Objektorienterad Programmering DAT043

Föreläsning 3
22/1 -18
Moa Johansson
(delvis baserat på Fredrik Lindblads material)

Kom ihåg: Visa Javas API. Nämn att slidesens också har tillhörande text på kurshemsidan.
Påminn om att kod som lämnas in till labbarna ska vara tydlig, välstrukturerad, kommenterad etc. enligt instruktioner på labsidan. Annars kan man få retur på labben trots att programmet producerar “rätt” svar.

Dagens föreläsning

- Objekt - hur definierar vi nya klasser och objekt?
- Objekttyper.
- Konstruktor-metoder: att skapa nya objekt.
- Inkapsling och synlighet: `public/private`

På förra förläsningen talade vi lite kort om några Objekttyper: t.ex. String. Idag går vi mer på djupet och går igenom hur man skapar egna objekttyper och konstruerar objekt av dessa typer, samt hur Java organiserar kod i klasser.

Repetition: Objektorienterad programmering

Ur SAOL:

2 ⟨språkv.⟩ satsdel som **uttrycker föremål för handling**
– I sammansättn. *objekt-* (vanl. till *objekt 1*), *objekts-* (vanl. till *objekt 2*).

- ”Ada Lovelace programmerade den första datorn”.
- ”Programmet skriver ut ”Hello World” i terminalförstret”.
- ”Jag flyttar tornet två steg åt höger”.
- “Ta bort alla siffror ur strängen”.

Från föreläsning 1: Namnet ”objektorienterad programmering” kommer från språkvetenskapen, där det benämner en satsdel som är föremål för handling.

Alltså: Inom objektorienterad programmering struktureras programmen efter vilka objekt som vi vill modellera.

Repetition: Objektorienterad programmering

- **Objekt:** modeller av det programmet hanterar, e.g.
  - terminalfönster,
  - spelpjäser i ett schackprogram…
- **Klasser:** moduler i Java, innehåller, t.ex.
  - Beskrivning av en **objekttyp.**
  - **Metoder:** vilka handlingar kan man utföra på objekten.
  - **Konstruktorer:** hur vi konstruerar objekt.

Objekt är alltså ”modeller” av det programmet hanterar (e.g. meddelanden som skrivs ut, spelpjäser i ett schackspelsprogram).
Beskrivningar/definitioner av objekt tillsammans med *metoder* (handlingar man kan göra med/på) objekten (jmfr. med funktioner) samlas i Java i *klasser*.

Den här föreläsningen handlar om att definiera våra egna objekt

Skapa objekt och klasser

I Java:

```
public class Person{
   public String förnamn;
   public String efternamn;
   public int ålder;
   public double längd;
   public double vikt;
   ....
   ....
}
```

I Haskell:

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , weight :: Float
                     }
```


Det är ofta praktiskt att kunna definiera nya typer när man programmerar, t.ex. för att “bunta ihop” värden som på något vis hör samman. Här antar vi att vi har fått i uppgift att skriva ett program som hanterar personinformation, och i vårt exempel är vi intresserade av för- och efternamn, ålder, längd och vikt.

Olika programmeringsspråk löser detta på olika sätt. I Java gör vi det genom att definiera en ny objekttyp i en ny klass som heter `Person` (I Haskell skulle man definiera en ny datatyp med s.k. *record-syntax)*. Eftersom Java är ett objektorienterat språk, där vi strukturerar programmet efter vilka objekt vi ska *göra saker med,* så definieras `Person` i en egen klass med samma namn. Den information vår nya objekttyp ska innehålla (namn, ålder, längd osv) kallas i Java för klassens/objektets *instansvariabler eller komponenter (*eller “*fields*” på engelska).

Vi definierar en typ av objekt per klass. (Detta skiljer sig från Haskell, där man ofta kan definiera många datatyper i samma modul). Enligt objektorienterad filosofi ska även metoder som gör saker med vår nya objekttyp Person också definieras i klassen Person. Klassnamn skrivs alltid med stor bokstav enligt Javas konventioner.

En klass som är deklarerad public som klassen Person, måste sparas i en fil med samma namn, dvs Public.java. Vanligtvis i Java definierar man också bara en klass per fil.

Skapa objekt och klasser: Konstruktormetoder

```
public class Person{
   public String förnamn;
   public String efternamn;
   public int ålder;
   public double längd;
   public double vikt;

   public Person(String fnamn, String enamn, int å,
                 double l, double v){
     förnamn = fnamn;
     efternamn = enamn;
     ålder = å;
     längd = l;
     vikt = v;
   }
}
```

```
Person p1 = new Person(“Alice”, “Alicedotter”, 60, 1.60, 60.5);
Person p2 = new Person(“Bob”, “Bobsson”, 25, 1.85, 93.0);
```


Varje objekt av typ Person som vi konstruerar kommer att ha sina egna kopior, eller *instanser*, av komponenterna `förnamn`, `efternamn`, `ålder`, `längd` och `vikt`. Hur konstruerar vi då nya objekt av typen Person?

Via speciella metoder som kallas konstruktorer (constructor methods)! En konstruktor-metod har alltid samma namn som klassen/objekttypen. En klass kan ha en eller flera konstruktorer som representerar olika sätt att skapa objekt av typen.

Nya objekt av typen skapas med operatorn `new` samt ett anrop till konstruktorn.

Skapa objekt och klasser: Konstruktormetoder

```
public Person(String fnamn, String enamn, int å,
                double l, double v){
    förnamn = fnamn;
    efternamn = enamn;
    ålder = å;
    längd = l;
    vikt = v;
  }

public Person(String förnamn, String efternamn){
  this.förnamn = förnamn;
  this.efternamn = efternamn;
  ålder = -1; // okänt
  längd = -1  // okänt
  vikt = -1   // okänt
}
```

```
Person p1 = new Person("Alice", "Alicedotter", 60, 1.60, 60.5);
Person p2 = new Person("Bob", "Bobsson", 25, 1.85, 93.0);
Person p3 = new Person("Calle", "Carlsson");
```


En klass kan ha en eller flera konstruktorer som representerar olika sätt att skapa objekt av typen. Här definierar vi en andra konstruktor som vi kan använda för att konstruera ett objekt av typ Person när vi inte bryr oss om eller känner till ålder, längd och vikt.

Notera att vi kan döpa parametrarna till konstruktorn till samma namn som instansvariablerna! Hur ska vi nu veta vad som är vad? Jo, vi kan använda notationen `this` vilket talar om att vi menar instansvariabeln på "this object", dvs det objekt vi håller på att konstruera. `this.förnamn` refererar alltså till instansvariabeln i objektet vi konstruerar, medan bara `förnamn` refererar till konstruktormetodens argument.

Det är OK att ha flera metoder med samma namn i samma klass i Java. Så länge de har olika parametrar och/eller returtyp kan kompilatorn skilja på dem.

Skapa objekt och klasser: Instansmetoder

```
public class Person{
   public String förnamn;
   public String efternamn;
   public int ålder;
   public double längd;
   public double vikt;
   ....
   // Beräkna body-mass-index
   public double bmi(){
      return vikt/(längd*längd);
   }
}
```

```
Person p1 = new Person(“Alice”, “Alicedotter”, 60, 1.60, 60.5);
Person p2 = new Person(“Bob”, “Bobsson”, 25, 1.85, 93.0);

double aBMI = p1.bmi();
double bBMI = p2.bmi();
```


Vi definierar en typ av objekt per klass. (Detta skiljer sig från Haskell, där man ofta kan definiera många datatyper i samma modul). Enligt objektorienterad filosofi ska även metoder som gör saker med vår nya objekttyp Person också definieras i klassen Person. Exempelvis en metod som räknar ut en persons BMI.

Men, metoden `bmi` refererar till `vikt` och `längd`, trots att den inte tar några argument! Varifrån kommer dessa värden?
`bmi` är en s.k. i*nstansmetod*, det vill säga en metod som kallas ***på (eller av) ett objekt av typ `Person`.*** Alla objekt av typ `Person` har ju komponenterna längd, vikt osv. och det är de specifika värdena från det objekt som metoden kallas på som avses.

Genom att samla alla metoder som har att göra med objekt av typen Person i samma klass, så samlas kod som hör ihop på ett naturligt sätt enligt objektorienterad filosofi. Vi utgår alltså från objekten när vi delar upp vårt program i olika filer (klasser).
(I ett funktionellt programmeringsspråk skulle man kanske dela upp kod på ett annat vis, t.ex. efter funktion. Ett exempel från ett av de projekt jag är inblandad i: En modul som heter “Pretty” som innehåller kod som Pretty-printar många olika data-typer. Så skulle man inte göra i Java, då skulle varje datatyp ha en egen klass med en egen Pretty-print metod).

Demo: Person.java, TestPerson.java

Inkapsling och synlighet

```
public class Person{
   public String förnamn;
   public String efternamn;
   public int ålder;
   public double längd;
   public double vikt;
   ....
   public double bmi(){
      return vikt/(längd*längd);
   }
}
```

I en annan klass kan objekten ändras om t.ex. `längd` är `public`:

```
Person p1 = new Person(“Alice”, “Alicedotter”, 60, 1.60, 60.5);
Person p2 = new Person(“Bob”, “Bobsson”, 25, 1.85, 93.0);
....
p1.längd = 165;
```


Vi har i flera exempel sett ordet “`public`” framför metoder och instansvariabler. Vad betyder det?
Jo, det har att göra med synlighet: en metod som är public kan t.ex. kallas utanför klassen där den är definierad (exempelvis kan konstruktorerna och metoden bmi() kallas från klassen TestPerson.java. Detta är oftast naturligt.

Ibland vill man dock “gömma” implementationsdetaljer från användare av sin klass. Om instansvariabler, är deklarerade `public` så kan man komma åt dem varifrån som helst, och även ändra dem. Detta är inte alltid önskvärt, så därför brukar instansvariabler deklareras “`private`” istället. Det innebär att bara metoder i klassen har tillgång till dem. Detta för att objektens interna representation inte ska exponeras för omgivningen, den kod som använder klassen.

I vårt exempel råkar en användare uppdatera p1 på ett felaktigt sätt (i cm istället för m). Detta går eftersom man har direkt åtkomst till objektens interna representation.

Det kan även finnas anledning att deklarera en metod som privat, om det gäller en intern hjälpmetod som bara är tänkt att användas inom klassen och aldrig utifrån.

Inkapsling och synlighet

```
public class Person{
   private String förnamn;
   private String efternamn;
   private int ålder;
   private double längd;
   private double vikt;
   ....
   public void setLängd(double nylängd){
     // enkel "sanity check", folk är mellan (säg) 0.3-3m.
     if(0.3 <= nylängd && nylängd <= 3.0)
       längd = nylängd;
     else
       System.out.println("Längd ska anges i meter. Godkända
                           värden är 0.3-3.0m.");
   }
   public double getLängd(){
    return längd;
   }
}
```


Varför vill man inte detta? Dels så vill man hålla all kod som hör ihop med hur man valt att representera objekten på samma plats, i klassen, så att övrig kod inte kan förstöra tillståndet i ett objekt. I stora projekt är det ofta olika personer som implementerar klassen och som använder dem och man vill att beroendena mellan klass-koden och den användande koden ska bli så små som möjligt. Därför döljer man representationen av objekten och definierar ett antal metoder som användaren anropar istället för att manipulera direkt med instansvariablerna. Metoderna utgör ett gränssnitt som skiljer de olika delarna av koden åt. Den som använder klassen kan inte ställa till det i den interna representationen genom sin okunskap. Den som implementerar klassen kan utan problem ändra den interna representationen utan att resten av koden behöver ändras, så länge gränssnittet förblir detsamma.

I exemplet ovan är objektet inte särskilt komplext och valet av representation är naturligt, så att låta komponenterna vara public är inte så dumt. Man i allmänhet när det gäller mer komplexa objekt är valet av representation inte självklart och risken att göra fel stor. Då är möjligheten att dölja representationen för omvärlden väsentlig.

Om vi gör instansvariablerna ovan `private` så försvinner möjligheten för en användare av klassen att direkt avläsa och förändra värdena. Detta gör att man behöver så kallade *getters och setters,* metoder som bara är till för att göra detta.

En annan fördel med att "gömma" den interna representationen är att vi kan ändra den, så länge de publika metoderna förblir desamma, kommer vi inte att "ha sönder" kod som använder klassen.

DEMO: Person2.java

Klassvariabler: `static`

```
public class Person{
  // Klassvariabler: delas av alla objekt/metoder
  private static double minLängd = 0.3;
  private static double maxLängd = 3.0;

   // Instansvariabler: en uppsättning per objekt av typ Person
   private String förnamn;
   private String efternamn;
   private int ålder;
   private double längd;
   private double vikt;
   ....
   public Person(String fnamn, String enamn, int å,
                 double l, double v){
     förnamn = fnamn;
     efternamn = enamn;
     ålder = å;
     vikt = v;
     if(minLängd <= l && l <= maxLängd)
        längd = l;
      else ....;
   }
   ...
   public void setLängd(double nylängd){
     if(minLängd <= nylängd && nylängd <= maxLängd)
        längd = nylängd;
       ....
```


Variabler som deklareras statiska kallas klassvariabler och är globala variabler. När ett program körs finns bara *en instans av klassvariabler, inte en per objekt som gäller för instansvariablerna*.

Statiska metoder kan bara komma åt statiska variabler. För att anropa statiska metoder eller komma åt statiska variabler utanför klassen inleder man man klassens namn följt av punkt. Vi har t.ex. använt metoden Integer.parseInt. Detta är en statisk metod som tillhör klassen Integer.

Exempel: Vi såg tidigare metoden setLängd, som innehöll en kontroll av att det nya värdet på längden av en Person var inom rimliga gränser. Även konstruktorn för klassen bör kontrollera detta. Här skulle ett bra alternativ vara att explicit deklarera dessa gränser som statiska variabler, eftersom alla instanser av Person skall ha en längd inom samma spann. Vill man då ändra dessa gränser är det enkelt att göra det på ett och samma ställe, nämligen de statiska klassvariablerna minLängd och maxLängd. På detta sätt undviker vi att vi av misstag anger olika spann i olika metoder.

Klassmetoder aka statiska metoder

```
public class Person{
  // Klassvariabler: delas av alla objekt/metoder
  private static double minLängd = 0.3;
  private static double maxLängd = 3.0;

  // Instansvariabler: en uppsättning per objekt av typ Person
   private String förnamn;
   private String efternamn;
   private int ålder;
   private double längd;
   private double vikt;

  // Klassmetoder
  public static double getMinLängd(){return minLängd;}
  public static double getMaxLängd(){return maxLängd;}
```

```
Person p1 = new Person("Alice", "Alicedotter", 60, 1.60, 60.5);

p1.bmi() // Rätt, bmi är en instansmetod, kallas av objekt.
Person.bmi() // FEL!

Person.getMinLängd() // Statisk metod kallas med Klassnamn.
p1.getMinLängd() // Dock tillåts även detta.
```


De metoder vi såg tidigare under första veckans föreläsningar var oftast statiska, och deklarerades med attributet `static`

Metoder som deklareras som statiska kallas klassmetoder och de har inget aktuellt objekt som implicit argument. I alla exempel vi gjort tidigare har metoder varit statiska av denna anledning. Main-metoden måste deklareras som static och public.

Om vi vill skriva metoder för att komma åt de statiska variablerna minLängd och maxLängd är det naturligt att även dessa är statiska. De har ju inget att göra med specifika objekt av typen Person (instanser av klassen Person) utan hör direkt till själva klassen.

Demo: Person3.java

Samanfattning

Vi definierar ***nya objekttyper i klasser***. Ett objekt med typen som definierats i en klass kallas ofta en *instans av klassen*.

- **Instansvariabler** - en uppsättning *per objekt av typen*.
- **Klass- eller statiska variabler** - en *per klass*. Delas av alla objekt.
- **Instansmetoder** - kallas alltid *på ett objekt* av klassens typ: `obj.instansmetod(argument)`. Här blir `obj` som ett extra (implicit) argument till metoden.
- **Klass- eller statiska metoder** - hör till klassen. Kallas med klassnamnet som prefix: `Klassnamn.statiskMetod(argument)`

Värdet `null`

- Förvalda värden: om vi inte initierat en variabel explicit ännu. e.g. 0 för typen `int`, `false` för typen `boolean` osv.
- För objekttyper och arrayer är förvalt värde specialvärdet `null`.
- `null` betyder att variabeln inte refererar till någonting.

```
Person p1;
...
if (p1 == null){
   // Hantera fallet att p1 inte är initierad.
}
```

```
Person[] personer = new Person[100];
// Sant eller falskt?
boolean isArrayNull = personer == null;
boolean isElementNull = personer[0] == null;
```

Förvalt värde för objekt och arrayer som inte initieras explicit är ett specialvärde som heter null. Detta innebär att variabeln inte refererar till någonting. *(Möjligheten att sätta ett objekt till null används ganska mycket för att situationer där man kan använda Maybe A i Haskell, d.v.s. man kanske har ett A. Man kan säga att alla objekt av klassen A i java egentligen är Maybe A.)*

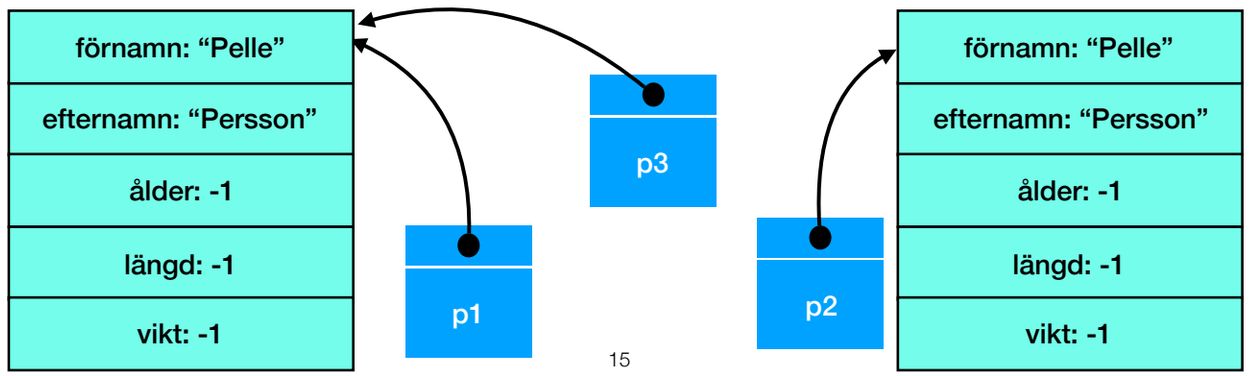
Man kan testa om en variabel refererar till ett objekt eller inte, i.e. om den har initierats eller ej.
I exemplet har vi initierat arrayen `personer` till en ny array av längd 100. Så `isArrayNull` är falsk. Däremot har vi ännu inte initierat elementen i arrayen. Dessa har därför sina förvalda värden. Eftersom `Person` är en objektyp är det förvalda värdet null, så `isElementNull` är sann.

Övning: Om vi vill initiera arrayen personer med faktiska objekt instället för null, hur gör vi då? Objekten ska existera, men behöver inte ännu ha definierade värden på instansvariablerna. Tips: Lägg till en konstruktor utan argument till Person-klassen.

Jämföra objekt

```
Person p1 = new Person(“Pelle”, “Persson”);
Person p2 = new Person(“Pelle”, “Persson”);
Person p3 = p1;

System.out.println(p1 == p2);  // -> false
System.out.println(p1 == p3);  // -> true
```



Man kan också jämföra objektvariabler med varandra. Detta avgör om de refererar till exakt samma instans eller ej, ej om deras komponenter är lika.

Om man vill kunna jämföra innehållet i objekt skriver man oftast istället en (instans)metod som heter equals(...). Där kan man definiera hur objektets komponenter ska jämföras. Se t.ex. equals för String:
String s1 = “Pelle”;
String s2 = “Pelle”;
s1 == s2; //false
s1.equals(s2); // true

Övning:
Skriv en metod `public boolean equals(Person otherPerson){...}` för klassen Person! Kontrollera att p1.equals(p2) returnerar true för exemplet ovan.

Övning: Vad är värdet på x1, x2, x3 om vi skriver:
p1.setLängd(1.70);
p2.setLängd(1.94);
p3.setLängd(1.78);
double x1 = p1.getLängd();
double x2 = p2.getLängd();
double x3 = p3.getLängd();

Konstanter: `final`

```
class Employee {
  // Klasskonstant
  public final static int minSalary = 15000;

  // Konstant instansvariabel. Kan ej ändras efter initiering.
  private final int id;
  private String name;
  private int salary;

  public Employee(int id) {
    this.id = id;
  }

  public void setId(int newId) {
    id = newId;  // Detta går inte.
  }
}
```


Man kan ange att variabler ska vara konstanter, d.v.s. ej gå att ändra. Det gör man genom attributet final. Både instansvariabler och klassvariabler kan ha attributet `final`.

Konstanter kan förstås initieras i deklarationen, men även initieras och ändras i klassens konstruktorer. Det är först när konstruktorn är klar som värdet fixeras och inte kan ändras mer.

Exempel 2:
I klassen Person används värdet -1 för att ange att ålder/längd/vikt är okänt. Ett snyggare sätt att tydligöra detta är att använda statiska konstanter, t.ex.
public final static double OKÄND_LÄNGD = -1;
public final static double OKÄND_VIKT = -1;
public final static int OKÄND_ÅLDER = -1;
Notera att det är säkert att låta dessa konstanter vara publika, eftersom de är deklarerade “final” kan de inte ändras ändå.

Nu kan kod utanför klassen testa ifall ett objekt har okänt värde för t.ex. längd genom att testa för likhet med konstanten Person.OKÄND_LÄNGD, och behöver inte bry sig om exakt vilket värde vi internt använder för att representera detta (ännu ett exempel på inkapsling). Detta är god programmeringspraktik! Exempel:
if (p1.getLängd() == Person.OKÄND_LÄNGD)
System.out.println(“Längd okänd!”);