Kort om klasser och objekt

En introduktion till GUI-programmering i Java

Klasser

En *klass* är en *mall* för hur man ska beskriva på något.
Antag att vi har en klass, **`Bil`**.
Den klassen innehåller en lista på *egenskaper* (*attribut, variabler*) som en bil har, till exempel

- `marke`
- `modell`
- `arsmodell`
- `farg`
- `korstracka`

En klass innehåller ofta dessutom ett antal *metoder*. En *metod* är något som beskriver ett skeende (något utförs, något beräknas, något förändras…)   I klassen **`Bil`** finns kanske metoderna

- `korFramat()`
- `backa()`
- `beraknaAndrahandsVarde()`
- `beraknaBensinAtgang()`

Egenskaperna beskriver hur en bil *ÄR*
Metoderna beskriver vad en bil *GÖR* eller vad som GÖRs med en bil.

I varje klass som är definierad i ett javaprogram finns en speciell metod som har samma namn som klassen.
Om vår klass `Bil` existerar i ett javaprogram så finns metoden

- `Bil()`

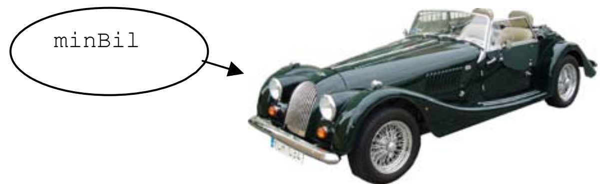
Den här speciella metoden kallas *konstruktor* och används när man skriver kod som skapar *objekt* ur klassen.
Nu ska vi lära oss vad ett objekt är:

Objekt

Jag beskriver min bil, och jag använder klassen **`Bil`** som mall:

`minBil`

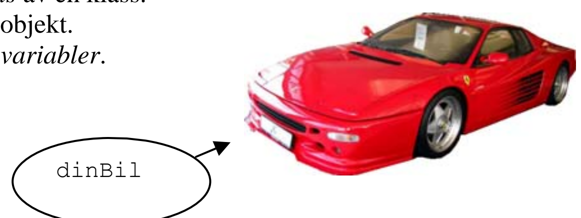
- `marke:      Morgan`
- `modell:     PlusFour`
- `arsmodell:  1969`
- `farg:       British Racing Green`
- `miltal:     7350`



- `korFramat(){` *här finns en beskrivning på hur bilen gör när den går framåt* `}`
- `backa()  {` *här finns en beskrivning på hur bilen gör när den backar* `}`
- `getAndrahandsVarde(){` *här beräknas andrahandvärdet* `}`
- `getBensinAtgang(){` *här beräknas bensinåtgång* `}`

Klassen `Bil` är en *mall* för hur jag ska beskriva en bil.
När jag ger egenskaperna *värden* och *definierar* vad metoderna innehåller så är det ju ett *speciellt exemplar* av en bil jag talar om. Ett sådant speciellt exemplar kallas för ett *objekt*. `minBil` är ett *objekt* ur klassen `Bil`.
I stället för ordet *objekt* ur en klass använder man ibland ordet *instans* av en klass.
När man *skapar* ett objekt säger man ibland att man *instansierar* ett objekt.
Egenskaperna (attributen, variablerna) i ett objekt kallas ofta *instansvariabler*.

Nu: Beskriv ännu ett objekt av klassen `Bil`. Beskriv *din* (dröm)bil!



Skapa ett objekt
På föregående sida stod det att i alla klasser finns en speciell metod som används när objekt skapas i ett javaprogram. Den metoden kallas *konstruktor* och har samma namn som klassen. Om vi tänker oss att klassen `Bil` existerar och att vi vill skapa objektet `minBil` i ett javaprogram så skulle vi skriva så här:

```
minBil = new Bil();
```

Man skapar (instansierar) alltså ett nytt objekt genom att anropa konstruktorn, och anropet föregås av ordet `new`. När detta är gjort existerar objektet `minBil` och kan användas.

Referera till objekt
Vi gav mitt objekt "namnet" `minBil`. Vi säger hellre *referensen* `minBil`, för vi använder detta ord när vi *refererar* till ett objekt , dess variabler och dess metoder. Antag att vi har målat `minBil` i silverfärg och att vi vill ändra värdet hos variabeln `färg` hos detta objekt. I ett javaprogram skriver vi så här:

```
minBil.farg = "Silver";
```

Om vi vill backa `emmasBil` *anropar* vi naturligtvis metoden `backa()` för detta objekt:

```
emmasBil.backa();
```

Skrivsättet *`referens.variabel`* eller *`referens.metod()`* kallas *punktnotation.*

Om metoder
Du har redan lagt märke till att alla metoder skrivs med parenteser efter metodnamnet, `backa()`, till exempel. Ibland är en metod definierad med *parametrar*. Då förväntar sig metoden att man skriver in något mellan parenteserna. Det man skriver inom parenteserna kallas *argument.* Om metoden `backa()` vore definierad med en heltalsparameter, så att den krävde ett heltalsargument, skulle den anropas med till exempel:

```
emmasBil.backa(15);
```

Argumentet påverkar naturligtvis metoden på något sätt, det beror på hur funktionen är definierad.
(Anropet `emmasBil.backa(15)` kanske får `emmasBil` att backa 15 meter exempelvis.)

Vissa metoder är definierade så att den *returnerar* ett *värde.* I metoden `getAndrahandsVarde()` sker en beräkning av bilens andrahandvärde. Beräkningen bygger nog på variablerna `arsmodell` och `miltal`. Metoden är säkert definierad så att det beräknade värdet returneras vid anrop. När en sådan funktion anropas måste vi "fånga upp" det returnerade värdet i en variabel, eller skriva ut värdet.
Så här till exempel:

```
begagnatPris = minBil.getAndrahandsVarde();
```

eller

```
System.out.println(”Priset är” + minBil.getAndrahandsVarde() );
```

Metoder som returnerar ett värde kräver ofta ett argument. Metoden `getBensinAtgang()` är troligen definierad så att man anger körsträckan (i kilometer) som argument och då returneras bensinåtgången i liter.
Så här till exempel:

```
literBensin = minBil.getBensinAtgang(60);
```

(Metoden returnerar hur många liter bensin som gick åt för att köra 60 km.)

Metoder som bara får något att hända och som *inte* returnerar ett värde kallas för *void*-metoder.
Metoder som returnerar ett värde kan man kalla *icke-void*-metoder. Ofta kallas sådana metoder *funktioner.*
Metoder som returnerar ett värde inleds ofta med ordet `get`, t.ex **`get`**`AndrahandsVarde()`

Privata variabler och publika metoder
Ovan ändrade vi färgen hos objektet `minBil` med satsen `minBil.farg = "Silver";`
Oftast väljer man att låta egenskaperna (attributen, instansvariablerna) i ett objekt vara *privata*, vilket innebär att man inte kan påverka dem, t.ex. ändra dess värde, "utanför" objektet.
Men om man nu vill att det ska vara tillåtet att ändra värdet på variabeln `farg` "utifrån" så låter man oftast detta ske via en metod. Så här kanske:

```
minBil.setFarg("Silver");
```

Metoden `setFarg()` är då definierad så att variabeln `farg` tilldelas det värde som skrivs in som argument.
Metoder som påverkar en variabel inne i objektet har ofta ett namn som börjar med `set` , t.ex **`set`**`Farg()`

Detta är typiskt för objekt: Instansvariablerna brukar vara *privata* (de kan endast påverkas inuti det egna objektet) medan metoderna är *publika* (de kan användas även "utanför" objektet ifråga).

Klassmetoder och klassvariabler
För instansvariabler gäller att de finns i lika många upplagor som det finns objekt. Exempel:

`emmasBil.farg` har värdet "`Röd`"
`johansBil.farg` har värdet "`Grön`"
`majasBil.farg` har värdet "`Vit`"

För metoder gäller samma sak:
T.ex. metoden `getAndrahandsVarde()` ger olika resultat för `emmasBil` , `johansBil` och `majasBil` .

I vissa klasser finns det något som kallas *klassvariabler*. En sådan tillhör *klassen* och inte de enskilda objekten.
En klassvariabel finns i *endast ett* exemplar.

I vissa klasser finns det något som kallas *klassmetoder*. En sådan tillhör *klassen* och inte de enskilda objekten.
En klassmetod opererar inte på de enskilda objekten, utan är en fristående metod.

En "vanlig" variabel eller metod refererar man som bekant till med objektets referens ("objektnamnet").
En klassvariabel eller klassmetod refererar man till med *klassens* namn.
Exempel:
I Java finns en klass som heter `Math`. Den har bara klassmetoder och några klassvariabler (egentligen konstanter)
När man använder dessa använder man klassnamnet som referens, till exempel:

```
omkrets = 2 * Math.PI * radie;            (   omkrets = 2 · π ·radie   )
d = s * Math.sqrt(2);                     (   d = s√2                  )
```

Arv
Tänk dig att det finns en klass med ett antal egenskaper och metoder. Antag nu att vi behöver en klass som är *nästan* likadan men att den borde vara försedd med ytterligare några egenskaper och kanske någon mer metod.
I Java finns en företeelse som heter *arv*. När en klass *ärver* från en annan klass betyder det att den ärvande klassen får samma uppsättning instansvariabler och metoder som klassen den ärver ifrån. Dessutom kan man lägga till ytterligare variabler och metoder i den nya klassen. Den ursprungliga klassen kallas *superklass*, och klassen som ärver kallas *subklass*. I språket Java uttrycks arv med ordet `extends`. Exempel:
Om vi vill skapa klassen `Sportbil` kan vi utnyttja att vi redan har en klass som heter `Bil` och skriva så här:

```
public class Sportbil extends Bil { . . . }
```

Konventioner
Klassnamn brukar skrivas med versal i början, t.ex. `Bil`
Referenser till objekt, variabelnamn, metodnamn brukar skrivas med gemener. Om referensen eller namnet är sammansatt låter man det andra och tredje ledet få inledande versal, t.ex `getBensinAtgang()`
Använd endast tecknen `A-Z`, `a-z`, `0-9` och eventuellt understrykningstecknet, _ .

Nu är det dags att programmera Java!

Java innehåller ett stort antal färdiga klasser. Om vi vill skapa ett program med *grafiskt användargränssnitt* (*graphic user interface*) (ett *GUI*-program) utnyttjar vi färdiga klasser för själva ”programfönstret” samt för knappar, textrutor, kryssrutor med mera.

Klasserna ligger lagrade i "*paket*", *packages*. Klasser som har med GUI att göra (knappar, kryssrutor och andra *grafiska komponenter,* samt färger, grafik med mera) finns i paketen `java.awt` och `javax.swing`.
För att komma åt en klass i ett paket måste man *importera* paketet. Det gör man allra först i ett javaprogram

Varje javaprogram har som bekant en `main()`-metod. Det är den som kör igång då vi startar programmet.
Vi ska skriva ett program som först skapar ett objekt ur klassen `JFrame`.
Sen låter vi objektet anropa några (void-)metoder. När metoden `setVisible()` anropas med argumentet `true` så visas objektet på skärmen! Här är programmet:

```
1   import javax.swing.*;
2   public class VisaJFrame {
3     public static void main( String args[] )  {
4         JFrame f ;
5         f = new JFrame();
6         f.setSize(200,300);
7         f.setLocation(100,200);
8         f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
9         f.setVisible(true);
10     }
11  }
```

Skapa klassen `VisaJFrame` . Skriv in koden. Spara och provkör. Ett windowsfönster visas!

Läs kommentarer till programmet noga:

Rad

1. Vi importerar paketet `javax.swing`. Asterisken (*) betyder att vi har tillgång till *alla* klasser i paketet (men här behöver vi bara `JFrame`)

2. Ordet `public` betyder att klassen är synlig "utifrån" andra klasser

3. `main`()-metoden har du använt många gånger förr. Ordet `public` betyder att metoden kan köras "utifrån"

4. Du vet att man måste deklarera variabler (t.ex. `int i;`). På samma sätt måste man deklarera objekt!
   Här deklarerar vi att vi tänker använda ett objekt ur `JFrame`-klassen, som har referensen `f`

5. Här skapas (instansieras) objektet (instansen) `f` genom anrop av konstruktorn i klassen `JFrame` och med ordet `new` före konstruktornamnet.
   Satserna på rad 4 och 5 skrivs ofta ihop, så här: `JFrame f = new JFrame();`

6. Objektet `f`.s metod `setSize()` anropas med parametrarna `200,300`. Objektet har instansvariabler för bredd och höjd. Dessa får nu värdet 200 resp. 300, det vill säga fönstrets storlek bestäms här.

7. Objektet `f`.s metod `setLocation()` anropas med parametrarna `100,200`. Objektet har instansvariabler för placering i x-led resp y-led. Dessa får nu värdet 100 resp. 200, det vill säga fönstrets placering bestäms här.

8. Den här metoden gör så att fönstret stängs på normalt sätt när man klickar kryssrutan uppe till höger.

9. Då metoden `setVisible()` anropas med argumentet `true` visas objektet (fönstret) på skärmen!

Ändra argumenten i metoderna så att fönstret får annan storlek och annan placering,

Låt oss nu skriva en *egen* klass, `MinJFrame`, som *ärver* från `JFrame`.
`MinFrame` ska visa en etikett, ett `JLabel`-label-objekt. En etikett visar en fast text som inte kan ändras.
Så här blir klassen:

```
1   import javax.swing.*;
2   import java.awt.*;
3   public class MinJFrame extends JFrame {
4        JLabel etikett;
5        public MinJFrame()  {
6             etikett = new JLabel("Hej världen!");
7             etikett.setForeground(Color.red);
8             this.add(etikett);
9        }
10  }
```

Skapa klassen `MinJFrame`. Skriv in koden och spara.   Läs kommentarer:

Rad

1. Import av swing-paketet (där finns klasserna `JFrame` och `JLabel` med flera)
2. Import av awt-paketet (sdär finns klassen `Color` med flera)
3. `MinFrame` ärver från `JFrame` (ärver variabler och metoder från `JFrame`)
4. Deklarera `JLabel`-objektet `etikett`
5. Vi definierar konstruktorn. Den är `public`, det vill säga den kan anropas "utifrån", från andra klasser.
6. Här skapas ett objekt, med referensen `etikett` , ur klassen `JLabel`.   .
7. Vi anropar `etiketts` void-metod `setForeground()` med argumentet `Color.red` (en konstant i klassen `Color`) Texten i labeln kommer att visas med röd färg.
8. När ett objekt refererar till sig själv används referensen `this`. Här anropas alltså `JFrame`-objektets void-metod `add()`. Argumentet är `etikett`. Anropet `this.add(etikett)` placerar ut `etikett` på `this`, alltså på `JFrame`-objektet (fönstret som vi ser) .

Observera att `MinJFrame` *inte* är ett program. Det finns ju ingen `main()`-metod.
Nu skriver vi ett program vars uppgift är att skapa och visa ett objekt ur klassen `MinJFrame`:

```
1   import java.awt.*;
2   public class Start {
3      public static void main(String args[]){
4         MinJFrame f = new MinJFrame();
5          f.setSize(200,200);
6          f.setLocation(200,300);
7          f.setLayout(new FlowLayout());
8          f.setDefaultCloseOperation(MinJFrame.EXIT_ON_CLOSE);
9         f.setVisible(true);
10     }
11  }
```

Skapa klassen `Start`. Skriv in koden, spara och kör.   Läs kommentarer:

Rad

1. Import av awt-paketet (som har klassen `FlowLayout` som används på rad 7)

4. Här skapas (instansieras) objektet `f` genom anrop av konstruktorn i klassen `MinJFrame` med ordet `new` före konstruktornamnet. Koden i konstruktorn `MinJFrame()` exekveras, det vill säga objektet `etikett` skapas, får röd färg och läggs ut i fönstret.

7. Grafiska komponenter kan placeras ut på en yta på olika sätt beroende på vilken *layout* som är vald.
Detta metodanrop ger *flowlayout* som vi kommer att använda tills vidare.
Flowlayout innebär att komponenter placeras ut centrerade, uppifrån och ned.

Klassen `MinJFrame` är *inte* ett program. Det finns ingen `main()`-metod.
Låt oss skriva in en `main()`-metod i klassen. Dess uppgift ska vara att skapa och visa ett objekt ur klassen `MinJFrame`. En sådan `main()`-metod har vi ju redan, nämligen i klassen `Start` !

Alltså: Markera `main()`-metoden i klassen `Start` och kopiera in den i klassen `MinJFrame`. Så här:

```
import javax.swing.*;
import java.awt.*;
public class MinJFrame extends JFrame {
      JLabel etikett;

      public MinJFrame() {
            etikett = new JLabel("Hej världen!");
            etikett.setForeground(Color.red);
            this.add(etikett);
      }

      public static void main( String args[] ) {
             MinJFrame f = new MinJFrame();
             f.setSize(200,200);
             f.setLocation(200,300);
             f.setLayout(new FlowLayout());
             f.setDefaultCloseOperation(MinJFrame.EXIT_ON_CLOSE);
             f.setVisible(true);
      }
}
```

Spara och exekvera `MinJFrame`.

Med denna introduktion bör du kunna arbeta med kapitel 4 och därpå följande kapitel i *Java Javisst!* på ett meningsfullt sätt.

Programmet `MinFrame` kan du ha som modell för många GUI-program.

Repetition

Förklara för dig själv vad nedanstående ord och begrepp betyder:

		När används orden
Klass	Instansiera	`new`
Objekt	Punktnotation	`extends`
Attribut	Privat	`this`
Instansvariabel	Publik	
Metod	Arv	
Konstruktor	Superklass	
Void metod	Subklass	
Funktion	GUI	
Parameter	Package/Paket	
Argument		