Objektorienterad programmering i Java

Föreläsning 3

Täcker i stort sett kapitel 8 (7) i kursboken
Java Software Solutions

Läsanvisningar

Den här föreläsningen är uppbyggd som en fortsättning av exemplet från föreläsning 2. Vi kommer att ta upp viktiga delar i kapitel 8 (7) och exemplifiera dessa. Exemplet bör du "skriva in" kompilera och exekvera. Föreläsningen ersätter inte boken utan skall användas som ett komplement till boken.

Du bör gå igenom denna föreläsning och motsvarande delar i kap 8 (7) innan du börjar med laboration 2.

Objektorientering

I inledning till föreläsning 2 nämns att objektorientering är;

•En metod för att avbilda verkligheten i en modell.
I modellen har man anammat människans förmåga att uppfatta
världen i konkreta saker och ting vilka i modellen representeras av objekt.

Vi utökar, något, utsagan om vad objektorientering är;

Människan har en förmåga att klassificera, indela i kategorier, saker och ting (objekt). Exempelvis Hus kan indelas i Villor, Radhus, Höghus etc. Datorer kan indelas i Mac, Pc etc. Gemensamt för Villor och, Radhus är att de är Hus. Det som är en egenskap för Hus är också egenskap för Villa, Radhus. Egenskaper ärvs.
Denna mekanism (arv) är central i objektorienterade ansatser.

Arv kap 8 (7) sid 286-
(321-)

Klass: gemensam beskrivning för grupper av objekt som har samma attribut. Man kan betrakta klass som en modell för hur ett objekt ska se ut. Samtliga objekt i en och samma klass skall ha samma uppsättning av attribut. Samtliga attribut skall (bör) vara aktuella för samtliga objekt.
De objekt som har avvikande attribut kan brytas ut ur klassen och bilda egen klass. Här följer ett exempel på att skapa en arvsmiljö.

Vi har klassen FlygPlan. Attribut är regnr, motorstyrka, vingbredd, färg och qtal. qtal är ett mått på segelplans flygförmåga.
I klassen flygplan ingår såväl segelplan som motorplan.
Attribut motorstyrka kan ej tillämpas för segelplan.
Vi bryter därför ut segelplan ur klassen flygplan och skapar klassen segelplan.

Arv forts.

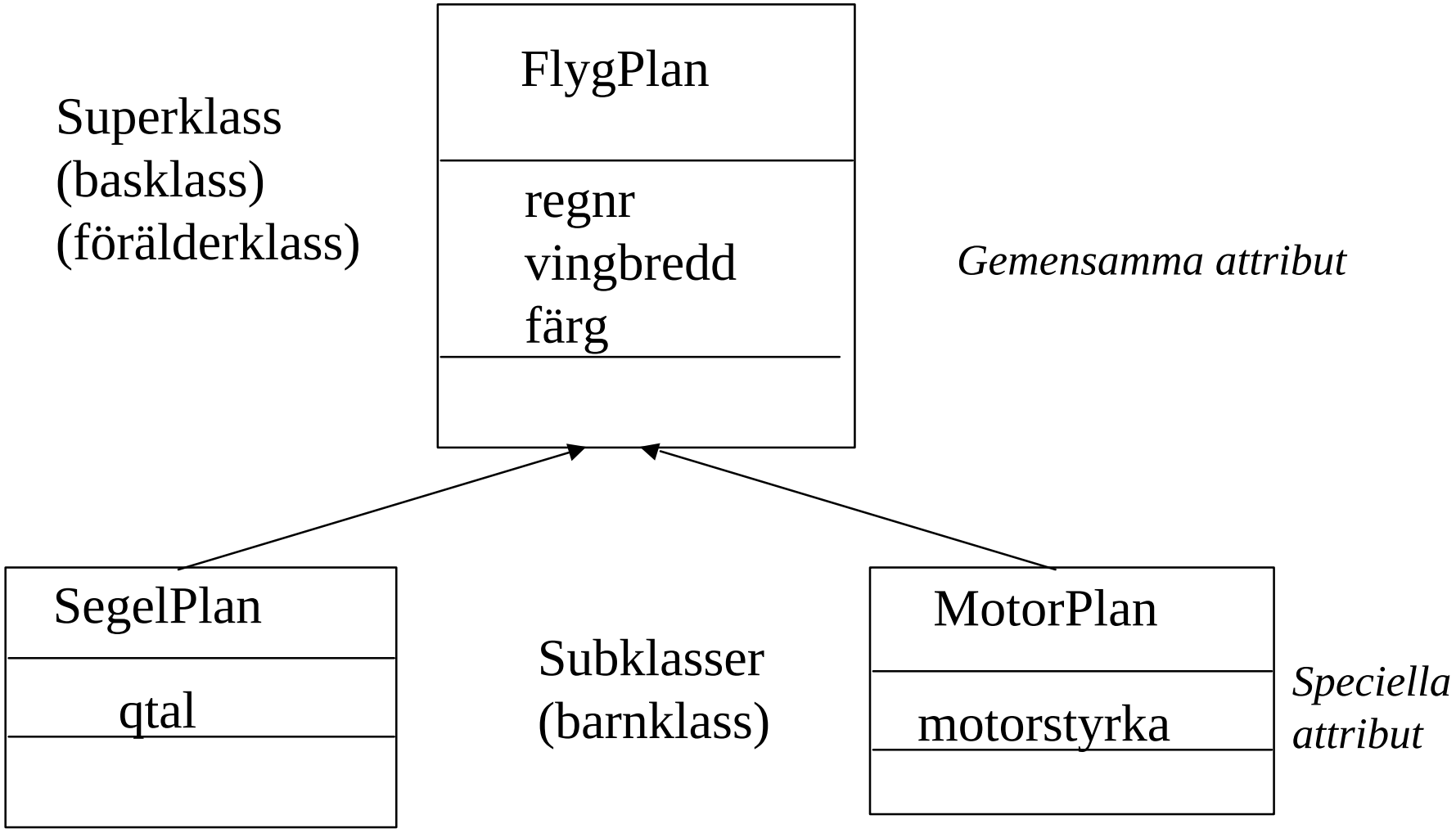
Klassen flygplan består därefter endast av motorplan. Vi kan därför döpa om klassen till motorplan. Vi har nu två klasser motorplan och segelplan. Till klasserna påför vi tillhörande attribut.

Attribut för klassen SegelPlan: regnr, vingbredd, färg, qtal

MotorPlan: regnr, vingbredd, färg, motorstyrka.

Vi ser att det finns gemensamma attribut (regnr, vingbredd, färg) för klasserna. Vi bryter ut gemensamma attribut för dessa klasser skapar (återskapar) klassen Flygplan. Vi har nu klasserna FlygPlan, SegelPlan, MotorPlan. Se nästa bild.

Arv. Klassdiagram



Arv forts.

Föregående bild är ett klassdiagram. Detta visar att FlygPlan kan vara av typen SegelPlan eller MotorPlan. Klassen FlygPlan kallas för superklass, SegelPlan och MotorPlan för subklass.
Subklasser ärver attribut från tillhörande superklass.
Attribut för SegelPlan är: regnr, vingbredd, färg, Q-tal.
MotorPlan är: regnr, vingbredd, färg, motorstyrka.

FlygPlan är antingen SegelPlan eller MotorPlan. Det finns bara dessa typer av FlygPlan som objekt (i verkligheten, fysiskt). Det finns inga objekt FlygPlan. Klassen FlygPlan kallas då abstrakt.
Klassdiagrammet beskriver ett arv om subtypen är en typ av supertypen.Således, Segelplan **är en** typ av FlygPlan!

Arv forts.

Vad vinner vi med att skapa arvsmekanism?
Svar; (bland annat).
-Färre attributsförekomster,
Vi kan använda regnr, vingbredd och färg för SegelPlan och MotorPlan.
-Enklare tillägg/borttag av attribut. Tillägg av nytt attribut i klassen FlypPlan kan tillämpas för Segelplan och Motorplan.

Observera. Samma arvsmekanism som gäller för **attribut** gäller också för **metoder**.

Exempel forts från bild 6

Våra klasser SegelPlan och MotorPlan är ganska lika, dvs
de har en del gemensamt i form av gemensamma attribut.
Använd arv för att sammanföra det som är gemensamt.

Vi skapar en klass (FlygPlan) som innehåller det som är gemensamt för de båda plantyperna. Därefter kan SegelPlan
och MotorPlan ärva från denna klass och utökas med det som är specifikt för dessa klasser.
Vi vinner: Samma attribut och kod (metoder) kan användas av flera klasser.Ytterligare plantyper kan skapas genom arv från den generella (super-) klassen FlygPlan.
Det enda vi tills vidare är intresserade av är att kunna skriva ut allt om en plantyp.

Exempel

Generellt och lika för plantyperna är:

Alla flygplan har ett :

regnr => attribut innehållande regnr.

vingbredd=> attribut innehållande vingbredd

färg=>attribut innehållande färg

utskrift av regnr,vingbredd och färg=> metod

Speciellt för plantyperna.

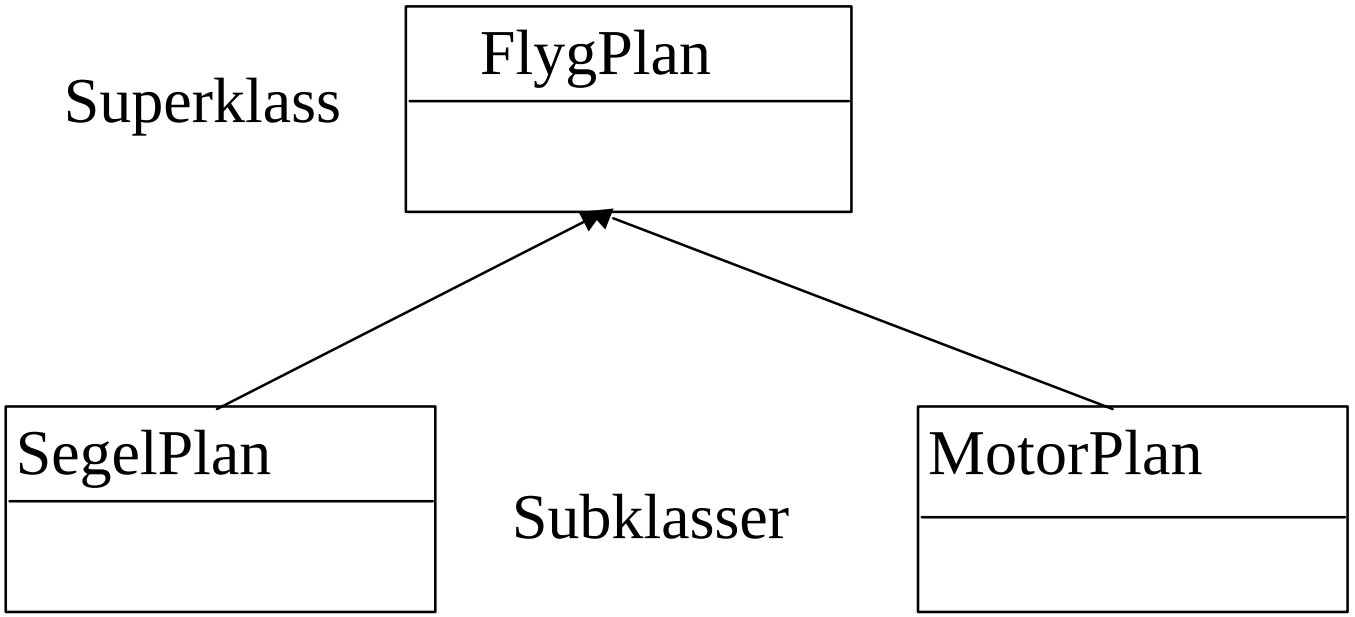
qtal=>attribut för segelplan

motorstyrka=>attribut för motorplan

Utskrift av qtal eller motorstyrka => metoder

Det *generella* tillförs superklassen, det *speciella* aktuell subklass.

Superklass och subklass



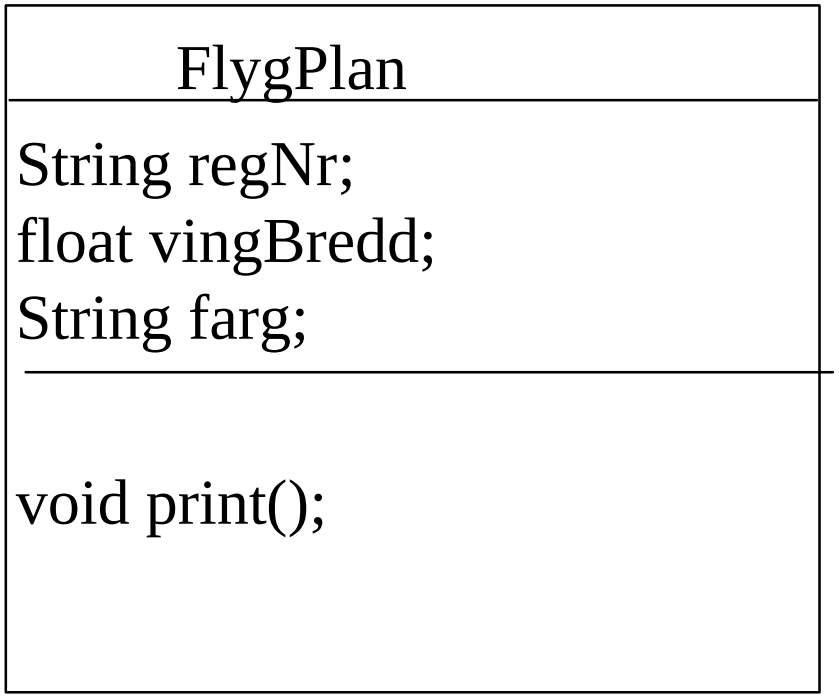
Superklass kallas också för förälderklass (parent class) eller basklass (base class).
Subklass kallas också för barnklass (child class).

Superklass FlygPlan

Klass

Attribut

Metod



SuperklassFlygPlan

```
class FlygPlan {

   protected String regNr;
   protected float vingBredd;
   protected String farg;

   public void print() {
      System.out.print("regnr:" +regNr+"bredd="+ vingBredd + farg);
   } // method print

   public FlygPlan ( String pregNr, float pvingBredd,String pfarg) {
      regNr=pregNr;
      vingBredd= pvingBredd;
      farg=pfarg;
   } //constructor

} // class FlygPlan
```

Modifierare, publika och privata

Man kan modifiera tillgänglighet för attribut och metoder.
Med tillgänglighet menas grad av möjlighet för andra än klassen / metoden att använda (se) attribut och eller anropa metoder.

Metoderna => i normalfall **publika** (public), åtkomlighet utifrån.
Attributen => i normalfall **privata** (private), endast åtkomliga i klassen.

För att göra attributen privata används modifieraren `private`**.**
För att förändra innehållet i ett attribut kan man skapa `publika` metoder som handhar detta. Grundtanken är att endast klassens metoder skall kunna påverka attributen.

Tillgänglighet (synlighet, visibility) behandlas utförligt i appendix F.

Inkapsling sid 143-(184-)

Klassen kapslar in (encapsulate) klassens attribut och metoder. Attribut och metoder tillhör klassen och bildar en gemensam enhet.
Varje objekt har sina attribut och kan anropa (använda) metoder som specificeras i klassen.

En "nivå" av inkapsling erhålls med modifieraren **private,** vilket innebär att attributet eller metoden endast kan användas inom klassen. Metoder o attribut som är private ärvs ej av subklass, men är definierade för subklassen, så de kan alltså nås via superklassen.
se sid 292 (364-).

Ett objekt är inkapslat. All interaktion med objektet sker via de metoder som finns definierade för det. se sid 143-148 (184-)

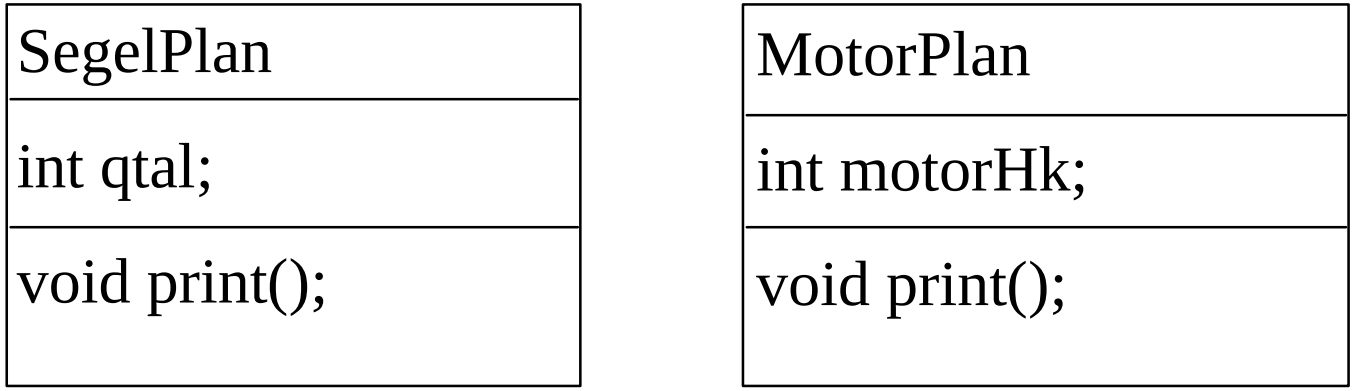
protected sid 289 (326-)

Man kan ange `protected` för superklassens

attribut, för att de ska ärvas av subklasser, attributen kan då nås direkt i subklassen.

På så sätt bibehålls inkapsling, vilket inte är fallet om vi skulle göra attributen publika, då kan de nås även av andra klasser än subklasser.

Subklasserna SegelPlan och MotorPlan



"Service" och "Support" - metoder se sid 144

(-)

Tjänstemetoder (service methods): de metoder som utifrån kan utföras på objektet. De metoder som objektet står till tjänst med.

Stödmetoder (support methods): de metoder som används inom objektet, dvs övriga metoder.

super se sid 290 (327)

`super` anropar superklassens konstruktor. Om den används måste den stå som första sats i subklassens konstruktor.

Önskvärt: Initiering av alla attribut med konstruktorn.

Lösning: konstruktorn i SegelPlan anropar konstruktorn i super-klassen FlygPlan.

```
Public SegelPlan(String pregNr,float pvingBredd,String pfarg,int pqtal)
{
    super(pregnr,pvingBredd,pfarg); // anrop av konstruktorn i klassen FlygPlan
    qtal=pqtal;

} // konstruktor SegelPlan
```

Subklassen SegelPlan

```
class SegelPlan extends FlygPlan {

   private int qtal;

   public void print()
   {
      super.print();
      System.out.println("qtal=" + qtal);
   } // metoden print

   public SegelPlan(String pregNr,float pvingBredd,String pfarg, int pqtal)
   {
      super(pregNr,pvingBredd,pfarg);
      qtal=pqtal;
   } //  konstruktor SegelPlan

}// klassen SegelPlan
```

MotorPlan

```
class MotorPlan extends FlygPlan {

   private int motorHk;

   public void print() {
      super.print();
      System.out.println("motorstyrka:" + motorHk);
   } // metoden print

   MotorPlan (String pregNr,float pvingBredd,String pfarg,int hk)
   {
      super(pregNr,pvingBredd,pfarg);
      motorHk=hk;
   } // konstruktor MotorPlan

} // klassen MotorPlan
```

extends se sid 286-287 (323)

När en klass ska ärva från en annan klass används nyckelordet `extends`.

```
class subClass extends superClass
{
        // klassbeskrivning
}
```

Subklassen ärver alla attribut och metoder som inte är **privata**.

Subklassen har tillgång till de ärvda attributen och metoderna som om de vore skrivna i subklassen.

Abstrakta klasser

(338-)

En abstrakt klass är en klass som:

- Har ofta (minst) en abstrakt metod.
- Inte kan instansieras (går inte att skapa objekt av klassen).
- Ser till att alla subklasserna implementerar den/de abstrakta metoden/metoderna.

ordet `abstract` skall stå före `class`

ex: `abstract class Fordon`

Det går att skapa objekt av superklassen `FlygPlan`.

Inte lämpligt => `FlygPlan` som varken är segelplan eller motorplan

Abstrakt klass

Hur förändras vår implementation?

Ordet `abstract` före `class` gör en klass abstrakt.

```
abstract class FlygPlan{    …..

}
```

Nu kan man **inte** instansiera `FlygPlan`, dvs man kan ej skapa ett *objekt* efter "ritningen" `FlygPlan`

PlanTest2

```
class PlanTest2 {

   public static void main (String[] args){

      SegelPlan s1=new SegelPlan("SE-123",12.4f,"vit",18);
      MotorPlan m1=new MotorPlan("SE-234", 8.2f,"röd",300);

      s1.print();
      m1.print();

   } // metod main

} // klass PlanTest2
```

- Lägg märke till att du måste ange parametrar i samma ordning som du gjort i konstruktorn för klassen
- Lägg märke till att för flyttalslitteraler (float) måste man sätta ett f efter (12.4f) annars antas den vara av typ double

Polymorfism sid 315

När en operation kan utföras på objekt av olika klasser men med olika implementation för varje klass, kallas det för polymorfism (en av flera olika typer av polymorfism).

Exempel beräkna ytan av en rektangel eller kvadrat

Metoden area i klassen Kvadrat:

```
public float area(float sida) {
   yta=sida*sida;
}
```

Metoden area i klassen Rektangel

```
public float area(float sida1, float sida2) {
   yta=sida1*sida2;
}
```

Polymorfism forts

```
class Kvadrat{

   protected float sida1;

   public Kvadrat(float psida1){
      sida1=psida1;
   }// konstruktor-metoden

   public float area(){
      return sida1*sida1;
   }//metoden area

}// klass Kvadrat
```

Polymorfism och overriding sid 296 (331-)

```
class Rektangel extends Kvadrat{
   private float sida2;
   public Rektangel(float psida1,float psida2){
      super(psida1);//anropar superklassens konstruktor
      sida2=psida2;
   }// konstruktor
   public float area(){
      return sida1*sida2;     // sida1 går att använda eftersom den är
                              // protected i superklassen och
                              // därmed ärvs av subklassen.
   }// metod area
}// klass Rektangel
```

Metoden area i subklassen overrides (~sätter sig över, upphäver) metoden area i superklassen.

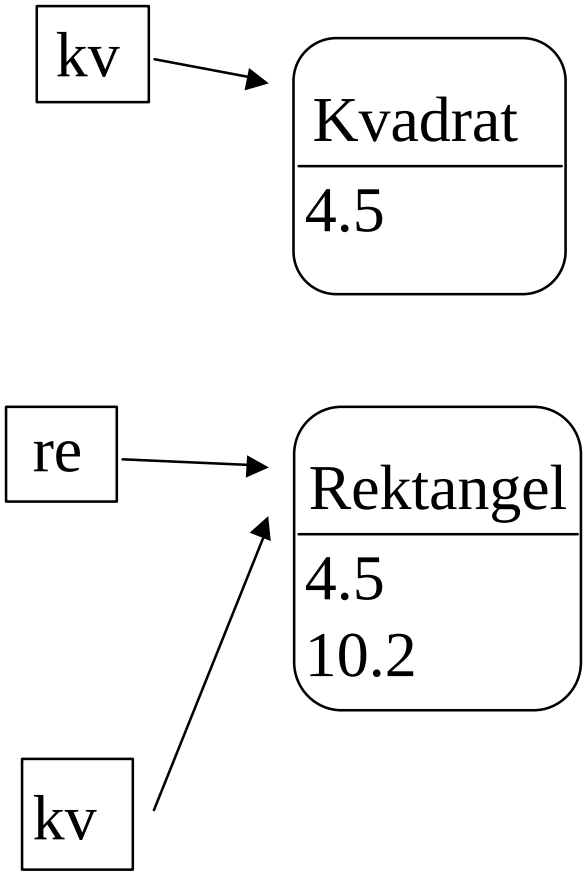
Polymorfism ,dynamisk bindning

```
class TestArea
{
   public static void main(String[] args){
   1.) Kvadrat kv=new Kvadrat(4.5f);
   2.) Rektangel re=new Rektangel(4.5f,10.2f);

      System.out.println("Kvadrat ytan="+kv.area());

   3.) kv=re;// Nu refererar kv till samma rektangel som re.

      System.out.println("ytan är nu=" +kv.area());
   }// metod main
}// class TestArea
```

Så här blir utskriften:

```
Kvadrat ytan=20.25
ytan är nu=45.899998
```

Dynamisk bindning forts



1.) `Kvadrat kv=new Kavadrat (4.5f);`

Nu är kv en referens till en Kvadrat med sidan 4.5

2.) `Rektangel re=new Rektangel(4.5f,10.2f);`

Nu är re en referens till en Rektangel med sidorna 4.5 och 10.2

3.) `kv=re;`

Nu refererar kv till Rektangeln

Polymorfism (201-)

En annan form av polymorfism är att man definierar flera metoder med samma namn, vanligt för konstruktorer. De måste dock ha olika *signatur*. Det innebär att för att ha samma namn så måste på något av dessa skilja:

-Antal parametrar.

-Typ på dessa parametrar.

-Ordningsföljden på parametrarna.

Ex:

```
public Kvadrat() {
      sida1=1.0f;     // om man vill ha 1.0 som defaultvärde
}// konstruktor utan parameter

public Kvadrat(float psida) {
      sida1=psida;
}// konstruktor med en parameter
```

"Överskuggning",overriding sid296

(331-)

Överskuggning (overriding) eller omdefiniering innebär att en metod som finns i en superklass omdefinieras i en subklass.

Metoden har samma huvud som den i superklassen.

Metoden har olika implementation, dvs metodkropparna är olika.

"Överskuggning" forts

```
class Javisst {
   public void skriv() {
      System.out.println("Javisst är Java bra");
   }//metod skriv
}// klass Javisst

class KanskeJava extends Javisst {
   public void skriv() {
      System.out.println("Kanske Java");
      System.out.println("Men är inte säker");
   }//metod skriv
}// klass KanskeJava
```

Metoden skriv är annorlunda i kroppen men lika i huvudet i både superklassen Javisst och subklassen KanskeJava

Dynamisk bindning

- Dynamisk bindning innebär att vilken metod (av de överskuggade) som ska anropas bestäms dynamiskt i exekveringsögonblicket.
- Den metod som väljs avgörs av vilken klasstyp objektet är.
- Ett objekt av en subklass som anropar metoden kommer att aktivera den kod som finns för metoden.

Polymorfism och dynamisk bindning (343-)

Vi kan genom överskuggning implementera samma metod i samma arvshierarki men med olika implementation (dvs olika i olika klasser)
=> *polymorfism*

se metoden area, bild 28

I exekveringsögonblicket avgör objektets klasstyp vilken implementation som ska användas
=> *dynamisk bindning*

`kv=re;` *se bild 28*

Klasser

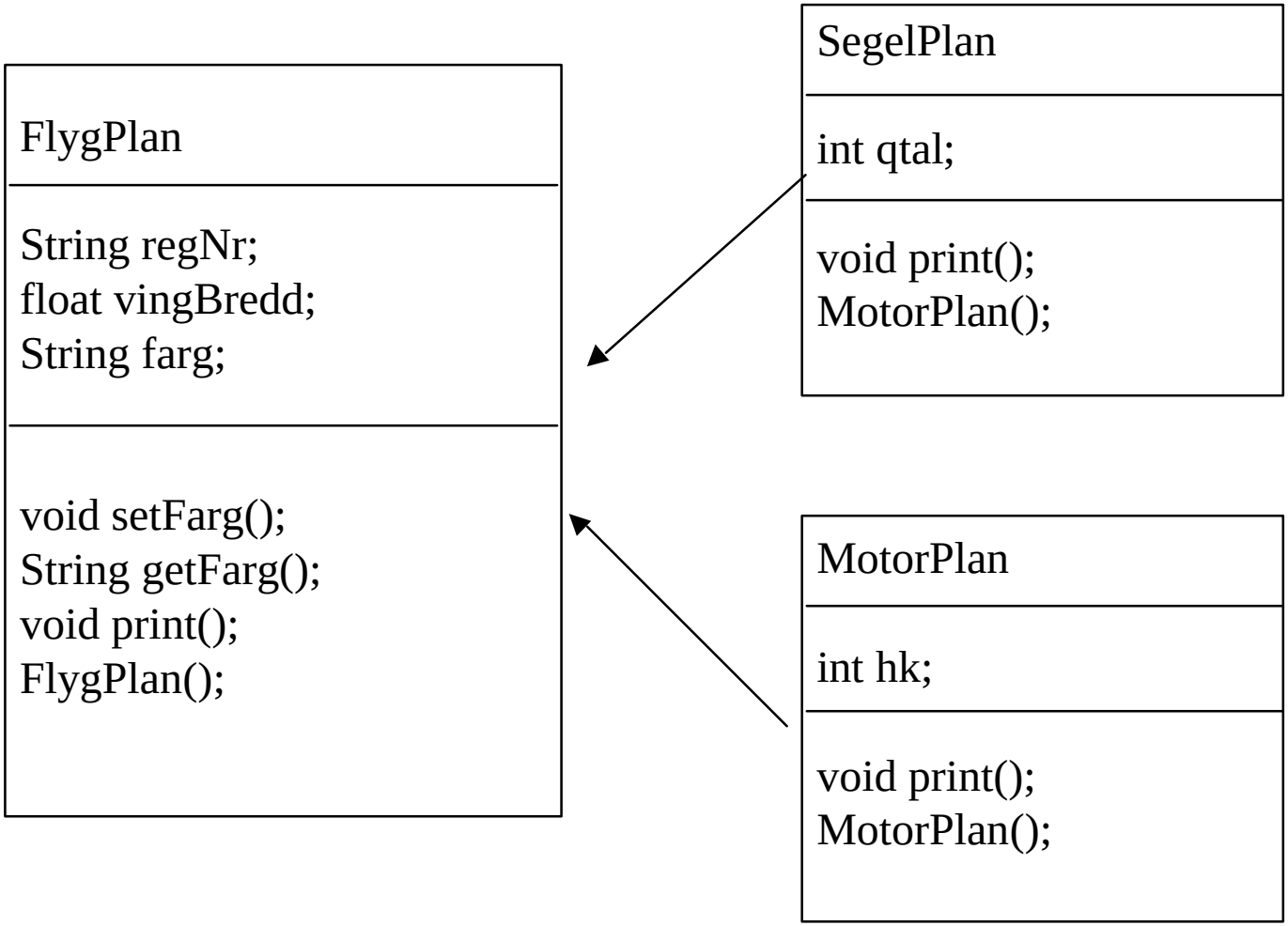
I regel behövs **två** metoder för varje attribut,

- För att förändra värdet.
- För att hämta värdet.

Vi ska nu utöka klassen `FlygPlan` med två metoder:

- **Förändra färg** på ett `FlygPlan`.
- **Hämta färg** hos ett `FlygPlan`, för att kunna skriva ut den.

Klassdiagrammet



Implementation av klassen FlygPlan

```
abstract class FlygPlan{

   protected String regNr;
   protected float vingBredd;
   protected String farg;

   public void setFarg(String pfarg){
      farg=pfarg;
   }// metod setFarg för att förändra färgen hos ett objekt

   public String getFarg(){
      return farg;
   }// metod getFarg för att hämta  färgen hos ett objekt

   public void print(){
      System.out.print("regnr=" +regNr+"bredd="+ vingBredd + farg);
   }// metod print

   public FlygPlan ( String pregNr, float pvingBredd,String pfarg){
      regNr=pregNr;
      vingBredd= pvingBredd;
      farg=pfarg;
   }//Konstruktor

} // klassen FlygPlan
```

Kan ersättas med:

```
getFarg()
```

SegelPlan

```
class SegelPlan extends FlygPlan
{
   private int qtal;

   public void print() {
      super.print();
      System.out.println("qtal=" + qtal);
   }// metoden print

public SegelPlan(String pregNr,float pvingBredd,String pfarg,int pqtal)
   {
      super(pregNr,pvingBredd,pfarg);
      qtal=pqtal;
   }//  konstruktor SegelPlan

}// klassen SegelPlan
```

MotorPlan

```
class MotorPlan extends FlygPlan {

   private int motorHk;

   public void print() {
      super.print();
      System.out.println("motorstyrka=" + motorHk);
   }// metoden print

   public MotorPlan(String pregNr,float pvingBredd,String pfarg,int hk)
   {
      super(pregNr,pvingBredd,pfarg);
      motorHk=hk;
   }// konstruktor MotorPlan

}// klassen MotorPlan
```

Testprogrammet

```
class PlanTest3 {
   public static void main (String[] args){

      SegelPlan s1=new SegelPlan("SE-123",12.4f,"vit",22);
      MotorPlan m1=new MotorPlan("SE-234",8.2f,"röd",300);

      s1.print();
      m1.print();

      System.out.println("m1:s färg=" + m1.getFarg());

      m1.setFarg("svart"); // ändra till svart färg
      System.out.println("nu är m1:s färg ändrad till ” +m1.getFarg());
   }// metod main
}// klass PlanTest3
```

Detta skrivs ut vid exekvering:

```
regnr=SE-123bredd=12.4vitqtal=22
regnr=SE-234bredd=8.2rödmotorstyrka=300
m1:s färg=röd
nu är m1:s färg ändrad till svart
```

Variabler och metoder igen


```
class Paket{
   private float langd;
   private float hojd;
   private float bredd;

   public float volym(){
      return hojd*langd*bredd;
//    return hojd*basyta(); // alternativ lösning
   }// metoden volym

   public float basyta(){
      return langd*bredd;
   }// metoden basyta

   public Paket(float l, float h, float b){
      langd=l;   hojd=h;  bredd=b;
   }// konstruktor för klassen Paket
}// klassen Paket
```

Metoder variabler forts.

Här är ett testprogram till klassen Paket från förra bilden

```
class Testa{

   public static void main( String[] args) {

      Paket julklapp=new Paket(10.2f,23.4f,4.5f);

      System.out.println("Volym= " +  julklapp.volym());
      System.out.println("Basyta=" + julklapp.basyta());

   }// metoden main

}// klassen Testa
```

Lägg märke till att för flyttalskonstanter (float) måste man sätta ett f efter (4.5f) annars antas den vara av typ double

Summering

- Arv
- Superklass, subklass
- Abstrakt klass, abstrakt metod
- super
- överskuggning av metod
- polymorfism
- dynamisk bindning