Föreläsning 7

Muterbara och icke-muterbara klasser
Delegering
Gränssnittet för en klass
Metoden equals()

Muterbara kontra icke-muterbara klasser

Objekt som tillhör en icke-muterbar klass kan inte ändra sitt tillstånd, d.v.s. det tillstånd som objektet får när det skapas bibehålls under objektets hela livstid.

Ett icke-muterbart objekt har således ett *garanterat beteende*, dess värde kan avläsas hur många gånger som helst och samma resultat erhålls varje gång.

Muterbara kontra icke-muterbara klasser

Är klassen `Employee` icke-muterbar?

```
import  java.util.Date;
 public class Employee {
    private String name;
    private double salary;
    private Date hireDate;
    public Employee (String name, Date hireDate, double salary) {
       this.name = name;
       this.salary = salary;
       this.hireDate = hireDate;
    }//constructor
    public String getName() {
       return name;
    }//getName
    public double getSalary() {
       return salary;
    }//getSalary
    public Date getHireDate() {
       return hireDate;
    }//getHireDate
 }//Employee
```

Anm: `String` är en icke-muterbar klass och `Date` är en muterbar klass.

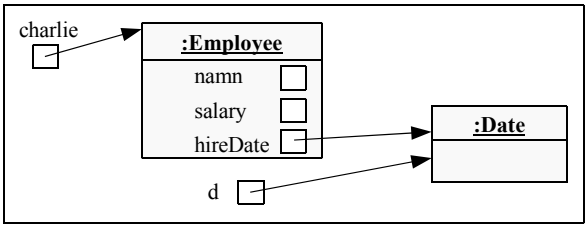
Muterbara kontra icke-muterbara klasser

Klassen har inga mutator-metoder, men den är likväl muterbar!

Avläsningsmetoderna verkar harmlösa, men de utgör en dold fara. Betrakta kodsekvensen nedan:

```
Employee charlie = . . .;
Date d = charlie.getHireDate();
d.setTime(t);
```

Vad som inträffar illustreras av följande bild



Den interna representationen i objektet är exponerad!

Muterbara kontra icke-muterbara klasser

I och med att satsen `d.setTime(t)` exekveras ändras tillståndet av objektet `charlie`. Detta är troligen inte vad utvecklaren av klassen `Employee` tänkt sig!

Botemedlet är att klona objektet innan det ges ut:

```
public Date getHireDate() {
  return hireDate;
  return (Date) hireDate.clone();
}
```

Kloning är mer klurigt än vad man vid en första anblick kan tro. Vi återkommer till detta senare.

Muterbara kontra icke-muterbara klasser

Är klassen icke-muterbar nu, när en klonad kopia av `hireDate`-objektet fås vid avläsning?

Nej! Betrakta nedanstående kod:

```
Date d = new Date();
Employee e = new Employee(”Kalle Anka”, d);
d.setTime(...);
```

En ondskefull programmerare skapar ett `Date`-objekt, behåller en referens till objekt själv och skickar denna referens till konstruktorn för klassen `Employee`. Således kommer denna referens att bli delad.

Lösningen är att låta konstruktorn klona referensen till `Date`-objektet:

```
public Employee (String name, Date hireDate, double salary) {
   this.name = name;
   this.salary = salary;
   this.hireDate = hireDate;
   this.hireDate = (Date) hireDate.clone();
}//constructor
```

Muterbara kontra icke-muterbara klasser

Slutsats:

En accessor-metod skall aldrig returnera ett muterbart attribut. I stället skall en klonad kopia av attributet returneras. Emellertid är det säkert att returnera attribut av primitiva typer och referenser till icke-muterbara objekt.

Icke-muterbara klasser

Vilka klasser skall vara icke-muterbara?

Det generella svaret är ”så många som möjligt”.

Särskilt gäller detta klasser på den lägsta nivån, dvs klasser som representerar data. Dessa objekt är ”värdeobjekt” och skall vara icke-muterbara. Omslagsklasserna för de primitiva typerna, t.ex `Integer` och `Double` är exempel på sådana icke-muterbara klasser.

Icke-muterbara klasser

När man väl bestämt sig för att göra en klass `A` icke-muterbar, hur skall man gå tillväga för att förvissa sig om att den verkligen blir icke-muterbar? Här är de tre viktigaste sakerna som skall göras:

- Gör alla instansvariabler i `A` privata.
- Uteslut alla mutator-metoder.
- Förhindra att subklasser till `A` överskuggar metoder genom att göra metoderna **final**, eller förhindra arv genom att göra klassen `A` **final**.

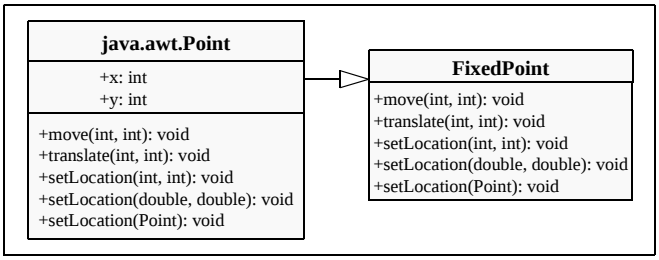
Det finns dock, som vi sett, en del andra subtila problem som måste beaktas.

En nackdel med icke-muterbara objekt är att om man skulle vilja ändra dess tillstånd (värde) måste man skapa ett helt nytt objekt, även om vi aldrig kommer att använda det ursprungliga objekt igen.

Delegering

Antag att du behöver en klass som är identisk med en muterbar klass, förutom att klassen måste vara icke-muterbar. Som exempel kan vi säga att vi behöver en klass `FixedPoint` som är en icke-muterbar version av klassen `java.awt.Point`. Två förslag på att åstadkomma detta är:

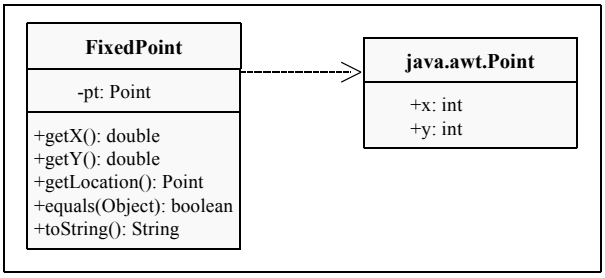
1. Strunta i klassen `Point` och skriv klassen `FixedPoint` från grunden.
2. Låt `FixedPoint` vara en subklass till `Point`, och överskugga metoderna `move`, `translate` och `setLocation` på så sätt att de inte gör någonting alls.



Varför är dessa förslag dåliga?

Delegering

En tredje och betydligt bättre metod är att använda delegering.



Detta innebär att vi gör en ”wrapper”-klass:

- klassen tillhandahåller ett gränssnitt som endast innefattar read-only metoder
- klassen har ett objekt av klassen `Point` i vilken klassen sparar sin data.

Delegering

Ett första, men felaktigt, implementering:

```
public class FixedPoint {
   private Point pt;
   public FixedPoint(Point p) }
      this.pt = p;
   }//constructor
   public FixedPoint(int x, int y) {
      this.pt = new Point(x,y);
   }//constructor
```

WRONG!

```
   public double getX() {
      return pt.getX();
   }//getX
   public double getY() {
      return pt.getY();
   }//getY
   public Point getLocation() {
      return pt;
   }//getLocation
}//FixedPoint
```

Vid en första anblick ser denna klass ut att vara icke-muterbar. Men är den det?

Delegering

Vi har exakt samma problem som i vårt tidigare exempel med klassen `Employee`.

Vad sker när nedanstående kod exekveras?

```
FixedPoint fp = new FixedPoint(3,4);
Point loc = fp.getLocation();
loc.x = 5;
Point p = new Point(3,4);
FixedPoint fp = new FixedPoint(p);
p.x = 5;
```


Delegering

En korrekt implementation av `FixedPoint` har följande utseende:

```
public final class FixedPoint {
   private final Point pt;
   public FixedPoint(Point p) }
      this.pt = new Point(p);
   }//constructor
   public FixedPoint(int x, int y) {
      this.pt = new Point(x,y);
   }//constructor
   public double getX() {
      return pt.getX();
   }//getX
   public double getY() {
      return pt.getY();
   }//getY
   public Point getLocation() {
      return new Point(pt);
   }//getLocation
}//FixedPoint
```

Användning av final

Man kan markera en instansvariabel (eller klassvariabel) som **final** för att förhindra att variabeln förändrar sitt värde efter det att den har skapats.

Detta är ett bra sätt att visa att klassen är icke-muterbar.

Dock betecknar **final**-modifieraren endast att innehållet i variabeln är ett konstant värde. *Tillståndet hos objektet som variabeln refererar till kan förändras!!*

Även om vi deklarerar instansvariablerna i klassen `Employee` till **final**, som i exemplet bredvid, kvarstår samma problem som tidigare vad avser möjligheten att förändra tillståndet i `Date`-objektet som refereras av instansvariabeln `hireDate`.

```
public class Day {
   private final int day;
   private final int month;
   private final int year;
   . . .
}//Day
```

```
import  java.util.Date;
public class Employee {
   private final String name;
   private final double salary;
   private final Date hireDate;
   . . .
}//Employee
```


Användning av final

Man kan markera en parameter som **final**. Detta betyder att metoden inte får förändra parameterns värde.

Man kan markera en metod med **final**. Detta betyder att metoden inte kan överskuggas i en subklass.

Man kan deklarera en klass med **final**. Detta betyder att klassen inte kan användas som superklass.

Att designa en klass

Hur en klass skall utformas och implementeras måste avgöras från två olika infallsvinklar – utvecklarens perspektiv och användarens perspektiv. Programmerare utvecklar klasser för att de skall användas av andra programmerare.

Den som utvecklar klassen har sina speciella målsättningar med klassen, såsom effektiva algoritmer och lätthanterlig kod.

De som använder klassen vill kunna förstå och använda klassen utan att bry sig om den interna representationen. De vill ha en uppsättning metoder som är tillräckliga för att kunna lösa sin programmeringsuppgift, men som samtidigt är så begränsad att det är enkelt att förstå hur metoderna i klassen skall användas på bästa sätt.

I mindre programmeringsprojekt, som labbarna på denna kurs, är det samma programmerare som både utvecklar en klass och sedan använder klassen. Rollerna som säljare och kund upplevs som diffus. Men försök att särskilja dessa båda perspektiv.

The Newspaper Metaphor

En klass skall vara utformad som en välskriven tidningsartikel.

Artikeln läses uppifrån och ned. Högst upp finns en rubrik som talar om vad artikeln handlar om. Ingressen ger en översikt och kontext om historien som beskrivs, utan att ta upp detaljerna. Fortsätter man läsa får man slutligen hela historien med alla detaljer.

En tidning består av många artiklar, var av de flesta är mycket korta. Somliga är längre, men mycket få är så långa att de upptar mer än en sida. Detta gör en tidning användarvänlig. Skulle en tidning vara en enda långt oorganiserad gytter av text skulle ingen läsa tidningen.

Det samma gäller källkoden för ett program!

Vad skall en klass tillhandahålla?

Vi skall här diskutera ett antal bedömningsgrunder för att avgöra hur bra gränssnittet för en klass är.

- sammanhållning (*cohesion*)
- bekvämlighet (*convenience*)
- tydlighet (*clarity*)
- konsistens (*consistency*)
- fullständighet (*completeness*)

Sammanhållning (cohesion)

En klass skall ha *ett väl avgränsat ansvarsområde*, d.v.s. vara en abstraktion av ett avskilt koncept eller fenomen.

Alla operationer i klassen skall passa logiskt ihop för att stödja ett särskilt syfte.

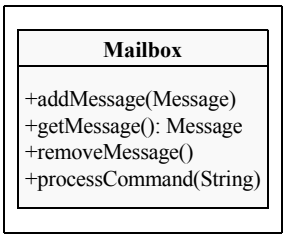
Denna princip kallas *The Single Responsibility Principle*, en annan beteckning för samma sak är *Separation of concern*.

> *A class has a single responsibility. It represents one thing. It does one thing only and does it well. (Tom DeMarco)*

Sammanhållning (cohesion)

Betrakta klassen `Mailbox`:

Metoden `processCommand` skiljer sig från alla andra metoder i klassen `Mailbox`.



De övriga metoderna utför operationer på en och samma abstraktion; nämligen en brevlåda som innehåller meddelanden.

Metoden `processCommand` lägger dock ytterligare en egenskap till brevlådan, nämligen att handha kommandon.

Det vore mycket bättre att ha en annan klass som handhar kommandon, och låta brevlådan göra vad den är bra på, d.v.s. att lagra meddelanden.

The Single Responsibility Principle

A class should have only one reason to change. (Robert Martin)

Om en klass har flera ansvarsområden kommer de olika ansvars-områdena att bli kopplade till varandra. Varje ansvarsområde kan leda till förändringar i klassen p.g.a. ändrade krav. Förändringar i ett ansvarsområde kan påverka klassens förmåga att hantera de andra ansvarsområdena.

En klass som har många ansvar kommer också att ha kopplingar till många andra klasser. Klassen riskerar att behöva modifieras då andra klasser modifieras.

En klass med fler än ett ansvarsområde ger bräcklighet.

Single Responsibility Principle

> *One of the criteria I use is to try to describe a class in 25 words or less, and not to use "and" or "or". If I can't do this, then I may actually have more than one class. (Brian Button)*

Klasser som försöker göra många saker, blir inte bara sårbara för förändringar, utan de blir också oöverblickbara och svåra att förstå.

Slutsats: Bryt ner stora saker för att minska komplexiteten.

Bekvämlighet (*convenience*)

Ett gränssnitt kan vara komplett i den meningen att det tillhandahåller tillräckligt med operationer för att utföra allt nödvändigt. Men som användare av klassen skall man inte behöva utföra en serie av operationer för att lösa en begreppsmässigt enkel uppgift.

Ett bra gränssnitt skall inte bara tillhandahåll operationer för att utföra uppgiften, utan operationer som *gör det enkelt* att lösa uppgiften.

Att läsa indata från System.in är en mycket vanlig uppgift i ett program. Tyvärr har System.in ingen metod för att läsa in en rad. Före Java 5.0 var man tvungen till att innesluta System.in i en InputStreamer och sedan i en BufferedReader, vilket var mycket obekvämt. Problemet löstes med att slutligen införa klassen Scanner. Men varför tog det så lång tid?

Tydlighet (clarity)

Gränssnittet till en klass skall vara tydligt och begripligt.
Förvirrade programmerare skriver dålig och felaktig kod.

Tydlighet (clarity)

Låt oss titta på ett exempel från Javas standardbibliotek.

```
LinkedList<String> list = new LinkedList<String>();
list.add("A");
list.add("B");
list.add("C");
```

För att genomlöpa listan använder vi en iterator:

```
ListIterator<String> iterator = list.listIterator();
while (iterator.hasNext())
   System.out.println(iterator.next());
```

Iteratorns position är en position mellan två element i listan, precis som |-markören i ett ordbehandlingssystem står mellan två tecken. Metoden `add` i `ListIterator` lägger ett element före iteratorns position, exakt som i ett ordbehandlingssystem.

```
ListIterator<String> iterator = list.listIterator(); // |ABC
iterator.next(); // A|BC
iterator.add("X"); // AX|BC
```

Tydlighet: fortsättning på exemplet

Men metoden `remove` i `ListIterator` är allt annat än intuitiv. I ett ordbehandlingssystem tar `remove` bort elementet till vänster om markören. Således skulle man förvänta sig följande:

```
ListIterator<String> iterator = list.listIterator(); // |ABC
iterator.next(); // A|BC
iterator.add("X"); // AX|BC
iterator.remove(); // A|BC
iterator.remove(); // |BC
```

Dock är båda anropen `remove()` ogiltiga!

I dokumentationen av `remove` står:

> *”Removes from the list the last element that was returned by next or previous. This call can only be made once per call to next or previous. It can be made only if add has not been called after the last call to next or previous.”*

För att ta bort dessa två element måste man, för vart och ett av elementet, först hoppa över elementet och sedan omedelbart därefter ta bort det!

Konsistens (*consistency*)

Operationerna i en klass skall vara konsistenta med varandra avseende namn, parametrar, returvärde samt beteende.

I Javas standardbibliotek finns exempel på inkonsistenser.

För att ett objekt (som representerar ett datum) i klassen `GregorianCalendar` anger man

new GregorianCalendar(year, month – 1, day)

Alltså förväntar sig konstruktorn att månad anges som ett heltal mellan 0 och 11, däremot skall dag anges som ett heltal mellan 1 och 31!

Mycket logiskt!

Ett annat exempel är metoden `substring` i klassen `String`:

substring(start, end)	returnerar delsträngen med startposition `start` och slutposition `end`−1.

Fullständighet (*completeness*)

Gränssnittet till en klass skall vara komplett. Det skall stödja alla operationer som på ett naturligt sätt är associerade med den abstraktion som klassen representerar.

Låt oss titta på klassen `java.util.Date`. Antag vi har kodsegmentet

```
Date start = new Date();
// Do some work
Date stop = new Date();
```

Vi vill nu ta reda på hur många millisekunder det har förflutit mellan tidpunkten då `start` skapades och tidpunkten då `stop` skapades. Klassen `Date` har metoderna `before` och `after` som kan användas för att ta reda på om `start` skapats före eller efter `stop`. Men det finns ingen metod för att beräkna skillnaden mellan `start` och `stop`.

Det är ingen allvarlig brist, då vi kan beräkna tiden vi söker genom att använda metoden `getTime()`

```
long difference = stop.getTime() -start.getTime();
```

men det hade känts naturligare att kunna skriva

```
long difference = stop.getDifference(start);
```


The Uniform Access Principle

Denna princip säger att den service som en klass tillhandahåller skall vara åtkomlig med en enhetlig notation, som inte avslöjar om resultatet åstadkoms genom beräkning eller om det finns lagrat i minne.

```
public class Rectangle {
   . . .
   public final int area;
   public Rectangle(int width, int height) {
      this.width = width;
      this.height = height;
      area = width * height;
   }
   public int getPerimeter() {
      return 2 * (width + height);
   }
   . . .
}//Rectangle
```

```
Rectangle rek = new Rectangle(5, 12);
 . . .
int theArea = rek.area;
int thePerimeter = rek.getPerimeter();
```

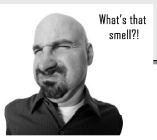
Bryter mot *uniform access*

Expert pattern

Denna princip säger att det objekt som har den information som behövs för att utföra en uppgift skall utföra uppgiften.

```
public class Phone {
   private final String unformattedNumber;
   public Phone(String unformattedNumber) {
      this.unformattedNumber = unformattedNumber;
   }
   public String getAreaCode() {
      return unformattedNumber.substring(0,3);
   }
   public String getPrefix() {
      return unformattedNumber.substring(3,6);
   }
   public String getNumber() {
      return unformattedNumber.substring(6,10);
   }
}
```

```
public class Customer {
   private Phone mobilePhone;
   public String getMobilePhoneNumber() {
      return "(" +  mobilePhone.getAreaCode()
               + ") " + mobilePhone.getPrefix()
               + "-" + mobilePhone.getNumber();
   }
   . . .
}
```



Expert pattern

```
public class Phone {
   private final String unformattedNumber;
   public Phone(String unformattedNumber) {
      this.unformattedNumber = unformattedNumber;
   }
   public String getAreaCode() {
      return unformattedNumber.substring(0,3);
   }
   public String getPrefix() {
      return unformattedNumber.substring(3,6);
   }
   public String getNumber() {
      return unformattedNumber.substring(6,10);
   }
  public String toFormattedString() {
     return "(" + getAreaCode() + ") " + getPrefix() + "-" + getNumber();
   }
}
```

```
public class Customer {
   private Phone mobilePhone;
   public String getMobilePhoneNumber() {
      return mobilePhone.toFormattedString();
   }
   . . .
}
```

Principen “Tell, don't ask”

Denna princip säger att ett objekt inte skall fråga ett annat objekt om dess tillstånd, fatta något beslut baserat på detta tillstånd och därefter tala om för det andra objektet vad det skall göra.

```
public class Lung {
   private int oxygenAmount;
   . . .
   public int getOxygenAmount() {
      return oxygenAmount;
   }
   public void breatheSomeAir() {
      // do the necessary action to breathe
   }
}
```


```
public class Human {
   private Lung theLung;
   . . .
   public void breathe() {
      Lung myLung = getLung();
      if (myLung.getOxygenAmount() < 0) {
         myLung.breatheSomeAir();
      }
   }
   public Lung getLung() {
      return theLung;
   }
}
```


Principen “Tell, don't ask”

```
public class Lung {
   private int oxygenAmount;
   . . .
   public int getOxygenAmount() {
      return oxygenAmount;
   }
   public void breatheSomeAir() {
      if (getOxygenAmount() < 0) {
         // do the necessary action to breathe
   }
}
```

```
public class Human {
   private Lung theLung;
   . . .
   public void breathe() {
      Lung myLung = getLung();
      myLung.breatheSomeAir();
   }
   public Lung getLung() {
      return theLung;
   }
}
```

Responsibility implies non-interference. (Timothy Budd)

Law of Demeter: *Don't talk to strangers*

Ju fler klasser som en klass samverkar med, desto bräckligare och svårare att förstå bli klassen.

Tanken med *Law of Demeter* är att en klass skall veta så lite som möjligt om den interna strukturen hos andra klasser. Genom att undvika att anropa metoder på ett objekt som returneras från en annan metod minimerar man mängden beroenden och ser till att objekten inte bryter inkapslingen av data.

Law of Demeter säger att en metod `m` i klassen `C` endast skall samverka med objekt som:

1. skapats av `m`
2. är instansvariabler i `C`
3. är argument till `m`
4. är objektet själv (**this**)

Law of Demeter: *Don't talk to strangers*

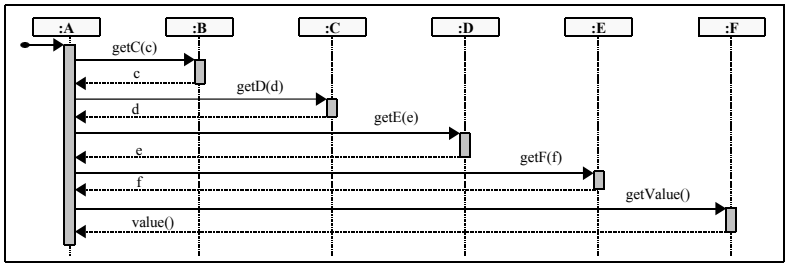
Betrakta nedanstående anrop

```
Balance balance = centralControl.getBank(b).getBranch(r).
                        getCostmer(c).getAccount(a).getBalance();
dog.getBody().getTail().wag();
```

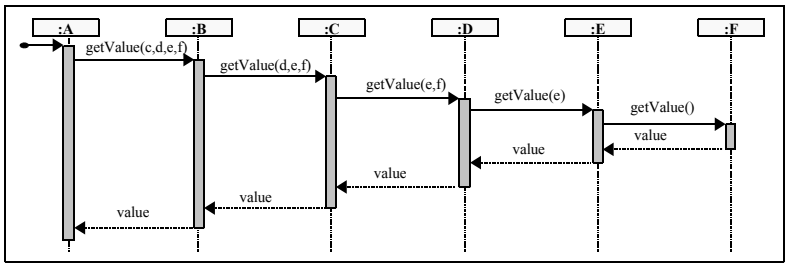
Det första anropet är beroende av 5 objekt och det andra anropet är beroende av 3 objekt. Enligt *Law of Demeter* skall dessa anrop ersättas med följande anrop:

```
Balance balance = centralControl.getBalance(b, r, c, a);
dog.wagTail();
```

Law of Demeter: *Don't talk to strangers*



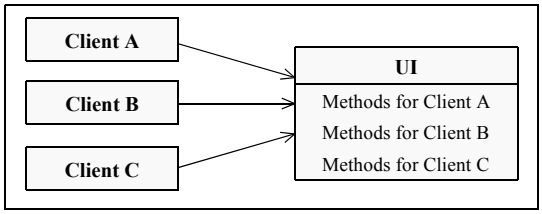
***T**alk only to your immediate friends*



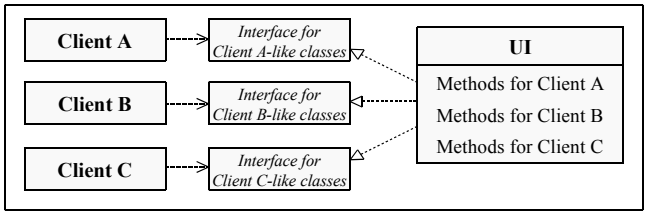
Interface Segregation Principle (ISP)

Classes should not be forced to depend on methods that they do not use.

Design som bryter mot ISP



Design som följer ISP



Klassen Object

I Java är alla klasser subklasser till `java.lang.Object`.

Klassen `Object` innehåller följande metoder

clone	skapar en kopia av objektet.
equals	jämför om objektet är lika med ett annat objekt.
getClass	returnerar vilken klass objektet tillhör vid runtime.
hashCode	returnerar hashkoden för objektet.
toString	returnerar en `String`-representation av objektet.
samt	ett antal metoder för synkronisering i multitrådade program.

För att en klass skall betraktas som fullständig skall klassen överskugga metoderna `toString`, `clone`, `equals` och `hashCode`. Särskilt gäller detta om klassen kommer att användas i ett klassbibliotek.

Metoden equals()

I klassen `Object` har metoden `equals()` följande utseende:

```
public boolean equals(Object obj) {
   return this == obj;
}//equals
```

Detta betyder att om vi har två objekt betraktas de som lika endast om de är *alias*.

Alla klasser behöver därför definiera vad som menas med att ett objekt är lika med ett annat objekt.

Metoden equals()

`equals`-metoden används på många ställen i bl.a. `Collection`-klasserna.

Här är ett typiskt exempel på användning av `equals`:

```
private Object[] elementData  =  . . .;
. . .
public int indexOf(Object elem) {
   for (int i = 0; i < elementData.length ; i++)
      if (elem.equals(elementData[i])
         return i;
   }
   return -1;
}//indexOf
```

Alla klasser behöver definiera vad som menas med att ett objekt av klassen är lika med ett annat objekt.

Metoden equals()

Låt oss titta på klassen `Triangle`:

```
import  java.awt.Point;
public class Triangle {
   private Point p1, p2, p3;
   public Triangle(Point p1, Point p2, Point p3) {
      if (p1 == null)
         p1 = new Point(0,0);
      if  (p2 == null)
         p2 = new Point(0,0);
      if  (p3 == null)
         p3 = new Point(0,0);
      this.p1 = p1;
      this.p2 = p2;
      this.p3 = p3;
   }//constructor
   . . .
}//Triangle
```

Metoden equals()

Vid en första anblick kan det tyckas vara enkelt att avgöra likheten mellan två objekt av klassen `Triangle` – om samtliga tre hörnpunkter är lika i de båda trianglarna borde trianglarna vara lika. Detta leder således till att `equals`-metoden skulle få följande utseende:

```
//-- Felaktig --//
public boolean equals(Object otherObject) {
    if ( !(otherObject instanceof Triangle))
       return false;
    Triangle other = (Triangle) otherObject;
    return p1.equals(other.p1)
            && p2.equals(other.p2)
            && p3.equals(other.p3) ;
}//equals
```

WRONG!

Metoden equals()

Låt oss nu göra ett litet testprogram

```
Triangle t1 = new Triangle(new Point(0,0), new Point(1,1), new Point(1,0));
Triangle t2 = new Triangle(new Point(2,2), new Point(4,3), new Point(1,0));
Triangle t3 = new Triangle(new Point(0,0), new Point(1,1), new Point(1,0));
System.out.println(t1.equals(t2));
System.out.println(t1.equals(t3));
```

Utskriften blir vad vi förväntar oss:

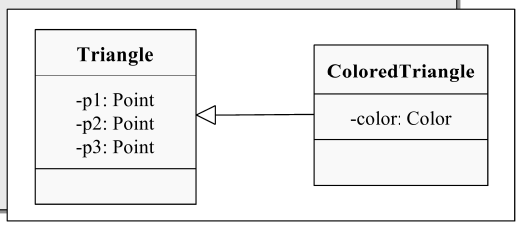
```
false
true
```

d.v.s. att `t1` och `t2` är olika, samt att `t1` och `t3` är lika.

Metoden equals()

Låt oss nu krångla till det hela genom att införa en subklass `ColoredTriangle` till klassen `Triangle`.

```
import  java.awt.Point;
import  java.awt.Color;
public class ColoredTriangle extends Triangle {
   private Color color;
   public ColoredTriangle(Color color, Point p1, Point p2, Point p3) {
      super(p1, p2, p3);
      if  (color == null)
         color = Color.RED;
      this.color = color;
   }//constructor
   . . .
}//ColoredTriangle
```



Metoden equals()

Låt oss nu göra ett nytt testprogram:

```
Triangle t1 = new Triangle(new Point(0,0), new Point(1,1), new Point(1,0));
Triangle t2 = new Triangle(new Point(2,2), new Point(4,3), new Point(1,0));
ColoredTriangle t3 = new ColoredTriangle(Color.GREEN, new Point(0,0),
                                         new Point(1,1), new Point(1,0));
System.out.println(t1.equals(t2));
System.out.println(t1.equals(t3));
```

Utskriften blir

```
false
true
```

d.v.s. att `t1` och `t3` är lika. Men är verkligen ett objekt av klassen `Triangle` lika med ett objekt av klassen `ColoredTriangle`??
Vi måste i metoden `equals` ta hänsyn till vilka typer som objekten i jämförelsen har.

Metoden equals()

I klassen `Object` finns metoden `getClass()` som returnerar vilken runtime-klass ett objekt har. Denna metod kommer nu väl till pass

```
//-- Fortfarande felaktig --//
public boolean equals(Object otherObject) {
   if (otherObject.getClass() != this.getClass())
      return false;
   Triangle other = (Triangle) otherObject;
   return p1.equals(other.p1) && p2.equals(other.p2) && p3.equals(other.p3);
}//equals
```

Testkör vi nu med samma exempel som tidigare får vi med denna variant av `equals`-metoden utskriften

```
false
false
```

Detta resultat är vad vi vill ha. Dock är `equals`-metoden fortfarande inte korrekt!

Metoden equals()

I *The Java Language Specification* anges att `equals`-metoden skall ha följande egenskaper:

Skall vara *reflexiv*:	För varje icke-**null** referens `x` skall det gälla att `x.equals(x)` returnerar **true**.
Skall vara *symmetrisk*:	För alla referenser `x` och `y` skall det gälla att `x.equals(y)` returnerar **true** om och endast om `y.equals(x)` returnerar **true**.
Skall vara *transitiv*:	För alla referenser `x`, `y` och `z` skall gälla att om `x.equals(y)` returnerar **true** och `y.equals(z)` returnerar **true** så skall också `x.equals(z)` returnera **true**.
Skall vara *konsistent*:	Om objekten till vilka `x` och `y` refererar inte har förändrats skall upprepade anrop av `x.equals(y)` returnera samma värde.

För alla icke-**null** referenser `x` skall gälla att `x.equals(null)` skall returnera **false**.

Metoden `equals()`

I `equals`-metoden för klassen `Triangle` måste vi ta hand om fallet då objektet som jämförelsen utförs mot är **null**.

```
//-- Slutlig version av equals i klassen  Triangle--//
public boolean equals(Object otherObject) {
   if (otherObject == null)
      return false;
   if (otherObject.getClass() != this.getClass())
      return false;
   Triangle other = (Triangle) otherObject;
   return p1.equals(other.p1) && p2.equals(other.p2)
            && p3.equals(other.p3) ;
}//equals
```


Metoden equals()

Vad händer när följande kod körs?

```
ColoredTriangle ct1 = new ColoredTriangle(Color.RED, new Point(0,0),
                                          new Point(1,1), new Point(1,0));
ColoredTriangle ct2 = new ColoredTriangle(Color.BLUE, new Point(0,0),
                                          new Point(1,1), new Point(1,0));
System.out.println(ct1.equals(ct2));
```

Utskriften blir

true

`ColorTriangle` ärver `equals`-metoden från `Triangle`, och metoden `equals` i `Triangle` beaktar inte komponenten `color`.

Även klassen `ColoredTriangle` måste överskugga `equals`-metoden!!

Metoden equals()

Metoden `equals` i klassen `ColoredTriangle` får följande utseende:

```
//-- equals i klassen  ColoredTriangle--//
public boolean equals(Object otherObject) {
   return super.equals(otherObject) &&
            color.equals(((ColoredTriangle) otherObject).color) ;
}//equals
```

Den som är observant, har naturligtvis konstaterat att vi definierat likhet mellan två trianglarna genom att instansvariablerna `p1` är lika, `p2` är lika och `p3` är lika. Men trianglarna kan vara lika även i andra situationer. Detta överlåtes dock som övning.

Ett dilemma

Antag att vi har tre variabler `t1`, `t2`, och `t3` som är deklarerade av typen `Triangle`.

Antag också att de trianglar som dessa variabler refererar till, samtliga har samma hörnpunkter. Betraktade som objekt av typen `Triangle` är de således lika.

Men antag nu att `t2` i verkligheten refererar till ett objekt av typen `ColoredTriangle`, utan att användaren är medveten om detta.

Detta innebär att `t1.equals(t3)` och `t2.equals(t3)` ger olika värden (**true** respektive **false**), vilket inte är vad användaren förväntar sig.

Våra `equals`-metoder bryter mot *Liskov Substitution Principle*!

Lösning på dilemmat

Det finns ingen bra lösning på detta dilemma, utan är beroende på hur klasserna skall användas:

1. ta bort `equals`-metoden i `ColoredTriangle`**.** Innebär att två objekt av klassen `ColoredTriangle` betraktas som lika även om de har olika färg.
2. ta bort `equals`-metoden i `Triangle`. Detta innebär att två objekt av klassen `Triangle` betraktas som lika endast om de är alias.
3. ta bort superklass/subklass förhållandet mellan `Triangle` och `ColoredTriangle`. Innebär att vi förlorar möjlighet till polymorfism.
4. acceptera att vi bryter mot LSP.

Metoden hashCode()

Metoden `hashCode()` skall alltid överskuggas när man överskuggar `equals`-metoden. Motiveringen är att när man lagrar ett objekt i en hashtabell skall två objekt som är lika ha samma hashkod för att hamna på samma plats i tabellen. Platsen ges av värdet som metoden `hashCode` returnerar, vilket är ett stort heltal.

Om x.equals(y), så är x.hashCode() == y.hashCode()

Har man en klass för vilken man skall omdefiniera metoden `hashCode` räcker det i allmänhet att nyttja attributens hashkoder och addera dessa. Möjligen att också multiplicera koderna med ett primtal innan additionen.

```
//-- hashCode för klassen Triangle  --//
public int hashCode() {
  return 7*p1.hashCode() + 11*p2.hashCode() + 13*p3.hashCode();
}//hashCode

//-- hashCode för klassen ColoredTriangle  --//
public int hashCode() {
  return 17*super.hashCode() + 19*color.hashCode();
}//hashCode
```