koden. Opplysninger som programmet skriver ut, tilsvarer tilstanden til `CD`-objektene illustrert i Figur 3.6.

Program 3.1 *Objekter*

```
public class CDSampler {
  public static void main(String args[]) {
    // Opprett 2 CD-er.
    CD favorittAlbum = new CD();
    CD jazzAlbum = new CD();
```

```
    // Sett tilstand til CD-ene.
    favorittAlbum.settTittel("Java Jam Hits");
    favorittAlbum.settAntallSpor(8);
    jazzAlbum.settTittel("Java Jazz Hits");
    jazzAlbum.settAntallSpor(10);

    // Skriv tilstand til CD-ene.
    System.out.println("Tittel på favorittalbum: " + favorittAlbum.hentTittel());
    System.out.println("Antall spor på favorittalbum: " +
                       favorittAlbum.hentAntallSpor());
    System.out.println("Tittel på jazzalbum: " + jazzAlbum.hentTittel());
    System.out.println("Antall spor på jazzalbum: " + jazzAlbum.hentAntallSpor());
  }
}
```

Utdata fra programmet:

```
Tittel på favorittalbum: Java Jam Hits
Antall spor på favorittalbum: 8
Tittel på jazzalbum: Java Jazz Hits
Antall spor på jazzalbum: 10
```

3.2 Strenger

Tekst er en vanlig form for representasjon av informasjon. Tekst består av *tegn*. I programmeringsspråk fremstår tekst som *en sekvens av tegn* og kalles *tekststrenger* eller bare *strenger*. Java tilbyr en primitiv datatype, `char`, og en ferdiglaget klasse, `String`, som kan brukes til å håndtere henholdvis tegn og strenger.

Tegn og strenger

I datamaskinen blir hvert tegn representert ved hjelp av en tallkode. Dette gjelder alle tegn, inklusive alle bokstaver, alle siffer og andre spesielle tegn. Java bruker en standard som kalles Unicode for å representere tegn. Denne standarden sier hvilken tallkode et tegn skal ha.

Den primitive datatypen `char` representerer tegn ved en 16-bits tallkode, slik at det er mulig å representere 65 536 (2^{16}) forskjellige tegn i denne datatypen. Dette er nok til å representere de fleste tegn som inngår i verdens språk. Unicode-verdier angis vanligvis i heksadesimale sifre. For eksempel har bokstaven a Unicode-verdi `\u0061`, sifferet `0` har Unicode-verdi `\u0030`, og tegnet Ÿ har Unicode-verdi `\u0178`. Prefikset `\u` angir at det er et Unicode-tegn.

For å angi et tegn som en *tegnverdi* i et Java-program må vi omslutte enten tegnet eller Unicode-verdien til tegnet i enkle apostrofer (`'`), for eksempel kan bokstav `a` angis enten som `'a'` eller som `'\u0061'`. Denne notasjonen definerer et *tegnlitteral* som representerer en bestemt tegnverdi. Uten apostrofer vil tegnet `a` alene bli tolket som et enbokstavs navn i programmet. Noen eksempler på tegnlitteraler er gitt i Tabell 3.1. Som vist i tabellen må vi plassere en bak-

overskråstrek (\) foran en enkel apostrof ('), og begge tegnene (\') må omsluttes i enkle apostrofer ('\'') for å angi en enkel apostrof (') som en tegnlitteral. Enkelte spesielle tegn, som ikke har synlig representasjon, men har spesiell betydning (for eksempel linjeskift med Unicode-verdi \u000a), har predefinerte litteraler (for eksempel linjeskiftlitteral '\n').

Tabell 3.1 *Tegn, Unicode-representasjon og tegnlitteraler*

Tegn	Desimalverdi	Unicodeverdi	Tegnlitteral
0 (siffer null)	48	\u0030	'0'
a	97	\u0061	'a'
?	63	\u003f	'?'
apostrof: '	39	\u0027	'\''
anførselstegn: "	34	\u0022	'\"'
bakoverskråstrek: \	92	\u005c	'\\'
linjeskift	10	\u000a	'\n'
tabulator	9	\u0009	'\t'
mellomrom	32	\u0020	' '

Den primitive datatypen char kan brukes til å deklarere variabler som kan lagre tegnverdier. En tegnlitteral har datatype char.

```
char linjeskift = '\n', tabulator = '\u0009';
char tegn1, tegn2, tegn3, tegn4;
tegn1 = tegn4 = 'a'; tegn2 = tegn3 = 'b';
```

Siden tegn representeres som heltallskode, kan et tegn inngå som en heltallsoperand i et aritmetisk uttrykk, slik følgende eksempler viser:

```
int sumTallkode = tegn1 + tegn2 + tegn3 + tegn4;  // 97+98+98+97 => 390
int tall = '5' - '0';                             // 53 - 48 => 5
```

Tallkodene til små bokstaver (a til z), store bokstaver (A til Z) og alle sifre (0 til 9) er nummerert sekvensielt i Unicode-tegnsettet, med små bokstaver fra \u0061 (a) til \u007a (z), store bokstaver fra \u0041 (A) til \u005a (Z) og sifre fra \u0030 (0) til \u0039 (9) (se også vedlegg D). Vi kan sammenligne tegn, og da er det tallkodene til tegnene som blir sammenlignet:

```
boolean test1 = tegn1 == tegn4; // sann
boolean test2 = tegn1 > tegn2;  // usann,
                                // siden 'a' (\u0061) < 'b' (\u0062)
```

Analogt med tegnlitteraler kan vi definere *strenglitteraler* ved å omslutte en sekvens av tegn i anførselstegn ("). I motsetning til tegnlitteraler som er heltallsverdier, er strenglitteraler ekte objekter. For eksempel definerer strenglitteralen "abba" et objekt av klasse String. Dette

objektet lagrer tegnene `'a'`, `'b'`, `'b'` og `'a'`. Andre eksempler på strenglitteraler er gitt i Tabell 3.2. Den viser også resultatet dersom strengene blir skrevet ut, for eksempel ved kall til `System.out.println()`-metoden. Legg merke til at anførselstegn i selve strengen må «beskyttes» med bakoverskråstrek, og at en strenglitteral ikke kan spenne over flere linjer i kildekoden.

Tabell 3.2 *Strenglitteraler*

Strenglitteral	Utskrift
`"Velkommen til Portveien 2"`	`Velkommen til Portveien 2`
`""`	Den tomme strengen har ingen tegn som kan skrives ut.
`"!"`	`!`
`"\"Skjerp deg!\" sa kongen."`	`"Skjerp deg!" sa kongen.`
`"En streng kan ikke gå over flere linjer."`	Kompileringsfeil.
`"Brekk en streng\n med linjeskiftlitteral."`	`Brekk en streng` ` med linjeskiftlitteral.`

Strengsammenslåing

Som med andre referansevariabler kan vi deklarere variabler av klassen `String` som kan referere til strenglitteraler:

```
String fornavn = "Leif", etternavn = "Eriksen";
```

Tegnene lagret i et `String`-objekt kan ikke endres, dvs. at et `String`-objekt er *uforanderlig* (eng. *immutable*). Å kalle metoder i `String`-klassen som tilsynelatende endrer strengen i et `String`-objekt, resulterer i et nytt `String`-objekt med den modifiserte strengen. Vi skal se flere eksempler på dette senere i avsnittet.

En nyttig operasjon vi kan gjøre på strenger, er å slå sammen to strenger for å lage en ny streng som inneholder tegn fra den første etterfulgt av tegn fra den andre. Denne *strengsammenslåingen* kalles også *konkatenering* eller *sammenkjeding* (eng. *concatenation*). Til dette formålet brukes den binære operatoren +, som også brukes til aritmetisk addisjon:

```
String fullstendigNavn = fornavn + " " + etternavn; // "Leif Eriksen"
```

Dersom en av operandene til denne operatoren er en streng, vil den andre operanden blir konvertert til en streng dersom den ikke er det før strengsammenslåingen. Strengsammenslåingen skjer fra venstre mot høyre og resulterer i et nytt `String`-objekt som inneholder den endelige strengen. Strengsammenslåingen returnerer referansen til det nye `String`-objektet. Etter utføringen av kodelinjen ovenfor, vil referansen `fullstendigNavn` referere til strenglitteralen `"Leif Eriksen"`.

Program 3.2 viser en del eksempler på strengsammenslåing. Etter utføring av linje (1) vil referansen `kurs` referere til strengen `"Grunnkurs i programmering"`. Merk forskjellen på resultatet i linjene (2) og (3). I linje (2) slår vi sammen strenglitteralene `"I"`, `"N"` og `"F"` med heltallsvariabelen `kursnummer`, som resulterer i strengen `"INF100"`, siden heltallsverdien 100 fra variabelen `kursnummer` blir konvertert til strengen `"100"`. I linje (3) *adderer* vi tegnlitteralene `'I'` (heltallsverdi 73), `'N'` (heltallsverdi 78) og `'F'` (heltallsverdi 70) med heltallsverdien i variabelen `kursnummer` (dvs. heltallsverdi 100), som resulterer i heltallsverdien 321. Denne heltallsverdien blir konvertert til strengen `"321"` som blir sammenslått med strenger i de øvrige operandene.

Program 3.2 *Strengsammenslåing*

```
public class Konkatenering {

  public static void main(String args[]) {
    String kurs = "programmering";
    kurs = "Grunnkurs i " + kurs;                                  // (1)
    System.out.println("kurs:\t" + kurs);
    int kursnummer = 100;
    String kurs1 = "I" + "N" + "F" + kursnummer + ": " + kurs; // (2)
    String kurs2 = 'I' + 'N' + 'F' + kursnummer + ": " + kurs; // (3)
    System.out.println("kurs1:\t" + kurs1);
    System.out.println("kurs2:\t" + kurs2);
  }
}
```

Utdata fra programmet:

```
kurs: Grunnkurs i programmering
kurs1:  INF100: Grunnkurs i programmering
kurs2:  321: Grunnkurs i programmering
```

Å opprette strengobjekter

Tidligere i dette avsnittet har vi sett hvordan en variabel kan tilordnes en strenglitteral. Angivelse av en strenglitteral fører implisitt til opprettelse av et `String`-objekt som inneholder tegnstrengen, og det er referanseverdien til dette objektet som blir tilordnet variabelen:

```
String stjerne = "madonna";
```

Dersom flere variabler tilordnes samme strenglitteral, vil alle disse variablene være *aliaser*. De vil referere til samme `String`-objekt som strenglitteralen refererer til (se Figur 3.7).

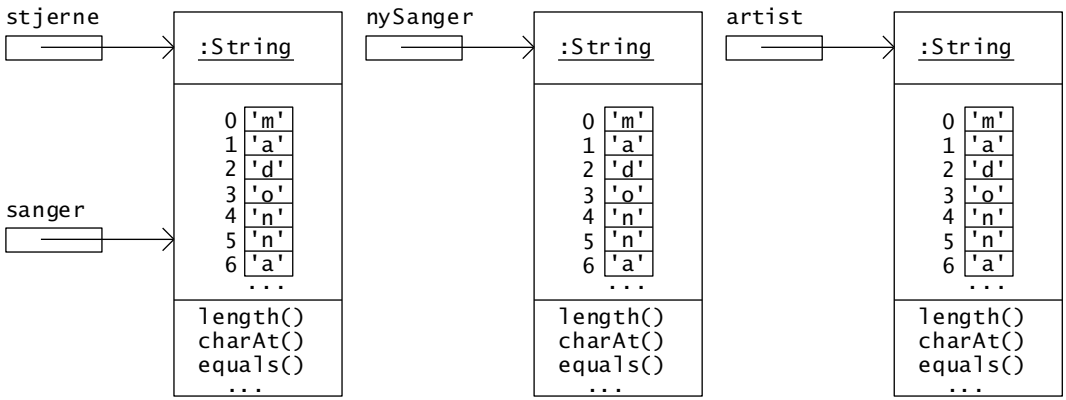
```
String sanger = "madonna"; // referansen sanger betegner samme
                           // String-objektet som referansen stjerne.
```

En annen måte å opprette `String`-objekter på er å bruke `new`-operatoren sammen med et `String`-konstruktørkall:

```
String nySanger = new String("madonna"); // (1)
String artist   = new String(nySanger);  // (2)
```

Bruk av `new`-operatoren fører til opprettelsen av et *nytt* `String`-objekt. Linje (1) oppretter et nytt `String`-objekt basert på strenglitteralen `"madonna"`, og linje (2) oppretter et nytt `String`-objekt basert på `String`-objektet referert ved variabelen `nySanger`. Variablene `stjerne`, `nySanger` og `artist` refererer til tre forskjellige `String`-objekter som har samme tilstand (se Figur 3.7).

Figur 3.7 *Strengopprettelse*




```
String stjerne = "madonna";
String sanger  = "madonna";
String nySanger = new String("madonna"); // (1)
String artist   = new String(nySanger);  // (2)
```

Strengsammenligning

Sammenligning av strenger er basert på *leksikografisk ordning*, dvs. tegn i tilsvarende posisjon i de to strengene sammenlignes basert på sine Unicode-verdier. Telefonopplysninger i en telefonkatalog eller ord i en ordbok er oppført etter leksikografisk ordning. Vi sier at strengen `"abba"` er *mindre* enn strengen `"aha"` siden Unicode-verdien til det andre tegnet, `'b'`, i strengen `"abba"` er mindre enn Unicode-verdien til det andre tegnet, `'h'`, i strengen `"aha"`.

Metoden `compareTo()` kan brukes til å sammenligne to strenger. Vi kaller denne metoden på den ene strengen og sender den andre strengen som parameter (se avsnitt 4.3 på side 116). Returverdien fra kallet til denne metoden angir resultatet fra sammenligningen, slik det er vist i Tabell 3.3. I programmet kan man da teste resultatet for å bestemme videre handling.

```
int resultat1 = stjerne.compareTo(sanger);     // == 0
int resultat2 = stjerne.compareTo(nySanger);   // == 0

String gruppe1 = "abba", gruppe2 = "aha";
int resultat3 = gruppe2.compareTo(gruppe1);    // > 0
int resultat4 = gruppe1.compareTo(gruppe2);    // < 0
if (resultat4 < 0)  // Sann i dette tilfellet.
  System.out.println(gruppe1 + " er mindre!"); // Skriver: abba er mindre!
```

I Tabell 3.3 ser vi at metodene `compareTo()` og `equals()` tar en parameter av klassen `Object` (se avsnitt 6.2 på side 179). En slik parameter gjør det mulig å sende et hvilket som helst objekt som parameter til metoden. Kompilatoren vil godta dette, men under utføringen kan metoden feile dersom det viser seg at det ikke var et riktig objekt å sammenligne med.

Tabell 3.3 *Utvalgte metoder fra `String`-klassen*

`java.lang.String`	
`int compareTo(Object s2)`	Sammenligner to strenger. For eksempel, gitt kodelinjen: `int resultat = s1.compareTo(s2);` der `s1` og `s2` er strenger, kan vi konstatere følgende, avhengig av verdien i `resultat`: Dersom `resultat < 0`, er streng `s1` mindre enn streng `s2`. Dersom `resultat == 0`, er streng `s1` lik streng `s2`. Dersom `resultat > 0`, er streng `s1` større enn streng `s2`.
`boolean equals(Object s2)`	Sammenligner to strenger for likhet, dvs. om de respektive strengene har identisk sekvens av tegn, og returnerer sann dersom dette er tilfellet.
`int length()`	Returnerer antall tegn i strengen, dvs. *lengden* på strengen.
`static String valueOf(T t)`	Avhengig av type `T` returneres en strengrepresentasjon av verdien i `t`. Type `T` kan for eksempel være `boolean`, `char`, `double`, `float`, `int` eller `long`.
`char charAt(int indeks)`	Returnerer tegn angitt ved `indeks` i strengen. Første tegn i strengen har *indeks* 0. Ulovlig indeksverdi vil resultere i et unntak av typen `IndexOutOfBoundsException`.

`int indexOf(int tegnkode)`
`int indexOf(String delstreng)`
`int indexOf(int tegnkode,`
`int startIndeks)`
`int indexOf(String delstreng,`
`int startIndeks)`
	Returnerer indeks til `tegnkode` eller indeks til begynnelsen av `delstreng` i strengen, ellers returnerer den -1. Argumentet `startIndeks` kan brukes til å starte søkingen fra en bestemt indeks, ellers begynner søkingen i indeks 0 som er indeks til første tegn i strengen.

`String substring(int startIndeks,` `int sluttIndeks)`	Returnerer en ny streng som består av alle tegn f.o.m. `startIndeks` og t.o.m. `(sluttIndeks-1)` fra det aktuelle strengobjektet metoden blir kalt på. Den returnerte strengen har lengde `(sluttIndeks-startIndeks)`. Ulovlig indeksverdi vil resultere i et unntak av typen `IndexOutOfBoundsException`.
`String toLowerCase()` `String toUpperCase()`	Returnerer en ny streng der alle *bokstavene* fra den opprinnelige strengen er omgjort til henholdsvis enten små bokstaver eller store bokstaver.
`String trim()`	Returnerer en ny streng der usynlige tegn i begynnelsen og slutten av den opprinnelige strengen er fjernet. Disse usynlige tegnene kan være for eksempel mellomromtegn, tabulatortegn og linjeskifttegn.

Likhet mellom to strenger betyr at begge String-objektene har tegnstrenger med identisk *innhold*. Dersom man bare vil sammenligne to strenger for likhet basert på innhold, er det enklere å bruke equals()-metoden enn compareTo()-metoden.

```
boolean flaggA = stjerne.equals(sanger);          // sann
boolean flaggB = stjerne.equals(nySanger);        // sann
```

Det kan være fristende å bruke likhetsoperatoren, ==, til å sammenligne to strenger for likhet, men det vil ikke gi forventet resultat. Likhetsoperatoren sammenligner to *referanser* for å avgjøre om de refererer til det samme objektet, dvs. om de er aliaser.

```
boolean flagg1 = (stjerne == sanger)      // sann
boolean flagg2 = (stjerne == nySanger)    // usann
```

Konvertering mellom primitive verdier og strenger

Svært ofte er det nødvendig å konvertere primitive verdier til strenger. Metoden valueOf() i String-klassen vil konvertere en primitiv verdi til et String-objekt som inneholder tegnrepresentasjon av denne verdien.

```
String tallStr = String.valueOf(3.14);           // "3.14"
String boolStr = String.valueOf(true);           // "true"
```

Dersom en streng inneholder tegnrepresentasjon av én primitiv verdi og vi ønsker å konvertere den til riktig primitiv type, er det best å bruke en passende parse*Type*()-metode fra wrapper-klasser (se avsnitt 3.4 på side 70).

Dersom en streng inneholder tegnrepresentasjon av flere primitive verdier, er det hensiktsmessig å bruke java.util.Scanner-klassen til å lese strengen og konvertere tegnene til primitive verdier.

```
Scanner avsøker = new Scanner("101 25,0 2005");        // (1)
int    kode = avsøker.nextInt();                       // 101
double moms = avsøker.nextDouble();                    // 25.0
long   år   = avsøker.nextLong();                      // 2005L
```

Legg merke til at flyttall i strengen er angitt ifølge regler som gjelder for *lokale tilpasninger* for et land. I Norge brukes komma (,) i stedenfor punktum (.) foran desimaler.

Andre nyttige strengmetoder

Tabell 3.3 viser et utvalg av metoder fra String-klassen. Klassen har svært mange nyttige metoder for opprettelse av nye String-objekter, og for sammenligning av, søking i og konvertering av String-objekter. Her skal vi nøye oss med å vise bruken av noen vanlige metoder.

Dersom indeksen i noen av disse metodene ikke er en lovlig verdi i intervallet f.o.m. 0 t.o.m. n-1, der n er strenglengden, kastes det et unntak av typen IndexOutOfBoundsException. Unntak signaliserer kjørefeil, og er omhandlet i kapittel 13. Program 3.3 viser bruk av noen av

metodene. Linjen (1) er kommentert bort, siden et `String`-objekt ikke kan sammenlignes med et `Integer`-objekt.

Program 3.3 *Bruk av diverse strengmetoder*

```
public class DiverseStrengMetoder {

  public static void main(String args[]) {
    String gruppe1 = "abba", gruppe2 = "aha";
    int resultat3 = gruppe2.compareTo(gruppe1);           // > 0
//  int resultat4 = gruppe2.compareTo(new Integer(10));   // (1) Kompileringsfeil!
    if (resultat3 > 0) // Sann i dette tilfellet.
      // "aha" er større leksikografisk.
      System.out.println(gruppe2 + " er større leksikografisk!");
    if (gruppe1.length() > gruppe2.length())      // 4 > 3
      // "abba" er større i lengde.
      System.out.println(gruppe1 + " er større i lengde!");

    String stjerne = "madonna";
    int lengde = stjerne.length();                // 7
    System.out.println(stjerne.charAt(lengde-4)); // o (indeks: 3, dvs 4.tegn)
    System.out.println(stjerne.indexOf('n'));     // 4
    System.out.println(stjerne.substring(0,3));   // "mad"
  }
}
```

Utdata fra programmet:

```
aha er større leksikografisk!
abba er større i lengde!
o
4
mad
```

3.3 Manipulering av referanser

En referanseverdi identifiserer et objekt i minnet. Et Java-objekt kan bare manipuleres via dets referanseverdi. Når referanseverdien er lagret i en referanse, kan vi bruke referansen til å håndtere objektet.

Referansetyper

De primitive datatypene definerer hvilke verdier som er lovlige og hvilke operasjoner som kan utføres på disse verdiene. Analogt med en primitiv datatype, definerer en klasse en datatype som vi kaller *referansetyper*. Senere i boken skal vi se på andre referansetyper enn klasser.

Tilordning

Akkurat som en `int`-variabel bare kan lagre `int`-verdier, kan en referansevariabel av en spesifikk referansetype bare lagre referanseverdier til objekter av denne typen. Analogt med at vi kan endre verdien i en `int`-variabel, kan vi endre referanseverdien i en referansevariabel. På denne måten kan en referansevariabel referere til forskjellige objekter av samme referansetype til forskjellige tider.

Ett objekt, flere referanser

Den samme referanseverdien kan tilordnes flere variabler. Tilordningen kopierer aldri objektene, men kun referanseverdien. Når en referanseverdi har blitt tilordnet flere variabler, kaller vi disse variablene aliaser for det objektet som blir referert ved referansevariablene. Objektet kan håndteres ved å bruke hvilket som helst av dets aliaser. Figur 3.7 viser at referansene `stjerne` og `sanger` er aliaser til samme `String`-objekt.

Litteralen `null`

Litteralen `null` er en spesiell referanseverdi som kan tilordnes hvilken som helst referanse. Referanseverdien null indikerer at referansen ikke refererer til noe objekt. Etter tilordning av null vil objektet som tidligere ble referert ved referansen, ikke lenger være tilgjengelig via denne referansen. Program 3.4 viser at en kjørefeil kan oppstå når vi bruker en referanse som har verdi `null`.

Sammenligning av objekter

Under delavsnittet «Strengsammenligning» på side 64 så vi på forskjellen mellom sammenligning av strenginnhold og sammenligning av strengreferanser. Vi skal generalisere diskusjonen om strenger til vilkårlige objekter, og vi skal begrense oss til objektlikhet. Hva betyr det at to biler er like eller to øl er like? For å sammenligne objekter av en klasse for likhet etter bestemte kriterier, må klassen gi en egen implementering av metoden `equals()`. Denne metoden har en egen stilling i Java, og brukes til å sammenligne to objekter for likhet. Metoden `equals()` burde kontrollere at det er mening i å sammenligne objektene. For eksempel er det ingen opplagt grunn til å sammenligne en øl med en bil, like lite som et `String`-objekt skal sammenlignes med et `CD`-objekt. Klassen `String` implementerer `equals()`-metoden for sammenligning av strenginnhold i to `String`-objekter. Vi kommer til å utdype emnet objektsammenligning når vi skal diskutere arv i kapittel 7.

Likhetsoperatoren, `==`, kan brukes til å finne ut om to referanser av samme type refererer til samme objekt, ved å sammenligne referanseverdier lagret i referansene. For et eksempel på om to referanser er aliaser, se delavsnittet «Ett objekt, flere referanser» på side 68.

Program 3.4 er et eksempel på hvordan to referanser kan ombyttes. Figur 3.8 illustrerer stegene i prosessen. For å få dette til, må vi bruke en ekstra referansevariabel. Legg merke til at

tilordningen kopierer referanseverdier og ikke objekter. Figuren viser også hvilke aliaser som blir laget underveis.

Utdata fra Program 3.4 viser at en referansevariabel (`gruppenavn`) med verdi `null` har strengrepresentasjon `"null"`. Under kjøringen avbryter programmet i linje (6) med beskjed om at et *unntak* av typen `NullPointerException` har funnet sted. Årsaken er at vi bruker referansen `gruppenavn` som har verdi `null`, til å kalle `length()`-metoden. Moralen er at en referanse må referere til et objekt *før* den brukes.

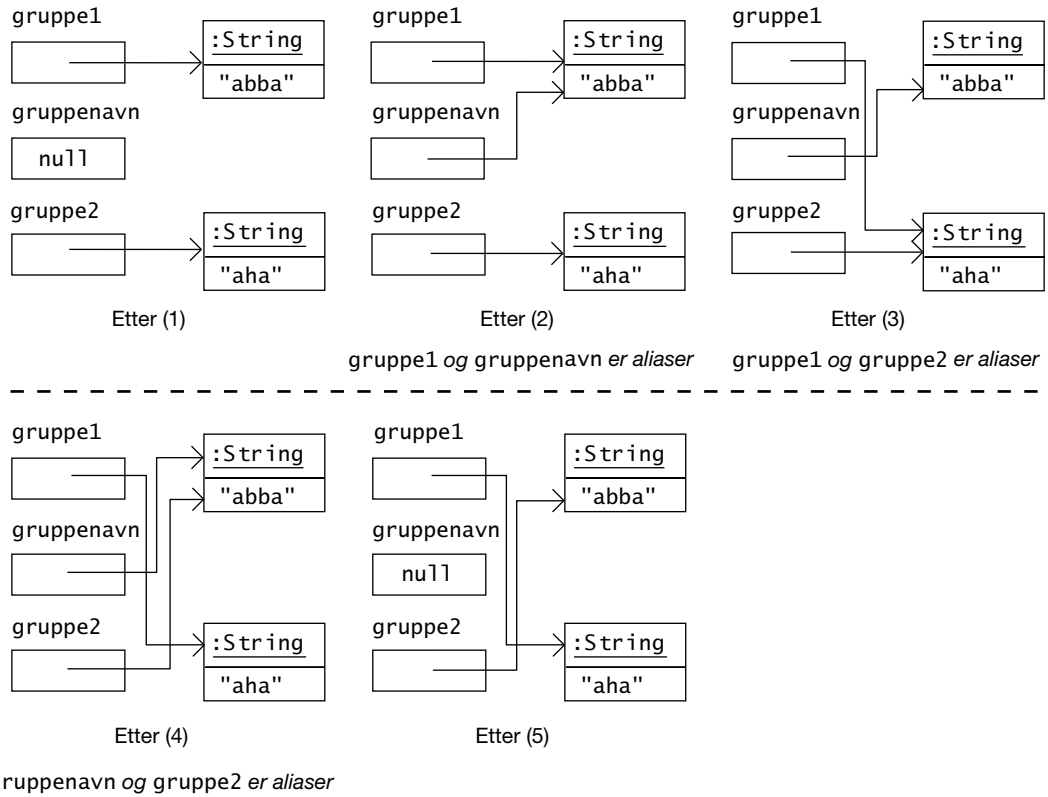
Program 3.4 *Ombytting av referanser*

```
public class ReferanseOmbytting {
  public static void main(String[] args) {
    String gruppe1 = "abba", gruppe2 = "aha", gruppenavn;   // (1)
    gruppenavn = gruppe1;                                   // (2)
    gruppe1 = gruppe2;                                      // (3)
    gruppe2 = gruppenavn;                                   // (4)
    gruppenavn = null;                                      // (5)
    System.out.println("gruppe1 refererer til: " + gruppe1);
    System.out.println("gruppe2 refererer til: " + gruppe2);
    System.out.println("gruppenavn refererer til: " + gruppenavn);
    System.out.println(gruppenavn.length());                // (6)
  }
}
```

Utdata fra programmet:

```
gruppe1 refererer til: aha
gruppe2 refererer til: abba
gruppenavn refererer til: null
Exception in thread "main" java.lang.NullPointerException
        at ReferanseOmbytting.main(ReferanseOmbytting.java:12)
```

Figur 3.8 *Ombytting av referanser*



3.4 Håndtering av primitive verdier som objekter

Java tilbyr innkapsling av primitive typer som heltall og flyttall, for eksempel `int` og `double`, slik at vi kan behandle primitive dataverdier som objekter. En klasse som kapsler inn en slik dataverdi, kalles en *wrapper-klasse* (eng. *wrapper class*).

Tabell 3.4 viser de klassene som Java definerer for å innkapsle primitive datatyper. Det er en wrapper-klasse for hver primitiv datatype, og navnet følger naturlig av typens navn, for eksempel blir `int`-verdier innkapslet vha. klassen `Integer`.

Tabell 3.4 *Wrapper-klasser for primitive verdier*

Primitiv datatype	Tilsvarende wrapper-klasse
`boolean`	`Boolean`
`char`	`Character`
`byte`	`Byte`

Primitiv datatype	Tilsvarende wrapper-klasse
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

Automatisk konvertering fra primitiv verdi til objekt

Konvertering fra primitive verdier til objekter av tilsvarende wrapper-klasser (eng. *auto-boxing)* er enkelt. Vi angir den primitive verdien der vi har bruk for tilsvarende objekt:

```
Integer iRef = 10;   // (1) Automatisk innkapsling, verdi 10 blir innkapslet.
```

Selv om typen til uttrykket på høyre side av tilordningsoperatoreren er `int` og typen til variabelen på venstre side er `Integer`, vil tilordningen gå riktig for seg. Verdien `10` vil automatisk bli innkapslet i et objekt av klassen `Integer`, og referanseverdien til det blir tildelt referansen `iRef`.

Automatisk konvertering fra objekt til primitiv verdi

Konvertering fra objekter av wrapper-klasser til primitive verdier (eng. *auto-unboxing*) reverserer innkapslingsprosessen. Vi angir bare referansen til wrapper-objektet der vi har bruk for tilsvarende primitiv verdi:

```
int j = iRef;       // (2) Automatisk utpakking, j får verdi 10.
```

Etter tilordningen er verdien `10`, innkapslet i wrapper-objektet referert ved referansen `iRef`, tilordnet variabelen `j`. De andre primitive datatypene kan pakkes inn på samme måte.

Eksplisitt konvertering mellom primitiv verdi og objekt

Det er også mulig å foreta eksplisitt omforming mellom primitive dataverdier og wrapper-objekter. Wrapper-klasser har konstruktører for å sette verdien, og metoder for å avlese verdien.

```
Integer iRef = new Integer(10);   // (1) Eksplisitt innkapsling.
int j = iRef.intValue();          // (2) Eksplisitt utpakking.
```

Metoden `intValue()` i klassen `Integer` returnerer verdien av tallobjektet som et heltall av typen `int`.

Program 3.5 *Eksempel på konvertering mellom strenger og primitive verdier*

```
class PrimitivVerdiWrapper {
  public static void main(String[] args) {
    // En vanlig primitiv verdi.
    int verdiInn = 2005;

    // To måter å lage et objekt fra en primitiv verdi:
    Integer verdiObjekt;
    verdiObjekt = new Integer(verdiInn);
    verdiObjekt = verdiInn;             // Enkel variant

    // To måter å lage en primitiv verdi fra et objekt:
    int verdiUt;
    verdiUt = verdiObjekt.intValue();
    verdiUt = verdiObjekt;              // Enkel variant
    assert(verdiInn == verdiUt);        // Påstand: samme primtive verdi
    System.out.println("verdiInn: " + verdiInn + ", verdiUt: " + verdiUt);
  }
}
```

Utdata fra programmet:

```
verdiInn: 2005, verdiUt: 2005
```

Nyttige metoder i wrapper-klasser

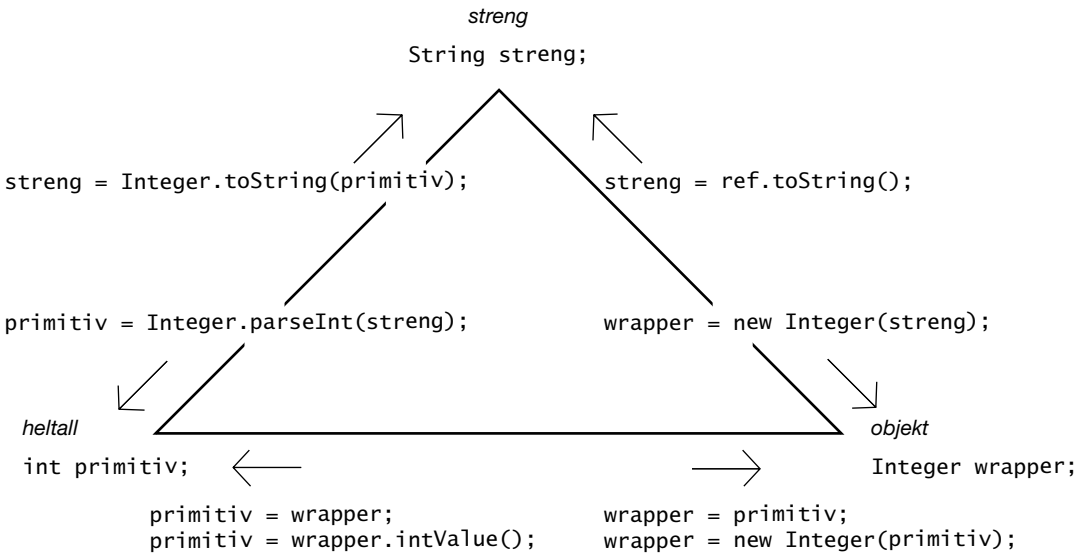
En del nyttige metoder i `Integer`-klassen er vist i Tabell 3.5. De øvrige numeriske wrapper-klassene tilbyr tilsvarende metoder. Vi noterer oss at det ikke er mulig å endre verdien som er lagret i et wrapper-objekt. Vi kan bare hente ut verdien.

Tabell 3.5 *Utvalgte metoder fra `Integer`-klassen*

`java.lang.Integer`	
`int intValue()`	Returnerer verdien av heltallsobjektet som en verdi av typen `int`.
`String toString()`	Konvertering fra wrapper-objekt til streng. Returnerer en streng-representasjon av `Integer`-objektets tilstand.
`static String toString(int i)`	Konvertering fra primitiv verdi til streng. Returnerer et `String`-objekt som inneholder strengrepresentasjon av den gitte heltallsverdien.
`static int parseInt(String s)`	Konvertering fra streng til primitiv verdi. Tolker en tekststreng som et heltall av typen `int`. Denne metoden håndterer kun strenger som inneholder siffer og minustegn. Den kaster et unntak av typen `NumberFormatException` (se kapittel 13) hvis strengen ikke representerer et heltall.

Figur 3.9 viser hvordan vi kan konvertere mellom en heltallsverdi, en streng og et `Integer`-objekt. Program 3.6 tester overgangene i Figur 3.9. Tilfelle A viser konvertering av en streng til et `Integer`-objekt som videre konverteres til en primitiv verdi, og denne verdien konverteres tilbake til en streng. Vi har benyttet oss av metoder fra Tabell 3.5. Under kjøring av programmet vil påstanden i (1) vise at vi får en streng som er identisk med strengen ved utgangspunktet. Analogt i tilfelle B vises det konvertering av en streng som representerer et flyttall, der strengen først konverteres til en primitiv verdi.

Figur 3.9 *Konvertering mellom heltall, strenger og wrapper-klassen `Integer`*



Program 3.6 *Eksempel på konverteringer mellom strenger, primitive verdier og wrapper-objekter*

```
class PrimitivVerdiRepresentasjoner {
  public static void main(String[] args) {
    String streng1, streng2;

    // Tilfelle A: streng --> wrapper --> primitiv --> streng
    streng1 = "2005";
    Integer iWrapper = new Integer(streng1);
    int iPrimitiv = iWrapper;
    streng2 = Integer.toString(iPrimitiv);
    assert(streng1.equals(streng2));  // (1)

    // Tilfelle B: streng --> primitiv --> wrapper --> streng
    streng1 = "12.5";
    double dPrimitiv = Double.parseDouble(streng1);
    Double dWrapper = dPrimitiv;
    streng2 = dWrapper.toString();
    assert(streng1.equals(streng2));  // (2)
  }
}
```

3.5 Mer om kontrollflyt

Dette avsnittet innfører to nye språkkonstruksjoner (`for(;;)`-løkke og `switch`-setning) som har med kontrollflyt å gjøre. Først beskrives imidlertid to typer operatorer som er nyttige i forbindelse med løkker: *utvidete tilordningsoperatorer* og *inkrement-/dekrementoperatorer*.

Gitt en tilordning på følgende form:

```
x = x + (y);                // (1)
```

dvs. at variabelen `x` inngår som en operand for en binær operator (i dette tilfellet `+`) på høyre siden av tilordningen (`=`), der `y` er et vilkårlig uttrykk, og resultatet tilordnes variabelen `x`. Da kan vi benytte oss av utvidete tilordningsoperatorer:

```
x += y;                     // (2)
```

Setning (1) er ekvivalent med setning (2), der `+=` er en utvidet tilordningsoperator. Program 3.7 viser er en del andre eksempler på utvidete operatorer.

Program 3.7 *Utvidete tilordningsoperatorer*

```
public class UtvideteTilordningsoperatorer {

  public static void main(String[] args) {
    int i = 5, j = 10;
    double x = 5.0, y = 10.5, z = 10.0;
    i += j;            // i = i + (j)
    x -= y + 30.5;     // x = x - (y + 30.5)
    j *= i + 20;       // j = j * (i + 20)
    y /= 5.0;          // y = y / (5.0)
    z %= 3.0;          // z = z % (3.0)
    System.out.println("i : " + i);
    System.out.println("x : " + x);
    System.out.println("j : " + j);
    System.out.println("y : " + y);
    System.out.println("z : " + z);
  }
}
```

Utdata fra programmet:

```
i : 15
j : 350
x : -36.0
y : 2.1
z : 1.0
```

Svært ofte skal verdien i en variabel økes med 1. Begge tilordninger nedenfor oppnår dette:

```
x = x + 1;                // (1)
x += 1;                   // (2)
```

I slike tilfeller kan vi også benytte oss av *inkrementoperatoren*:

```
x++;                    // (3) Verdien i x økes med 1.
```

Setningen i linje (3) oppnår det samme som setningene i (1) og (2). Vi kan bruke *dekrementoperatoren* (`--`) dersom verdien i en variabel skal minkes med 1.

```
x--;                    // Verdien i x minkes med 1.
```

Inkrement- og dekrementoperatorer kommer i to varianter, som *pre-* (`++x` og `--x`) og *post*operatorer (`x++` og `x--`), det vil si som foranstilte og etterstilte operatorer. Det har ingen betydning hvilken variant man bruker i eksemplene ovenfor, men som Program 3.8 viser, bør man være varsom dersom uttrykk som `x++` og `x--` skal inngå i en beregning.

Program 3.8 *Inkrement- og dekrementoperatorer*

```
public class InkrementOgDekrementoperatorer {

  public static void main(String[] args) {
    int i = 10, j;
    j = i++;                // (1) j får verdi 10, og i får verdi 11
    System.out.println("j : " + j + " og i: " + i);
    i = 10;
    j = ++i;                // (2) j får verdi 11, og i får verdi 11
    System.out.println("j : " + j + " og i: " + i);
  }
}
```

Utdata fra programmet:

```
j : 10 og i: 11
j : 11 og i: 11
```

I Program 3.8 ser vi at `i` ble økt med `1` i både linje (1) og linje (2), men `j` fikk tilordnet forskjellige verdier. Grunnen til forskjellen er at preinkrementoperatoren `++` i uttrykket (`++i`) adderer `1` til `i` og deretter bruker den nye verdien til `i` som verdien av uttrykket. Derimot bruker postinkrementoperatoren `++` i uttrykket (`i++`) den inneværende verdien til `i` som verdien av uttrykket og adderer deretter `1` til `i`. Dersom vi bare er interessert i bivirkningen av disse operatorene på operanden, er det likegyldig om vi bruker pre- eller postvarianten.

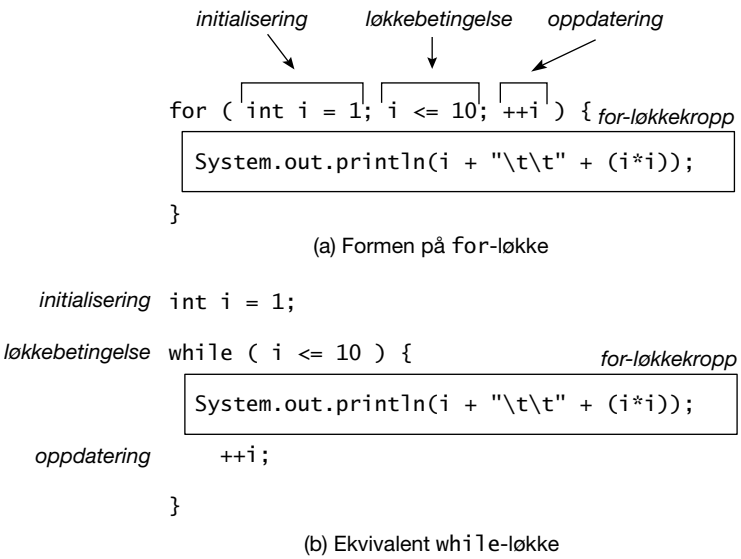
Tellerkontrollerte løkker

Java tilbyr `for(;;)`-setningen for å konstruere fleksible løkker. Denne setningen blir mest brukt for *tellerkontrollerte løkker* der antall ganger løkkekroppen skal utføres, er kjent på forhånd. Formen på `for(;;)`-setningen er vist i Figur 3.10a. I *initialiseringsdelen* tilordnes en *startverdi* (`1`) til en *løkkevariabel* (`i`). Initialiseringsdelen blir utført bare en gang. Verdien til løkkevariabelen sammenlignes med en *sluttverdi* (`10`) i *løkkebetingelsen* (`i <= 10`). Dersom betingelsen er sann, utføres *(for-)løkkekroppen*. Umiddelbart etter utføring av løkkekroppen inkrementeres løkkevariabelen i *oppdateringsdelen* (`++i`). Oppdatering av løkkevariabelen i oppdateringsdelen sørger for at løkkebetingelsen en gang blir usann. Løkkebetingelsen sjekkes etter hver oppdatering. Dersom betingelsen er usann, avsluttes løkken, og utføringen

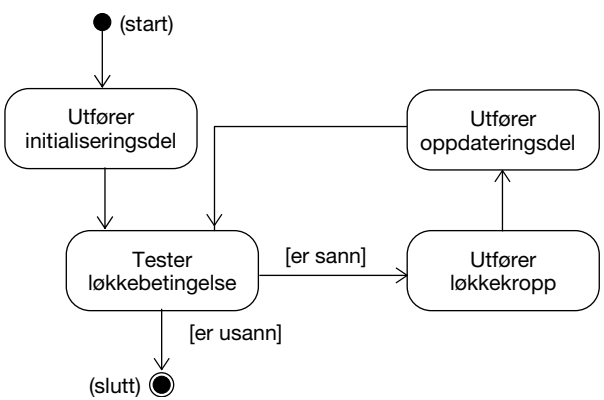
fortsetter etter `for`-setningen. Så lenge betingelsen er sann, utføres løkkekroppen en gang til. Denne syklusen gjentar seg helt til løkkebetingelsen blir usann. Denne utføringen er illustrert i Figur 3.11.

Løkken i Figur 3.10 blir utført ti ganger. Figur 3.10b viser saksgangen i utføring av en `for`-setning ved hjelp av en `while`-løkke. Legg merke til at initialiseringsdelen utføres bare en gang. Oppdateringsdelen utføres alltid etter `for`-løkkekroppen. Løkkekroppen blir ikke utført i det hele tatt, dersom løkkebetingelsen er usann ved inngangen til løkken. Løkken vil aldri stoppe (kalt *evig løkke*), dersom løkkebetingelsen aldri blir usann.

Figur 3.10 *Formen på `for(;;)`-setning*



Figur 3.11 *Utføring av en `for(;;)`-setning*



Vi har beskrevet en enkel form av `for(;;)`-setning for tellerkontrollerte løkker, der en løkkevariabel blir initialisert først og deretter oppdatert, slik at løkkebetingelsen til syvende og sist blir usann, med den følge at løkken avsluttes. Initialiseringsdelen i Figur 3.10a inneholder en

deklarasjon av variabel `i`. En variabel som er deklarert i en `for`-setning, kan bare brukes i `for(;;)`-setningen den er deklarert i, og er ikke tilgjengelig utenfor løkken. I kodelinjene nedenfor er ikke de lokale variablene `i` og `m` tilgjengelige utenfor `for(;;)`-setningen:

```
for (int i = 1; i <= 5; i++) {
  int m = 20;
  System.out.println(i + "\t\t" + (i*m));
}
System.out.println("i har verdi: " + i); // Kompileringsfeil!
System.out.println("m har verdi: " + m); // Kompileringsfeil!
```

Program 3.9 bruker en `for(;;)`-løkke for å stave et telefonnummer som ord. Telefonnummeret leses som en streng fra tastaturet. En `for(;;)`-setning brukes til å behandle alle tegn i strengen. Hvert tegn i strengen er et siffer i telefonnummeret. En `if`-setning for hvert siffer oversetter dets tegn til korresponderende ord. Strengen med ordene skrives ut til slutt. Løkkekroppen utføres for verdiene fra 0 til n-1 for løkkevariabelen `teller`, der n er antall tegn i strengen. Løkkebetingelsen (`teller < innlestStreng.length()`) sørger for å avslutte løkken når `teller` blir lik `innlestStreng.length()`, dvs. n.

Program 3.9 *Enkel bruk av `for(;;)`-setning*

```
import java.util.Scanner;
// Program oversetter tlf.nr. til ord: if-versjon
public class Telefon {
  public static void main(String args[]) {
    char tegn;
    String tlfStreng = "";
    Scanner tastatur = new Scanner(System.in);
    System.out.print("Skriv telefonnummer (som 55584152): ");
    String innlestStreng = tastatur.next();
    for (int teller = 0; teller < innlestStreng.length(); teller++) {
      tegn = innlestStreng.charAt(teller);
      if (tegn == '1') tlfStreng += "en ";
      else if (tegn == '2') tlfStreng += "to ";
      else if (tegn == '3') tlfStreng += "tre ";
      else if (tegn == '4') tlfStreng += "fire ";
      else if (tegn == '5') tlfStreng += "fem ";
      else if (tegn == '6') tlfStreng += "seks ";
      else if (tegn == '7') tlfStreng += "sju ";
      else if (tegn == '8') tlfStreng += "åtte ";
      else if (tegn == '9') tlfStreng += "ni ";
      else if (tegn == '0') tlfStreng += "null ";
      else System.out.println(tegn + " er ikke et siffer!");
    }
    System.out.println(tlfStreng);
  }
}
```

Kjøring av programmet:

```
Skriv telefonnummer (som 55584152): 55264326
```

```
fem fem to seks fire tre to seks
```

«Baklengs» løkker

Det er mulig å lage en løkke som teller baklengs. Linje (2) i Program 3.10 viser utforming av en løkke hvor verdien i løkkevariabelen minkes med 1 hver gang løkkekroppen blir utført.

Startverdien (`10`) er større enn sluttverdien (`0`), slik at løkkebetingelsen (`i > 0`) blir usann når løkkevariabelen `i` får verdi `0`.

Nøstede løkker

Vi kan la løkkekroppen være en `for(;;)`-løkke. Da får vi nøstede `for(;;)`-løkker. Linje (3) i Program 3.10 viser utforming av nøstede løkker. Den ytre løkken løper fra 1 til 5, og den indre løkken løper fra 1 til 10. Dette betyr at for hver gjennomgang av den ytre løkken utføres den indre løkken ti ganger. Det medfører at løkkekroppen til den indre løkken utføres til sammen 50 ganger (5 × 10). Dette betyr også at verdien til løkkevariabelen for den indre løkken endres hyppigere enn verdien til løkkevariabelen for den ytre løkken.

Å endre løkkeutføring

To setninger i Java har spesiell betydning når de blir brukt i en løkkesetning. Utføring av en `break`-setning i en løkke fører til at resten av løkkekroppen ikke blir utført, og utføringen fortsetter etter løkken, dvs. at utføringen av løkken avsluttes. Linje (4) i Program 3.10 viser et eksempel på bruk av en `break`-setning i en `for`-løkke. Når `i` får verdi `4`, utføres `break`-setningen. Øvrige handlinger i løkkekroppen blir ikke utført, og utføringen fortsetter etter løkken.

Utføring av en `continue`-setning i en løkkesetning fører til at resten av løkkekroppen ikke blir utført. Hva som skjer etterpå, er avhengig av løkkesetningen. I en `while`-løkke vil utføringen fortsette med å teste løkkebetingelsen, mens i en `for(;;)`-løkke vil utføringen fortsette med oppdateringsdelen. Linje (5) i Program 3.10 viser et eksempel på bruk av en `continue`-setning i en `for(;;)`-løkke. Når `i` blir lik `3`, utføres `continue`-setningen. Øvrige handlinger i løkkekroppen blir ikke utført, og løkken fortsetter normalt med utføring av oppdateringsdelen.

Setningene `break` og `continue` må brukes med omhu i en løkke, siden de bryter med standard utføring av en løkke og dermed kan gjøre programlogikken vanskelig å forstå.

Program 3.10 *Flere eksempler på bruk av `for(;;)`-setning*

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

    // (1) Enkel for-løkke
    // Skriv ut kvadrat av tall fra 1 til 5.
    System.out.println("Kvadrat av tall fra 1 til 5");
    for ( int i = 1; i <= 5; i++ ) {
      System.out.println(i + "\t\t" + (i*i));
```

```
    }

    // (2) "Baklengs" for-løkke
    // Skriv ut kvadrat av tall fra 5 til 1.
    System.out.println("Kvadrat av tall fra 5 til 1");
    for (int i = 5; i > 0; i--) {
      System.out.println(i + "\t\t" + (i * i));
    }

    // (3) Nøstede for-løkker
    // Skriv ut gangetabeller fra 1 til 5.
    for (int i = 1; i <= 5 ; i++) {
      System.out.println("Multiplikasjonstabell for " + i);
      for (int j = 1; j <= 10; j++) {
        System.out.println(i + "\tx\t" + j + "\ter\t" + (i*j));
      }
    }

    // (4) for-løkke med en break-setning.
    System.out.println("Kvadrat av tall fra 1 til 5.");
    for (int i = 1; i <= 5; i++) {
      if (i == 4) {
        System.out.println("Avslutter løkken når i er lik " + i);
        break;
      }
      System.out.println(i + "\t\t" + (i * i));
    }

    // (5) for-løkke med en continue-setning.
    System.out.println("Kvadrat av tall fra 1 til 5.");
    for (int i = 1; i <= 5; i++) {
      if (i == 3) {
        System.out.println("Hopper over resten av løkkekroppen når i lik " + i);
        continue;
      }
      System.out.println(i + "\t\t" + (i * i));
    }
  }
}
```

Utdata fra programmet:

```
Kvadrat av tall fra 1 til 5
1        1
2        4
3        9
4        16
5        25
Kvadrat av tall fra 5 til 1
5        25
4        16
```

```
3        9
2        4
1        1
Multiplikasjonstabell for 1
1    x    1    er    1
1    x    2    er    2
1    x    3    er    3
...
Kvadrat av tall fra 1 til 5.
1   1
2   4
3   9
Avslutter løkken når i er lik 4
Kvadrat av tall fra 1 til 5.
1   1
2   4
Hopper over resten av løkkekroppen når i lik 3
4   16
5   25
```

Nyttig å vite om løkker

Initialiseringen og oppdateringen kan utelates i en `for(;;)`-setning. Utføringen vil fremdeles foregå som beskrevet ovenfor, men nå er det ingen operasjoner å utføre i disse delene. Løkkebetingelsen kan også utelates, men det kan ha store konsekvenser. Mangel på løkkebetingelse betyr at løkkebetingelsen er *sann*, slik at løkken aldri vil slutte. En `break`-setning kan være nyttig for å unngå en evig løkke. Her er et eksempel på en evig løkke:

```
for (;;) { // alltid sann
  System.out.println("Stopp! Jeg vil av!");
}
```

En programmeringsfeil som ofte forekommer er at en løkke blir utført en gang for mye eller en gang for lite. I koden nedenfor vil ikke verdien i variabelen `sisteVerdi` bli summert dersom verdien i den er et partall. (Partall er tall som er delelig med 2.) Startverdien til løkkevariabelen `teller` er 2 og økes med 2. Dette betyr at verdien i variabelen `teller` alltid er et partall. Når variabelen `teller` blir lik partall i variabelen `sisteVerdi`, er (`teller < sisteVerdi`) usann, og løkken vil avslutte uten å summere variabelen `teller`.

```
int resultat = 0;
for (int teller = 2; teller < sisteVerdi; teller += 2) {
  resultat += teller;
}
```

Det hender ofte at løkkevariabelen ikke er initialisert riktig. I koden nedenfor er variabelen `resultat` initialisert til `0`. Etter utføring av løkkekroppen forblir den fortsatt `0`, slik at løkkebetingelsen alltid vil være sann i `do-while`-løkken. Sannsynligvis burde variabelen `resultat` vært initialisert til `1`.

```
resultat = 0;
do {
  resultat *= 2;          // alltid 0
} while (resultat < 100); // evig løkke
```

Det er viktig å sørge for at løkkebetingelsen er fornuftig. I koden nedenfor får variabelen `teller` aldri verdi `100` slik at løkken kan avslutte.

```
resultat = 0;
for (int teller = 1; teller != 100; teller += 2) {
  resultat += teller;
} // evig løkke
```

Løkkekroppen kan utføres mange ganger i en løkke. Tiden det tar å utføre et program, avhenger av hvor mange ganger handlinger i løkker blir utført. Dersom vi kan effektivisere utføring av løkker, kan vi forbedre kjøretiden til programmet. Alle beregninger som ikke endrer verdi under utføring av løkken, kalles *løkkeinvarianter*. Ved å flytte dem ut av løkken kan vi korte ned kjøretiden til programmet.

```
double pi = 3.14;
for (double r = 0.0; r < 100.0; r++) {
  System.out.println("radius: " + r +
                     ", omkrets: " + (2.0 * pi * r));
  // (2.0 * pi) endrer seg ikke i løkken, og kan flyttes ut.
}
```

I koden ovenfor blir uttrykket (`2.0 * pi`) beregnet under hver gjennomgang av løkken, selv om dets verdi ikke endrer seg under utføring av løkken. Uttrykket (`2.0 * pi`) er en løkkeinvariant. Vi kan flytte beregning av dette ut av løkken, der vi beregner det én gang og lagrer denne verdien i en ny variabel. Denne variabelen kan vi benytte i løkken i stedet for løkkeinvarianten.

```
double pi = 3.14;
double faktor = 2.0 * pi;
for (double r = 0.0; r < 100.0; r++) {
  System.out.println("radius: " + r + ", omkrets: " + (faktor * r));
}
```

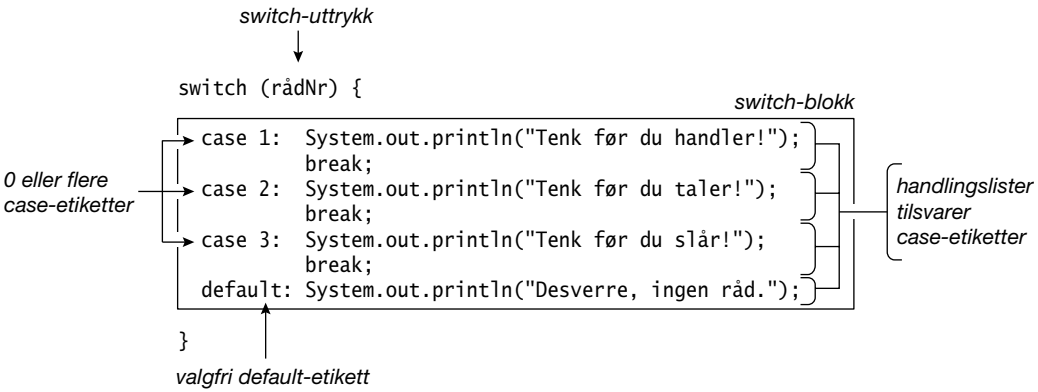
Tenk igjennom utføringen ved første og siste gjennomgang av løkken for å unngå en-for-mye- eller en-for-lite-feil og utilsiktede evige løkker. Kontroller også en del mellomliggende gjennomganger. Du sparer mye tid som du ellers må bruke på *avlusing* (eng. *debugging*) av programmet, dvs. endre, kompilere, kjøre og finne feil i programmet.

Bruk av flervalgsetningen `switch`

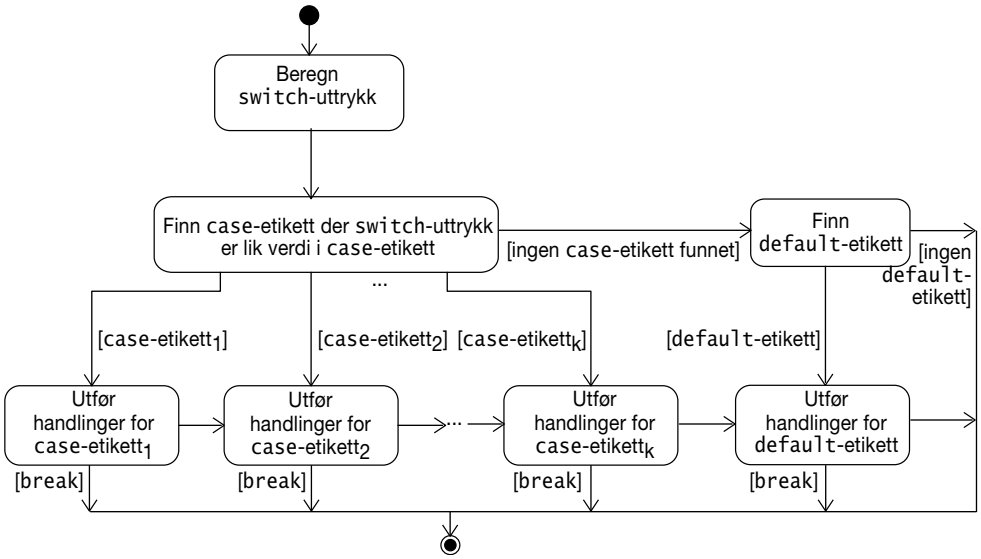
En `switch`-setning kan brukes til å velge én blant flere alternative handlinger, basert på verdien til et uttrykk. Figur 3.12 viser formen til en `switch`-setning. Verdien til `switch`-uttrykket blir sammenlignet med verdiene til `case`-*etikettene* (eng. *case label*), én om gangen. Dersom uttrykksverdien er lik verdien til en `case`-etikett, utføres den tilsvarende *handlingslisten*. Dersom verdien til variabelen `rådNr` er `2`, vil handlingslisten som tilsvarer `case`-etikett med verdi

2 bli utført. Denne handlingslisten består av et kall til `println()`-metoden og en `break`-setning. Vi får først skrevet ut `"Tenk før du taler!"`, etterpå utføres `break`-setningen som fører til at resten av `switch`-setningen blir hoppet over, og så fortsetter programutføringen etter `switch`-setningen.

Figur 3.12 *Flervalgsetningen `switch`*



Figur 3.13 *Utføring av en `switch`-setning*



Utføringen av en `switch`-setning er illustrert i Figur 3.13. Dersom ingen `case`-etikett er lik uttrykksverdien, utføres handlingslisten som tilhører `default`-etiketten, dersom denne er spesifisert. Dersom `default`-etiketten mangler, blir ingen handling utført, og utføringen fortsetter etter `switch`-setningen. Dersom verdien til variabelen `rådNr` er `4` (som ikke er lik noen av `case`-etikettene), vil handlingslisten som tilsvarer `default`-etikett, bli utført. Denne handlingslisten består av et kall til `println()`-metoden. Vi får skrevet ut `"Dessverre, ingen råd."`, og programutføringen fortsetter etter `switch`-setningen. Program 3.11 viser et eksempel på bruk av

en switch-setning. Programmet leser et telefonnummer fra tastaturet og «staver» det ut i ord i terminalvinduet.

Program 3.11 *Bruk av en switch-setning*

```
import java.util.Scanner;
// Programmet oversetter tlf.nr. til ord: switch-versjon
public class TelefonII {
  public static void main(String args[]) {
    char tegn;
    String tlfStreng = "";
    Scanner tastatur = new Scanner(System.in);
    System.out.print("Skriv telefonnummer (som 55584152): ");
    String innlestStreng = tastatur.next();
    for (int teller = 0; teller < innlestStreng.length(); teller++) {
      tegn = innlestStreng.charAt(teller);
      switch (tegn) {
        case '1':
          tlfStreng += "en ";   break;
        case '2':
          tlfStreng += "to ";   break;
        case '3':
          tlfStreng += "tre ";  break;
        case '4':
          tlfStreng += "fire "; break;
        case '5':
          tlfStreng += "fem ";  break;
        case '6':
          tlfStreng += "seks "; break;
        case '7':
          tlfStreng += "syv ";  break;
        case '8':
          tlfStreng += "åtte "; break;
        case '9':
          tlfStreng += "ni ";   break;
        case '0':
          tlfStreng += "null "; break;
        default:
          System.out.println(tegn + " er ikke et siffer!");
      } // end switch
    } // end for
    System.out.println(tlfStreng);
  }
}
```

Kjøring av programmet:

```
Skriv telefonnummer (som 55584152): 55264326
fem fem to seks fire tre to seks
```

Det er ikke nødvendig å omslutte en handlingsliste med blokknotasjon, {}. Uttrykksverdien og verdien til en case-etikett testes for *likhet*, mens betingelsen i en if-else-setning tillater et

vilkårlig sannhetsuttrykk. Variabler deklarert i en switch-blokk kan ikke brukes utenfor denne blokken. Har man behov for en slik variabel, må den deklareres foran switch-setningen. Variabelen tlfStreng er en slik variabel i Program 3.11.

Verdiene til case-etikettene må være litteraler, enten heltall, tegn eller av oppramstype (se avsnitt 4.6 på side 138), og ikke boolske eller flyttall. Samme litteral kan ikke forekomme flere ganger i samme switch-setning.

Dersom en break-setning mangler på slutten av en handlingsliste, vil utføringen fortsette i påfølgende handlingslister. Flere case-etiketter kan også dele en handlingsliste. Program 3.12 bruker felles handlingsliste for flere case-etiketter.

Program 3.12 *Felles handlinger for case-etiketter i en switch-setning*

```
import java.util.Scanner;
public class Aarstid {
  public static void main(String args[]) {
    Scanner tastatur = new Scanner(System.in);
    System.out.print("Tast månedsnummer: ");
    int tall = tastatur.nextInt();
    switch (tall) {
      case 12: case 1: case 2: // felles handlinger for flere verdier
        System.out.println("Det er vinter!");
        break;
      case 3: case 4: case 5:
        System.out.println("Det er vår!");
        break;
      case 6: case 7: case 8:
        System.out.println("Det er sommer!");
        break;
      case 9: case 10: case 11:
        System.out.println("Det er høst!");
        break;
      default:
        System.out.println(tall + " er ikke en lovlig måned!");
    } // end switch
  }
}
```

Kjøring av programmet:

```
Tast månedsnummer: 4
Det er vår!
```

3.6 Tabeller

Svært ofte er det behov for å organisere dataverdier slik at de kan behandles som en gruppe eller en samling. Tabeller er en måte å lage slike *samlinger* med dataverdier på. For eksempel kan vi lage en tabell for å holde rede på antall tekstmeldinger sendt hver dag i en uke fra en

mobiltelefon. En slik samling vil da ha sju heltall, ett for hver dag, som angir antall tekstmeldinger sendt denne dagen. Hvordan finner man for eksempel antall tekstmeldinger sendt på en onsdag i en slik samling for en gitt uke? For å svare på dette må vi først se nærmere på egenskapene til tabeller.

En tabell er en datastruktur som har en fast *lengde*, som angir hvor mange dataverdier det er plass til i tabellen. Vi sier at en tabell er delt opp i et fast antall *elementer*. Verdiene lagres i elementene. For å unngå å blande hummer og kanari i en tabell må verdiene i elementene tilhøre en bestemt type (kalt *elementtype*). Posisjonene til elementene i en tabell er nummerert fra 0 og oppover. Posisjonen til et element er gitt ved elementets *indeks*. Laveste indeks, 0, angir første element i en tabell, og dersom det er avsatt plass til n elementer i tabellen, har det siste elementet indeks n-1. Dersom vi bestemmer at indeks 0 i tabellen med antall tekstmeldinger ovenfor angir den første dagen i uken (mandag), vil indeksen 2 angi antall tekstmeldinger sendt på en onsdag. En slik tabell er illustrert i Figur 3.14 på side 86.

I Java er tabeller objekter, og Java tilbyr språkkonstruksjoner for *deklarasjon*, *opprettelse*, *initialisering* og *bruk* av tabeller. Vi har nevnt en tabell med heltallsverdier som angir antall tekstmeldinger. Dette er et eksempel på en tabell med verdier av en primitiv datatype, nemlig heltall. Vi kan lage tabeller av heltall, flyttall, tegn og sannhetsverdier. Siden elementene i en tabell lagrer verdier, og siden referanseverdier også er verdier, kan elementene lagre referanseverdier som refererer til objekter. Vi kan altså lage *tabeller av objekter*. Elementene er da referanser til objekter som tilhører en bestemt referansetype (for eksempel en bestemt klasse). Siden tabeller kan inneholde referanser til objekter, og tabeller er objekter, kan vi konstruere *tabeller av tabeller* (se delavsnittet «Tabeller av flere dimensjoner» på side 94).

Oppretting og bruk av tabeller

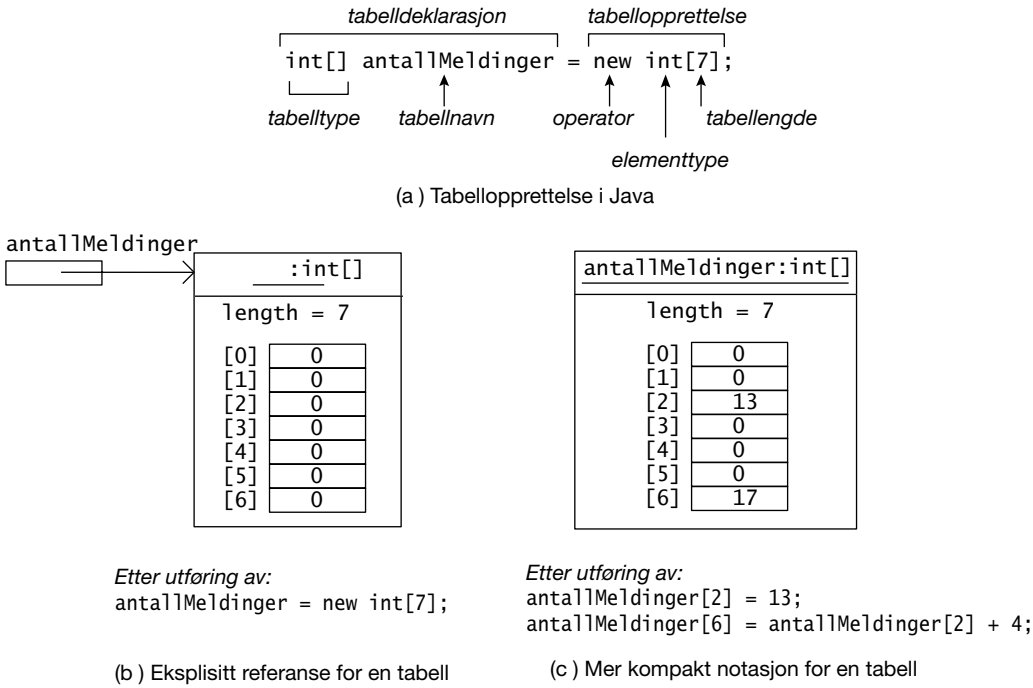
På lik linje med en deklarasjon av en referansevariabel for å referere til et objekt:

```
String tittel;
```

kan vi deklarere en referansevariabel for å referere til en tabell, siden tabeller er objekter:

```
int[] antallMeldinger;
```

Figur 3.14 *Tabeller: Deklarasjon, opprettelse, og initialisering med standardverdier*



Referansen `antallMeldinger` kan vi betrakte som tabellens navn og bruke til å håndtere tabellen. Uten klammer `[]` vil deklarasjonen ovenfor angi at `antallMeldinger` er en variabel av type `int` og ikke en tabell. Legg merke til at deklarasjonen

```
int[] antallMeldinger;
```

ikke oppretter noen tabell, kun en referanse som kan referere til en tabell av heltall. Deklarasjonen sier ikke noe om hvor stor tabellen skal være, markert med tomme klammer, `[]`. I tabelldeklarasjonen ovenfor er tabelltypen `int[]`, og elementtypen er `int`. Dette betyr at referansen `antallMeldinger` kan referere til en hvilken som helst tabell så lenge elementene i den er heltall.

Vi har sett hvordan objekter opprettes fra klasser med `new`-operatoren sammen med et konstruktørkall. For å opprette tabeller brukes også `new`-operatoren, men her sammen med den ønskede tabellengden, slik som dette:

```
antallMeldinger = new int[7];
```

Dette oppretter en tabell som lagrer sju `int`-verdier. Lengden til en tabell kan ikke endres når tabellen først er opprettet. Legg merke til at opprettelsen ovenfor ikke sier noe om hvilke sju heltallsverdier som lagres i tabellen. Derfor vil alle elementene i tabellen få tildelt verdien 0. Regelen er nemlig at når en tabell opprettes slik som vist her, vil elementene alltid automatisk initialiseres til *standardverdien* til elementtypen (jf. Figur 3.14b). Tabell 3.6 på side 87 viser standardverdien til primitive typer og referansetyper. Oppretter vi en tabell av type `boolean`,

vil alle elementene ha verdi `false`. Oppretter vi en tabell der elementene skal referere til objekter, blir elementene initialisert med `null`.

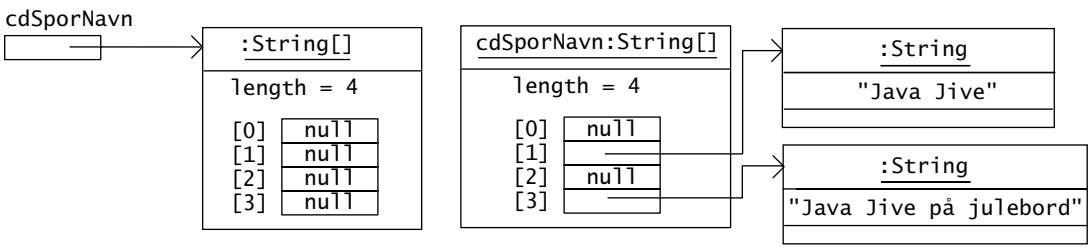
Deklarasjonen nedenfor viser opprettelse av en tabell `cdSporNavn` som kan inneholde fire `String`-objekter (som er navn på fire spor), og der vi har kombinert deklarasjon av tabell-referanse og tabellopprettelse (jf. Figur 3.15a):

```
String[] cdSporNavn = new String[4];
```

Tabell 3.6 *Standardverdier*

Type	Standardverdi
`boolean`	`false`
`char`	`'\u0000'`
Heltall (`int`, `long`)	`0`
Flyttall (`double`)	`+0.0d`
Alle referansetyper	`null`

Figur 3.15 *Tabell av objekter*



Etter utføring av:
`String[] cdSporNavn = new String[4];`

(a) Opprettelse av en tabell av objekter

Etter utføring av:
```
cdSporNavn[1] = "Java Jive";
cdSporNavn[3] = cdSporNavn[1] + " på julebord";
```

(b) Initialisering av tabell med objekter

I Figur 3.15a ovenfor, har ingen strenger blitt opprettet for spornavn, og alle elementene i tabell `cdSporNavn` har verdi `null`.

Hver tabell har et felt med navn `length` som angir tabellengden, dvs. antall elementer i tabellen, og som kan leses med punktnotasjon. Uttrykket `antallMeldinger.length` har verdi 7 (jf. Figur 3.14) mens uttrykket `cdSporNavn.length` (jf. Figur 3.15) har verdi 4.

Hvordan refererer vi til elementene i en tabell? Vi må angi tabellnavn og indeks til elementet i tabellen. Syntaksen for å referere til antall tekstmeldinger sendt på onsdag i tabell `antallMeldinger` er gitt ved `antallMeldinger[2]`. Tilsvarende vil `antallMeldinger[6]` angi antall tekstmeldinger sendt på søndag. Elementene har standardverdier (0 i dette tilfellet) når tabel-

len blir opprettet. Vi kan endre elementverdier, for eksempel ved tilordning (jf. Figur 3.14c):

```
antallMeldinger[2] = 13;              // Indeksverdien 2 angir onsdag.
```

Dersom vi vet at antall tekstmeldinger på søndag var fire mer enn antallet på onsdag, kan vi oppdatere antall tekstmeldinger på søndag slik:

```
antallMeldinger[6] = antallMeldinger[2] + 4; // Indeksverdien 6 angir søndag.
```

Legg merke til at notasjon `antallMeldinger[6]` oppfører seg nøyaktig likt en vanlig variabel, og den kan behandles akkurat på samme måte som en enkel variabel. Siden notasjon `antallMeldinger[6]` krever tabellnavnet og en indeks, kaller vi den en *indeksert variabel*.

Vi kan tenke oss at for å håndtere sju forskjellige heltallsverdier som representerer antall tekstmeldinger for hver dag i uken, kunne vi ha deklarert sju forskjellige vanlige variabler. Det kan virke overkommelig i dette tilfellet, men hva hvis vi skulle behandle ti tusen forskjellige heltallsverdier? Det samme gjelder også for tabeller av objekter. Notasjon `cdSporNavn[3]` er en referanse til `String`-objektet angitt av referanseverdien i elementet med indeks 3 i tabell `cdSporNavn`. Referansen `cdSporNavn[3]` kan vi bruke som en hvilken som helst referansevariabel. Her er et par eksempler (jf. Figur 3.15b):

```
cdSporNavn[1] = "Java Jive";                    // "Java Jive"
cdSporNavn[3] = cdSporNavn[1] + " på julebord"; // "Java Jive på julebord"
```

Vi har hele tiden brukt et heltall som indeks, faktisk kan indeks være et vilkårlig uttrykk som evaluerer til en heltallsverdi. Selvfølgelig må indeksverdien være større eller lik 0 og mindre enn lengden på tabellen (0 ≤ indeksverdien < tabellengde). Under utføringen blir indeksverdien kontrollert automatisk, og dersom den er ugyldig, vil utføringen vil bli avbrutt, markert med et unntak av typen `ArrayIndexOutOfBoundsException`.

Siden tabeller er objekter, kan vi lage aliaser til tabeller:

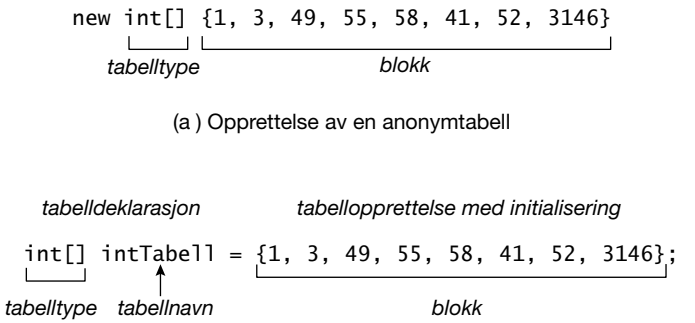
```
int[] meldingsTeller = antallMeldinger;
String[] sporTitler = cdSporNavn;
```

Referansene `antallMeldinger` og `meldingsTeller` er aliaser til samme tabell. Det samme gjelder også referansene `sporTitler` og `cdSporNavn`. Alle aliaser til en tabell kan brukes til å håndtere tabellen. Notasjon `antallMeldinger[2]` og `meldingsTeller[2]` refererer begge til antall meldinger sendt på onsdag i tabellen.

Initialisering av tabeller

Vi har sett at når vi oppretter en tabell med `new`-operatoren sammen med angivelse av tabellengden, får alle elementene standardverdien til elementtypen. Dersom vi ønsker å lagre andre verdier i tabellen ved opprettelsen, må vi eksplisitt lagre dem i tabellen. Dette kaller vi *(eksplisitt) tabellinitialisering*. Figur 3.16 viser to språkkonstruksjoner som oppretter og samtidig initialiserer en tabell. Disse formene er praktiske når man ønsker å opprette små tabeller med spesifikke verdier.

Figur 3.16 *Tabellopprettelse og initialisering*



(a) Opprettelse av en anonymtabell

(b) Tabellopprettelse og initialisering med blokknotasjon sammen med tabelldeklarasjon

Formen i Figur 3.16a er et uttrykk som returnerer en referanseverdi til en *anonymtabell* (eng. *anonymous array*) av angitt tabelltype og som er initialisert til verdiene angitt i blokken. Lengden på tabellen blir implisitt bestemt av antall verdier i blokken. Verdiene i blokken tilhører elementypen (`int`), og disse blir satt inn i tabellen med den første verdien i indeks 0, den andre verdien i indeks 1 osv. Referanseverdien til tabellen kan tilordnes en tabellreferanse av riktig type:

```
int[] tallrekke3;                // Kun deklarasjon, ingen opprettelse av tabell
// Tabellopprettelse, eksplisitt initialisering og tilordning:
tallrekke3 = new int[] {1, 3, 49, 28, 2, 1948}; // (1) Angir 6 verdier

// Tabelldeklarasjon, opprettelse, eksplisitt initialisering og tilordning:
int[] tallrekke4 = new int[] {55, 58, 41, 52};  // (2) Angir 4 verdier
```

Merk at tabelltypen (`int[]`) er angitt to ganger i linje (2), en på hver side av tilordningen. Dette er overflødig. I dette tilfellet kan formen i Figur 3.16a forenkles til formen i Figur 3.16b:

```
int[] tallrekke4 = {55, 58, 41, 52};           // Enkel variant
```

Notasjonen i Figur 3.16b kan kun brukes kombinert med tabelldeklarasjon:

```
tallrekke4 = {55, 58, 41, 52};                 // Kompileringsfeil!
```

Program 3.13 demonstrerer eksempler på tabelldeklarasjoner, opprettelse, initialisering og tilordning som vi har møtt i dette avsnittet.

Nedenfor viser vi en del eksempler på initialisering av forskjellige typer tabeller:

```
// (1) En heltallstabell med 8 elementer
int[] intTabell = {1, 3, 49, 55, 58, 41, 52, 3146};
// (2) En boolsktabell med 4 elementer
boolean[] boolsktabell = new boolean[] {true, false, false, true};
// (3) En tegntabell med 3 tegn
char[] tegnTabell = {'h', 'e', 'i'};
// (4) En flyttallstabell med 4 flyttall
double flyttallstabell = new double[] {25.0, 3.14, 1.5, 0.75};
// (5) En tabell med 4 strenger
```

```
   String[] kjæledyr = {"krokodiller", "elefanter", "krokofanter", "elediller"};
   // (6) En tabell for 3 strenger,
   //     men kun 2 elementer er initialisert med ikke-null verdier.
   kjæledyr = new String[] {"katt", null, "hund"};
```

Program 3.13 *Tabelldeklarasjon, opprettelse, initialisering og tilordning*

```
public class TabellInitialisering {
  public static void main(String[] args) {
    // Tabelldeklarasjon:
    int[] tallrekke1;  // Kun deklarasjon, ingen opprettelse av tabell

    // Tabellopprettelse, initialisering med standard verdi og tilordning:
    tallrekke1 = new int[7];            // Angir tabellengde ved opprettelsen
    assert(tallrekke1 != null);         // Tabellen har blitt opprettet
    assert(tallrekke1.length == 7);

    // Kombinert tabelldeklarasjon, opprettelse,
    // initialisering med standard verdi og tilordning:
    int[] tallrekke2 = new int[7];

    // Tabelldeklarasjon:
    int[] tallrekke3;  // Kun deklarasjon, ingen opprettelse av tabell
    // Tabellopprettelse, eksplisitt initialisering og tilordning:
    tallrekke3 = new int[] {1, 3, 49, 28, 2, 1948}; // (1) Angir 6 verdier
    assert(tallrekke3.length == 6);
    assert(tallrekke3[0] == 1);    // Første verdien er gitt ved indeks 0
    assert(tallrekke3[5] == 1948); // Siste verdien er gitt ved siste indeks
    //... og resten av tabellen er også initialisert med angitte verdier.

    // Tabelldeklarasjon, opprettelse, eksplisitt initialisering og tilordning:
    int[] tallrekke4a = new int[] {55, 58, 41, 52};  // (2) Angir 4 verdier
    int[] tallrekke4b = {55, 58, 41, 52};            // Enkel variant
//  tallrekke4a = {55, 58, 41, 52};                  // Kompileringsfeil!

    System.out.println("Tabelldeklarasjon, opprettelse, initialisering" +
                       " og tilordning gikk bra.");
  }
}
```

Kjøring av programmet:

```
Tabelldeklarasjon, opprettelse, initialisering og tilordning gikk bra.
```

Tegntabeller og strenger

Program 3.14 demonstrerer at en tabell av tegn ikke er det samme som en streng. I en tabell av tegn (`char[]`) inneholder hvert element et tegn, mens en streng er et `String`-objekt som inneholder en sekvens av tegn som utgjør strengen. Programmet viser i linje (1) hvordan man får tak i lengden, og i linjene (2) og (3) viser det hvordan man gjør oppslag. Vi kan alltid bygge en streng fra en tegntabell, slik det er vist i linje (4). Programmet feiler i linje (5), siden påstanden ikke er sann.

Program 3.14 *Tegntabeller og strenger*

```
public class TegnTabellerOgStrenger {
  public static void main(String[] args) {
    char[] tegntabell = {'h', 'e', 'i'};
    String ord = "hei";
    assert(tegntabell.length == ord.length());  // (1) Sammenligner lengder
    assert(tegntabell[1] == 'e');   // (2) Oppslag i tabell
//  assert(ord[1] == 'e');          // Kompileringsfeil, streng er ikke tabell.
    assert(ord.charAt(1) == 'e');   // (3) Oppslag i streng
    assert(ord.equals(new String(tegntabell))); // (4) Bygge streng fra en tegntabell.
    assert(ord.equals(tegntabell)) : "Tegntabeller og strenger er ulike."; // (5)
  }
}
```

Kjøring av programmet:

```
Exception in thread "main" java.lang.AssertionError: Tegntabeller og strenger er ulike.
        at TegnTabellerOgStrenger.main(TegnTabellerOgStrenger.java:11)
```

Iterering over en tabell

En vanlig oppgave i programmering er å behandle alle elementer i en tabell i tur og orden for å utføre samme handling på hvert enkelt element i tabellen. Denne operasjonen kalles *iterering* (eng. *iteration*) over en tabell.

For å iterere over en tabell med n elementer ønsker vi å starte med første element ved indeks 0, fortsette med indeks 1, 2, 3, osv. og avslutte med siste element ved indeks n-1.

Tellerkontrollert løkke med `for(;;)`-setning egner seg for iterering over tabeller. Indeksen initialiseres til 0 i initialiseringsdelen av `for(;;)`-løkken. Etter behandlingen av et element økes indeksen med 1 i oppdateringsdelen av `for(;;)`-løkken. Hva skal løkkebetingelsen være? Så lenge indeksen er *mindre* enn n er det fremdeles flere elementer igjen som løkkekroppen skal behandle. Løkkekroppen velger elementet som skal behandles ved å slå opp i tabellen på inneværende indeks.

```
// Mønster for iterering over en tabell:
for (int indeks = 0; indeks < tabell.length; indeks++ ) {
  // ... behandle elementet tabell[indeks] her ...
}
```

Mønsteret ovenfor vil garantere at gjennomløp av tabellen vil omfatte alle verdier i tabellen, der indeksverdien vil ligge i gyldig intervall og aldri føre til unntaket `ArrayIndexOutOf-BoundsException`.

Program 3.15 tar for seg tre enkle problemstillinger for tabellen `antallMeldinger`. Problem (1) går ut på å finne antall dager som har antall tekstmeldinger større enn eller lik en gitt grenseverdi. Vi leser grenseverdien (`grense`) for antall tekstmeldinger fra tastaturet, og har en teller (`antallDager`) som vi bruker til å telle antall dager som oppfyller dette kriteriet. Under hver gjennomgang av løkkekroppen sammenligner vi om inneværende dag (angitt ved

løkkevariabelen `indeks`) har et antall tekstmeldinger (angitt ved `antallMeldinger[indeks]`) som er større enn eller lik denne grenseverdien. Dersom det er tilfellet, øker vi telleren.

Problem (2) i Program 3.15 går ut på å finne laveste antall tekstmeldinger sendt på en dag. Vi bruker en variabel (`lavesteAntall`) for å holde rede på hittil laveste antall tekstmeldinger sendt på en dag, mens vi itererer over tabellen. Under hver gjennomgang tester vi om antall tekstmeldinger for inneværende dag (`antallMeldinger[indeks]`) er lavere enn antall tekstmeldinger gitt ved variabelen `lavesteAntall`. Dersom dette er tilfellet, så har den inneværende dagen det laveste antallet vi har sett så langt. Variabelen `lavesteAntall` må derfor oppdateres med verdien for den inneværende dagen.

Legg merke til at vi initialiserer variabelen `lavesteAntall` med verdien til elementet `antallMeldinger[0]`. Vi konstaterer at `antallMeldinger[0]` har laveste antall meldinger hittil. Vi kan ikke bruke en verdi som er lavere enn alle verdier i tabellen. Alternativt kan vi bruke en verdi som er større enn alle verdier i tabellen, for eksempel, `Integer.MAX_VALUE`.

Problem (3) i Program 3.15 går ut på å finne høyeste antall tekstmeldinger sendt på en dag, samt navn på dagene så mange meldinger ble sendt. Først finner vi høyeste antall tekstmeldinger sendt på en dag, slik som i problem (2). For å finne dagene med høyeste antall meldinger itererer vi over tabellen en gang til.

Program 3.15 *Iterering over en tabell*

```
import java.util.Scanner;
public class Iterering1 {

  public static void main(String[] args) {
    String[] ukedagNavn = {"Mandag", "Tirsdag", "Onsdag", "Torsdag", "Fredag",
                           "Lørdag", "Søndag"};
    int[] antallMeldinger = new int[7];
    antallMeldinger[0] = 20;                        // mandag
    antallMeldinger[1] = 12;                        // tirsdag
    antallMeldinger[2] = 13;                        // onsdag
    antallMeldinger[3] = antallMeldinger[1];        // torsdag
    antallMeldinger[4] = 10;                        // fredag
    antallMeldinger[5] = antallMeldinger[0];        // lørdag
    antallMeldinger[6] = antallMeldinger[2] + 4;    // søndag

    Scanner tastatur = new Scanner(System.in);

    // Problem (1) Finn antall dager som har antall meldinger
    // større eller lik et visst antall tekstmeldinger.
    System.out.print("Angi grense på antall tekstmeldinger: " );
    int grense = tastatur.nextInt();
    int antallDager = 0;
    for (int indeks = 0; indeks < antallMeldinger.length; indeks++ ) {
      if (antallMeldinger[indeks] >= grense)
        antallDager++;
```

```
    }
    System.out.println("Antall dager med mer enn " + grense +
                       " tekstmeldinger: " + antallDager);

    // Problem (2) Finn laveste antall tekstmeldinger sendt på en dag.
    int lavesteAntall = antallMeldinger[0];
    for (int indeks = 0; indeks < antallMeldinger.length; indeks++ ) {
      if (lavesteAntall > antallMeldinger[indeks])
        lavesteAntall = antallMeldinger[indeks];
    }
    System.out.println("Laveste antall tekstmeldinger: " + lavesteAntall);

    // Problem (3) Finn høyeste antall tekstmeldinger sendt på en dag,
    // samt navnet på de dagene så mange meldinger var sendt.
    // Finn maks antall meldinger sendt på en dag.
    int maxAntall = 0;
    for (int indeks = 0; indeks < antallMeldinger.length; indeks++ ) {
      if (maxAntall < antallMeldinger[indeks])
        maxAntall = antallMeldinger[indeks];
    }
    System.out.println("Høyeste antall tekstmeldinger: " + maxAntall);
    // Skriv ut alle dager som sendte maks antall meldinger.
    System.out.println("Dager med høyeste antall tekstmeldinger:");
    for (int indeks = 0; indeks < antallMeldinger.length; indeks++ ) {
      if (maxAntall == antallMeldinger[indeks])
        System.out.println(ukedagNavn[indeks]);
    }
  }
}
```

Kjøring av programmet:

```
Angi grense på antall tekstmeldinger: 13
Antall dager med mer enn 13 tekstmeldinger: 4
Laveste antall tekstmeldinger: 10
Høyeste antall tekstmeldinger: 20
Dager med høyeste antall tekstmeldinger:
Mandag
Lørdag
```

Program 3.16 viser iterering over en tabell av `String`-objekter. Programmet skriver ut alle spor som inneholder ordet `"Java"`. Under hver gjennomgang av løkkekroppen er inneværende objekt referert ved referanseverdien i `cdSporNavn[sporNr]`, der `sporNr` er den inneværende indeksen i tabellen `cdSporNavn`. For hvert spornavn kaller vi metoden `indexOf()` fra `String`-klassen, med strenglitteral `"Java"` som argument. Hvis returverdien ikke er -1, betyr det at ordet `"Java"` forekommer i inneværende spornavn.

Program 3.16 *Iterering over en tabell av strenger*

```
public class Itererring2 {

  public static void main(String[] args) {
    String[] cdSporNavn = new String[4];
    cdSporNavn[0] = "Fritjof i fjøset";
    cdSporNavn[1] = "Java Jive";
    cdSporNavn[2] = "Hæla i taket";
    cdSporNavn[3] = cdSporNavn[1] + " på julebord";
    for (int sporNr = 0; sporNr < cdSporNavn.length; sporNr++ ) {
      if(cdSporNavn[sporNr].indexOf("Java") != -1) {
        System.out.println(cdSporNavn[sporNr]);
      }
    }
  }
}
```

Utdata fra programmet:

```
Java Jive
Java Jive på julebord
```

Tabeller av flere dimensjoner

Tabeller som vi har sett hittil, kalles *enkle* eller *éndimensjonale tabeller*, og *én indeks* var nok til å identifisere elementer i tabellen. Det er mulig å lage *flerdimensjonale tabeller*, og det trengs flere indekser for å identifisere elementer i slike tabeller. Vi skal se nærmere på hvordan vi deklarerer, oppretter, initialiserer og benytter flerdimensjonale tabeller.

Opprettelse og initialisering av flerdimensjonale tabeller

I Program 3.15 ble det definert en enkel tabell, `antallMeldinger`, for å holde rede på antall tekstmeldinger som ble sendt hver dag i uken fra en mobiltelefon. For å håndtere slike opplysninger for flere mobiltelefoner kan vi bruke flere enkle tabeller, men en slik fremgangsmåte kan fort bli tungvint, for eksempel hvis programmet skal endres til å håndtere et stort antall mobiltelefoner. Opplysninger om tre mobiltelefoner for en hel uke er vist oppstilt i tabellform i Figur 3.17a. Telefonene blir identifisert med en mobilindeks, og ukedag med en dagindeks. Elementet med mobilindeks 2 og dagindeks 0 har verdi 18. Disse to indeksene er nødvendige for å finne antall tekstmeldinger for en gitt telefon og en gitt dag. Dette er et eksempel på en todimensjonal tabell og krever to indeksoppslag for å identifisere elementene i den flerdimensjonale tabellen.

Flerdimensjonale tabeller kan opprettes i Java ved å bygge tabeller av tabeller. Følgende setning deklarerer en referanse (`ukeTabell`) og oppretter en todimensjonal tabell som har tre rader og sju kolonner (jf. Figur 3.17b):

```
int[][] ukeTabell = new int[3][7]; // En todimensjonal tabell
```

Indeksangivelse tolkes fra venstre mot høyre, slik at den første indeksen angir rad og den andre angir kolonne i denne raden. Antall indekser på begge sider av tilordningsoperatoren i setningen ovenfor må være det samme. Alle de 21 elementene får standardverdi 0 ved opprettelsen.

Figur 3.17 *Oppstilling i tabellform*

Antall mobiltelefoner er 3.
Antall ukedager er 7.

dagindeks angir ukedag

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
[0]	12	10	22	33	19	27	16
[1]	45	55	44	34	39	15	11
[2]	***18***	26	36	40	24	11	20

mobilindeks angir mobiltelefon → [2]

Element med mobilindeks 2 og dagindeks 0 har verdi 18.

(a) Oppstilling i tabellform

tabelldeklarasjon *tabelloppretteIse*

```
int[][] ukeTabell = new int[3][7];
```

Antall indekser må være i samsvar.

(b) Opprettelse av en tabell av tabeller

Vi ønsker å lagre verdier fra Figur 3.17a i tabellen `ukeTabell`. Vi kan deklarere og opprette en todimensjonal tabell, og tilordne verdiene til elementene enkeltvis:

```
// Deklarasjon og opprettelse
int[] ukeTabell = new int[3][7];
// Initialisering av 1. mobiltelefon
ukeTabell[0][0] = 12; ukeTabell[0][1] = 10; ukeTabell[0][2] = 22;
ukeTabell[0][3] = 33; ukeTabell[0][4] = 19; ukeTabell[0][5] = 27;
ukeTabell[0][6] = 16;
// Initialisering av 2. mobiltelefon
ukeTabell[1][0] = 45; ukeTabell[1][1] = 55; ukeTabell[1][2] = 44;
ukeTabell[1][3] = 34; ukeTabell[1][4] = 39; ukeTabell[1][5] = 15;
ukeTabell[1][6] = 11;
// Initialisering av 3. mobiltelefon
ukeTabell[2][0] = 18; ukeTabell[2][1] = 26; ukeTabell[2][2] = 36;
ukeTabell[2][3] = 40; ukeTabell[2][4] = 24; ukeTabell[2][5] = 11;
ukeTabell[2][6] = 20;
```

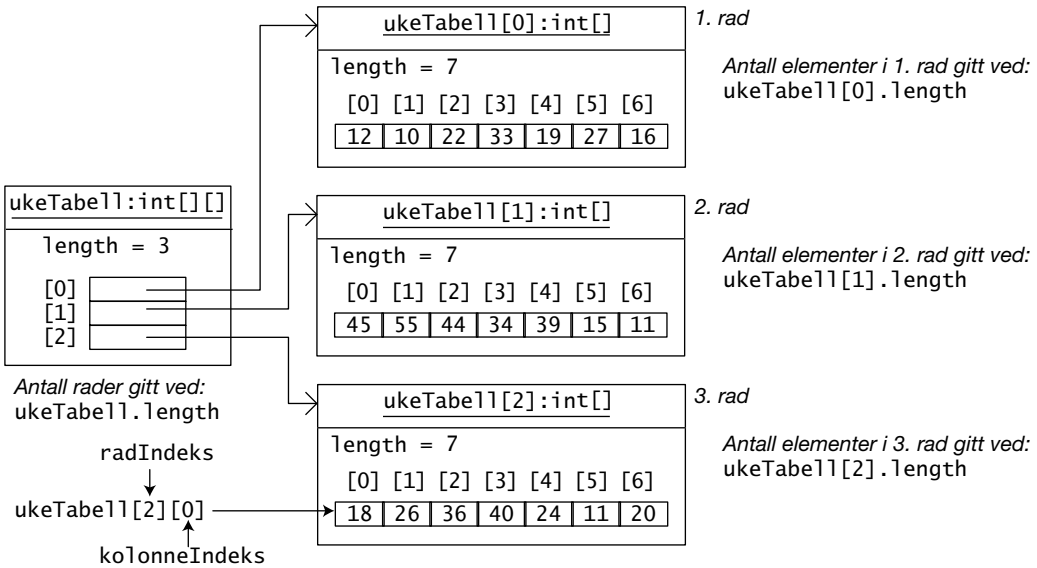
Det er viktig å legge merke til at `ukeTabell[2][0]` og `ukeTabell[0][2]` er to forskjellige elementer i tabellen. Element `ukeTabell[2][0]` angir 3. mobiltelefon på mandag, mens element `ukeTabell[0][2]` angir første mobiltelefon på onsdag. Det er viktig å ha klart for seg hva indeksene representerer, og være konsekvent i bruken av dem.

Vi kan også deklarere, opprette og initialisere tabellen `ukeTabell` med følgende syntaks:

```
int[][] ukeTabell = { // Deklarasjon, opprettelse og initialisering.
  {12, 10, 22, 33, 19, 27, 16},  // 1. mobiltelefon
  {45, 55, 44, 34, 39, 15, 11},  // 2. mobiltelefon
  {18, 26, 36, 40, 24, 11, 20}   // 3. mobiltelefon
};
```

Vi ser at selve deklarasjonen er den samme (`int[][]`), og vi har benyttet oss av blokknotasjon for initialisering av tabellen. Den ytre blokken omslutter indre blokker som angir verdier for hver mobiltelefon. Antall indre blokker angir antall mobiltelefoner, og antall verdier i hver indre blokk tilsvarer antall dager i uken.

Figur 3.18 *Eksplisitt struktur for en tabell av tabeller*



Tabell av tabeller, `ukeTabell`, har type `int[][]`, dvs. *en tabell av tabeller av* `int`.
Hver rad, `ukeTabell[radIndeks]`, har type `int[]`, dvs. *en tabell av* `int`.
Hvert element i hver rad, `ukeTabell[radIndeks][kolonneIndeks]`, har type `int`.

Som konsept er tabellen vi har opprettet vist i Figur 3.17a, men datastrukturen som egentlig ble opprettet i Java, er vist i Figur 3.18. Referansen `ukeTabell` er egentlig en tabell som har lengde 3, der hvert element tilsvarer en rad i tabelloppstillingen vist i Figur 3.17a. Hver rad er i sin tur en enkel tabell som har lengde 7, der hvert element i en rad er et heltall som angir antall tekstmeldinger. Alle flerdimensjonale tabeller er bygd opp av flere tabeller, slik det er vist i Figur 3.18. Slike datastrukturer kalles derfor *tabeller av tabeller*. Notasjon `ukeTabell[2][0]` angir et element i strukturen, der venstre indeks angir mobiltelefon og høyre indeks angir ukedag for denne mobiltelefonen. Notasjon `ukeTabell[2]` refererer til den enkle tabellen som inneholder data for den tredje mobiltelefonen. Uttrykket `ukeTabell[2].length` angir antall elementer i denne ukedagtabellen, nemlig sju.

Deklarasjonen

```
int[][] ukeTabell; // ren deklarasjon
```

deklarerer referansen `ukeTabell` til å være en tabell av tabeller av `int` (dvs. har type `int[][]`). Hver rad, `ukeTabell[radIndeks]`, er en tabell av `int` (dvs. har type `int[]`), og hvert element, `ukeTabell[radIndeks][kolonneIndeks]`, er et heltall (dvs. har type `int`).

Program 3.17 *Flerdimensjonale tabeller*

```
public class FlerdimensjonaleTabeller {

  public static void main(String[] args) {

    int[][] ukeTabell = { // Deklarasjon, opprettelse og initialisering.
```

```
        {12, 10, 22, 33, 19, 27, 16},  // 1. mobiltelefon
        {45, 55, 44, 34, 39, 15, 11},  // 2. mobiltelefon
        {18, 26, 36, 40, 24, 11, 20}   // 3. mobiltelefon
      };

    // Problem (1) Skriv ut opplysninger i tabellform
    for (int mobilIndeks = 0; mobilIndeks < ukeTabell.length; mobilIndeks++) {
      System.out.print("Mobil indeks " + mobilIndeks + ": ");
      for (int dagIndeks = 0; dagIndeks < ukeTabell[mobilIndeks].length;
           dagIndeks++) {
        System.out.print(ukeTabell[mobilIndeks][dagIndeks] + "\t");
      }
      System.out.println();
    }

    // Problem (2) Finn totalt antall tekstmeldinger sendt fra
    // mobiltelefon med indeks 1 for hele uken.
    int sumUke = 0;
    for (int dagIndeks = 0; dagIndeks < ukeTabell[1].length; dagIndeks++) {
      sumUke += ukeTabell[1][dagIndeks];
    }
    System.out.println(
      "Totalt antall meldinger sendt fra mobiltelefon med indeks 1: " + sumUke);

    // Problem (3) Finn totalt antall meldinger sendt fra alle
    // mobiltelefoner på onsdag (dag indeks 2)
    int sumMeldinger = 0;
    for (int mobilIndeks = 0; mobilIndeks < ukeTabell.length; mobilIndeks++) {
      sumMeldinger += ukeTabell[mobilIndeks][2];
    }
    System.out.println(
      "Totalt antall meldinger sendt fra alle mobiltelefoner på onsdag: " +
      sumMeldinger);

    // Problem (4) Finn hvilke dager totalt antall meldinger sendt fra
    // alle mobiltelefoner var over 100.
    for (int dagIndeks = 0; dagIndeks < ukeTabell[0].length; dagIndeks++) {
      int sumDag = 0;
      for (int mobilIndeks = 0; mobilIndeks < ukeTabell.length; mobilIndeks++) {
        sumDag += ukeTabell[mobilIndeks][dagIndeks];
      }
      if (sumDag > 100)
        System.out.println("Ukedag med indeks " + dagIndeks +
                           " hadde over 100 tekstmeldinger.");
    }
  }
}
```

Utdata fra programmet:

```
Mobil indeks 0: 12    10    22    33    19    27    16
Mobil indeks 1: 45    55    44    34    39    15    11
Mobil indeks 2: 18    26    36    40    24    11    20
Totalt antall meldinger sendt fra mobiltelefon med indeks 1: 243
Totalt antall meldinger sendt fra alle mobiltelefoner på onsdag: 102
Ukedag med indeks 2 hadde over 100 tekstmeldinger.
Ukedag med indeks 3 hadde over 100 tekstmeldinger.
```

Bruk av flerdimensjonale tabeller

Program 3.17 benytter den flerdimensjonale tabellen `ukeTabell` til å løse en del problemstillinger.

Problem (1) er å skrive ut opplysningene oppstilt i tabellform. Hver mobilindeks gjennomgås i tur og orden med en ytre løkke, og uketabellen for hver mobiltelefon skrives ut som en rad av tall i en indre løkke.

Problem (2) finner totale antall tekstmeldinger sendt fra mobiltelefon med indeks 1. Dette gjøres ved å summere elementene i rad `ukeTabell[1]`, der hvert element i raden er gitt ved `ukeTabell[1][dagIndeks]`, der `dagIndeks` går fra `0` til `ukeTabell[1].length-1`.

Problem (3) i Program 3.17 finner totalt antall meldinger sendt fra alle mobiltelefoner på onsdag (dag indeks 2). Dette gjøres ved å summere verdiene til alle rader for en bestemt kolonne (dag indeks 2), der hvert element i kolonnen er gitt ved `ukeTabell[mobilIndeks][2]` og `mobilIndeks` går fra `0` til `ukeTabell.length`.

Problem (4) i Program 3.17 finner hvilke dager totalt antall meldinger sendt fra alle mobiltelefoner var over 100. Vi må summere alle telefoner for hver dag (analogt med problem (3)) og sammenligne for å finne ut om summen er større enn grensen på 100 meldinger. Den indre løkken summerer hver kolonne (dvs. summerer alle telefoner for en dag), og den ytre løkken sørger for at alle kolonner (dvs. alle dager) blir behandlet. Siden alle rader har samme lengde, har vi benyttet `ukeTabell[0].length` for å angi antall dager i uken for alle telefoner.

Flerdimensjonale tabeller med variabel radlengde

Hver rad i en todimensjonal tabell er en enkel tabell, men slike indre tabeller trenger *ikke* å ha samme lengde. Lengden på radene kan være forskjellige. Dersom vi vil lage en oversikt over måling av nedbør på forskjellige steder i hvert fylke, kan hver rad representere et fylke og hver kolonne representere måling på et bestemt sted. Det er ikke sikkert at alle fylker har samme antall målesteder. Koden nedenfor viser hvordan en tabell som modellerer dette kan opprettes og initialiseres. Legg merke til at i linje (1) spesifiserer vi lengden på den første dimensjonen fra venstre som skal tilsvare antall fylker. Siden hvert element i tabellen `nedbør` er en enkel tabell, kan hver rad opprettes individuelt, slik som i (2). Etterpå kan tabellen `nedbør` initialiseres på vanlig måte.

```
// Opprett en todimensjonal tabell med tilstrekkelige antall rader for fylker.
double[][] nedbør = new double[3][]; // (1)

// (2) Opprett en enkel tabell for hvert fylke med tilstrekkelige antall målesteder.
nedbør[0] = new double[2]; // To målesteder
nedbør[1] = new double[1]; // Ett målested
nedbør[2] = new double[4]; // Fire målesteder
```

Alternativt kan vi bruke blokk-notasjon for å opprette og initialisere tabellen `nedbør`, dersom alle målingene for hvert fylke er kjent (se Program 3.18). Legge merke til at den indre `for`-løkken i linje (2) eksplisitt bruker lengden til hver rad i tur og orden.

Legg merke til at vi ikke kan endre på antall rader i en flerdimensjonal tabell, men vi kan endre på hvilke indre tabeller den inneholder. For eksempel, dersom vi skal erstatte alle nedbørstall for et fylke, kan vi lage en ny enkel tabell og erstatte den gamle indre tabellen på følgende måte:

```
nedbør[fylkeNr] = new double[] {25.5, 12.75, 9.5}; // Tre målesteder
```

Program 3.18 *Todimensjonal tabell med varierende radlengde*

```
public class VarierendeRadlengde {

  public static void main(String args[]) {

    // (1) Oppretter og initialiserer den todimensjonale tabellen nedbør.
    double[][] nedbør = {
      {56.6, 30.2},          // To målesteder
      {20.5},                // Ett målested
      {15.8, 7.0, 45.8, 0.6} // Fire målesteder
    };

    // Skriv ut målingene fra den todimensjonale tabellen nedbør.
    for (int fylkeNr = 0; fylkeNr < nedbør.length; fylkeNr++) {
      System.out.printf("Nedbør for fylke%2d: ", fylkeNr);
      for (int stedIndeks = 0; stedIndeks < nedbør[fylkeNr].length; stedIndeks++)//(2)
        System.out.printf("%10.2f", nedbør[fylkeNr][stedIndeks]);
      System.out.println();
    }
  }
}
```

Utdata fra programmet:

```
Nedbør for fylke 0:      56,60     30,20
Nedbør for fylke 1:      20,50
Nedbør for fylke 2:      15,80      7,00     45,80      0,60
```

3.7 Mer om iterering over samlinger: den forenklete for-løkken

Ofte er det bruk for å behandle *alle* elementer i en samling, der vi bare er interessert i å *lese* verdiene i tur og orden, uten å endre selve samlingen. Tidligere har vi brukt en `for`-løkke til dette formålet, for eksempel for å få tak i verdiene i tabellen `antallMeldinger` i Problem (1) fra Program 3.15:

```
// Problem (1) Finn antall dager som har antall meldinger
   // større eller lik et visst antall tekstmeldinger.
   ...
   for (int indeks = 0; indeks < antallMeldinger.length; indeks++ ) {
     if (antallMeldinger[indeks] >= grense)
       antallDager++;
   }
```

Den *forenklete* `for`-løkken (eng. *foreach loop*) er skreddersydd for å behandle alle elementer i en samling, og i løkkekroppen er det mulig å få tak i elementverdiene i samlingen, én om gangen. Figur 3.19a viser formen på den forenklete `for`-løkken og Figur 3.19b viser omskriving av `for`-løkken gitt ovenfor.

Figur 3.19 *Den forenklete for-løkken*

```
for (løkkevariabeldeklarasjon : samlingen )
  løkkekropp
```

(a) Formen på den forenklete `for`-løkken

```
for (int element : antallMeldinger) {
  if (element >= grense)
    antallDager++;
}
```

(b) Eksempel på den forenklete `for`-løkken

I denne boken skal vi bruke betegnelsen `for(;;)`-løkke på den «klassiske» `for`-løkken og betegnelsen `for(:)`-løkke på den forenklete `for`-løkken. Løkkekroppen utføres for *hver verdi* (angitt ved løkkevariabelen) i samlingen. I Figur 3.19b utføres `if`-setningen i løkkekroppen for hvert element i tabellen `antallMeldinger`.

Den forenklete `for`-løkken kjører gjennom den angitte samlingen, og for hver gjennomkjøring blir løkkekroppen utført og løkkevariabelen blir tildelt en ny verdi fra samlingen, helt til alle verdiene har blitt behandlet. Løkkevariabelen må deklareres i «hodet» til `for(:)`-løkken, og er kun synlig i løkkekroppen.

Dersom koden i løkkekroppen endrer verdien i løkkevariabelen, endres ikke verdien i samlingen. Dersom vi skal få tak i indeks til elementene i en tabell eller endre verdiene i en tabell, er det hensiktsmessig å bruke en `for(;;)`-løkke.

Vi kan også nøste `for(:)`-løkker, for eksempel for å kjøre gjennom alle verdier i en flerdimensjonal tabell. Problem (1) fra Program 3.17 bruker to nøstede `for(;;)`-løkker:

```
// Problem (1) Skriv ut opplysninger i tabellform
   for (int mobilIndeks = 0; mobilIndeks < ukeTabell.length; mobilIndeks++) {
     System.out.print("Mobil indeks " + mobilIndeks + ": ");
     for (int dagIndeks = 0; dagIndeks < ukeTabell[mobilIndeks].length;
          dagIndeks++) {
```

```
      System.out.print(ukeTabell[mobilIndeks][dagIndeks] + "\t");
    }
    System.out.println();
  }
```

Koden ovenfor kan forenkles ved å bruke to nøstede `for(:)`-løkker:

```
// Problem (1) Skriv ut opplysninger i tabellform
  int radNr = 0;
  for (int[] mobilTabell : ukeTabell) {
    System.out.print("Mobil indeks " + (radNr++) + ": ");
    for (int meldinger : mobilTabell) {
      System.out.print(meldinger + "\t");
    }
    System.out.println();
  }
```

Koden ovenfor kjører gjennom en todimensjonal tabell (`int[][]`) angitt ved variabelen `ukeTabell`. Hver element i denne todimensjonale tabellen har typen `int[]`, som er typen til løkkevariabelen `mobilTabell` i den ytre løkken. Den indre løkken kjører gjennom en enkel tabell med heltallsverdier (`int[]`). I den indre `for(:)`-løkken angir løkkevariabelen `meldinger` en heltallsverdi i en enkel tabell angitt ved løkkevariabelen `mobilTabell` deklarert i den ytre løkken. For å få tak i indeksen til hvert element i den todimensjonale tabellen `ukeTabell`, bruker vi en ekstra variabel (`radNr`).

Senere i boken skal vi se på andre samlinger enn tabeller som også kan gjennomkjøres med `for(:)`-løkken (se oppramstyper i avsnitt 4.6 og dynamiske datastrukturer i kapittel 10).

3.8 Pseudo-slumptallgenerator

I simulering og dataspill er det ofte behov for å generere tilfeldige tall. For et terningspill er det, for eksempel, nødvendig å simulere et terningkast. Ideelt sett er det lik sannsynlighet (en sjettedel) for at hvert tall (fra 1 til 6) på terningen kan bli resultatet av et terningkast. Tall generert på denne måten kalles *slumptall* (eng. *random numbers*). Å generere ekte slumptall vha et program er ikke mulig. Å garantere lik sannsynlighet for alle tilfeller er svært vanskelig, slik at vi må nøye oss med *pseudo-slumptall* (eng. *pseudo-random numbers*), dvs. tall som oppfører seg tilnærmet som slumptall. Mye forskning har gått ut på å lage matematiske formler som kan produsere gode pseudo-slumptall. Klassen `java.util.Random` implementerer pseudo-slumpverdigeneratorer for primitive datatyper (`booelan`, `byte`, `int`, `long`, `float`, `double`), men her skal vi nøye oss med å se på en pseudo-slumptallgenerator for `int`-verdier.

Først oppretter vi et objekt av klassen `Random`:

```
Random generator = new Random();
```

Deretter kaller vi metoden `nextInt()` på dette objektet:

```
int nyttTall = generator.nextInt();
```

Metodekallet ovenfor vil returnere et tilfeldig heltall i intervallet [-2^{31}, $2^{31}-1$] som er verdiområdet for datatypen `int`.

Svært ofte er vi interessert i å generere slumptall innenfor et mindre verdiområde. Følgende kode vil for eksempel returnere et tilfeldig heltall i intervallet [0, 10]:

```
nyttTall = generator.nextInt(11);      // tilfeldig heltall i [0, 10]
```

Dersom parameterverdien i metodekallet er *n*, blir heltall generert fra intervallet [0, *n*-1]. Dersom vi ønsker en annen nedre grense enn verdien 0, kan vi legge til en *forskyvning*:

```
nyttTall = 2 + generator.nextInt(11);  // tilfeldig heltall i [2, 12]
```

Forskyving på 2 ovenfor avbilder intervallet [0, 10] på [2, 12]. Dersom verdiene i intervallet skal ha *avstand* større enn 1, kan vi gange det genererte slumptallet med avstandsverdien:

```
nyttTall = 2 + 3*generator.nextInt(5); // tilfeldig heltall i {2, 5, 8, 11, 14}
```

Avstand på 3 mellom verdiene vil resultere i at uttrykket `3*generator.nextInt(5)` vil returnere verdier fra mengden {0, 3, 6, 9, 12}, og forskyvning på 2 vil sørge for at verdien til variabelen `nyttTall` vil være fra mengden {2, 5, 8, 11, 14}.

Simulering av terningkast

Program 3.19 viser hvordan vi kan kaste to terninger:

```
int resultat = (1 + generator.nextInt(6)) + // terning 1
               (1 + generator.nextInt(6));  // terning 2
```

Programmet kaster terningene et visst antall ganger lest fra tastaturet. Det teller hvor mange ganger verdiene fra 2 til 12 forekommer, dvs. resultatet av et kast med to terninger, i en heltallstabell. Denne hyppigheten blir skrevet ut til slutt.

Slik vi har brukt pseudo-slumptallgeneratoren hittil, kan ikke den samme rekkefølgen av slumptallverdier garanteres hver gang programmet kjøres. Dette er tilfellet fordi pseudo-slumptallgenereringen er basert på klokkeslettet i det programmet starter. Dette er opplagt forskjellig hver gang programmet kjøres. Dersom vi ønsker den samme rekkefølgen av slumptallverdier hver gang, kan vi angi et *frø* (eng. *seed*) i `Random`-klassekonstruktøren:

```
Random nyGenerator = new Random(31);  // Frøet er primtall 31.
```

Frøet er vanligvis et primtall, siden slike tall egner seg for å lage gode slumptallgeneratorer.

Program 3.19 *Hyppighet av terningkast med to terninger*

```
import java.util.Random;
import java.util.Scanner;
//Simulerer to terninger ved hjelp av pseudo-slumptall.
//Teller opp resultatet av kast med to terninger i intervallet [2,12]
public class TerningKastHyppighet {
  public static void main(String[] args) {
```

```
    int[] hyppighet = new int[13]; // Vi ser bort i fra hyppighet[0] og hyppighet[1].
    Scanner tastatur = new Scanner(System.in);
    System.out.print("Angi antall ganger terningene skal kastes: ");
    int antall = tastatur.nextInt();

    Random generator = new Random();  // Pseudo-slumptallgenerator

    for (int i = 1; i <= antall; i++) {
      // Kast to terninger.
      int resultat = (1 + generator.nextInt(6)) + // terning 1
      (1 + generator.nextInt(6));  // terning 2
      // Øk hyppighetstelleren for verdien til dette kastet.
      hyppighet[resultat]++;
    }

    for (int i = 2; i < hyppighet.length; i++) {
      System.out.printf("Resultat %1d: %4d, sannsynlighet: %.2f%n",
          i, hyppighet[i], ((double)hyppighet[i]/antall));
    }
  }
}
```

Utdata fra programmet:

```
Angi antall ganger terningene skal kastes: 4000
Resultat 2:  136, sannsynlighet: 0,03
Resultat 3:  220, sannsynlighet: 0,06
Resultat 4:  377, sannsynlighet: 0,09
Resultat 5:  448, sannsynlighet: 0,11
Resultat 6:  521, sannsynlighet: 0,13
Resultat 7:  667, sannsynlighet: 0,17
Resultat 8:  552, sannsynlighet: 0,14
Resultat 9:  420, sannsynlighet: 0,11
Resultat 10:  342, sannsynlighet: 0,09
Resultat 11:  206, sannsynlighet: 0,05
Resultat 12:  111, sannsynlighet: 0,03
```

3.9 KONTROLLSPØRSMÅL

Spørsmål 3.1 En klasse spesifiserer _____ og _____ til objekter som kan opprettes fra klassen.

Spørsmål 3.2 Instansvariabler representerer _____, og instansmetoder representerer _____ til objekter som kan opprettes fra en klasse.

Spørsmål 3.3 Hvilke utsagn er sanne:

(a) Et objekt opprettes fra en klasse.

(b) En klasse opprettes fra et objekt.

(c) En referanse lagrer referanseverdi til et objekt.

(d) En referanse lagrer referanseverdi til en klasse.

(e) To objekter kan være aliaser.

(f) Referanser som refererer til samme objekt, er aliaser.

(g) Referansene `ref1` og `ref2` er aliaser dersom `(ref1==ref2)` er sann.

Spørsmål 3.4 Følgende kodelinje vil føre til at et objekt av klassen `CD` blir opprettet og blir referert ved variabelen `minCD`. Sant eller usant?

```
CD minCD;
```

Spørsmål 3.5 Gitt klassen `CD`, omskriv følgende kodelinjer slik at de er lovlige:

```
CD cd1 = CD;
CD cd2 = CD();
CD cd3 = new CD;
CD cd4 = new;
new CD cd5;
```

Spørsmål 3.6 Gitt en referanse til et objekt, hvilken notasjon må brukes for å kalle en metode eller for å referere til et felt i objektet?

Spørsmål 3.7 Gitt referansen `minCD`, som refererer til et objekt av klassen `CD`, hvilke uttrykk er lovlige:

(a) `minCD.hentTittel()`

(b) `CD.hentTittel()`

(c) `minCD.hentTittel`

(d) `minCD.tittel`

(e) `CD.tittel`

Spørsmål 3.8 Hva er forskjellen mellom `'z'` og `"z"`?

Spørsmål 3.9 Anta at vi har to referanser, `str1` og `str2`, som refererer til `String`-objekter. Dersom `(str1==str2)` er sann, er `str1.equals(str2)` alltid sann. Men det motsatte er ikke alltid sant. Forklar hvorfor.

Spørsmål 3.10 Hva vil bli utskrift fra følgende kodelinjer:

```
System.out.println(2000 + 2);
System.out.println(2000 + '2');
System.out.println(2000 + "2");
```

Spørsmål 3.11 Hvor mange objekter av klassen `CD` eksisterer etter utføring av denne koden:

```
CD cd1 = new CD();
Cd cd2 = cd1;
```

Spørsmål 3.12 Hva vil bli utskrift fra følgende kodelinjer:

```
int teller = 4;
System.out.println(teller++);
System.out.println(++teller);
System.out.println(teller--);
System.out.println(--teller);
```

Spørsmål 3.13 Dersom antall gjennomganger er kjent, vil man velge en `for`-løkke eller en `while`-løkke?

Spørsmål 3.14 Hvilke deler (initialisering, løkkebetingelse, oppdatering) kan utelates i en `for`-løkke?

Spørsmål 3.15 Hvilke utsagn er sanne om en `for`-løkke? Anta at løkkekroppen ikke har noen setninger som sender kontroll ut av løkken.

(a) Initialiseringen utføres én gang.

(b) Løkkebetingelsen blir utført minst én gang.

(c) Løkkekroppen blir utført minst én gang.

(d) Dersom løkkekroppen blir utført, blir oppdateringen og løkkebetingelsen alltid utført.

Spørsmål 3.16 Skriv følgende kode som en `for`-løkke. Hvor mange ganger blir løkkekroppen utført?

```
int i = 10, sum = 0;
while (i >= 5) {
  sum += i;
  i -= 2;
}
int sum = 0;
for (int i = 10; i >= 5; i -= 2) {
  sum += i;
}
```

Spørsmål 3.17 Hva er verdien til variabelen `i` etter at `for`-løkken er utført? Hvor mange ganger blir løkkekroppen utført?

(a)
```
int i;
for (i = 0; i < 10; i += 2)
  System.out.println(i);
```

(b)
```
for (int i = 0; i < 10; i += 3)
  System.out.println(i);
```

Spørsmål 3.18 Hvilke utsagn er sanne om en `switch`-setning:

(a) En `default`-etikett må alltid være med i en `switch`-setning.

(b) Boolske verdier og flyttallsverdier er ikke lovlige som `case`-etiketter.

(c) En strenglitteral kan brukes som `case`-etikett.

(d) En `break`-setning er en del av en `switch`-setning.

Spørsmål 3.19 Omskriv følgende kode ved hjelp av en `switch`-setning:

```
if (i == 10 || i == 20)
  System.out.println("10 eller 20");
else if (i == 15)
  System.out.println("15");
else
  System.out.println("Ulovlig");
```

Spørsmål 3.20 Gitt at referansen `rad` refererer til en tabell, hvordan finner vi hvor mange elementer denne tabellen har?

Spørsmål 3.21 Nevn tre forskjellige måter å bruke []-notasjonen på.

Spørsmål 3.22 Hvilke kodelinjer oppretter en tabell av heltall, som blir referert ved referansen `alder`?

(a) `int[] alder;`

(b) `int[] alder = int[100];`

(c) `int alder = new int[10];`

(d) `int[] alder = new int(10);`

(e) `int[] alder = new int[];`

(f) `int[] alder = new int[10];`

Spørsmål 3.23 Hvilke utsagn er sanne om tabeller?

(a) Alle tabeller har en metode med navn `length()`.

(b) Alle tabeller har et felt med navn `length`.

(c) Gitt at referansen `tabRef` refererer til en tabell av heltall, er første element i denne tabellen angitt ved `tabRef[1]`.

(d) Når en tabell opprettes med `new` *type*`[`*n*`]`, blir alle elementene i tabellen alltid initialisert med standardverdien til *type*.

Spørsmål 3.24 Hvilken kode vil opprette og initialisere en tabell med `CD`-er?

(a)
```
CD[] samling = { new CD(), new CD(), new CD() };
```

(b)
```
CD[] samling;
samling = { new CD(), new CD(), new CD() };
```

(c)
```
CD[] samling = new CD[3];
samling = { new CD(), new CD(), new CD() };
```

(d)
```
CD[] samling = new CD[3];
samling[0] = new CD();
samling[1] = new CD();
```

```
samling[2] = new CD();
```

(e)
```
CD[] samling = new CD[10];
samling = new CD[] { new CD(), new CD(), new CD() };
```

(f)
```
CD[] samling = new CD[] { new CD(), new CD() };
```

Spørsmål 3.25 Hva er typen til `navn`, `navn[1]` og `navn[1][2]` når vi har følgende deklarasjon?

```
String[][] navn = String[5][4];
```

Spørsmål 3.26 Hvilken kode vil opprette og initialisere en *tabell av tabeller* med `CD`-er?

(a)
```
CD[][] nySamling = {
  { new CD() },
  { new CD(), new CD() },
  { new CD(), new CD(), new CD() }
};
```

(b)
```
CD[][] nySamling;
 nySamling = {
  { new CD() },
  { new CD(), new CD() },
  { new CD(), new CD(), new CD() }
};
```

(c)
```
CD[][] nySamling = new CD[3][2];
nySamling = {
  { new CD(), new CD() },
  { new CD(), new CD() },
  { new CD(), new CD() }
};
```

(d)
```
CD[][] nySamling = new CD[3][2];
nySamling[0][0] = new CD(); nySamling[0][1] = new CD();
nySamling[1][0] = new CD(); nySamling[1][1] = new CD();
nySamling[2][0] = new CD(); nySamling[2][1] = new CD();
```

(e)
```
CD[][] nySamling  = new CD[][] {
  { new CD() },
  { new CD(), new CD() },
  { new CD(), new CD(), new CD() }
};
```

(f)
```
CD[][] nySamling  = new CD[][] {
  null, null, null
};
nySamling[0] = new CD[] { new CD() };
nySamling[1] = new CD[] { new CD(), new CD() };
nySamling[2] = new CD[] { };
```

(g)
```
CD[][] nySamling = {
  { new CD() },
  { new CD(); new CD(); new CD() },
  { new CD(); new CD() }
};
```

Spørsmål 3.27 Hvilke utsagn er sanne om en `for(:)`-løkke? Anta at løkkekroppen ikke har noen setninger som sender kontroll ut av løkken.

(a) Løkkevariabelen kan ikke deklareres utenfor løkken.

(b) Samlingen kan være tom.

(c) Løkkekroppen blir utført minst én gang.

(d) Tilordning til løkkevariabelen vil endre verdiene i samlingen.

Spørsmål 3.28 Hvilke `for(:)`-løkker er lovlige? Anta at vi har følgende deklarasjon:

```
String[] samling = new String[5];
```

(a)
```
String element;
for (element : samling) {
  System.out.println(element);
}
```

(b)
```
for (String element : samling) {
  System.out.println(element);
}
```

(c)
```
for (String element : samling) {
  element = "2005";
  System.out.println(element);
}
```

(d)
```
int i = 0;
for (String element : samling) {
  System.out.println(i++ + ": " + element);
}
```

Spørsmål 3.29 Hvilke utsagn er sanne om pseudo-slumptallgenerering? Anta følgende deklarasjoner:

```
Random generator = new Random();
int tall;
```

(a) Variabelen `tall` kan bare få en verdi fra intervallet [0, 5], inklusiv, etter tilordningen:

```
tall = generator.nextInt(5);
```

(b) Variabelen `tall` kan bare få en verdi fra intervallet [-5, 0], inklusiv, etter tilordningen:

```
tall = -5 + generator.nextInt(6);
```

(c) Variabelen `tall` kan bare få en verdi fra intervallet [-5, 5], inklusiv, etter tilordningen:

```
tall = -5 + generator.nextInt(10);
```

(d) Variabelen `tall` kan bare få en verdi fra mengden {0, 2, 4, 6, 8} etter tilordningen:

```
tall = 2 * generator.nextInt(5);
```

(e) Variabelen `tall` kan bare få en verdi fra mengden {3, 5, 7, 9, 11} etter tilordningen:

```
tall = 3 + 2 * generator.nextInt(5);
```

3.10 OPPGAVER

Oppgave 3.1 Lag et program som leser inn en sekvens av heltall fra tastaturet, og skriver ut en rapport om hvor mange ganger tallene 0, 1, ..., 9 forekom i den innleste sekvensen. Anta at sekvensen avsluttes med et negativt heltall. For eksempel, hvis brukeren skriver

```
8
0
2
8
8
9
5
9
-3
```

skal programmet skrive denne rapporten:

```
0 forekommer 1 gang
2 forekommer 1 gang
5 forekommer 1 gang
8 forekommer 3 ganger
9 forekommer 2 ganger
```

Hint: Opprett en tabell for ti heltall som kan lagre antall ganger et tall har forekommet (`int[] ganger = new int[10];`).

Oppgave 3.2 Utvid oppgave 3.1 slik at programmet også skriver ut et *histogram* (også kalt *søylediagram*) der søylene er liggende. En søyle tilsvarer antall forekomster av hvert tall.

Eksempel:

```
0: *
1: ***
2: *
3:
4: **
5: *
6: ****
7:
```

```
8: ***
9: **
```

Oppgave 3.3 Modifiser programmet fra oppgave 3.2 slik at programmet skriver ut et histogram der søylene er stående.

Eksempel:

```
 *
     *               *     *
     *         *     *     *  *
  *  *  *     *  *  *     *  *
==0==1==2==3==4==5==6==7==8==9==
```

Oppgave 3.4 Et *palindrom* er en streng som har samme tegnsekvens enten man leser den forfra eller bakfra. For eksempel er `"aha"`, `"abba"` og `"00700"` alle palindromer ifølge denne definisjonen. Strengen kan godt ha flere ord, og du kan velge å ignorere mellomrom i den hvis du vil. For eksempel er `"roma amor"` et palindrom. Skriv et program som gitt en streng rapporterer om strengen er et palindrom. Programmet må bruke en `for`-løkke til å bestemme et palindrom.

Oppgave 3.5 Lag et program som leser inn en liste av ti heltall fra tastaturet og legger dem inn i en tabell som er indeksert fra 0 til 9, og så rapporterer posisjonen til det laveste tallet i tabellen.

Hvis posisjonen til det minste tallet er en annen enn posisjon 0, så skal programmet bytte om posisjonene til tallet i posisjon 0 og det minste tallet i tabellen. Nå er det minste tallet i tabellen posisjon 0, og tallet som tidligere var i posisjon 0, er nå der det minste tallet var. Til slutt skal programmet skrive ut hele tabellen slik som den nå ser ut.

Oppgave 3.6 Lag et program som leser inn en streng og rapporterer hvor mange små og store bokstaver som forekommer i strengen. Programmet skal bruke `String`-klassen og ikke tabeller.

Eksempel:

```
Skriv en streng:
STOR i KJEFTEN, liten i toppen.
Antall små bokstaver: 13
Antall store bokstaver: 11
Antall små bokstaver er større enn antall store bokstaver.
```

Oppgave 3.7 Lag et program som lar brukeren gjette en hemmelig kode. Programmet skal bruke `String`-klassen og ikke tabeller.

Oppgave 3.8 Omskriv Program 3.16 slik at det bruker den forenklete `for`-løkken.

Oppgave 3.9 Utvid Program 3.17 slik at verdier til den todimensjonale tabellen `ukeTabell` blir lest fra tastaturet.

Oppgave 3.10 Utvid Program 3.17 slik at det beregner totalt antall tekstmeldinger sendt hver dag og totalt antall tekstmeldinger sendt fra hver mobiltelefon:

```
Dag nr.         0    1    2    3    4    5    6    Sum(mobil):
Mobil indeks 0: 12   10   22   33   19   27   16   139
Mobil indeks 1: 45   55   44   34   39   15   11   243
Mobil indeks 2: 18   26   36   40   24   11   20   175
Sum(dag):       75   91   102  107  82   53   47
```

Oppgave 3.11 Lag et program som gitt et heltall n beregner og rapporterer verdien til $n!$, dvs. 1*2*3*4* . . . *(n-1)*n, der $n > 0$.

Oppgave 3.12 Omskriv `switch`-setningen i Program 3.12 ved hjelp av en `if-else`-setning.

Oppgave 3.13 Skriv et program som leser en streng og skriver ut substrenger angitt med `'|'`-tegnet fra strengen. For eksempel vil strengen `"DOTT-KOM|IT-plass 1|5020|Bergen"` resultere i at følgende substrenger blir skrevet ut:

```
DOTT-KOM
IT-plass 1
5020
Bergen
```

Oppgave 3.14 Skriv et program som, gitt et beløp i kroner og ører, veksler dette i sedler og mynter. Det finnes sedler pålydende 1000 kroner, 500 kroner, 200 kroner, 100 kroner og 50 kroner. Det finnes mynter for 20 kroner, 10 kroner, 1 krone og 50 øre.

Ørebeløp mindre enn 25 øre avrundes nedover til 0 øre, ørebeløp større enn eller lik 25 øre, men mindre enn 75 øre, avrundes til 50 øre, og ørebeløp større enn eller lik 75 øre avrundes oppover til 1 krone.

Programmet forsøker alltid å veksle slik at sedlene og myntene det veksles til, har høyest mulig pålydende. For eksempel vil kr 5732.60 bli vekslet slik:

- 1000 kr: 5 stk.
- 500 kr: 1 stk.
- 200 kr: 1stk.
- 20 kr: 1 stk.
- 10 kr: 1 stk.
- 1 kr: 2 stk.
- 50 øre: 1 stk.

Oppgave 3.15 Skriv et program for kasinospillet *Craps*. Spillet går ut på å kaste to terninger, der summen av «øynene» øverst på de to terningene (kalt *poengsum*) bestemmer videre forløp i spillet. Programmet simulerer spillet ved å håndheve følgende spilleregler:

- Hvis poengsummen er 7 eller 11 ved 1. kast, har spilleren vunnet. Poengsummen kalles for *natural*.

- Hvis poengsummen er 2 (*snake eyes*), 3 (*cross eyes*) eller 12 (*box cars*) ved 1. kast, har spilleren tapt.
- Hvis poengsummen er 4, 5, 6, 8, 9 eller 10 ved 1. kast, etablerer denne poengsummen et *point* og spilleren kan kaste videre. Spilleren fortsetter å kaste terningene helt til poengsummen blir enten lik 7 eller lik *point*. Poengsummen lik 7 betyr spilleren har tapt, mens poengsummen lik *point* betyr spilleren har vunnet.

Oppgave 3.16 Skriv et program for *MasterMind*-spillet. Spillet går ut på å gjette en hemmelig kode som består av et fast antall siffer. Programmet lager koden, og gir tilbakemelding til spilleren for hvert forsøk i følgende form:

- Antall *okser* (eng. *bulls*) i forsøket, dvs. antall siffer som er i koden og som står på riktig plass i koden.
- Antall *kuer* (eng. *cows*) i forsøket, dvs. antall siffer som er i koden men ikke står på riktig plass i koden.

I kildekoden skal det være mulig på en enkel måte å endre antall siffer som inngår i koden og antall forsøk en spiller kan foreta seg. Klassen `Random` skal brukes til å lage koden. I denne versjonen skal det *ikke* brukes tabeller, bare `String`-klassen, for lagring av koden og forsøket.

Eksempel (der koden er 9464):

```
Forsøk nr. 1: 4444
Antall okser: 2
Antall kuer: 0
Forsøk nr. 2: 4675
Antall okser: 0
Antall kuer: 2
Forsøk nr. 3: 3494
Antall okser: 2
Antall kuer: 1
...
```

Oppgave 3.17 Omskriv programmet for *MasterMind*-spillet fra oppgave 3.16, slik at det bruker tabeller, istedenfor `String`-klassen, for lagring av koden og forsøket.