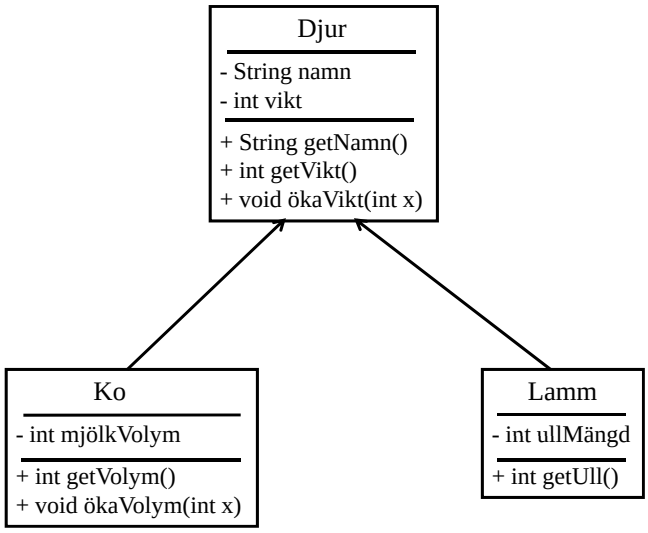
OOP Objekt-orienterad programmering

Föreläsning 10

Arv och klasshierarkier
Polymorfism

Stefan Möller



Stefan Möller

extends

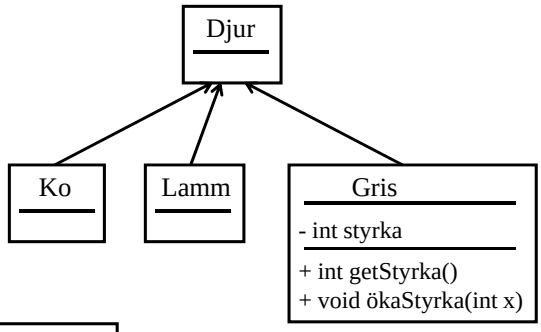
För att tala om vilken klass man ärver från används **extends**

```
class Djur{
   private String namn;
   private int vikt;
   public String getNamn(){
      return namn;
   }
   public int getVikt(){
      return vikt;
   }
   public void ökaVikt(int x){
      vikt+=x;
   }
}
```

```
class Ko extends Djur{
   private int mjölkVolym;
   public int getVolym(){
      return mjölkVolym;
   }
   public void ökaVolym(int x){
      mjölkVolym+=x;
   }
}
```

```
class Lamm extends Djur{
   private int ullMängd;
   public int getUll(){
      return ullMängd;
   }
}
```

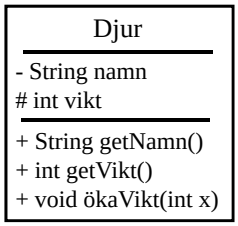

Lätt att lägga till nya klasser




```
class Gris extends Djur{
   private int styrka;
   public int getStyrka(){
      return styrka;
   }
   public void ökaStyrka(int x){
      styrka+=x;
   }
}
```

Synlighetsmodifieraren protected

Anger att ett attribut kan användas i subklasser till denna klass.




```
class Djur{
   private String namn;
   protected int vikt;
   public String getNamn(){
      return namn;
   }
   public int getVikt(){
      return vikt;
   }
   public void ökaVikt(int x){
      vikt+=x;
   }
}
```


Konstruktorer

Subklassernas konstruktorer MÅSTE anropa superklassens konstruktor. Görs med:

super(...);

Måste stå överst i konstruktorn.

```
class Djur{
   private String namn;
   protected int vikt;
   public Djur(String na, int v){
      namn=na;
      vikt=v;
   }
}
```

```
class Ko extends Djur{
   private int mjölkVolym;
   public Ko(String na, int v, int mjö){
      super(na, v);
      mjölkVolym=mjö;
   }
}
```

```
class Lamm extends Djur{
   private int ullMängd;
   public Lamm(String na, int ull){
      super(na, 40);
      ullMängd=ull;
   }
}
```

Ett Lamm har alltid startvikten 40

Default-super-anrop

Om man utelämnar super(...) så lägger kompilatorn in ett default-anrop utan argument:

super();

För att det skall fungera MÅSTE superklassen ha en argumentlös konstruktor.

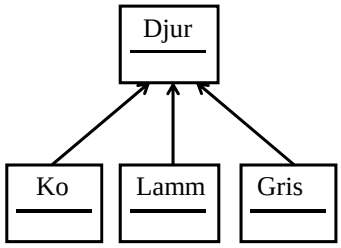
```
class Djur{
   private String namn;
   protected int vikt;
   public Djur(String na, int v){
      namn=na;
      vikt=v;
   }
}
```

```
class Gris extends Djur{
   private int styrka;
   public Gris(int sty){

      styrka=sty;
   }
}
```


Man skapar (oftast) endast objekt av klasserna "längst ner" i hierarkin.

```
Ko k=new Ko("Rosa", 117, 3);
Lamm l=new Lamm("Bäbä 1", 37);
Gris g=new Gris(88);
```



En referens av typen djur kan referera till vilken subklass som helst.

```
Djur d1, d2, d3;
d1=new Ko("Blenda", 195, 4);
d2=new Lamm("Bäbä 2", 43);
d3=new Gris(107);
```

Praktiskt om man t.ex. vill ha en "blandad" samling:

```
Djur[] allaDjuren=new Djur[25];
ArrayList<Djur> djuren = new ArrayList<Djur>();
```

Åtkomst

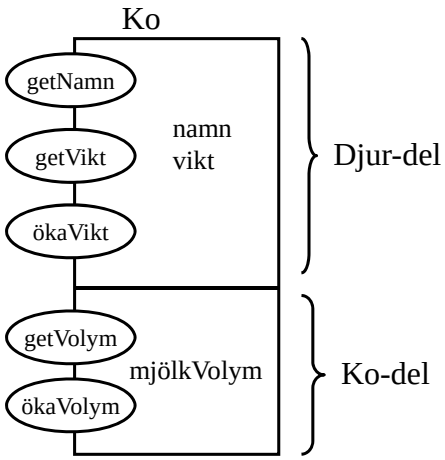
Objekt som skapas består av flera olika "logiska" delar. Det skapade objektet är "alltihop" men kan betraktas som flera hopsatta delar.

Vad man kommer åt hos objektet beror på hur referensen är deklarerad.

```
Ko k=new Ko("Rosa", 112, 4);
Djur d=new Ko("Mu", 174, 5);
```

Via k kommer man åt allt.

Via d kommer man endast åt det som finns i Djur-delen.



Cast & instanceof

Djur-pekaren **pekar på hela** objektet. Kan man komma åt Ko-delen?
Ja - genom att kvalificera (eller cast'a) referensen:

```
((Ko)d).ökaVolym(2);
```

Detta fungerar endast om det verkligen ÄR ett Ko-objekt.
Skulle det råka vara ett Lamm-objekt blir det exekveringsfel.

Man bör kontrollera att det verkligen är ett Ko-objekt innan man gör castingen. Att kolla vilket sorts objekt det är görs med instanceof.

```
if (d instanceof Ko)
   ((Ko)d).ökaVolym(2);
```

Att sätta referenser till objekt:

`d = k;` Detta går bra
`k = d;` Går ej, måste cast'a: `k = (Ko)d;`

Vi antar nu att alla djur har ett visst värde.
Värdet räknas ut på olika sätt beroende på djursort:

- Ko - värdet är vikten * mjölkvolymen
- Lamm - värdet är kvadraten på ullmängden
- Gris - värdet är vikt * 3 + styrkan

Eftersom värdet är beroende av andra attribut och dessa andra attribut kan ändras implementerar vi värdet som en metod.

I alla Djur-subklasserna skall vi ha metoden:

```
public int värde(){
   //räknas ut olika beroende på vilken djursort
}
```



```
class Ko extends Djur{
   private int mjölkVolym;
   public int värde(){
      return vikt*mjölkVolym;
   }
}
```

Attributet vikt kan användas eftersom det är deklarerat som protected i Djur

```
class Lamm extends Djur{
   private int ullMängd;
   public int värde(){
      return ullMängd*ullMängd;
   }
}
```

```
class Gris extends Djur{
   private int styrka;
   public int värde(){
      return vikt*3+styrka;
   }
}
```

Åtkomst från t.ex. en Ko-referens:

```
Ko k;
int summa=k.värde();
```

Från en Djur-referens:

```
Djur d;
if (d instanceof Ko)
   summa=((Ko)d).värde();
```

Anta att vi vill ha en metod som får en ArrayList med blandade djur som argument och som returnerar sammanlagda värdet av dessa djur.

```
int summaVärde(ArrayList<Djur> djuren){
   int summa = 0;
   for (Djur aktuell : djuren)
      if (aktuell instanceof Ko)
         summa += ((Ko)aktuell).värde();
      else if (aktuell instanceof Lamm)
         summa += ((Lamm)aktuell).värde();
      else if (aktuell instanceof Gris)
         summa += ((Gris)aktuell).värde();

   return summa;
}
```

Så här går bra att göra. Dock lite omständigt. Vi vet ju att **alla** djur har ett värde. Borde kunna göras enklare...
Speciellt dåligt om vi lägger till ett nytt sorts djur, t ex Häst. Då måste metoden summaVärde ändras.

abstract

Vi vill markera i Djur-delen att det finns en metod värde i subclasserna.
Detta för att kunna komma åt metoden värde från en Djur-referens.

```
class Djur{
    private String namn;
    protected int vikt;

    abstract public int värde();

}
```

Så fort det finns minst en abstract metod i en klass KAN man inte skapa en instans av klassen. Abstract-deklarationen talar om att metoden finns definierad i en subklass till denna klass. Om man har en abstract metod måste dessutom hela klassen vara abstract, dvs vi måste skriva:

```
abstract class Djur{
```

Man kan ange att en klass är abstract utan att ha abstracta metoder.
Detta för att förhindra instansiering (man kan då ej göra new).

Polymorfism - dynamisk bindning

Har vi deklarerat värde() som abstract i klass Djur kan summaVärde skrivas om:

```
int summaVärde(ArrayList<Djur> djuren){
   int summa = 0;
   for (Djur aktuell : djuren)
        summa += aktuell.värde();

   return summa;
}
```

Nu kommer "rätt" värde-metod att bli anropad. Detta kallas dynamisk bindning eftersom kompilatorn inte kan veta vilken metod som skall anropas utan det beror på vad som råkar ligga på respektive ställe i ArrayListan. Under körning binds alltså anropet till den värde-metod som är aktuell.

Denna mekanism kallas också polymorfism.

Skulle vi nu vilja ha ett annat sorts djur, t ex en häst som har attributet antal vunna lopp och värdet antalVinster * vikt kan vi enkelt lägga till denna klass:

```
class Horse extends Djur{
   private int antalVinster;
   public int värde(){
      return antalVinster * vikt;
   }
}
```

OBS - måste även ha en konstruktor ...

Nu behöver inte metoden summaVärde ändras - den fungerar automatiskt även med objekt av typen Horse.

Att på detta sätt kunna lägga till nya klasser utan att behöva ändra i "gammal" kod är mycket kraftfullt och en av (många) fördelar med objektorientering.

Att det går beror på mekanismerna arv och polymorfism.

Anta att vi vill lägga till att alla djur kan äta. Det äter en viss vikt mat och djurets vikt ökar med halva vikten av det den ätit. Detta gäller alla djur utom grisar. Grisens vikt ökar med hela vikten av det den ätit + att grisens styrka ökar med 1.

I klass Djur lägger vi till metoden äta enligt:

```
public void äta(int mat){
   vikt += mat/2;
}
```

I klass Gris lägger vi in en annan äta nämligen:

```
public void äta(int mat){
   vikt += mat;
   styrka++;
}
```

Nu kan vi direkt med en Djur-referens anropa metoden

```
Djur d;  //Kan referera till Ko, Lamm, Gris eller Horse
d.äta(4);
```

Rätt äta-metod anropas. Att på detta sätt i Gris skriva en egen äta som omdefinierar äta hos Djur kallas **överskuggning**. Vi överskuggar default-äta hos Djur och den dynamiska bindningen gör att rätt äta anropas.

Polymorfism eller dynamisk bindning kan alltså åstadkommas på två olika sätt:

1. I superklassen deklareras en abstract metod som sedan måste implementeras i alla subclasser. Detta gjordes med värde.

2. I superklassen implementeras en "default"-metod och denna gäller för alla subklasser som INTE har en egen metod som överskuggar superklassens. Detta gjordes med äta.

I båda fallen får vi det beteende vi vill ha, nämligen att metoderna kan anropas från en Djur-referens och rätt metod kommer alltid att bli anropad.

Vi kan dessutom lätt utöka modellen med nya typer av Djur utan att behöva ändra i det som fanns tidigare.