Föreläsning 17–19

Andreina Francisco
Lars-Henrik Eriksson
(based on slides by Tobias Wrigstad)

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

A Social Network

The social network FooBar works much like Facebook. Each user has a profile, and each profile is linked to zero or more friends. There are two types of friends, close friends and acquaintances. A user can post status updates that can mention other profiles. A status update can be liked or disliked by users (including the user who posted it), and commented on. Any comment may also be liked or disliked.

Each profile has an event log that contains status updates in reverse chronological order, ie. most recent first. An event log for user A lists all status updates that A has made, as well as all relevant status updates for all friends. A status update with a close friend is always considered relevant, but for others, it should be "hot" if A is mentioned in the status update or in any of its comments. A status update is also considered hot if the number of dislikes + the number of dislikes + the number of comments exceeds a certain threshold value T.

Users can "push" each other. If A pushes to B, this means that A's all status updates are considered relevant to B for a week and vice versa. You cannot have more than 5 "active pushers" at a time.

One user can ask to be friends with another user, who has to accept first. Friendship is a symmetrical relationship. A user can list one or more friends as close friends. Close friendships are not necessarily symmetrical. However, there is a special category of close friendship that is symmetrical, close to kinship and "in a relationship". A user can list one or more friends as family / in a relationship, and they must accept first. It is possible to terminate all types of friendships, including close friendships and relationships.

Each **user** has a **profile**, and each profile is linked to zero or more **friends**. There are two types of friends: **close friends** and **acquaintances**. A user can post **status updates** that can mention other profiles. A status update can be liked or disliked by users (including the user who posted it), and commented on. A **comment** may also be liked or disliked.

Each profile has an **event log** that contains status updates in reverse chronological order, ie. most recent first. An event log for user A lists all status updates that A has made, as well as all relevant status updates for all friends. A status update with a close friend is always considered relevant, but for others, it should be "hot" if A is mentioned in the status update or in any of its comments. A status update is also considered hot if the number of dislikes + the number of dislikes + the number of comments exceeds a certain threshold value T.

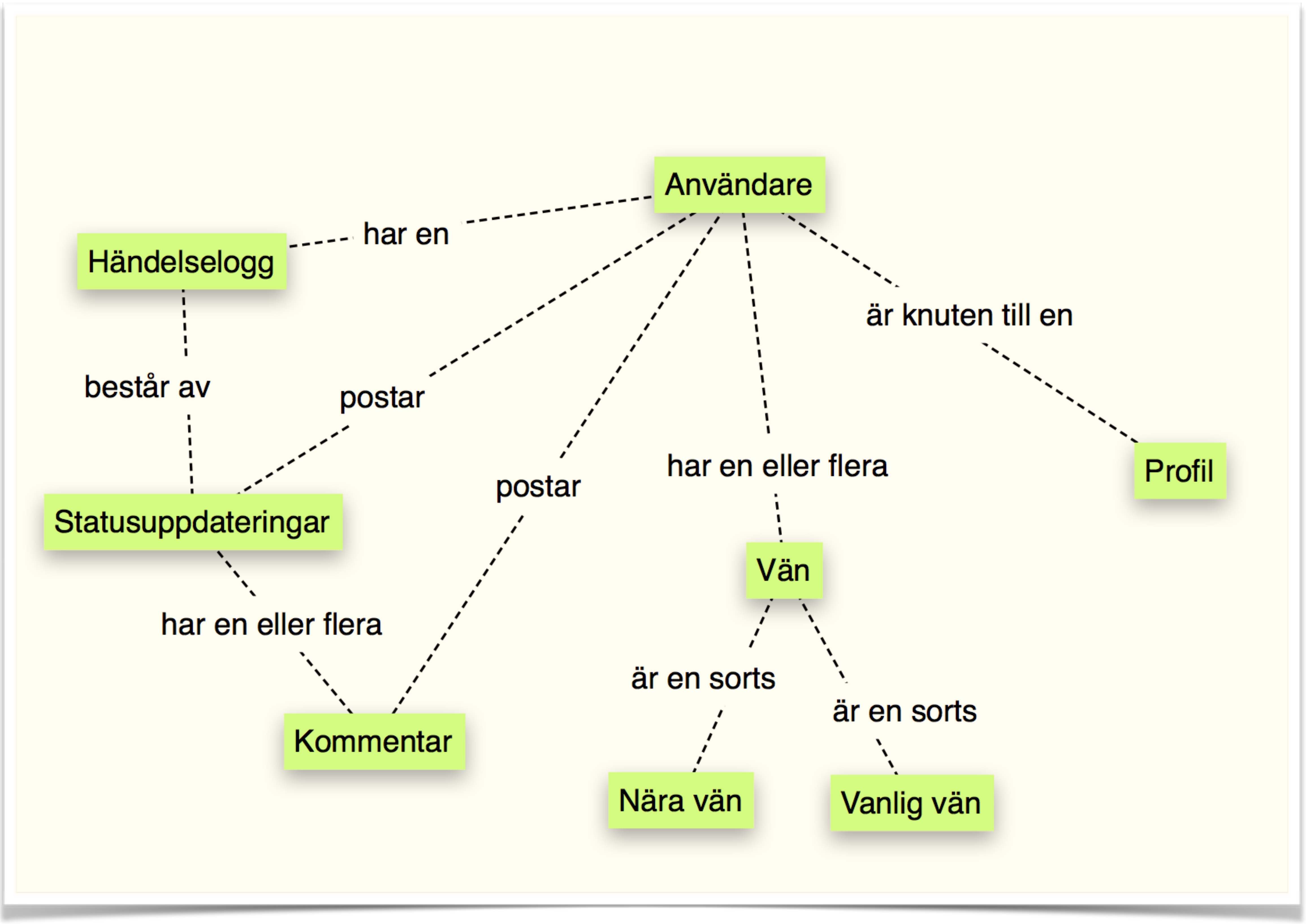
Objects

- User
- Profile
- Friends

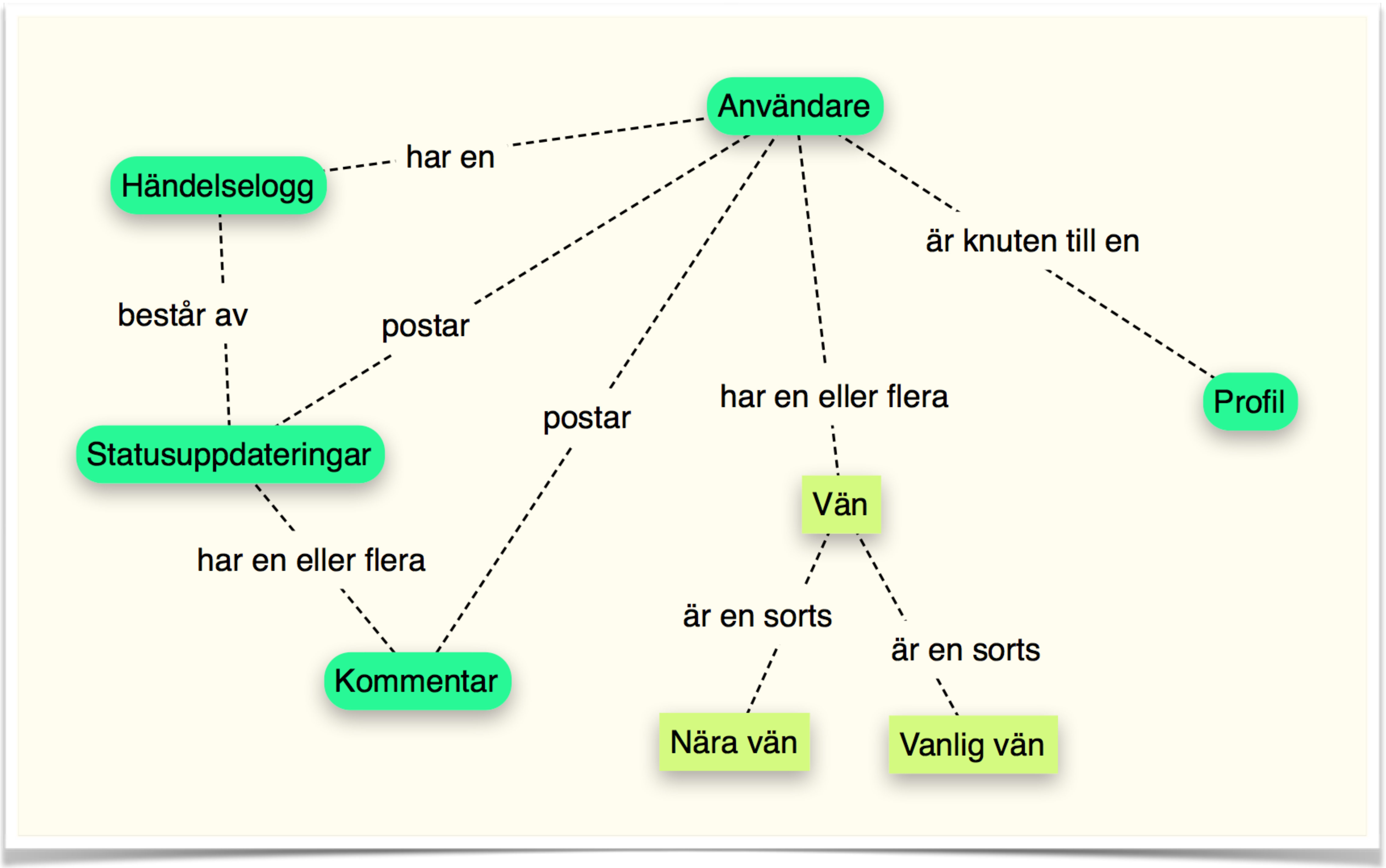
  There are close friends

  … and ordinary friendship
- Status updates
- Comments
- Event logs

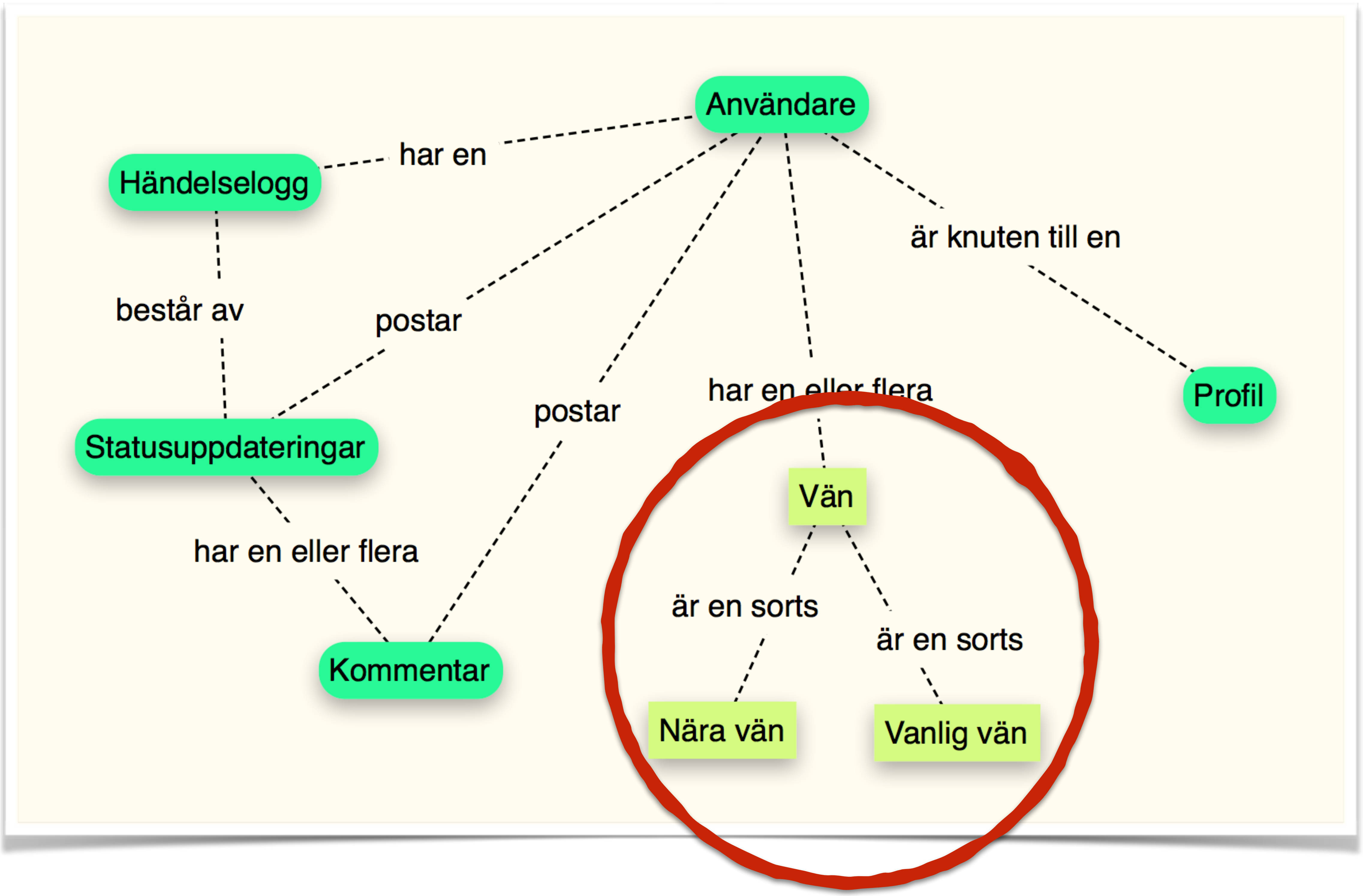
Object Relationships



Classes


Användare
har en
Händelselogg
är knuten till en
består av
postar
postar
har en eller flera
Profil
Statusuppdateringar
Vän
har en eller flera
är en sorts
är en sorts
Kommentar
Nära vän
Vanlig vän

Not Classes!


Användare
har en
Händelselogg
är knuten till en
består av
postar
postar
har en eller flera
Profil
Statusuppdateringar
Vän
har en eller flera
är en sorts
är en sorts
Kommentar
Nära vän
Vanlig vän

Why is “Friend” NOT a Class?

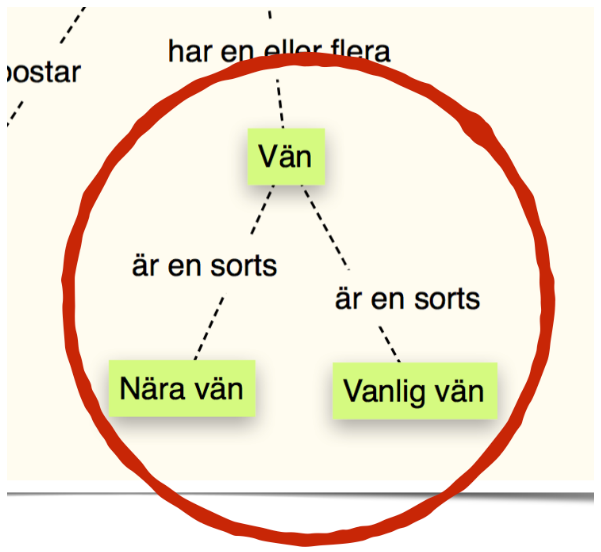
- Friendship is a **relationship** between two objects

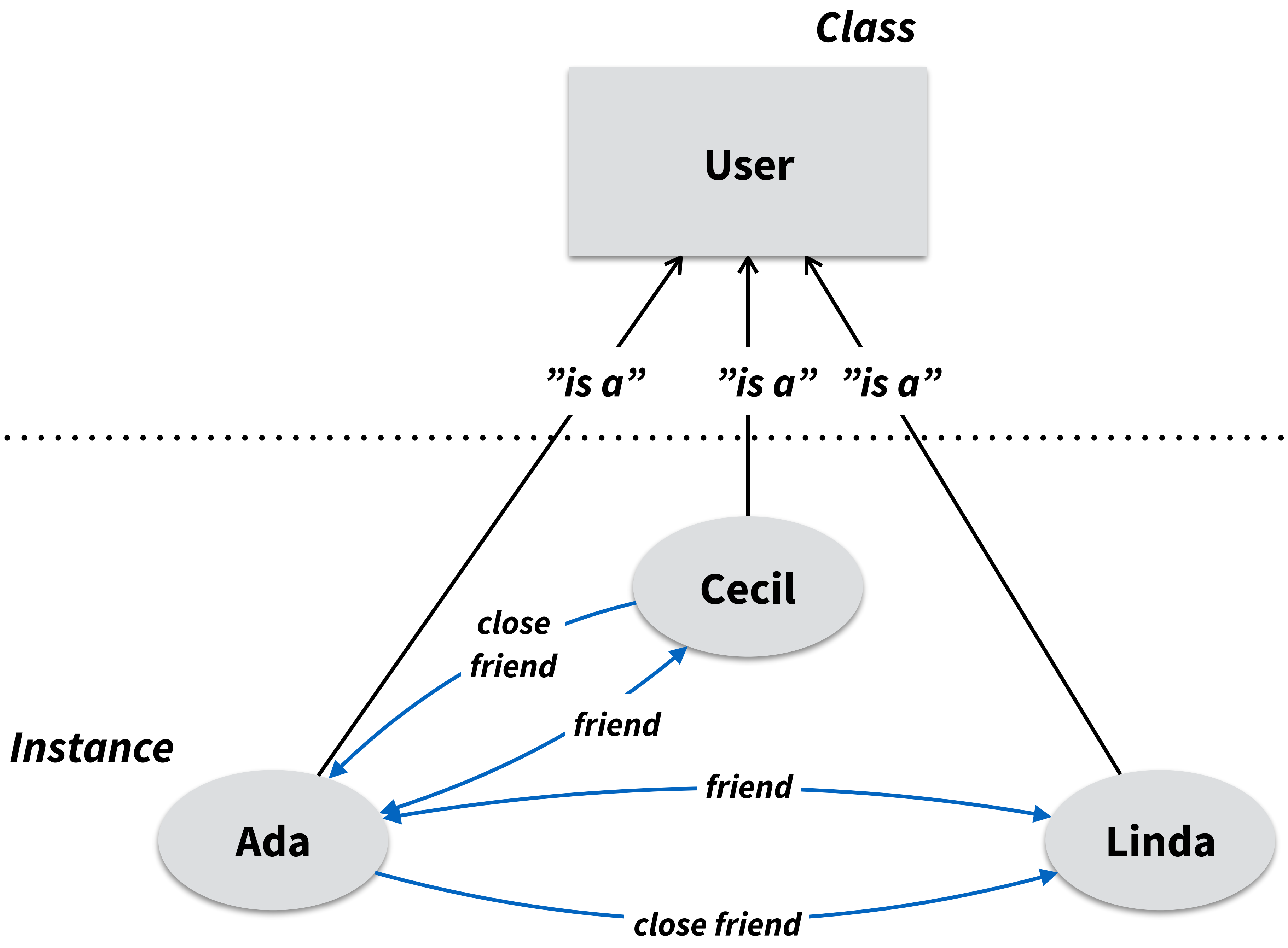
  Being a friend is not something that defines the concept of user

- Two users are required to model friendships

```
class User {
    User[] friends;
    User[] closeFriends;

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

- A **FriendshipBracelet** could be a class, but a friend is not!





relationship between objects

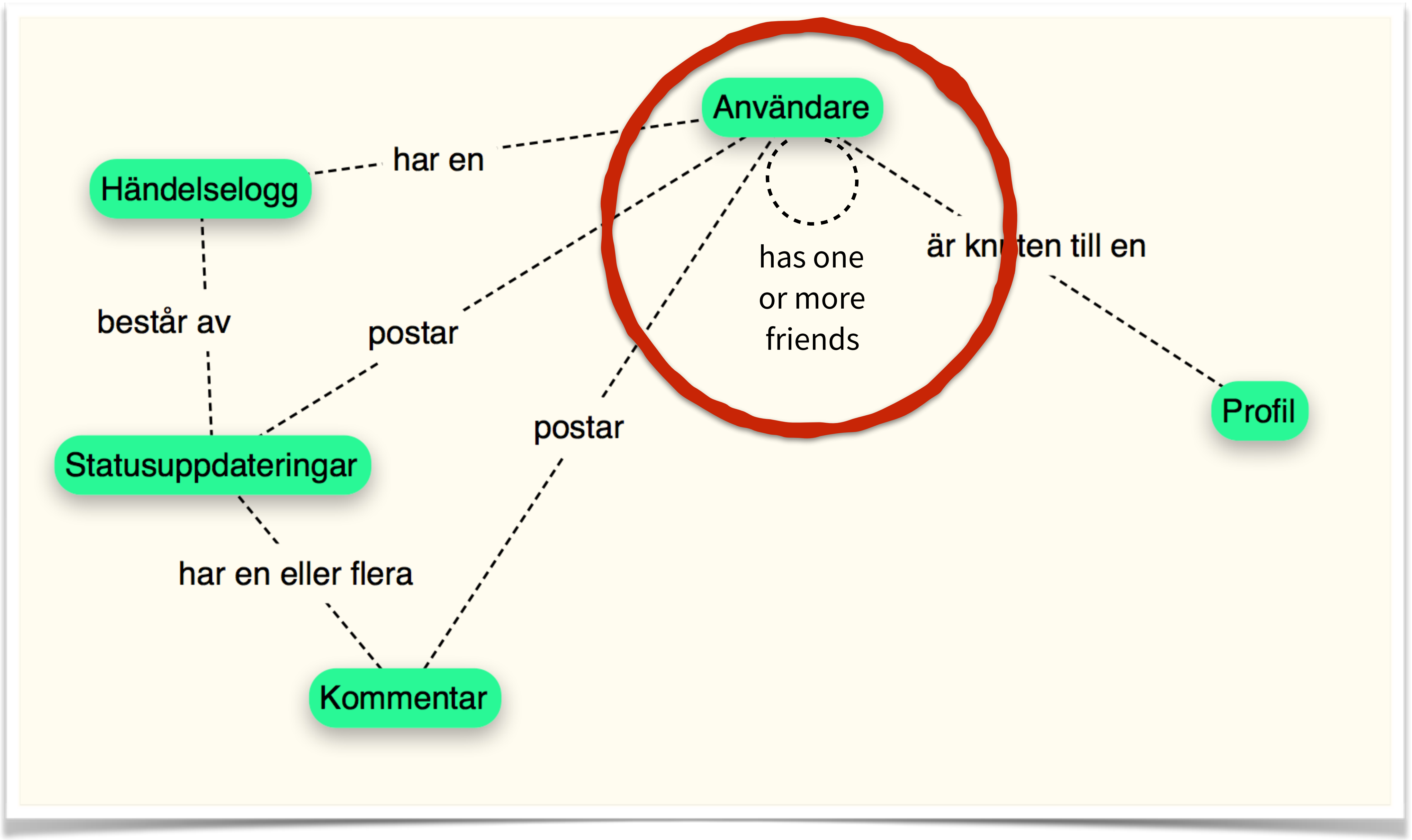
```
class User {
    User[] friends;
    User[] closeFriends;

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

The relationship is modelled as a pointer/connection between the objects

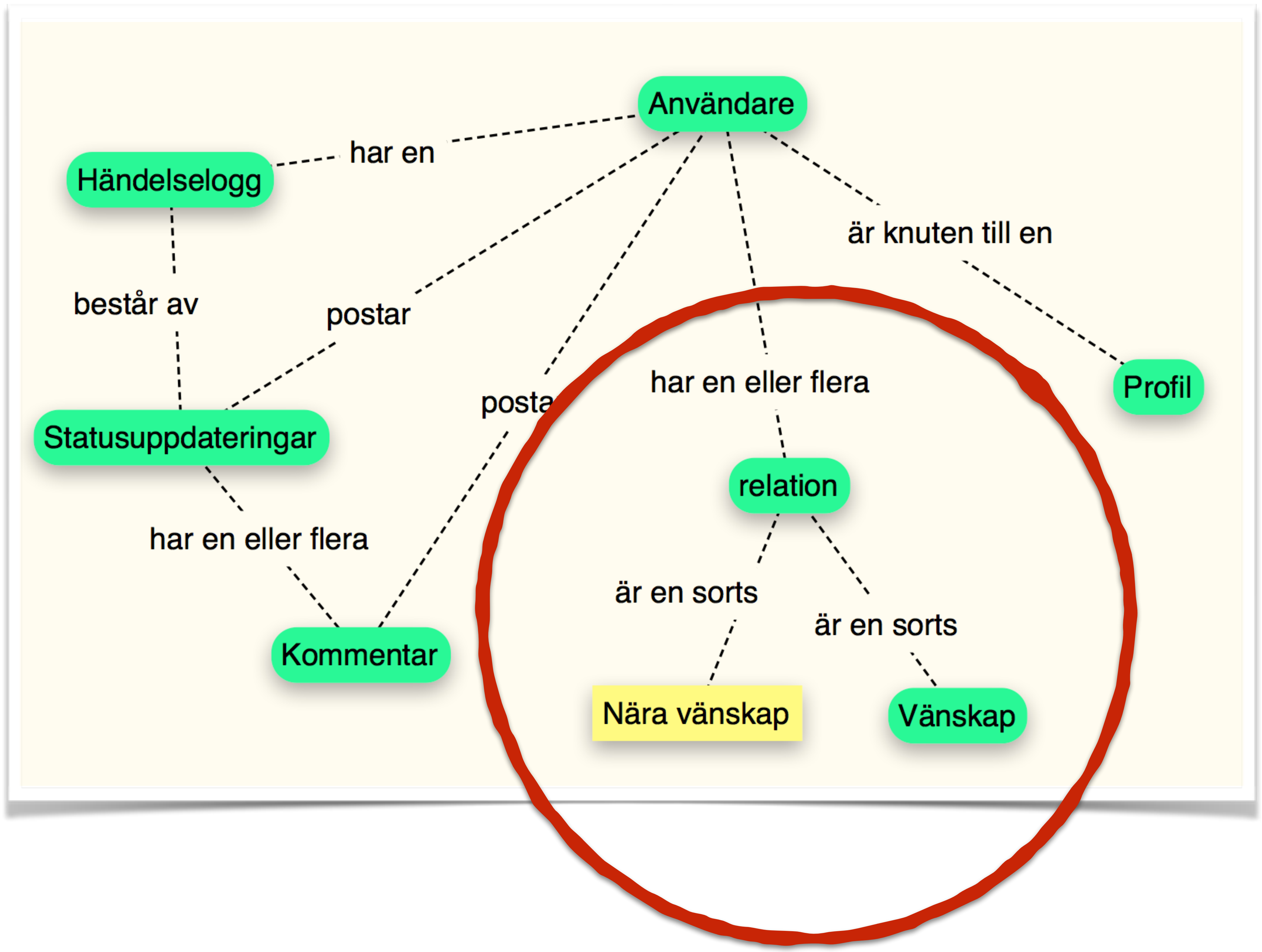
```
class Relation { ... }
class Friendship extends Relation { ... }

class User {
    Relation[] connections;

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

The relationship modelled as instances of specific classes

Friendship as a Reflexive Relationship between Users (symmetrical) (user)



Relationship unified as an object (more powerful)



Classes from objects

- Most objects we have seen are examples of concepts in the domain of the system and should therefore have corresponding classes

  The program is a model of reality!

  Some properties of objects did not give rise to new classes

  Sometimes it makes sense to model relationships as objects / classes

- **Inheritance**: relations between classes

  Different types of relationships (all the lines saying "is a kind")

  Sometimes it makes sense to generalise and create a common superclass for classes that are very similar

Inheritance

Mechanism for *reuse* of code / design / ideas

A class can build on another class
— add new fields, methods
— redefine methods

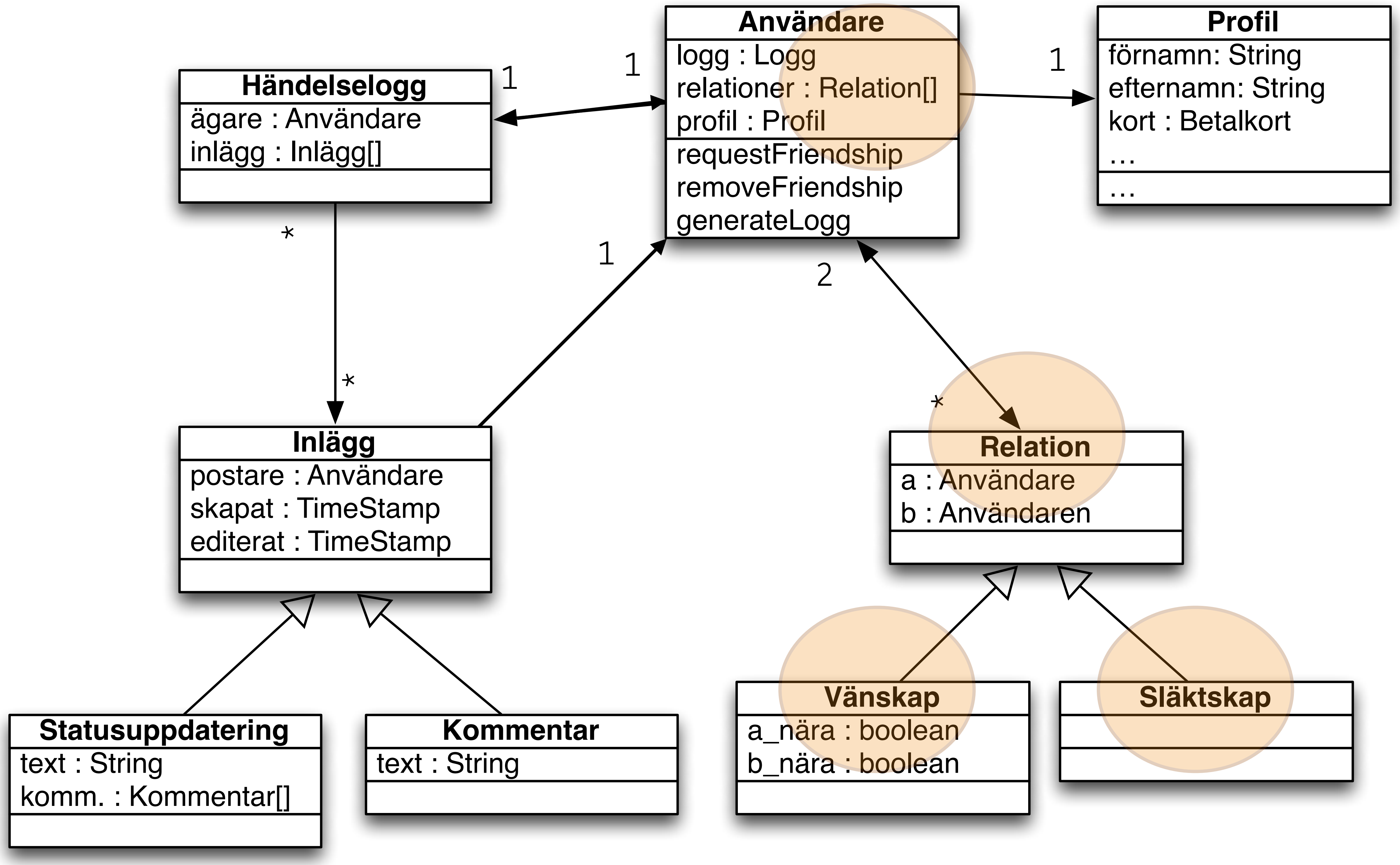
A modularisation strategy

A class can be a specialisation of another class
— *Student is a specialisation of Human*,
— *Human is a specialisation of Mammals*,
— *Mammals are a specialisation of Creature*

A lot of hype, not as useful as "they" say

However, where inheritance works, it is very good!

An Initial Class Diagram [UML]

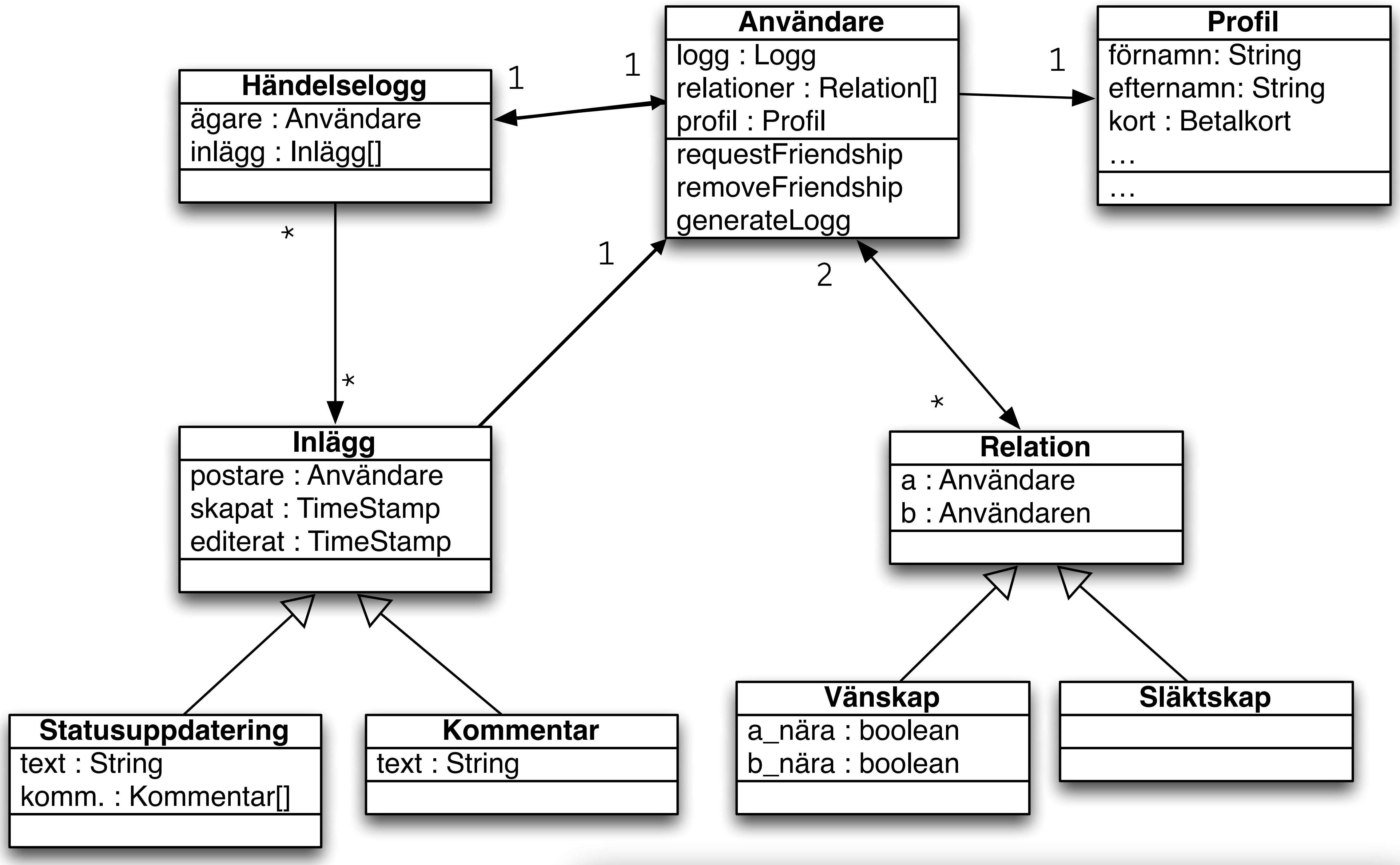


Class diagram expressed in code (partial)

```
class EventLog

class User

class Profile

class Post
class StatusUpdate extends Post
class Comment extends Post
```

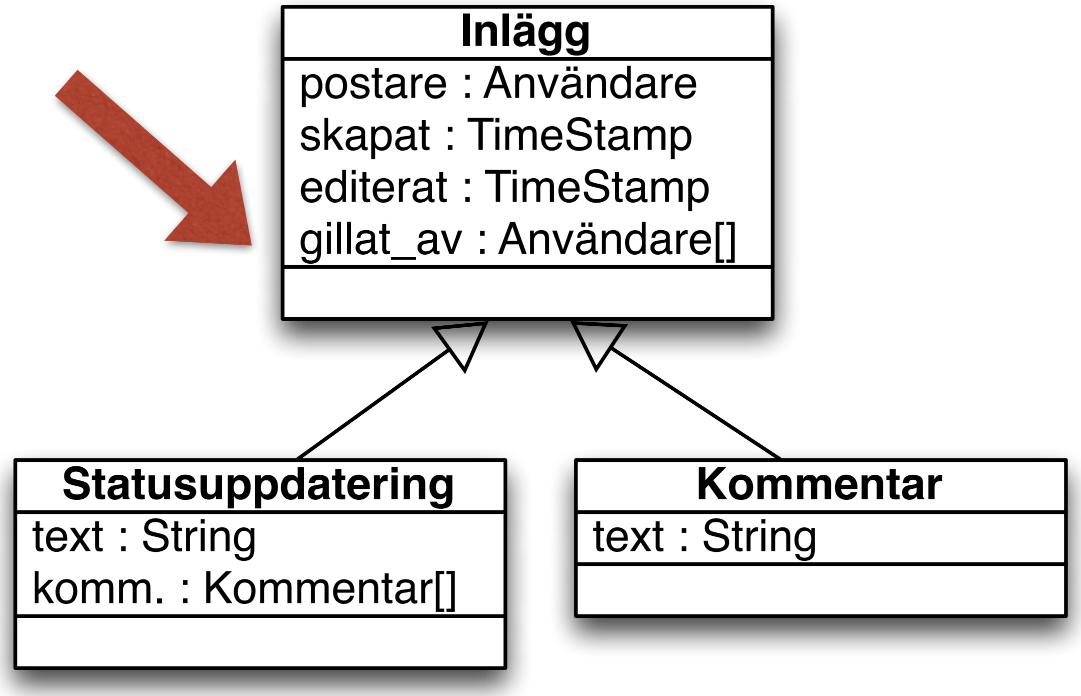
```
class Relation
class Friendship extends Relation
class Relationship extends Relation
```

An Initial Class Diagram [UML]



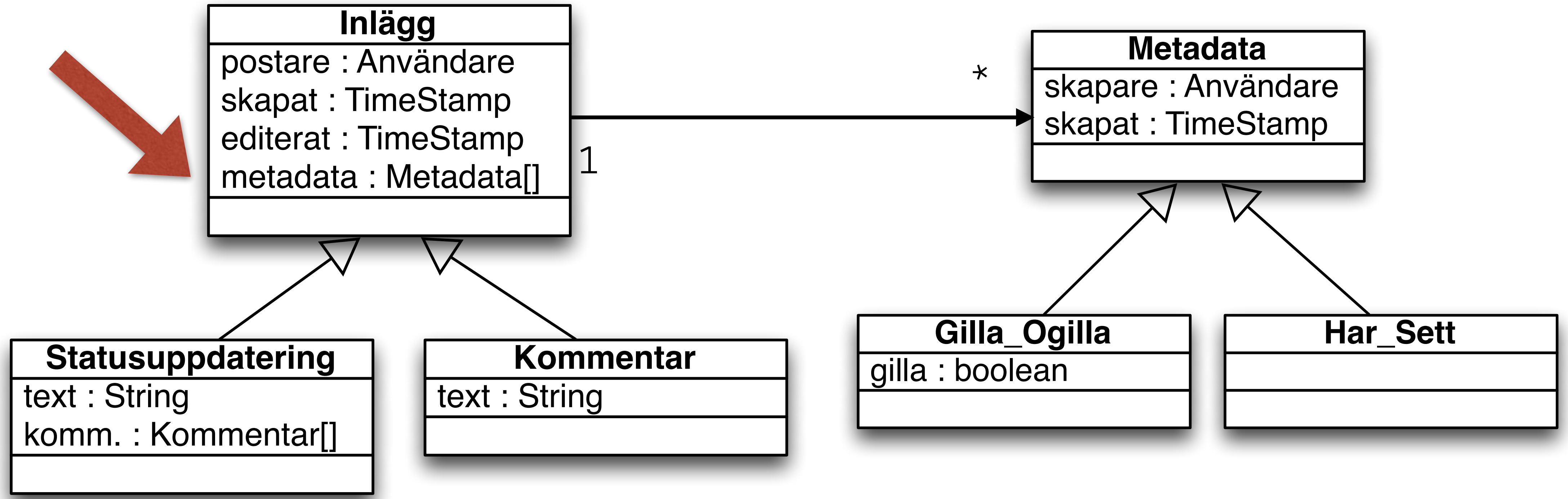
Where / how do we add support to like posts?

Attempt # 1: “liked_by"



If a user likes a post, add that person to a “liked_by" list

Attempt # 2: "Metadata"



Associate each post with arbitrary metadata

Differences Between Attempts # 1 and # 2

Attempt #1

- The logic for metadata lies in the Post class
- Must create a dislike_by, seen_by, etc. for each new property you want to add
- Simple and direct

Attempt #2

- The metadata logic is taken out of posts and can be changed "freely"
- Easily add a new type of metadata

- More complicated
  - Must follow some kind of pattern to "take advantage of new metadata"

Separate Metadata From Posts

```
class List_Dislike extends Metadata {
    String decorate(String s) {
        // ...
    }
}
```

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

Actual metadata unknown

Dynamic binding

Arv (eng. inheritance)

- Let A and B be classes; A has the variables X and Y, as well as the methods M and N
- If B inherits from A, B also receives X and Y and M and N
- Inheritance captures generalisation — specialisation (superclass—subclass)
  - Avoid code repetition
  - Captures relationship between classes in the code
  - In statically typed languages such as Java (C ++, C #, etc.): important for polymorphism!
- Metadata as superclass, `Like_Dislike` and `Seen` are subclasser of `Metadata`
  - In an array of `Metadata` you can mix `Like_Dislike` and `Seen`
- Metadata should be an **abstract** class that cannot be instantiated

Partial class hierarchy for relationships between people

```java
abstract class Relation {
    Person a;
    Person b;
}

class Friendship extends Relation {
    boolean a_close;
    boolean b_close;
}

class Family extends Relation {}
```

Inheritance in Javas API

- java.lang.Object
  - java.util.AbstractCollection<E>
    - java.util.AbstractList<E>
      - java.util.AbstractSequentialList<E>
        - java.util.LinkedList<E>

$\}$ *Depth 5*

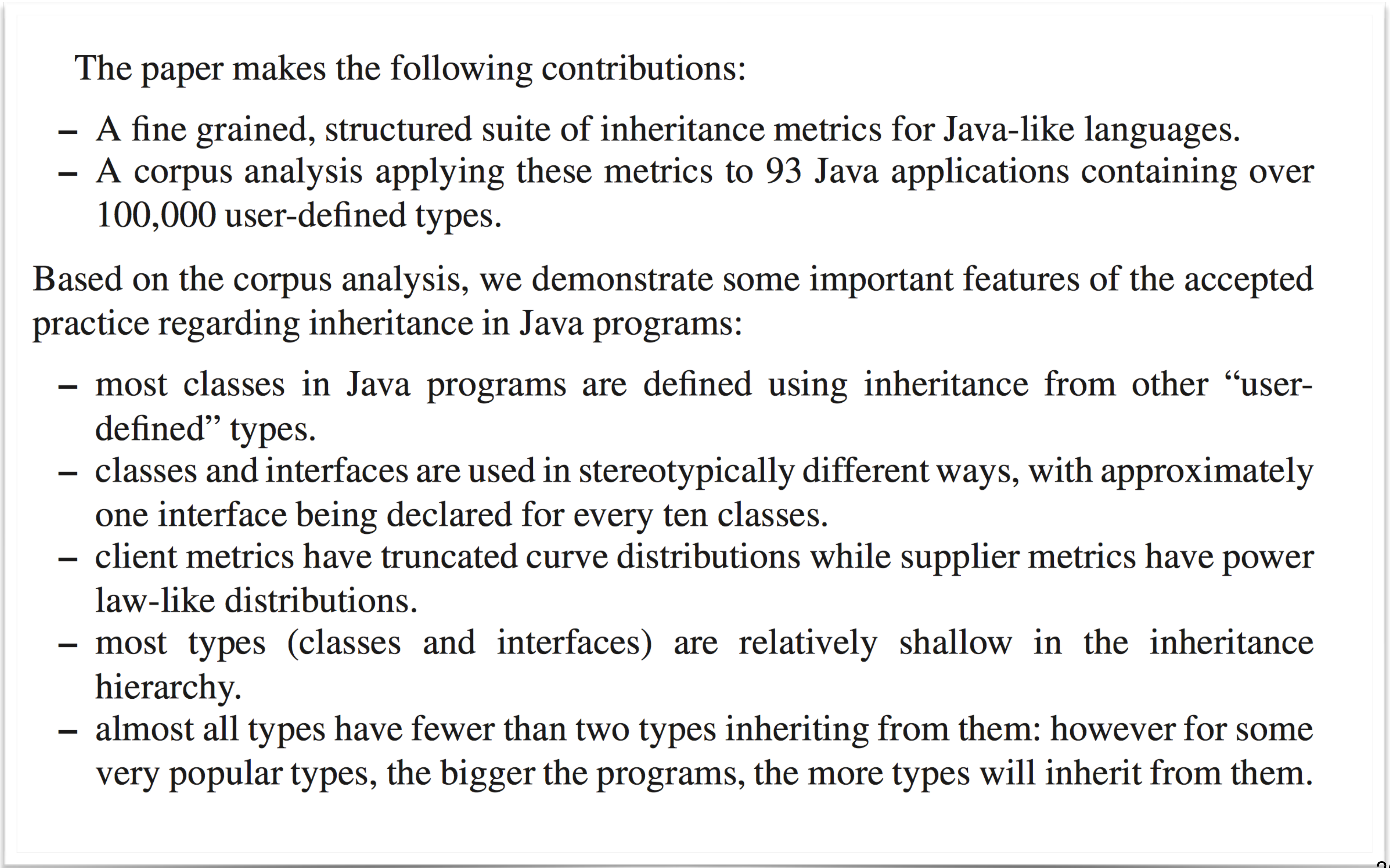
- java.lang.Object
  - java.awt.Component
    - java.awt.Container
      - java.awt.Window
        - javax.swing.JWindow

$\}$ *Depth 5*

- In stable libraries, inheritance often fits well
- The amount of inheritance in "regular programs" is significantly less

How do Java Programmers Use Inheritance?

[Tempero et al. 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.

Inheritance and Constructors

The purpose of a constructor is to initiate objects

One should not be able to observe an object in an "inconsistent state"

Ex.: the class `Point2D` class has two private variables x and y
`Point2D` has a constructor that takes an x value and a y value - forces all points to have valid values on all axes (consistent state)

A constructor for a subclass initiates "its part" of the object

Ex.: the class `Point3D` inherits from `Point2D`, and adds a z-axis
`Point3D` has a constructor that takes x, y, z
`Point3D` can't write to x and y - they're private!
`Point3D` must delegate to `Point2D’`s constructor to initialise x and y

Inheritance and Constructors (Example)

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

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

Inheritance and Constructors (Example)

```java
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 mutualCloseFriendship(final Person a, final Person b) {
        return new Friendship(a, b, true, true);
    }
}
```

Abstract Class

```
abstract class Relation { ... }

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

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

OO in 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 in C

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

Value-semantics

```
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

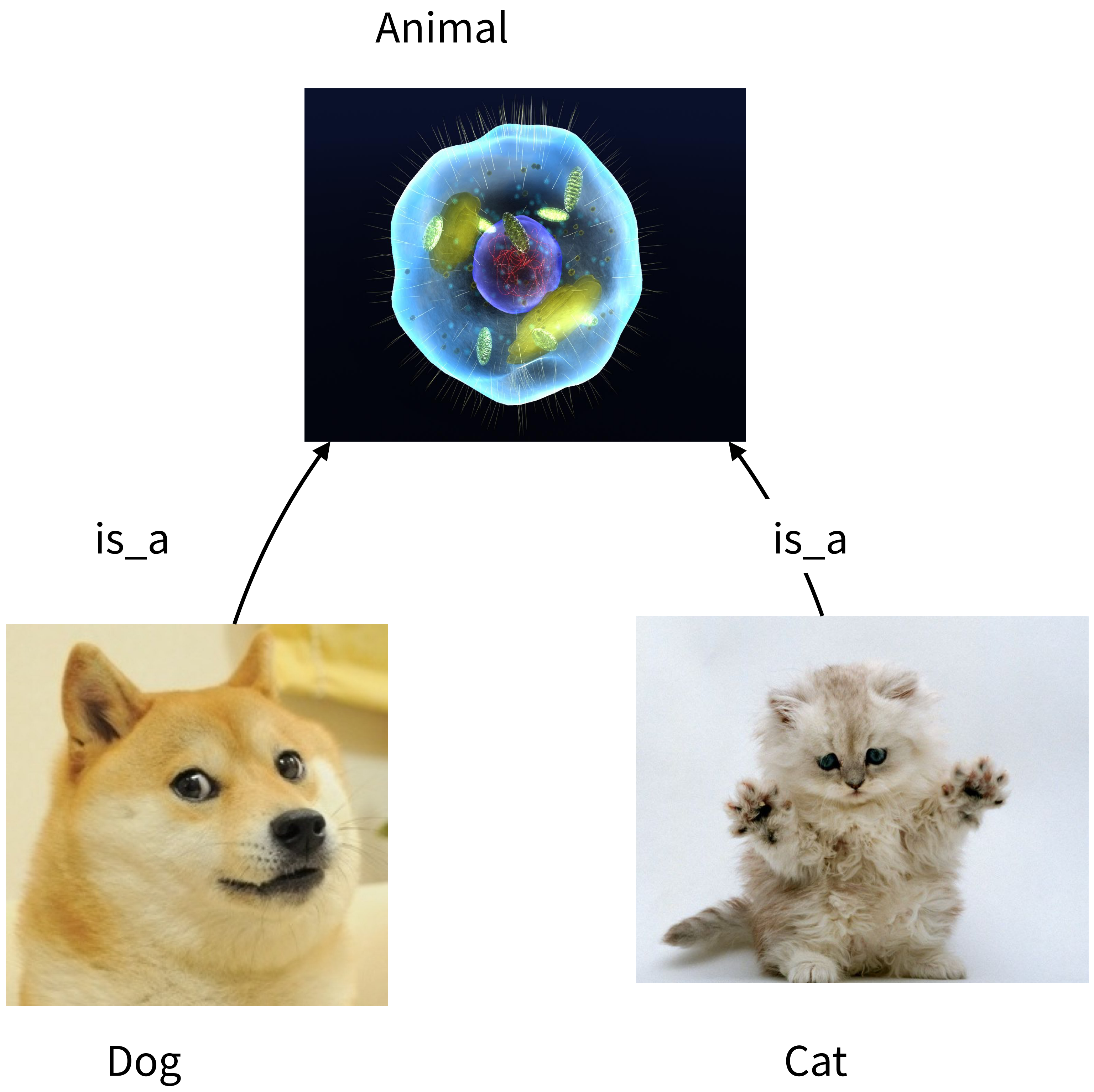
```java
class Friendship extends Relation {
    boolean a_close;
    boolean b_close;
    Friendship(final Person a,
               final Person b,
               final boolean a_close,
               final boolean b_close) {
        super(a, b);
        this.a_close = a_close;
        this.b_close =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

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

```c
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);
}
```

You can program object oriented in C, but you get nothing for free

Inheritance and specialisation # 2 "overriding"


Animal
is_a
is_a
Dog
Cat

Overriding (Method Specialisation)

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

Overriding (Method Specialisation)

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

class Cat extends Animal {
    void makeSound() { System.out.println("Meeow!"); }
}
```

Overriding (Method Specialisation)

```
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 Abstract Methods

```
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 Animal {
    public static void main(String[] args) {
    Animal[] animals = { new Cat(), new Dog(), new Cat() };
    for (Animal a : animals)
        a.makeSound();
    }
}
```

Overriding – Example 2 (see “super call”)

```
abstract class Animal {
  private 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; }
    }
  }
}
```

Call the `setName` method in the nearest superclass

A Better Dog

```
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);
    }
}
```

static binding!

Static binding in Java occurs during compile time while dynamic binding occurs during runtime.

…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);
    }
}
```

static bindning

Examples of Encapsulation and Inheritance

```
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!”;
   }
}
```

Forces subclasses to provide one

`name` is not private!

Overriding vs. Overloading

- Overriding allows a subclass B to redefine a method m, inherited from superclass A

  Shadowing: m in A does not become visible in instances of B

  Super calls allow B's methods to call A's m

- Overloading allows us to define several methods with the same name, but different parameter types

  Ex.:
  ```
  checkPrice(int p) // p is the price in pennies
  checkPrice(float p) // p is the price in SEK
  ```

- Overloading is a common source of error and should *not* be overused

  Hard to know that there are several methods of the same name

  It can be difficult to figure out which method is used

Overriding vs. Overloading

- Overloading is a common source of error

  Overloaded methods are bonded using static binding while overridden methods are bonded using dynamic binding at runtime

```
public class StaticBindingTest {
    public static void main(String args[]) {
        Collection c = new HashSet();
        StaticBindingTest et = new StaticBindingTest();
        et.sort(c);
    }
    //overloaded method takes Collection argument
    public Collection sort(Collection c) {
        System.out.println("Inside Collection sort method");
        return c;
    }
    //another overloaded method which takes HashSet argument which is sub class
    public Collection sort(HashSet hs) {
        System.out.println("Inside HashSet sort method");
        return hs;
    }
}
```

Inheritance and Subtypes

Root Class Object

- All Java classes that do not have an explicit superclass inherit the class `Object`

  This means that all inheritance hierarchies form a tree whose root is `Object`

- Each class in a Java program creates a new type

  `Relation`, `Dog`, `Animal`, etc.

- In Java, subclassing entails subtyping

  `Dog` **`extends`** `Animal` means that `Dog` is a subtype of `Animal`

- Since class hierarchies are rooted in `Object`, everything is a subtype of `Object`

  This means that `Object x = …;` is legal, not matter what … is

Subtyping

- Liskovs principle of substitution (LSP):

  If T is a subtype of T’ then we can replace all occurrences of T' objects in a program with T type objects

  This is true, in the sense that all the methods in T 'are also in T

  However - no guarantees that the methods actually behave appropriately

- Java has two kinds of types:

  object types are created by classes or interface (we will see them soon!)

  primitive types are inherited from C (ie, int, float, etc.)

- Primitive types have no subtype

Type test and type conversion

- In Java, there is metadata about objects at runtime

  expr **`instanceof`** *Type* ⇒ true/false

  Ex.: `connections[42]` **`instanceof`** `Friendship`

- Conversion from one type to another (as in C)

  (*Type*) *expr* ⇒ crashes the program if expr does not have the type Type

  Ex. `(Friendship) connections[42]`

  Pattern:

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

Problem with Inheritance: The Fragile Base Class Problem

- A superclass does not know its child classes and therefore cannot predict the effects of change

  At **best**: the child class does not compile

  At **worst**: everything compiles, BUT the meaning of the program changed

- In many OO program languages, inheritance tends to provide privileged access to the superclass methods and fields

  Extra strong dependencies from subclass to superclass (who do not know them!)

Inheritance and subtype polymorphism

- Subtype polymorphism in Java

  (Liskov again)

  If B inherits from A, a B object can be used wherever an A object is expected

  until now: B subtype A ⇒ B subclass A

```
Shape x = new Square(); // OK if Square inherits from Shape
```

- Java's static typing and the lack of multiple implementation inheritance limits the power of subtype polymorphism

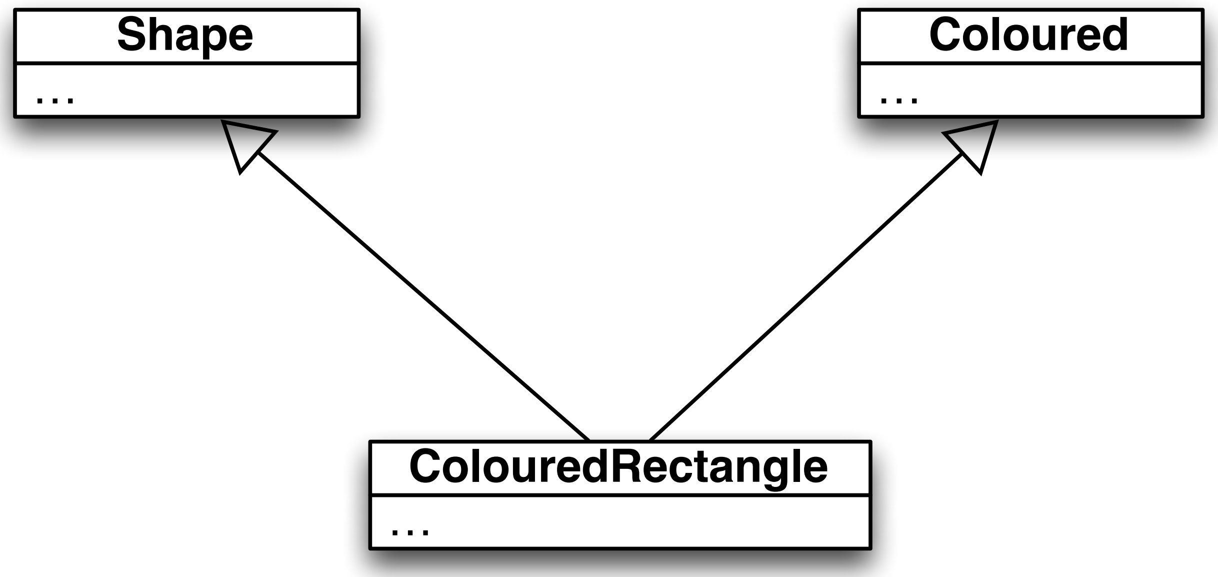
  If I have a B that inherits from A and I want to be able to do:

```
C c = new B(); // Ex. C = ColouredShape and B = Square
```

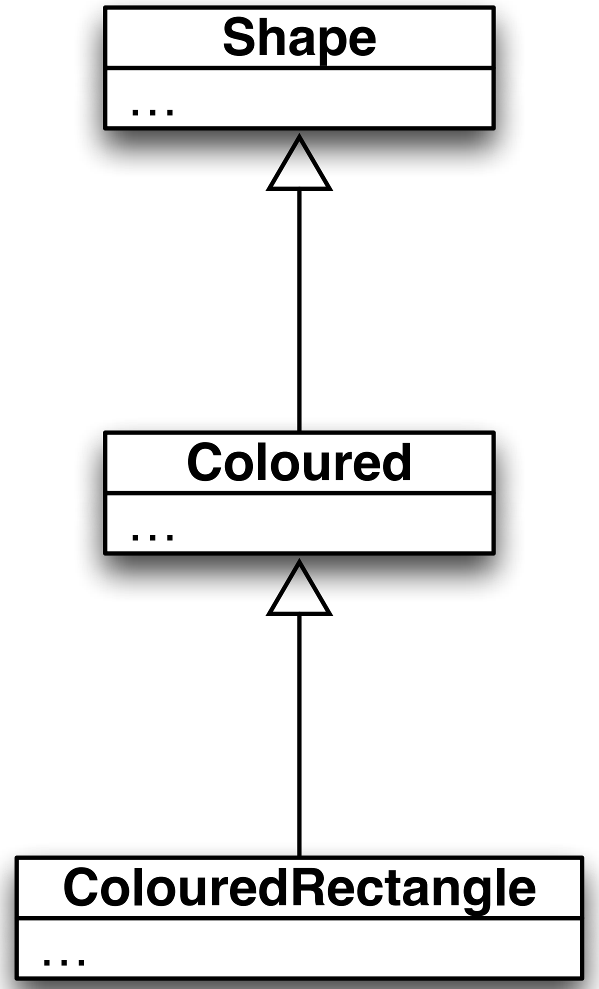
  I must do so that B also inherits C, but B inherits A already!

Java har enkelt arv [single inheritance]

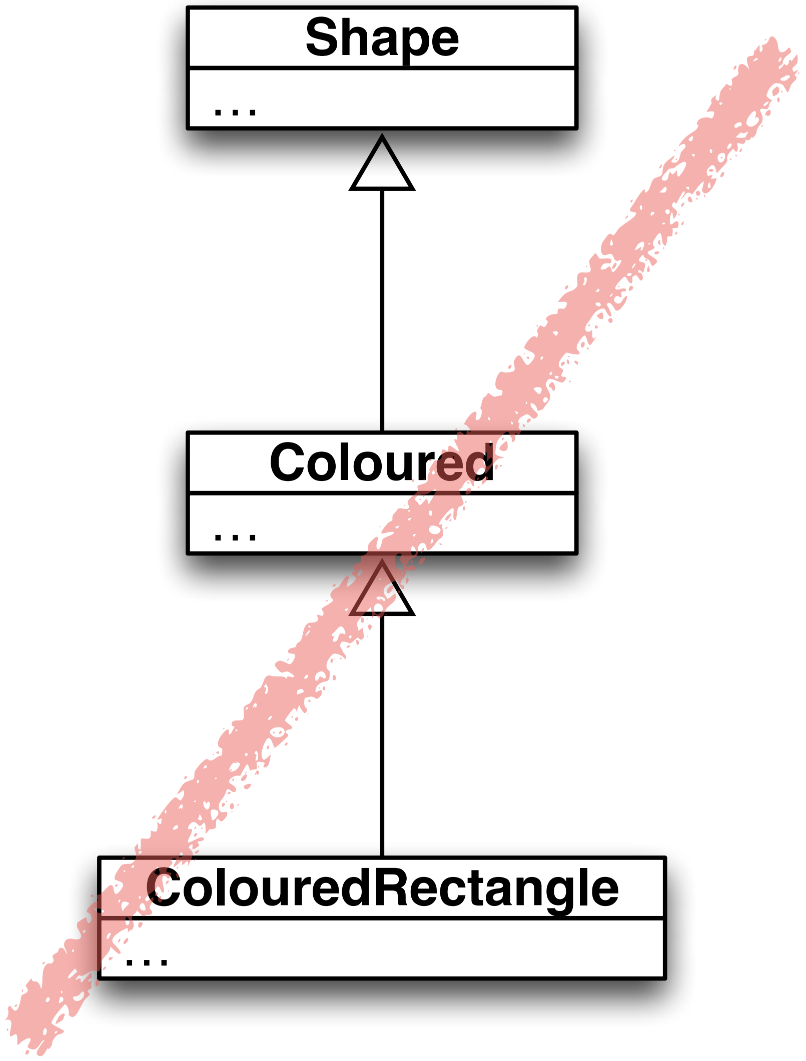
- Multipelt inheritance is NOT possible in Java, ie this does not work:

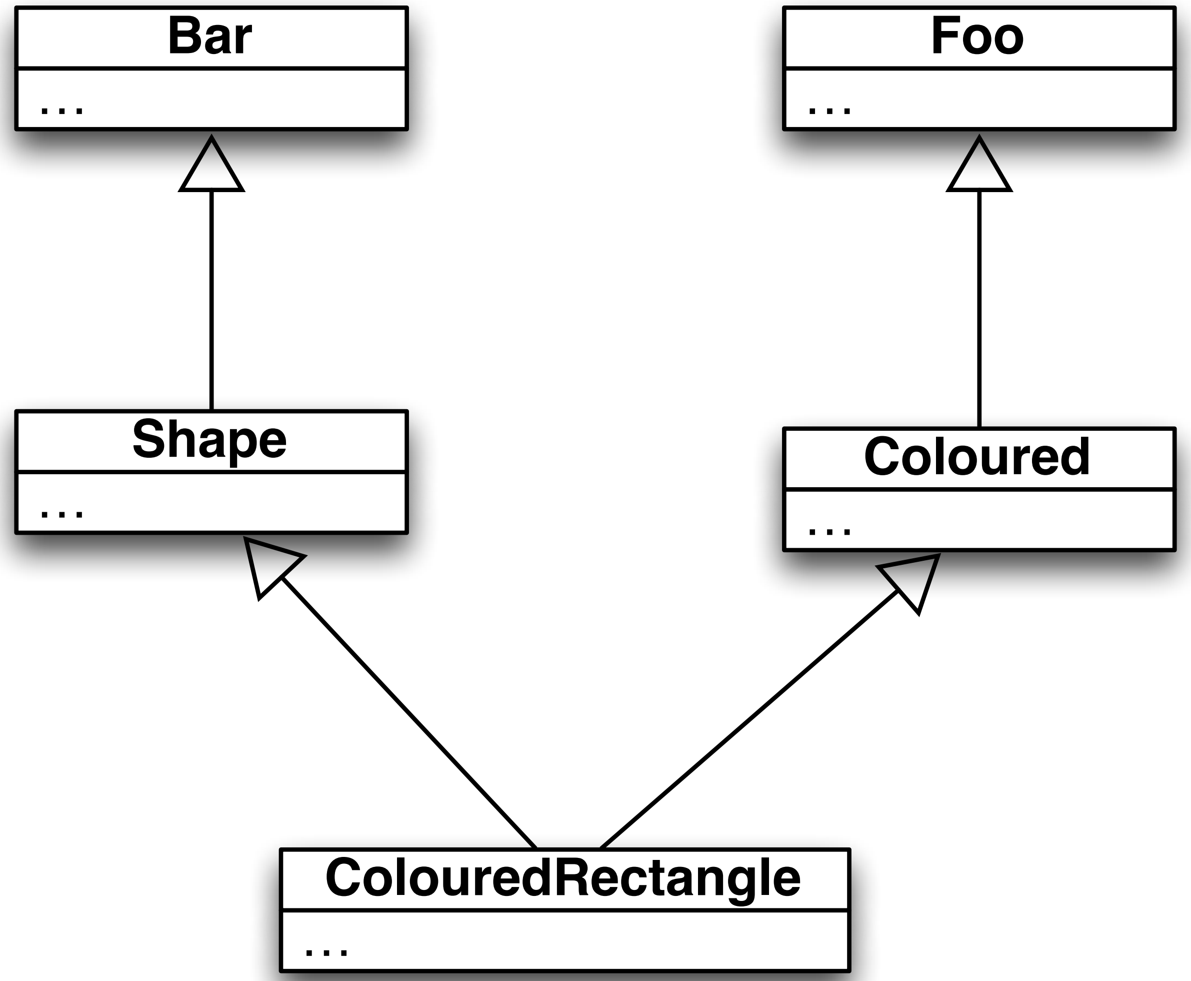


Solution #1: **Linearization of the inheritance hierarchy**



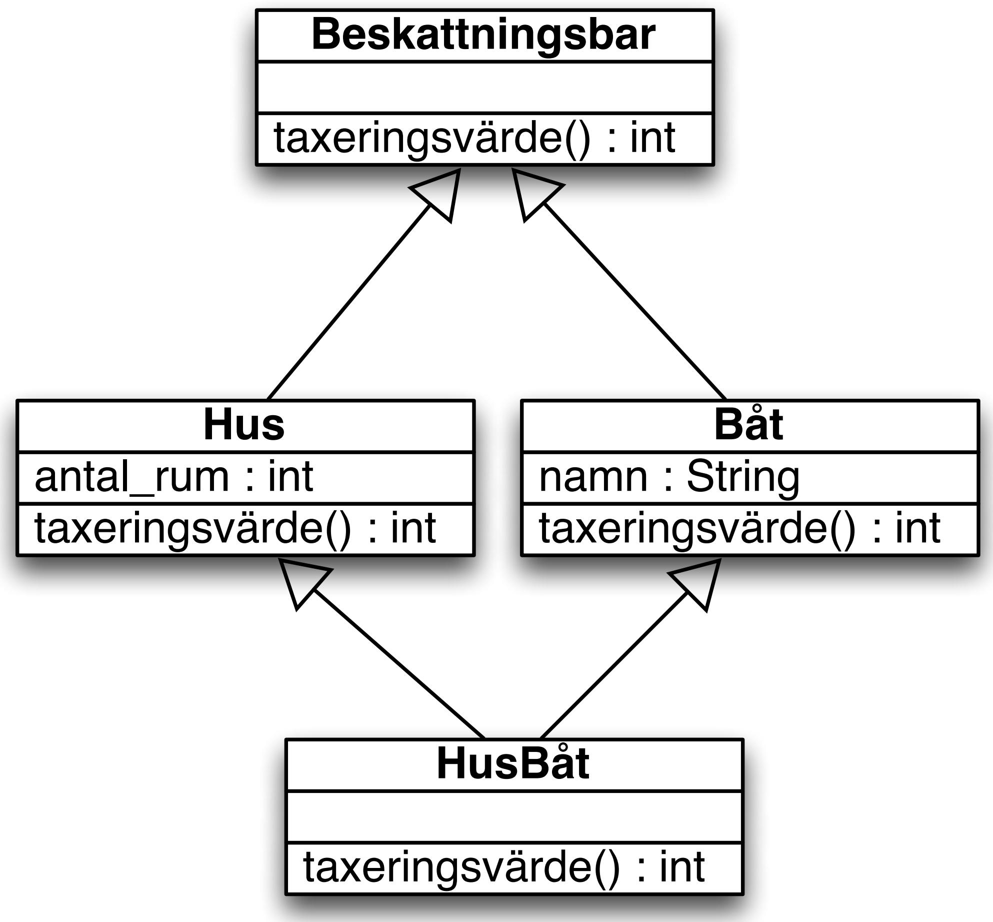
Solution #2: **Allow multiple superclasses**

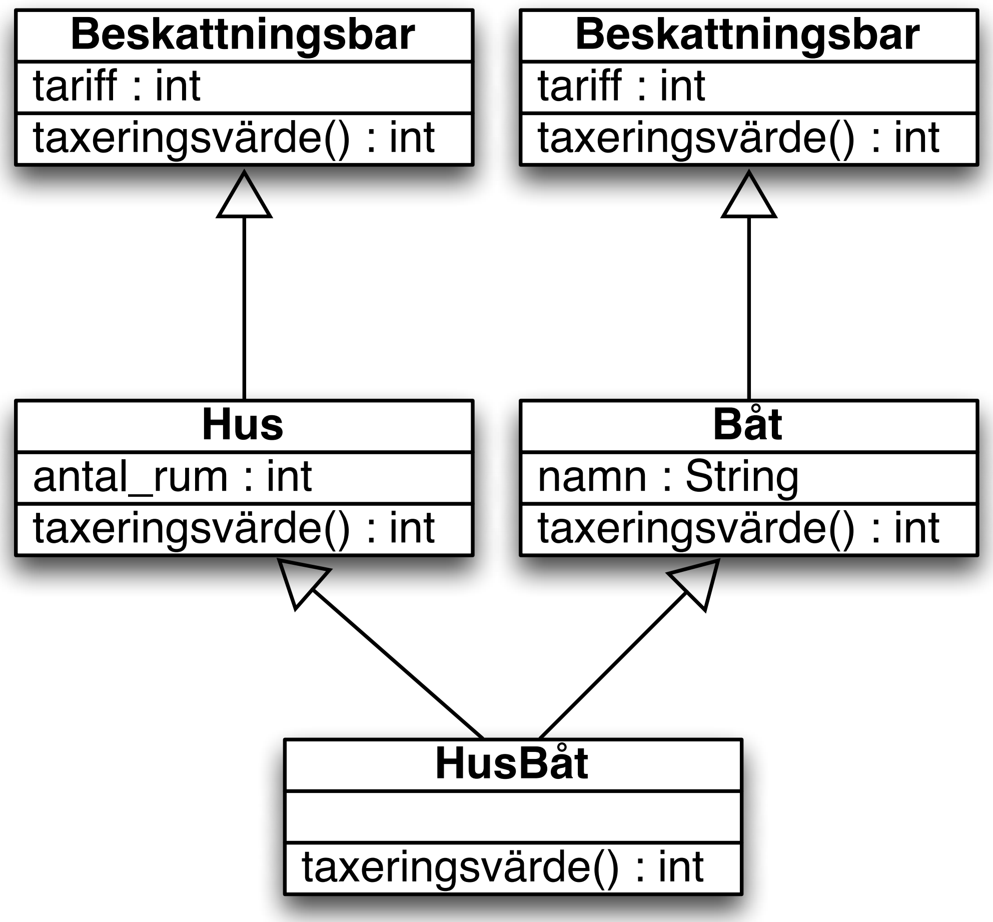




Not usually possible …

Multiple inheritance is problematic





How many tariffs?

Multiple inheritance has no given solution

- A tariff is right sometimes

  `Båt` and `Hus` and `HusBåt` have the same permissible values on the tariff

  Problem: incompatible behavior

- Two tariffs are right sometimes

  `Båt`-part and `Hus`-part of the `Husbåt` each have their own tariff

  Problem: what does `this.tariff` or `taxeringsvärde()` mean in the Class `Husbåt`?

- Possible solutions

  Allow both (C ++)

  Require the programmer to resolve all conflicts in the subclass (Eiffel, Java 8, etc.)

  Avoid multiple inheritance (Java <8, C #, Ruby, etc.)

The Interface Concept

Java's prescribed solution: Interface

- Same strength as multiple inheritance for subtype polymorphism
- None of the disadvantages of multiple inheritance
- Exempel:

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

- (By multiple inheritance above is meant really multiple implementation inheritance)

Prior to Java 8, Interface could be described as ≈ "fully abstract classes without permission"

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

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

Interface

- A specification of a protocol, i.e. a number of method signatures

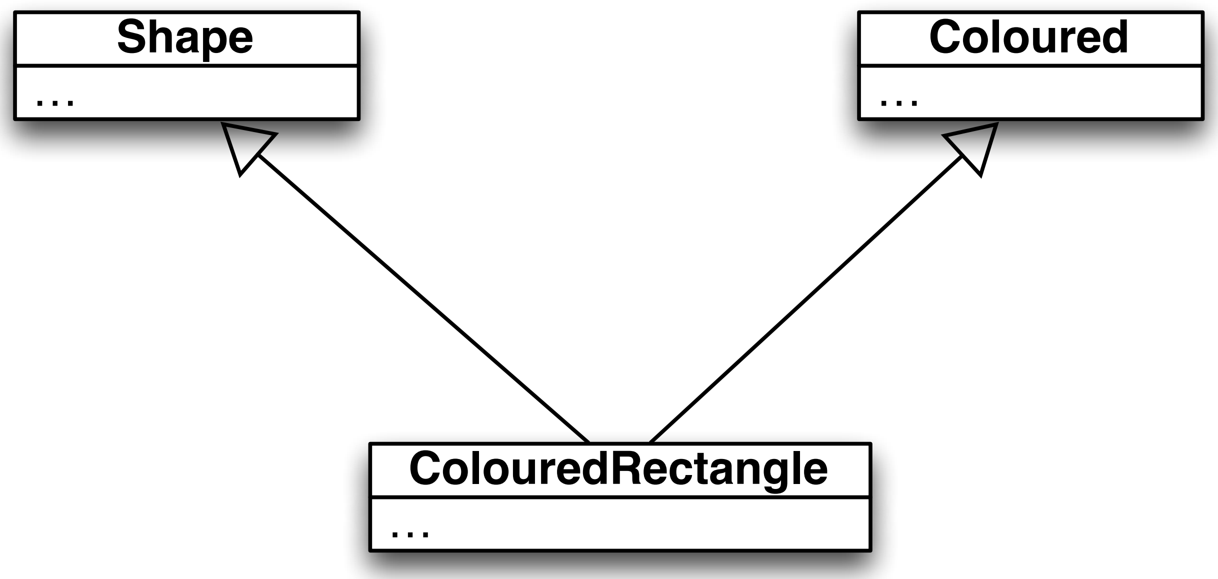
  Can also be seen as a contract

- Example:

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

Java allows multiple inheritance for interface

- If `Shape`, `Coloured` och `ColouredRectangle` are interfaces, this is allowed:



Multiple inheritance between interface

- Interfaces can inherit from each other
- There is no "root interface" in the same way that Object is a root class
- Multiple inheritance (even from the same class) is unproblematic and therefore permissible

  Except cycles!
- Overriding does not exist because implementations are not in interface

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

OK!

Java 8 and default-methods

- Starting with Java 8, an interface can also contain implemented methods

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

- A class that implements two interfaces with conflicting default methods must explicitly provide its own

- Syntax to control binding on super calls:

```
Type.super.methodName(arg);
```

Link between classes and interface

- A class can *implement* an interface

  A nominal relationship in the form of an **`implements`** declaration in the class header

```
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());
    }
    ...
```

Link between classes and interfaces

- Why does the following code not work?

```
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;
    }
    ...
```

How can we know that these fields are in a Coloured?

Link between classes and interfaces

- Partial implementation of an interface

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

public class Circle implements Coloured {
    private Point center;
    private int radius;
    public Colour getColour() { ... }
    public void setColour(Colour c) { ... }
}
```

// missing!

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);      // Missing!
}

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
```

Abstract classes

- Classes that help build heritage hierarchies and are not intended to be instantiated

  `new A();` does not work if `A` is an abstract class

- A class `K` can be declared as abstract with the keyword **`abstract`**:

  `public abstract class K …`

- A class `K` must be declared abstract if

  `K` partially implements an interface, or

  `K` has a method `M` that is declared abstract, or

  `K` inherits an abstract method `M` without overriding/ specialising it

Interface and subtype polymorphism

- Making Coloured an interface solves our previous problem

  However, each class must implement the functionality itself - it is not inherited!

```java
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?
```

Parametric Polymorphism in 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 Base...List {
    private class Link {
        Link next;
        BaseballPlayer value;
    }
    private Link first;
}
```

```
public class Boolea
    private class Link {
        Link next;
        boolean 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 BaseballP
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
```

```
List list1 = new List(); // for int
List list2 = new List(); // for Baseball...
List list3 = new List(); // for boolean

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
```

Must cast values to use them.

```
List list1 = new List();

int v1 = 7;
int v2 = 17;
boolean v3 = false;

list1.add(v1);
list1.add(v2);
list1.add(v3);

int v4 = (int) list1.get(0);
int v5 = (int) list1.get(1);
int v6 = (int) list1.get(2); // runtime error!!
```

Parametrisk polymorfism

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

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

```
List<Person>

List<String>

List<Object>
```

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

Parametriskt polymorfa typer

```java
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
BaseballPl
boolean v3

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

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

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 objektmotsvarigheter (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;
}
```

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

Parameter

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…

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

- …till…

```
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”

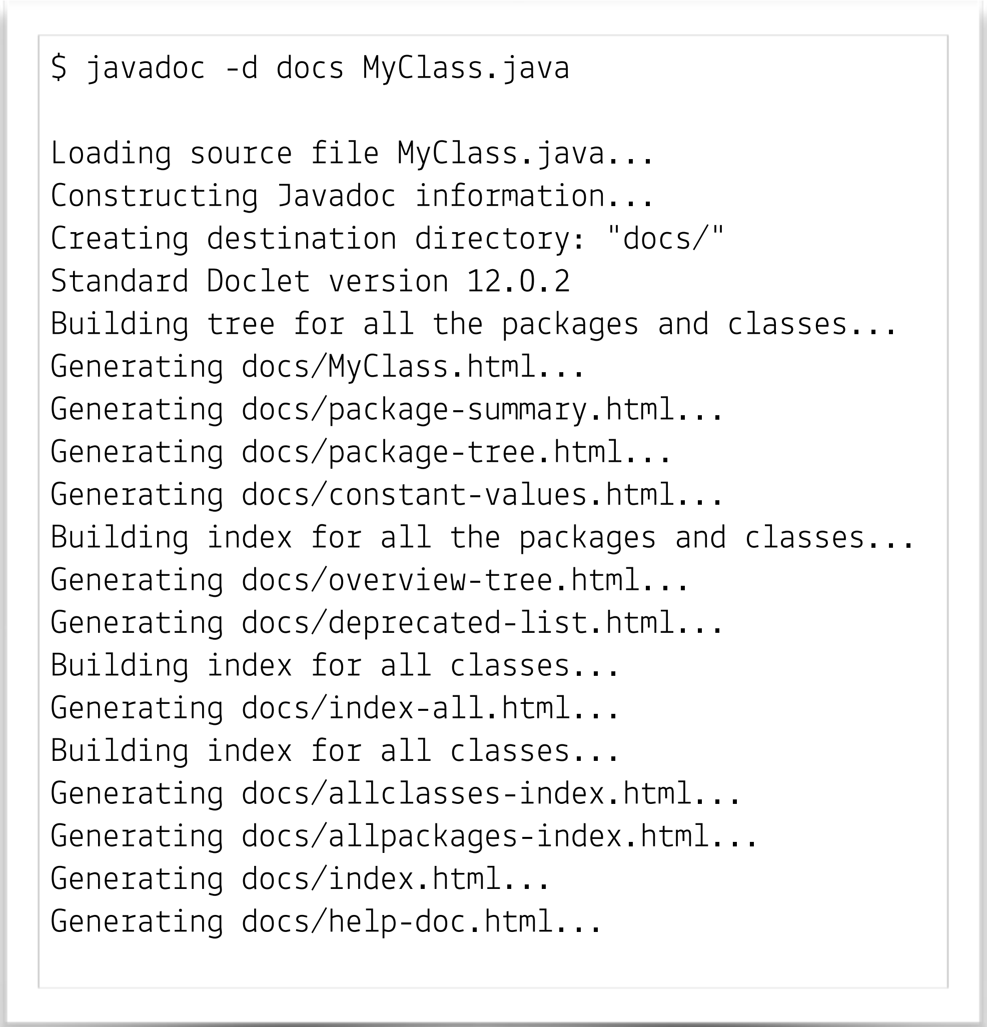
```
public class StringList extends List<String> { }

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

javadoc och doxygen

- JavaDoc och Doxygen är verktyg för att halvautomatiskt generera dokumentation från kod.
- Speciella kommentarer i koden används för att ge kompletterande information.
- Hela dokumentationen av JavaAPI:et är skapat med hjälp av JavaDoc.
- Doxygen fungerar med flera programspråk än Java men har en komplicerad konfigurationsfil.

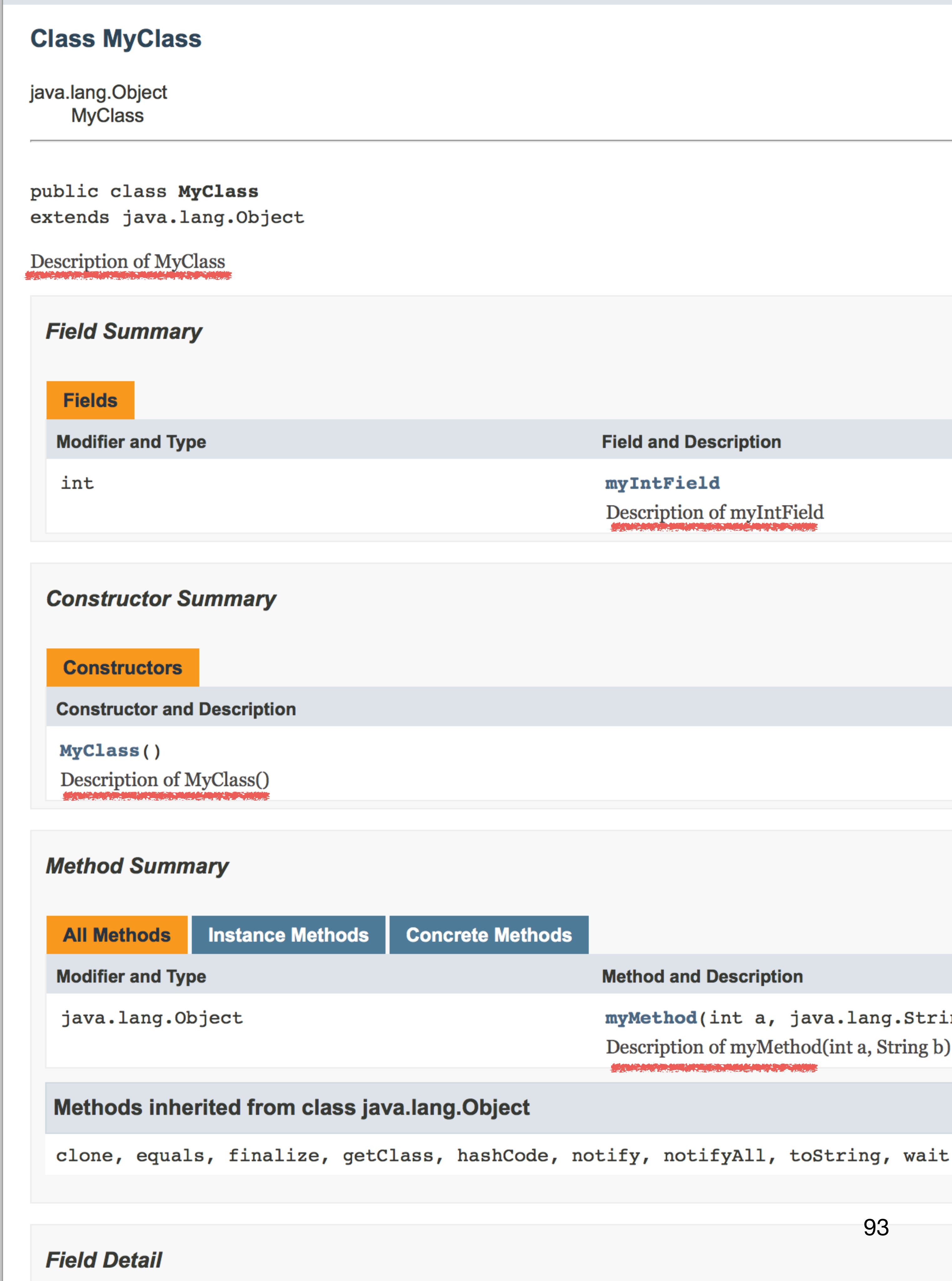
MyClass.java

```java
/** Description of MyClass
 *
 * @author John Doe
 * @author Jane Doe
 * @version 6.0z Build 9000 Jan 3, 1970.
 */
public class MyClass
{
    class myException extends Exception {}

    /** 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 = null;
        // Blah Blah Blah...
        return c;
  }
}
```



```
$ javadoc -d docs MyClass.java

Loading source file MyClass.java...
Constructing Javadoc information...
Creating destination directory: "docs/"
Standard Doclet version 12.0.2
Building tree for all the packages and classes...
Generating docs/MyClass.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/deprecated-list.html...
Building index for all classes...
Generating docs/index-all.html...
Building index for all classes...
Generating docs/allclasses-index.html...
Generating docs/allpackages-index.html...
Generating docs/index.html...
Generating docs/help-doc.html...
```

Användning av javadoc. HTML-kod skapas.



Class MyClass

java.lang.Object
    MyClass

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

Description of MyClass

Field Summary

Fields

Modifier and Type	Field and Description
`int`	`myIntField` Description of myIntField

Constructor Summary

Constructors

Constructor and Description
`MyClass()` Description of MyClass()

Method Summary

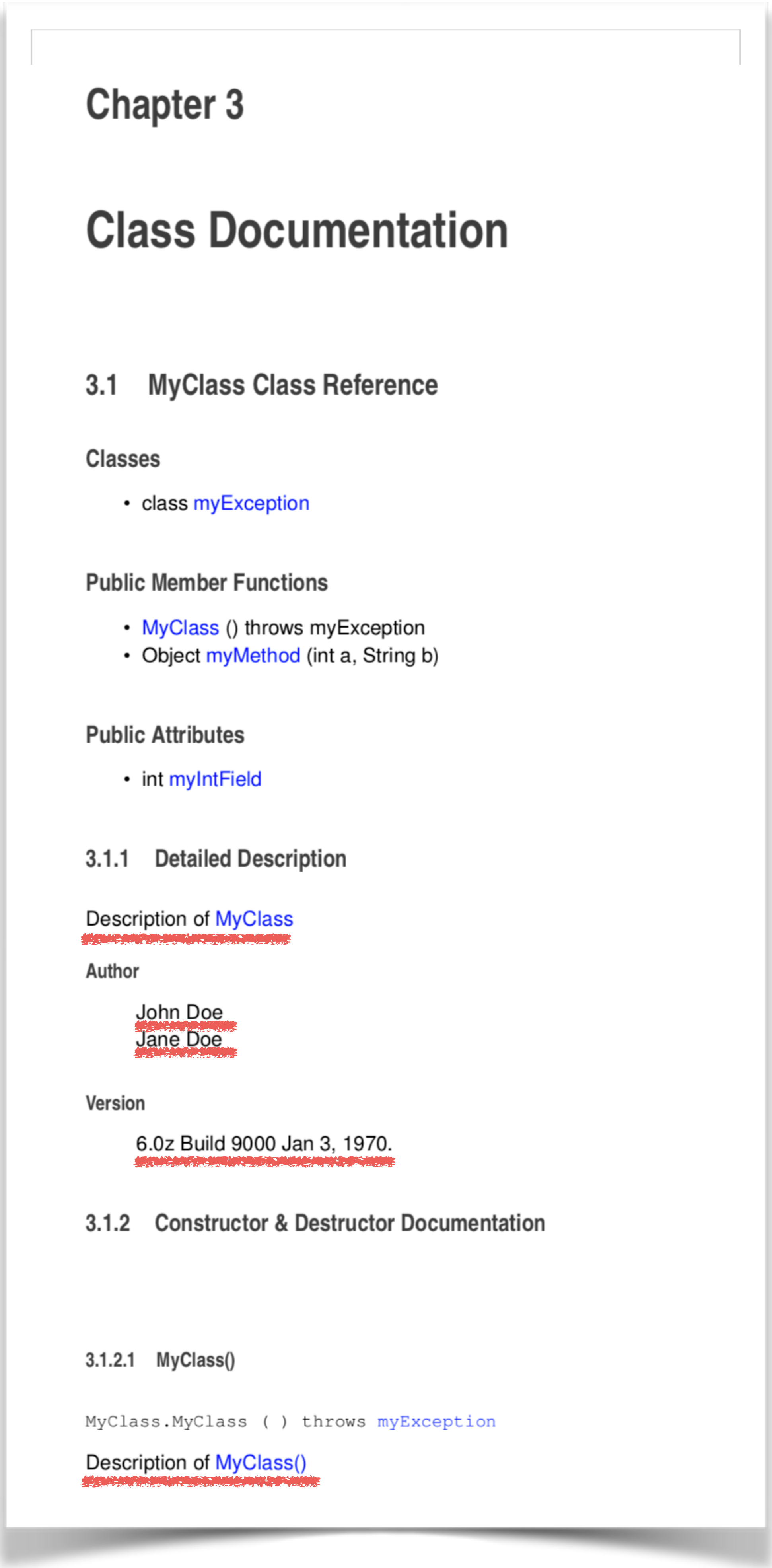
All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
`java.lang.Object`	`myMethod(int a, java.lang.Stri` 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



Chapter 3

Class Documentation

3.1 MyClass Class Reference

Classes

- class myException

Public Member Functions

- MyClass () throws myException
- Object myMethod (int a, String b)

Public Attributes

- int myIntField

3.1.1 Detailed Description

Description of MyClass

Author

John Doe
Jane Doe

Version

6.0z Build 9000 Jan 3, 1970.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 MyClass()

```
MyClass.MyClass ( ) throws myException
```

Description of MyClass()

```
doxygen -g MyClass.doxy
```

Skapar en fil med inställningar för doxygen

```
emacs MyClass.doxy
```

Ändra på INPUT-inställningen så att rätt fil(er) pekas ut

```
doxygen -s MyClass.doxy
```

Generera dokumentationen

Användning av doxygen.
Både HTML och LaTeX-kod skapas.