Kapitel 3. Husdjur & Fisk – Javaklasser

Translation by Leif Lourié

Java program består av *klasser* som beskriver saker (objekt) som finns på riktigt. Även om det finns många olika sätt att skriva program på, så är de flesta överens om att *objekt-orienterad* programmering är det bästa sättet. Detta betyder att duktiga programmerare börjar med att bestämma vilka objekt som ska finnas i programmet och vilka Javaklasser som ska beskriva dessa. Först därefter sätter de igång med att skriva Java kod.

Klasser och Objekt

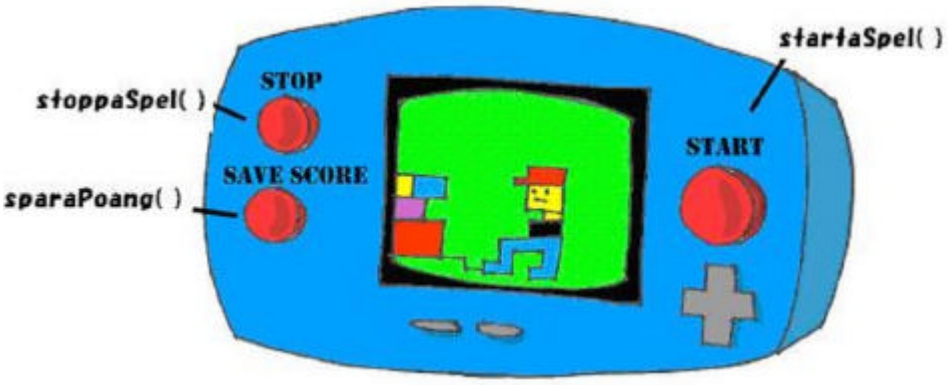
Klasser i Java kan ha *metoder* och *egenskaper*.

Metoder beskriver funktioner som klassen kan utföra.

Egenskaper beskriver innehållet i klassen.

Låt oss skapa en klass kallad `DataSpel`. Denna klass kan ha flera metoder som beskriver *vad objekt av denna klass kan göra:*

starta spelet, stoppa spelet, spara poäng och så vidare. Denna klass kan även ha egenskaper: pris, skärm, färg, antal handenheter, etc.



I Java skulle denna klass kunna se ut så här:

```
class DataSpel {
  String farg;
  int price;

  void startaSpel() {
  }
  void stoppaSpel() {
  }
  void sparaPoang(String spelarensNamn, int poang) {
  }
}
```

Vår klass `DataSpel` borde likna andra klasser som beskriver dataspel – alla har skärmar med olika storlekar och färg, alla utför liknande funktioner och alla kostar pengar.

Vi kan vara mer exakta och skapa en annan Javaklass kallad `GameBoyAdvance`. Den är också av typen dataspel, men har en del egenskaper som är speciella för modellen GameBoy Advance, till exempel kassettsort.

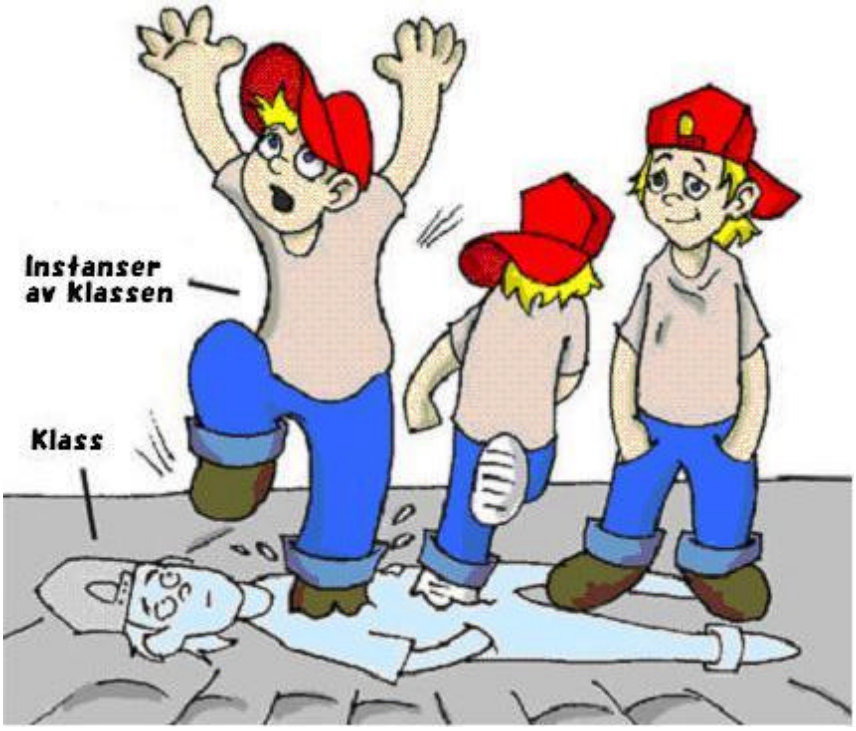
```
class GameBoyAdvance {
  String kasettSort;
  int skarmStorlek;

  void startaSpel() {

  }
  void stoppaSpel() {

  }
}
```

I denna klass, `GameBoyAdvance`, finns det två egenskaper – `kasettSort` och `skarmStorlek` och två metoder – `startaSpel()` och `stoppaSpel()`. Men dessa metoder kan inte utföra något ännu eftersom de inte innehåller någon Java kod.

Förutom ordet *klass*, måste du vänja dig vid att använda ordet *objekt*.

Meningen "att skapa en instans av en klass" betyder att skapa en kopia i datorns minne, ett *objekt*, som följer beskrivningen för klassen.

En konstruktionsritning för en GameBoy Advance beskriver den egentliga spelmaskinen på samma sätt som en Javaklass beskriver den instans av objektet som finns i datorns minne. Processen att bygga spelmaskiner i fabriken baserat på ritningen liknar processen att skapa objektinstanser från klassen `GameBoy` i Java.



I de flesta fall kan man inte använda en Javaklass förrän man har skapat en instans av denna. På samma sätt som speltillverkaren tillverkar tusentals spelmaskiner utifrån samma beskrivning. Fastän varje kopia är en instans av samma klass så kan de ha olika *egenskaper* (olika *värden* i sina medlemsvariabler) – en del är blåa medan andra är silverfärgade och så vidare. Med andra ord, ett program kan skapa *flera instanser* av `GameBoyAdvance`.

Datatyper

Variabler används för att beskriva klassens egenskaper, parametrar till metoder eller för tillfällig lagring av information. Variabler måste deklareras innan du kan använda dem.

Låt oss titta på den matematiska formeln y=x+2. I Java så måste du deklarera variablerna x och y som någon form av numerisk *datatyp*, exempelvis integer eller `double`:

```
int x;
int y;
```

De följande två raderna visar hur du kan tilldela *värden* till dessa variabler. Om ditt program tilldelar värdet fem till variabeln x, så kommer det att leda till att variablen y blir sju:

```
x=5;
y=x+2;
```

I Java kan du även ändra värdet i en variabel på ett lite speciellt sätt. I de följande två raderna deklareras först variabeln y som en integer och sätts till värdet fem varefter denna ändras från fem till sex:

```
int y=5;
y++;
```

De två plustecknen betyder att värdet i variabeln y ökas med ett.

Även i de följande två kodraderna kommer värdet i variabeln `minPoang` att vara sex:

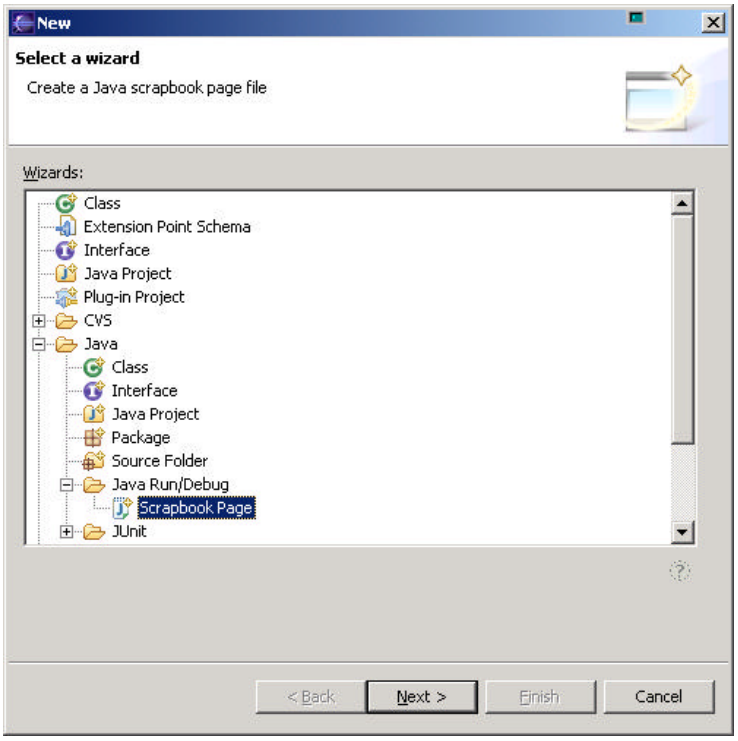
```
int minPoang=5;
minPoang=minPoang+1;
```

Du kan även använda multiplikation, division och subtraktion på samma sätt. Titta på den följande koden:

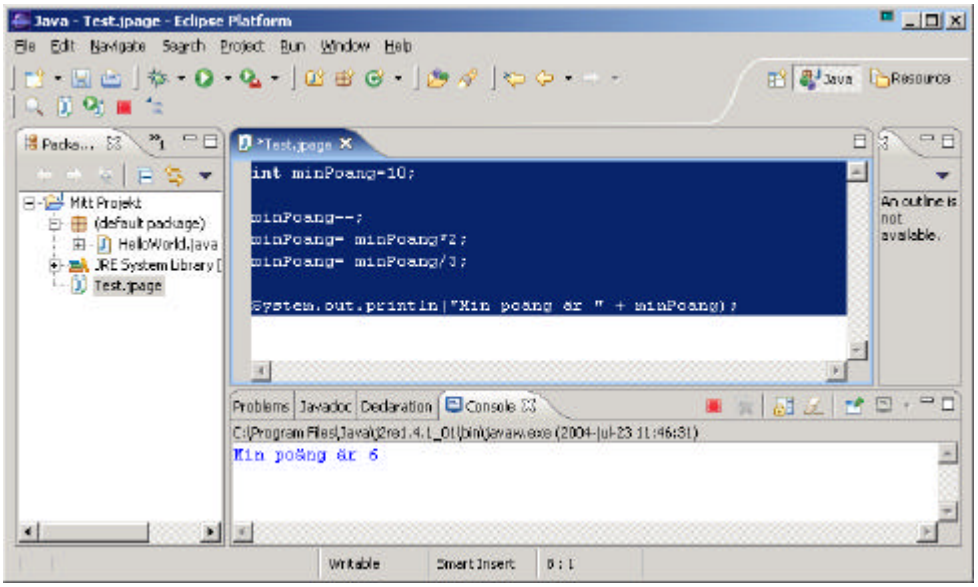
```
int minPoang=10;

minPoang--;
minPoang=minPoang*2;
minPoang=minPoang/3;
```

`System.out.println("Min poäng är " + minPoang)`Vad den här koden skriver ut? Eclipse har en bra funktion kallad *scrapbook*

som hjälper dig att testa kodsnuttar (som den här ovanför) utan att du behöver skriva ett fullständigt program. Välj menyn *File*, *New*, *Other* och välj *Scrapbook Page* enligt nedan:



Skriv ordet `Test` som namn på din scrapbook fil.
Skriv sedan in de fem raderna som ändrar `minPoang` i scrapbook, markera dem och klicka på det lilla förstoringsglaset som finns i verktygsraden.



För att se resultatet av beräkningen, klicka på fliken som heter console, längst ner på skärmen:

```
Min poäng är 6
```

I detta exempel är indatat till metoden `println()` sammansatt av två delar – texten "Min poäng är " och värdet för variabeln `minPoang`, som är sex. Att skapa en `Sträng` genom att sätta samman flera delar kallas *konkatenering* . Även om `minPoang` är ett nummer, så är Java tillräckligt smart för att göra om värdet till en `Sträng`, och sätta samman den med texten *Min poäng är*.

Några exempel på hur du kan ändra värdet på variabler:

```
minPoang=minPoang*2; är detsamma som minPoang*=2;
minPoang=minPoang+2; är detsamma som minPoang+=2;
minPoang=minPoang-2; är detsamma som minPoang-=2;
minPoang=minPoang/2; är detsamma som minPoang/=2;
```

Det finns åtta enkla, eller *primitiva* datatyper i Java och du måste bestämma vilken du ska använda beroende på vilken sorts information och vilken storlek på informationen du vill lagra i dina variabler:

- Fyra datatyper för att lagra numeriska värden – `byte`, `short`, `int`, och `long`.

- Två datatyper för att lagra decimala värden – `float` och `double`.

- En datatyp för att lagra enstaka tecken – `char`.

- En *logisk* datatyp kallad `boolean` som bara tillåter två möjliga värden: `true` eller `false`. (`sant` eller `falskt`)

Du kan sätta ett startvärde i variablen redan när du deklarerar den. Detta kallas att *initiera variabeln*:

```
char betyg = 'A';
int antalStolar = 12;
boolean spelaMusik = false;
double inkomst = 23863494965745.78;
float pris = 12.50f;
long totaltAntalBilar =4637283648392l;
```

I slutet på de två sista raderna ser du ett `f` som betyder float och ett `l` som betyder long.

Om du inte initierar variablerna så kommer Java att göra det ändå genom att sätta numeriska variabler till noll, boolska variabler till false och char variabler får specialvärdet ‘\u0000’.

Om du använder det speciella nyckelordet `final` när du deklarerar en variabel så betyder det att du bara kan sätta ett värde i denna variabel en gång och sedan aldrig ändra den. Denna typ av variabel kallas ofta för *konstant*. I Java rekommenderar vi att du använder stora bokstäver när du namnsätter konstanter:

```
final String HUVUDSTAD="Stockholm";
```

Förutom primitiva datatyper så kan du använda Javaklasser för att deklarera variabler. Varje primitiv datatyp har en motsvarande *wrapper* klass, till exempel `Integer`, `Double`, `Boolean`, etc. Dessa klasser innehåller metoder för att konvertera data från en datatyp till en annan.

Den primitiva datatypen `char` används ju för att lagra ett tecken, men Java har även klassen `String` som används för att arbeta med textsträngar, till exempel:

```
String efterNamn="Andersson";
```

I Java så kan ett variabelnamn inte börja med en siffra och inte innehålla mellanslag. Dessutom så kan våra speciella svenska tecken inte användas i Java kod. Så Å, Ä, Ö kan inte användas i variabelnamn, metodnamn eller klassnamn. Däremot kan kommentarer och textsträngar innehålla svenska tecken.

En `bit` är den minsta delen data som kan lagras i datorns minne. Den kan innehålla antingen 1 eller 0.

En `byte` består av åtta bitar.

En `char`, i Java, använder två byte i minnet.

En `int` och en `float` använder fyra byte vardera.

En `long` och en `double` använder åtta byte vardera.

Numeriska datatyper som använder fler byte kan lagra större tal.

1 kilobyte (KB) består av 1024 bytes

1 megabyte (MB) består av 1024 kilobyte

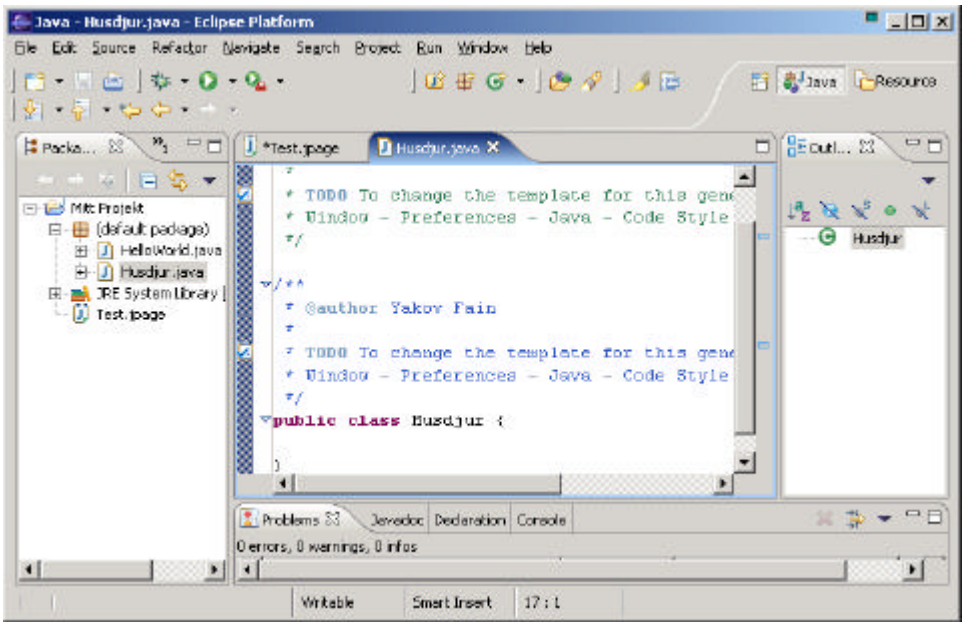
1 gigabyte (GB) består av 1024 megabyte

Att skapa ett Husdjur

Låt oss skapa en klass för Husdjur. Först ska vi bestämma vad husdjuret ska kunna göra. Vad sägs om att äta, sova och prata? Vi kommer att programmera in detta i metoderna för klassen `Husdjur`. Vi ska även ge vårt husdjur följande egenskaper : ålder, längd, vikt och färg.

Börja med att skapa en ny Javaklass kallad `Husdjur` i *My First Project* som finns beskrivet i Kapitel 2, men klicka inte i boxen för att skapa metoden `main()`.

Din skärm borde se ut ungefär som nedan:



Nu är vi redo att deklarera attribut och metoder i vår klass, `Husdjur`. Javaklasser och metoder har sin kod mellan måsvingar. Varje måsvinge i början av en klass eller metod måste ha en motsvarande måsvinge i slutet:

```
class Husdjur{
}
```

När vi ska deklarera attributen för vår klass så måste vi bestämma vilka datatyper som vi ska använda. Jag föreslår en `int` för ålder, `float` för vikt och längd samt `String` för husdjurets färg.

```
class Husdjur{
   int alder;
   float vikt;
   float langd;
   String farg;
}
```

Om du kommer ihåg, så kan vi inte använda våra speciella svenska tecken i Javakod. I koden står det därför *alder* istället för ålder, *langd* istället för längd och *farg* istället för färg.

Nästa steg är att lägga till metoder till klassen. Innan du deklarerar metoder måste du bestämma om de ska ha några inparametrar och om de ska returnera någonting:

- Metoden `sova()` ska bara skriva ut meddelandet *God natt, vi ses imorgon* – den behöver inte några inparametrar och kommer inte att returnera någonting.

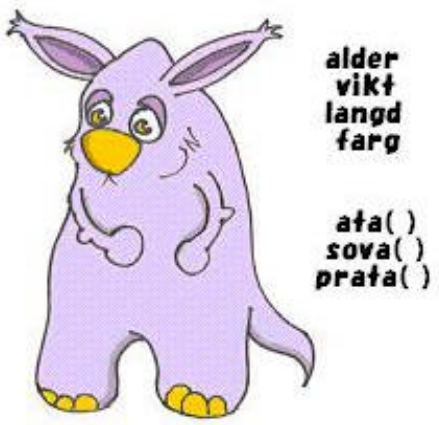
- Detsamma gäller för metoden för att äta – `ata()`. Den ska skriva ut meddelandet *Jag är så hungrig... Mata mig!*

- Metoden `prata()` ska också skriva ut ett meddelande, men husdjuret kommer att "säga" (skriva) det ord eller läte som vi talar om för det. Vi kommer att skicka med det ordet till metoden `prata()` som en *inparameter*. Metoden ska bygga ihop lätet med denna inparameter och returnera detta till det anropande programmet.

Så nu ser klassen `Husdjur` ut så här:

```
public class Husdjur {
 int alder;
 float vikt;
 float langd;
 String farg;

 public void sova(){
        System.out.println(
                    "God natt, vi ses imorgon");
 }

 public void ata(){
  System.out.println(
  "Jag är så hungrig... Mata mig!");
 }

 public String prata(String ettOrd){
      String svar = "OK!! OK!! " + ettOrd;
            return svar;
 }
}
```

Den här klassen beskriver ett gulligt djur som finns i den verkliga världen:



Låt oss titta närmare på *signaturen* för metoden `sova()`:

```
public void sova()
```

Den säger oss att denna metod kan anropas från vilket annat Java program som helst (`public`) och att metoden inte returnerar något data (`void`). De två paranteserna utan något emellan betyder att metoden inte har några parametrar , eftersom den inte behöver något extra data – den skriver alltid samma text.

Signaturen för metoden `prata()` ser ut så här:

```
public String prata(String ettOrd)
```

Även den här metoden kan anropas från andra Java program men kommer att returnera en text eftersom nyckelordet `String` finns framför metodnamnet. Dessutom förväntar sig metoden att få data i form av text i parametern `String ettOrd`.

Hur bestämmer man om en metod ska returnera ett värde eller inte? Om metoden utför någon form av förändring av datat och behöver skicka tillbaka någonting till den som anropar så måste den returnera detta. Men klassen `Husdjur` har ju ingen som anropar! Det är riktigt, så låt oss då skapa en ny klass `Husse` (eller `Matte`). Denna klass har en metod `main()` som innehåller

kod för att prata med klassen `Husdjur`. Använd Eclipse och skapa ännu en klass, kallad `Husse`, och denna gång klicka för valet att skapa metodsen `main()`. Kom ihåg, utan denna metod kan du inte *köra* klassen som ett program. Ändra koden som genereras av Eclipse så att den ser ut så här:

```
public class Husse {

  public static void main(String[] args) {

   Husdjur mittHusdjur = new Husdjur();

   String reaktion;

   mittHusdjur.ata();
   reaktion = mittHusdjur.prata("Tweet!! Tweet!!");
   System.out.println(reaktion);

   mittHusdjur.sova();

  }
}
```

Glöm inte att slå *Ctrl-S* för att spara och kompilera klassen!
För att köra klassen `Husse` klicka på Eclipse menyn *Run*, *Run…* , *New* och skriv namnet på klassen: `Husse`. Klicka på knappen *Run* och programmet kommer att skriva följande:

```
Jag är så hungrig... Mata mig!
OK!! OK!! Tweet!! Tweet!!
God natt, vi ses imorgon
```

`Husse` är den *anropande klassen* och startar med att skapa en *instans* av `Husdjur` genom att deklarera variabeln `mittHusdjur` och använder sedan Java kommandot `new`:

```
Husdjur mittHusdjur = new Husdjur();
```

Denna rad deklarerar en variabel av typen `Husdjur` (precis... du kan använda alla klasser i Java som vilken datatyp som helst). Nu vet variabeln `mittHusdjur` var instansen av `Husdjur` finns i datorns minne och du kan använda variabeln för att anropa metoderna som finns i klassen, till exempel:

```
mittHusdjur.ata();
```

Om metoden returnerar ett värde måste du anropa den på ett litet annorlunda sätt. Deklarera en variabel som är av samma typ som det värde som returneras och anropa sedan metoden på följande sätt:

```
String reaktion;

reaktion = mittHusdjur.prata("Tweet!! Tweet!!");
```

Efter de här kodraderna har det returnerade värdet lagrats i variabeln `reaktion` och titta gärna på vad som finns där:

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

Arv – en Fisk är också ett Husdjur

Vår klass `Husdjur` ska hjälpa oss att förstå ännu en viktig del av Java kallat *arv*. I det verkliga livet ärver varje människa olika egenskaper från sina föräldrar. På samma sätt kan man i Java skapa en ny klass baserad på en annan.

Klassen `Husdjur` har beteende och egenskaper som är desamma för många husdjur – de äter, sover, en del av dem låter, de har olika färg och så vidare. Å andra sidan, husdjur är olika – hundar skäller, fiskar simmar och låter inte, papegojor talar bättre än hundar. Men alla äter, sover, har vikt och längd. Därför är det enklare att skapa klassen `Fisk` och låta den *ärva* gemensamma beteenden och egenskaper från klassen `Husdjur`, istället för att skapa `Hund`, `Papegoja` eller `Fisk` från grunden varje gång.

Det speciella nyckelordet `extends` är det som fixar arvet:

```
class Fisk extends Husdjur{

}
```

Man kan säga att `Fisk` är en *subklass* av klassen `Husdjur` och att `Husdjur` är en *superklass* för klassen `Fisk`. Med andra ord så använder du klassen `Husdjur` som mall för att skapa klassen `Fisk`.

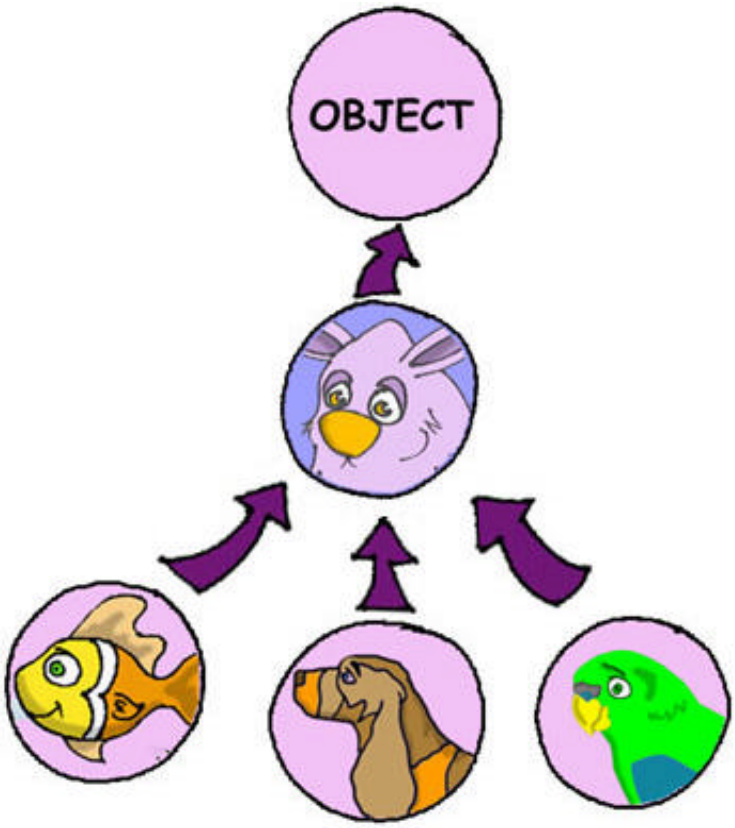
Även om du inte gör något mer med klassen `Fisk` så kan du ändå använda alla metoder och egenskaper som har ärvts från klassen `Husdjur`. Kolla här:

```
Fisk minLillaFisk = new Fisk();
```

```
minLillaFisk.sova();
```

Fast vi inte har deklarerat några metoder i klassen `Fisk` så kan vi använda metoden `sova()` i dess superklass!

Att skapa subklasser i Eclipse är superenkelt! Välj menyn *File*, *New*, *Class*, och skriv `Fisk` som klassnamn. Byt ut `java.lang.Object` i fältet "superclass" mot `Husdjur`.



Men kom ihåg att syftet med att skapa en subklass från `Husdjur` är att lägga till det som är speciellt för fiskar och återanvända den koden som är gemensam för alla husdjur.

Det är dags att avslöja en hemlighet - alla klasser i Java ärver från en super-duper klass kallad Object, oavsett om du använder ordet extends eller inte.

Men Javaklasser kan inte ha två föräldrar.
Om detsamma gällde för människor så hade barnen inte varit subklasser av sina föräldrar.

Alla pojkar hade ärvt alla egenskaper från Adam och alla flickor hade varit subklasser av Eva. :)

Alla husdjur kan inte dyka men fiskar är jätteduktiga på det.
Låt oss lägga till metoden `dyka()` i klassen `Fisk`.

```
public class Fisk extends Husdjur {

   int djup=0;

   public int dyka(int hurDjupt){
      djup = djup + hurDjupt;
      System.out.println("Dyker ner " + hurDjupt +
                                          " meter");
      System.out.println("Jag är " + djup +
                                  " meter under ytan ");
      return djup;
   }
}
```

Metoden `dyka()` har en *parameter* `hurDjupt` som anger hur många meter till som fisken ska dyka ner. Vi har även deklarerat en klassvariabel, `djup`, som innehåller information om djupet som fisken befinner sig på. Denna variabel kommer att uppdateras varje gång du anropar metoden `dyka()`. Dessutom returnerar metoden värdet på variabeln `djup` till det anropande programmet.

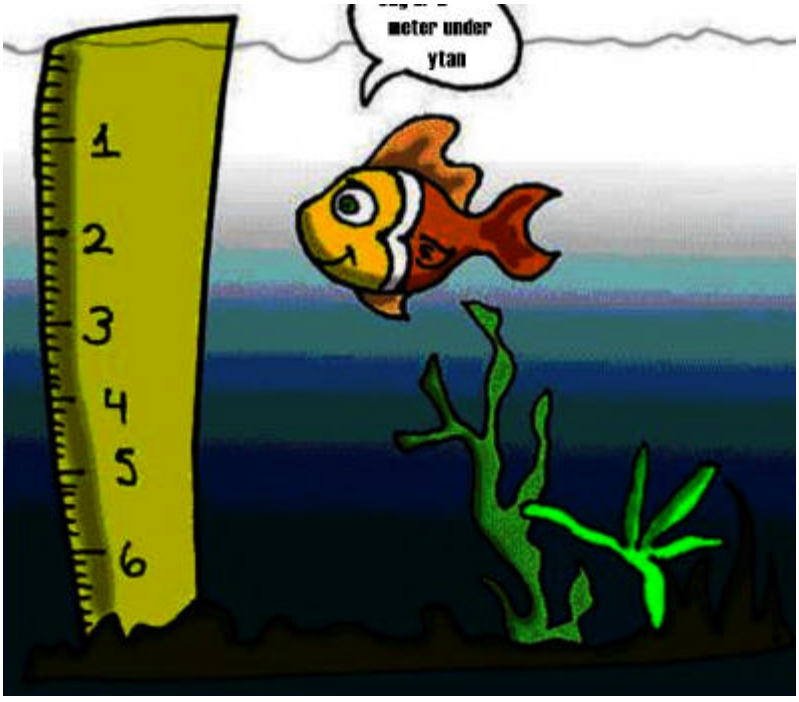
Skapa nu en ny klass, `FiskMatte` som ser ut så här:

```
public class FiskMatte {

	public static void main(String[] args) {

		Fisk minFisk = new Fisk();

		minFisk.dyka(2);
		minFisk.dyka(3);

		minFisk.sova();
	}
}
```

Metoden `main()` instansierar objektet Fisk och anropar metoden `dyka()` två gånger med olika värden. Därefter anropas metoden `sova()`. När du kör programmet `FiskMatte` skrivs följande meddelanden ut:

```
Dyker ner 2 meter
Jag är 2 meter under ytan
Dyker ner 3 meter
Jag är 5 meter under ytan
God natt, vi ses imorgon
```

Har du tänkt på att förutom metoden `dyka()` som finns i klassen `Fisk` så anropar `FiskMatte` även metoden `sova()` som finns i superklassen `Husdjur`? Det är det som är poängen med arv – du behöver inte kopiera koden som finns i klassen `Husdjur` – använd bara ordet `extends` så kan klassen `Fisk` använda `Husdjurets` metoder!



En sak till, även om metoden `dyka()` returnerar värdet för `djup` så använder inte vår `FiskMatte` detta. Det är helt okej! Vår `FiskMatte` behöver inte värdet men det kan finnas andra program som också använder `Fisk` och de kanske kommer att tycka det är viktigt. Tänk om klassen `FiskTrafikKontroll` skapades för att godkänna om fisken fick dyka till ett visst djup. Den klassen skulle behöva veta var fiskarna fanns på för djup för att kunna undvika traffikolyckor. ✍.

Överlagring av metoder

Som du vet så pratar inte fiskar (åtminstone inte så vi hör det). Men vår klass `Fisk` har ärvt metoden `prata()` från klassen `Husdjur`. Det betyder att du du skulle kunna skriva kod på det här viset:

```
minFisk.prata();
```

Hoppsan, vår fisk började prata… Om du inte vill att detta ska vara möjligt så måste klassen `Fisk` *överlagra* Husdjurets metod

`prata()`. Det gör du genom att i subklassen skapa en metod med exakt samma signatur som i superklassen. Då kommer subklassens metod att användas istället för den som finns i superklassen. Låt oss lägga till metoden `prata()` i klassen `Fisk`.

```
public String prata(String blaha){
  return "Men hörru! Fiskar kan ju inte prata.";
}
```

Lägg nu till följande metodanrop i klassen FiskMatte:

```
minFisk.prata("Hejsan");
```

När du kör programmet så kommer det att skriva ut:

```
Men hörru! Fiskar kan ju inte prata.
```

Detta visar att Husdjurets method `prata()` har *överlagrats*, eller med andra ord, ersatts.

Om en metod är deklarerad som `final`, så kan denna metod inte överlagras, till exempel:

```
final public void sova(){…}
```

Oboy! Vi har verkligen lärt oss en massa i det här kapitlet – nu är vi värda en rast!

Mera Läsning

1.Java Datatyper:
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html

2.Om Arv:
http://java.sun.com/docs/books/tutorial/java/concepts/inheritance.html T

Övning

1. Skapa en ny klass kallad `Bil` med följande metoder:

```
public void starta()
public void stanna()
public int gasa(int enStund)
```

Metoden `gasa()` ska returnera hur långt bilen har kört efter en stund. Använd denna formel för att räkna ut hur långt bilen kör:

```
antalKilometer = enStund * 60;
```

2. Skapa sedan en klass för den som äger bilen – `BilAgare` – som skapar en instans av `Bil` och anropar metoderna. Resultaten från dessa metodanrop måste skrivas ut med hjälp av `System.out.println()`.

Övning för smartskallar

Skapa en subklass av `Car` kallad `JamesBondCar` och överlagra metoden `drive()` i denna. Använd följande formel för att räkna ut sträckan:

```
antalKilometer = enStund * 180;
```

Var kreativ. Skriv ut lite roliga meddelanden!