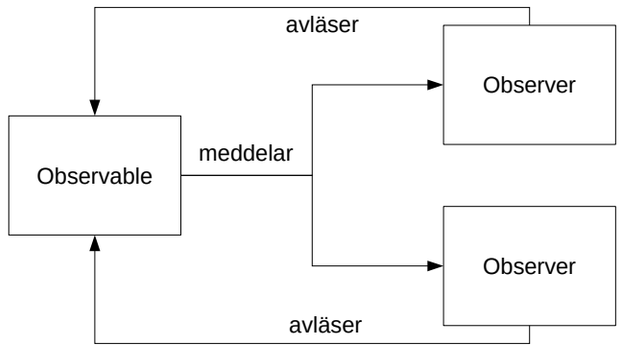
Objektorienterad Programmering DAT043

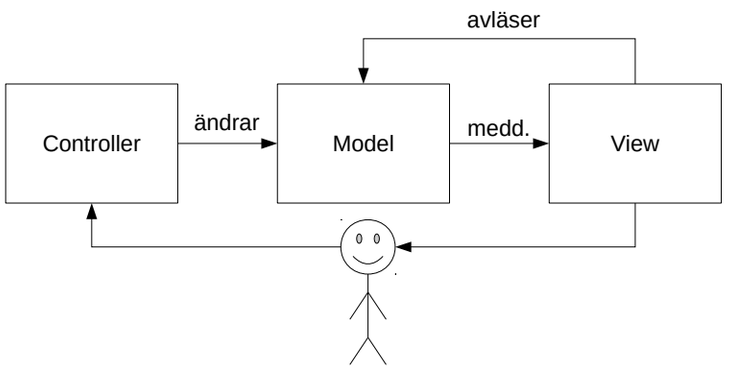
Föreläsning 7
5/2 -18
Moa Johansson
(delvis baserat på Fredrik Lindblads material)

Designmönstret Observer



En eller flera objekt registrerar sig som observers hos ett objekt som är observable.
När en förändring sker meddelar observable-objektet alla observers om detta.
När observers-objekten är grafiska komponenter talar man istället om
Model-View. Förändringar i modellen meddelas till vyn/vyerna som uppdaterar sitt
Innehåll.

Modell-View-Controller



I designmönstret Model-View-Controller finns det också ett eller flera objekt, grafiska komponenter, som utför ändringar i modellen enligt användarens kommandon. Ett objekt har ofta både rollen View och Controller.

MVC: Modell

Exempelmall för modell-klasser i MVC:

```
import java.beans.*;

public class Model {

      private final PropertyChangeSupport pcs
        = new PropertyChangeSupport(this);

      public void addPropertyChangeListener(PropertyChangeListener l) {
        pcs.addPropertyChangeListener(l);
      }
      public void removePropertyChangeListener(PropertyChangeListener l){
        pcs.removePropertyChangeListener(l);
      }
      . . .
      // variabler och metoder specifika för modellen
      // när förändring skett, anropa pcs.firePropertyChange
    }
```

Kan använda infrastrukturen för PropertyChange i paketet java.beans. Här finns klassen PropertyChangeSupport, vars objekt kan användas för att hålla reda på vilka lyssnare som ska notifieras när något händer med modellen. Vi implementerar två metoder för att lägga till/ta bort lyssnare. Dessa lyssnare “kommer från” vy-klasser.

Denna mall kan användas för modeller.

MVC: Vy (view) och Controller

Exempelmall/pseudokod för vy-klasser i MVC:

```
public class View extends JSomeComponent
                    implements PropertyChangeListener {
  Model model;
  public View(Model theModel) {
    model = theModel;
  }
  @Override
  public void propertyChange(PropertyChangeEvent evt) {
    // avläs modell och uppdatera komponenten
  }
}
```

En vy-klass har sen ovanstående struktur. Notera att de är en subklass av någon lämplig JComponent, samt implementerar Intefacet PropertyChangeListener, för att kunna “registera sig” med hjälp av motsvarande metoder i Modell-klassen. Notera även att vyn ofta har sin modell som instansvariabel.

En controller-klass känner också till sin modell (har en modell som instansvariabel) och förändrar modellen via det gränssnitt som denna tillhandahåller. Ändringar gjorda av controllen kommer automatiskt återspeglas i vyerna.

DEMO: MVCDemo

Repetition: Grunderna för klasser

```
public class A {
  int s = 0;
  double[] arr;

  public A() {
    arr = new double[10];
  }

  public A(int n) {
    arr = new double[n];
  }

  public int size() {
    return s;
  }
}


A a1 = new A();
A a2 = new A(20);
```

- Klasser definierar hur objekt med ett visst syfte ser ut, i.e. definierar en s.k. objekttyp.
- Klassen talar om vilka variabler/fält en instans/ett objekt av denna typ ska innehålla.
- Klassen innehåller också ett antal metoder som kan utföras för en instans av klassen (på ett objekt av denna objekttyp).
- I dessa metoder kan man referera till variablerna i aktuell instans. Man kan också anropa de andra metoderna.
- I en klass kan det finnas en eller flera konstruerare som initierar nya instanser. Konstruerare definieras som metoder utan resultattyp och med samma namn som klassen. De har en valfri uppsättning argument. Den av dessa som passar med de argument som angetts anropas när man använder new-operatorn. Initiering av variabler kan också ske på samma sätt som för lokala variabler, ex. int x = ...;

En klass kan sakna konstruerare. Det går då att skapa objekt med new C() och variablerna får sina default-värden. Denna parameterlösa konstruktor “ärvs” från alla klassers implicita superklass, Object.

Repetition: forts.

```
public class A {
  public int x;
  public A(int x) {
    this.x = x;
  }
  public A() {
    this(0);
  }
  public void neg() {
    x = -x;
  }
}
 public static f(A o) {
  o.x = 7;
}

A a = new A();
// a.x == 0
a.x = 5;
// a.x == 5
f(a);
// a.x == 7
a.neg();
// a.x == -7
```

- Objekt/instanser av klasser är av referenstyp (liksom arrayer) d.v.s. när de skickas som argument och resultat till och från metoder och sätts med tilldelningsoperatorn (=) så kopieras bara pekaren till objekt. Alltså har t.ex. ändringar av ett argument i en metod också effekt utanför metoden.
- För att komma åt metoder och fält i ett objekt använder man punkt-operatorn. Inuti klassens metoder och initieringar kan man hänvisa till dess variabler och metoder utan punkt-operatorn.
- Om ett fält (en variabel i klassen) överskuggas av en lokal variabel i en metod så kan man hänvisa till klassens variabel med this.varnamn
- En konstruerare kan anropa en annan genom this(…) där argumenten stämmer överens med signuturen för en konstruerare.
- Metoder kan, liksom konstruerare, överlagras, d.v.s. ha samma namn, så länge de inte har samma antal argument och samma argumenttyper.

Repetition: Interfaces

```
public interface Show {
  String show();
}
public interface Scalar {
  double toDouble();
}

public class AnInt
    implements Show,
               Scalar {
  public int x;
  public String show() {
    return Integer.toString(x);
  }
  public double toDouble() {
    return x;
  }
}

Show o1 = new AnInt();
Scalar o2 = new AnInt();
o1.show();  // ok
o2.toDouble();  // ok
o1.toDouble();  // ej ok
o2.show();  // ej ok
```

•Interface används för att definiera ett sätt att kommunicera med objekt som kan tillhöra olika klasser.
•I ett interface anger man ett antal metoder (namn, argumenttyper, resultattyp) som definierar ett gränssnitt.
•En klass kan implementera ett interface. Man skriver class C implements I.
•En klass som implementerar ett interface måste definiera alla metoder som finns i interfacet.
•Flera klasser kan implementera samma interface. På det sättet kan man ha alternativa implementeringar av samma funktionalitet. Man kan deklarera en variabel att vara av en viss interface-typ .Variabeln kan då tilldelas värdet av objekt som tillhör en klass som implementerar interfacet. Vid skapandet av objektet måste man dock ange en konkret klass.
En klass kan implementera flera interface:
class C implements I1, I2, I3 { ... }

Statiska metoder och variabler

```
public class A {
  static int idCount;
  int id;
  String s;

  static {
    idCount = 0;
  }

  public static void setIdCount(int ic) {
    idCount = ic;
    // s = "går ej";
  }

  public A() {
    id = idCount++;
    s = "går bra";
  }
}

A.setIdCount(100);
```

- Metoder och fält av den typen som beskrivs på föregående slide kallas instansmetoder/-variabler.
- En variabel kan deklareras som static vilket innebär att det bara finns en kopia av den, inte en för varje instans. Kan ses som en global variabel som är relaterade till klassen och dess instanser.
- En metod kan deklareras som static vilket innebär att den inte anropas för en viss instans. Metoden ligger i klassen för att den ändå är relaterad till denna och dess objekt på något sätt.
- Såda variabler och metoder kallas klassvariabler/-metoder.
- En klass kan användas som ett bibliotek för metoder som hör ihop och innehåller då bara statiska element. Exempel: Math
- Statiska element kommer man åt genom att använda punkt-operatorn på klassnamnet.
- En instansmetod kan använda både instans- och klassmetoder och -variabler. En klassmetod kan bara använda klassmetoder och -variabler.
- Man kan initiera klassvariabler med en statisk konstruerare: static { … }

Konstanter

```
public class A {
  final int id;
  final AnInt obj;

  String s;

  public A(int id) {
    this.id = id;
    obj = new AnInt();
  }

  public void f() {
    // id = 3;  // går ej
    obj.x = 3;  // går
  }
}
```

•Variabler som deklareras som final måste initieras i deklarationen eller tilldelas ett värde en gång i konstruerarna. Sedan kan de bara avläsas. Dessa används alltså som konstanter, värden som man vill sätta en gång för alla i början och sedan inte ändra.
•När det gäller variabler av referenstyp är det **bara referensen som blir konstant**. **Innehållet kan ändå ändras**. Därför kan det vara **missvisande att deklarera ett objekt som final om det inte är icke-muterbart. String är ett exempel vars objekt är icke-muterbara.**

Synlighet: public/private

```
public class A {
  public int x;
  private int y;
  int z;
  public A() {
  }
  private A(int w) {
  }
  public void f() {
  }
  private void g() {
  }
}

// dessa är tillgängliga
// överallt:
A a = new A();
a.x = 1;
a.f();
// dessa bara i klassen:
// A a2 = new A(1);
// a.y = 1;
// a.g();
```

- En stor poäng med klasser är att kunna dölja den interna representationen för användaren av klassen och låta denna läsa av och förändra tillståndet hos instanser via ett gränssnitt av metoder.
- Man kan använda modifierarna public och private när man deklarerar variabler och metoder för att välja dess synlighet.
- public innebär att programkod var som helst i programmet kan komma åt variabeln eller metoden.
- private innebär att enbart kod i klassens metoder kan komma åt elementet.
- En metod/variabel inte har varken public- eller private-modifierare är inte åtkomlig för vem som helst, d.v.s. har ungefär samma tillgänglighet som private-medlemmar. (Den får tillgängligheten “package” som default, mer om detta senare.)
- Samma tillgänglighetsregler gäller konstruerare så dessa deklareras normalt som public.

Klasser i klasser

```
public class A {
  static int y;
  int x;
  public static class B {
    void f() {
      y = 1;
      // x = 1; – ej
    }
  }
  public class C {
    void f() {
      y = 1;
      x = 1;
    }
  }
  static void f() {
    B b = new B();
    // C c = new C(); – ej
  }
  void g() {
    B b = new B();
    C c = new C();
  }
}

A a = new A();
A.B b = new A.B();
// A.C c = new A.C(); – ej
A.C c = a.new C();
```

- Variabler i klasser kan förstås vara instanser av andra klasser (och av klassen själv).
- När en klass använder objekt av någon annan klass för den interna representationen kan det vara naturligt att definiera hjälpklassen inuti huvudklassen.
- Detta är möjligt och klasser i klasser kan liksom andra medlemmar har modifierarna static, public och private.
- I en statisk medlemsklass kan man inte nå instansmedlemmerna hos huvudklassen (utan att ha en instans av huvudklassen). Dessa fungerar alltså ungefär som om man använder en hjälpklass som är definierad utanför huvudklassen.
- För en icke-statisk medlemsklass är varje instans associerad med en instans av huvudklassen. Kod i medlemsklassen kan referera till instansmedlemmarna i huvudklassen.

Arv (Inheritance)

```
public class A {
  public int x;
  public void f() {
    x = 1;
  }
}
public class B extends A {
  public int y;
  public void g() {
    x = y = 1;
    f();
  }
}

A a1 = new A();
a1.x = 2;
A a2 = new B();
a2.x = 2;
a2.f();
// a2.y = 2; – ej
// a2.g(); – ej
B b1 = new B();
b1.x = 2;
b1.y = 2;
b1.g();
```

- En klass kan ärva en annan klass. Det innebär att den automatiskt har alla variabler och metoder som den ärvda klassen har.
- Den ärvande klassen kan sedan lägga till nya variabler och metoder, d.v.s. utöka den ärvda klassen.
- Man skriver class B extends A {…} om klass B ska ärva klass A.
- Klasser kan ärva i flera nivåer (B ärver A, C ärver B etc). En klass B som ärver en klass A i ett eller flera steg kallas sub-klass till A, medan A kallas super-klass till B.
- Flera klasser kan ärva samma klass, men en klass kan bara ärva en klass.
- En klass som inte explicit ärver en annan klass ärver javas grundklass, Object. Den är superklass till alla klasser.
- På liknande sätt som för gränssnitt så är, om B är sub-klass till A, ett objekt av klass B också ett objekt av klass A.
- Interface kan också bygga vidare på andra interface och man skriver då extends precis vid klass-arv.

Arv:överskuggning

```
public class A {
  public int x;
  public A() {
    x = 0;
  }
  public A(int x) {
    This.x = x;
  }
  public void f() {
    x = 3;
  }
}
```

```
public class B extends A {
  int y;
  public B() {
    super(2);
    y = 1;
  }
  @Override
  public void f() {
    x = 4;
  }
}
```

```
A a = new A();
// a.x = 0
a.f();
// a.x = 3
B b = new B();
// b.x = 2
b.f();
// b.x = 4
```

•När ett objekt skapas anropas en konstruerare för klassen själv och för alla super-klasser.
•Om inget anges om vilken konstruerare som ska anropas i den ärvda klassen så blir det den utan argument.
•Man kan ange vilken konstruerare som ska anropas i den ärvda klassen genom super(…)
•En ärvande klass kan ersätta definitionen av metoder som definierades i någon superklass. Detta kallas överskuggning (overriding).
Gör man detta rekommenderas man sätta attributet @Override på metoden.
* Metoder som är deklarerade final kan inte överskuggas i en subklass.

Notera, Dynamisk bindning:
B b = new B();
A ab = b;
ab.f(); // Denna avser fortfarande den överskuggade metoden f i klass B! ab är “egentligen” av typ B.
ab.super.f(); // Denna avser metoden f i superklassen A.

Arv: tillgänglighet

```
public class A {
  public int x;
  protected int y;
  private int z;
  public void f() {
    x = y = z = 1;
  }
}
public class B extends A {
  public void g() {
    x = y = 1;
    // z = 1; – ej
  }
}

A a = new A();
a.x = 1;
// a.y = 1; – ej, utanför klass A eller subklass till A.
// a.z = 1; – ej
```

•Klassmedlemmar som saknar tillgänglighetsmodifierare eller är deklarerade som private är inte tillgängliga i subklasser.
•Modifieraren protected kan användas för att göra medlemmar tillgängliga i subklasser (men inte överallt som public gör).

Notera:
Om B också haft en instansvariabel int x, så döljs den ärvda instansvariabeln x från klass A. Denna kan då (i subklassen B) refereras till med super.x.
B b = new B();
int y = b.x; //avser instansvariabeln x i klass B.
int z = b.super.x; // avser instansvaribeln x som ärvts från klass A.
A a = b;
int w = a.x; // här kommer vi åt instansvariabeln x i klass A (oavsett att a “egentligen” refererar till ett objekt av typ B).

Typomvandlingar

```
public class A {
  public int x;
}
public class B extends A {
  public int y;
}

A a1 = new A();
A a2 = new B();
B b2 = (B)a2;
b2.y = 3;
B b1 = (B)a1;  // runtime-fel
```

- Som sagt kan man tilldela en variabel ett objekt som tillhör en subklass/implementerande klass till variabelns klass/interface.
- Om man vill gå i andra riktningen får man explicit omvandla (type cast) typen: (B)x. Då sker en kontroll då programmet körs för att se att objektet är en instans av den angivna typen (eller en subklass till den).
- Man kan ta reda på om ett objekt tillhör en klass eller en av dess subklasser med operatorn instanceof

Abstrakta klasser

```
public abstract class A {
  public int x;
  public abstract int f();
}
public class B extends A {
  public int f() {
    return x;
  }
}

// A a1 = new A(); – ej
B b = new B();
int x1 = b.f();
A a2 = new B();
int x2 = a2.f();
```

- Metoder kan deklareras utan definition med modifieraren abstract. Meningen är att en sub-klass ska ge en definition av metoden.
- Om en klass innehåller deklaration av abstrakta metoder så måste hela klassen ha modifieraren abstract.
- Man kan inte skapa objekt av en abstrakt klass.
- Abstrakta klasser är tänkta att utgöra en gemensam grund för konkreta subklasser.
- En abstrakt klass är ett mellanting mellan en konkret klass och ett interface.

Interface med default

```
public interface MyInterface{
  String method1();
  int method2(int arg);
  default boolean method3(int arg){
     return arg > 0; }
```

```
public class A implements MyInterface{
  public int x;

  public String method1() {
    Integer.toString(x);
  }
  public int method2(int arg){
     return x+arg;
  }
}
```

```
A a = new A();
String s = a.method1(); //OK: --> “0”
int y = a.method2(1);   //OK: --> 1
boolean b = a.method3(5); // OK: --> true
```

I senare versioner av Java (fr.o.m. Java 8) har man lagt till ytterligare en variant av Interfaces som påminner mycket om Abstrakta klasser: Interface med default-implementationer. Dessa påminner mycket om abstrakta klasser, men eftersom de är interfaces kan en och samma klass implementera flera Interfaces (se kursbok, kan leda till kompileringsfel om namnkonfliker uppstår).

I exemplet behöver klassen A bara implementera metoderna method1 och method2 från MyInterface. method3 har en default-implementation, så om inget annat anges i klassen A (overrides) används denna.

Anledningen till att man la till denna funktionalitet i Java 8 var att man önskade lägga till fler metoder till vissa Interfaces i Javas bibliotek, utan att samtidigt behöva ändra alla existerande klasser som implementerade interfacet (dessa hade ju annars saknat en metod). Här kan man ju diskutera om denna lösning är elegant eller ej, men den kommer sig av pragmatism. Det är lättare att få en förändring accepterad om det inte kräver att alla andra också måste ändra sin kod...