Kapittel 4:
Utforming av egne klasser

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

Klassedeklarasjon

- I Java består en klassedeklarasjon av en liste med *variabeldeklarasjoner* og *metodedeklarasjoner*.

```
// vanlig oppsett
<klassehode> {
    <variabeldeklarasjoner>
    ...
    <metodedeklarasjoner>
} // rekkefølgen til deklarasjoner er likegyldig.
```

- En klassedeklarasjon identifiseres med et `<klassehode>` som inneholder nøkkelordet `class` og navnet på klassen.

```
class Lys {
    ...
}
```


Utforming av klassen Lys

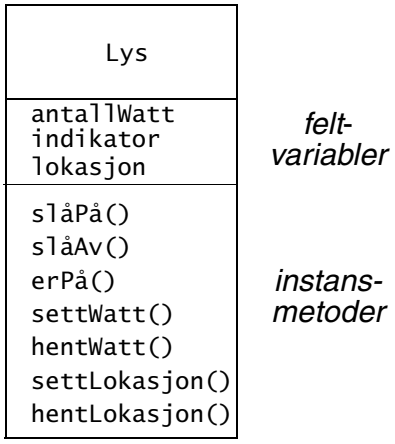
Spesifikasjon av en klasse

Klassenavn:

`Lys`

Feltvariabler:	**Metoder:**
`antallWatt`	`slåPå()`
`indikator`	`slåAv()`
`lokasjon`	`erPå()`
	`settWatt()`
	`hentWatt()`
	`settLokasjon()`
	`hentLokasjon()`

Grafisk notasjon for en klasse



Deklarasjon av egenskaper: feltvariabler

```
/**
 * Klassen Lys modellerer abstraksjonen Lys.
 */
class Lys {
    int antallWatt;    // lysstyrken
    boolean indikator; // av == false, på == true
    String lokasjon;   // hvor lys er plassert
    ...
}
```

- Klassedeklarasjonen angir nå 3 *feltdeklarasjoner* som deklarerer 3 *feltvariabler*.
  - Feltene kan lagre informasjon om lysstyrken (`antallWatt`), om lyset er av eller på (`indikator`), og hvor det er plassert (`lokasjon`).

Objekter

- En klasse må *instansieres* for å opprette et objekt.
- Instansiering (dvs. å opprette objekter) består av:
  - opprettelse av objektet vha. `new`-operatoren og *konstruktørkall*.
  - initialisering av objektets *tilstand* vha. en konstruktør.
- Det er ofte hensiktsmessig å deklarere en *referanse* for å lagre *referanseverdien* til objektet.
  - Referansen kan nå brukes til å manipulere objektet.

Referansedeklarasjon

Deklarasjon:

```
<klasseNavn> <objektReferanse>;
```

- `<klasseNavn>` er navnet på klassen som skal instansieres.
- deklarasjonen alene oppretter en *referanse* for et objekt av denne klassen.

`Lys stueLys;` fører til opprettelse av en referanse for et objekt av klassen Lys:

navn: `stueLys`
type : `ref(Lys)`

null

- jfr. *tabeller* og `String`-klassen.

Objektopprettelse

Opprettelse:

- Selve objektet opprettes ved å anvende `new`-operatoren sammen med et *konstruktørkall*.
- Vi kan kombinere deklarasjon med opprettelse:
  `<klasseNavn> <objektReferanse> = new <konstruktørkall>;`
- `new`-operatoren oppretter et objekt av angitt klasse.
  - allokerer minne til objektet.
  - alle objektets feltvariabler blir initialisert til *standardverdier.*
  - før operatoren returnerer referanseverdien til objektet, utføres *konstruktørkallet* som er spesifisert sammen med `new`-operatoren.
- Et objekt *lever* så lenge *den automatiske spilloppsamleren* (*automatic garbage collector*) ikke har slettet det fra minnet.
  – Sletting skjer ikke så lenge programmet fremdeles har referanser som kan aksessere objektet.

Konstruktørkall

- Et konstruktørkall har følgende form:

```
<klasseNavn>(<parameterliste>);
```

  – Konstruktører kan deklareres i klassedeklarasjonen, og de ligner på metoder.
  – Konstruktør uten argumentliste kalles *standardkonstruktør:*

```
<klasseNavn>() {...}
Lys() {...}
```

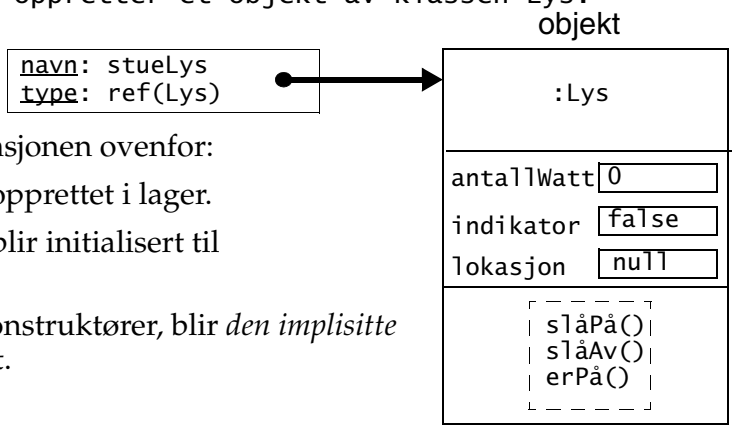
  – Dersom en klasse ikke har noen konstruktører, opprettes *den implisitte standardkonstruktøren* for klassen:

```
<klasseNavn>() {/*tom kropp*/}
Lys() {}
```

  – Den implisitte standardkonstruktøren gjør ingen ting, siden den har tom kropp.

Den implisitte standardkonstruktøren

```
Lys stueLys = new Lys(); // oppretter et objekt av klassen Lys.
```



Saksgangen i utføring av deklarasjonen ovenfor:

- Et objekt av klassen `Lys` blir opprettet i lager.
- Alle feltvariabler til objektet blir initialisert til *standardverdier*.
- Siden klassen `Lys` ikke har konstruktører, blir *den implisitte standardkonstruktøren* benyttet.

Standardverdier (*default values*) for feltvariabler

- Java initialiserer et objekts *feltvariabler* til standardverdier når objektet opprettes.

Feltvariabels datatype:	Standardverdi:
`boolean`	**`false`**
`char`	`'\u0000'`
Heltall (`byte`, `short`, `int`, `long`)	`0`
Flyttall (`float`, `double`)	`+0.0f` eller `+0.0d`
Referanse	`null`

Deklarasjon av atferd: metoder

- Atferd defineres v.h.a. *metoder*.

Syntaks:

```
<adgangsmodifikator> <returtype> <metodenavn> (<formellParameterliste>) {

  /* metodekropp: deklarasjoner og setninger */

}
```

Klassen Lys: metoder

```
/**
 * Klassen Lys modellerer abstraksjonen Lys.
 */
class Lys {
    int antallWatt;    // lysstyrke
    boolean indikator; // av == false, på == true
    String lokasjon;   // hvor lys er plassert
    /** Standardkonstruktør */
    Lys() {
        antallWatt = 0;
        indikator  = false;
        lokasjon   = "X";
    }

    /** Metode for å slå på lys */
    void slåPå() {
        indikator = true;
        System.out.println("Lys i lokasjon " + lokasjon + " er slått på.");
    }
```



```
    /** Metode for å slå av lys */
    void slåAv() {
        indikator = false;
        System.out.println("Lys i lokasjon " + lokasjon + " er slått av.");
    }
    /** Metode for å finne ut om lys er av eller på */
    boolean erPå() {
        return indikator;
    }
    /** Metode for å sette lokasjon til lys */
    void settLokasjon(String lok) {
        lokasjon = lok;
    }
    // Andre metoder ...
}
```

Adgangsmodifikatorer

- <*adgangsmodifikator*> angir *synlighet* til metoden:

`public`	Metoden er tilgjengelig fra alle andre klasser: alle metodene i klassen `Lys` er `public`, og utgjør *klassens kontrakt*.
`private`	Metoden er kun tilgjengelig i klassen den er deklarert i.
ingen adgangsmodifikator (kalt *standard* eller *default* eller *pakkesynlighet*)	Metoden er kun tilgjengelig for klasser i den samme pakken som metoden er deklarert i. (*Mer om dette i det videregående kurset.*)

Returverdi

- <*returtype*> er *datatypen* til verdien som metoden returnerer dersom den blir utført.
  - `void` betyr at metoden ikke returnerer noe verdi.
  - Metoden `slåAv()` returnerer ingen verdi, mens metoden `erPå()` returnerer en boolsk verdi.

Signatur: Metodenavn og formelle parametere

- <*metodenavn*> er navnet på metoden: `slåAv` og `erPå` er metodenavn.
- <*formellParameterliste*> er data som metoden trenger for å gjøre jobben sin.
  - Hverken `slåAv` eller `erPå` har noen parametere, angitt med `()`.
  - Metoden `settLokasjon` har en formell parameter `lok` av type **`String`**, angitt med `(String lok)`.

- <*metodenavn*> og <*formellParameterliste*> utgjør *signaturen* til metoden.
  - Metoden `slåPå` har signaturen `slåPå()`.
  - Metoden `settLokasjon` har signaturen `settLokasjon(String)`.

Metodekropp og `return`-setning

- *Metodekroppen* er en *blokk* som inneholder deklarasjoner av *lokale variabler* som metoden bruker, og *setninger* som angir handlinger som metoden foretar seg.
  - Hverken metodekroppen til `slåAv()` eller `erPå()` inneholder noen deklarasjoner. (*Eksempel med lokale variabeldeklarasjoner kommer senere.*)
  - Handlingen i metoden `slåAv()` er å sette feltvariabelen `indikator` til `false`, og gi melding om at lyset er slått av.
  - Handlingen i metoden `erPå()` er å returnere verdien til feltvariabelen `indikator` (vha. `return`-setningen).
    - `return`-setningen *returnerer* denne verdien til den *kallende* metoden.
    - Dersom metoden `erPå()` ikke har en `return`-setning, vil kompilatoren gi en feilmelding.
    - *Returverdien* må matche *returtype* deklarert i *metodedeklarasjonen*.
    - `return`-setningen kan enten returnere en *verdi av primitiv datatype* eller en *referanseverdi til et objekt*.
    - `return`-setningen avslutter utføring av en metode.
      - `return`-setningen alene (uten en returverdi) kan brukes til å avslutte utføring av en `void`-metode.

Lokale variabler og feltvariabler

- La oss utvide klasse `Lys` med en metode som beregner hva det koster å ha lyset på.

```
class Lys {
    // Feltvariabler
    int antallWatt;       // lysstyrken
    boolean indikator;    // av == false, på == true
    String lokasjon;      // hvor lys er plassert
    // ...
    double kostnad(int antallTimer) {
        // Lokale variabler
        double kWh_pris = 0.35;        // øre pr. kiloWatt-time
        double pris = (antallWatt * antallTimer/ 1000.0)* kWh_pris;
        return pris;
    }
    // ...
}
```

- Metoden `kostnad()` deklarerer *3 lokale variabler*: `kWh_pris`, `pris` og `antallTimer`.
  - NB! *Parametere er også lokale variabler.*

Forskjeller mellom feltvariabler og lokale variabler

- feltvariabler (f.eks. `antallWatt`) er deklarert i klassekroppen, mens lokale variabler (f.eks. `kWh_pris`) er deklarert i metodekroppen.
- feltvariabler kan deklareres med nøkkelordet `private`, mens lokale variabler ikke kan det.
- lokale variabler i en metode er ikke tilgjengelige i andre metoder til klassen, dvs lokale variabler kan bare brukes i metoden de er deklarert i.
- lokale variabler eksisterer så lenge metoden blir utført, mens feltvariabler eksisterer så lenge objektet eksisterer.

```
class Lys {
    double kostnad(int antallTimer) {
        double kWh_pris = 0.35;    // øre pr. kiloWatt-time
        return (antallWatt * antallTimer/ 1000.0)* kWh_pris;
    }
    void settLokasjon(String lok) {
        kWh_pris = 0.5;            // Kompilator klager dersom kWh_pris ikke er
                                   // deklarert.
    }
}
```


Lokal blokk

- Parametervariabler kan *ikke* redeklareres i metodekroppen.
- En lokal variabel i en blokk kan redeklareres dersom blokkene er *disjunkte*.
- En lokal variabel *allerede* deklarert i en blokk kan *ikke* redeklareres i en nøstet blokk.

```
public static void main(String[] args) {
    String args = ""; // kan ikke redeklarere parametere.
    char siffer;
    for (int teller = 0; teller < 10; ++teller) {
        switch (siffer) {
            case 'a': int i; // OK
            default:  int i; // allerede deklarert i denne blokken
        } // switch

    } // for
    int teller; // OK
} // main
```

Metodeklassifisering

- *Mutatorer:*
  - skrive-operasjoner som endrer objektets tilstand.
  - disse må opprettholde dataintegritet for objektets tilstand.
  - deklareres vanligvis som `public`.
  - utgjør en del av klassens kontrakt.
  - F.eks. er metoden `slåPå()` en mutator.
- *Selektorer:*
  - lese-operasjoner som har adgang til objektets tilstand, og som ikke forandrer tilstand.
  - deklareres vanligvis som `public`.
  - utgjør en del av klassens kontrakt.
  - F.eks. er metoden `erPå()` en selektor.
- *Hjelpemetoder:*
  - Operasjoner som brukes av andre metoder i klassen for å implementere atferd.
  - deklareres vanligvis som `private`.
  - utgjør ikke en del av klassens kontrakt, men klassens implementasjon.

Meldinger: Metodekall

Eksempel på en klient av klassen `Lys`:

```
public class Klient {
    public static void main(String[] args) {  // kallende metode
        // ...
        Lys stueLys = new Lys();
        stueLys.slåPå(); // melding vha. metodekall
        // ...
    }
}
```

- `stueLys.slåPå()` er et *metodekall*.
  - `stueLys` er referansen som betegner objektet som skal motta denne meldingen.
  - `'.'` skiller referansen og metodenavnet.
  - `slåPå` er navnet på metoden som må være definert i klassen til objektet.
  - `()` angir *aktuell parameterliste* som angir verdier metoden skal bruke.
  - Et metodekall på et objekt fører til utføring av den tilsvarende metoden i klassen til objektet.
  - Metodekallet `stueLys.slåPå()` returnerer ingen verdi, siden metoden `slåPå()` er en `void`-metode.

Eksempel på en klient av klassen `Lys` (forts.)

```
public class Klient {
    public static void main(String[] args) {
        Lys stueLys = new Lys();
        boolean lysFlagg = stueLys.erPå();
        if (lysFlagg) {
            System.out.println("Stuelys er på.");
        } else {
            System.out.println("Stuelys er av.");
        }
        stueLys.settLokasjon("stuen");
    }}
```

- `stueLys.erPå()` er et annet metodekall.
  - Metodekallet fører til utføring av metoden `erPå()` deklarert i klassen `Lys`.
  - Metodekallet returnerer en verdi av type `boolean`, som kan brukes på lik linje med andre verdier/variabler av denne typen.
- `stueLys.settLokasjon("stuen")` er et metodekall med *én aktuell parameter* `"stuen"`.
  - Den aktuelle parameteren `"stuen"` blir *matchet* med den tilsvarende formelle parameteren `(String lok)` i metodedeklarasjonen til `settLokasjon()`, og brukt til å initialisere feltvariabelen `lokasjon` for objektet `stueLys`.

Metoder uten parametere

Eksempel: Kunder bestiller kebab.

- `Kunder` har ingen muligheter til å påvirke hva slags kebab de får.
- `KebabFabrikk` kan tilby forskjellige metoder for valg av kebab varianter, men dette er ikke den beste løsningen.
- Bruk av *parametere* fører til mer *fleksible og gjenbrukbare* metoder: m.a.o. meldinger kan gjøres mer spesifikke ved tilleggsinformasjon.

Programmet fortsetter rett etter kallet ved retur fra den kalte metoden.

```
class Kebab {
    ...
}

class KebabFabrikk {
    Kebab bestilling() {
    ...
    }
    ...
}

class Kunder {
    ...
    void kjøpKebab () {
        KebabFabrikk kebabHus = new KebabFabrikk();
        Kebab nyKebab = kebabHus.bestilling();
        ...
    }
    ...
}
```

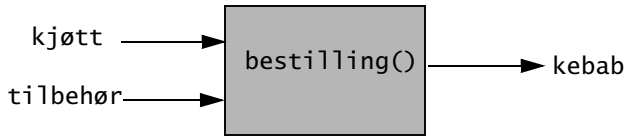
den kalte metoden

den kallende metoden

melding er et kall til en bestemt metode

Parameterangivelse i metodekall: Aktuelle parametere

- Istedenfor å skrive metoder for alle mulige situasjoner, *sender* den kallende metoden *spesifikk informasjon* til *generaliserte metoder* vha. *parametere*.



Metoder som *svarte bokser*: klienter trenger ikke å vite om *innmaten* i boksen, bare hvordan den *brukes*.

Aktuelle parametere angir *aktuelle objekter* som skal brukes i dette kallet.

Hvordan oppfyller `KebabFabrikken` denne bestillingen?

```
class Kunder {
    ...
    void kjøpKebab () {
        KebabFabrikk kebabHus = new KebabFabrikk();

        // opprett aktuelle parametere
        Kjøtt kylling = new Kjøtt("kylling");
        Tilbehør løk = new Tilbehør("løk");
        // bestill kebab
        Kebab kyllingKebab = kebabHus.bestilling ( kylling, løk );
        ...
    }
    ...
}
```

aktuelle parametere

Parameterdeklarasjon: Formelle parametere

- Vi må definere metoden `bestilling` slik at den kan akseptere parameterene.

```
class KebabFabrikk {
    Kebab bestilling ( Kjøtt kjøttValg, Tilbehør tilbehørValg ) {
        // kildekode for å lage kebab
    }
    ...
}
```

formelle parametere

- Formelle parametere er som *plassholdere*, f.eks. `x` og `y` i ligningen $x^2 + y^2 = 1$
- Formelle parametere har ingen verdi før de får *verdier til aktuelle parametere* som blir overført når metoden blir kalt – og verdier kan *variere* fra kall til kall.
- Formelle parameternavn trenger *ikke* å være lik aktuelle parameternavn.
  – de er lokale variabler til metoden.
- Formelle parametere angis med tilhørende *datatype* (f.eks. primitive datatyper eller klasser) - dette gjelder *ikke* for aktuelle parametere.

Parameteroverføring: *referanseverdi for objekter*

```
class Kunder {
    ...
    void kjøpKebab () {
        KebabFabrikk kebabHus = new KebabFabrikk();

        // opprett parametere
        Kjøtt kylling = new Kjøtt("kylling");
        Tilbehør løk = new Tilbehør("løk");
        // bestill kebab
        Kebab kyllingKebab = kebabHus.bestilling(kylling [•] , løk [•] );
        ...
    }
    ...
}

class KebabFabrikk {
    Kebab bestilling(Kjøtt kjøttValg [•] , Tilbehør tilbehørValg [•] ) {
        // kildekode for å lage kebab
    }
    ...
}
```

Formelle parametere refererer til samme objekter som aktuelle parametere, m.a.o. parameteroverføring foregår vha. kopiering av referanseverdier dersom vi har objekter som parametere.

Først overføres referanseverdier, så utføres metoden.

:Kjøtt
...

:Tilbehør
...

Referansene kan brukes til å endre tilstanden til objektene.

Parameteroverføring: *verdier av primitive datatyper* **(I)**

- Formelle parametere tar verdier av *primitive datatyper.*

```
class Kunder {
    ...
    void kjøpKebab () {
        KebabFabrikk kebabHus = new KebabFabrikk();
        ...
        // beregn pris for alle kebaber spist.
        // anta fast pris pr. kebab
        double pris = kebabHus.beregnPris( 4 , 30.50 );
        ...
    }
    ...
}

class KebabFabrikk {
    double beregnPris(int antallKebab 4 ,double prisPrStk 30.5 ) {
        // kode for å beregne prisen
    }
    ...
}
```

Formelle parametere får verdien til aktuelle parametere, m.a.o. parameteroverføring foregår v.h.a. kopiering av primitive verdier dersom vi har formelle parametere av primitive datatyper.

Først initialiseres formelle parametere, så utføres metoden.

Aktuell parameter kan være et vilkårlig uttrykk.

Parameteroverføring: *verdier av primitive datatyper* (II)

- Aktuelle parametere kan være et uttrykk som evaluerer til en *primitiv verdi*.

```
class Kunder {
    ...
    void kjøpKebab () {
        KebabFabrikk kebabHus = new KebabFabrikk();
        ...
        int antall = 4;
        double stkPris = 30.5;
        double pris = kebabHus.beregnPris(antall 4 ,stkPris 30.5 );
        ...
    }
    ...
}

class KebabFabrikk {
    double beregnPris(int antallKebab 4 ,double prisPrStk 30.5 ) {
       // kode for å beregne prisen
    }
    ...
}
```

Først initialiseres formelle parametere, så utføres metoden.

Verdien til aktuelle parametere er uendret ved retur.

Formelle parametere kan aldri endre verdiene til aktuelle parametere.

Parameteroverføring: *referanseverdien til tabeller*

```
class SelectionSort {
    int a[] = {8,4,6,2,1};
    ...
    void sort() {
        for (int ind = 0; ind < a.length - 1; ++ind)
            ombytt(a, ind, minstIndeks(a , ind));
        ...
    }
    ...

    int minstIndeks(int tab[] , int startIndeks) {
    ...
    }
    void ombytt(int tab[], int i, int j) {
    ...
    }
}
```

[0]	8
[1]	4
[2]	6
[3]	2
[4]	1

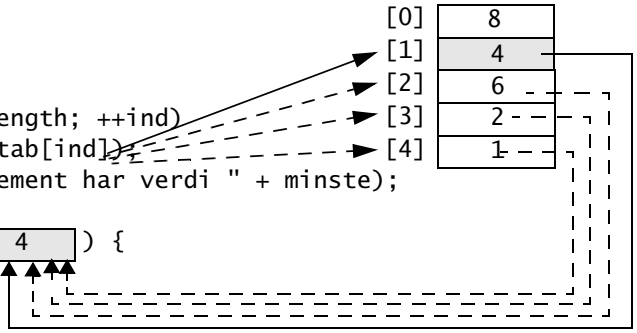
Først overføres verdien av tabellreferansen, så utføres metoden.

Legg merke til at et kall forekommer i et annet kall. Kall som forekommer som parameter, blir utført først. I dette tilfellet vil `minsteIndeks()`*-metoden blir kalt før* `ombytt()`*-metoden.*

Tabellreferansen kan brukes til å lese eller skrive elementverdier.

Parameteroverføring: *tabellelement av primitiv type*

```
class FinnMinste {
    int tab[] = {8,4,6,2,1};
    void finnMinste() {
        int minste = tab[0];
        for (int ind = 1; ind < tab.length; ++ind)
            minste = minimum(minste, tab[ind]);
        System.out.println("Minste element har verdi " + minste);
    }
    int minimum(int i,        int j  4  ) {
        int min = i;
        if (j < i) min = j;
        return min;
    }
}
```

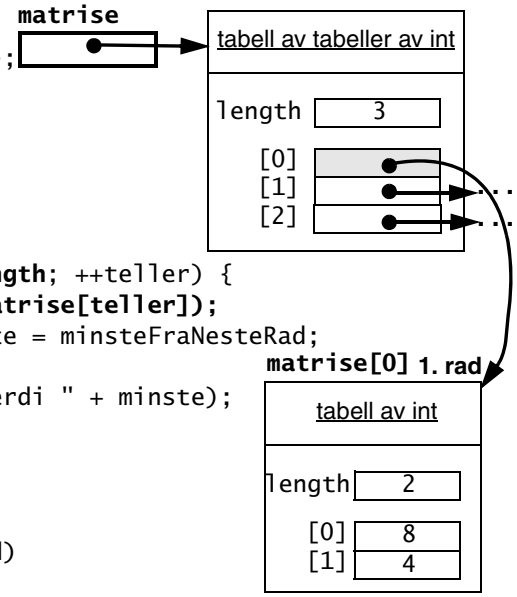


Hvert element i `tab` *er av primitiv datatype (*`int`*), og verdien i hvert element overføres.*

```
public class MinsteKlient {
    public static void main(String[] args) {
        new FinnMinste().finnMinste();
    }
}
```


Parameteroverføring: *tabell av tabeller*

```
public class FinnMinsteMxN {
    int matrise[][] = {{8,4},{6,2,2},{9,4,1,7,1}};

    void finn() {
        int minste = finnMinste(matrise[0]);
        for (int teller = 1; teller < matrise.length; ++teller) {
            int minsteFraNesteRad = finnMinste(matrise[teller]);
            if (minsteFraNesteRad < minste) minste = minsteFraNesteRad;
        }
        System.out.println("Minste element har verdi " + minste);
    }

    int finnMinste(int tab[]) {
        int minste = tab[0];
        for (int ind = 1; ind < tab.length; ++ind)
            minste = Math.min(minste, tab[ind]);
        return minste;
    }
}
```



Hvert element i `matrise` *er en tabell, slik at referanseverdien til tabellen overføres.*

Samsvar mellom formelle og aktuelle parametere

- 1-1 korrespondanse mellom formelle og aktuelle parametere.
  - rekkefølgen av og antall aktuelle parametere må matche rekkefølgen av og antall formelle parametere i metodedefinisjon.
  - aktuelle parametere må være *type-kompatible* med formelle parametere: *verdien til den aktuelle parameteren kan tilordnes den formelle parameteren*.
    - dersom den formelle parameteren er av primitive datatype, må den aktuelle parameteren være av samme primitive datatype (eller en datatype som kan implisitt konverteres til denne primitive datatypen).
    - dersom den formelle parameteren er av en referansetype, må den aktuelle parameteren være et objekt av samme klasse (eller et objekt av subklasser til denne klassen).
  - kompileringsfeil dersom rekkefølge, antall og type ikke er i samsvar.

Se også følgende eksempler: FinnMinsteV2.java, MinsteKlientV2.java og FinnMinsteMxNV2.java, MinsteKlientMxNV2.java

Eksempler med parameteroverføring

- Hva er galt? Logisk feil i program!

```
public class Ombytt {
    public static void main(String[] args) {
        int n = 10, m = 5;
        System.out.println("Før ombytting: n = " + n + " og m = " + m);
        ombytt(n, m);
        System.out.println("Etter ombytting: n = " + n + " og m = " + m);
    }
    static void ombytt(int i, int j) {
        int temp = i;
        i = j;
        j = temp;
    }
}
```

Utskrift:

```
% java Ombytt
Før ombytting: n = 10 og m = 5
Etter ombytting: n = 10 og m = 5
```

Implisitt konvertering under parameteroverføring

- Dette skjer når en aktuell parameter har en verdi som er av smalere type enn den til formell parameter.

```
class Kunder {
    ...
    void kjøpKebab () {
        KebabFabrikk kebabHus = new KebabFabrikk();
        ...
        // beregn pris for alle kebaber spist.
        // anta fast pris pr. kebab
        double pris = kebabHus.beregnPris(  4,  30  );

        ...
    }                                          30.0D
    ...
}
class KebabFabrikk {
    double beregnPris(int antallKebab, double prisPrStk) {
        // kode for å beregne prisen
    }
    ...
}
```


- Rekkefølge/ type-kompatibilitet:

```
class Kunder {
    ...
    void kjøpKebab () {
        KebabFabrikk kebabHus = new KebabFabrikk();
        ...
        // beregn pris for alle kebaber spist.
        // anta fast pris pr. kebab
        double pris = kebabHus.beregnPris(30.50, 4); // rekkefølge: logisk feil?

        ...
    }                                          4.0D
    ...
}
class KebabFabrikk {
    double beregnPris(int antallKebab, double prisPrStk) {
        // kode for å beregne prisen
    }
    ...
}
```

- Hva skjer i dette programmet?

```
public class RefParametere {
    public static void main(String[] args) {
        String streng = "Hold meg";
        System.out.println("Før: " + streng);      // Før: Hold meg
        konkat(streng);
        System.out.println("Etter: " + streng);    // Etter: Hold meg
    }
    static void konkat(String str) {
        System.out.println("Inn konkat: " + str);  // Inn konkat: Hold meg
        str = str + " fast";
        System.out.println("Ut konkat: " + str);   // Ut konkat: Hold meg fast
    }
}
```

  – formelle parametere er lokale variabler, og kan behandles deretter.
    - De kan *tilordnes* verdier i den kalte metoden, men det forandrer *ikke* verdiene til de aktuelle parameterene.

Oppsummering

Datatype til formelle parametere:	Parameteroverføring:
primitiv datatype	overføring av primitiv verdi
referansetype	overføring ved referanseverdi

Syntaks til *formelle parametere*:

`(<type1> <formparam1>, <type2> <formparam2>,…, <typen> <formparamn>)`

der `<typei>` er enten en *primitiv datatype* eller en *referansetype*, og `<formparami>` er en *identifikator* (med eller uten tabelloperatoren `[]`).

Syntaks til *aktuelle parametere* (også kalt *argumenter*):

`(<aktparam1>, <aktparam2>, ..., <aktparamn>)`

der `<aktparami>` er et *uttrykk* (aritmetisk, boolsk, streng) eller en *referanse*.

- Den *tomme* parameterlisten er angitt som `()`.
- Formelle og aktuelle parametere må matche m.h.t. *rekkefølge*, *antall*, og *type*.

Det aktuelle objektet: `this`

- Når en metode kalles på et objekt, hvordan kan metoden referere til selve objektet?

  Bruk nøkkelordet `this`!
- `this` er referansen til *det aktuelle objektet* som holder på å utføre metoden.
- `this`-referansen sendes *implisitt* i kall til instansmetoder, og kan brukes til å referere til *alle* medlemmer i klassen.
- `this`-referansen kan brukes for å referere til feltvariabler som er *skygget*.

```
class Lys {
    // Feltvariabler
    int antallWatt;
    boolean indikator;
    String lokasjon;
    void settVerdier(int antallWatt, boolean indikator, String lokasjon){
        this.antallWatt = antallWatt;
        this.indikator = indikator;
        this.lokasjon = lokasjon);
    }
    // Andre metoder ...
}
```


Eksempel: Bruk av `this`

```
class GUIvindu {
    GUIknapp knapp;
    GUIvindu() { // Et vindu har alltid en knapp.
        knapp = new GUIknapp(this); //Det aktuelle objektet sendt som parameter
        // ...
    }
    void foretaHandling() {
        // ...
    }
}
class GUIknapp {
    Vindu hovedvindu;
    GUIknapp(GUIvindu vindu) { // En knapp har en referanse til et vindu.
        hovedvindu = vindu;
        // ...
    }
    void informerVindu() {
         hovedVindu.foretaHandling(); // Knappen kan informere vinduet.
    }
}
```

Statiske medlemmer i klasser

- Det finnes tilfeller der *medlemmer* bare skal beholdes i klassen, og ikke blir del av objekter som opprettes.

Eksempel: Vi ønsker å holde rede på hvor mange objekter av klassen `Lys` er opprettet.

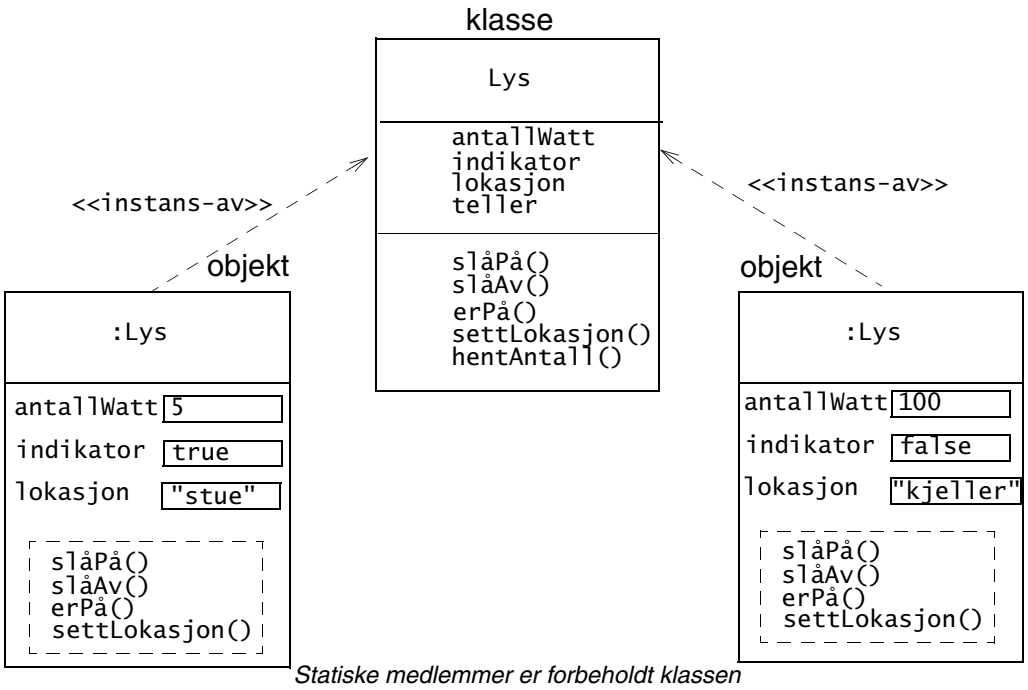
- Vi trenger en *global* teller, men den kan ikke være i hvert objekt!
- Vi trenger en *global* metode som kan påkalles for å finne ut hvor mange objekter som har blitt opprettet.

- Klienter kan kalle *statiske metoder* og aksessere *statiske variabler* v.h.a. klassenavn eller via referanser av denne klassetypen.

Begrep:

statiske metoder	*Metoder* som bare tilhører klassen
statiske variabler	*Variabler* som bare tilhører klassen
statiske medlemmer	statiske variabler *og* metoder

Statiske medlemmer i klasser (forts.)



Statiske medlemmer er forbeholdt klassen

Eksempel: statiske medlemmer

```
class Lys {
    // Statisk variabel
    static int teller;
    // Feltvariabler
    int antallWatt;    // lysstyrken
    boolean indikator; // av == false, på == true
    String lokasjon;   // hvor lys er plassert
    // Statiske metoder
    static void økTeller() {
        ++teller;
    }
    static int hentAntall() {
        return teller;
    }
    ...
}
```

- Etter at vi har opprettet et objekt av klassen `Lys`, kaller vi alltid den statiske mutatormetoden `økTeller()` for å øke verdien til den statiske variabelen `teller`.
- Den statiske selektormetoden `hentAntall()` returnerer verdien til den statiske variabelen `teller`.

Eksempel: statiske medlemmer (forts.)

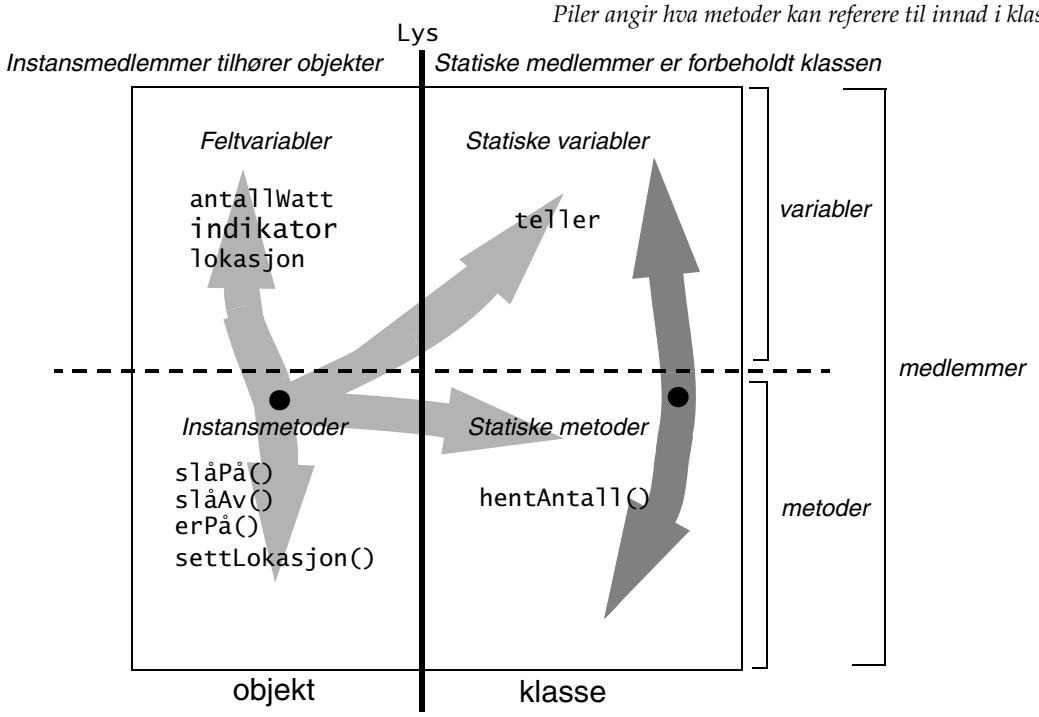
```
public class StaticDemo {
   public static void main(String[] args) {

      System.out.println("Antall Lys-objekter opprettet: " + Lys.hentAntall() );
      System.out.println("Oppretter et Lys-objekt.");
      Lys stuelys = new Lys();
      stuelys.økTeller();
      System.out.println("Antall Lys-objekter opprettet: " + stuelys.hentAntall() );

      System.out.println("Oppretter en tabell av Lys-objekter.");
      Lys[] lysTab = new Lys[10];
      for (int i = 0; i < 10; i++) {
          lysTab[i] = new Lys();
          lysTab[i].økTeller();
      }
      System.out.println("Antall Lys-objekter opprettet: " + Lys.hentAntall() );
   }
}
```

Vi kan alltid bruke klassenavn.

Dersom vi har deklarert referanser, kan vi også bruke referanser.

Å referere til medlemmer innad i klassen

Piler angir hva metoder kan referere til innad i klassen.



Klassemedlemmer

- En klassedeklarasjon kan inneholde følgende medlemmer:
  - *Feltvariabler*
  - *Instansmetoder*
  - *Statiske variabler*
  - *Statiske metoder*
- Feltvariabler er lokalisert til et objekt, dvs. hvert objekt har sin kopi av alle feltvariabler.
- Statiske variabler er globale for alle objekter av klassen, dvs. bare én kopi av en statisk variabel eksisterer sammen med klassen.
- Bare én implementasjon av en metode eksisterer i klassen, og denne benyttes av alle objekter til klassen.
- Statiske medlemmer har *ikke* direkte adgang til instansmedlemmer innad i klassen.
- Statiske medlemmer til en klasse eksisterer uavhengig av objekter til klassen.
- Ingen `this`-referanse blir overført for statiske metoder.
- Klienter kan bruke statiske medlemmer *uten* å opprette objekter.

Typer

- En *type* definerer en mengde av *lovlige verdier* og *operasjoner* som kan anvendes på disse verdiene.

Primitive datatyper	• primitive verdier
Referansetyper: • Klasser • Tabeller	• referanseverdier som betegner objekter

- Primitive datatyper er nedfelt i Java.
- Tabeller er også nedfelt i Java.
- Klasser er *brukerdefinerte* typer.

Definisjonsområde

- *Definisjonsområdet til en variabeldeklarasjon* (også kalt *scope*) er hvor variabelen kan *brukes direkte* med dens enkeltnavn uten å angi hvor den er deklarert.

Innad i en klasse:

- En instansmetode i en klasse kan bruke *alle* medlemmer i klassen direkte.
- En statisk metode i en klasse kan bare bruke andre *statiske* medlemmer i klassen direkte.

Innad i en blokk:

- gjelder *lokale variabler* (parametere + variabler i *metodekroppen*).
- Definisjonsområdet til en lokal variabel begynner ved dens deklarasjon og avslutter der blokken den er deklarert i slutter (*se figur med lokal blokk*).

Levetid

- *Levetid til en variabeldeklarasjon* er perioden variabelen eksisterer i minnet under utføring.

Levetid for lokale variabler:

- gjelder *alle lokale variabler* (parametere + variabler i *metodekroppen*).
- lokale variabler i en blokk opprettes ettersom variabeldeklarasjoner blir utført i blokken.
- lokale variabler må initialiseres eksplisitt før bruk.
- lokale variabler slutter å eksistere etter at kontrollen går ut av blokken.

Levetid for feltvariabler:

- gjelder *feltvariabler* i et objekt.
- Ved *instansiering* blir feltvariabler opprettet og automatisk initialisert til standardverdier dersom ingen eksplisitt initialisering foretas av programmet.
- feltvariabler eksisterer så lenge objektet de tilhører, eksisterer.

Levetid for statiske variabler:

- gjelder *statiske variabler* til en klasse.
- Ved *lasting* av klassen under utføring, blir *statiske variabler* opprettet og initialisert kun én gang.
- Statiske variabler eksisterer så lenge klassen eksisterer.

Kommandolinjeargumenter

- Formell parameter `args` til metoden `main` er en *tabell av strenger* (`String[]`) der hver streng tilsvarer et argument gitt på kommandolinjen.

```
public class Args {

    public static void main(String[] args) {

        for (int i = 0; i < args.length ; ++i)
            System.out.println(args[i]);

    } // main
}
```

  – opplysninger kan brukes til å skreddersy programmet etter brukerens ønsker.

Kompilering og kjøring:

```
% javac Args.java
% java Args grønn 18pkt 24x80
grønn
18pkt
24x80
%
```


Konstruktører

- En *konstruktør* har samme navn som klassen den tilhører:
  `<klasseNavn>(<parameterliste>) {...}`
- Dersom en klasse *ikke* definerer en konstruktør, blir *den implisitte standardkonstruktøren* automatisk anvendt ved opprettelsen av objekter for klassen.
  – En klasse kan eksplisitt definere standardkonstruktøren for å foreta nødvendige handlinger.
  `<klasseNavn>() {...}`
  – En konstruktør kan ikke returnere en verdi.
- En konstruktør blir utført når et objekt opprettes med `new`-operatoren.
- En konstruktør kan bl.a.
  – initialisere feltvariabler.
  – utføre operasjoner som måtte være nødvendig for å initialisere objektet, f.eks. opprette andre objekter hvor nødvendig.

Konstruktørdeklarasjon

- Klassen kan definere konstruktører som kan benyttes ved opprettelsen for å initialisere objektets tilstand til andre verdier enn standardverdier.

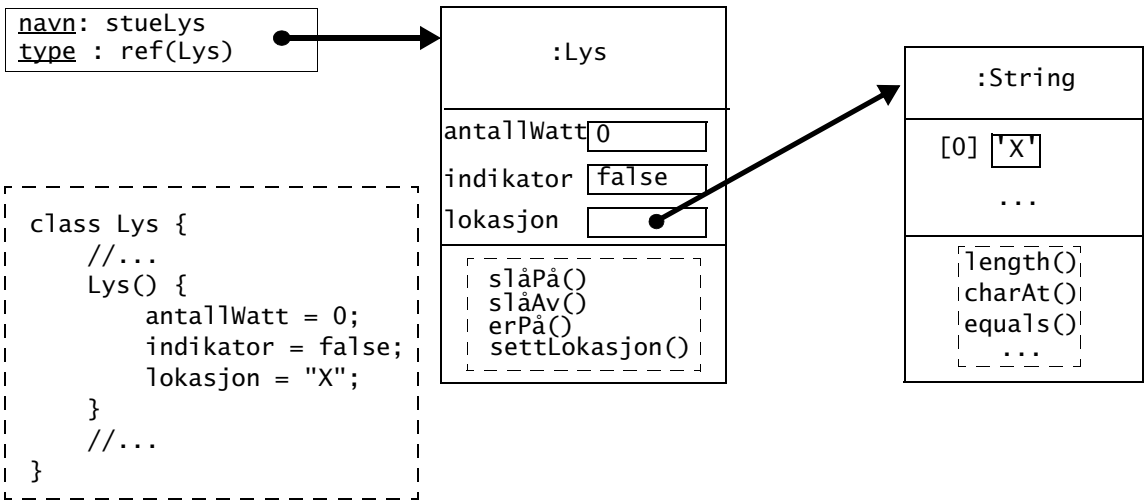
```
class Lys {
    // Feltvariabler kan enten initialiseres i feltvariabledeklarasjoner:
    int     antallWatt = 0;       // lysstyrken
    boolean indikator  = false;   // av == false, på == true
    String  lokasjon   = "";      // hvor lys er plassert
    // eller så kan den eksplisitte standardkonstruktøren brukes:
    Lys() {
        antallWatt = 0;
        indikator  = false;
        lokasjon   = "X";
    }
    // og/eller definere konstruktører som klienter kan bruke til å initialisere
    // feltvariabler med eksplisitte verdier:
    Lys(int watt, boolean ind, String lok) {
        antallWatt = watt;
        indikator  = ind;
        lokasjon   = lok;
    }
}
```


Eksempel på en klient av klassen `Lys`:

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

        Lys stueLys = new Lys(); // Bruker den eksplisitte standardkonstruktøren.

        // Vi kan kombinere deklarasjon, opprettelse og initialisering med
        // eksplisitte verdier:
        Lys kjellerLys = new Lys(30, true, "kjeller");
    }
}
```

Den eksplisitte standardkonstruktøren

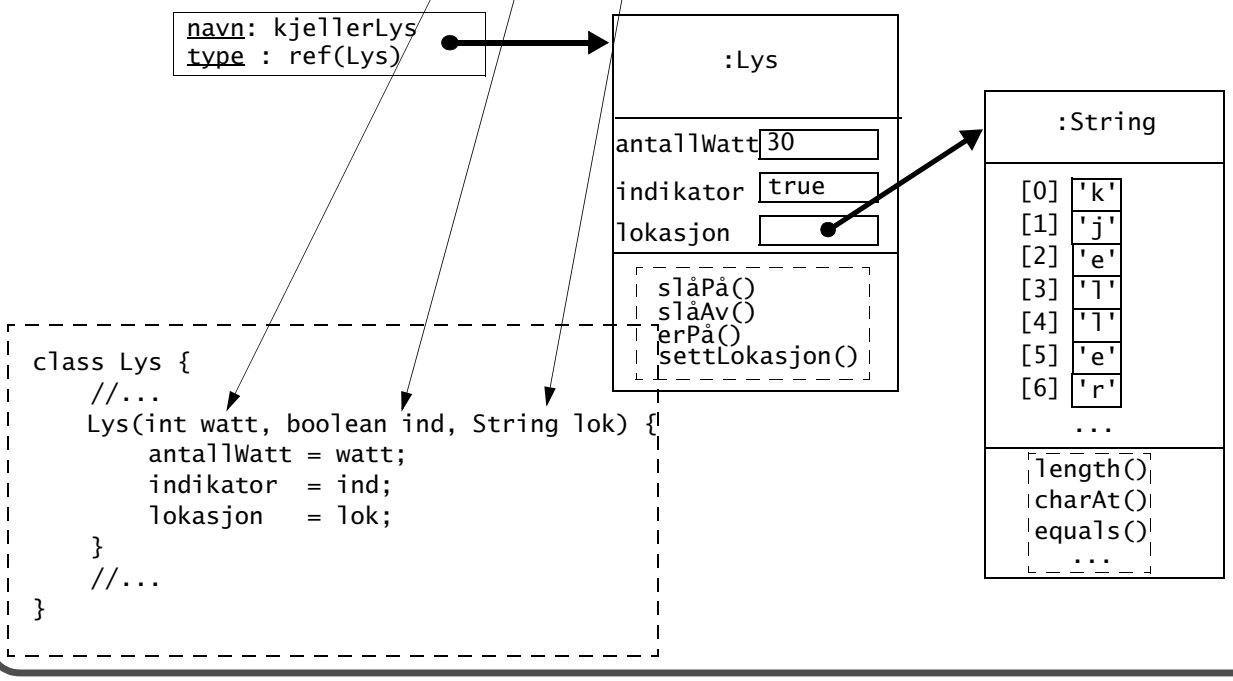
```
Lys stueLys = new Lys(); // bruker den eksplisitte standardkonstruktøren
```



Bruk av standardkonsuktøren fører til at alle objekter får samme tilstand ved opprettelsen.

Ikke-standardkonstruktører

```
Lys kjellerLys = new Lys(30, true, "kjeller"); // oppretter et objekt av klassen Lys
```



Konstruktøroverlasting

- Vi kan definere flere konstruktører for klassen `Lys` som har samme (klasse)navn:

```
class Lys {
    ...
    Lys() {                        // 1: den eksplisitte standardkonstruktøren
        antallWatt = 0;
        indikator  = false;
        lokasjon   = "X";
    }
    Lys(int watt, boolean ind) {            // 2: En ikke-standard konstruktør
        antallWatt = watt;
        indikator  = ind;
    }
    Lys(int watt, boolean ind, String lok) { // 3: En ikke-standard konstruktør
        antallWatt = watt;
        indikator  = ind;
        lokasjon   = lok;
    }
    ...
}
```

Dette kalles *konstruktøroverlasting*. Eksempelet viser *overlasting i antall parametere*.
Tilstanden til objektet ved opprettelsen vil være avhengig av konstruktøren som ble utført.

En del merknader om konstruktører

- Dersom ingen konstruktør er definert for klassen, blir den *implisitte standardkonstruktøren* utført når et objekt blir opprettet med `new`-operatoren.
- Dersom konstruktør(er) er definert i klassen, blir den *implisitte standardkonstruktøren ikke* opprettet.
- Dersom bare *ikke-standardkonstruktør(er)* er definert i klassen, er det feil å kalle den implisitte *standardkonstruktøren*.
- Konstruktører garanterer at et objekt får en *konsistent* tilstand ved opprettelsen.

Oppramstyper

- En oppramstype definerer et *fast antall oppramskonstanter*.

```
enum SivilStatus {
  UGIFT, GIFT, SEPARERT, SAMBOER, REGISTRERT_PARTNER, ALENE_FORSØRGER
}
```

  – Vi må spesifisere *alle* oppramskonstanter til en oppramstype i deklarasjonen.
- Vi kan deklarere referansevariabler av en oppramstype, på likelinje med andre referansetyper:

```
SivilStatus status = SivilStatus.SAMBOER;
```

- Vi kan sammenligne oppramskonstanter:

```
if (status == SivilStatus.SAMBOER) {
  System.out.println(status + ": skattelignes hver for seg.");
} else {
  System.out.println(status + ": se skatteregler.");
}
```


- Vi kan sammenligne oppramskonstanter i en `switch`-setning:

```
switch(status) { // (1)
  case UGIFT:
    System.out.println(status + ": skattelignes i skatteklasse 1.");
    break;
  case SEPARERT: case SAMBOER:
    System.out.println(status + ": skattelignes hver for seg.");
    break;
  case ALENE_FORSØRGER:
    System.out.println(status + ": skattelignes i skatteklasse 2.");
    break;
  default:
    System.out.println(status + ": skattelignes som ektefelle.");
}
```

  – Når vi kaller `toString()`-metoden i en oppramskonstant, blir navnet på oppramskonstanten skrevet ut, f.eks.

```
ALENE_FORSØRGER: skattelignes i skatteklasse 2.
```

- Vi kan lage en tabell med oppramskonstanter til en oppramstype ved å kalle den statiske metoden `values()`:

```
SivilStatus[] statusTabell = SivilStatus.values();
```

- Vi kan iterere over *en tabell med oppramskonstanter* med den forenkelete løkken:

```
for (SivilStatus sivilstatus : statusTabell) {
  System.out.println(sivilstatus);
}
```


Generell form for oppramstyper

- En oppramstype kan deklarere konstruktører og andre medlemmer, slik i en klassedeklarasjon.
  - Konstruktører kan ikke kalles direkte, siden vi ikke kan opprette nye objekter av oppramstyper med `new`-operatoren.
  - Instansmedlemmer kan bare kalles på objekter angitt ved oppramskonstanter.

```
enum Maaltid {
  // Oppramskonstanter for måltider.
  FROKOST(7,30), LUNSJ(12,15), MIDDAG(19,45);
  // Felt for klokkeslett for å servere måltid.
  private Klokkeslett serveringstid;
  // Konstruktør for et måltid.
  Maaltid(int tt, int mm) { serveringstid = new Klokkeslett(tt, mm); }
  // Metode for å returnere klokkeslett for måltid.
  public Klokkeslett hentServeringstid() {
    return this.serveringstid;
  }
}
```

```
/** Klokkeslett er gitt som timer (0-23) og minutter (0-59). */
class Klokkeslett {
  // Felt for et klokkeslett.
  int timer;
  int minutter;

  /** Konstruktør for å lage et klokkeslett */
  public Klokkeslett(int timer, int minutter) {
    assert (0 <= timer && timer <= 23 &&
            0 <= minutter && minutter <= 59) :
            "Ugyldig time og/eller minutter";
    this.timer = timer;
    this.minutter = minutter;
  }

  /** Strengrepresentasjon av et klokkeslett på formen TT:MM */
  public String toString() {
    return String.format("%02d:%02d", timer, minutter);
  }
}
```



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

    // (1) Lag en tabell av måltider.
    Maaltid[] måltidTabell = Maaltid.values();

    // (2) Skriv ut spisetider.
    for (Maaltid måltid : måltidTabell)
      System.out.println(måltid + " serveres kl. " +
                         måltid.hentServeringstid());
  }
}
```

Utføring:

```
FROKOST serveres kl. 07:30
LUNSJ serveres kl. 12:15
MIDDAG serveres kl. 19:45
```