

# Föreläsning 17–20

---

Tobias Wrigstad

*Klasser och arv, konstruktorer, overriding,  
interface, parametrisk polymorfism, m.m*



# En liten Java-parlör

Svenska	Engelska	Svenska	Engelska
Objekt	Object	Metodspecialisering	Overriding
Klass	Class	Överlagring	Overloading
Arv	Inheritance	Överskuggning	Shadowing
Instansvariabel / fält	Instance variable / field	Klasshierarki	Class hierarchy
Metod	Method	Aggregering	Aggregation
Superklass / basklass	Super class / base class	Typomvandling	Type cast
Subklass / härledd klass	Sub class / derived class	Polymorfism	Polymorphism
Abstrakt klass	Abstract class	Barnklass	Child / sub / derived class
Superanrop	Super call	Instantieras	Instantiate

# Hur tolkar man javakompilatorns felmeddelanden

```
<Filnamn.Java>:<Radnummer>: error: <Beskrivning av felet>  
    <information om var det uppstår>  
    <övrig hjälpinformation om tillämpligt>
```

```
CommonCompilerErrors.java:66: error: cannot find symbol  
    LinkedList myList;  
           ^  
        symbol:   class LinkedList  
        location: class ErrorThree
```



# Förstå kompilatorns språk

Kompilatorn säger	Betyder i regel
cannot find symbol	Felstavat namn, eller namnet är inte synligt ännu, t.ex. inte importerat in, alt finns inte
method X cannot be applied	Argumentlistans typer fel (för få argument, fel ordning, fel argument?)
incompatible types	Typen på högersidan är inte kompatibel med den till vänster Är de subtyper?
X cannot be converted to String	Glömt att anropa <code>toString()</code> ?



## Ett socialt nätverk

Det sociala nätverket FooBar fungerar ungefär som Facebook. Varje användare har en profil, och varje profil är knuten till noll eller flera vänner. Det finns två sorters vänner, nära vänner och övriga. En användare kan posta statusuppdateringar som kan nämna andra profiler. En statusuppdatering kan gillas eller ogillas av användare (inklusive postaren själv), samt kommenteras på. Varje kommentar kan också gillas eller ogillas.

Till varje profil finns knuten en händelselogg som innehåller statusuppdateringar i omvänt kronologisk ordning, dvs. senast först. I en händelselogg för användaren A visas alla statusuppdateringar som A har gjort, samt alla relevanta statusuppdateringar för alla vänner. En statusuppdatering hos en nära vän anses alltid relevant, men för övriga gäller att den skall vara "het" eller att A nämns i statusuppdateringen eller någon av dess kommentarer. En statusuppdatering anses vara het om den antalet gillanden + antalet ogillanden + antalet kommentarer överstiger ett visst tröskelvärde T.

Användare kan "knuffa till" varandra. Om A knuffar till B betyder detta att A:s alla statusuppdateringar anses relevanta för B i en vecka och tvärtom. Man kan inte ha fler än 5 "aktiva knuffar" samtidigt.

En användare kan be om att få bli vän med en annan användare, som måste tacka ja först. Vänskap är en symmetrisk relation. En användare kan lista en eller flera vänner som nära vänner. Nära vänskap är inte nödvändigtvis symmetrisk. En särskild kategori av nära vänskap finns dock som är symmetrisk, nämligen släktskap och "i ett förhållande". En användare kan lista en eller flera vänner som släkt/i ett förhållande, och dessa måste tacka ja först. Man kan utan vidare säga upp alla typer av vänskap, inklusive nära vänskap och förhållanden.



Varje användare har en **profil**, och varje profil är knuten till noll eller flera **vänner**. Det finns två sorters vänner, **nära vänner** och **övriga**. En användare kan posta **statusuppdateringar** som kan nämna andra profiler.

En statusuppdatering kan gillas eller ogillas av användare (inklusive postaren själv), samt kommenteras på. Varje **kommentar** kan också gillas eller ogillas.

Till varje profil finns knuten en **händelselogg** som innehåller statusuppdateringar i omvänt kronologisk ordning, dvs. senast först. I en händelselogg för användaren A visas alla statusuppdateringar som A har gjort, samt alla relevanta statusuppdateringar för alla vänner. En statusuppdatering hos en nära vän anses alltid relevant, men för övriga gäller att den skall vara "het" eller att A nämns i statusuppdateringen eller någon av dess kommentarer. En statusuppdatering anses vara het om den antalet gillanden + antalet ogillanden + antalet kommentarer överstiger ett visst tröskelvärde T.



# Objekten

---

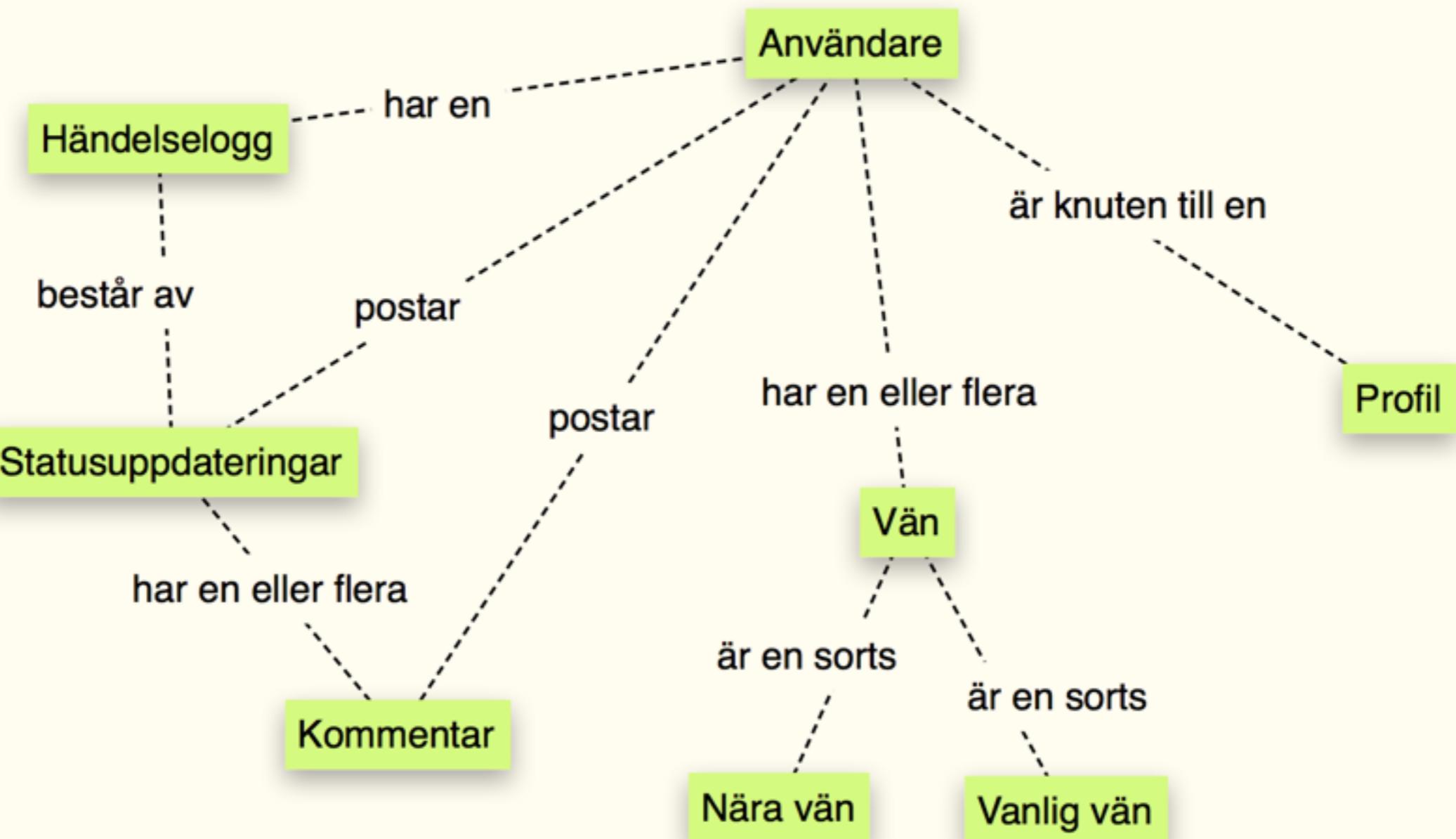
- Användare
- Profil
- Vänner

Det finns nära vänner

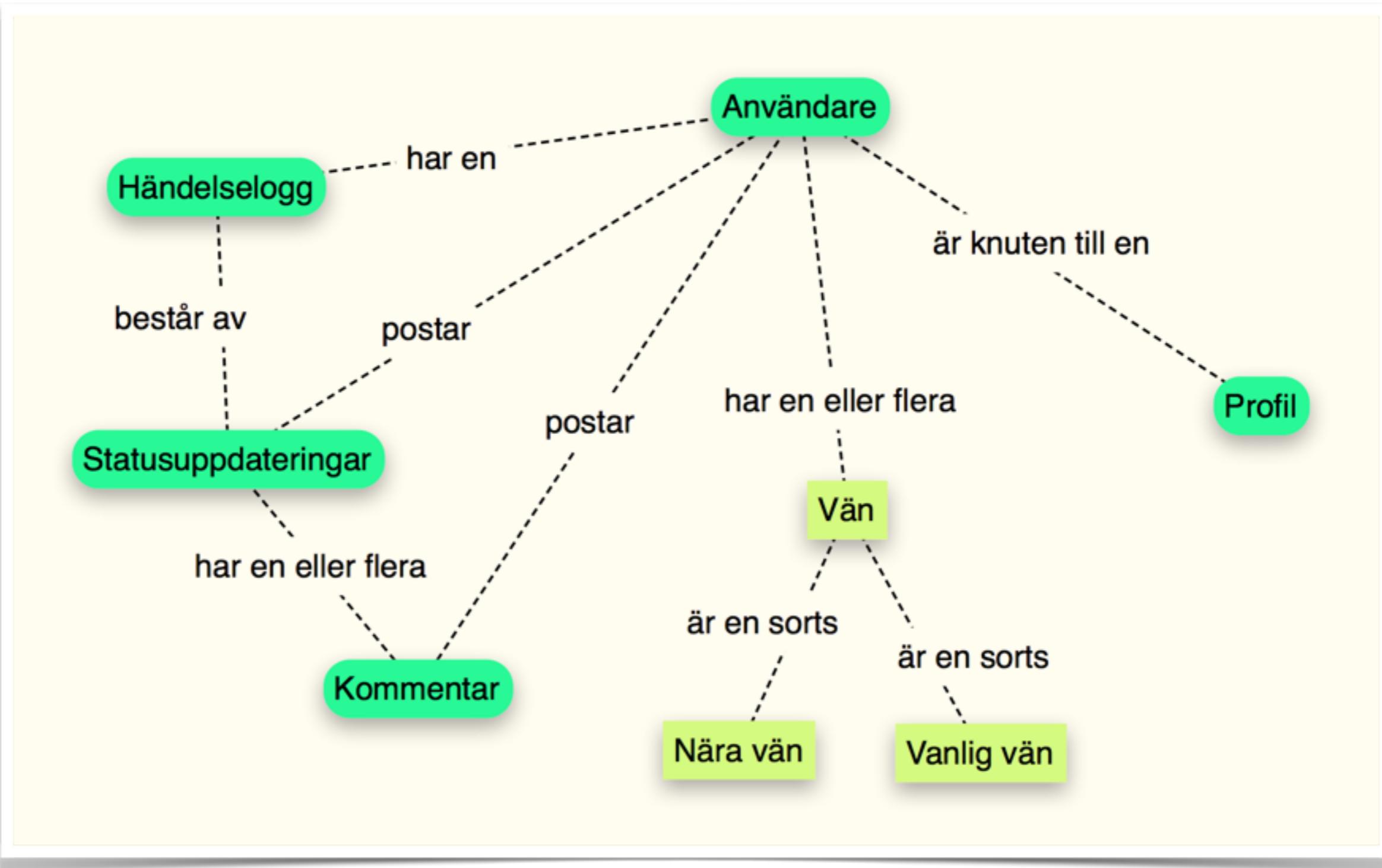
... och vanlig vänskap

- Statusuppdateringar
- Kommentarer
- Händelseslogg

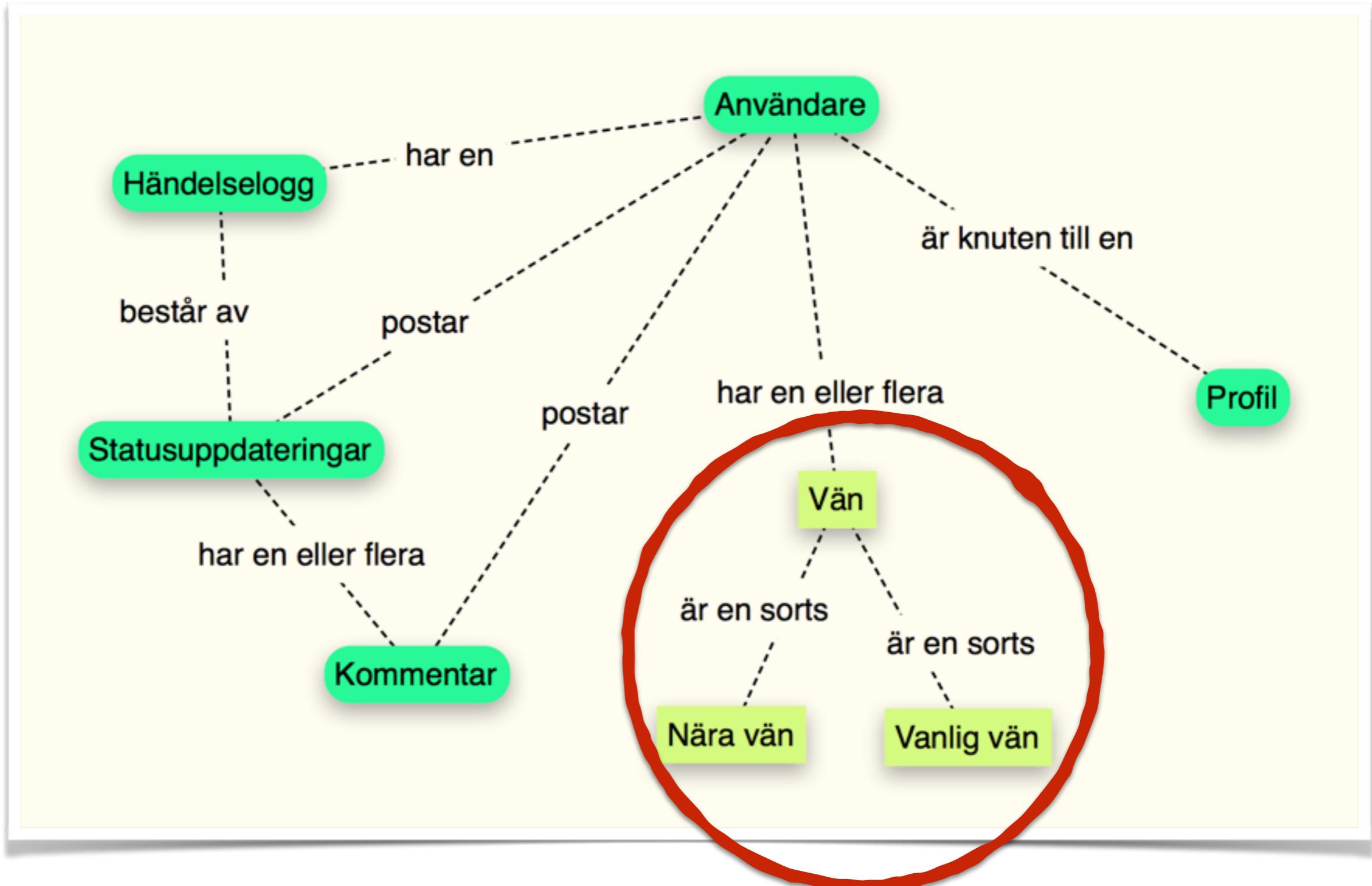
# Objektens relationer



# Klasser



# Inte klasser!



# Varför är ”vän” inte en klass?

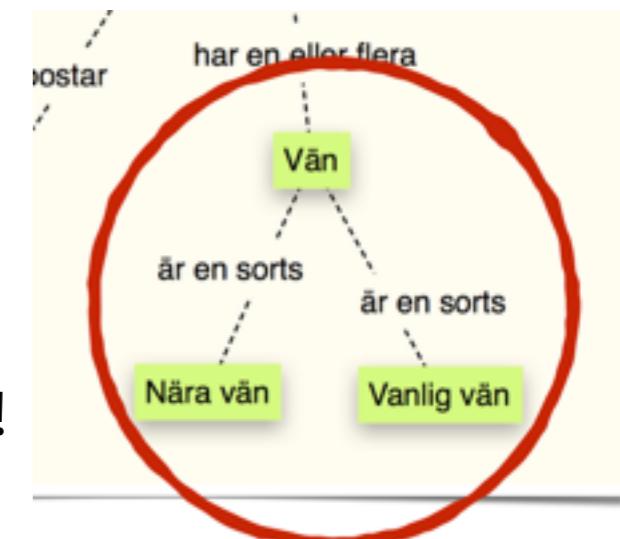
- Vänskap är en **relation** mellan två objekt (och en *vän* är en **roll**)

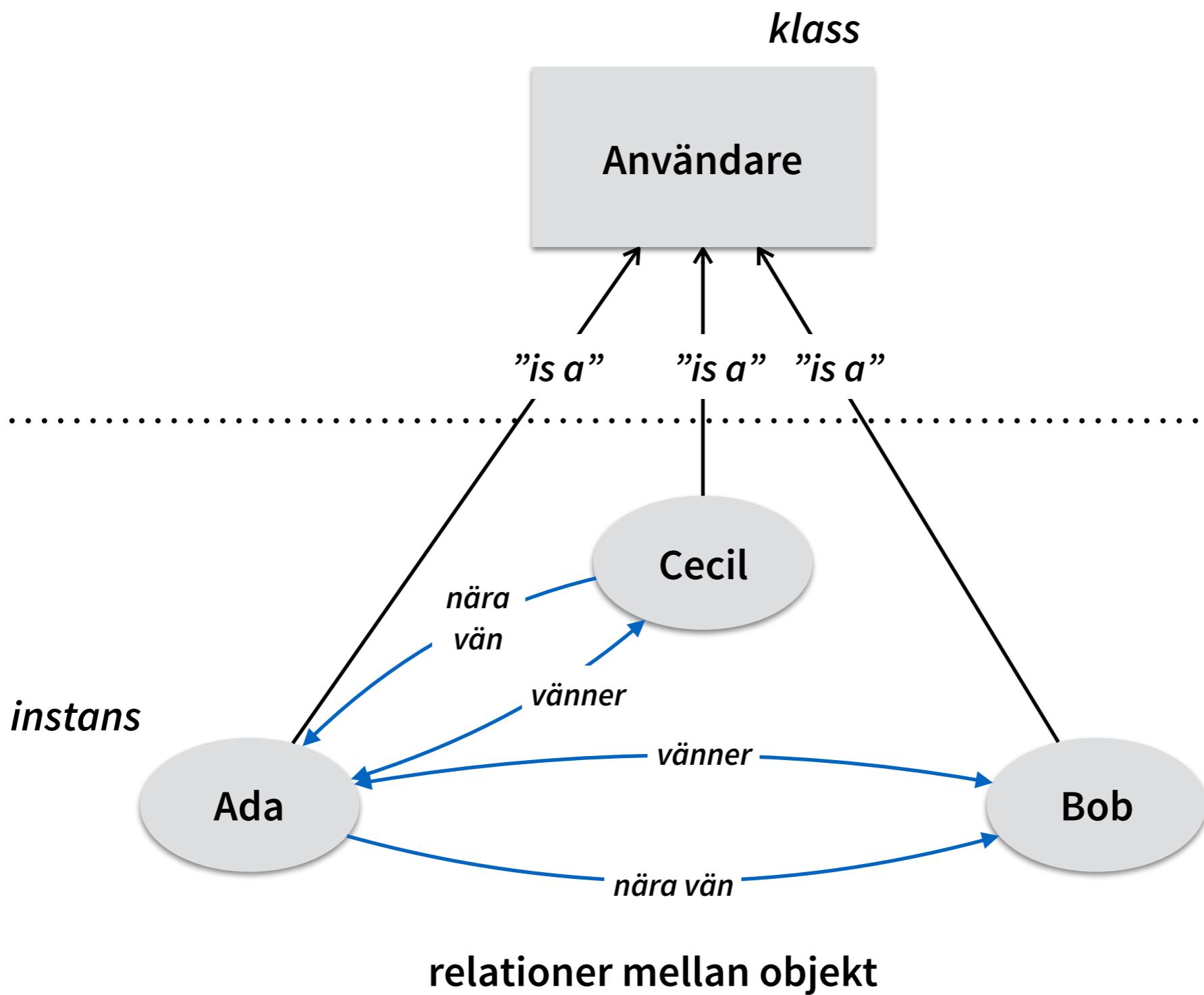
Att vara en vän är inte något som definierar användarkonceptet

- Det räcker med två användare för att modellera vänskap

```
class User {  
    User[] friends;  
    User[] closeFriends;  
  
    void addFriend(final User u) {  
        this.friends[...] = u;  
        u.friends[...] = this;  
    }  
}
```

- Ett **VänskapsBand** skulle kunna vara en klass, men en vän är det inte!





```
class User {  
    User[] friends;  
    User[] closeFriends;  
  
    void addFriend(final User u) {  
        this.friends[...] = u;  
        u.friends[...] = this;  
    }  
}
```

Relationen modellerad som pekare  
mellan objekten

Relationen modellerad som  
instanser av specifika klasser

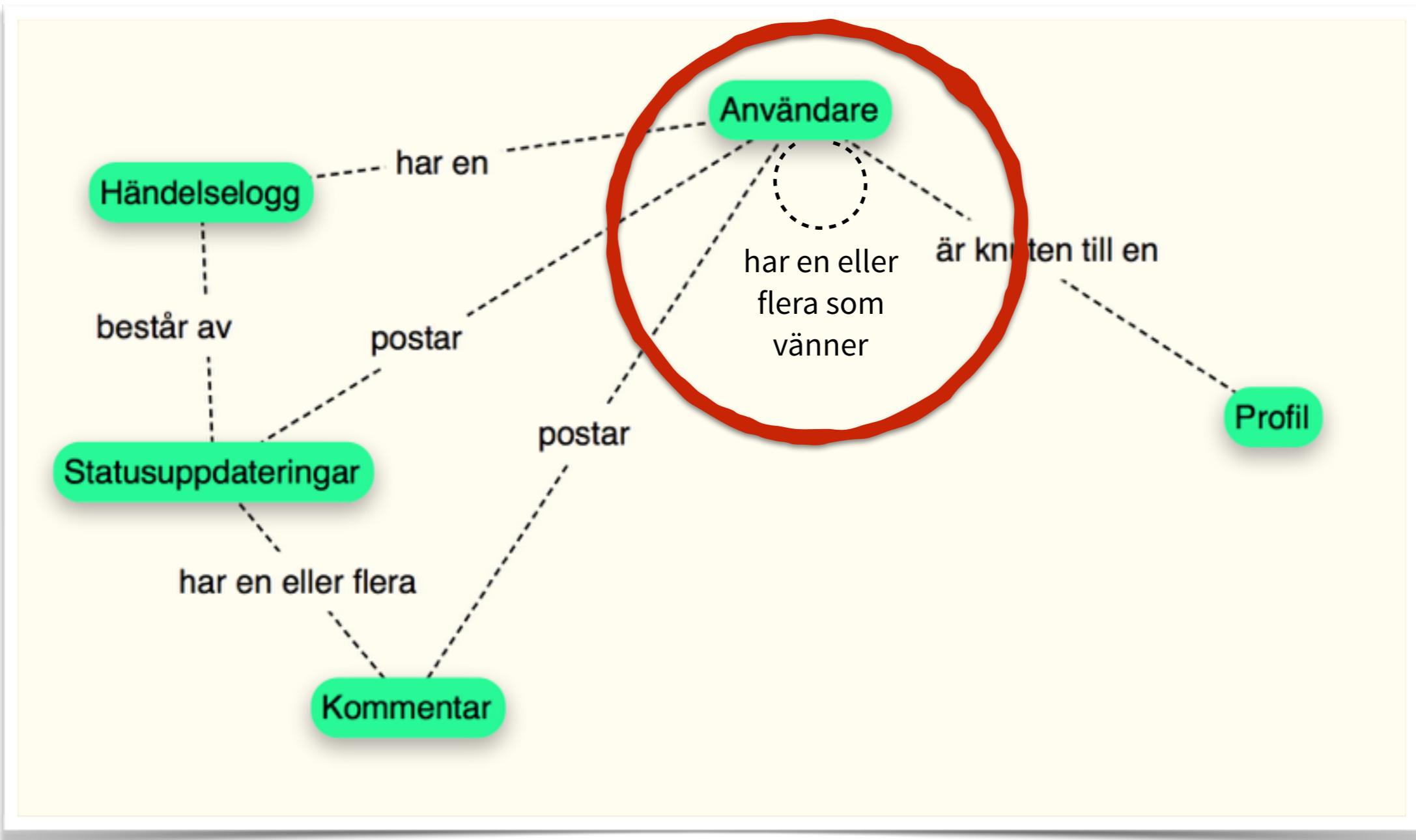


```
class Relation { ... }  
class Friendship extends Relation { ... }  
  
class User {  
    Relation[] connections;  
  
    void addFriend(final User u) {  
        final Friendship f = new Friendship(this, u);  
        this.connections[...] = f;  
        u.connections[...] = f;  
    }  
}
```

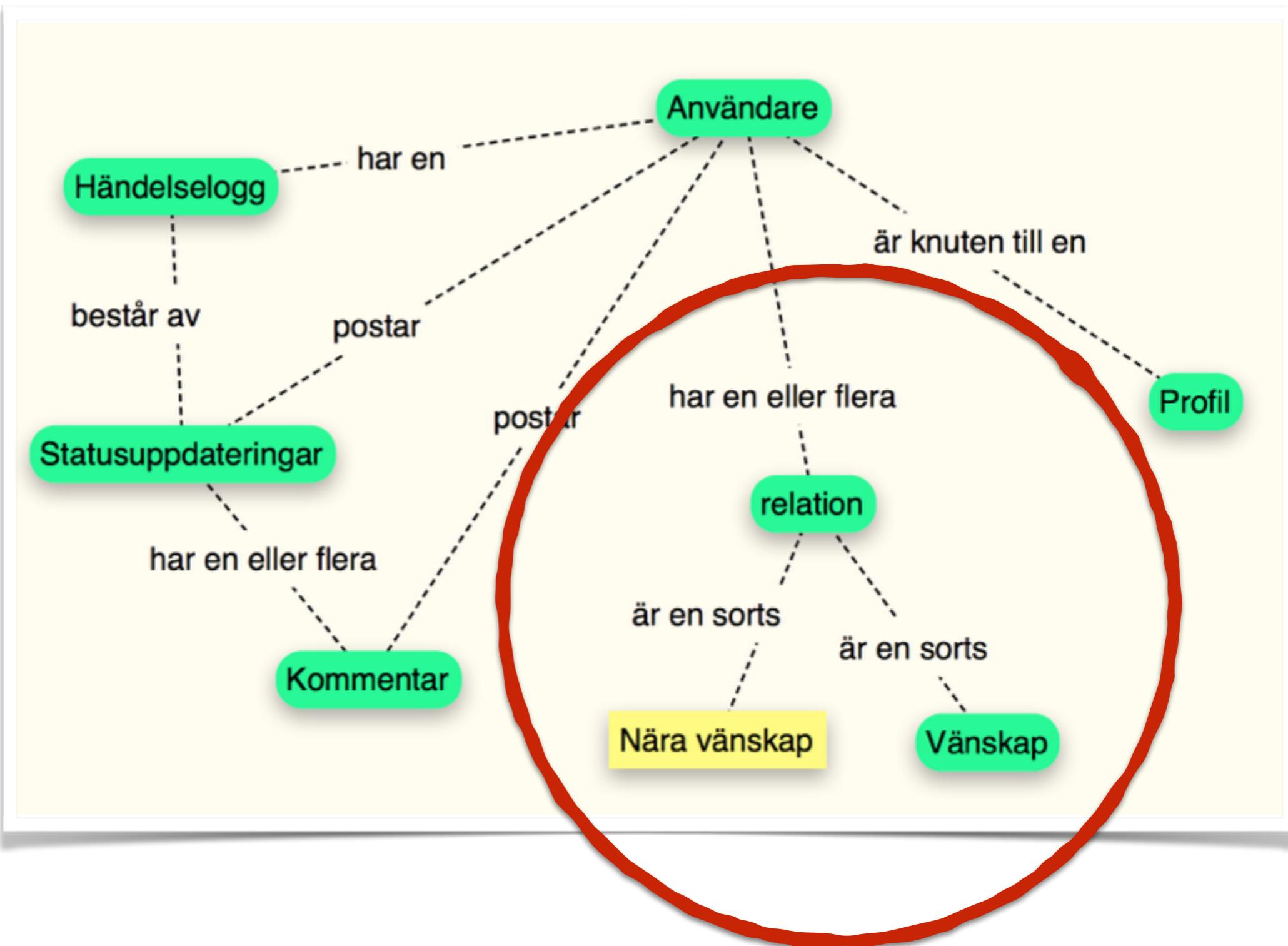


# Vänskap som en reflexiv relation mellan Användare (symmetrisk)

(användare)



# Relation förtinligrat som ett objekt (mer kraftfullt)



# Klasser från objekt

---

- De flesta objekt vi såg är exempel på koncept i domänen för systemet och bör därför ha motsvarande klasser

Programmet är en modell av verkligheten!

Vissa egenskaper hos objekt gav inte upphov till nya klasser

Ibland kan det vara vettigt att modellera relationer som objekt / klasser

- Arv: relationer mellan klasser

Olika typer av relationer (alla linjer med "är en sorts")

Ibland kan det vara vettigt att generalisera och skapa en gemensam superklass för klasser som är väldigt lika

# Arv

---

## Mekanism för återanvändning av kod/design/idéer

En klass kan bygga vidare på en annan klass

- lägga till nya fält, metoder
- omdefiniera metoder

Amerikanska skolan

## En *modulariseringssstrategi*

vs.

En klass kan vara en *specialisering* av en annan klass

- *Student* är specialisering av *Människa*,
- *Människa* är en specialisering av *Däggdjur*,
- *Däggdjur* är en specialisering av *Varelse*

Skandinaviska skolan

## Mycket hype, inte så användbart som ”de” säger

Där arv fungerar, är det dock mycket bra!

## Our Thesis:

- The Essence of Inheritance is that it lets us go from the concrete to the abstract
- It does this using *ex post facto* parameterization: taking a constant and turning it into a parameter



# This just in! (2016-10-31)

## The Essence of Inheritance

Andrew P. Black<sup>1</sup>, Kim B. Bruce<sup>2</sup>, and R. James Noble<sup>3</sup>

<sup>1</sup> Portland State University, Oregon, USA,  
black@cs.pdx.edu,

<sup>2</sup> Pomona College, Claremont, California, USA,  
kim@cs.pomona.edu

<sup>3</sup> Victoria University of Wellington, New Zealand kx@ecs.vuw.ac.nz

**Abstract.** Programming languages serve a dual purpose: to communicate programs to computers, and to communicate programs to humans. Indeed, it is this dual purpose that makes programming language design a constrained and challenging problem. Inheritance is an essential aspect of that second purpose: it is a tool to improve communication. Humans understand new concepts most readily by first looking at a number of concrete examples, and later abstracting over those examples. The essence of inheritance is that it mirrors this process: it provides a formal mechanism for moving from the concrete to the abstract.

**Keywords:** inheritance, object-oriented programming, programming languages abstraction, program understanding

### 1 Introduction

Shall I be abstract or concrete?

An abstract program is more general, and thus has greater potential to be reused. However, a concrete program will usually solve the specific problem at hand more simply.

One factor that should influence my choice is the ease with which a program can be understood. Concrete programs ease understanding by making manifest the action of their subcomponents. But, sometimes a seemingly small change may require a concrete program to be extensively restructured, when judicious use of abstraction would have allowed the same change to be made simply by providing a different argument.

Or, I could use inheritance.

The essence of inheritance is that it lets us avoid the unsatisfying choice between abstract and concrete. Inheritance lets us start by writing a concrete program, and then later on abstracting over a concrete element. This abstraction step is *not* performed by editing the concrete program to introduce a new parameter. That is what would be necessary without inheritance. To the contrary: inheritance allows us to treat the concrete element *as if it were a parameter*, without actually changing the code. We call this *ex post facto* parameterization; we will illustrate the process with examples in Sections 2 and 3.

Arv handlar om att inkrementell beskrivning

Inte om återanvändning

Inte om subtypning

Arv kräver refactoring

Gå från konkret till abstrakt





OBJECT ORIENTATION: PAST, PRESENT, FUTURE

KN

The SIMULA I language report from 1965 opens with these sentences:

"The two main objects of the SIMULA language are:

To provide a language for a precise and standardised description of a wide class of phenomena, belonging to what we may call "discrete event systems".

To provide a programming language for an easy generation of simulation programs for "discrete event systems"."

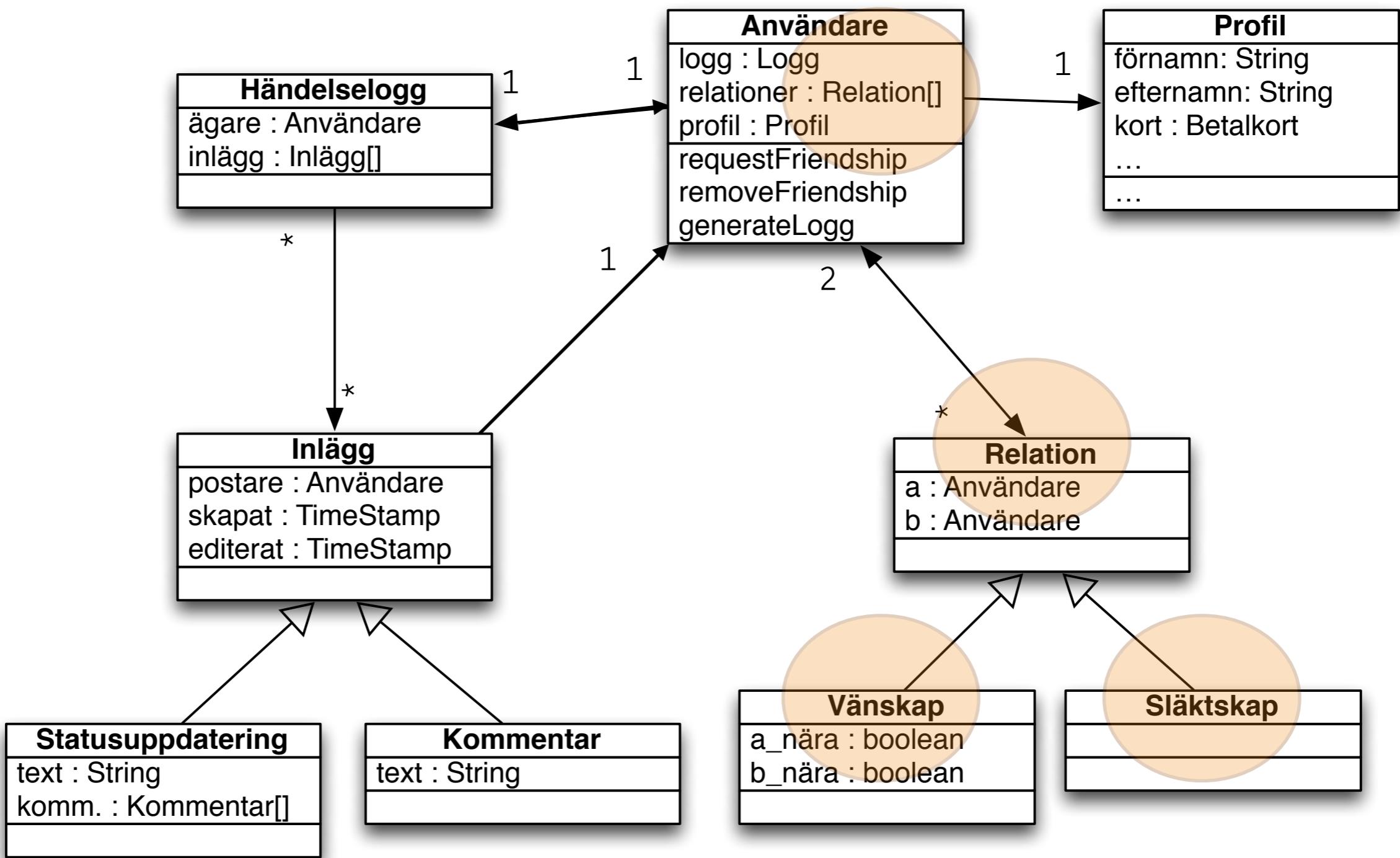
KN

Madsen 2016  
KRISTEN NYGAARD  
© Kristen Nygaard, 2006

3

3

# Ett initialt klassdiagram [UML]



# Klassdiagrammet uttryckt i kod (partiellt)

**class** Händelselogg

**class** Användare

**class** Profil

**class** Inlägg

**class** Statusuppdatering **extends** Inlägg

**class** Kommentar **extends** Inlägg

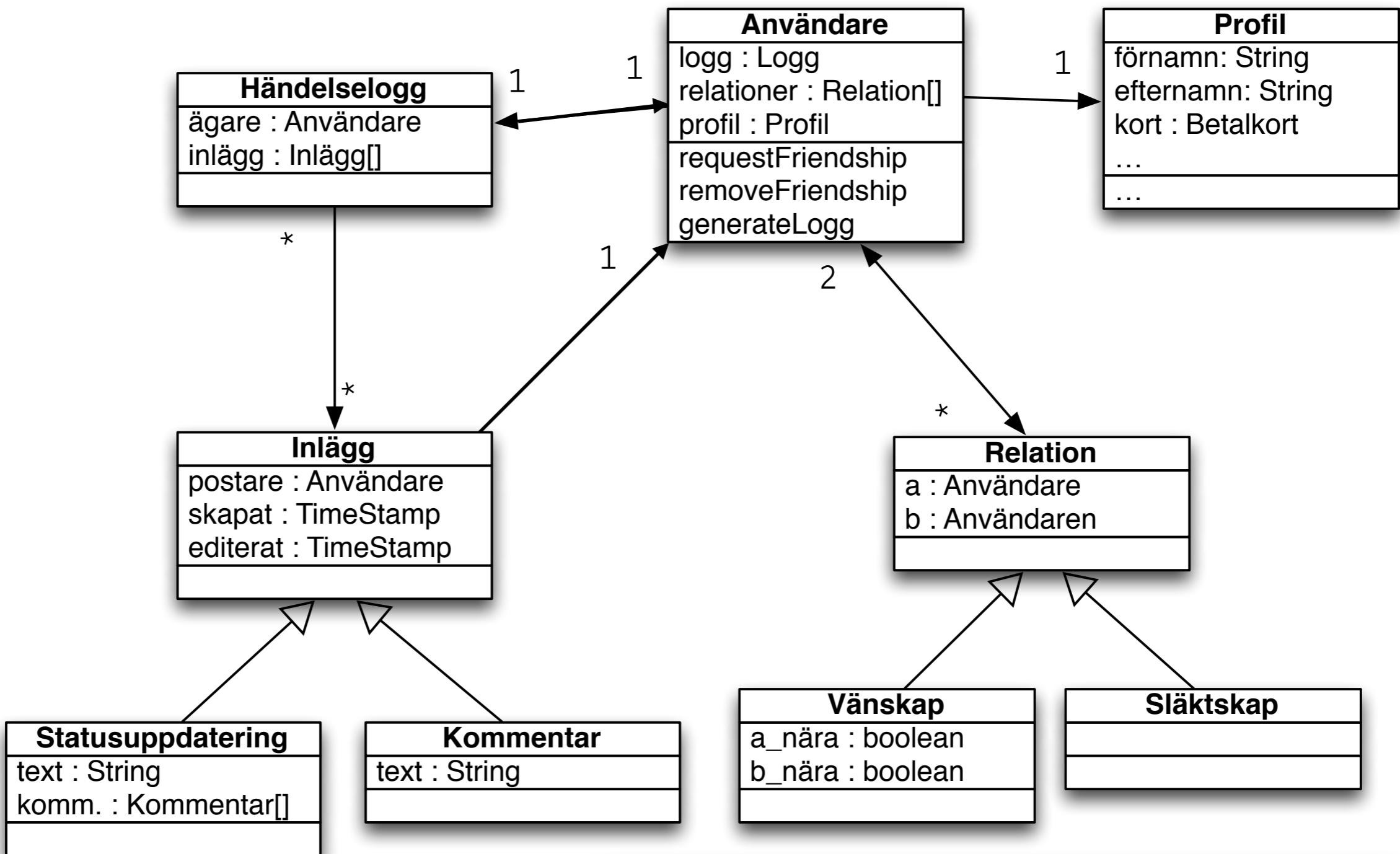
**class** Relation

**class** Vänskap **extends** Relation

**class** Slätskap **extends** Relation



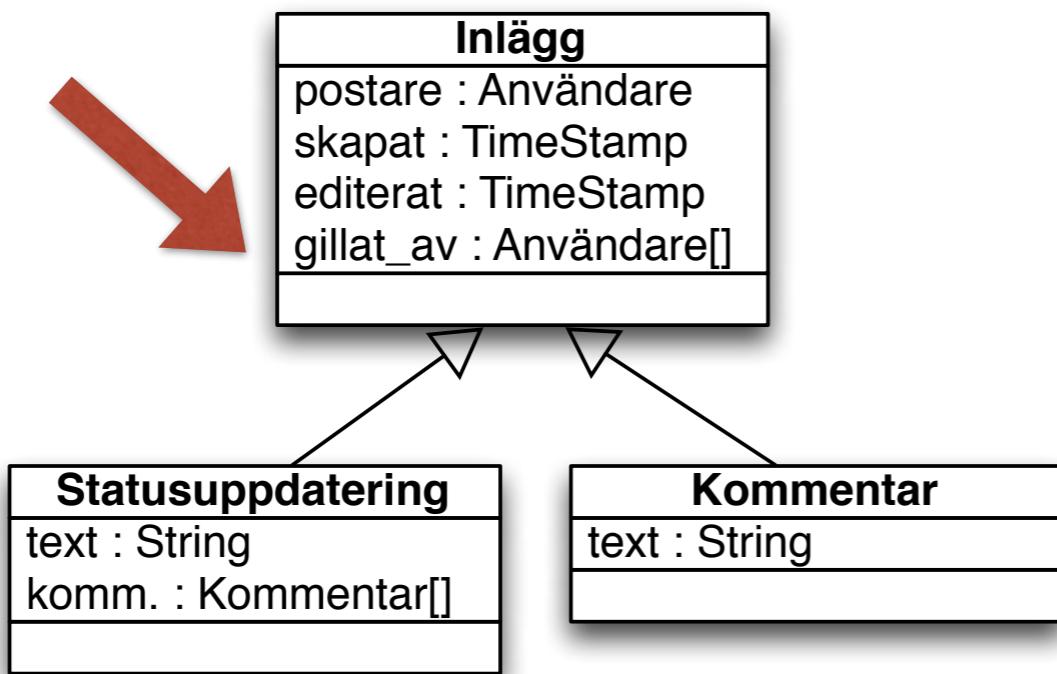
# Ett initialt klassdiagram [UML]



Var/hur lägger vi till stöd för att gilla inlägg?



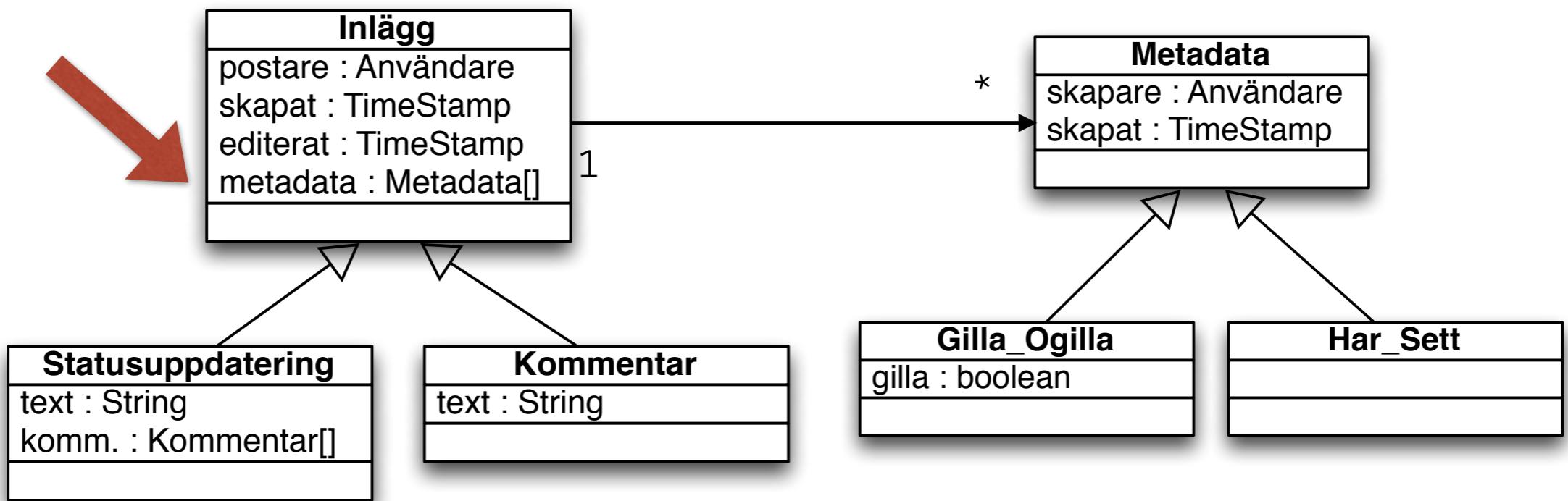
## Försök #1: "gillat\_av"



*Om en användare gillar ett inlägg, lägg till vederbörande i en lista "gillat\_av"*



## Försök #2: "metadata"



Associera varje inlägg med godtyckligt metadata



# Skillnader mellan försök #1 och #2

---

## Försök #1

Logiken för metadata ligger i Inlägg-klassen

Måste skapa en ogillat\_av, sett\_av, etc. för varje ny egenskap man vill lägga till

Enkelt och direkt

## Försök #2

Logiken för metadata är fakturerat ut ur inlägg och kan ändras "fritt"

Enkelt att lägga till en ny typ av metadata

Mer komplicerat

Måste följa någon form av mönster för att "dra nytta av ny metadata"

# Separera metadata från inlägg

```
class Gilla_Ogilla extends Metadata {  
    String decorate(String s) {  
        // ... s är inläggstexten "hittills"  
    }  
}
```

```
class Inlägg {  
    Metadata[] metadata;  
    String text;  
    String render() {  
        String html = text;  
        for (Metadata m : metadata) {  
            html = m.decorate(html);  
        }  
        return html;  
    }  
}
```



Faktisk metadata okänd



Dynamisk bindning



# Arv (eng. inheritance)

---

- Låt A och B vara klasser; A har variablerna X och Y, samt metoderna M och N
- Om B ärver av A får B också X och Y och M och N
- Arv fångar generalisering—specialisering (superklass—subklass)

Undvika upprepning av kod

Fångar relation mellan klasser i koden

I statiskt typade språk som Java (C++, C#, m. fl.): viktig för polymorfism

- Metadata superklass, Gillia\_Ogilla och Sedd är subklasser till Metadata  
I en array av Metadata kan man blanda Gillia\_Ogilla och Sedd
- Metadata borde vara en **abstrakt** klass som inte kan instantieras

# Partiell klasshierarki för relationer mellan två personer

```
abstract class Relation {  
    Person a;  
    Person b;  
}  
  
class Friendship extends Relation {  
    boolean a_close;  
    boolean b_close;  
}  
  
class Family extends Relation {}
```



# Arv i Javas API

---

- `java.lang.Object`
  - `java.util.AbstractCollection<E>`
    - `java.util.ArrayList<E>`
    - `java.util.AbstractSequentialList<E>`
      - `java.util.LinkedList<E>`
- `java.lang.Object`
  - `java.awt.Component`
    - `java.awt.Container`
      - `java.awt.Window`
        - `javax.swing.JWindow`
- I stabila bibliotek passar arv ofta bra, både ur amerikansk och skandinavisk synpunkt
- Mängden arv i ”vanliga program” betydligt mindre

}

*Djup 5*

}

*Djup 5*

# How do Java Programmers Use Inheritance?

[Tempero, Yang and Noble, ECOOP 2008]

The paper makes the following contributions:

- A fine grained, structured suite of inheritance metrics for Java-like languages.
- A corpus analysis applying these metrics to 93 Java applications containing over 100,000 user-defined types.

Based on the corpus analysis, we demonstrate some important features of the accepted practice regarding inheritance in Java programs:

- most classes in Java programs are defined using inheritance from other “user-defined” types.
- classes and interfaces are used in stereotypically different ways, with approximately one interface being declared for every ten classes.
- client metrics have truncated curve distributions while supplier metrics have power law-like distributions.
- most types (classes and interfaces) are relatively shallow in the inheritance hierarchy.
- almost all types have fewer than two types inheriting from them: however for some very popular types, the bigger the programs, the more types will inherit from them.

# How do Java Programmers Use Inheritance?

[Tempero, Yang and Noble, ECOOP 2008]

The paper makes the following contributions:

- A fine grained, structured suite of inheritance metrics for Java-like languages.
- A corpus analysis applying these metrics to 93 Java applications containing over 100,000 user-defined types.

Based on the corpus analysis, we demonstrate some important features of the accepted practice regarding inheritance in Java programs:

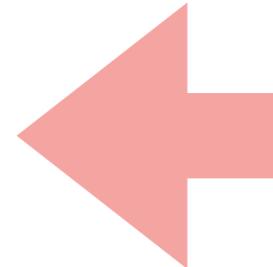
- most classes in Java programs are defined using inheritance from other “user-defined” types.
- classes and interfaces are used in stereotypically different ways, with approximately one interface being declared for every ten classes.
- client metrics have truncated curve distributions while supplier metrics have power law-like distributions.
- most types (classes and interfaces) are relatively shallow in the inheritance hierarchy.
- almost all types have fewer than two types inheriting from them: however for some very popular types, the bigger the programs, the more types will inherit from them.

# Arv och konstruktorer

---

## **En konstruktors syfte är att initiera objekt**

Man skall inte kunna observera ett objekt i ett "inconsistent state"



Ex.: klassen Point2D har två privata variabler x och y

Point2D har en konstruktor som tar ett x-värde och ett y-värde — tvingar alla punkter att ha valida värden på alla axlar (consistent state)

## **En konstruktor för en subklass initierar "sin del" av objektet**

Ex.: klassen Point3D ärver Point2D, och lägger till en z-axel

Point3D har en konstruktor som tar x, y, z

Point3D kan inte skriva till x och y — de är privata!

Point3D måste delegera till Point2D:s konstruktor att initiera x och y

# Arv och konstruktorer (Exempel)

```
class Point2D {  
    private int x;  
    private int y;  
    Point2D(final int x, final int y) {  
        this.x = x;  
        this.y = y;  
    }  
    ...  
}
```

```
class Point3D extends Point2D {  
    private int z;  
    Point3D(final int x, final int y, final int z) {  
        super(x, y); // anropar superklassens konstruktur  
        this.z = z;  
    }  
    ...  
}
```

# Arv och konstruktorer

```
abstract class Relation {
    Person a;
    Person b;
    Relation(final Person a, final Person b) {
        assert(a.equals(b) == false);
        this.a = a;
        this.b = b;
    }
}

class Friendship extends Relation {
    boolean a_close;
    boolean b_close;
    Friendship(final Person a,
               final Person b,
               final boolean a_close,
               final boolean b_close) {
        super(a, b);
        this.a_close = a_close;
        this.b_close = b_close;
    }
}

static Friendship newMutualClose(final Person a, final Person b) {
    return new Friendship(a, b, true, true);
}
```

Friendship.newMutualClose(p1, p2)



# Abstrakta klasser

Vi introducerar en generalisering,  
inte en ritning för att bygga nya objekt

```
abstract class Relation { ... }

Person p1 = ...;
Person p2 = ...;

Relation r = new Relation(p1, p2); // kompilerar ej
```



oo i C



# OO in C

```
struct relation
{
    person_t *a;
    person_t *b;
};
```

```
abstract class Relation {
    Person a;
    Person b;
    Relation(final Person a, final Person b) {
        assert(a.equals(b) == false);
        this.a = a;
        this.b = b;
    }
}
```

```
struct relation *init_relation(struct relation *r, person_t *a, person_t *b)
{
    assert(cmp_person(a, b) != 0);
    r->a = a;
    r->b = b;
    return r;
}
```

```
struct relation *new_relation(person_t *a, person_t *b)
{
    assert(false); // relation is abstract!
}
```



# OO i C

Värde-  
semantik

```
struct relation
{
    person_t *a;
    person_t *b;
};
```

```
struct friendship
{
    struct relation;
    bool a_close;
    bool b_close;
};
```

```
abstract class Relation {
    Person a;
    Person b;
    Relation(final Person a,
              final Person b) {
        assert(a.equals(b) == false);
        this.a = a;
        this.b = b;
    }
}
```

```
class Friendship extends Relation {
    boolean a_close;
    boolean b_close;
    Friendship(final Person a,
               final Person b,
               final boolean a_close,
               final boolean b_close) {
        super(a, b);
        this.a_close = a_close;
        this.b_close = b_close;
    }
}
```



# OO i C

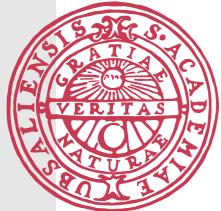
```
struct relation *init_friendship(struct friendship *f,
                                  person_t *a, person_t *b,
                                  bool a_close, bool b_close)
{
    init_relation((struct relation *) f, a, b);
    f->a_close = a_close;
    f->b_close = b_close;
    return f;
}
```

```
class Friendship extends Relation {
    boolean a_close;
    boolean b_close;
    Friendship(final Person a,
               final Person b,
               final boolean a_close,
               final boolean b_close) {
        super(a, b);
        this.a_close = a_close;
        this.b_close = b_close;
    }
}
```

```
struct relation *new_friendship(person_t *a, person_t *b, bool a_close, bool b_close)
{
    return init_friendship(malloc(sizeof(struct friendship)), a, b, a_close, b_close);
}
```

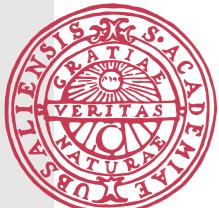


*Det går utmärkt att  
programmera objektorienterat  
i C, man får bara ingenting  
gratis*

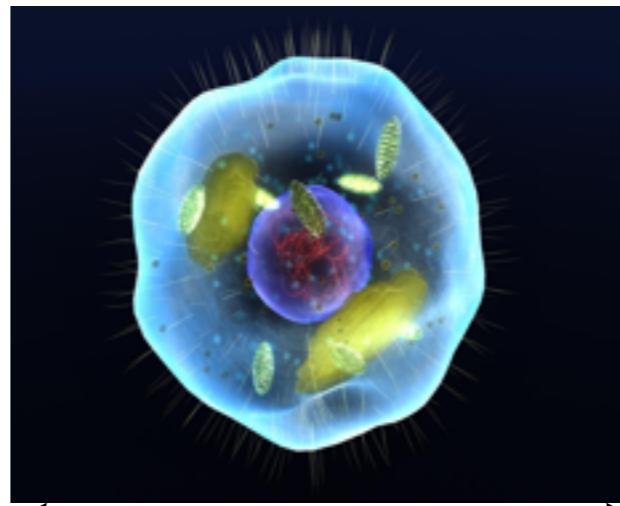


# Arv och specialisering #2

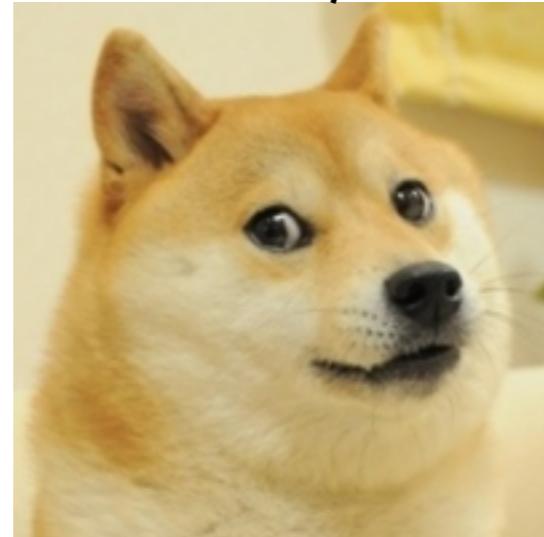
## ”overriding”



Animal



is\_a



Dog

is\_a



Cat



# Overriding (Metodspecialisering)

```
abstract class Animal {  
    void makeSound() { System.out.println("Does not make sense!"); }  
}
```



# Overriding (Metodspecialisering)

```
abstract class Animal {  
    void makeSound() { System.out.println("Does not make sense!"); }  
}  
  
class Cat extends Animal {  
    void makeSound() { System.out.println("Meeow!"); }  
}
```



# Overriding (Metodspecialisering)

```
abstract class Animal {  
    void makeSound() { System.out.println("Does not make sense!"); }  
}  
  
class Cat extends Animal {  
    void makeSound() { System.out.println("Meeow!"); }  
}  
  
class Dog extends Animal {  
    void makeSound() { System.out.println("Wuff!"); }  
}
```



# Overriding (Metodspecialisering)

```
abstract class Animal {  
    void makeSound() { System.out.println("Does not make sense!"); }  
}  
  
class Cat extends Animal {  
    void makeSound() { System.out.println("Meeow!"); }  
}  
  
class Dog extends Animal {  
    void makeSound() { System.out.println("Wuff!"); }  
}  
  
class Driver {  
    public static void main(String[] args) {  
        Animal[] animals = { new Cat(), new Dog(), new Cat() };  
        for (Animal a : animals)  
            a.makeSound();  
    }  
}
```



# Overriding av abstrakta metoder

```
abstract class Animal {  
    abstract void makeSound();  
}  
  
class Cat extends Animal {  
    void makeSound() { System.out.println("Meeow!"); }  
}  
  
class Dog extends Animal {  
    void makeSound() { System.out.println("Wuff!"); }  
}  
  
class Driver {  
    public static void main(String[] args) {  
        Animal[] animals = { new Cat(), new Dog(), new Cat() };  
        for (Animal a : animals)  
            a.makeSound();  
    }  
}
```



# Overriding – exempel 2 (visar ”superanrop”)

```
abstract class Animal {  
    String name = null;  
  
    void setName(final String name) { this.name = name; }  
  
    String getName() { return name; }  
}  
  
class Cat extends Animal { ... }  
  
class Dog extends Animal {  
    final String[] okDogNames = { "Amanda", "Prima" , "Tassie" };  
  
    void setName(final String name) {  
        for (String ok : okDogNames) {  
            if (ok.equals(name)) { super.setName(name); return; }  
        }  
    }  
}
```



Anropa metoden setName i närmaste superklasss



# En bättre hund

```
class Dog extends Animal {  
    final String[] okDogNames = { "Amanda", "Prima" , "Tassie" };  
  
    void makeSound() { System.out.println("Wuff!"); }  
  
    boolean isOkName(final String name) {  
        for (String ok : okDogNames) {  
            if (ok.equals(name)) { return true; }  
        }  
        return false;  
    }  
  
    void setName(final String name) {  
        if (this.isOkName(name)) super.setName(name);  
    }  
}
```



# Utökningsmöjligheter...

```
class ShowDog extends Dog {  
    final String[] okShowDogNames = { "Pet", "Goldie", "Winner" };  
  
    boolean isOkName(final String name) {  
        for (String ok : okShowDogNames) {  
            if (ok.equals(name)) { return true; }  
        }  
        return super.isOkName(name); } } } } } }
```



statisk bindning



# ...eller inte utökningsmöjligheter?

```
class Dog extends Animal {  
    final String[] okDogNames = { "Amanda", "Prima" , "Tassie" };  
  
    void makeSound() { System.out.println("Wuff!"); }  
  
    private boolean isOkName(final String name) {  
        for (String ok : okDogNames) {  
            if (ok.equals(name)) { return true; }  
        }  
        return false;  
    }  
  
    void setName(final String name) {  
        if (isOkName(name)) super.setName(name);  
    }  
}
```

statisk bindning



# Exempel på inkapsling och arv

```
abstract class Animal {  
    String name = null;  
  
    void setName(final String n) { if (isOkName(n)) this.name = n; }  
  
    abstract boolean isOkName(final String name);  
}  
  
class Cat extends Animal {  
    bool isOkName(final String name) { return true; }  
}  
  
class Dog extends Animal {  
    String[] okDogNames = { "Amanda", "Prima" , "Tassie" };  
  
    bool isOkName(final String name) {  
        // cheat!  
        this.name = "Captain Beefheart!";  
    }  
}
```

Tvingar subklasser att  
tillhandahålla en sådan

name är inte private!



# Overriding vs. Overloading

---

- Overriding tillåter en subklass *B* att omdefiniera en metod *m*, ärvt från superklass *A*

**Shadowing:** *m* i *A* blir inte synlig i instanser av *B*

Superanrop tillåter *B*:s metoder att anropa *A*:s *m*

- Overloading tillåter oss att definiera flera metoder med samma namn, men skilda parametertyper (eller antal parametrar)

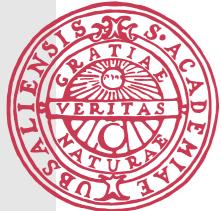
Ex.: `checkPrice(int p)` // *p* är priset i ören  
`checkPrice(float p)` // *p* är priset i kronor

- Overloading är en vanlig felkälla och skall *inte* överanvändas

Svårt att veta att det finns flera metoder med samma namn (följer fel logik)

Kan vara svårt att räkna ut vilken vilken metod som används

# Arv och subtypning



# Rotklassen Object

---

- Alla Java-klasser som inte har en explicit superklass ärver klassen Object

Det betyder att alla arvshierarkier formar ett träd vars rot är Object

- Varje klass i ett Java-program skapar en ny typ

Relation, Dog, Animal, etc.

- I Java medför subclassing subtypning

Dog **extends** Animal betyder att Dog är en subtyp av Animal

- Eftersom klasshierarkier är rotade i Object är alla typer subtyper av Object

Det betyder att Object x = ...; är legit, oavsett vad ... är

# Subtypning

---

- Liskovs substitutionsprincip:

Om  $T$  är en subtyp av  $T'$  så kan vi ersätta alla förekomster av  $T'$ -objekt i ett program med objekt av typen  $T$

Detta är sant, såtillvida att alla metoder i  $T'$  också finns i  $T$

Dock – inga garantier att metoderna faktiskt beter sig på ett lämpligt sätt

- I Java finns två sorters typer:

**objekttyper** skapas av klasser eller interface (vi skall snart se dem!)

**primitiva typer** är ärvda från C (alltså, int, float, etc.)

- Primitiva typer har ingen subtypning

# Typtest och typomvandling

---

- I Java finns metadata om objekt under körning

*expr instanceof Type* ⇒ true/false

Ex.: `connections[42] instanceof Friendship`

- Omvandling från en typ till en annan (som i C)

*(Type) expr* ⇒ kraschar programmet om *expr* inte har typen *Type*

Ex. `(Friendship) connections[42]`

Mönster:

```
if (connections[42] instanceof Friendship) {  
    ... (Friendship) connections[42];  
}
```

# Problem med arv: Fragile Base Class Problem

---

- Superklassen känner inte till sina barnklasser och kan därför inte förutse effekterna av förändring

I bästa fall: barnklassen kompilerar inte

I värsta fall: alla kompilerar, programmets betydelse ändrad

- I många OO-programspråk tenderar arv att ge privilegierad åtkomst till superklassens metoder och fält

Extra starka beroenden från subklass till superklass (som ej känner till dem!)

- I den amerikanska skolan brottas man ofta med oönskad inkludering

# Arv och subtypspolymorfism

---

- Subtypspolymorfism i Java

(Liskov igen)

*Om B ärver A kan ett B-objekt användas varhelst ett A-objekt förväntas*

Hittils: B subtyp A  $\Rightarrow$  B subklass A

Shape x = **new** Square(); // OK om Square ärver av Shape

- Javas statiska typning och avsaknad av multipelt implementationsarv begränsar kraftfullheten hos subtypspolymorfism

Om jag har en B som ärver av A och jag vill kunna göra:

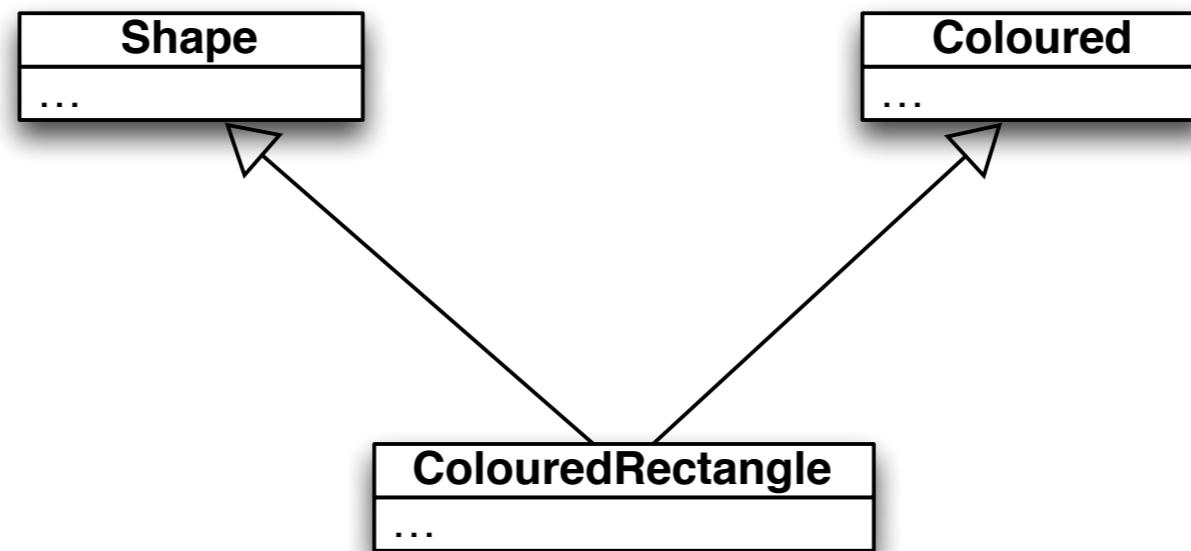
C c = **new** B(); // Ex. C = ColouredShape och B = Square

måste jag göra så att B också ärver C, men B ärver ju redan A!

# Java har enkelt arv [single inheritance]

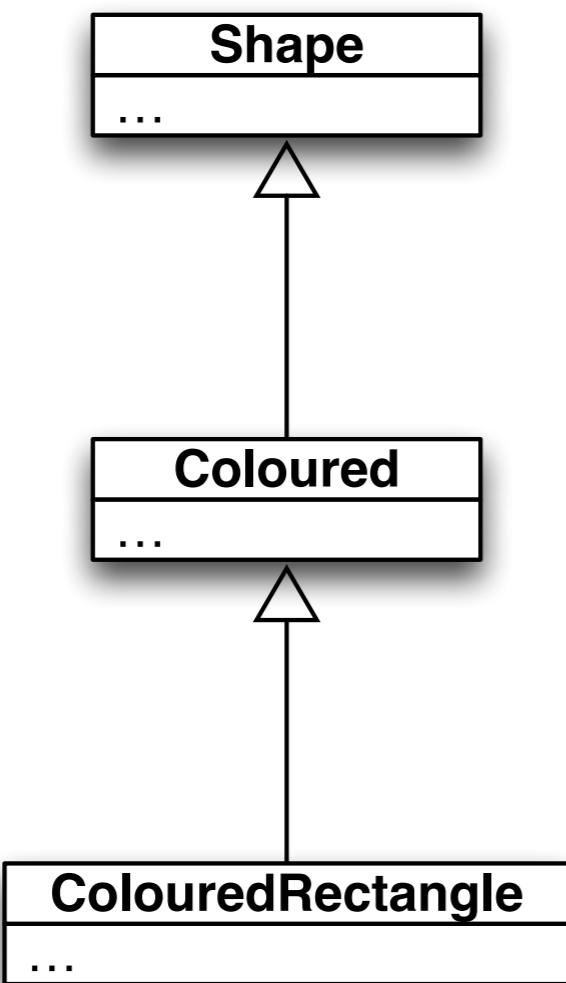
---

- Multipelt arv är **inte** möjligt i Java, dvs. detta går ej:

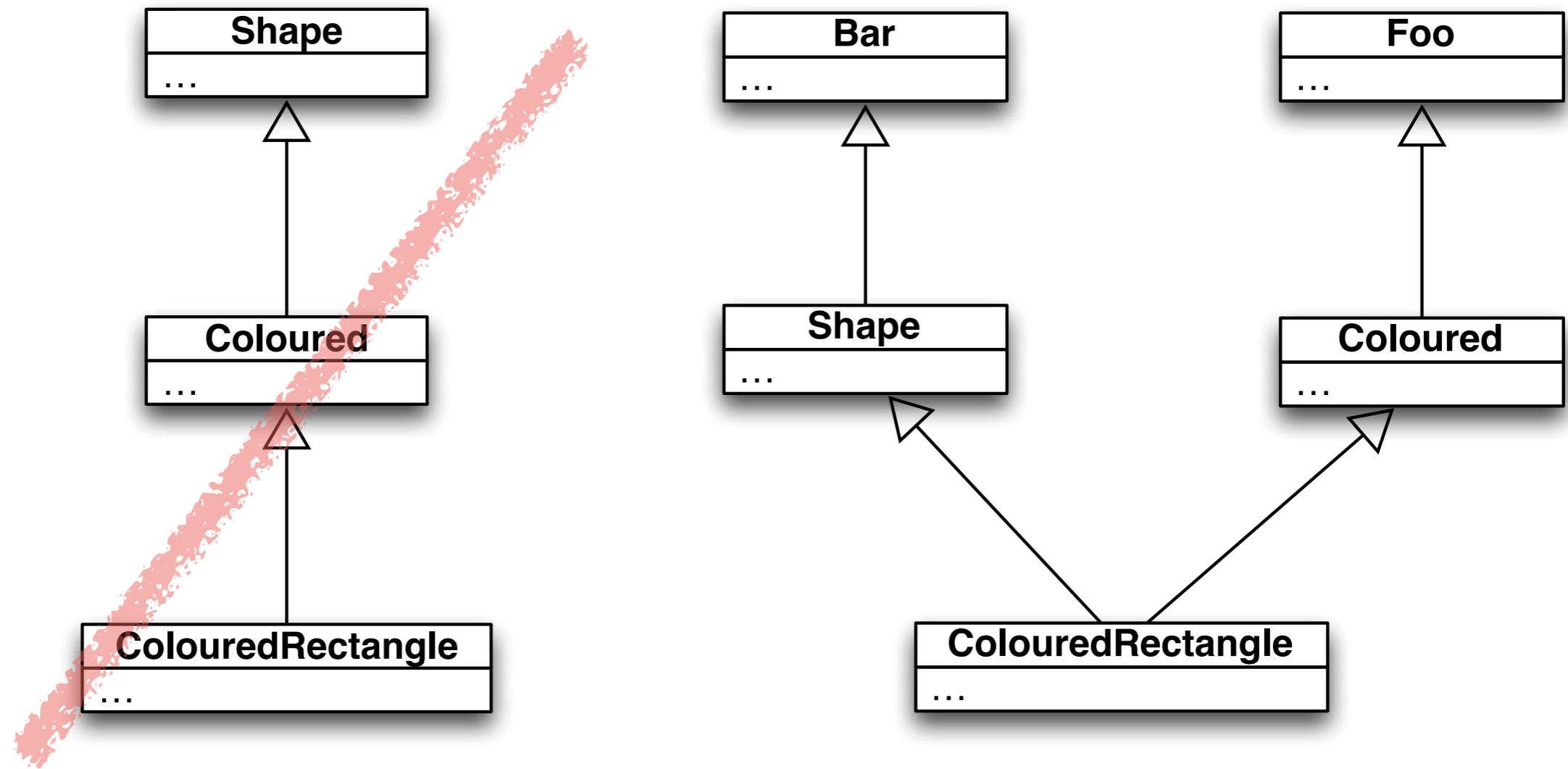


# Lösning #1: Linearisering av arvshierarkin

---

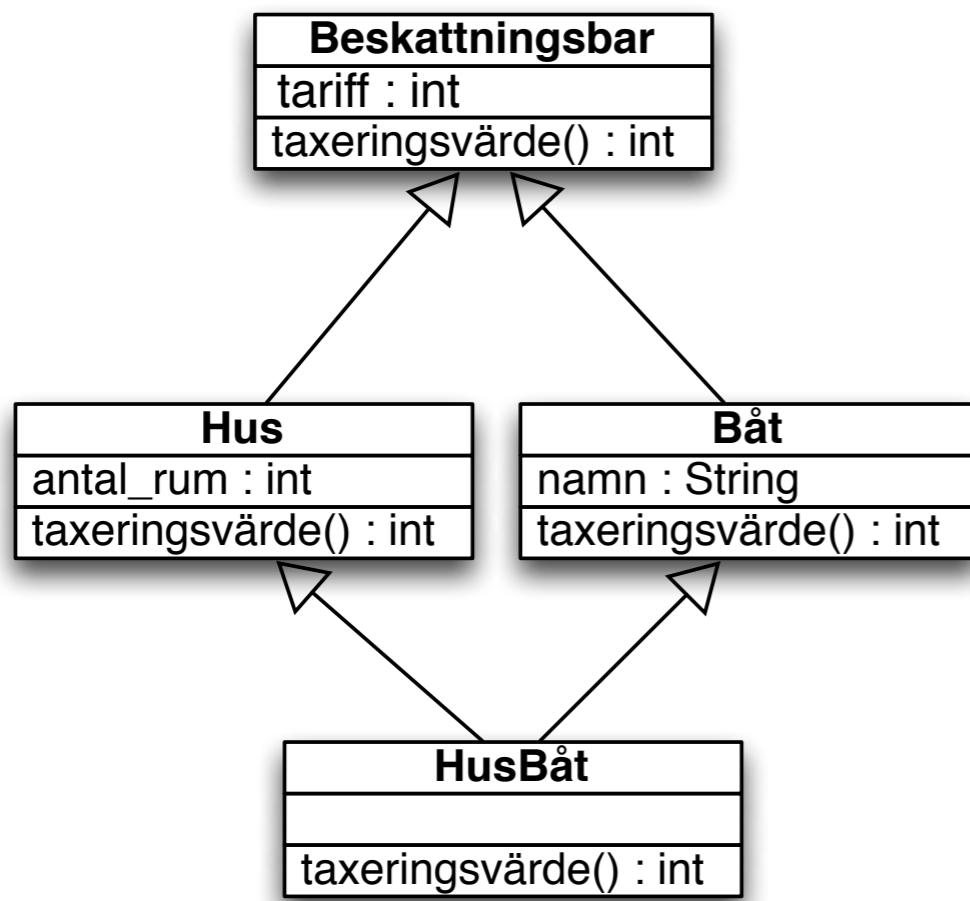


## Lösning #2: Tillåt flera superklasser

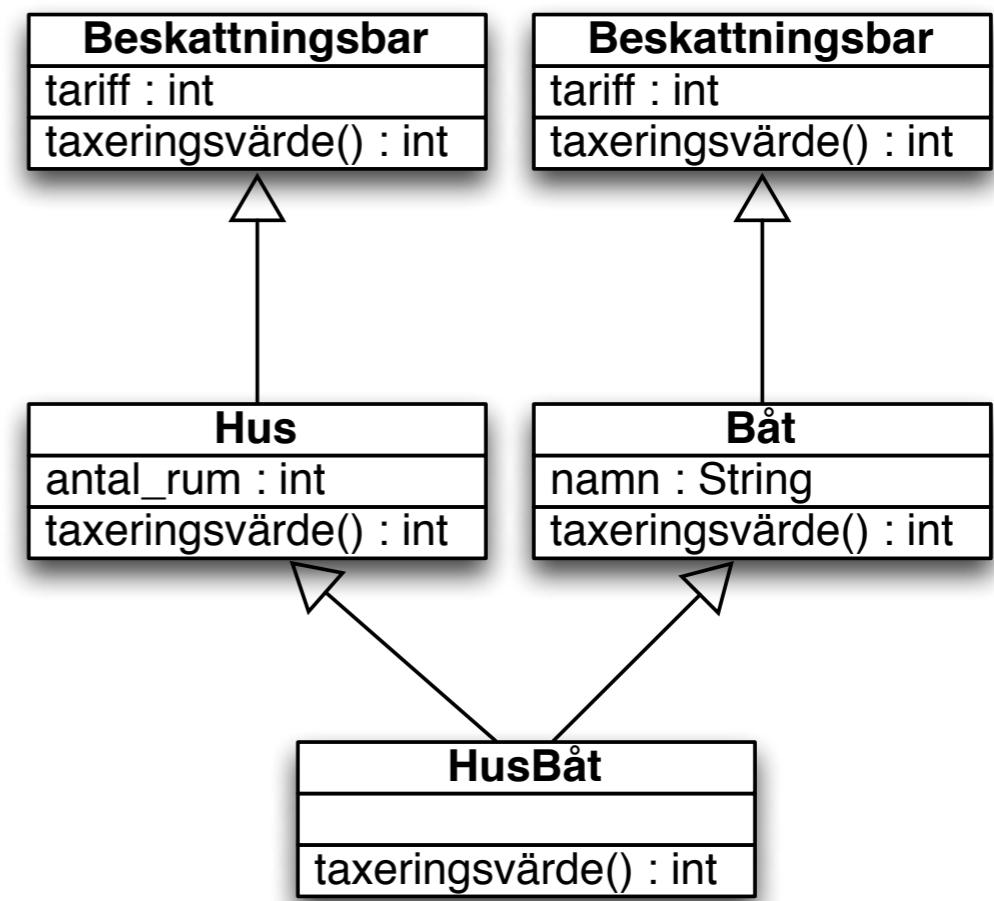


I regel inte möjligt...

# Multipelt arv är problematiskt



*Hur många tariffer?*



*Hur många tariffer?*

# Multipelt arv har inte någon given lösning

---

- En tariff är rätt ibland

Båt och Hus och HusBåt har samma tillåtna värden på tariffen

**Problem:** inkompatibelt beteende

- Två tariffer är rätt ibland

Båt-delen och Hus-delen av Husbåten har var sin tariff

**Problem:** vad betyder `this.tariff` eller `taxeringsvärd()` i Husbåt-klassen?

- Möjliga lösningar

Tillåt båda (C++)

Kräv att programmeraren löser alla konflikter i subklassen (Eiffel, Java 8, m.fl.)

Undvik multipelt arv (Java <8, C#, Ruby, m.fl.)

# Interface-begreppet



# Javas föreskrivna lösning: Interface

---

- Samma styrka som multipelt arv för subtypspolymorfism
- Inga av nackdelarna med multipelt arv
- Exempel:

```
public interface Coloured {  
    public Colour getColour();  
    public void setColour(Colour c);  
    public void paintSameAs(Coloured obj)  
}
```

- (Med multipelt arv ovan avses egentligen multipelt implementationsarv)

# Interface

---

- En specifikation av ett protokoll, d.v.s. ett antal metodsignaturer

Kan även ses som ett kontrakt

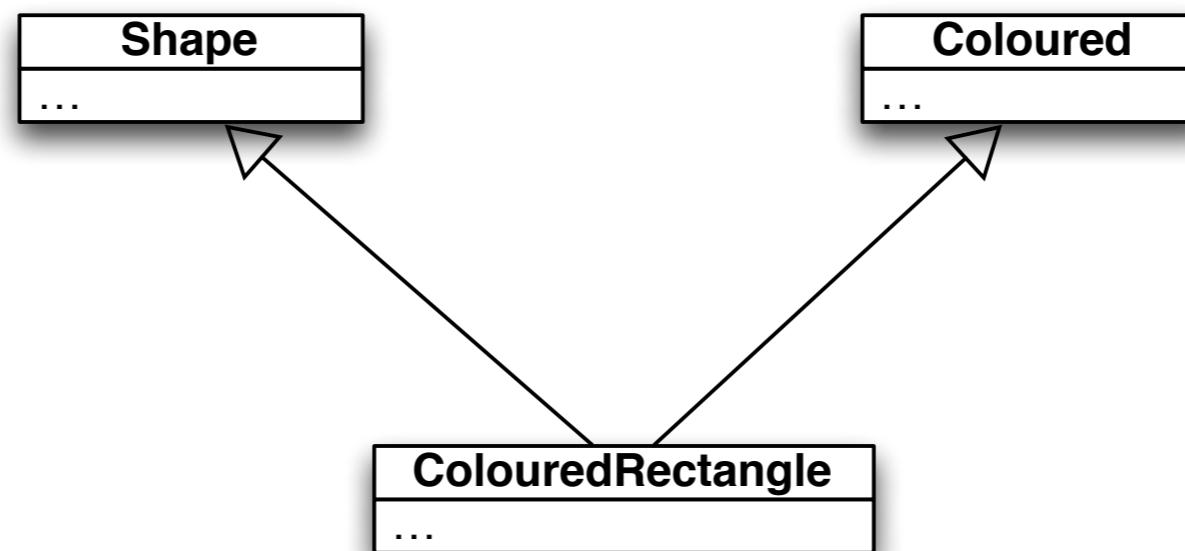
- Exempel:

```
public interface Coloured {  
    public Colour getColour();  
    public void setColour(Colour c);  
    public void paintSameAs(Coloured obj)  
}
```

# Java tillåter multipelt arv för interface

---

- Om Shape, Coloured och ColouredRectangle är interface är detta tillåtet:



# Multipelt arv mellan interface

---

- Interface kan ärva av varandra
- Det finns inget ”rotinterface” på samma sätt som Object är en rotklass
- Multipelt arv (även från samma klass) är oproblematiskt och därför tillåtet

*Dock ej cykler!*

- Specialisering (overriding) finns inte eftersom implementationer ej finns i interface

```
interface A extends B { ... }  
interface B {}  
interface C extends A, B { ... }  
interface D extends B, C { ... }
```

OK!

# Java 8 och default-metoder

---

- Från och med Java 8 kan ett interface innehålla även implementerade metoder

```
interface Comparable {  
    int compareTo(Object other);  
    default boolean lessThan(Object other) {  
        return compareTo(other) < 0;  
    }  
}
```

- En klass som implementerar två interface med konflikterade default-metoder måste explicit tillhandahålla en egen.
- Syntax för att styra bindning vid superanrop:

Type.super.metodNamn(arg);

# Koppling mellan klasser och interface

---

- En klass kan *implementera* ett interface

En nominell relation i form av en **implements**-deklaration i klasshuvudet

```
public class Square implements Coloured {  
    private double R;  
    private double G;  
    private double B;  
    public Colour getColour() { return new Colour(R,G,B); }  
    public void setColour(Colour c) {  
        R = c.getRedComponent();  
        G = c.getGreenComponent();  
        B = c.getBlueComponent();  
    }  
    public void paintSameAs(Coloured obj) {  
        this.setColour(obj.getColour());  
    }  
    ...
```

# Koppling mellan klasser och interface

---

- Varför fungerar följande kod inte?

```
public class Square implements Coloured {  
    private double R;  
    private double G;  
    private double B;  
    ...  
    public void paintSameAs(Coloured obj) {  
        this.R = obj.R;  
        this.G = obj.G;  
        this.B = obj.B;  
    }  
    ...
```

Hur kan vi veta att dessa  
fält finns i en Coloured?

# Koppling mellan klasser och interface

- Partiell implementation av ett interface

```
public interface Coloured {  
    public Colour getColour();  
    public void setColour(Colour c);  
    public void paintSameAs(Coloured obj);  
}  
  
// Saknas!  
  
public class Circle implements Coloured {  
    private Point center;  
    private int radius;  
    public Colour getColour() { ... }  
    public void setColour(Colour c) { ... }  
}
```

# Koppling mellan klasser och interface

- Partiell implementation av ett interface

```
public interface Coloured {  
    public Colour getColour();  
    public void setColour(Colour c);  
    public void paintSameAs(Coloured obj); // Saknas!  
}  
  
public class Circle implements Coloured {  
    private Point center;  
    private int radius;  
    public Colour getColour() { ... }  
    public void setColour(Colour c) { ... }  
}
```

Circle.java:1: Circle is not abstract and does not override  
abstract method paintSameAs(Coloured) in Coloured

# Abstrakta klasser

---

- Klasser som hjälper till att bygga arvshierarkier och inte är avsedda att instantieras

`new A();` går ej om A är abstrakt klass

- En klass K kan deklareras som abstrakt med nyckelordet **abstract**:

`public abstract class K ...`

- En klass K måste deklareras abstrakt om

K implementerar ett interface partiellt, eller

K har en metod M som är deklarerad abstrakt, eller

K ärver en abstrakt metod M utan att override:a/specialisera den

# Interface och subtypspolymorfism

---

- Att göra Coloured till ett interface löser vårt tidigare problem

*Dock måste varje klass själv implementera funktionaliteten – den ärvs ej!*

```
public interface Coloured { ... }
public class Shape { ... }
public class Square extends Shape implements Coloured { ... }
```

```
Square s = new Square();
Shape x;
Coloured c;
x = s;
c = s;
x = c; // Does not compile! Why?
c = x; // Does not compile! Why?
```

# Parametrisk Polymorfism i Java



# Parametrisk polymorfism i Java – ”Generics”

- Ibland behöver man skriva kod som fungerar på samma sätt för objekt av flera olika typer – varför är det dåligt att kopiera kod som vi har gjort här?

```
public class IntList {  
    private class Link {  
        Link next;  
        int value;  
    }  
    private Link first;  
}  
  
public class BooleanList {  
    private class Link {  
        Link next;  
        boolean value;  
    }  
    private Link first;  
}
```

```
public class Base...List {  
    private class Link {  
        Link next;  
        BaseballPlayer value;  
    }  
    private Link first;  
}
```

# En naiv lösning

---

- ...och den enda före Java 1.5 (för många många år sedan)
- Hur skiljer sig denna implementation från de på föregående sida?

```
public class List {  
    private class Link {  
        Link next;  
        Object value;  
    }  
    private Link first;  
}
```

# Typsäkerhet

---

```
IntList list1 = new IntList();
BaseballPlayerList list2 = new BaseballPlayerList();
BooleanList list3 = new BooleanList();

int v1 = 7;
BaseballPlayer v2 = new BaseballPlayer(...);
boolean v3 = false;

list1.add(v1); // ok
list1.add(v2); // does not compile
list1.add(v3); // does not compile
list2.add(v1); // does not compile
list2.add(v2); // ok
list2.add(v3); // does not compile
list3.add(v1); // does not compile
list3.add(v2); // does not compile
list3.add(v3); // ok
```

# En lista av Object kan innehålla alltting...

---

```
List list1 = new List();
List list2 = new List();
List list3 = new List();

int v1 = 7;
BaseballPlayer v2 = new BaseballPlayer(...);
boolean v3 = false;

list1.add(v1); // ok
list1.add(v2); // compiles, but is it safe?
list1.add(v3); // compiles, but is it safe?
list2.add(v1); // compiles, but is it safe?
list2.add(v2); // ok
list2.add(v3); // compiles, but is it safe?
list3.add(v1); // compiles, but is it safe?
list3.add(v2); // compiles, but is it safe?
list3.add(v3); // ok
```

# Fungerar detta eller ej?

---

```
List list1 = new List();

int v1 = 7;
BaseballPlayer v2 = new BaseballPlayer(...);
boolean v3 = false;

list1.add(v1); // ok
list1.add(v2); // compiles, but is it safe?
list1.add(v3); // compiles, but is it safe?

int v4 = (int) list1.get(1);
BaseballPlayer v5 = (BaseballPlayer) list1.get(2);
boolean v6 = (boolean) list1.get(3);
```

ClassCastException on line ...

# Parametrisk polymorfism

---

- Introducerades i Java 1.5
- Implementationen något begränsad på grund av bakåtkompatibilitet

```
public class List <ElementType> {           List<Person>
    private class Link {                    List<String>
        Link next;                         List<Object>
        ElementType value;                  }
        private Link first;                }
    }
```

- En klass introducerar en typ
- En parametriskt polymorf klass introducerar en typkonstruktör som kan användas för att skapa typer

# Parametriskt polymorfa typer

---

```
List<Integer> list1 = new List<Integer>();
List<BaseballPlayer> list2 = new List<BaseballPlayer>();
List<Boolean> list3 = new List<Boolean>();

int v1 = 7;
BaseballPlayer v2 = new BaseballPlayer(...);
boolean v3 = false;

list1.add(v1); // ok
list1.add(v2); // does not compile
list1.add(v3); // does not compile
list2.add(v1); // does not compile
list2.add(v2); // ok
list2.add(v3); // does not compile
list3.add(v1); // does not compile
list3.add(v2); // does not compile
list3.add(v3); // ok
```

# Parametriskt polymorfa typer

```
List<Integer> list1 = new List<Integer>();  
List<BaseballPlayer> list2 = new List<BaseballPlayer>();  
List<Boolean> list3 = new List<Boolean>();
```

```
int v1 = 7  
BaseballP  
boolean v3
```

Utvikning: varför Integer och Boolean och inte  
**int** och **bool**?

```
list1.add(  
list1.add(  
list1.add(  
list2.add(  
list2.add(  
list2.add(  
list3.add(  
list3.add(  
list3.add(v3); // ok
```

Svar: en **int** är en primitiv typ, och Java stöder  
inte primitiva typargument till typkonstruktorer.

Java konverterar automatiskt mellan primitiver  
(t.ex. **int**) och deras objekt motsvarigheter (t.ex.  
Integer) varför denna kod fungerar!  
(Detta kallas för autoboxing.)

# Att kedja typparametrar

---

- Om vår lista inte använt en inre klass...

```
public class List {  
    private Link first;  
}  
public class Link {  
    private Link next;  
    private Object value;  
}
```

# Att kedja typparametrar

---

- Om vår lista inte använt en inre klass...

```
public class List {  
    private Link first;  
}  
  
public class Link {  
    private Link next;  
    private Object value;  
}
```

```
public class List<E> {  
    private Link<E> first;  
}  
  
public class Link<E> {  
    private Link<E> next;  
    private E value;  
}
```

# Att kedja typparametrar

- Om vår lista inte använt en inre klass...

```
public class List {  
    private Link first;  
}  
  
public class Link {  
    private Link next;  
    private Object value;  
}
```

Parameter

```
public class List<E> {  
    private Link<E> first;  
}  
  
public class Link<E> {  
    private Link<E> next;  
    private E value;  
}
```

Argument

Parameter

Argument

Användande som typ

# Manipulation av objekt av typparameter-typ

---

- Vad kan man göra med en variabel vars typ är okänd?

Eller – bättre uttryckt – vilken typ har en variabel vars typ är en typparameter?

```
public class List<E> {
    private Link first;
    private class Link {
        Link next;
        E value;
        void m() {
            value.frob(); // kompilerar?
        }
    }
}
```

# Rotklassen till undsättning

---

- Under huven expanderas...
- ...till...

```
public class List<E> {  
    private class Link {  
        Link next;  
        E value;  
    }  
    private Link first;  
}
```

```
public class List<E extends Object> {  
    private class Link {  
        Link next;  
        E value;  
    }  
    private Link first;  
}
```

# Rotklassen till undsättning

- En övre gräns (upper bound) för en typparameter låter oss bättre resonera om vad den kan bindas till
- I listan till höger kan E bindas till alla typer som ärver av Object
- Mer specifika typer är också möjliga, som här:  
*Nu kan man anropa metoder på value som finns i Shape-klassen*
- Priset är att List<String> ej längre är möjlig då String inte är en subtyp till Shape

```
public class List<E extends Object> {  
    private class Link {  
        Link next;  
        E value;  
    }  
    private Link first;  
}
```

```
public class List<E extends Shape> {  
    private class Link {  
        Link next;  
        E value;  
    }  
    private Link first;  
}
```

# En ”svag” implementation

- Under huven kompileras...

```
public class List<ElementType> {  
    private class Link {  
        Link next;  
        ElementType value;  
    }  
    private Link first;  
}
```

- ...ned till...

```
public class List {  
    private class Link {  
        Link next;  
        Object value;  
    }  
    private Link first;  
}
```

Detta förklarar varför vi inte kunde binda ElementType till int förut – eftersom en int inte är ett Object.

# ...och med explicita övre gränser

---

- Under huven kompileras...

```
public class List<E extends Shape> {  
    private class Link {  
        Link next;  
        E value;  
    }  
    private Link first;  
}
```

- ...ned till...

```
public class List {  
    private class Link {  
        Link next;  
        Shape value;  
    }  
    private Link first;  
}
```

# En ”svag” implementation

- Under huven kompileras...

```
List<Integer> list1 = new List<Integer>();  
int v1 = 7;  
boolean v3 = false;  
  
list1.add(v1); // ok  
list1.add(v3); // does not compile
```

- ...ned till...

Detta är fortfarande typsäkert eftersom all interaktion med listan skyddas av (Integer)-omvandlingar!

```
List list1 = new List();  
int v1 = 7;  
boolean v3 = false;  
  
list1.add((Integer) v1); // ok  
list1.add((Integer) v3); // does not compile
```

# Man kan binda typparametrar ”överallt”

---

```
public class StringList extends List<String> { }

public class Foo {
    public List<Boolean> getBar() { ... }
}
```

# Titta också på...

---

- Screencasten om parametrisk polymorfism
- Titta på hur parametrisk polymorfism används i Javas standardbibliotek
- Notera användanden som t.ex.

Comparable<K>

Class<K>

etc.

- Varför det kan vara problematiskt att ge inre klasser (i motsats till nästlande klasser) typparametrar istället för att bara använda den omslutande klassens typparametrar

# Doxygen el. JavaDoc

---

- Doxygen är ett verktyg för att extrahera kommentarer ur kod och generera dokumentation

Hela JavaAPI:et är skapat med hjälp av JavaDoc från koden

# MyClass.java

```
/** Description of MyClass
 *
 * @author John Doe
 * @author Jane Doe
 * @version 6.0z Build 9000 Jan 3, 1970.
 */
public class MyClass
{
    /** Description of myIntField */
    public int myIntField;
    /** Description of MyClass()
     *
     * @throws MyException Description of when the exception is thrown
     */
    public MyClass() throws myException
    {
        // Blah Blah Blah...
    }
    /** Description of myMethod(int a, String b)
     *
     * @param a Description of a
     * @param b Description of b
     * @return Description of c
     */
    public Object myMethod(int a, String b)
    {
        Object c;
        // Blah Blah Blah...
        return c;
    }
}
```



```
$ javadoc -d docs MyClass.java

Generating docs/package-frame.html...
Generating docs/package-summary.html...
Generating docs/package-tree.html...
Generating docs/constant-values.html...
Building index for all the packages and classes...
Generating docs/overview-tree.html...
Generating docs/index-all.html...
Generating docs/deprecated-list.html...
Building index for all classes...
Generating docs/allclasses-frame.html...
Generating docs/allclasses-noframe.html...
Generating docs/index.html...
Generating docs/help-doc.html...
```

## Resultatet!

### Class MyClass

java.lang.Object  
MyClass

```
public class MyClass
extends java.lang.Object
```

#### Description of MyClass

#### Field Summary

##### Fields

###### Modifier and Type

int

###### Field and Description

myIntField

Description of myIntField

#### Constructor Summary

##### Constructors

###### Constructor and Description

MyClass()

Description of MyClass()

#### Method Summary

##### All Methods

##### Instance Methods

##### Concrete Methods

###### Modifier and Type

java.lang.Object

###### Method and Description

myMethod(int a, java.lang.String b)

Description of myMethod(int a, String b)

#### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait

#### Field Detail



# gc.h

```
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>
#ifndef __gc__
#define __gc__
/// The opaque data type holding all the heap data
typedef struct heap_t heap_s;
/// The signature of the trace function
typedef void *(*trace_f)(heap_s *h, void *obj);
/// The signature of object-specific trace functions. It will be
/// called for its specific objects, and be given a generic trace
/// function f to be called on each pointer inside obj.
typedef void *(*s_trace_f)(heap_s *h, trace_f f, void *obj);
/// Create a new heap with bytes total size (including both spaces
/// and metadata), meaning strictly less than bytes will be
/// available for allocation.
///
/// \param bytes the total size of the heap in bytes
/// \return the new heap
heap_s *h_init(size_t bytes);
/// Delete a heap.
///
/// \param h the heap
void h_delete(heap_s *h);
/// Delete a heap and trace, killing off stack pointers.
///
/// \param h the heap
/// \param dbg_value a value to be written into every pointer into h on the stack
void h_delete_dbg(heap_s *h, void *dbg_value);
/// Allocate a new object on a heap with a given format string.
///
/// Valid characters in format strings are:
/// - 'd' -- for sizeof(int) bytes 'raw' data
/// - '*' -- for a sizeof(void *) bytes pointer value
/// - '\0' -- null-character terminates the format string
///
/// \param h the heap
/// \param layout the format string
/// \return the newly allocated object
///
/// Note: the heap does *not* retain an alias to layout.
void *h_alloc_struct(heap_s *h, char *layout);
/// Allocate a new object on a heap with a given size, and
/// object-specific trace function.
///
/// \sa s_trace_f
///
/// \param h the heap
/// \param bytes the size in bytes
/// \param f the object-specific trace function
/// \return the newly allocated object
void *h_alloc_union(heap_s *h, size_t bytes, s_trace_f f);
/// Allocate a new object on a heap with a given size.
///
/// Objects allocated with this function will *not* be
/// further traced for pointers.
///
/// \param h the heap
/// \param bytes the size in bytes
/// \return the newly allocated object
void *h_alloc_data(heap_s *h, size_t bytes);
/// Manually trigger garbage collection.
///
/// Garbage collection is otherwise run when an allocation is
/// impossible in the available consecutive free memory.
///
/// \param h the heap
/// \return the number of bytes collected
size_t h_gc(heap_s *h);
/// Returns the available consecutive free memory.
///
/// \param h the heap
/// \return the available consecutive free memory.
size_t h_avail(heap_s *h);
#endif
```



# Project Work Test Implementation

The screenshot shows a doxygen-generated file reference for the file `gc.h`. The top navigation bar includes links for Main Page, Classes, and Files (which is currently selected), along with a search bar. Below the navigation is a secondary navigation bar with File List and File Members. A link to the source code of the file is also present.

## gc.h File Reference

#include <stdlib.h>  
#include <stdbool.h>  
#include <stdint.h>

Go to the source code of this file.

## Typedefs

typedef struct heap\_t heap\_s  
The opaque data type holding all the heap data.

typedef void \*(\* trace\_f )(heap\_s \*h, void \*obj)  
The signature of the trace function.

typedef void \*(\* s\_trace\_f )(heap\_s \*h, trace\_f f, void \*obj)

## Functions

heap\_s \* h\_init (size\_t bytes)  
void h\_delete (heap\_s \*h)  
void h\_delete\_dbg (heap\_s \*h, void \*dbg\_value)  
void \* h\_alloc\_struct (heap\_s \*h, char \*layout)  
void \* h\_alloc\_union (heap\_s \*h, size\_t bytes, s\_trace\_f f)  
void \* h\_alloc\_data (heap\_s \*h, size\_t bytes)  
size\_t h\_gc (heap\_s \*h)  
size\_t h\_avail (heap\_s \*h)

## Typedef Documentation

typedef void\*(\* s\_trace\_f)(heap\_s \*h, trace\_f f, void \*obj)

The signature of object-specific trace functions. It will be called for its specific objects, and be given a generic trace function f to be called on each pointer inside obj.

doxygen -g gc.doxy

Skapar en fil med  
inställningar

emacs gc.doxy

Ändra på input så  
att rätt filer pekas ut

doxygen -s gc.doxy

Generera  
dokumentationen