## Kapittel 3:
# Bruk av objekter

***Redigert av:***

Khalid Azim Mughal (`khalid@ii.uib.no`)

***Kilde:***

*Java som første programmeringsspråk (3. utgave)*

Khalid Azim Mughal, Torill Hamre, Rolf W. Rasmussen
Cappelen Akademisk Forlag, 2006.
ISBN: 82-02-24554-0
`http://www.ii.uib.no/~khalid/jfps3u/`
*(NB! Boken dekker opptil Java 6, men notatene er oppdatert til Java 7.)*



## Emneoversikt

- Innføring i objektmodellen
- Strenger (`String`-klassen)
- Primitive verdier som objekter
- Mer om løkker: `for`-løkke
- Mer om valgsetninger: `switch`-setning
- Tabeller (*arrays*)
  - deklarasjon, opprettelse, initialisering, bruk
  - fler-dimensjonale tabeller
- Den forenklete `for`-løkken
- Pseudo-slumptallgenerator

## Innføring i objektmodellen

| Viktige aspekter ved objektmodellen: | Realisering av objektmodellen v.h.a. |
|---|---|
| • Abstraksjoner<br>• Innkapsling | • Klasser<br>• Objekter |



## Hva menes med en abstraksjon?

- En *abstraksjon* betegner vesentlige egenskaper ved et objekt, slik at vi kan skille det fra andre typer objekter.
  - Vesentlige egenskaper er avhengige av problemstillingen.
- Abstraksjoner kan dreie seg om *håndgripelige* ting (kjøretøy, kart) og *konseptuelle* ting (møter, datoer, forskjellige prosesser).
- *Den største utfordringen i objektorientert systemutvikling er valg av riktige abstraksjoner.*

## Eksempel på en enkel abstraksjon:

- Hvilke vesentlige detaljer beskriver denne *dingsen*?
  - Abstraksjonsnavn: `Lys`
  - Lysets energiforbruk som måles i watt.
  - Lyset kan være av eller på.

  *Andre detaljer som fasong, matt/ikke-matt, lysfarge, sokkelstørrelse, osv. er mindre vesentlige for definisjon av abstraksjonen.*

  *Bestemte lys (objekter) vil ha vesentlige egenskaper spesifisert.*



## Modellering av abstraksjoner v.h.a. *klasser*

- En *klasse* definerer
  - alle *feltvariablene* (egenskaper/ attributter/ data), og
  - alle *metodene* (atferd/ operasjoner)

  til en abstraksjon.

Skjematisk spesifikasjon av en klasse

**Klasse `Lys`**

| Feltvariabler/ egenskaper: | Instansmetoder/ atferd: |
|---|---|
| `antallWatt` | `slåPå()` |
| `indikator` | `slåAv()` |
| | `erPå()` |

Grafisk notasjon for en klasse

| | `Lys` |
|---|---|
| *Feltvariabler* | `antallWatt`<br>`indikator` |
| *Instansmetoder* | `slåPå()`<br>`slåAv()`<br>`erPå()` |

## Klassedeklarasjoner

- En *klasse* betegner en kategori av *objekter*.
  - definerer objektenes atferd og egenskaper.
  - definerer *sjablong*, *arbeidstegning* eller *mal* for å lage objekter (også kalt *instanser*).
- Objekter (f.eks. `en100Wattspære`, `nattpære`, `spotlight`) må eksplisitt lages fra en klasse (f.eks. `Lys`).

| **Begrep som angår objekter:** | **Andre synonymer:** |
|---|---|
| *objekt* | instans |
| *feltvariabler* | *felt*, *instansvariabler*, egenskaper, attributter, data, datamedlem |
| *(instans)metoder* | atferd, operasjoner, medlemsfunksjoner |



## Klassedeklarasjoner (forts.)

```
/**
 * Klassen Lys
 * modellerer abstraksjonen Lys
 */
class Lys {
...
}
```

- Reservert ord `class` angir at vi skal definere en ny klasse.
- `Lys` er navnet på klassen.
- Blokken (som foreløpig er tom) etter klassenavnet vil inneholde definisjonen av klassen.

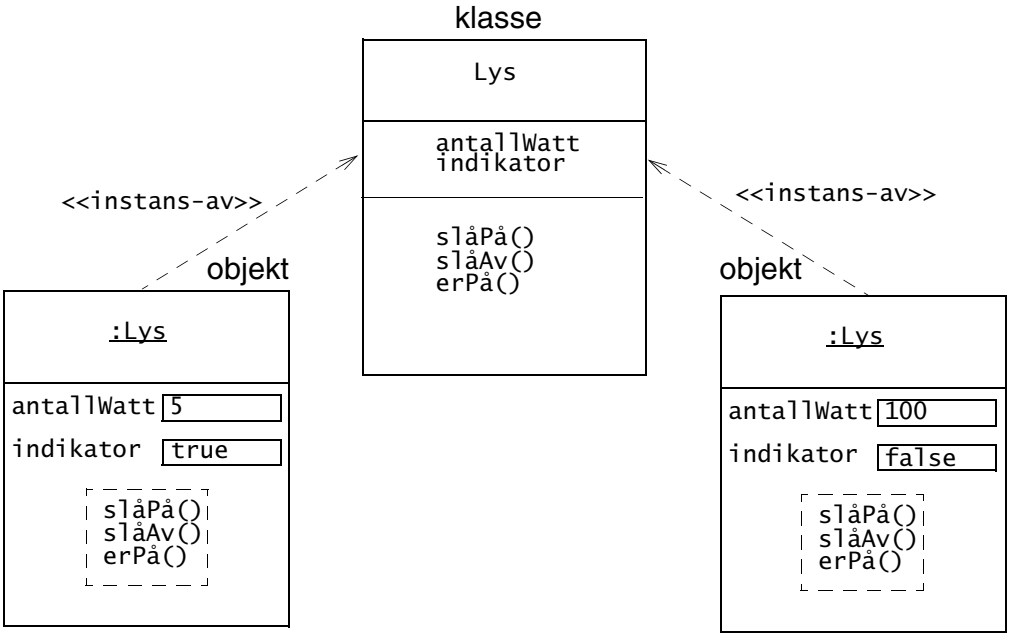
# Kildekode for klassen Lys

```
/** Klassen Lys modellerer abstraksjonen Lys */
class Lys {
  // Feltvariabler
  int antallWatt;    // "størrelsen" på lys.
  boolean indikator; // av == false, på == true

  // Instansmetoder
  void slåPå() {
    indikator = true;
  }
  void slåAv() {
    indikator = false;
  }
  boolean erPå() {
    return indikator;
  }
}
```

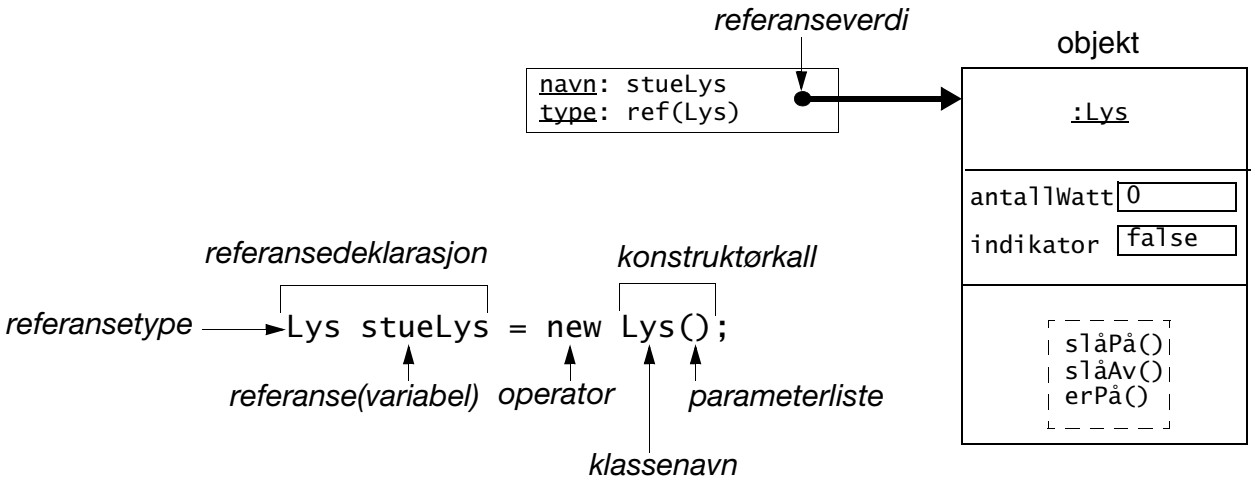


# Objekter som instanser av klasser




(UML: Unified Modeling Language)

# Opprettelse av objekter




Saksgangen i utføring av deklarasjonen ovenfor:

- Et objekt av klasse `Lys` blir opprettet i lageret.
- Alle feltvariabler til objektet blir initialisert til *standardverdier*.



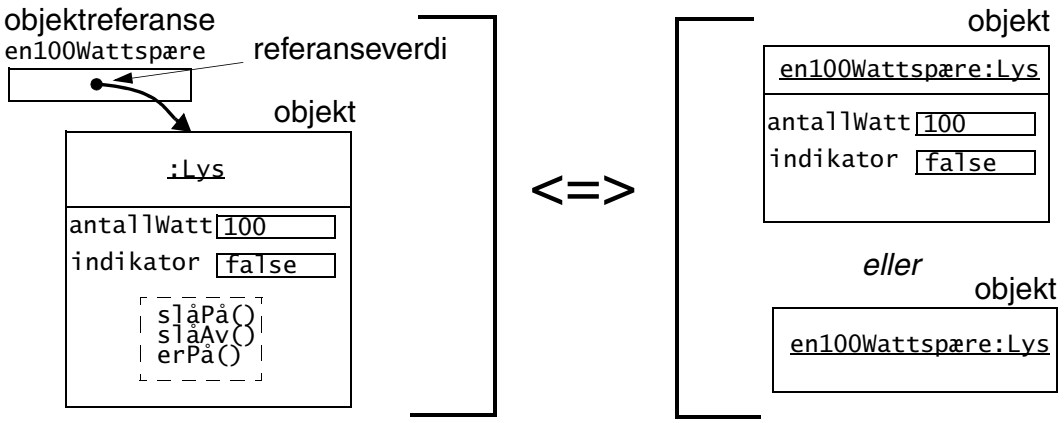
# Objekter, referanseverdier, referanser

- *Objekter* inneholder *feltvariabler*.
- Feltvariabler til forskjellige instanser av klassen `Lys` kan ha forskjellige verdier.
- Hvert objekt er entydig (d.v.s. har entydig identitet), selv om objektene har samme egenskaper.
- Verdiene til objektets feltvariabler utgjør, til enhver tid, objektets *tilstand*.
- *Atferden* til et objekt implementeres v.h.a. metoder.
  - objekter av samme klasse deler metodeimplementasjoner.
  - En nattpære kan gjøre alt som en 100W-lyspære kan gjøre.
- Hvordan skiller vi mellom objekter?
  - En måte er å gi navn til objekter, men det er ikke mulig i Java.
  - Et objekt har en entydig *referanseverdi* som identifiserer det.
  - Denne referanseverdien kan brukes til å håndtere objektet.
  - En referanseverdi kan lagres i en *objektreferanse* (ofte bare kalt *referanse*).
  - Vi sier at en referanse *betegner* det objektet som referanseverdien refererer til.
  - Ofte bruker vi navnet på referansen som om det var navnet på objektet.

*En **referanse** lagrer en **referanseverdi** som betegner et **objekt** opprettet fra en **klasse**.*



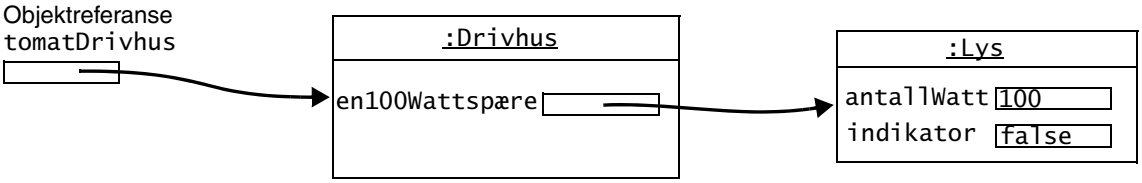
# Referanser og referanseverdier




- Når vi *instansierer* (skaper, lager, oppretter) et nytt objekt fra en klasse,
  - får vi en *referanseverdi* til dette objektet, og
  - vi kan lagre denne *referanseverdien* i en *objektreferanse* (også kalt *referansevariabel*).
- En referanse holder rede på hvor i minnet det nyopprettede objektet er lagret.
- Etter at objektet er instansiert kan *navnet* på denne referansen brukes til å sende *meldinger* til objektet.

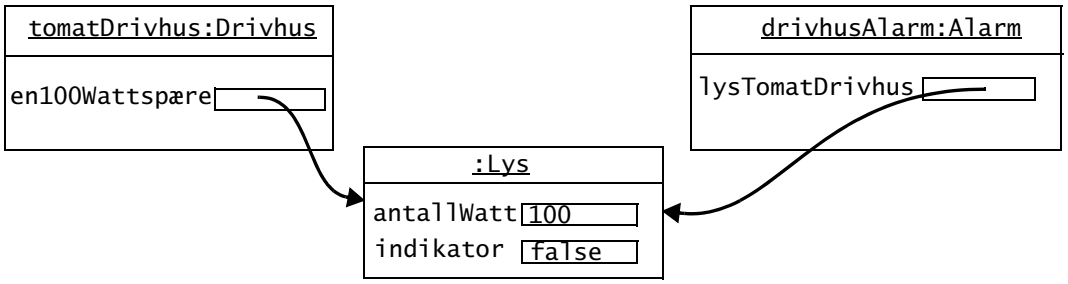
## Referanser og referanseverdier(forts.)

- I Java kan objekter bare inneholde *referanser* til andre objekter, ikke selve objektene.



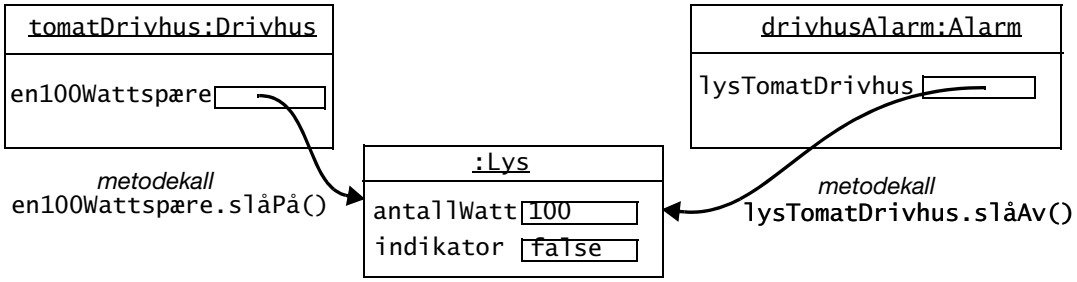

  – Verdien i feltvariabelen `en100Wattspære` er referanseverdien til et objekt av klassen `Lys`. (Legg merke til at en feltvariabel kan være en referanse til et objekt.)

- Samme objekt kan ha *flere* referanser. Slike referanser kalles *aliaser*.

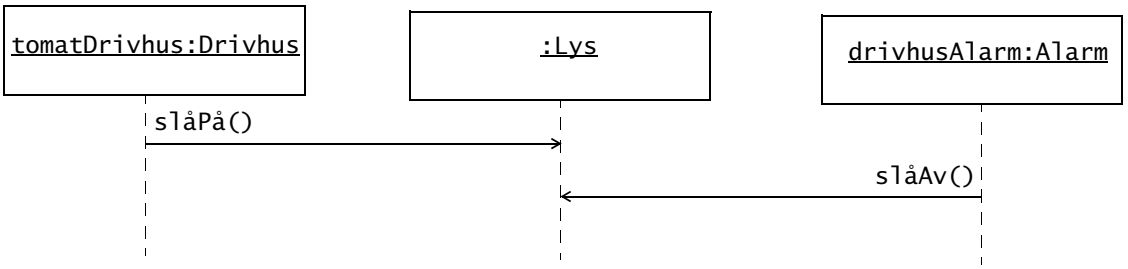





## Kommunikasjon mellom objekter v.h.a. *meldinger*




## Sekvensdiagram



- Gitt en referanse til et objekt, kan vi bruke denne referansen til å sende meldinger til (eller *kalle metoder på*) dette objektet for å foreta forskjellige handlinger.
  - metoder definerer atferd (v.h.a. handlinger).
  - objekter sender meldinger (kaller metoder) til hverandre for å igangsette forskjellige handlinger.
  - siden objektet `tomatDrivhus` har en referanse til et objekt av klassen `Lys`, kan objektet `tomatDrivhus` sende melding `slåPå()` til dette objektet (og det kan også objektet `drivhusAlarm`).



## Meldinger (forts.)

- En *melding* sendt fra et objekt, *sender*, til et annet objekt, *mottaker*, er et *kall* av en bestemt metode som mottakeren forstår.
- Hver melding består av:
  - referanse til instansen (mottaker)
  - navnet til metoden
  - evt. opplysninger (*parametere*) som metoden trenger for å upnbtføre sine handlinger.

```
en100Wattspære.erPå();
```

- Mottaker kan sende tilbake et *svar.*
  - Svar kalles for *returverdi.*
  - Mottaker kan sende tilbake en referanse (til opplysninger) som svar.
  - Mottaker kan sende informasjon (f.eks. resultatet av handlinger som ble utført) som svar til senderen.
  - Det er ikke nødvendig at mottakeren må returnere noe svar i det hele tatt.

```
boolean erPå() {
    return indikator; // Returnerer verdien i indikator.
}
```

## Manipulering av referanser

- En referanse av en spesifikk *referansetype* (f.eks. klasse) kan bare lagre referanseverdier til objekter av denne typen.
- Vi får returnert referanseverdien til objektet som opprettes med `new`-operatoren.

```
Lys stuelys = new Lys(); // Lagrer referanseverdi til Lys-objektet i
                         // referansen stuelys.
Lys nyttLys = stuelys;   // Aliaser. Begge betegner samme Lys-objektet.
                         // Bare referanseverdien blir kopiert, ikke
                         // objektet.
stuelys.slåPå();         // Lys-objektet ble slått på.
nyttLys.slåAv();         // Samme Lys-objektet ble slått av.
```

- Litteralen `null` kan lagres i hvilken som helst referansevariabel.
  – At en referanse er `null`, betyr at den ikke betegner et objekt.

```
nyttLys = null;
```



- En referanse må betegne et objekt *før* den brukes:

```
stuelys.slåPå();         // OK.
nyttLys.slåAv();         // Referansen nyttLys har null-verdi.
                         // Fører til NullPointerException-kjørefeil.
```

## Klassen `java.lang.String`

- En *streng* består av en *endelig sekvens av tegn.*
- Java tilbyr den ferdiglagde klassen `String` for å lage og håndtere strenger.
- Før vi begynner å se nærmere på strenger, ser vi på hvordan tegn blir håndtert i et Java-program.



## Tegnsett

- Tegn representeres i datamaskinen v.h.a. en kode. *Se også vedlegg D*.
- Java bruker et 16-bit UNICODE-tegnsett, som inneholder både ISO-Latin-1- og 7-bit ASCII-tegnsettene.
- Det er viktig å huske at kodene til sifrene (`'0'-'9'`), de store bokstavene (`'A'-'Z'`) og små bokstavene (`'a'-'z'`) følger etter hver andre.

| | Kode | |
|---|---|---|
| Tegn | Desimal, oktal, hexadesimal | UNICODE |
| `'0'` | `48, 060, 0x30` | `\u0030` |
| `'1'` | `49, 061, 0x31` | `\u0031` |
| `'9'` | `57, 071, 0x39` | `\u0039` |
| `'A'` | `65, 0101, 0x41` | `\u0041` |
| `'B'` | `66, 0102, 0x42` | `\u0042` |
| `'Z'` | `90, 0132, 0x5a` | `\u005a` |
| `'a'` | `97, 0141, 0x61` | `\u0061` |
| `'b'` | `98, 0142, 0x62` | `\u0062` |
| `'z'` | `122,0172, 0x7a` | `\u007a` |
| `' '` | `32, 040, 0x20` | `\u0020` |

## Tegn og strenger

- En *tegnlitteral* er et tegn omsluttet i enkle apostrofer:
  - Tegnlitteral `'a'` representerer UNICODE-verdien til tegnet `a`.
  - Tegnlitteral `'\u0061'` representerer også UNICODE-verdien til tegnet `a`.
  - Tegnlitteral `'\n'` representerer UNICODE-verdien til linjeskift.
  - Tegnlitteral `'\t'` representerer UNICODE-verdien til tabulator.
  - En tegnlitteral har type `char`, og opptar 16 bits for lagring.
- En *tegnkonstant* i Java kan deklareres, og innholdet kan ikke endres:

```
final char linjeskift = '\n';
final char tabulator = '\t';
```

- Tegn kan inngå i artimetiske uttrykk, da er det heltalskoden til tegnet som inngår som verdi:

```
int tall = 53 - '0';      // tall får verdi 5 (53 - 48).
int posisjon = 'K' - 'A'; // posisjon får verdi 10 (75 - 65).
```

- En *strenglitteral* er en sekvens av tegn omsluttet av doble apostrofer, f.eks.:

```
"Velkommen til Portveien 2!"
"Feil i inndata.\nTast på nytt." // vil gi utskrift på to linjer
"" // betegner den tomme strengen som har ingen tegn, og lengde 0.
```



## Strenger

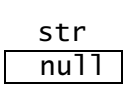
- Objekter av klassen `String` brukes til å håndtere sekvenser av tegn, dvs. strenger.
- Java har støtte for *deklarasjon*, *opprettelse*, *initialisering* og *manipulering* av `String`-objekter.
- Et `String`-objekt lagrer en streng og denne kan *ikke* endres, d.v.s. *tilstanden er fast og kan bare leses*.

Deklarasjon:

```
String <strengNavn>;
```

- deklarasjonen oppretter en *referanse* for et `String` objekt.

`String str;` **fører til opprettelse av en referanse for et** String**-objekt:**



## Strenger (forts.)

Opprettelse:

- Enkleste måte å opprette et `String`-objekt på, er å bruke en strenglitteral.

```
String <strNavn> = <strenglitteral>;

String adr = "Velkommen til Portveien 2!";
String farge = "grønn";
```

- `String`-objektet kan opprettes ved å anvende `new`-operatoren sammen med et *konstruktørkall*:

```
<strNavn> = new String(<parameter-liste>);

//oppretter et nytt String-objekt fra strenglitteral.
String adr = new String("Velkommen til Portveien 2!");
```

- Deklarasjon kan kombineres med opprettelsen:

```
String <strNavn> = new String(<parameter-liste>);

//Oppretter et nytt String-objekt som ikke har noen tegn, og lengde 0:
String navn = new String();
//Oppretter et nytt String-objekt fra String-objekt adr:
String nyAdr = new String(adr);
```



## To viktige `String`-metoder

```
String kurs = "databehandling";
// kurs inneholder referanseverdien til et anonymt String-objekt
```

- Hvert `String`-objekt har en instansmetode, `length()`, som returnerer antall tegn i strengen.
  - Metodekallet `kurs.length()` returnerer antall tegn i `String`-objektet `kurs`, d.v.s. `14`.
- Metoden `charAt(int i)` returnerer tegn gitt ved indeks `i` i strengen.
  - Metodekallet `kurs.charAt(4)` returnerer tegn `'b'` i `String`-objektet `kurs`.
  - NB! Startindeks er `0` og sluttindeks er `kurs.length()-1`.
  - Indeksverdien må kontrolleres, ellers får vi `StringIndexOutOfBoundsException`-kjørefeil.

```
String kurs = "databehandling";
char tegn = 'a';
int antall = 0, indeks = 0;
while (indeks < kurs.length()) {
    if (kurs.charAt(indeks) == tegn) antall = antall + 1;
    indeks = indeks +1;
}
System.out.println("Antall ganger tegn forekommer: " + antall);
```

## Strengsammenslåing: +

```
String kurs = "databehandling";
kurs = "Grunnkurs i " + kurs;
// Konkatenering på høyre side gir et nytt String-objekt
// med en ny verdi ("Grunnkurs i databehandling").
// kurs får referanseverdien til dette nye objektet.
```

- Dersom en av operandene til den binære operatoreren, +, er et `String`-objekt, blir det laget en *strengrepresentasjon* av den andre operanden, og resultatet blir et `String`-objekt:

```
int nummer = 110;
String kurs1 = "I" + nummer + ": " + kurs; // "I110: Grunnkurs i databehandling"
String kurs2 = 'I' + nummer + ": " + kurs; // "183: Grunnkurs i databehandling"

'I' + nummer + ": " + kurs
(73 + 110)   + ": " + kurs
183          + ": " + kurs
"183"        + ": " + kurs
"183: "             + kurs
"183: "             + "Grunnkurs i databehandling"
"183: Grunnkurs i databehandling"
```



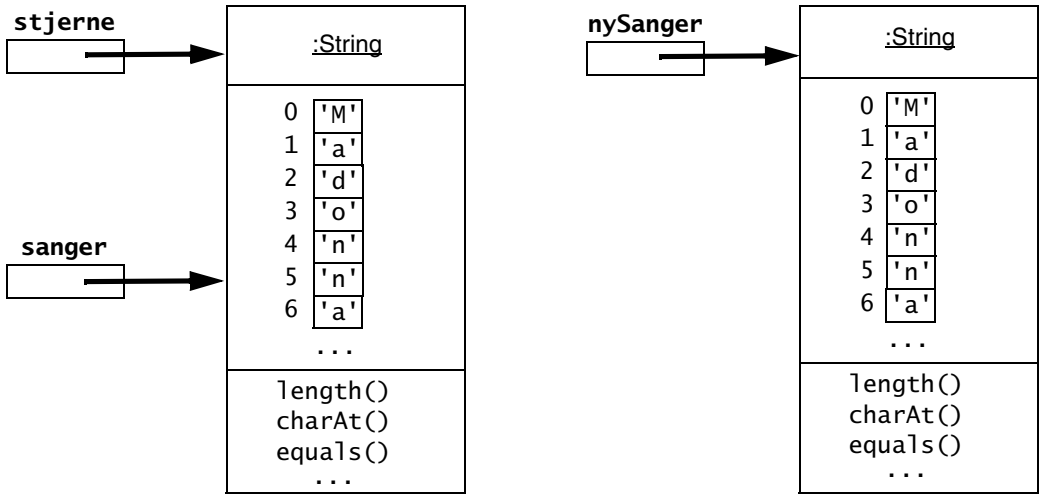
## Strengsammenligning

- Sammenligning av strenger er basert på *leksikografisk ordning*: tegn sammenlignes basert på deres UNICODE-verdier (jfr. telefonkatalog).
  - `'a' < 'b'` er sann siden UNICODE-verdien `\u0061` til tegn `a` er mindre enn UNICODE-verdien `\u0062` til tegn `b`.
  - Strengen `"abba"` er mindre enn strengen `"aha"` siden 2. tegn, `'b'`, i strengen `"abba"` er mindre enn 2. tegn, `'h'`, i strengen `"aha"`.
    (`"abba"` vil komme før `"aha"` i telefonkatalogen.)
- Likhetsoperatoren, ==, sammenligner to *referanser* for å avgjøre om de peker på samme objektet, dvs. avgjør om to referanser er *aliaser*.
- Metoden `equals()` sammenligner *innholdet* i to `String`-objekter for likhet.

# String-objekter

**Deklarasjon og opprettelse:**

```
String stjerne  = "Madonna";
String sanger   = "Madonna";
String nySanger = new String("Madonna");
```




**Referansesammenligning:**

```
(stjerne == sanger) er true.
(stjerne == nySanger) er false.
```

**Inneholdssammenligning:**

```
(stjerne.equals(sanger)) er true.
(stjerne.equals(nySanger)) er true.
```



# Andre String-metoder (Program 3.3, s. 63)

- Java tilbyr metoder for *opprettelse* av nye `String`-objekter, *sammenligning* av `String`-objekter, *søking* i `String`-objekter, og *konvertering* av `String`-objekter.

| | |
|---|---|
| `int compareTo(Object s2)` | Sammenlign en streng med en annen streng.<br>`s1.compareTo(s2)` returnerer en verdi som er `==0` eller `>0` eller `<0` som hhv indikerer at `s1` er lik `s2`, eller `s1` er større enn `s2`, eller `s1` er mindre enn `s2` leksikografisk. |
| `static String valueOf(T t)` | Returnerer en strengrepresentasjon av verdien i parameteren `t` av type `T`.<br>Dersom `i` har verdi `99`, vil `String.valueOf(i)` returnere `"99"`. |
| `int indexOf(int tegnkode)`<br>`int indexOf(String delstreng)`<br>`int indexOf(int tegnkode, int startIndeks)`<br>`int indexOf(String delstreng, int startIndeks)` | Finn indeks til et tegn eller *delstreng* i den aktuelle strengen.<br>Returnerer -1 dersom parameterverdien ikke finnes.<br><br>`int førstIndeks = s1.indexOf('a');`<br>`int andreIndeks = s1.indexOf('a', førstIndeks+1);` |

| | |
|---|---|
| `String substring(`<br>`  int startindeks,`<br>`  int sluttindeks)` | Opprett en ny streng som er en substreng av denne strengen:<br>`str.substring(0,4)` returnerer `str[0..3]`.<br>Merk sluttindeksen! |



## Primitive verdier som objekter

| Primitiv datatype | Tilsvarende wrapper-klasse |
|---|---|
| `boolean` | `Boolean` |
| `char` | `Character` |
| `byte` | `Byte` |
| `short` | `Short` |
| `int` | `Integer` |
| `long` | `Long` |
| `float` | `Float` |
| `double` | `Double` |

```
Integer iRef = 10;  // (1) Automatisk innkapsling, verdi 10 blir innkapslet.
int j = iRef;       // (2) Automatisk utpakking, j får verdi 10.

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

- Enklere å bruke automatisk innkapsling/utpakking

# for-løkke

- Den generelle løkkekonstruksjonen som er mest brukt for *teller-kontrollerte* løkker hvor antall repetisjoner (*iterasjoner*) er kjent på forhånd.

Syntaks:

```
for (<initialisering>; <løkkebetingelse>; <oppdatering>) {
    /* løkkekropp */
}
```

- `<initialisering>`: uttrykk for initialisering av *løkkevariabel.*
- `<løkkebetingelse>`: uttrykk for å avgjøre om løkkekroppen skal utføres, og der løkkevariabelen inngår.
- `<oppdatering>`: uttrykk som modifiserer løkkevariabelen.

Semantikk:

*<initialisering>;*

***gjenta mens*** *<løkkebetingelse> er sann:*

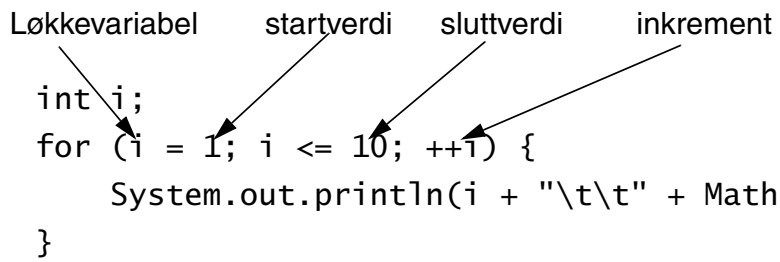
*<løkkekropp>*

*<oppdatering>*



# Eksempel: for-løkke

*Skriv ut kvadratroten av tall fra 1 til 10.*




```
int i;
for (i = 1; i <= 10; ++i) {
    System.out.println(i + "\t\t" + Math.sqrt(i));
}
```

## Definisjonsområde til løkkevariabelen i `for`-løkke

- *<initialisering>* og løkkekroppen kan inneholde deklarasjoner av variabler.
- Disse variablene kan bare *brukes* i `for`-løkken.
- Løkkevariabelen kan deklareres i *<initialisering>*:

```
// Skriv ut kvadratroten av tall fra 1 til 10.
for (int i = 1; i <= 10; ++i) {
    System.out.println(i + "\t\t" + Math.sqrt(i));
}
// løkkevariabelen i ikke tilgjengelig etter at løkken er ferdig.
```



## Fremgangsmåten ved skriving av et program

**1. Problembeskrivelse (tlf.nr. til ord)**

*Lag et program som leser inn et telefonnummer og skriver det ut som 8 ord. Når bruker f.eks. skriver:*

```
55584152
```

*skal programmet skrive på skjermen*

```
fem fem fem åtte fire en fem to
```

**2. Problemanalyse og presisering**

- Hvordan skal telefonnummeret angis?

  Formatet på data: `55-58-41-52` eller `55584152`. (Vi velger den siste måten.)
- Vi leser telefonnummeret som en streng, konverterer hvert tegn (som er et siffer) til ord, og bygger opp en resultatstreng med ord.

**3. Utforming:**

**Algoritme**

1. *Initialiser variabler*
2. *les telefonnummeret som en streng*
3. ***gjenta mens** ikke alle siffer i strengen er behandlet:*
   - *les et siffer fra strengen*
   - *konverter sifferet til ord, og oppdater resultatstrengen*
4. *Skriv ut resultatstrengen med ordene.*

*Steg 3 utgjør en løkke.*

**Data**

*En variabel,* `innlestStreng`*, til å lese inn telefonnummeret.*

*En variabel,* `tegn`*, til å holde rede på et siffer.*

*En variabel,* `tlfStreng`*, for å lagre ordene.*

**4. Koding: se neste side.**



```
import java.util.Scanner;
//Program oversetter tlf.nr. til ord
public class Telefon {
  public static void main(String[] args) {
    Scanner tastatur = new Scanner(System.in);
    System.out.print("Skriv telefonnummer (som 55584152):");
    String innlestStreng = tastatur.next();
    String tlfStreng = "";
    for (int teller = 0; teller < innlestStreng.length(); teller++) {
      char 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 += "syv "; }
      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):55584152
fem fem fem åtte fire en fem to
```

- Øvelse: Modifiser programmet slik at antall siffer i telefonnummeret blir kontrollert.



# Mer om `for`-løkke

- En `for`-løkke kan erstatte en `while`-løkke.

```
Scanner tastatur = new Scanner(System.in);

int tall = tastatur.nextInt();
int sum = 0;
while (tall != 0) {
    sum += tall;
    tall = tastatur.nextInt();
}
```

(1) Med `for`-løkke:

```
int tall = tastatur.nextInt();
int sum = 0;
for ( ; tall != 0; ) {  // Begge semikolonene må angis.
    sum += tall;
    tall = tastatur.nextInt();
}
```

(2) Logisk feil:

```
int tall = tastatur.nextInt();
for (int sum = 0; tall != 0; sum += tall) {
    tall = tastatur.nextInt();
}
```

- En `for`-løkke kan ikke erstatte en `do-while`-løkke fordi testing kommer etter løkkekroppen.
- Dersom alle uttrykk utelates i `for`-løkken, får vi en "evig løkke":

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

  – Evig løkke kan termineres med en `return`- eller en `break`-setning i løkkekroppen (forklart senere).



# Nøstede løkker

*Eksempel: Skriv ut gangetabeller fra 1 til 10.*

```
for (int i = 1; i <= 10 ; ++i) {
    System.out.println("Gangetabell for " + i);
    for (int j = 1; j <= 10 ; ++j) {
        System.out.println(i + "\tx\t" + j + "\ter\t" + (i*j)); // (1)
    }
}
```

- Hvor mange ganger blir linje (1) utført i koden overfor?

# "Baklengs" `for`-løkke

*Eksempel: Skriv ut kvadratroten av tall fra 10 til 1.*

```
for (int i = 10; i > 0 ; i--) {
    System.out.println(i + "\t\t" + Math.sqrt(i));
}
```

## Eksempler på løkker

- En-for-mye eller en-for-lite feil:
  (The banana problem: "I know how to spell banana. I just don't know when to stop.")
```
resultat = 0;
for (int teller = 2; teller < antall ; teller += 2) {
    resultat += teller;
} // dersom antall er partall, vil ikke dens verdi bli summert.
```
- Riktig initialisering av løkkevariabler:
```
resultat = 0;
do {
    resultat *= 2;
} while (resultat < 100); // evig løkke
```
- Kontroller at betingelsen er riktig:
```
// Eks. 1
resultat = 0;
for (int teller = 1; teller != 100 ; teller += 2) {
    resultat += teller;
} // evig løkke
```



```
// Eks. 2
int sum = 0;
for (int i = 1; i < 1 ; i++) { // testen er usann ved inngangen
    sum += i;
}
```
- Løkke-invarianter kan flyttes ut av løkkekroppen: løkke-effektivisering
```
// alt. 1
double sum = 0.0, pi = 3.14;
for (int i = 0; i < 100 ; ++i) {
    sum += Math.pow(pi * 2.0, i);// (pi*2) endrer seg ikke i løkken,
                                 // og kan flyttes ut.
}
// alt. 2
double sum = 0.0, pi = 3.14;
double inv = pi * 2.0;
for (int i = 0; i < 100 ; ++i) { sum += Math.pow(inv, i); }
```

*"Håndsimuler" alltid 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.*

## Valg av riktig løkke

- En `for`-løkke brukes vanligvis for matematiske beregninger som summering av endelige rekker.
- En `while`-løkke brukes vanligvis for situasjoner der løkkebetingelsen kan blir usann når som helst, dvs for *ikke-teller* repetisjoner.
- En `do-while`-løkke brukes i samme situasjoner som der `while`-løkke brukes, men når løkkekroppen må utføres minst en gang.

## Valg av løkker

- *Bak en kake for hver av dine 6 venner.*
- *Stå ved enden av bowling-banen og plukk alle kjegler, en om gangen, som er blitt slått over ende.*
- *Let gjennom kortstokken etter spar-dame.*
- *Still et spørsmål. Forsett å spørre dersom svaret er feil.*



## `break`-setning i løkker

- Utføring av en `break`-setning i en løkkekropp flytter programkontrollen ut av løkken.
  - Utføringen fortsetter etter løkken.
- Vanlig bruk av `break`-setning er for tidlig terminering av en løkke.

```
for (int i = 1; i <= 10; ++i) {
    if (i == 6) break; // terminerer løkken når i får verdi 6
    System.out.println(i + "\t\t" + Math.sqrt(i));
}
// Fortsett her.
```

## `continue`-setning i løkker

- Utføring av en `continue`-setning i en løkkekropp fører til at resten av løkkekroppen blir hoppet over, og utføringen fortsetter med neste iterasjon av løkken.
- I en `while`- eller en `do-while`-løkke, blir løkkebetingelsen testet umiddelbart etter at `continue`-setningen er utført.
- I en `for`-løkke, etter at `continue`-setningen er utført, utføres først oppdateringen av løkkevariabelen, og så blir løkkebetingelsen testet.

```
for (int i = 1; i <= 10; ++i) {
    if (i == 6) continue; // resten av løkkekroppen ikke
                          // utført når i får verdi 6.
    System.out.println(i + "\t\t" + Math.sqrt(i));
}
```



## Mer om valgsetning: `switch`-setning

- `switch`-setningen kan brukes til å velge én blant flere alternative handlinger, basert på verdien til et uttrykk (se figur 3.12, s. 75).

Syntaks:

```
switch (switch-uttrykk) {
    case verdi1:
        // handlingsliste
        break;
    case verdi2:
        // handlingsliste
        break;
    ...
    case verdiN:
        // handlingsliste
        break;
    default:
        // default handlingsliste
} // end switch
```

## `switch`-setning (forts.)

Semantikk:

- Sammenlign *`switch-uttrykk`*-verdien med én `case`-etikett (*label*) om gangen.
  - dersom *`switch-uttrykk`*-verdien er *lik* en `case`-etikett, utfør tilsvarende *`handlingsliste`*.
  - dersom ingen `case`-etikett er lik *`switch-uttrykk`*-verdien, utfør *`handlingsliste`* for `default`-etiketten.
- Den `default`-etiketten er valgfri.
- Dersom ingen *verdi* er lik *`switch-uttrykk`*-verdien, og `default`-etiketten mangler, blir ingen handling utført, og utføringen fortsetter etter `switch`-setningen.
- Etiketten må være en *entydig litteral* (heltall, tegn eller streng – ikke boolsk eller flyttall).
- En `break`-setningen brukes for å avslutte en *`handlingsliste`* (som kan være én eller flere setninger).
  - utføringen av `break`-setningen i en `switch`-setning flytter programkontrollen til setningen etter `switch`-setningen.
  - uten `break`-setningen vil utføringen fortsette i etterfølgende *`handlingsliste`*.
- Det er ikke nødvendig med blokknotasjon for *`handlingsliste`*.



- Rekkefølgen på `case`-etikettene og `default`-etiketten er likegyldig, så lenge hver *`handlingsliste`* avsluttes med en `break`-setning.
- *`switch-uttrykk`*-verdien og `case`-etiketten testes for *likhet*, mens betingelsen i en `if-else`-setning tillater et *vilkårlig sannhetsuttrykk*.

```
import java.util.Scanner;
//Programmet oversetter tlf.nr. til ord: switch-versjon
public class TelefonII {
  public static void main(String[] args) {
    Scanner tastatur = new Scanner(System.in);
    System.out.print("Skriv telefonnummer (som 55584152):");
    String innlestStreng = tastatur.next();
    String tlfStreng = "";
    for (int teller = 0; teller < innlestStreng.length(); teller++) {
      char tegn = innlestStreng.charAt(teller);
      switch (tegn) {
        case '1': tlfStreng += "en ";   break; // hopp ut av switch-setningen
        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);
```



```
  }
}
```

- Hvordan er programmet ovenfor forskjellig fra versjonen med `if-else`-setningen?

## `switch`-setning: Felles handlinger for `case`-etiketter

```
import java.util.Scanner;
//Programmet beregner årstid for en gitt måned
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 etiketter.
        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:   // Vurder bruk av en påstand: assert-setning.
        System.out.println(tall + " er ikke en lovlig måned!!!");
        break; // strengt tatt ikke nødvendig
    } // end switch
```



```
  }
}
```

Kjøring av programmet:

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

## switch-setning: Streng litteral som case-etikketter

```
import java.util.Scanner;
//Programmet beregner årstid for en gitt månedsnavn.
public class Aarstid2 {
  public static void main(String[] args) {
    Scanner tastatur = new Scanner(System.in);
    System.out.print("Tast månedsnavn: ");
    String navn = tastatur.next().toLowerCase(); // Konverter til småbokstaver.
    switch (navn) {
      case "desember": case "januar": case "februar":
        System.out.println("Det er vinter i " + navn + "!");
        break;
      case "mars": case "april": case "mai":
        System.out.println("Det er vår i " + navn + "!");
        break;
      case "juni": case "juli": case "august":
        System.out.println("Det er sommer i " + navn + "!");
        break;
      case "september": case "oktober": case "november":
        System.out.println("Det er høst i " + navn + "!");
        break;
      default:   // Vurder bruk av en påstand: assert-setning.
        System.out.println(navn + " er ikke en lovlig måned!!!");
        break;   // strengt tatt ikke nødvendig
    } // end switch
```



```
  }
}
```

Kjøring av programmet:

```
Tast månedsnavn: Mai
Det er vår i mai!
```

## Tabeller

- Tabeller (*arrays*) kan brukes til å lage en *samling* av dataverdier.
  - En tabell består av et *endelig* antall *elementer*.
  - Elementer i en tabell kan være én av følgende:
    - enten *verdier* av en bestemt primitiv datatype,
    - eller *referanser* til objekter.
  - Alle elementer i en tabell har *samme elementtype*.
  - En tabell har en fast *lengde* som tilsvarer antall elementer i den, og kan ikke endres etter at tabellen er opprettet.
- I Java er tabeller ekte *objekter*, d.v.s. vi bruker en *referanse* til å behandle en tabell.
  - Java har språkkonstruksjoner for deklarasjon, opprettelse, initialisering og bruk av tabeller.



## Tabeller (forts.)

Deklarasjon:

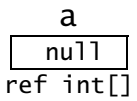
```
<elementType> <tabellNavn> [];
```

eller

```
<elementType>[] <tabellNavn>;
```

- `<elementType>` kan være en *primitiv datatype* eller en *referansetype* (f.eks. en klasse).
- deklarasjonen alene oppretter en *referanse* for tabellen.

`int[] a;` fører til opprettelse av en referanse for en *tabell av int*:



## Tabeller (forts.)

Opprettelse:

- selve tabellen opprettes ved å anvende `new`-operatoren.
- kan kombinere deklarasjon med opprettelsen:
  `<elementType>[] <tabellNavn> = new <elementType>[<antallElement>];`

```
double[] d = new double[5]; // oppretter et tabell-objekt med 5
                            // elementer av type double.
```

- kan opprette elementene dersom tabellreferansen er deklarert fra før:
  `<tabellNavn> = new <elementType>[<antallElement>];`

```
a = new int[8];             // oppretter et tabell-objekt med 8
                            // elementer av type int
```

- Elementene blir *alltid* initialisert til *standardverdier* for `elementType` ved opprettelsen.
- Hver tabell har et felt, `length`, som angir antall elementer i tabellen.
  – Verdi til `a.length` er lik antall elementer i tabellen `a`, dvs `8`.



## Tabeller (forts.)

Bruk:

`<tabellNavn> [<indeks>]`

- `indeks` er et heltallsuttrykk, og oppfyller følgende relasjon:
  `0 ≤ indeks < antallElement`
  – `0` og `(antallElement-1)` er h.h.v. nedre og øvre grenser for `indeks`.
  – Indeks kontrolleres automatisk under utføring.
  – Dersom `indeks ≥ antallElement` eller `indeks < 0`, fører det til `ArrayIndexOutOfBoundsException`-kjørefeil.
- Notasjonen ovenfor tolkes som en variabel:
  – `a[2]` angir 3. element i tabellen `a`.

```
a[2] = 5;
a[2] = 3 * a[2];            // a[2] får verdi 15.
```

# Eksempel på en tabell

```
// deklarasjon og opprettelse
int[] kommode = new int[10];
// indekserte variabler: kommode[0], kommode[1], ... ,
//                       kommode[9];
// Hver indeksert variabel tilsvarer en kommodeskuff.
// int kommode0, kommode1, kommode2, kommode3, kommode4,
//     kommode5, kommode6, kommode7, kommode8, kommode9;

// eksplisitt initialisering
kommode[0] = 29;
```

*indeksert variabel*

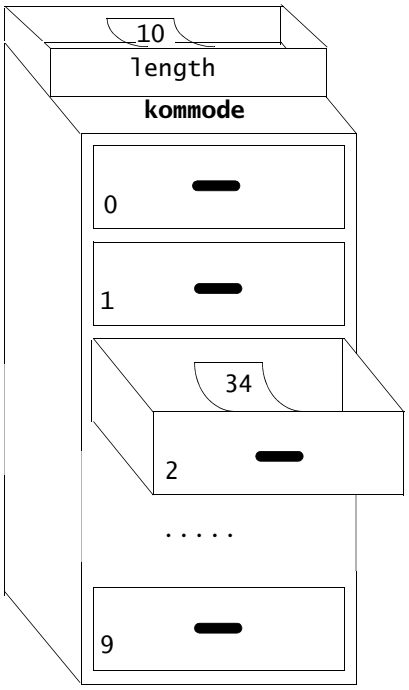
*indeks*

```
// bruk
kommode[2] = kommode[0]+5;
```

*tabell navn*

```
...
for (int i = 0; i < kommode.length; i++)
    kommode[i] = 1;

// kommode[10] finnes ikke!
```

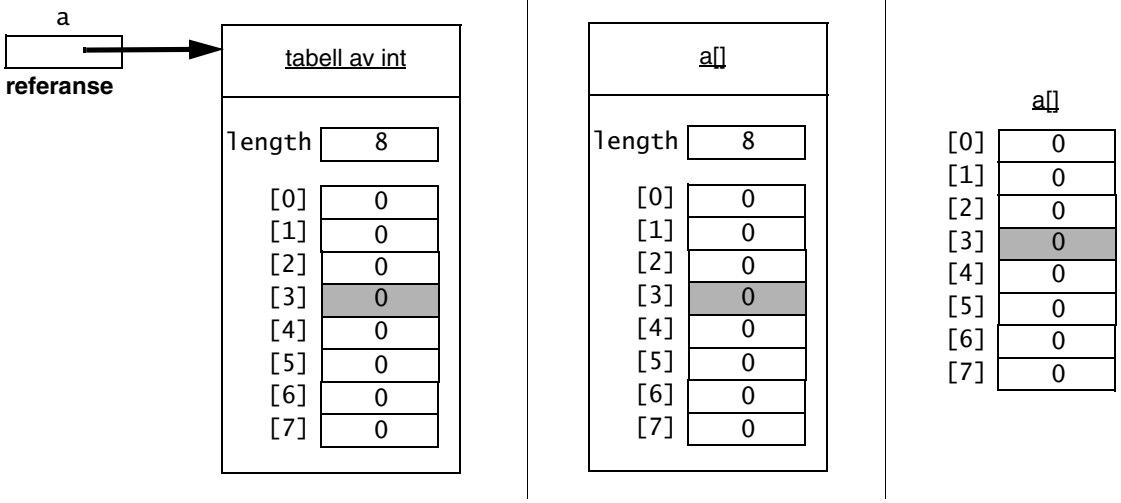





# Tabeller: grafisk notasjon

**Deklarasjon, opprettelse og standardverdi-initialisering:**

```
int[] a = new int[8];
```

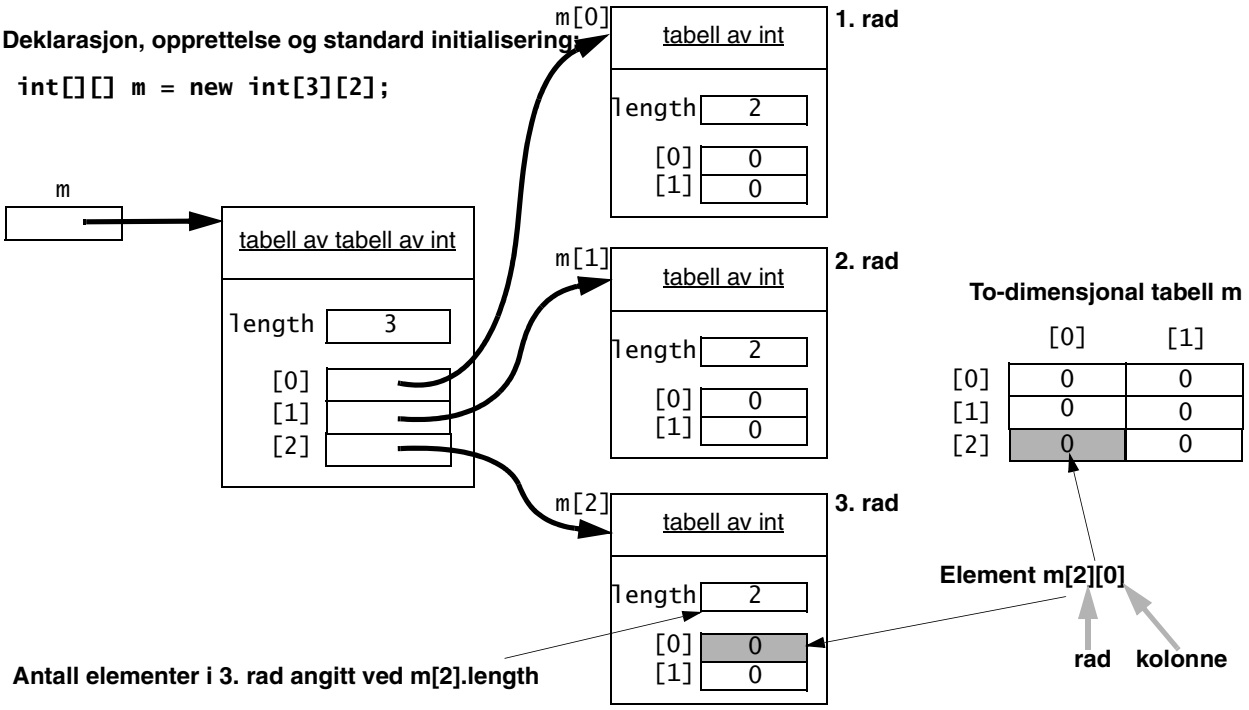



**a[3] er 4. element i tabellen.**

# Fler-dimensjonale tabeller: *tabeller av tabeller*

**Deklarasjon, opprettelse og standard initialisering:**

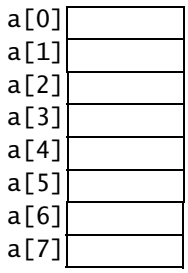
```
int[][] m = new int[3][2];
```





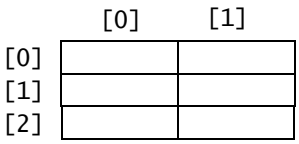

# Lagring av tabell-elementer

En-dimensjonal tabell a

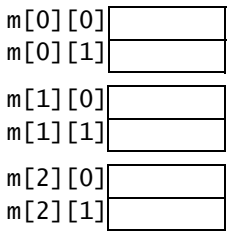



Elementene til tabell a er lagret *sekvensielt*.

To-dimensjonal tabell `m`




er lagret som




Elementene til tabell `m` er lagret *rekkevis*.

## Programmønstre – noen enkle algoritmer

- Finn minste verdi i en tabell:

```
int[] tabell = new int[N_ELEMENT];
// Anta at tabell er initialisert med "ekte" verdier.
int minste = tabell[0];
for (int teller = 1; teller < tabell.length; ++teller) {
    if (tabell[teller] < minste) {
        minste = tabell[teller];
    }
}
```



- Finn minste verdi i en 2-dimensjonal tabell:

```
int[][] matrise = new int[N_ELEMENT][M_ELEMENT]; // N x M matrise
// Anta at matrise er initialisert med "ekte" verdier.
int minste = matrise[0][0];
for (int teller1 = 0; teller1 < matrise.length; ++teller1) {
    // Finn minste i matrise[teller1] ;
    for (int teller2 = 0; teller2 < matrise[teller1].length; ++teller2) {
        if (matrise[teller1][teller2] < minste) {
            minste = matrise[teller1][teller2];
        }
    }
}
```

## Mer om initialisering av tabeller

- Eksplisitt initialisering i deklarasjon for én-dimensjonale tabeller:

```
// heltalls-tabell med 8 elementer
int[] tabell = {1, 3, 49, 55, 58, 41, 52, 3146}; // sammen med deklarasjon.
tabell[] = {1, 3, 49, 55, 58, 41, 52, 3146};     // Ulovlig.
tabell = {1, 3, 49, 55, 58, 41, 52, 3146};       // Ulovlig.
```

- Eksplisitt initialisering i deklarasjon for fler-dimensjonale tabeller:

```
double identitetsMatrise[][] = {
    {1.0, 0.0, 0.0, 0.0 }, // 1. rad initialiseres
    {0.0, 1.0, 0.0, 0.0 }, // 2. rad initialiseres
    {0.0, 0.0, 1.0, 0.0 }, // 3. rad initialiseres
    {0.0, 0.0, 0.0, 1.0 }  // 4. rad initialiseres
}; // 4 x 4 flyttallsmatrise
String[] kjæledyr = {"krokodiller", "elefanter", "krokofanter", "elediller"};
// kjæledyr er en tabell med 4 strenger
char[] tegnTabell = {'a', 'h', 'a'}; // tabell med 3 tegn
```

- Tabeller i en fler-dimensjonal tabell kan ha forskjellige lengder:

```
double[][] matrise = new double[4][]; // indeks lengst til venstre alltid angitt
for (int teller1 = 0; teller1 < matrise.length; teller1++) {
    matrise[teller1] = new double[teller1+1];
}
```



## Histogram

*Problem*: Programmet skal lese flyttall (som angir hva barna veier), dele dem i vektgrupper og skrive ut et histogram.

```
Angi vektene (0-199). En vekt pr. linje. Avslutt med -1.
11,2
13
12,3
82,4
12,9
...
-1
Vekt : Hyppighet
...
10   : ***
11   : *
12   : **
...
82   : *
83   :
...
199  : *****
```

**Datastruktur:**

- En tabell brukes til å telle vektene.
- Indeks i tabellen utgjør en vektgruppe.
- Verdien til et element angir antall barn registrert i denne vektgruppen, dvs. hyppigheten.

**Algoritme:**

- *Les vektene og plasser dem i riktige vektgrupper.*
- *Finn minste og største vekt registrert.*
- *For hver vektgruppe (fra minste til største vekt) skriv antall stjerner som tilvarer antall registreringer*

# Histogram (forts.)

```
import java.util.Scanner;
public class Histogram {
  public static void main(String[] args) {
    int N_ELEMENT = 200;
    int[] histTabell = new int[N_ELEMENT];
    Scanner tastatur = new Scanner(System.in);

    // Les vektene
    System.out.println("Angi vektene (0-199). En vekt pr. linje. Avslutt med -1.");
    int vekt = (int) Math.round(tastatur.nextDouble()); // Finn vektgruppe.
    while (vekt >= 0 && vekt < histTabell.length) {
      histTabell[vekt]++; // Registrering av vektgruppen.
      vekt = (int) Math.round(tastatur.nextDouble());
    }

    // Finner indeksen til den minste vekten som er registrert.
    int minste;
    for (minste = 0;
         (minste < histTabell.length) && (histTabell[minste] == 0);
         minste++);

    // og indeksen til den største vekten som er registrert.
    int største;
```



```
    for (største = histTabell.length -1;
         (største >= 0) && (histTabell[største] == 0);
         største--);

    // Skriv histogram
    System.out.println("Vekt\t:\tHyppighet");
    for (vekt = minste; vekt <= største; vekt++) {
      System.out.print(vekt + "\t:\t");

      for (int stjerne = 1; stjerne <= histTabell[vekt]; stjerne++) {
        System.out.print("*");
      }

      System.out.println();
    }
  }
}
```

## Den forenklete `for`-løkken

- Dersom vi bare er interessert i å *lese* alle verdier i en samling i tur og orden, kan vi bruke den forenklete `for`-løkken.
- Den kan *ikke* brukes til å endre verdiene i en samling.

*Syntaks:*

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

*Eksempel:*

```
Integer[] alderTabell = { 12, 65, 45, 60, 70, 45 };
int antallOver60 = 0;
for (int alder : alderTabell) {
  if (alder >= 60) {
    antallOver60++;
  }
}
System.out.println("Antall over 60: " + antallOver60);
```



## Pseudo-slumptallgenerator

- For å generere tilfeldig tall, kan vi bruke klassen `java.util.Random`.
- Slike tall kalles pseudo-slumptall, siden de er ikke helt tilfeldige.

*Saksgangen:*

```
Random generator = new Random(); // Opprett et objekt av klassen Random
```

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

```
int nyttTall = generator.nextInt(); // tilfeldig heltall i intervallet
                                    // [-2^31, 2^31-1]
```

- Vi kan spesifisere et argument *n* til metoden `nextInt()` for å returnere et heltall i intervallet [0, *n*-1].

```
nyttTall = generator.nextInt(11);      // tilfeldig heltall i [0, 10]
nyttTall = 2 + generator.nextInt(11);  // tilfeldig heltall i [2, 12]
nyttTall = 2 + 3*generator.nextInt(5); // tilfeldig heltall i
                                       // {2, 5, 8, 11, 14}
```

## Eksempel: pseudo-slumptallgenerator

```
//Simulerer terningkast ved hjelp av pseudo-slumptall
//og teller opp antall kast av hver verdi i intervallet [1,6]
import java.util.Random;
import java.util.Scanner;
public class TerningKastHyppighet2 {
  public static void main(String[] args) {
    int[] hyppighet = new int[7]; // Vi ignorerer element hyppighet[0].

    Scanner tastatur = new Scanner(System.in);
    System.out.print("Angi antall ganger terningen skal kastes: ");
    int antall = tastatur.nextInt();
    Random generator = new Random();  // Pseudo-slumptallgenerator
    for (int i = 1; i <= antall; i++) {
      // Kast terning. Gir et tilfeldig heltall i [1,6]
      int terningkastVerdi = 1 + generator.nextInt(6);
      // Øk hyppighetstelleren for verdien til dette kastet.
      hyppighet[terningkastVerdi]++;
    }
```



```
    for (int i = 1; i < hyppighet.length; i++) {
      System.out.printf("Antall %1d-ere: %4d, sannsynlighet: %.3f%n",
          i, hyppighet[i], ((double)hyppighet[i]/antall));
    }
  }
}
```

Kjøring av programmet:

```
Angi antall ganger terningen skal kastes: 200000
Antall 1-ere: 33510, sannsynlighet: 0,168
Antall 2-ere: 33414, sannsynlighet: 0,167
Antall 3-ere: 33428, sannsynlighet: 0,167
Antall 4-ere: 33115, sannsynlighet: 0,166
Antall 5-ere: 33111, sannsynlighet: 0,166
Antall 6-ere: 33422, sannsynlighet: 0,167
```