Enumeration

Uppräkningstyper, enumeration types

```
public static final int SEASON_WINTER = 0;
public static final int SEASON_SPRING = 1;
public static final int SEASON_SUMMER = 2;
public static final int SEASON_FALL   = 3;
```

Inte bra!

```
JLabel lab = new JLabel("Konstigt", SEASON_WINTER);
```

Använd uppräkningstyper istället:

```
enum Season { WINTER, SPRING, SUMMER, FALL }
```

- En *klass* med fyra konstanta fördefinierade object.
- Privat konstruktor.

```
Season se = Season.SUMMER;
If (se == Season.SPRING)

se = new Season();      // FEL!!
```

1

Enum ger inte varje grej en siffra utan skapar referenser till varje objekt.

104 +50

You should always use enums when a variable (especially a method parameter) can only take one out of a small set of possible values. Examples would be things like type constants (contract status: "permanent", "temp", "apprentice"), or flags ("execute now", "defer execution").

If you use enums instead of integers (or String codes), you increase compile-time checking and avoid errors from passing in invalid constants, and you document which values are legal to use.

BTW, overuse of enums might mean that your methods do too much (it's often better to have several separate methods, rather than one method that takes several flags which modify what it does), but if you have to use flags or type codes, enums are the way to go.

As an example, which is better?

```
/** Counts number of foobangs.
 * @param type Type of foobangs to count. Can be 1=green foobangs,
 * 2=wrinkled foobangs, 3=sweet foobangs, 0=all types.
 * @return number of foobangs of type
 */
public int countFoobangs(int type)
```

versus

```
/** Types of foobangs. */
public enum FB_TYPE {
 GREEN, WRINKLED, SWEET,
 /** special type for all types combined */
 ALL;
}

/** Counts number of foobangs.
 * @param type Type of foobangs to count
 * @return number of foobangs of type
 */
public int countFoobangs(FB_TYPE type)
```

In the second example, it's immediately clear which types are allowed, docs and implementation cannot go out of sync, and the compiler can enforce this.

Relationer

Det finns tre sorters relationer:

- Har
- Känner till
- Är

Har

Om man har ett objekt och detta objekt internt innehåller ett lite mindre objekt vilket utgörs av en del. Det stora objektet **har** andra.

Känner till

Ett objekt känner till ett annat.

Är

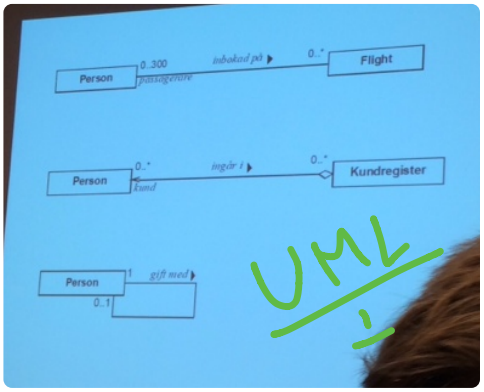
Ett objekt är som ett annat objekt fast lite till.

UML

Känner till
Strecket säger att det finns en relation, inte mer eller mindre.
Texten under säger vilken roll personen har.
Siffran säger om det finns några begränsningar för personen.
En flight i sin tur kan bara innehålla 0..300 passagerare.

Pilarna talar om riktningen.
Kundregistret har referenser till massa kunder. Det är kundregistret som "känner till" sina kunder.
Den tomma vita symbolen visar att det är en samling, alltså att den "känner till", men personerna i sin tur känner inte till registret.

Den sista rutan visar en "känner till" relation till sig själv.



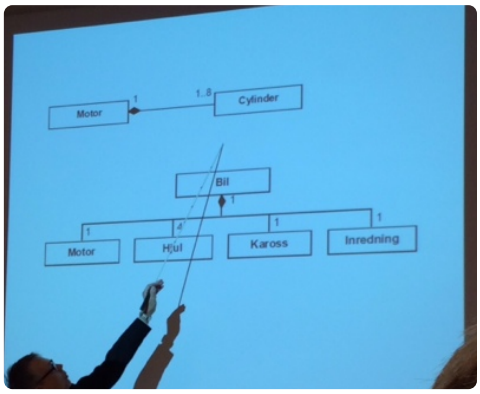
Har

Man får fejka med en "känner till" eftersom vi inte kan skapa ett objekt i ett annat objekt.

Om vi vill att en motor ska innehålla en cylinder och skriver:

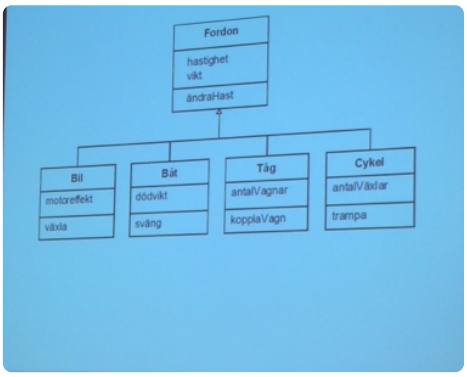
`Cylinder c1 = cylinder;`

skapar vi ett nytt objekt och refererar till objektet istället för att faktiskt skapa objektet i motorn.



Är

Handlar om arv.



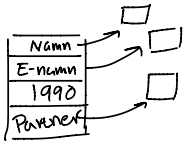
Exempel

```
public class Person {
  private String förnamn;            // "har"
  private String efternamn;          // "har"
  private int föddÅr;                // "har"
  private Person partner;            // "känner till"

  // konstruktorer och metoder ...
}

public class Konto {
  private Person kontohavare;        // "känner till"
  private double saldo;              // "har"
  private double intjänadRänta;      // "har"
  private static double räntesats; // klassvariabel

  // konstruktorer och metoder ...
}
```



Vi får fejka HAR-relationen.

Exempel KursDemo

```
public class KursDemo {

  public static void main(String[] arg) {
    Kurs k = new Kurs(043, "Objektorienterad programmering", "Jan",
                      60, 24, 36, 48);
        int n = k.avläsKursnummer();
        String kn = k.avläsKursnamn();
        Betygskrav kr = k.avläsKrav();
        System.out.println(n + " " + kn + " Godkänd: " + kr.trea);

        kr.trea = 10;
        System.out.println(n + " " + kn + " Godkänd: " + kr.trea);

        kr = k.avläsKrav();
        System.out.println(n + " " + kn + " Godkänd: " + kr.trea);

        String l = k.avläsLärare();
        l = "Nisse";
        System.out.println("Lärare: " + k.avläsLärare());

  }
}
```

```
public class Kurs {
  private int kursnummer;
  private String kursnamn;
  private String lärare;
  private Betygskrav krav;

  public Kurs(int nr, String knamn, String lär, int max,
              int trea, int fyra, int femma) {
    kursnummer = nr;
          kursnamn = knamn;
          lärare = lär;
          krav = new Betygskrav(max, trea, fyra, femma);
  }

  public int avläsKursnummer() {
    return kursnummer;
  }

  public String avläsKursnamn() {
    return kursnamn;
  }

  public String avläsLärare() {
    return lärare;
  }

  public Betygskrav avläsKrav() {
    //return krav;
    return new Betygskrav(krav.max, krav.trea, krav.fyra, krav.femma);
  }
}
```

Objekte av typen String kan aldrig ändras. Om du ändrar (like Nisse) så skapas ett nytt (new) objekt istället.

Var försiktig med att lämna ut referenser, skapa kopia!!

Exempel TidPunktmedAlarm

Tidpunkt med Alarm är en subklass till sin superklass Tidpunkt.

extends talar om att Tma är en subklass

```
public class TidpunktMedAlarm extends Tidpunkt {
  private int at, am, as;
  private boolean ring = false;

  public void sättAlarm (int tim, int min, int sek) {
    if ( tim>=0 && tim<24 && min>=0 && min<60 && sek>=0 && sek<60) {
      at=tim; am=min; as=sek; ring = true;
    }
    else
      System.out.println("Felaktig tidpunkt");
  }

  public void stängAlarm() {
    ring = false;
  }

  @Override
  public void ticka() {
    super.ticka();
    if (ring && avläsTim()==at && avläsMin()==am && avläsSek()==as) {
      Toolkit.getDefaultToolkit().beep();
    }
  }
}
```

"Är" Tidpunkt och Larm!!

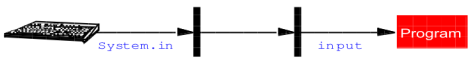
Om man vill ersätta en metod från superklassen kan man "override" den.
Super ticka innebär att vi anropar superklassens metod ticka!

I/O

Man kan kopiera `new BufferReader` och `new InputStreamReader` rakt av. De är samma varje gång.

Språkversion
Kan sättas i programmet t.ex:
`Locale.setDefault(new Locale("sv","SE"));`

Den styr decimalkomma, texter i dialogrutor, tid, mm.




```
import java.io.*;
public class InputDemo {
  public static void main(String[] arg) throws IOException{
    BufferedReader input = new BufferedReader
                           (new InputStreamReader(System.in));
    System.out.println("Vad heter du?");
    String s = input.readLine();
    System.out.println("Hej " + s + "!");
  }
}
```

1

Scanner

Scanner kopplas till stället den ska läsa ifrån (t.ex. System.in) och sen kan du scanna igenom inputen och använda massa flashiga funktioner på det.

Scanner kan användas till massa annat coolt också!

Scanner läser även från en String om man så önskar (se OHIO.pdf bild 4, F5)

```
String s = JOptionPane.showInputDialog"Skriv!");
Scanner sc = newScanner(s);

double sum = 0;
while (sc.hasNextDouble()) {
    double d = sc.nextDouble();
    sum += d;
}
```

```
import java.util.*;    // innehåller klassen Scanner
public class ScanDemo {
  public static void main(String[] arg) {
    // Skapa en scanner som läser från filen
    Scanner sc = new Scanner(System.in);
    System.out.println
          ("Skriv ett antal heltal. Avsluta med Ctrl-Z");
    int sum = 0;       // summan av talen
    while (sc.hasNextInt()) {   // finns det fler tal?
      int i = sc.nextInt();     // ja, läs nästa tal
      sum += i;
    }
    System.out.println("Summa: " + sum);
  }
}
```

3

Utskrifter

Genom att redigera lite kan man få till en snygg text!

Byt ut `System.out.print` mot `System.out.format`

% betyder att det kommer en kod vilken ska ersättas med ett värde.

02d betyder `Skriv ut decimalt, med minst två positioner och eventuella tomrum fylls med nollor.`

4.1f betyder `Skriv ut float, minst fyra positioner och eventuella tomrum lämnas tomma.`

```
System.out.println(k + " medeltemp " + t + " gr");
```

kan ge utskriften

```
9 medeltemp 9.691282744999999 gr
10 medeltemp 10.2888925 gr
```

Redigiering

```
System.format("%02d medeltemp %4.1f gr\n", k, t);
```

Ger

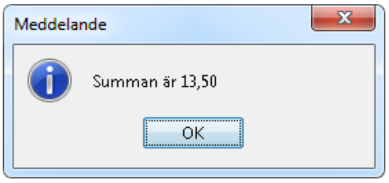
```
09 medeltemp  9,7 gr
10 medeltemp 10,3 gr
```

Koder: Se kursbok eller dokumentation av metoden format på nätet

Alternativt namn för `format` är `printf`

Format finns även i String vilken returnerar - tro det eller ej - en formaterad String,

```
String resultat = String.format("Summan är %.2f", sum);
JOptionPane.showMessageDialog(null, resultat);
```



Felkontroll

Genom att använda `try` så kan man kapsla in de satser som kan generera fel och fånga dem genom `catch`. Skulle det inte inträffa något fel händer ingenting men om någon tabbat sig så genereras ett exception varpå det kontrolleras i `catch` och om det är det "förväntade" felet går programmet vidare.

Skulle man inte ha någon `catch` som är avsedd att ta hand om det aktuella felet så kommer problemet följa med genom hela programmet och förbli o-fångat tills programmet är slut och kraschar.

Felkontroll

```
String s = ...;
try {
  x = Double.parseDouble(s);
}
catch (NumberFormatException e) {
  // hit kommer man om s innehåller felaktiga data
  ...
}
```

```
Scanner sc = new Scanner(...);
try {
  x = sc.nextDouble();
}
catch (InputMismatchException e1) {
  System.out.println("Felaktigt tal: " + sc.next());
}
catch (NoSuchElementException e2) {
  System.out.println("Indata saknas");
}
```