DD1385

Programutvecklingsteknik

Några bilder till föreläsning 2

Innehåll

- Interface
- Abstrakta klasser
- Klasshierarki och typhierarki
- Polymorfism och dynamisk bindning
- Polymorfi-exempel: Schack
- UML-översikt
- Klassen Object i Java

Interface

- Beskriver ett *abstrakt beteende*.
- Definierar en typ i Java
- Nära släkt med abstrakt klass.
- Interface/abstrakta klasser är mycket viktiga i
  - OO–prog i Java.
  - Java-biblioteken
  - Designmönster

Interface

Typdefinition, beskriver ett *abstrakt beteende.*

```
public interface Monster {
    public void walk();
    public void scream();
    public void eat();
}
```

Alla Monster-objekt har metoderna

```
...
Monster aake = ...
...
aake.eat();
aake.scream();
aake.walk();
```

Interface

aake måste vara ett konkret Monster, t.ex. ett CookieMonster

```
class CookieMonster implements Monster {

    public void eat(){
        // asks for cookies and eats them
    }
    public void scream(){
        // muffled sound,
        // mouth is usually full of cookies
    }
    public void walk() {
        // slow and peaceful
    }
}
```

Metoderna måste fyllas med verkligt innehåll

Attribut och fler metoder får finnas i CookieMonster

Interface

Ytterligare en sorts Monster: ScaryMonster

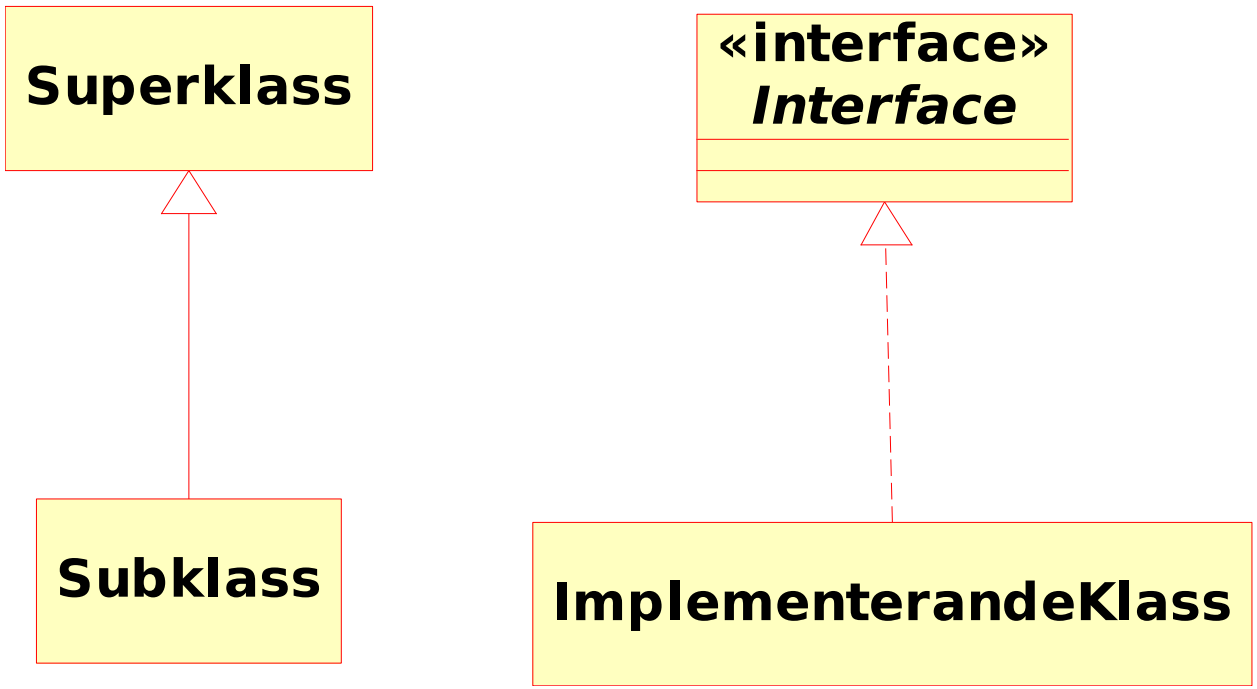
```
class ScaryMonster implements Monster {

    public void eat(){
        // EATS YOU, UNLESS YOU HAVE A PIZZA,
        // THEN IT TAKES YOUR PIZZA
    }
    public void scream(){
        // OOOAAAAAAGHHHHHH - LOUD AND SCARY
    }
    public void walk() {
        // HUGE LEAPS, QUICK AND UNPREDICTABLE
    }
}
```

Metoderna måste fyllas med verkligt innehåll

Attribut och fler metoder får finnas i CookieMonster

UML för arv och interface



Interface

Skapa och hantera Monster

```
Monster aake = new CookieMonster();
Monster loke = new ScaryMonster();
Monster aasa = new ScaryMonster();
```

```
void feedMonsters(Monster[] monsterfamily) {
        :
    for (Monster m : monsterfamily) {
        m.eat();
        if (Math.random() > 0.7)
            m.scream();
    }
   :
}
```

`m.eat()` och `m.scream()` gör olika saker för `CookieMonster` respektive `ScaryMonster`

Alla som implementerar Monster har typen Monster

Alla som implementerar Monster passar där typen Monster krävs

Objekt av CookieMonster har typerna CookieMonster och Monster

Objekt av ScaryMonster har typerna ScaryMonster och Monster

Interface

Interface kan kombineras med

- Arv från annan klass
- Nya metoder
- Nya attribut

En klass kan ärva från högst en annan klass men implementera flera interface

```
class MySubclass extends BigSuperclass
                         implements I1, I2, I3 {

    // concrete implementations of all methods
    // in I1, I2, I3 required

    // constructor, other methods and attributes
    // allowed
}
```

Objekt av `MySubclass` är typmässigt `I1` och `I2` och `I3` och ...

Javas lyssnarinterface

- Interaktion i grafiska fönster genererar *händelser (events)*
- Interface används för att definiera *lyssnarklasser*
- När händelse detekterats anropas metod i *lyssnarobjekt*.
- Lyssnarobjektet implementerar lämpligt *lyssnarinterface*.
- Lyssnarobjekt kopplas till komponenten där händelse väntas.
- En händelse generar anrop av metod i lyssnarobjektet.

Att göra

- Skapa grafiskt objekt att lyssna på, t.ex. en knapp *b*
- Implementera lyssnarinterface $\Rightarrow$ lyssnarklass $\Rightarrow$ *metod*
- Koppla objekt av lyssnarklassen till *b*

Ett tryck på knappen medför att *metod* anropas

Javas lyssnarinterface

Flera lyssnarinterface finns

ActionListener lyssnar bl.a. på knapptryckningar,

finns paketet `java.awt.event`

```
public interface ActionListener {
  public void actionPerformed (ActionEvent e);
}
```

Implementeras:

```
class ..... implements ActionListener{
  ...
  public void actionPerformed (ActionEvent e) {
     // do something
  }
}
```

Vilken klass ska implementera? Flera möjligheter finns.

Koppla lyssnare

Grafiska objekt som kan detektera händelser har metod för att koppla lyssnare. Om b är en knapp:

b.addActionListener (...);

Som parameter ges ett objekt som implementerar lyssnarinterfacet, här

ActionListener

Java-systemet detekterar händelse och anropar interfacets metod, här

actionPerformed()

Metoden actionPerformed()är definierad av programmeraren

Fönstret med knapp men utan händelse

```
import java.awt.*;
class AwtDemo2 extends Frame {
  Button b = new Button("PLEASE PRESS");

  AwtDemo2 ( ) {
    setSize (300 ,300);
    setVisible( true ) ;
    setBackground ( Color.cyan ); add(b);
  }

  public static void main (String[] u) {
    AwtDemo2 window = new AwtDemo2();
  }
}
```

Ge knapptryckningen betydelse

- Gör en klass till lyssnarklass implements Actionlistener
  implements ActionListener
- Definiera interface-metoden public void actionPerformed (...)
  här skrivs vad som ska ske vid knapptryckning
- Koppla lyssnarobjekt till knappen b.addActionListener(...)

Vid knapptryckning kommer Javasystemet att anropa
`actionPerfomed()`

Vi väljer här att göra fönsterklassen till lyssnarklass

Knapptryckningen byter bakgrundsfärg till blå så endast första trycket har effekt.

```
import java.awt.*;
import java.awt.event.*;
class AwtDemo3 extends Frame implements ActionListener {
  Button b = new Button("PLEASE PRESS");
  AwtDemo3 ( ) {
    setSize (300 ,300);
    setVisible (true );
    setBackground ( Color.cyan );
    add(b);
    b.addActionListener(this);
  }

  public void actionPerformed(ActionEvent e) {
    b.setBackground(Color.blue);
  }

  public static void main (String[] u) {
    AwtDemo3 window = new AwtDemo3();
  }
```

Knapptryckningen ändrar färg vid varje tryck. (import-satser och main-metod utelämnade i exemplet)

```
class AwtDemo4 extends Frame implements ActionListener{
  Button b = new Button("PLEASE PRESS");
  Color [] colors ={Color.blue,Color.red,Color.green,
                    Color.yellow };
  int colorindex = 0;
  AwtDemo4 ( ) {
    setVisible (true );
    setSize (300 ,300);
    setBackground (Color.cyan);
    add(b);
    b.addActionListener(this);
  }
  public void actionPerformed(ActionEvent e) {
    b.setBackground(colors[colorindex++%colors.length]);
  }
  public static void main (String[] u) {
    AwtDemo4 window = new AwtDemo4();
  }
}
```

Abstrakt klass

- Inleds med `abstract class`
- Kombination av vanlig klass och interface
- Kan innehålla vanliga metoder och variabler
- Innehåller oftast minst en abstrakt metod som definieras i subklass
- Objekt av klassen kan *inte* skapas

Exempel: Implementation av Schackpjäser

- De olika pjäserna har mycket gemensamt
- Men förflyttning sker på olika sätt
- Definiera en abstrakt klass med allt gemensamt
- Definiera en konkret subklass för varje pjästyp:
  - Bonde
  - Springare
  - Löpare
  - Torn
  - Dam
  - Kung

Exempel: Skiss av klasser

```
abstract class Schackpjas {
  Color farg; Image bild;
  Brade brade; Ruta ruta;

  Schackpjas (...) {...}   // konstruktor

  abstract boolean dragOK(Ruta r1, Ruta r2);
}
```

```
class Bonde extends Schackpjas {
  boolean dragOK(Ruta r1, Ruta r2) {
    // implementera bondens drag
  }
}
```

Konstruktor m.m. visas ej.

Exempel: Skiss av klasser

```
class Dam extends Schackpjas {
  boolean dragOK(Ruta r1, Ruta r2) {
    // implementera damens drag
  }
}
```

```
class Torn extends Schackpjas {
  boolean dragOK(Ruta r1, Ruta r2) {
    // implementera tornets drag
  }
}
```

```
class Springare extends Schackpjas {
  boolean dragOK(Ruta r1, Ruta r2) {
    // implementera springarens drag
  }
}
```

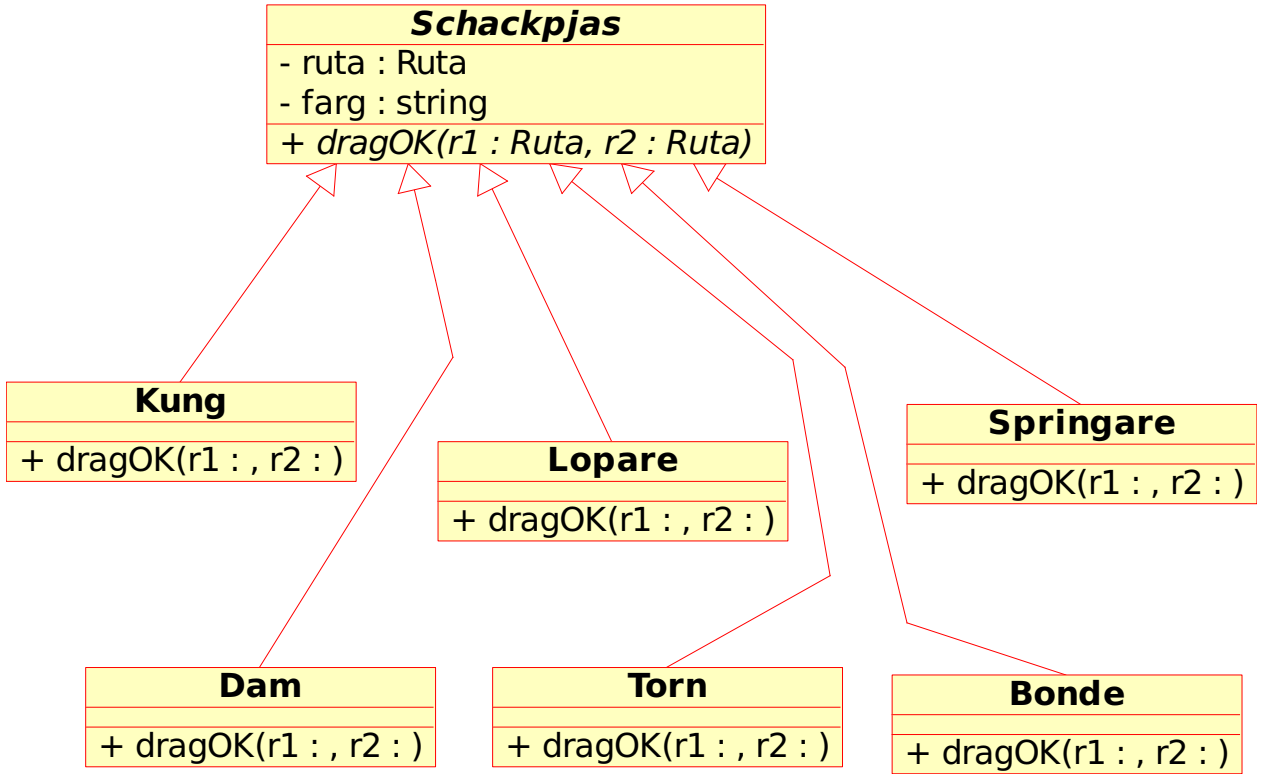
Klasshierarki = Typhierarki

- Klasshierarkin är en trädstruktur
- Referenser av viss typ får referera till objekt av typer som är nedanför i trädet.
- Variabler av typen Schackpjas får referera till objekt av typen
  Bonde, Dam, Kung, Torn ...
- En Schackpjas har många "former" (Bonde, Dam ...):
  POLYMORFISM

Dynamisk bindning

- En Schackpjas-referens "ser" endast Schackpjas-definitioner, t.ex. `dragOK`
- Om en metod, t.ex. `dragOK`, definierats om i en subklass så används den omdefinierade!
- Objektetets typ och inte referensens typ "väljer" metod. Detta är dynamisk bindning
- ```
  Schackpjas pjas = new ...//Dam, Kung, Torn ...
  ...
  pjas.dragOK(..)  //Dynamiskt metodval
                   //beroende av typen
                   //hos pjas-objektet
  ```
  ```

Klass/typ - hierarki för Schackpjäserna

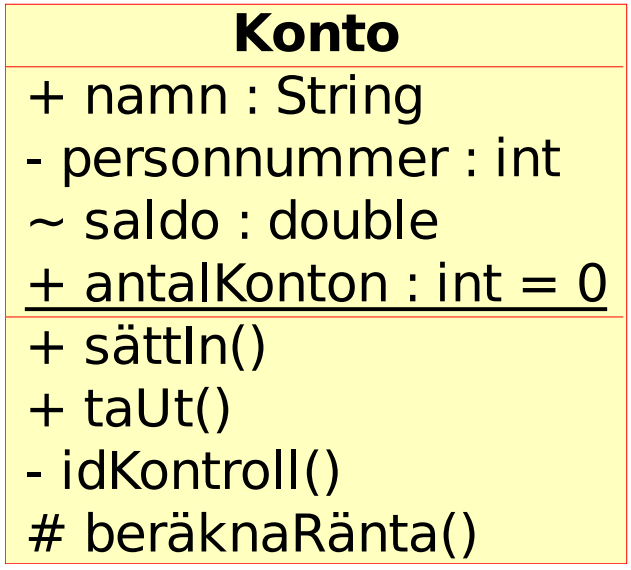


`dragOK()` är en *abstrakt* metod.

`dragOK()` implementeras olika i `Kung`, `Dam`, `Lopare` ....

Alla klasserna har fler metoder

UML – vanlig klass



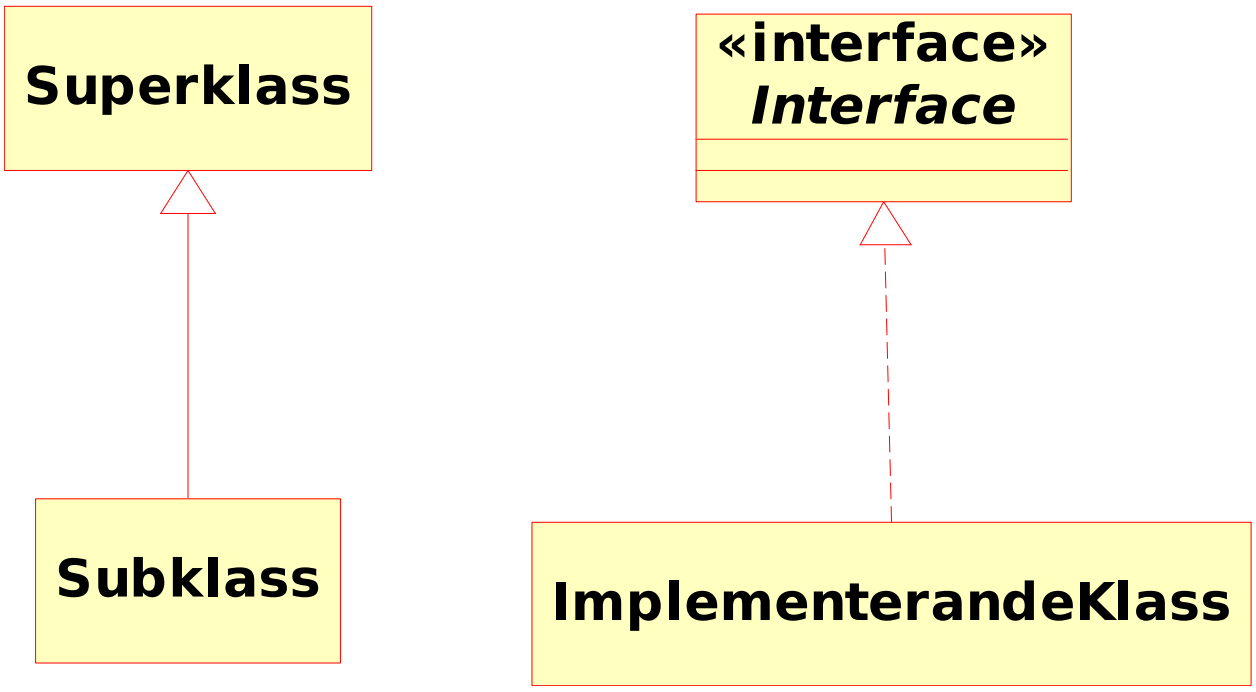
Variabler, ev. med typ

Metoder,
ev. typ,
ev. parameterlista

~ *paketsynlighet*
`#` `protected`
`+` `public`
`-` `private`
`understruket` betyder `static`

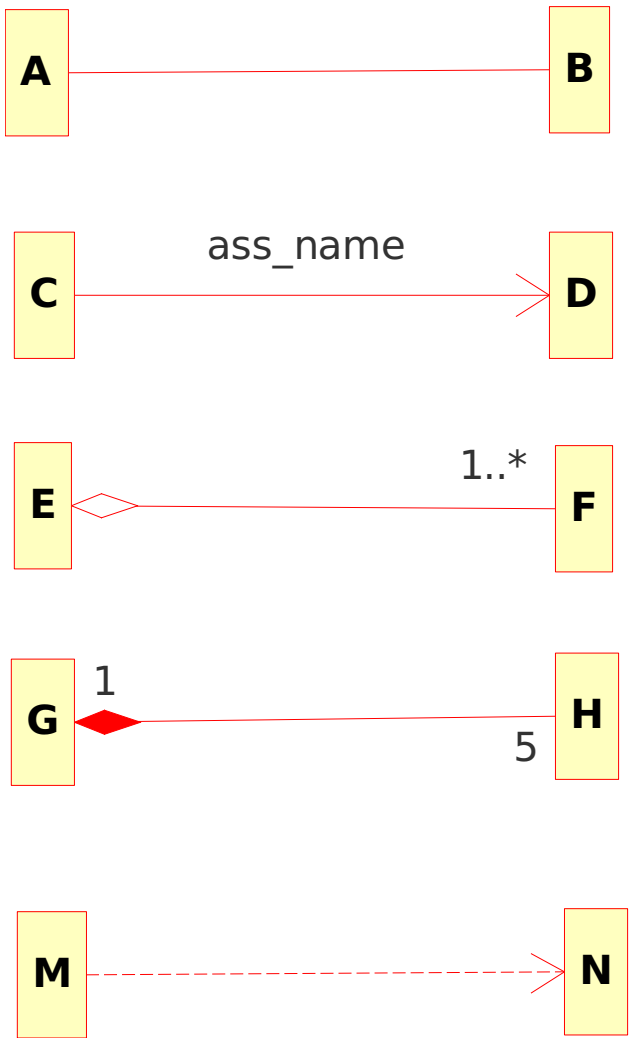
Java-specifik UML

UML – arv och implementation



Relationen ÄR

UML - associationer



Oriktad eller dubbelriktad association

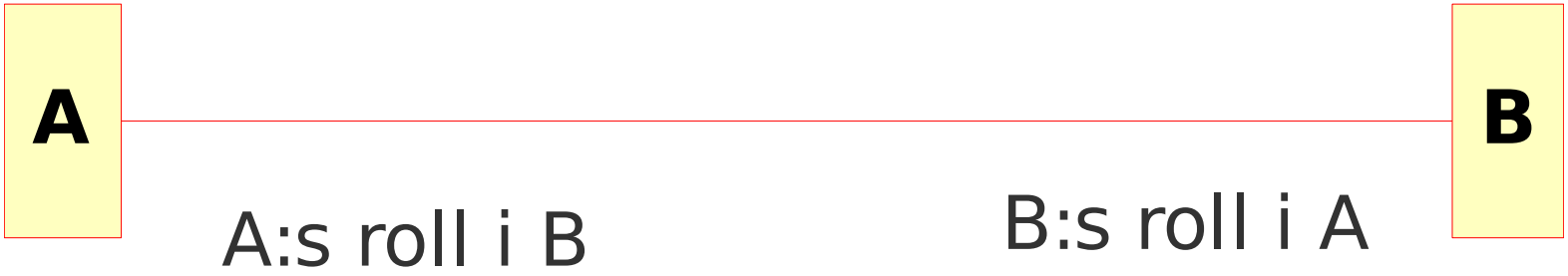
Riktad association,
har, känner till

Aggregat, har, består av,
äger ej

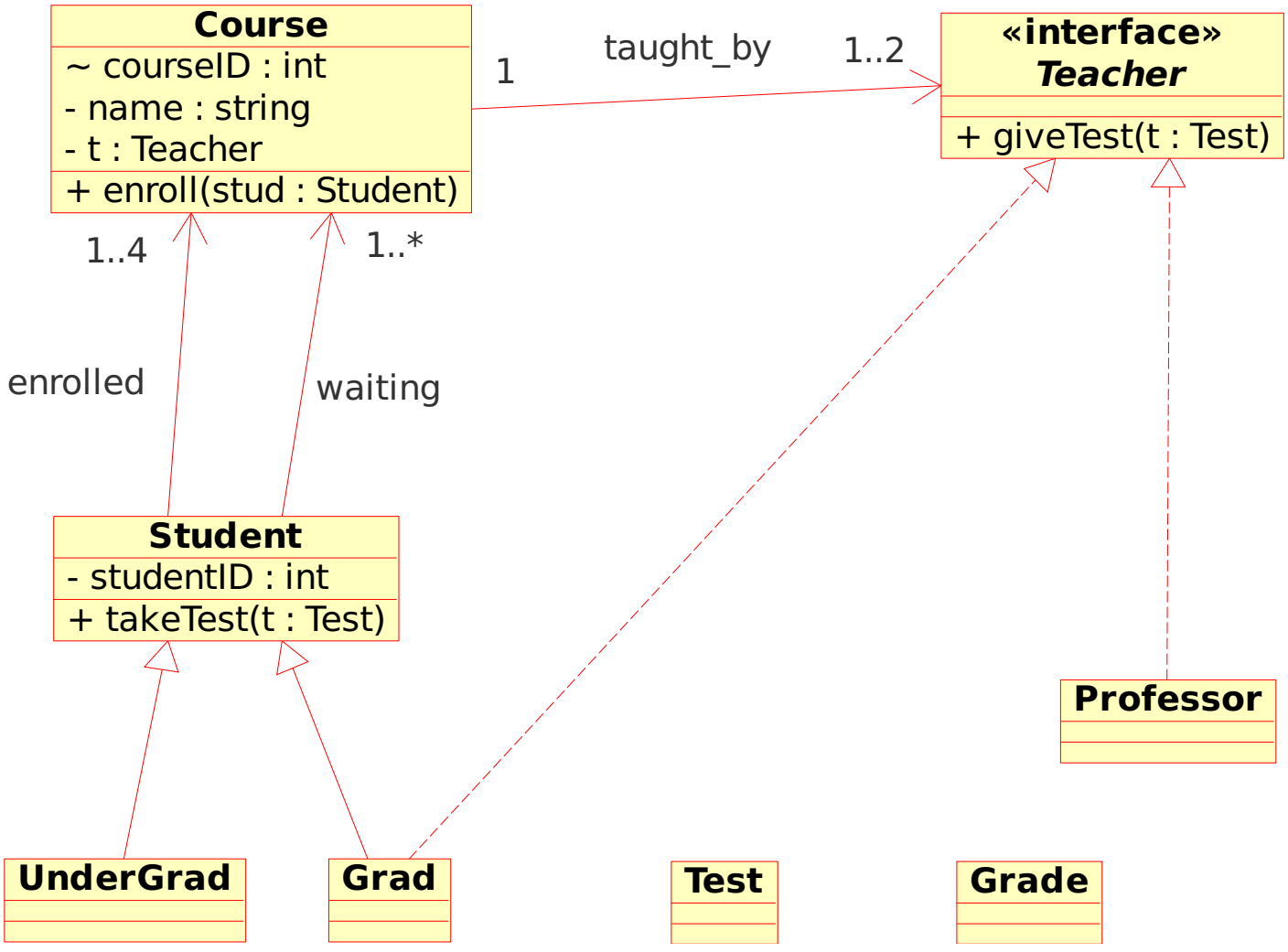
Komposition
har, består av, **äger**

Beroende
beror av,
metodparameter, lokal variabel

UML - dubbelriktad association



UML - exempel


Course
~ courseID : int
- name : string
- t : Teacher
+ enroll(stud : Student)
1
taught_by
1..2
«interface»
Teacher
+ giveTest(t : Test)
1..4
1..*
enrolled
waiting
Student
- studentID : int
+ takeTest(t : Test)
Professor
UnderGrad
Grad
Test
Grade

Klassen Object

- Superklass till *alla* Javaklasser
  - Överst i klasshierarkin
  - Mest generella typen i Java
  - `Object` - referens kan referera till alla objekt
  - Innehåller 11 metoder
- *Alla* klasser har 11 metoder från `Object`
- Flera metoder ärvs tomma eller väldigt enkla, kan/bör definieras om av subklasserna.

Metoderna i Object

String toString() String-repr.

boolean equals(Object o) likhet?

int hashCode()

Object clone() kopia

void finalize() anropas före GC

Class getClass() meta-information

void notify() väck *en* tråd

void notifyAll() väck *alla* trådar

void wait() lägg tråd

void wait(t) i

void wait(t,n) väntelåge

Jämförelse med == eller `equals()` ?

`==`

jämför variablers innehåll

primitiva typer: enkelt och självklart

referenstyper: samma princip

`equals()`

kan jämföra objekts innehåll

om man definierat om den

```
Spelkort s1 = new Spelkort("RUTER",5);
Spelkort s2 = new Spelkort("RUTER",5);
Spelkort s3 = s2;
```

`s1 == s2` ger **false** inte samma objekt men lika

`s2 == s3` ger **true** samma objekt

`s1.equals(s2)` kan ge **true**

om man definierat den rätt