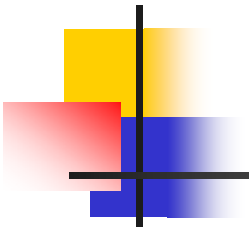
INF1000 (Uke 11)
Programmering

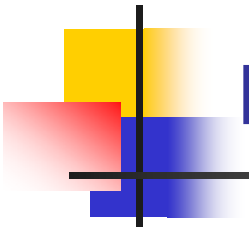
Grunnkurs i programmering
Institutt for Informatikk
Universitet i Oslo

Are Magnus Bruaset og Anja B. Kristoffersen



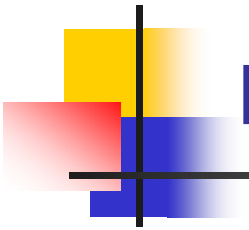
Innhold

- Litt repetisjon
- To måter å programmere på



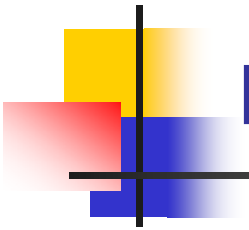
Rep: Hva er en metode?

- En metode er et valgfritt antall programsetninger som vi gir et navn
- Java: All kode i programmet er inne i en metode, som igjen inneholdes i en klasse



Rep: Metoder

- Skille mellom
  - *Deklarasjon* (skrive Java-kode og kompilere)
  - *Utføring* (det som skjer når vi kaller metoden)
- Når vi deklarerer en metode, skjer det ”ingen ting”
- En metode utføres hver gang den kalles fra koden i en annen metode:
  - Utførelsen av programmet hopper til starten av den kalte metoden
  - Har den kalte metoden parametere, kopieres verdiene brukt i kallet til metodens parameter-variable



Hva skjer når vi kaller en metode?

- Når vi *kaller* en metode, blir det opprettet et **metodeobjekt,** og verdiene brukt i kallet til parameterne blir kopiert over

- Dette metodeobjektet
  - inneholder alle lokale variabler og parameterne til metoden
  - når setningene i metoden utføres, brukes disse variablene og parametrene av metoden
  - metodeobjektet fjernes automatisk når metoden er ferdig utført og returnerer

- Merk forskjellen på å deklarere en metode, og at den utføres

```
class C {

  int  skrivAntall(int i){
       System.out.println(" Du har kalt meg med:" + i);
       return i+10;
}}

class D
{
  static int dobbel( int k) {
       return 2*k;
  }

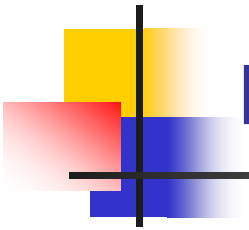
  void gjørMye(C cc, int v) {
       System.out.println(" gjørMye kalt");
       int j = cc.skrivAntall(v);
       System.out.println(" 1.verdien av j:" + j);
       j = dobbel(j);
       System.out.println(" 2.verdien av j:" + j);
       System.out.println(" 3.verdien av skrivAntall(j):"
                                  + cc.skrivAntall(j) );
  }

  public static void  main ( String[] args)  {
      C c = new C();
      D megSelv  = new D();

      megSelv.gjørMye(c,2);

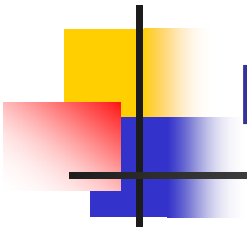
}}
```

```
>java D
gj°rMye kalt
Du har kalt meg med:2
1.verdien av j:12
2.verdien av j:24
Du har kalt meg med:24
3.verdien av skrivAntall(j):34
```



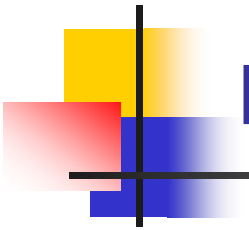
Hvorfor bruke metoder?

- Vi deler opp programmet i metoder fordi:
  - Noen program setninger brukes *flere* steder, eller:
  - Vi vil dele opp programmet i mindre deler
    - Ingen metode bør være lenger enn 30 linjer (og helst mindre)
  - Hver del gjør noe veldefinert som fremgår av navnet:
    - regner ut en bestemt formel
    - skriver ut en meny
    - leser noen data fra terminal eller fil
    - tegner ut opplysninger på skjermen
    - .....



Problemløsning med metoder

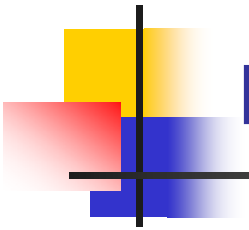
- Når vi har laget en metode, og vi har forsikret oss om at den er 'riktig', så har vi laget en *ny operasjon*
- Denne operasjonen er tilgjengelig i resten av koden, kan benyttes som om den var innebygd i Java
  - eks: skrive ut en meny, regne ut en bestemt formel,..
- Trenger ikke tenke på alle detaljene om *hvordan* denne operasjonen blir utført, bare *at* den blir gjort
- Vi har da laget et (lite) verktøy som kan gjenbrukes og lettere løse vårt større problem (hele systemet)



Bottom-up

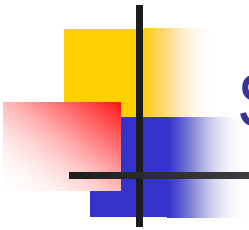
- Denne måte å programmere på heter *bottom-up* programmering og benyttes mye

- Eks: Java-biblioteket kan best forstås som en diger verktøykasse med nyttige operasjoner og datastrukturer som vi kan (og ofte bør) bruke for å lage vårt program



Hva er en klasse?

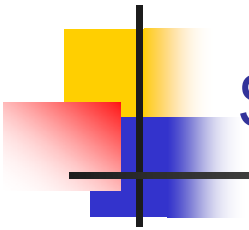
- En klasse er en beskrivelse av hvordan *ett* objekt av en bestemt type i vårt problem er
  - Inneholder variable som beskriver egenskaper for ett slikt objekt – eks:
    - Navn, adresse, studiepoeng, kurs... for klassen Student
    - Registreringsnummer, eier, type, årsmodel for klassen Bil
  - Inneholder metoder som er fornuftig handlinger for ett slikt objekt – eks:
    - skrivUtVitnemål(), meldPåEmne(),.. i klassen Student
    - beregnÅrsavgift(), skiftEier(),.. i klassen Bil



Skille mellom deklarasjon og bruk av en klasse

- Når vi deklarerer en klasse (= skriver Java-kode for) skjer det ”ingen ting” i programmet

- Når vi oversetter og starter opp programmet vårt med javac og java, skjer ”lite”:
  - De variable og metodene det står **`static`** foran blir tilgjengelige
  - Ingen kode (med unntak av main) utføres



Skille mellom deklarasjon og bruk av en klasse

- Først når vi sier **`new`** på en klasse, får vi laget et objekt av klassen
- Objektet inneholder alle variable og metoder som ikke har **`static`** foran deklarasjonen (objekt-variable og –metoder)
- Når vi sier **`new`**, kaller vi en konstruktør-metode i klassen, og først når den er ferdig, returnerer **`new`** det med det nye objektet

```
class Konto1 {
  String eier;
  int kontoNum, saldo = 0;

  Konto1(String e) {
       eier = e;
  }

  void  settInn(int beløp) {
       saldo = saldo + beløp;
  }

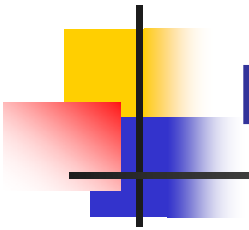
  boolean taUt(int beløp) {
        // moderne bank med muligheter for overtrekk
        saldo = saldo - beløp;
        return saldo > 0;
  }
}

class Bank1
{
  Konto1 [] kontiene = new Konto1[100000];

  public static void  main( String[] args)  {
     Bank1 b = new Bank1();

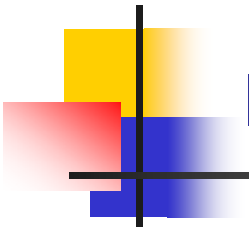
     for (int i = 0; i < b.kontiene.length; i++) {
        b.kontiene[i] = new Konto1("kunde nr." + i);
        b.kontiene[i].settInn(100);
    }

 }
}
```



Forskjeller mellom klasser og metoder

- Begge lager objekter når de kalles, MEN
- Et metode-objekt:
  - fjernes når metoden returnerer
  - inneholder ’bare’ variable og parametere som alle er skjult for resten av programmet
- Et objekt laget med **new** fra en klasse:
  - er i hukommelsen etter at det er laget (så lenge det minst er en peker som peker på det)
  - kan inneholde både metoder og variable, som kan nyttes av resten av programmet (med en peker og .)



Innkapsling

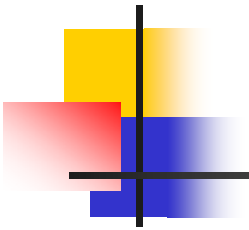
- Vi ønsker ofte at resten av systemet bare skal se deler av et objekt
  - eks: **`int saldo`** i Konto1-objektet bør være skjult, resten av programmet skal bare bruke **`settInn() og taUt()`** metodene.

- Vi kan regulere tilgangen til variable og metoder ved å sette enten:
  - **`private`**
  - **`public`**
  - **`protected`**

foran en metode eller deklarasjonen av en variabel

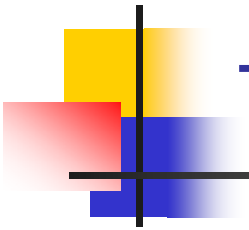
For 'små¨' systemer hvor alle .java filene ligger på samme filområde, gjelder

- **ingenting** foran en deklarasjon/metode
  - Fullt tilgjengelige for alle annen kode kompilert på samme filområde, men usynlig /sperret for kode kompilert på andre filområder
- **`private`** foran en deklarasjon/metode
  - Bare synlig fra kode i metoder deklarert i *samme klasse*, usynlig/sperret for all annen kode
- **`protected`** foran en deklarasjon/metode
  - Synlig i samme klasser og subklasser og synlig i klassene på samme filområdet, men usynlig/sperret i andre klasser (på andre filområder)
- **`public`** foran en deklarasjon/metode
  - Synlig for all annen kode



Innkapsling

- Delvis sperring av adgang til en variabel, sikrer oss at vi selv kan fullt ut bestemme hvordan variabelen skal endres



To måter å programmere på

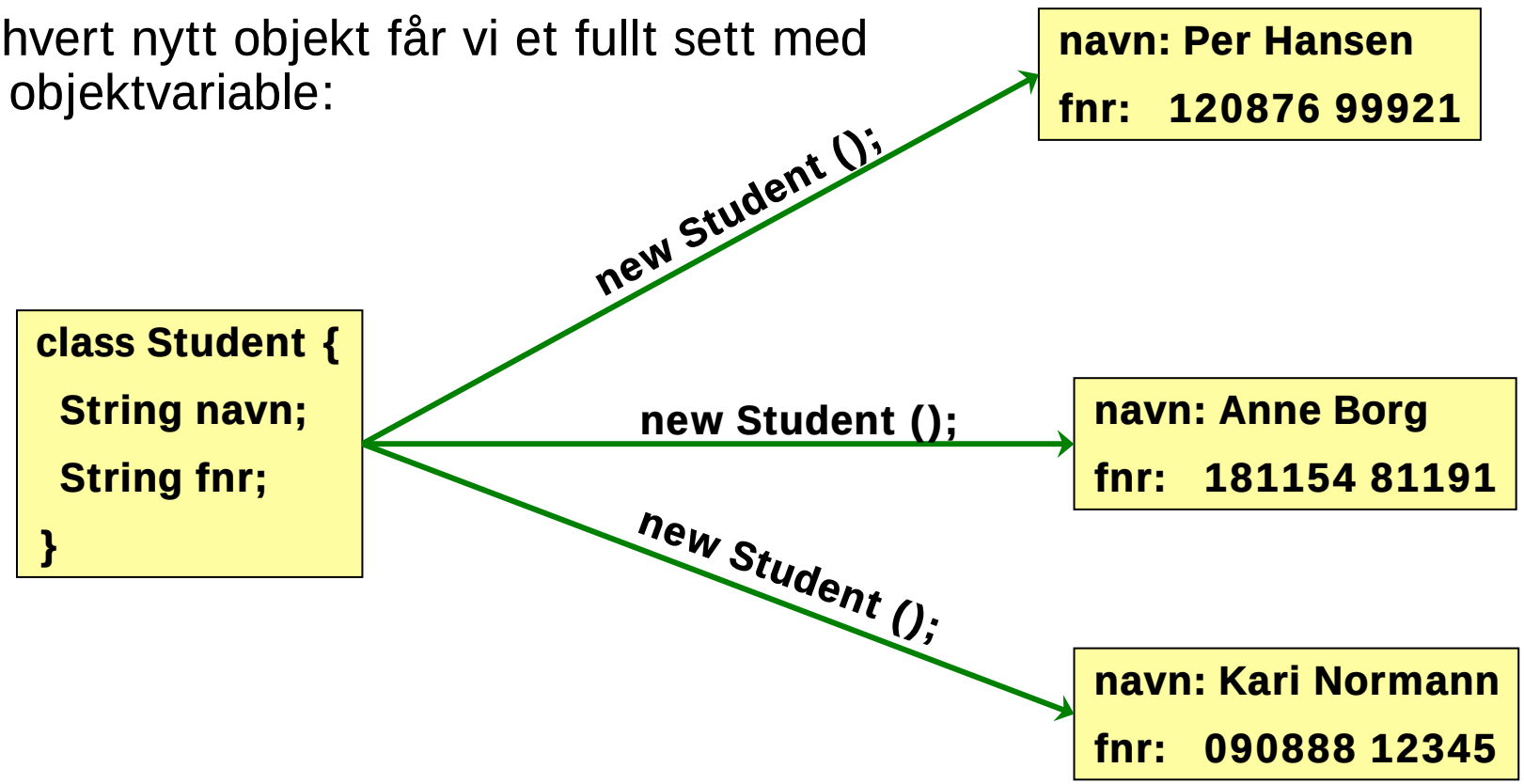
- **Statisk programmering:**
  - Var fokus i starten av kurset
  - Vi lager ikke objekter av klassene
  - Alle variable og metoder er deklarert som **`static`**
  - Begrepsmessig enkelt, men lite egnet for større programmer

- **Programmering med objekter:**
  - Er fokus for resten av kurset (og eksamen). Vi lager objekter av klassene (noen eller alle)
  - Variable og metoder er vanligvis *ikke* deklarert som **`static`**
  - Begrepsmessig noe mer komplisert, men mye bedre egnet for større programmer

Objektvariable

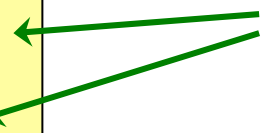
Objektvariable er variable på klassenivå som *ikke* er deklarert som **`static`**.

For hvert nytt objekt får vi et fullt sett med nye objektvariable:



Objektvariablenes levetid

```
class Student {
  String navn;
  String fnr;
}
```



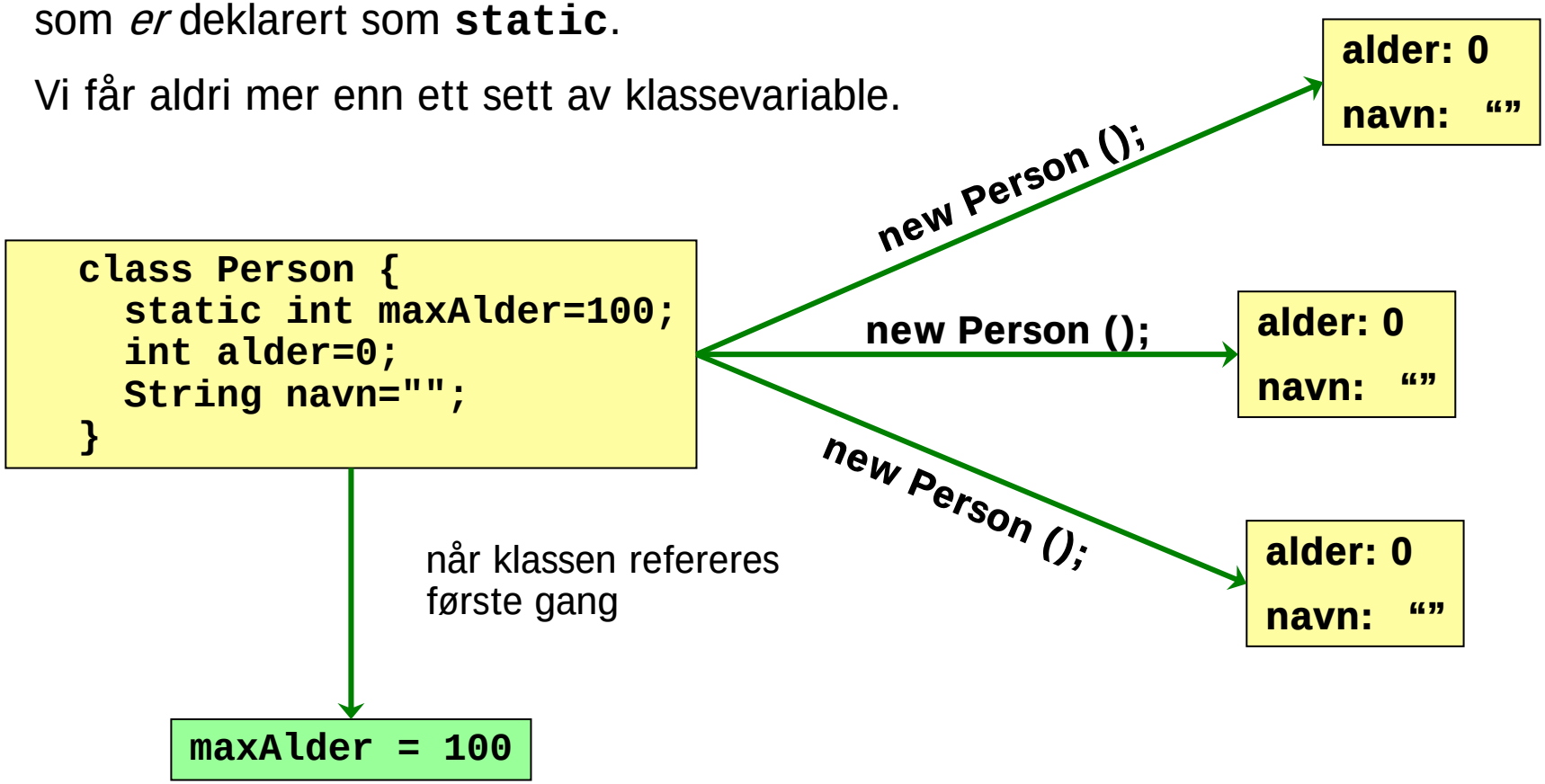
Disse variablene blir deklarert når vi lager et objekt av klassen ved å skrive **`new Student()`**, og de lever så lenge objektet lever.

- Objektvariablene blir til når objektet blir til (med new)
- Objektvariablene lever så lenge objektet finnes

Klassevariable

Klassevariable er variable på klassenivå som *er* deklarert som **`static`**.

Vi får aldri mer enn ett sett av klassevariable.



Klassevariablenes levetid

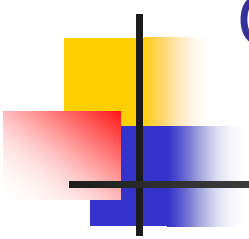
```
class Person {
  static int maxAlder=100;
  int alder=0;
  String navn="";
}
```

Denne variabelen blir deklarert når klassen `Person` *blir referert til for første gang* under kjøringen av programmet.

Variabelen lever helt til programmet avsluttes.

Første gang klassen Person blir referert til = første gang programeksekveringen "møter på" Person-klassen, f.eks. :

```
... new Person() ...
... Person.maxAlder ...
... Person.metode() ...
```



Oppsummering: Klassevariable og objektvariable

Objektvariable

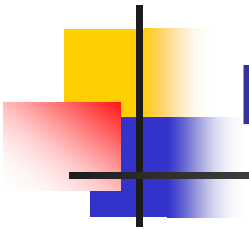
Bare definert i objekter av en klasse.

Hvert objekt har sitt eget sett med objektvariable.

Klassevariable

Definert selv om det ikke er laget objekter av klassen.

Alle objekter av klassen deler de samme klassevariablene.



Klassemetoder og objektmetoder

- Klassemetoder (static-metoder)
  - Definert selv om det ikke er laget noen objekter av klassen
  - Kan ”ses” av alle objekter av klassen
  - Kan brukes av andre gjennom dot-notasjon: <klassenavn>.metode(...)
  - Har ikke tilgang til objektvariable eller objektmetoder
- Objektmetoder
  - Bare definert i objekter av klassen
  - Kan ”ses” av objektet som metoden befinner seg i
  - Kan brukes av andre gjennom dot-notasjon: <peker>.metode(...)
  - Har tilgang til alle variable (både klassevariable og objektvariable) og alle metoder (både klassemetoder og objektmetoder)

Statisk programmering

```
class VolumBeregning {
  static double pi = 3.14;
  static int maxAntall = 10;
  public static void main (String [] args) {
     ...
}

static double finnVolum(double radius) {
  ...
}

static int finnSum(int k) {
  ...
}
}
```

Alle variable er deklarert som `static`

Alle metoder er deklarert som `static`

Programmering med objekter

```
class StudentRegister {
   public static void main (String [] args) {
      ...
   }
}
class Student {
   String navn;
   String fnr;
   void init() {
      ...
   }
   String finnNavn() {
      ...
   }
}
```

Variable er ikke deklarert som `static` (vanligvis)

Metoder er ikke deklarert som `static` (vanligvis)

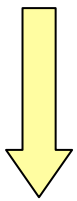
Hvorfor programmere med objekter? (1)

Med objekter kan vi ofte organisere våre data bedre.

Eksempel:

```
String [] navn = new String[100];
String [] fnr = new String[100];
int [] tlfnr = new int[100];
```

Informasjonen knyttet til en bestemt person er splittet opp i tre arrayer.



```
Class Person {
   String navn;
   String fnr;
   int tlfnr;

}

Person [] personreg = new Person[100];
```

Informasjonen knyttet til en bestemt person er samlet i et objekt.

Bedre organisering – særlig når det er mye data å holde styr på.

Hvordan programmere med objekter (2)?

Objekter tillater oss å samle data og tilhørende operasjoner

```
... data om studenter ...
... data om ansatte ...
... data om kurs ...
... student-metoder ...
... ansatt-metoder ...
... kurs-metoder ...
```

```
class Student {
   … data om studenter …
   … student-metoder …
}
class Ansatt {
   … data om ansatte …
   … ansatt-metoder …
}
class Kurs {
   … data om kurs …
   … kurs-metoder …
}
```

Her ligger alle data og alle metoder på samme sted

Metoder og data som hører sammen er samlet.

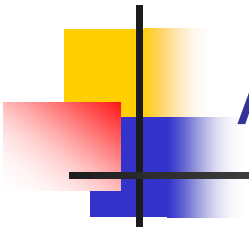
Lett å se hvilke metoder som jobber på hvilke data (modularisering av koden).

Lett å kopiere alt som har med personer å gjøre (data+metoder) til andre programmer (gjenbruk).

Hvordan programmere med objekter (3)?

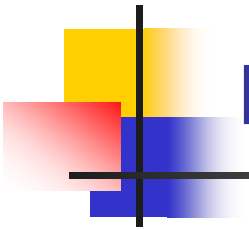
- Eksempel: Register for studentboliger
  - En rekke studenter → **`class Student`**
  - En rekke hybler → **`class Hybel`**
  - Et hybelhus (eller flere) → **`class Hybelhus`**

- En objektorientert løsning sørger for at
  - Variabler og metoder som logisk hører sammen også ligger samlet i programkoden
  - Variabler og metoder som ikke har noe med hverandre å gjøre holdes godt adskilt i programkoden



Analogi

- Ville du hjemme hos deg selv plassert verktøy, CD’er og søvtøy i samme skuff?
- Det er sannsynlig at du allerede tenker objektorientert!



Hvordan programmere med objekter

Grunnregel:

Et objekt skal inneholde data og operasjoner som naturlig hører sammen.

Et objekt kan representere:

- Et objekt i problemdomenet:
  - En konto, en bil, en student, et lån, en eiendom (brukes til å definere datamodell – en "modell av virkeligheten)
- Et objekt av mer programteknisk art:
  - Et skjermvindu, en fil, en tekststreng, en tabell (slike objekter ofte ikke del av datamodell – bare en effektiv grupperin av sammenhørende programelementer)

Initialisering av variable i et objekt

Anta at programmet inneholder denne klassen:

```
class Person {
   String navn;
   String fnr;

}
```

Når vi har laget et objekt (med **new**) ønsker vi å gi variablene i objektet verdier.

Prikk-notasjon

```
Person p = new Person();
p.Navn = “Petter”;
p.fnr = “15108559879”;
```

Init-funksjon

```
Person p = new Person();
p.init(“Petter”, “15108559879”);
```

Konstruktør

```
Person p = new Person(“Petter”, “15108559879”);
```

Finn- og sett-metoder

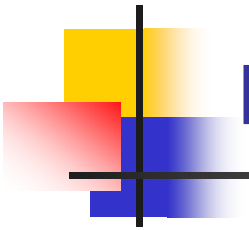
- Når man skal lage "virkelige" programmer er det vanlig å
  - Deklarere alle objektvariable som private
  - Bruke finn- og sett-metoder for å endre på objektvariable

```
class Eksempel {
   public static void main (String [] args) {
     Student stud = new Student();
     stud.settNavn("Petter");
     System.out.println(stud.finnNavn())
   }

}

class Student {
   private String navn;
   private String fnr;
   void settNavn(String navn) {this.navn = navn;}
   String finnNavn() {return this.navn;}

}
```



Konstruktør - repetisjon

- Konstruktør
  - spesiell type objektmetode som sikrer at objektet opprettes med fornuftige verdier i objektvariablene
- Konstruktører
  - Har alltid samme navn som klassen de ligger i
  - Utføres automatisk når et objekt opprettes med **`new`**
  - Har ingen returverdi, men skal ikke ha **`void`** foran seg
  - Overlastes ofte, dvs det er ofte flere konstruktører i en og samme klasse, hvor konstruktørene skiller seg fra hverandre ved antall parametre og/eller typen på parametrene

Eksempel

```
class TestSirkel {
   public static void main (String [] args) {
      Sirkel s1 = new Sirkel();
      System.out.println("Radius: " + s1.radius);
      Sirkel s2 = new Sirkel(5.0);
      System.out.println("Radius: " + s2.radius);
   }
}

class Sirkel {
   double radius;

   Sirkel () {
      radius = 1.0;
   }

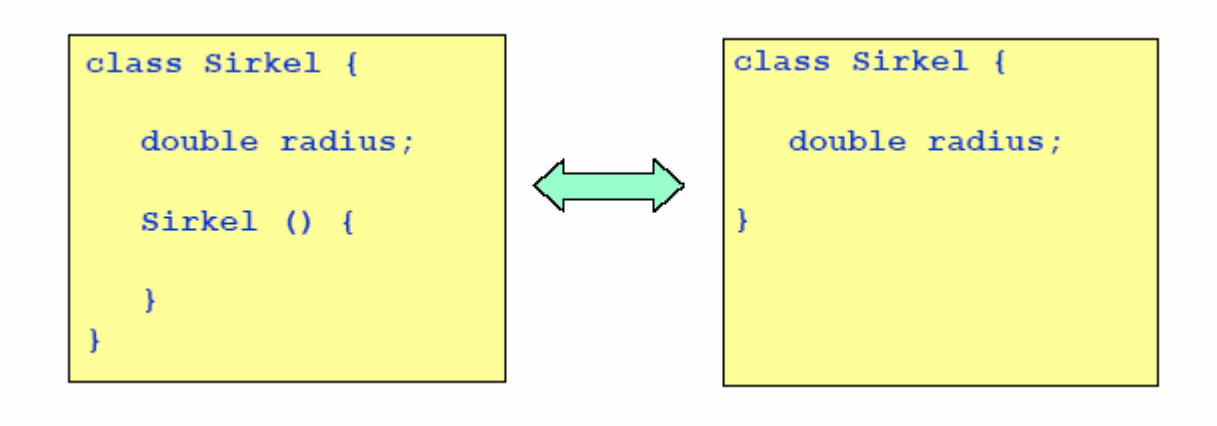
   Sirkel (double r) {
      radius = r;
   }
}
```

Konstruktør 1

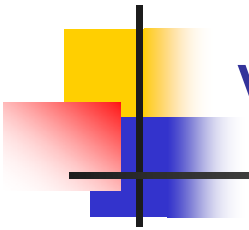
Konstruktør 2

Når det ikke finnes noen konstruktør

- Når en klasse ikke inneholder noen konstruktør, vil Java selv føye på en "tom konstruktør" uten parametre når programmet kompileres. Dermed er følgende to klassedeklarasjoner ekvivalente:



- Merk: dersom klassen inneholder en eller flere konstruktører, vil Java ikke føye på en "tom konstruktør" uten parametre.



Virker dette?

```
class Oppgave1 {
   public static void main (String [] args) {
         Bil b = new Bil();
   }
}
class Bil {
   String regnr;
   Bil (String regnr) {
         this.regnr = regnr;
   }
}
```

Valg av datamodell: eksempel

Eksempel:

Du har gitt en fil med opplysninger om hvor mange registrerte tilfeller det var av tre ulike sykdommer i Norge hvert av årene 1950…2000:

	INFLUENSA	KYSSESYKE	MENINGITT
1950	…	…	…
1951	…	…	…
1952	…	…	…
.			
.			
2000	…	…	…

Hvordan er det naturlig å modellere dette?

Noen muligheter

- Forslag 1: Gruppere tellinger relatert til samme sykdom
  - `class Sykdom { String navn;`
    `int[] antallTilfeller = new int[51]; }`
- Forslag 2: Gruppere tellinger foretatt samtidig
  - `class Aarsdata { int antInfluensa;`
    `int antKyssesyke; int antMeningitt; }`
- Forslag 3: Ingen gruppering - tre arrayer
  - `int[] influensatilfeller = new int[51];`
    `int[] kyssesyketilfeller = new int[51];`
    `int[] meningittilfeller = new int[51];`
- Forslag 4: Ingen gruppering – en 2D-array
  - `int[][] sykdomstilfeller = new int[3][51];`
- Beste datastruktur avhenger av hva du skal bruke dataene til.

Valg av datamodell: oblig 4

Anta at du har gitt en fil med opplysninger om en rekke værstasjoner:

4780	GARDERMOEN	202	ULLENSAKER	AKERSHUS
10400	RØROS	628	RØROS	SØR-TRØNDELAG
18700	OSLO-BLINDERN	94	OSLO	OSLO
25590	GEILO-GEILOSTØLEN	810	HOL	BUSKERUD
.....				

Her virker det naturlig å gruppere dataene stasjonsvis:

```
class Stasjon {
   String nummer;
   String navn;
   double høyde;
   String kommune;
   String fylke;

}
```

Valg av datamodell: oblig 4

For hver værstasjon har du gitt daglige værmålinger for et halvt år:

4780	01	01	1.5	0.0	-23.6	-13.3
4780	02	01	2.6	0.8	-13.7	-10.0
4780	03	01	4.6	1.3	-17.9	-11.7
4780	04	01	4.6	0.1	-23.3	-16.7
4780	05	01	5.7	0.0	-22.9	-15.7
4780	06	01	3.1	1.0	-22.9	-16.0
.....						

Hvordan modellerer vi disse dataene? Svaret er mindre opplagt.

Valg av datamodell: oblig 4

- **Forslag 1: Ingen gruppering**
  - ```
    class Stasjon {
      double[] maxvind;
      double[] nedbør;
      double[] mintemp;
      double[] maxtemp;
      ...
    }
    ```
- **Forslag 2: Gruppere målinger utført samtidig (dvs gruppere etter dag)**
  - ```
    class Dagdata {
      double maxvind;
      double nedbør;
      double mintemp;
      double maxtemp;
    }

    class Stasjon {
      Dagdata[] dagdata;
    }
    ```

Valg av datamodell: oblig 4

- **Forslag 3: gruppere etter både måned og dag**
  - ```
    class Dagdata {
      double maxvind;
      double nedbør;
      double mintemp;
      double maxtemp;
    }
    ```
  - ```
    class Maanedsdata {
      Dagdata[] dagdata = new Dagdata[31];
      int antDager; // Antall dager i måneden
    }
    ```
  - ```
    class Stasjon {
    Maanedsdata[] mdata = new Maanedsdata[6];
    }
    ```
- Dette er den varianten som blir anbefalt i hjelpenotatet for oblig 4
    ```

Organisering av veldig små programmer

- I *veldig små programmer* (noen få linjer) med en enkelt klasse og ingen`objekter av egne klasser, kan all programkode ligge i main-metoden:

```
import easyIO.*;
import java.io.*;
class VeldigLiteProgram {
 public static void main (String [] args) {
        In tast = new In();
        System.out.print("Gi et filnavn: ");
        String fnavn = tast.inLine();
        if (new File(fnavn).exists()) {
          System.out.println("Filen fins");
        } else {
         System.out.println("Filen fins ikke");
        }
 }
}
```

Organisering av alle andre programmer:

- I *alle andre programmer* bør **`main`**-metoden kun brukes til å få igang programmet. Eksempel:

```
class MittProgram {
 public static void main (String [] args) {
       Flyreservasjon fr = new Flyreservasjon();
       fr.ordreløkke();
 }
}
class Flyreservasjon {
 void ordreløkke() {
 ...
}
```

Variant 1

- Det er mulig å slå sammen de to klassene på forrige foil til en klasse:

```
class MittProgram {
public static void main (String [] args) {
 MittProgram mp = new MittProgram();
 mp.ordreløkke();
}

void ordreløkke() {
        ...
 }
}
```

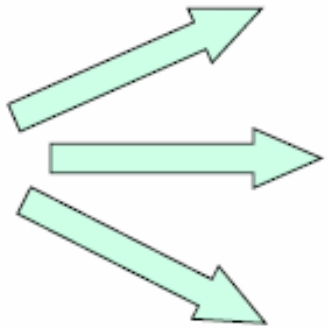
- Det blir en smakssak om man velger å gjøre det slik, eller slik som på forrige foil. I undervisningen vil du hovedsakelig se eksempler på den varianten som bruker to klasser.

Variant 2

- Istedet for å lage en egen ordreløkke-metode, kan koden for ordreløkken ligge i konstruktøren:
  - ```
    class MittProgram {
      public static void main (String [] args) {
              Flyreservasjon fr = new Flyreservasjon();
      }
    }
    ```
  - ```
    class Flyreservasjon {
      Flyreservasjon() {
              ... ordreløkken ...
      }
    }
    ```
- Dette blir også en smakssak, men varianten ovenfor bruker konstruktøren til noe annet enn initialisering av objektvariable og er mindre gjennomsiktig/selvforklarende enn å bruke en egen ordreløkke-metode.

Rollefordeling i større programmer

En klasse som inneholder alt



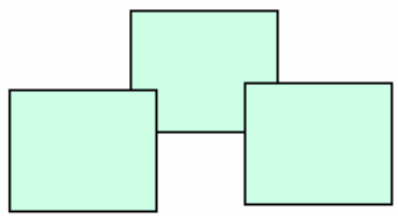
Klasse med main-metode
Inneholder main-metoden.

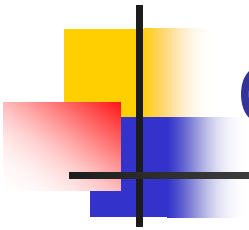
Gjør stort sett ikke annet enn å få programmet igang.

Klasse med ordreløkke
Holder på data-strukturen.

Inneholder evt ordreløkke, og metoder som kalles fra denne.

Øvrige klasser som inngår som del av datastrukturen





Oppsummering

- Legg ikke annen programkode i **`main`** enn det som skal til for å få igang programmet (typisk: lage et objekt og kalle på en metode i dette), unntatt i bittesmå programmer.
- I **`main`** kan du velge å lage et objekt av klassen som **`main`** ligger i - eller et objekt av en annen klasse. I kurset bruker vi stort sett siste variant.
- Metoden som kalles fra **`main`** er typisk den som er ansvarlig for programkontrollen (f.eks. en ordreløkke), eller en del av den.
- Programkontrollen kan ligge i en konstruktør eller i en annen metode (f.eks. **`void ordreløkke()`**). I kurset gjør vi stort sett det siste.