Arv, super-klasser & sub-klasser
(Generalisering & Specialisering)

RECAP

Klassedesign:
Kobling && Sammenhæng

Kode-duplikering

printWelcome():

```
...
System.out.println("You are " +
    currentRoom.getDescription());
System.out.print("Exits: ");
if (currentRoom.northExit != null) {
    System.out.print("north ");
}
if (currentRoom.eastExit != null) {
    System.out.print("east ");
}
if (currentRoom.southExit != null) {
    System.out.print("south ");
}
if (currentRoom.westExit != null) {
    System.out.print("west ");
}
System.out.println();
...
```

=

goRoom():

```
...
System.out.println("You are " +
    currentRoom.getDescription());
System.out.print("Exits: ");
if (currentRoom.northExit != null) {
    System.out.print("north ");
}
if (currentRoom.eastExit != null) {
    System.out.print("east ");
}
if (currentRoom.southExit != null) {
    System.out.print("south ");
}
if (currentRoom.westExit != null) {
    System.out.print("west ");
}
System.out.println();
...
```

refactoring

printLocationInfo():

```
private void printLocationInfo() {
    System.out.println("You are " +
        currentRoom.getDescription());
    System.out.print("Exits: ");
    if (currentRoom.northExit != null) {
        System.out.print("north ");
    }
    if (currentRoom.eastExit != null) {
        System.out.print("east ");
    }
    if (currentRoom.southExit != null) {
        System.out.print("south ");
    }
    if (currentRoom.westExit != null) {
        System.out.print("west ");
    }
    System.out.println();
}
```

printWelcome():

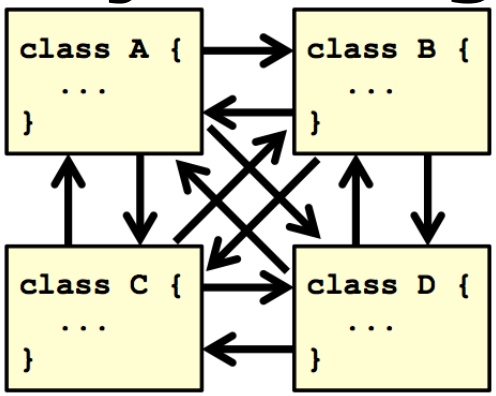
```
...
printLocationInfo();
...
```

goRoom():

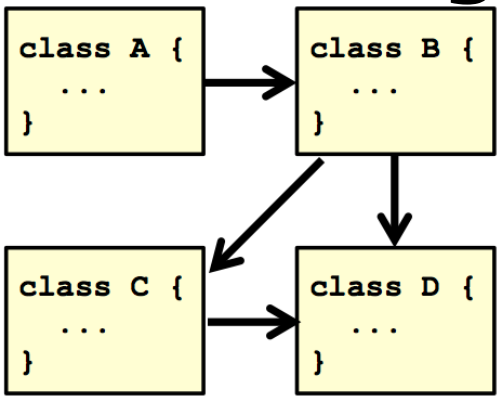
```
...
printLocationInfo();
...
```

Kobling

Høj kobling:



Lav kobling:



- **Mål for hvor tæt forbundne klasser er!**
- En vis grad af kobling er nødvendig
- Lav kobling er ønskværdig
- Problemer med høj kobling:
  - Ændringer og fejlretning er svære at lokalisere
  - Programmer bliver uoverskuelige
  - Alt involverer hurtigt mange forskellige klasser
  - Mange afhængigheder

Kobling

```
public Room northExit, eastExit, southExit, westExit;
```
Room

```
Room nextRoom = null;
if (direction.equals("north")) nextRoom = currentRoom.northExit;
if (direction.equals("east"))  nextRoom = currentRoom.eastExit;
if (direction.equals("south")) nextRoom = currentRoom.southExit;
if (direction.equals("west"))  nextRoom = currentRoom.westExit;
```
Game

☹

Game using Room's fields

refactoring

- Tilføj metode **`getExit()`** til **`Room`** klassen:

```
private Room northExit, eastExit, southExit, westExit;

Room getExit(String direction) {
    if (direction.equals("north")) return northExit;
    if (direction.equals("east")) return eastExit;
    if (direction.equals("south")) return southExit;
    if (direction.equals("west")) return westExit;
    return null;
}
```
Room

☺

Room's own (private) fields

```
Room nextRoom = currentRoom.getExit(direction);
```
Game

Ansvars-drevet design

- **Hver klasse har et ansvar**
- Ansvar kan handle om:
  - at vide ting (felter)
  - at gøre ting (metoder)

- ***"Enhver klasse bør være ansvarlig for håndtering af sine egne data"***

- **Fx:** I eksemplet havde `Room` ansvar for at kende sine udgange og bør derfor også have ansvaret for fx at liste dem

Sammenhæng (cohesion)

- ***Høj sammenhæng for klasser*** betyder at hver klasse har ***et velafgrænset og sammenhængende ansvarsområde***
  - En klasse bør svare til netop én type entitet (= ét fænomen)
- ***Høj sammenhæng for metoder*** betyder at hver metode gør netop én ting
- ***Konsekvenser af høj sammenhæng:***
  - Øget læselighed
  - Bedre mulighed for kode-genbrug
  - Det metoden gør burde kunne afspejles i navnet
  - Højere design-stabilitet

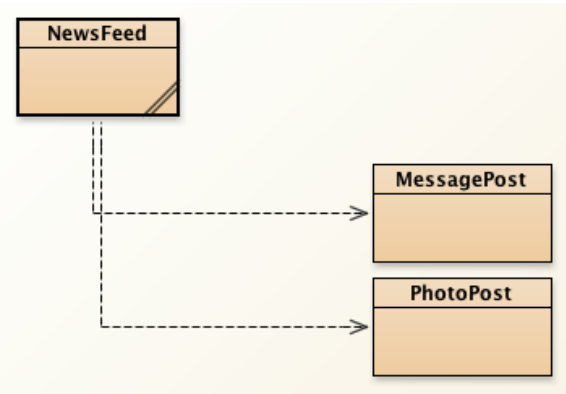
Arv

(Inheritance)

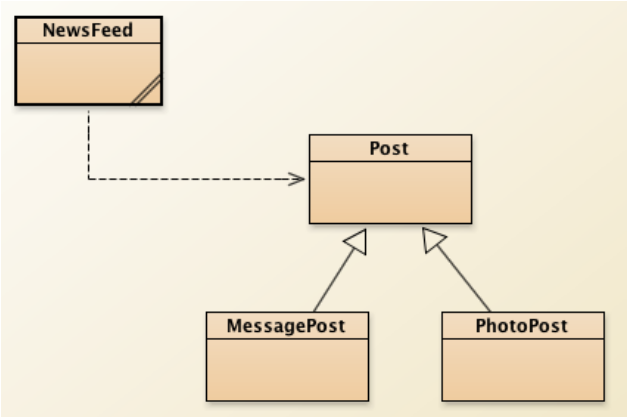
A G E N D A

- Objekt-orienteret programmering
- Arv (Nedarvning)
- Klassehierakier
- Substitutionsprincippet og tildeling
- Casting (Dynamisk cast)

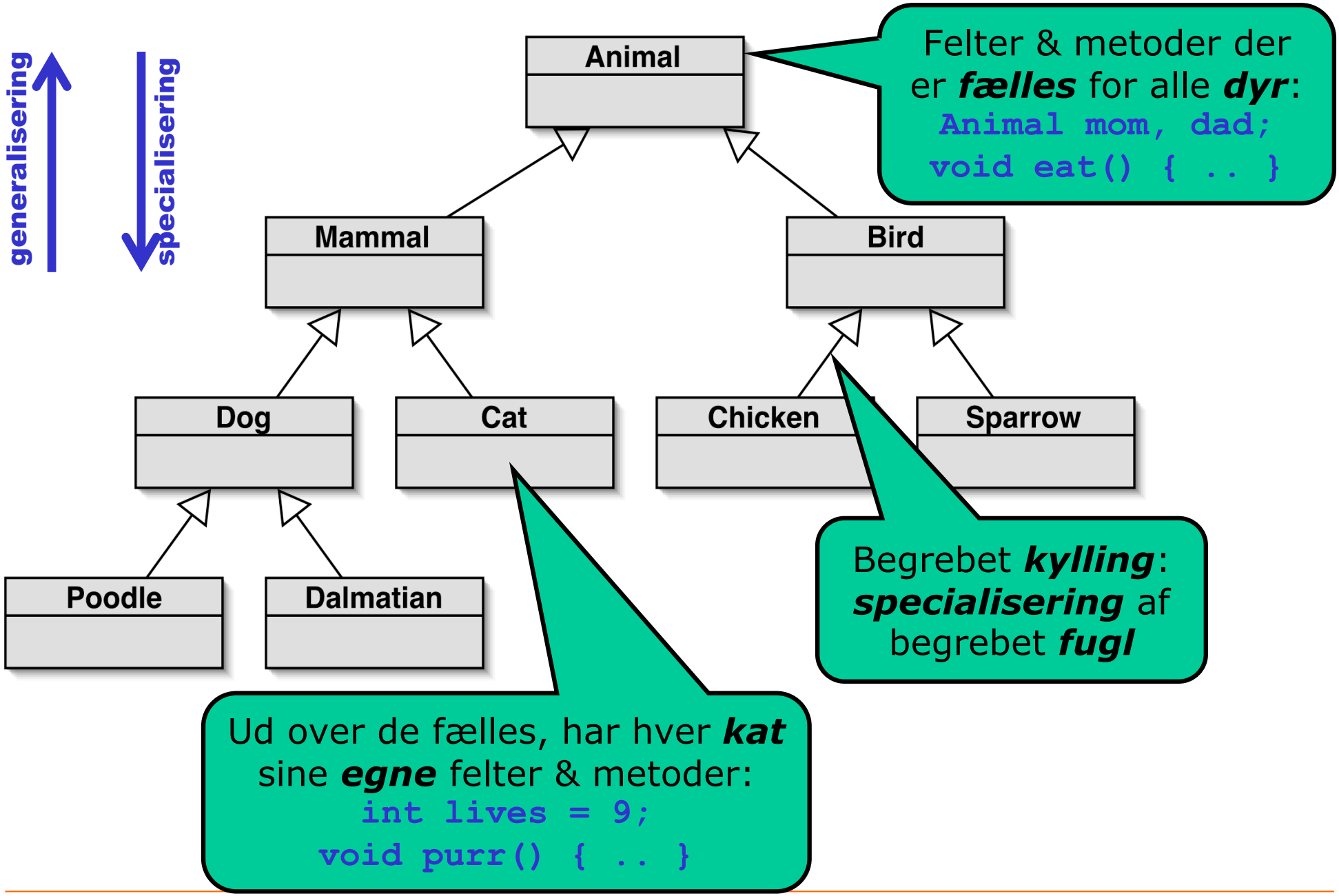
- Example: **`NewsFeed`** (à la Facebook)



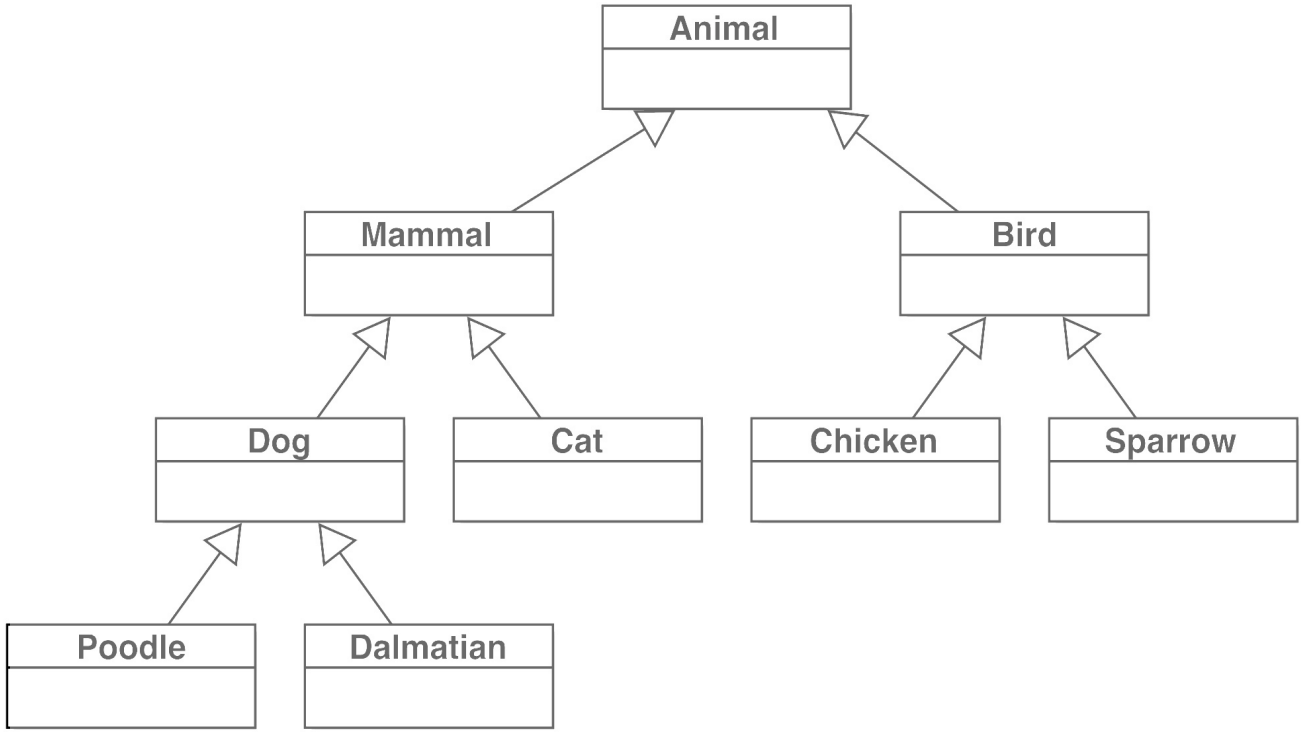
-vs-



Klassehierarkier er en velkendt ide



OPGAVE



- **Klasse-hierarki:**
  Lav et andet eksempel på et klasse-hierarki fra hverdagen med en højde på mindst fire.

Helt centrale OO begreber!

- **Arv: *sub-klasse* og *super-klasse***
  - ***Generalisering*** og ***Specialisering***
  - En klasse kan være en ***specialisering*** af en anden

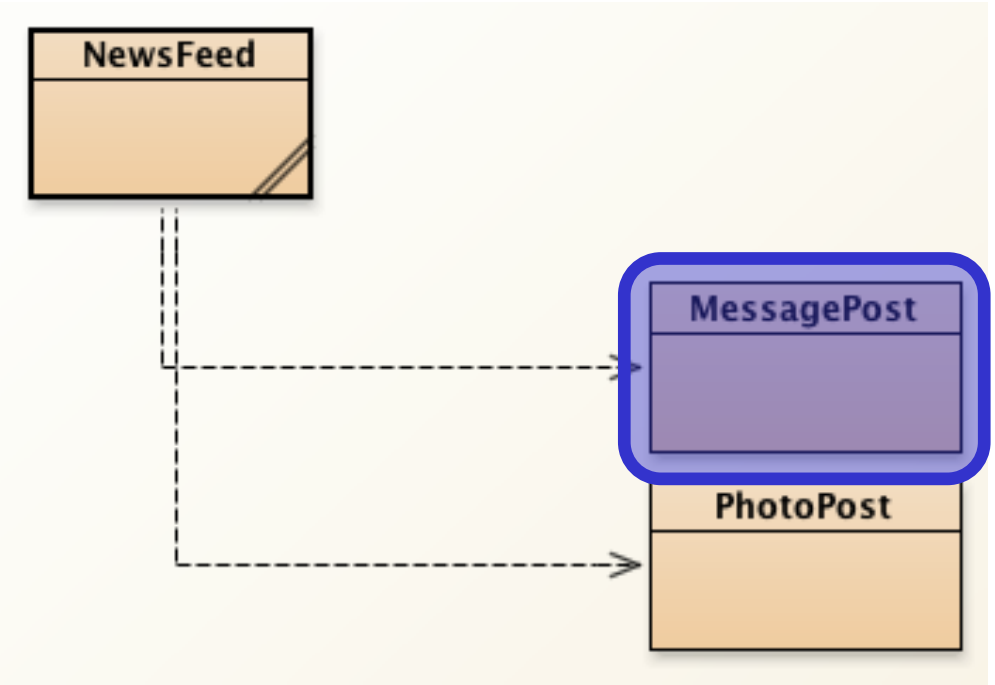
- **Subtyper:**
  - En subklasse er en subtype af sine superklasser

- **Substitutionsprincippet:**
  - Hvis `Student` er subklasse af `Person`,
    så kan en `Student` altid bruges hvor en `Person` behøves

(• **Virtual dispatching:** **[torsdag]**)
  - I kaldet `person.eat()` bestemmer det konkrete object (som person peger på) hvilken `eat()` metode kaldes!

NewsFeed (à la Facebook)

- **Example**: "`network-v1`":



MessagePost.java

```java
public class MessagePost {
  private String username;
  private String message;

  private long ts; // time-stamp
  private int likes;
  private ArrayList<String> comments;

  public MessagePost(String author,
                     String text) {

    username = author;
    message = text;

    ts = System.currentTimeMillis();
    likes = 0;
    comments = new ArrayList<String>();
  }

  ...
```

```java
...

public void like() {
  likes++;
}

public void unlike() {
  if (likes > 0) {
    likes--;
  }
}

public String getText() {
  return message;
}

public long getTimeStamp() {
  return ts;
}

...
```

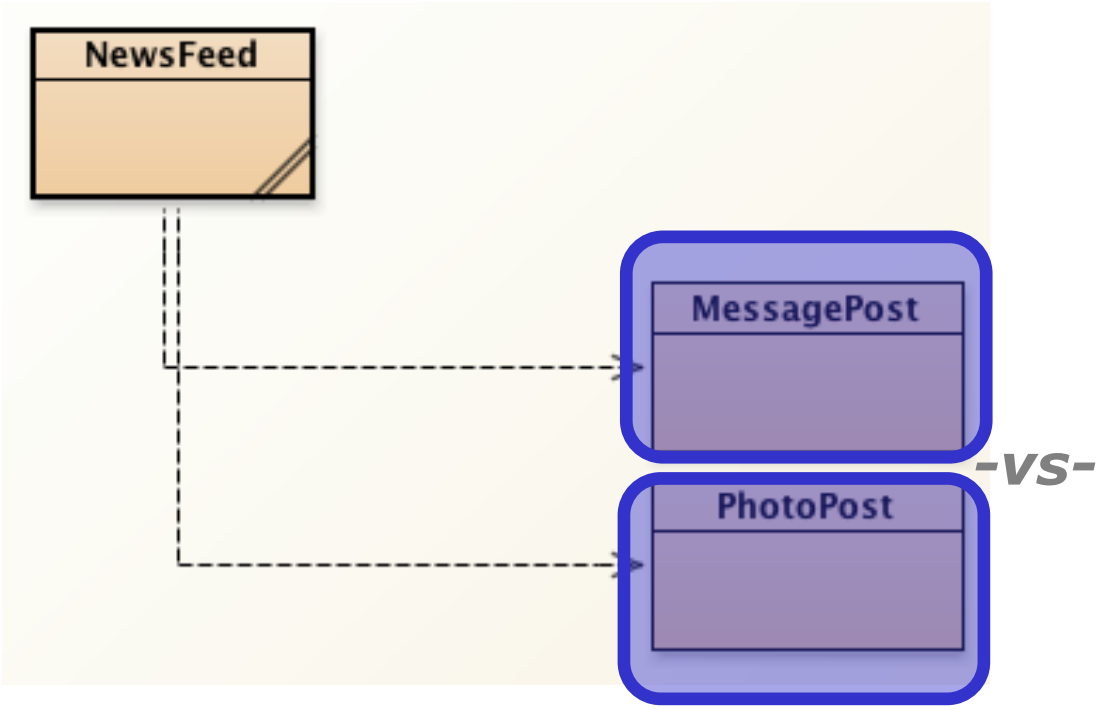
MessagePost.java

```
...

public void display() {
  System.out.println(username);
  System.out.println(message);


  System.out.print(timeString(ts));

  if (likes > 0) {
    System.out.println("  -  " +
      likes + " people like this.");
  } else {
    System.out.println();
  }

  if (comments.isEmpty()) {
    System.out.println(
        "   No comments.");
  } else {
    System.out.println(
        "   " + comments.size() +
        " comment(s).");
  }
}

...
```

```
...

 private String timeString(long time) {
   long now = System.currentTimeMillis();

   // time passed in milliseconds
   long pastMillis = now - time;
   long seconds = pastMillis/1000;
   long minutes = seconds/60;

   if (minutes > 0) {
     return minutes + " minutes ago";
   } else {
     return seconds + " seconds ago";
   }
 }
}
```

NewsFeed (à la Facebook)

- **Example**: "`network-v1`":

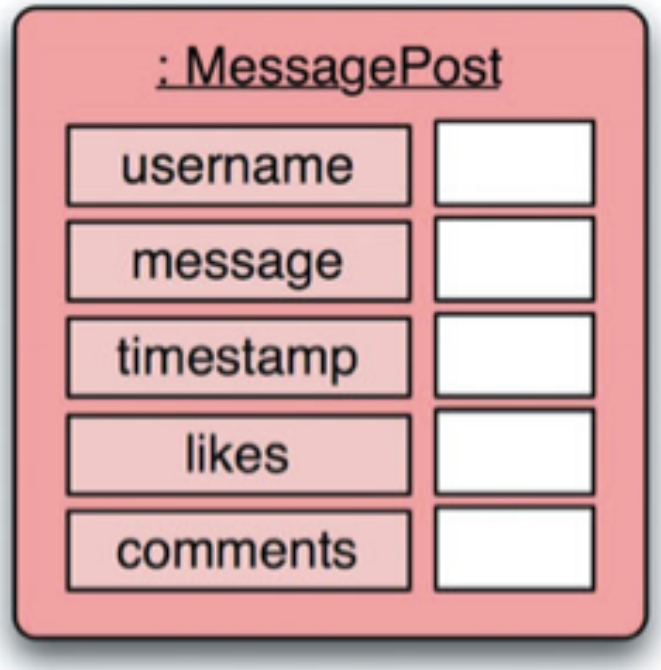


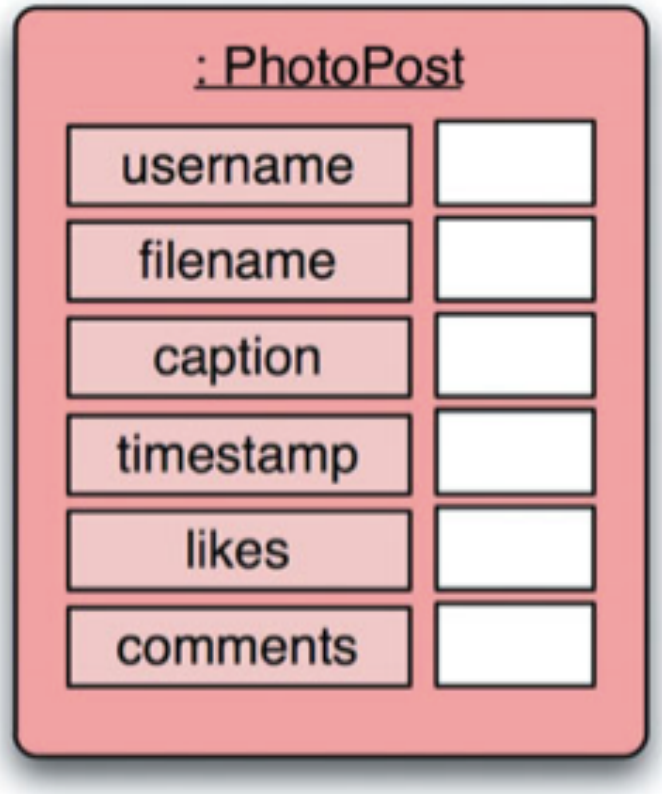
Problem!

≈ **Duplikering**! ☹

(Redundans)

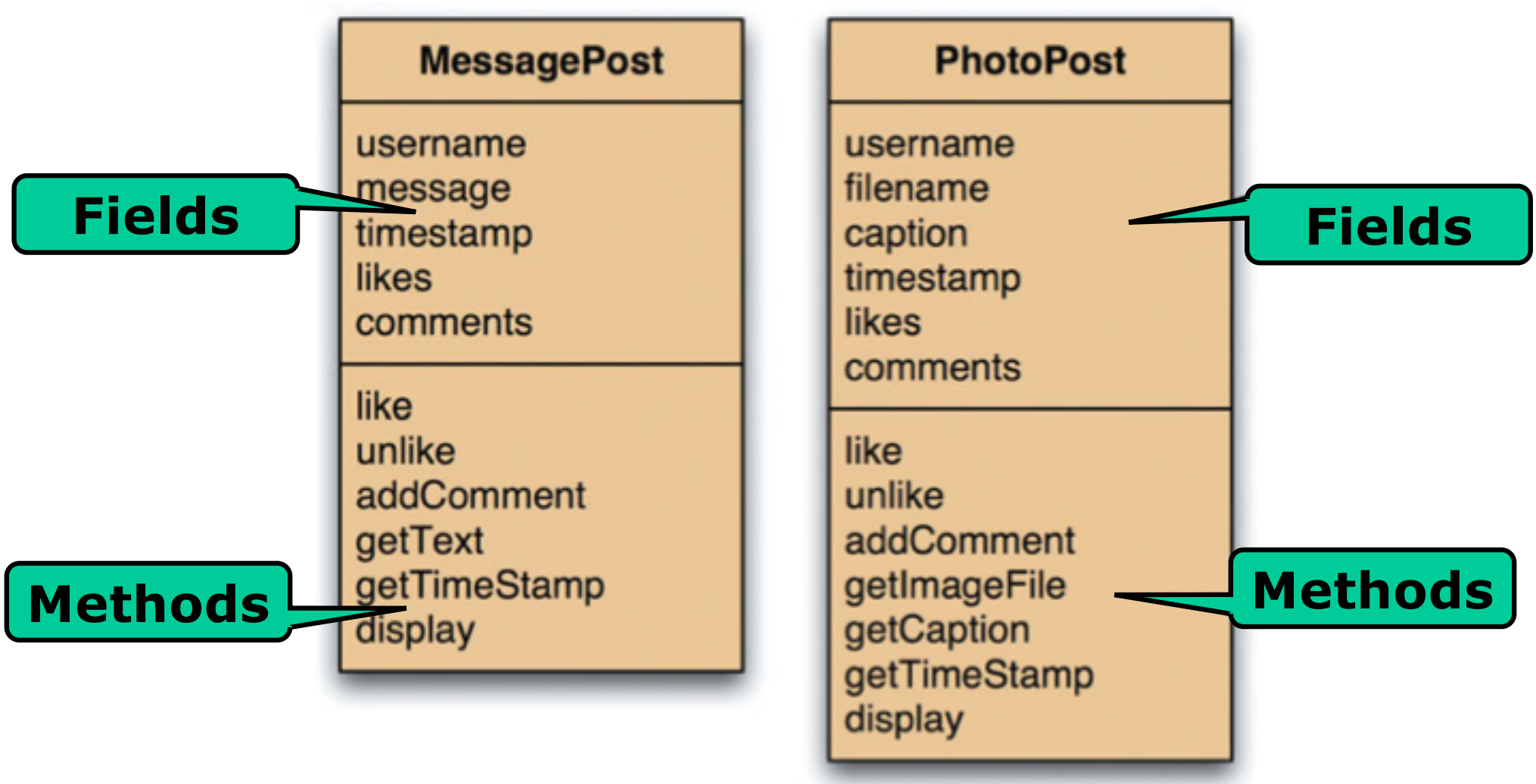
MessagePost og PhotoPost





<u>NB</u>: mange felter findes i ***begge klasser:***
`(username, timestamp, likes, comments)`

MessagePost og PhotoPost



<u>NB</u>: mange metoder findes i ***begge klasser:***
`(like(), unlike(), addComment(), ...)`

MessagePost.java

```
public class MessagePost {
  private String username;
  private String message;

  private long ts;
  private int likes;
  private ArrayList<String> comments;

  public MessagePost(String author,
                     String text) {

    username = author;
    message = text;

    ts = System.currentTimeMillis();
    likes = 0;
    comments = new ArrayList<String>();
  }

  ...
```

PhotoPost.java

```
public class PhotoPost {
  private String username;
  private String filename;
  private String caption;
  private long ts;
  private int likes;
  private ArrayList<String> comments;

  public PhotoPost(String author,
                   String filename,
                   String caption) {
    username = author;
    this.filename = filename;
    this.caption = caption;
    ts = System.currentTimeMillis();
    likes = 0;
    comments = new ArrayList<String>();
  }

  ...
```

Forskelle markerede med Rødt!

MessagePost.java

```java
...

public void like() {
  likes++;
}

public void unlike() {
  if (likes > 0) {
    likes--;
  }
}



public String getText() {
  return message;
}



public long getTimeStamp() {
  return ts;
}

...
```

PhotoPost.java

```java
...

public void like() {
  likes++;
}

public void unlike() {
  if (likes > 0) {
    likes--;
  }
}

public String getImageFile() {
  return filename;
}

public String getCaption() {
  return caption;
}

public long getTimeStamp() {
  return ts;
}

...
```

MessagePost.java

```
...

public void display() {
  System.out.println(username);
  System.out.println(message);


  System.out.print(timeString(ts));

  if (likes > 0) {
    System.out.println("  -  " +
      likes + " people like this.");
  } else {
    System.out.println();
  }

  if (comments.isEmpty()) {
    System.out.println(
        "   No comments.");
  } else {
    System.out.println(
        "   " + comments.size() +
        " comment(s).");
  }
}

...
```

PhotoPost.java

```
...

public void display() {
  System.out.println(username);
  System.out.println(
            "  [" + filename + "]");
  System.out.println("  " + caption);
  System.out.print(timeString(ts));

  if (likes > 0) {
    System.out.println("  -  " +
      likes + " people like this.");
  } else {
     System.out.println();
  }

  if (comments.isEmpty()) {
    System.out.println(
        "   No comments.");
  } else {
    System.out.println(
        "   " + comments.size() +
        " comment(s). ");
  }
}

...
```

MessagePost.java

```
  ...

  private String timeString(long time) {
    long now = System.currentTimeMillis();

    // time passed in milliseconds
    long pastMillis = now - time;
    long seconds = pastMillis/1000;
    long minutes = seconds/60;

    if (minutes > 0) {
      return minutes + " minutes ago";
    } else {
      return seconds + " seconds ago";
    }
  }
}
```

PhotoPost.java

```
  ...

  private String timeString(long time) {
    long now = System.currentTimeMillis();

    // time passed in milliseconds
    long pastMillis = now - time;
    long seconds = pastMillis/1000;
    long minutes = seconds/60;

    if (minutes > 0) {
      return minutes + " minutes ago";
    } else {
      return seconds + " seconds ago";
    }
  }
}
```

≈ **Duplikering!** ☹

MessagePost

Class MessagePost

```
java.lang.Object
  └ MessagePost
```

```
public class MessagePostextends java.lang.Object
```

This class stores information about a post in a social network. The main part of the p

Version:
0.1
Author:
Michael Kölling and David J. Barnes

Constructor Summary

`MessagePost(java.lang.String author, java.lang.String text)` Constructor for objects of class MessagePost.	

Method Summary

void	`addComment(java.lang.String text)` Add a comment to this post.
void	`display()` Display the details of this post.
java.lang.String	`getText()` Return the text of this post.
long	`getTimeStamp()` Return the time of creation of this post.
void	`like()` Record one more 'Like' indication from a user.
void	`unlike()` Record that a user has withdrawn his/her 'Like' vote.

PhotoPost

Class PhotoPost

```
java.lang.Object
  └ PhotoPost
```

```
public class PhotoPostextends java.lang.Object
```

This class stores information about a post in a social network. The main part of the post consists of a photo and a

Version:
0.1
Author:
Michael Kölling and David J. Barnes

Constructor Summary

`PhotoPost(java.lang.String author, java.lang.String filename, java.lang.String caption)` Constructor for objects of class PhotoPost.	

Method Summary

void	`addComment(java.lang.String text)` Add a comment to this post.
void	`display()` Display the details of this post.
java.lang.String	`getCaption()` Return the caption of the image of this post.
java.lang.String	`getImageFile()` Return the file name of the image in this post.
long	`getTimeStamp()` Return the time of creation of this post.
void	`like()` Record one more 'Like' indication from a user.
void	`unlike()` Record that a user has withdrawn his/her 'Like' vote.

MessagePost -vs- PhotoPost

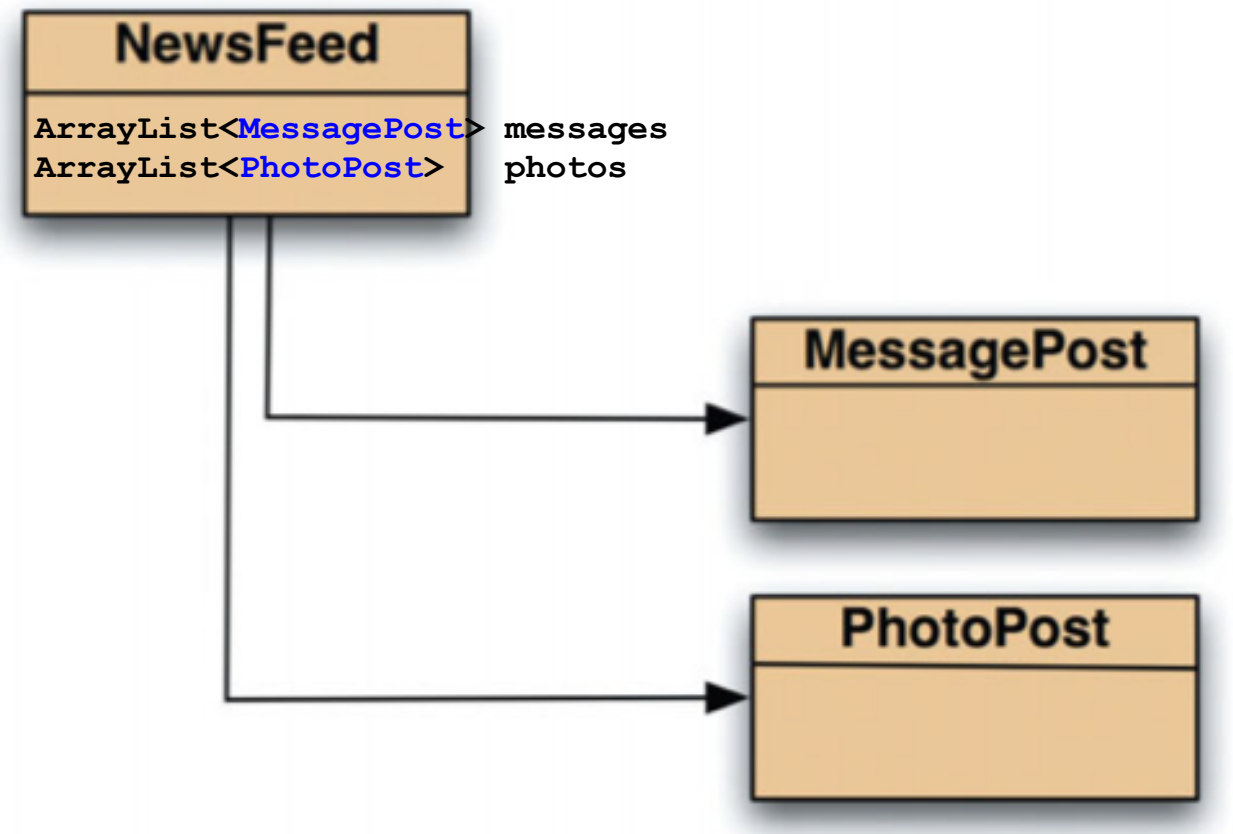
"Looking at both classes, we quickly notice that they are very similar.

This is not surprising, because their purpose is similar: both are used to store information about news-feed posts, and the different types of posts have a lot in common.

They differ only in their details, such as some of their fields and corresponding accessors and the bodies of the ***`display`*** *method."*

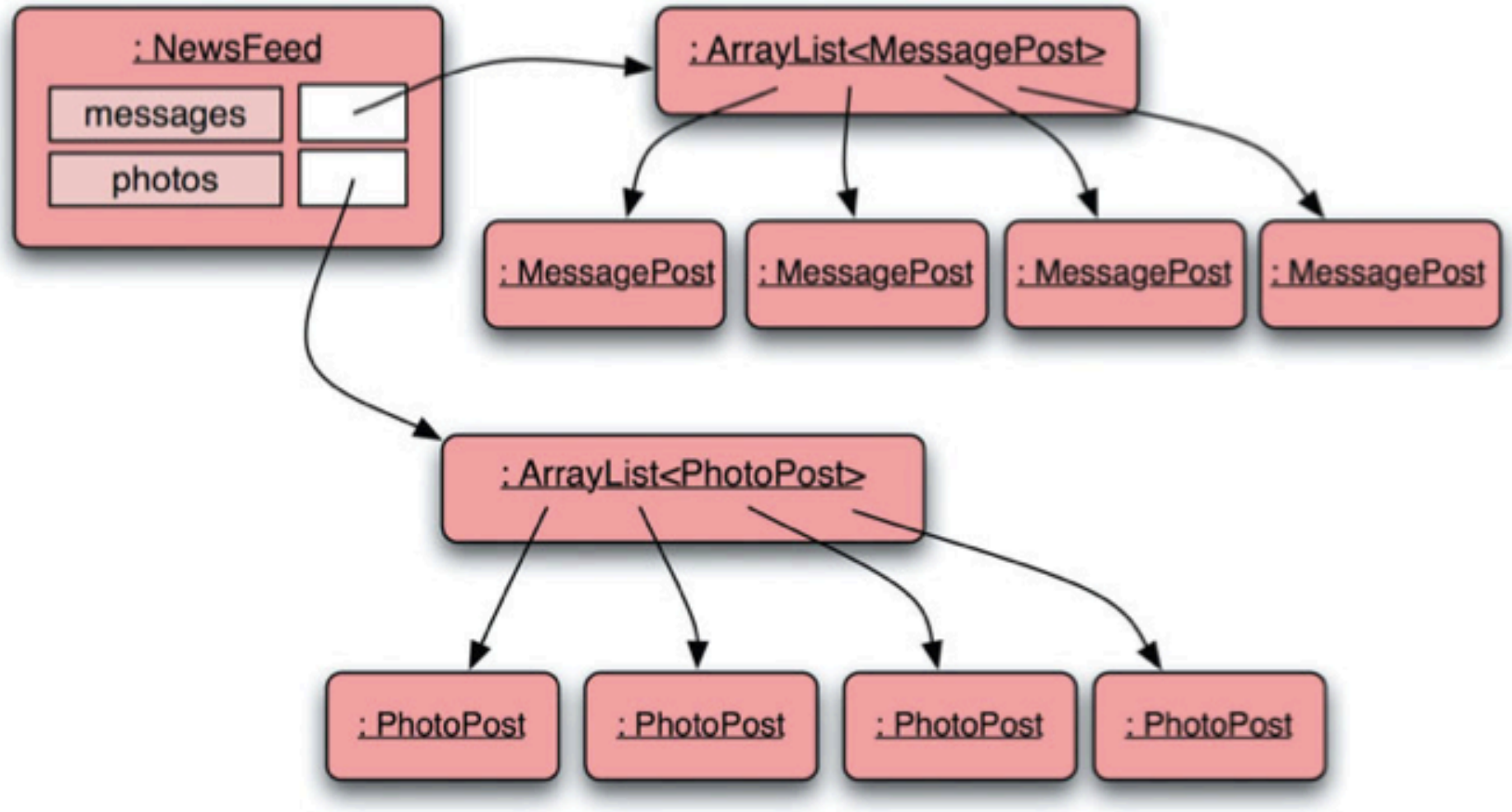
-- [Barnes & Kölling, p. 338]

Klassediagram

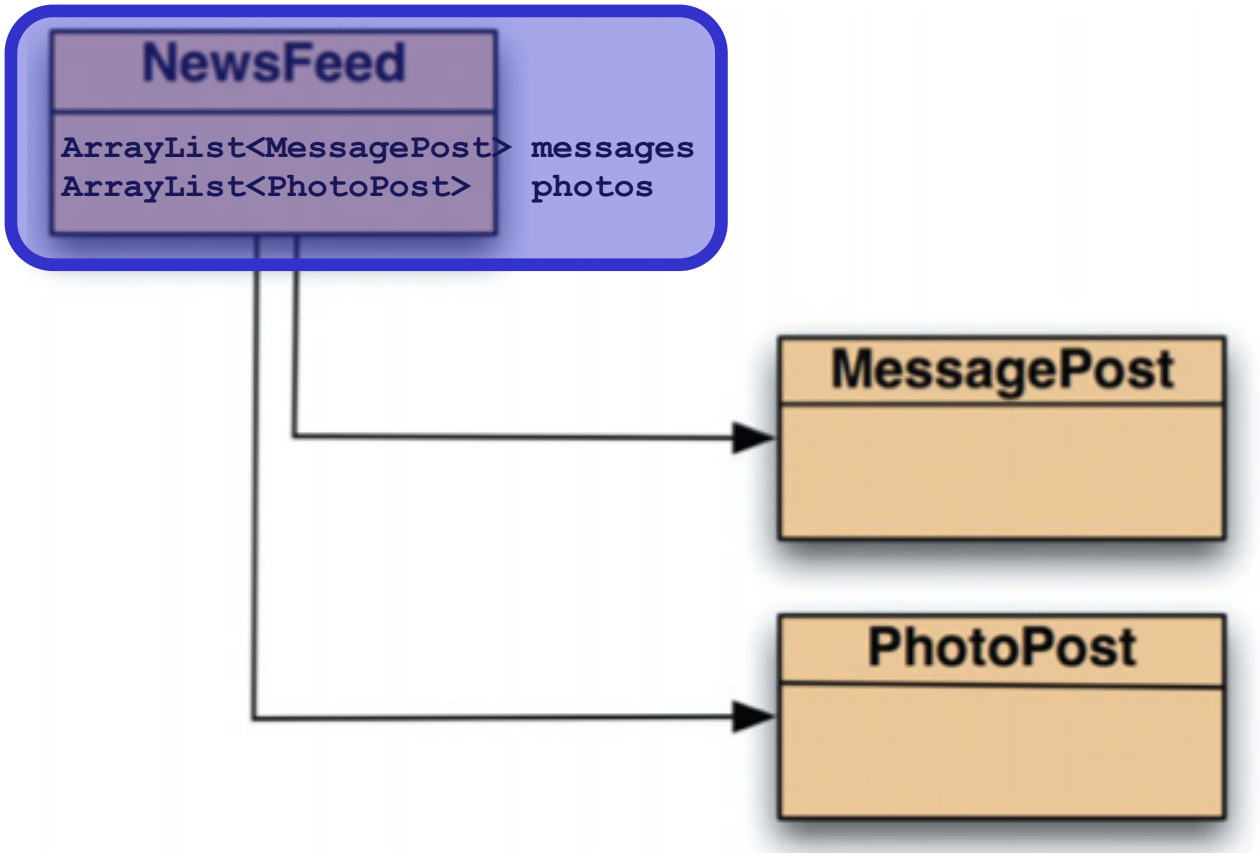


- NewsFeed bruger MessagePost og PhotoPost

Objekt-diagram
(4 MessagePosts & 4 PhotoPosts)



Klassediagram



Kildetekst for klassen NewsFeed

```
public class NewsFeed {
   private ArrayList<MessagePost> messages;
   private ArrayList<PhotoPost> photos;

   public NewsFeed() {
      messages = new ArrayList<MessagePost>();
      photos = new ArrayList<PhotoPost>();
   }

   public void addMessagePost(MessagePost message) {
      messages.add(message);
   }

   public void addPhotoPost(PhotoPost photo) {
        photos.add(photo);
   }

   public void show() {
      // display all text posts
      for (MessagePost message : messages) {
         message.display();
         System.out.println(); // line between posts
      }
      // display all photos
      for (PhotoPost photo : photos) {
         photo.display();
         System.out.println(); // line between posts
      }
   }
}
```

MessagePost

PhotoPost

MessagePost

PhotoPost

MessagePost

PhotoPost

MessagePost

PhotoPost

≈ **Duplikering!** ☹

Dårlig struktur

- Duplikering af kode i klasserne:
  - dobbelt arbejde at ***programmere*** det
  - dobbelt arbejde at ***dokumentere*** det
  - dobbelt arbejde at ***teste*** det
  - dobbelt arbejde at ***vedligeholde*** det
    - » fx. ændre comments fra String til Comment (2 steder)!
  - sværere at vedligeholde
  - større risiko for fejl
  - større risiko for inkonsistens
- Også kode-duplikering i klassen **`NewsFeed`**:
  - håndtere messages vs. håndtere photos
- Tilføje "**`EventPost`**" => *endnu mere* redundans (3x!)

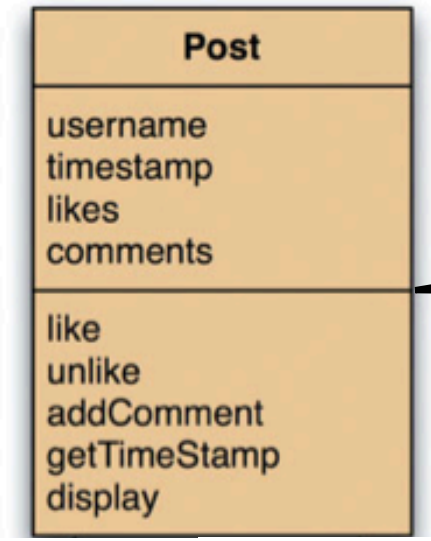
Løsning!

Objekt-Orientering ☺

(Arv / Inheritance)

OO Løsning: Arv (inheritance)

- Lav ny klasse `Post` der indeholder det ***fælles***:



Felter og metoder der findes i alle klasser

Felter og metoder der ***kun*** findes i `PhotoPost` klassen

Ord og begreber

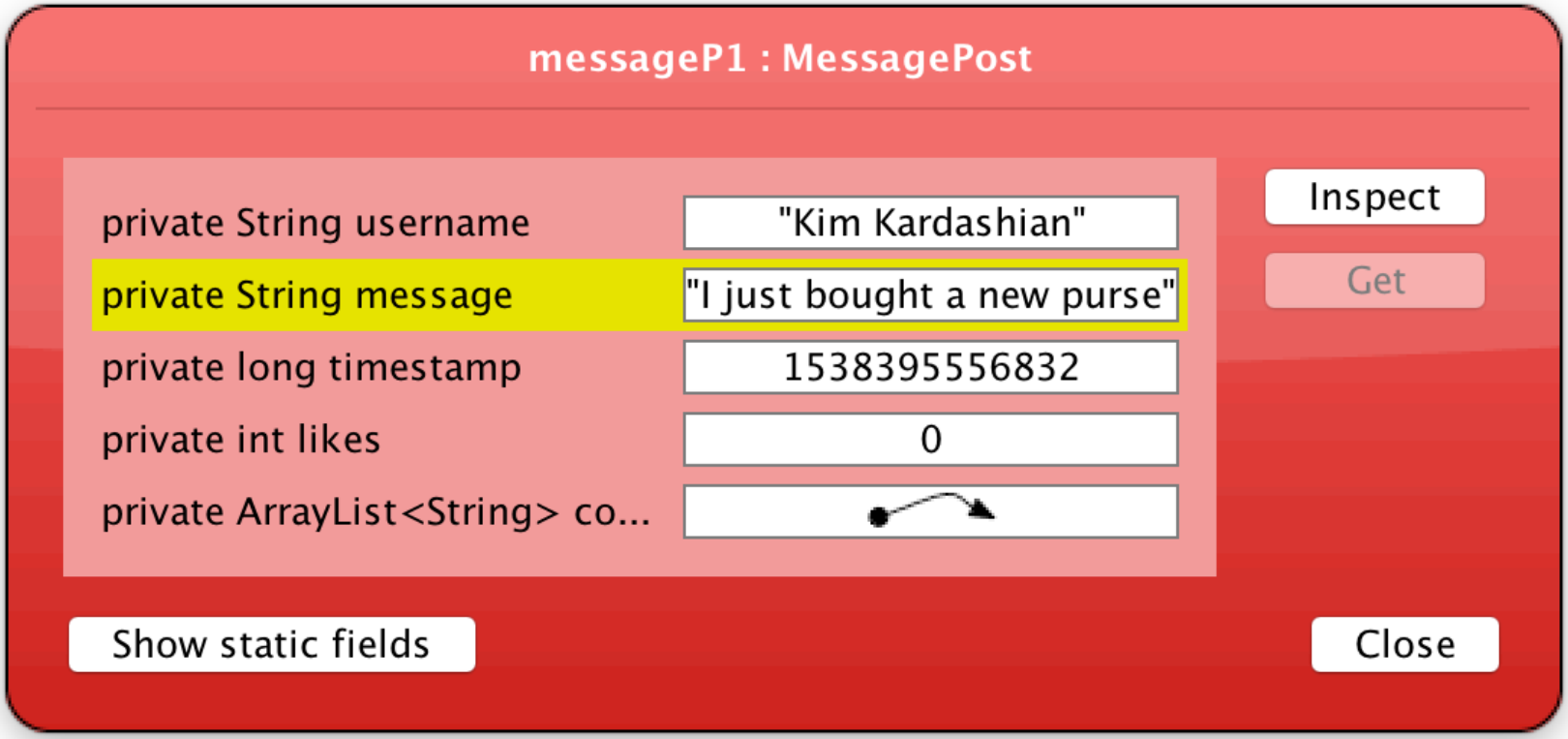
- Superklassen `Post`:
  - erklærer ***fælles*** felter og metoder
- Subklasserne `MessagePost` og `PhotoPost`:
  - ***arver alle*** superklassens felter og metoder
  - og ***erklærer egne*** særlige felter og metoder

Terminologi:

- Klassen `MessagePost` ***arver fra*** `Post`
- Klassen `MessagePost` er ***afledt af*** `Post`
- Klassen `MessagePost` er en ***subklasse*** *af* `Post`
- Klassen `MessagePost` er en ***specialisering*** af `Post`
- Klassen `Post` er ***superklasse til*** `MessagePost`
- Klassen `Post` er en ***generalisering af*** `MessagePost` & `PhotoPost`

Et `MessagePost` objekt

- Et **`MessagePost`** objekt har:
  - felter (og metoder) ***arvet fra*** `Post`; samt
  - felter (og metoder) som er ***specielle for*** **`MessagePost`**

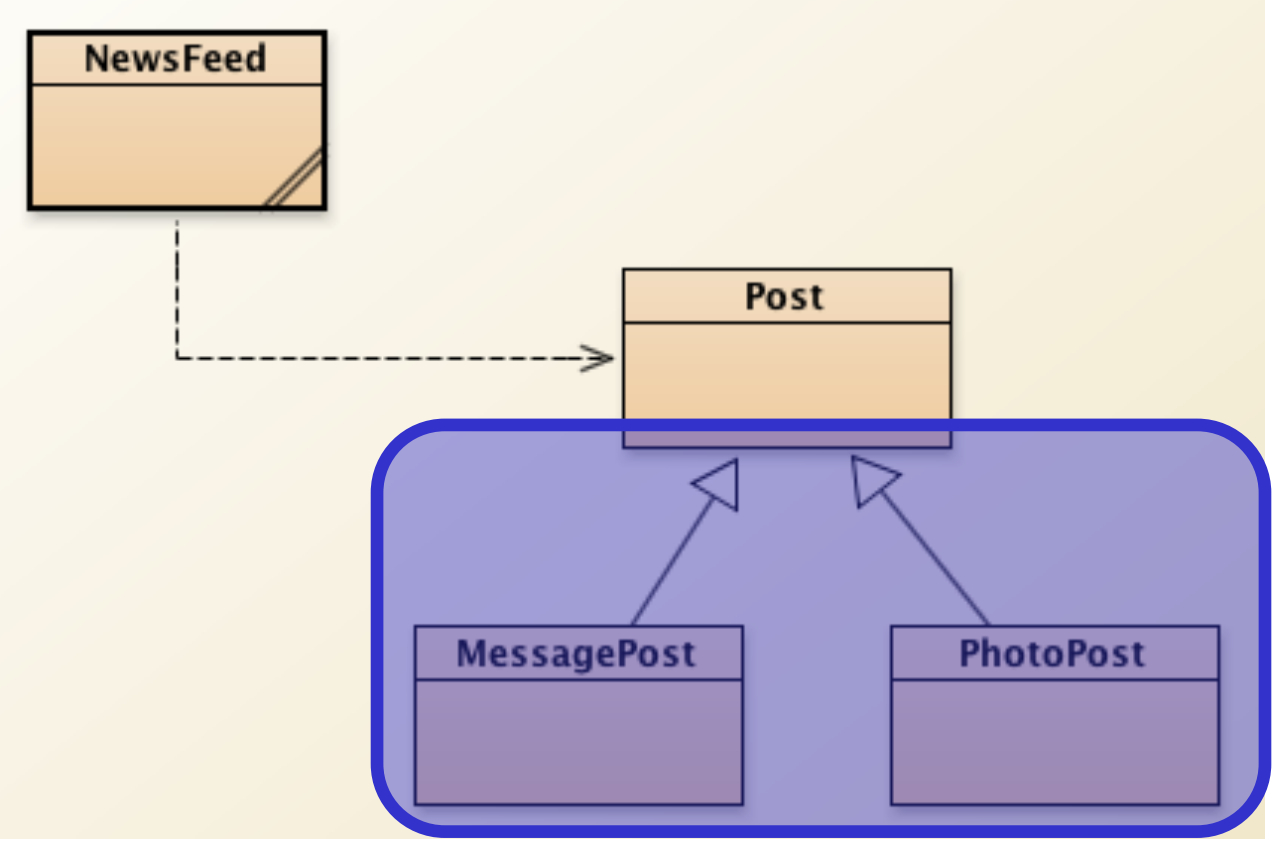


Flere ord

- Arv = nedarvning = inheritance
- Super-klasse = super-class = base class
- Sub-klasse = sub-class = derived class

Nyt klassehierarki med arv

- Kraftigt forbedret NewsFeed ***med arv:***



Hvordan udtrykkes arv i Java

```
public class Post {
    private String username;
    private long ts;
    private int likes;
    private ArrayList<String> comments;

    ... // methods
}
```

```
public class MessagePost extends Post {
    private String message;


    ... // methods
}
```

```
public class PhotoPost extends Post {
    private String filename;
    private String caption;

    ... // methods
}
```

"`MessagePost extends Post`" betyder at:
`MessagePost` er en ***subklasse af*** `Post`
(og `Post` er en ***superklasse for*** `MessagePost`)

Kildetekst for super-klasse `Post`

```
public class Post {
  private String username;
  private long timestamp;
  private int likes;
  private ArrayList<String> comments;

  public Post(String author) {
    username = author;
    timestamp = System.currentTimeMillis();
    likes = 0;
    comments = new ArrayList<String>();
  }

  public void like() {
    likes++;
  }

  public void unlike() {
    if (likes > 0) {
      likes--;
    }
  }

  public void addComment(String text) {
    comments.add(text);
  }

  public long getTimeStamp() {
    return timestamp;
  }

  ...
```

Alt det fælles fra **`MessagePost`** og **`PhotoPost`**

```
  ...

  public void display() {
    System.out.println(username);
    System.out.print(timeString(timestamp));

    if (likes > 0) {
      System.out.println("  -  " + likes +
                         " people like this.");
    } else {
      System.out.println();
    }

    if (comments.isEmpty()) {
      System.out.println("   No comments.");
    } else {
      System.out.println("   " +
      comments.size() + " comment(s).");
    }
  }

  private String timeString(long time) {
    ...
  }
}
```

Kildetekst for nye sub-klasser: (MessagePost og PhotoPost)

```
public class MessagePost extends Post {
  private String message;

  public MessagePost(String author,
                     String text) {
    super(author);
    message = text;
  }

  public String getText() {
    return message;
  }
}
```

Call the super-class's constructor

Speciel metode for MessagePost

```
public class PhotoPost extends Post {
  private String filename;
  private String caption;

  public PhotoPost(String author,
                   String filename,
                   String caption) {
    super(author);
    this.filename = filename;
    this.caption = caption;
  }

  public String getImageFile() {
    return filename;
  }

  public String getCaption() {
    return caption;
  }
}
```

Specielle metoder for PhotoPost

NB: man *skal altid* kalde super-klassens constructor som det første i sub-klassens constructor. Gør man ikke det, bliver der automatisk indsat et "implicit kald":

- `super(); // uden argumenter!`

NB: Ingen kodeduplikering! ☺

Kald til superklasse-konstruktoren

```
public class Post {
  private String username;
  private long timestamp;
  private int likes;
  private ArrayList<String> comments;

  public Post(String author) {
    username = author;
    timestamp = System.currentTimeMillis();
    likes = 0;
    comments = new ArrayList<String>();
  }
  ...
}
```

```
public class MessagePost extends Post {
  private String message;

  public MessagePost(String author,
                     String message) {
    super(author);
    this.message = message;
  }
  ...
}
```

Kalder konstruktor i superklassen

- Superklassens konstruktør skal kaldes
- Ellers indsættes automatisk **`super();`**

Et større eksempel

- **Super-klasse** (med 3 felter):

```
public class Person {
    private int age;
    private String name;
    private boolean isFemale;

    public Person(int age, String name, boolean isFemale) {
        this.age = age;
        this.name = name;
        this.isFemale = isFemale;
    }
}
```

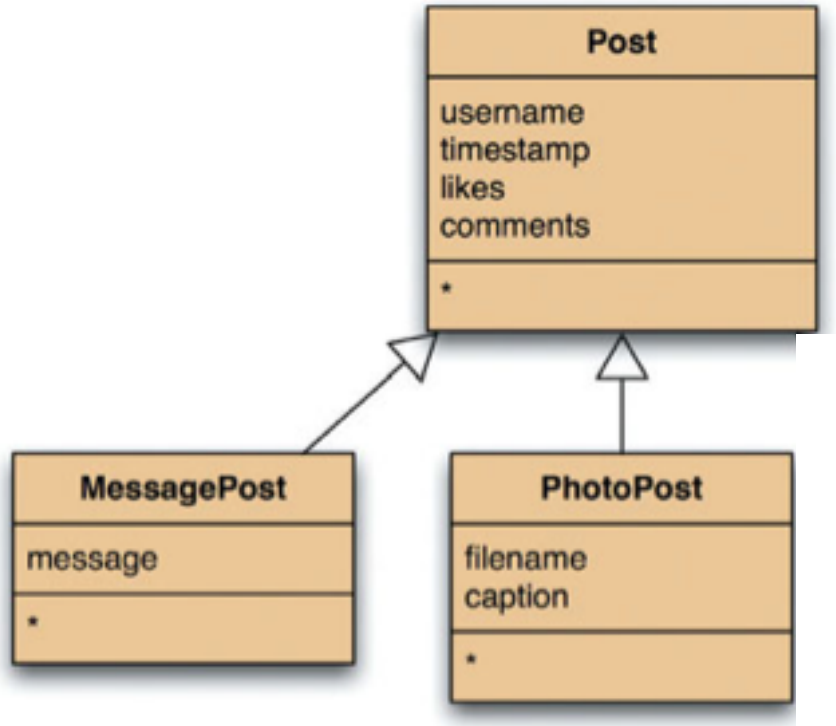
3 argumenter

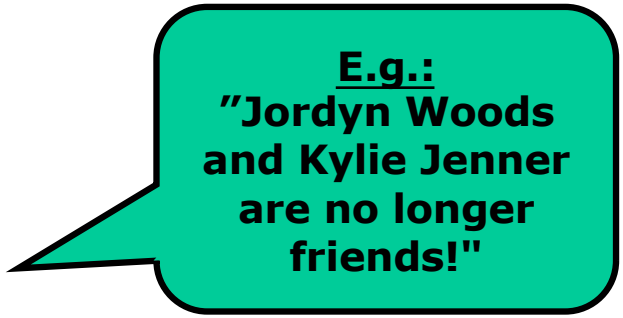
- **Sub-klasse** (+2 ekstra felter):

```
public class Student extends Person {
    private String username;
    private int id;

    public Student(int age, String name, boolean isFemale, String username, int id) {
        super(age, name, isFemale);
        this.username = username;
        this.id = id;
    }
}
```

3+2=5 argumenter

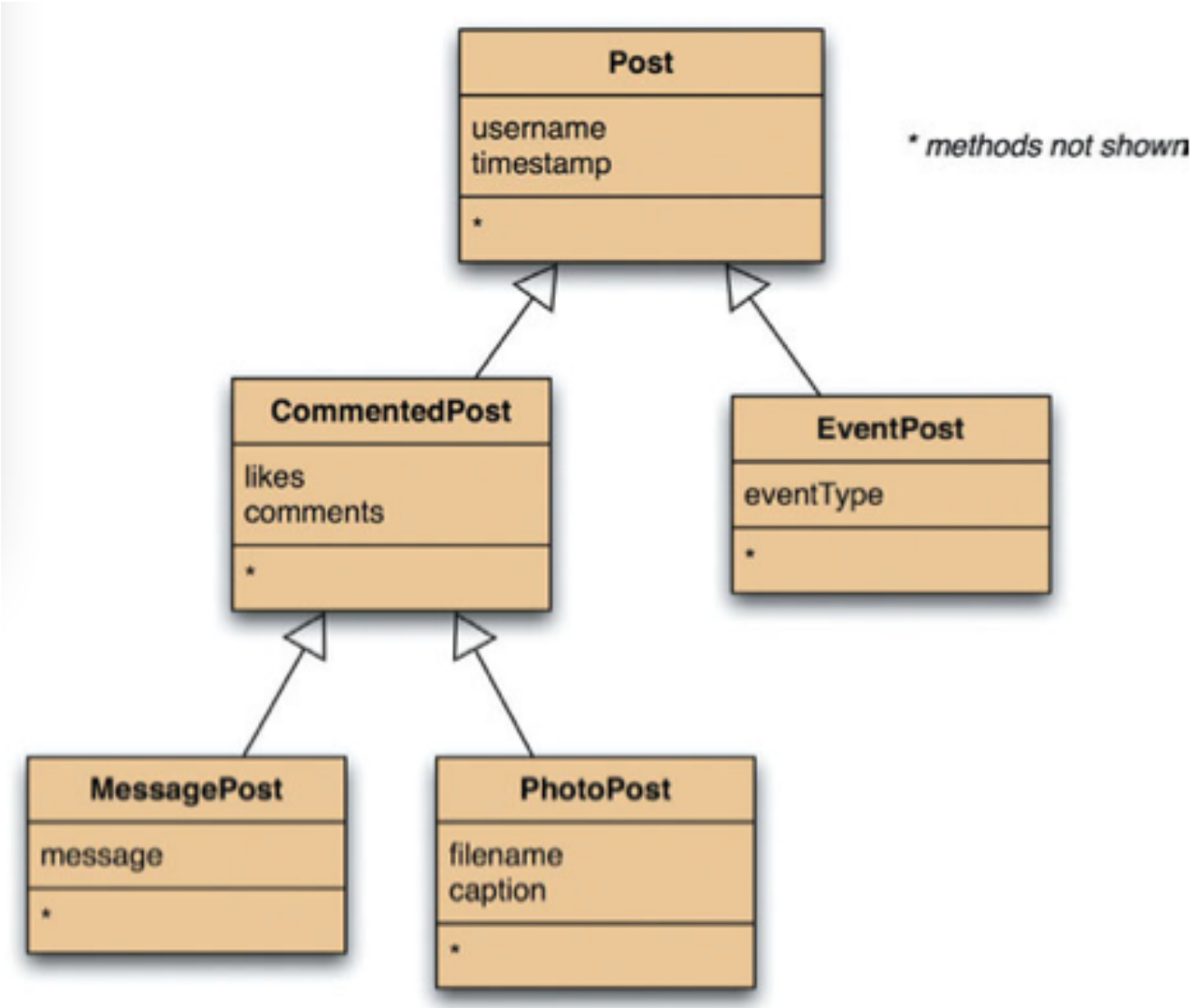
En hypotetisk udvidelse: "EventPost"





- ***Kode-genbrug*** (en af mange fordele ved OO)**! :-)**
- Lad os sige at EventPost's ikke har likes & comments.
  - (i.e., EventPost's kun er til FYI information)
- **Q:** Hvor skal vi så definere likes & comments?
  - **1)** Blot lade være med at bruge likes & comments i EventPost? **:-(**
  - **2)** Flytte likes & comments til sub-klasser MessagePost og PhotoPost? **:-(**
  - **3)** Lade være med at lade EventPost arve fra Post? **:-(**
  - **4)** eller...? [refactor class-hierarchy]

Klassehierarki med flere niveauer



:-)

- **NB:** Klassehierarkier kan være så dybe man vil

Fordele og Ulemper ved arv?
(indtil videre)

- **Q1:** Hvad er fordelene ved arv?
- **Q2:** Ulemper ved arv?

Opsamling

- **Fordele ved arv:**
  - ***Undgå kode-duplikering*** (og alle relaterede problemer)
  - ***Kode-genbrug*** (også for fremtidige klasser: EventPost)
  - ***Nemmere vedligeholdelse*** (ændringer i fælles: ét sted)
  - ***Nemt af udvide:***
    - » Fx. tilføjer man et felt `language` på `Post`, så får alle sub-klasserne det automatisk **!**

- **Ulemper ved arv:**
  - Det kræver "abstraktion" (generalisering/specialisering)
  - Det kræver planlægning
  - Det kan kræve refaktorisering
  - Et ulogisk klassehierarki gør vedligeholdelse umuligt

Sub-Typing

Den nye NewsFeed-klasse

```
public class NewsFeed {
   private ArrayList<Post> posts;

   public NewsFeed() {
      posts = new ArrayList<Post>();
   }

   public void addPost(Post post) {
      posts.add(post);
   }

   public void show() {
      // display all posts
      for (Post post : posts) {
         post.display();
         System.out.println();
      }
   }
}
```

Håndterer Post; dvs både MessagePost og PhotoPost

Håndterer Post; dvs både MessagePost og PhotoPost

Håndterer Post; dvs både MessagePost og PhotoPost

Håndterer Post; dvs både MessagePost og PhotoPost

- **NB:** Vi kan nu blot bruge Post i stedet for MessagePost henholdsvis PhotoPost

Substitutionsprincippet

- Metoden: `public void addPost(Post post);`

  ...i NewsFeed kan kaldes med:
  - argument af type: `Post` (som der står ovenfor)
  - argument af type: `MessagePost` (sub-type)
  - argument af type: `PhotoPost` (sub-type)

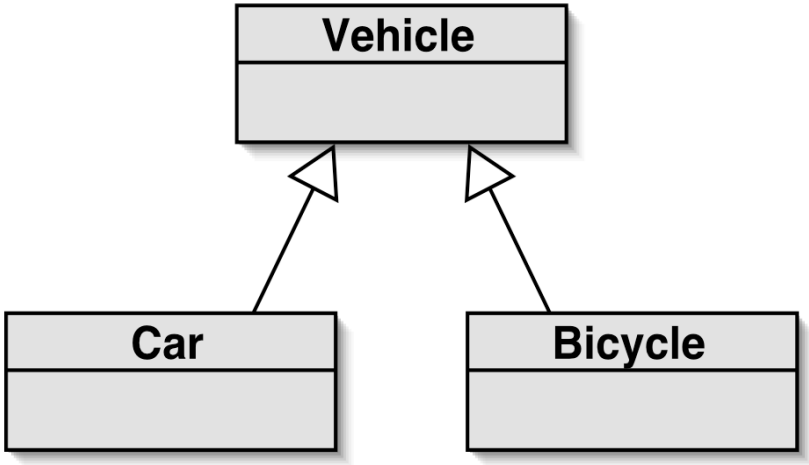
```
NewsFeed newsfeed = new NewsFeed();
MessagePost p1 = new MessagePost("Kim Kardashian",          // author
                                 "I just bought a purse");// message
newsfeed.addPost(p1); // add MessagePost!!!
PhotoPost p2 = new PhotoPost("Kim Kardashian",              // author
                             "purse.jpg",                   // filename
                             "Me and my purse");            // caption
newsfeed.addPost(p2); // add PhotoPost!!!
```

Liskov Substitution Principle:

*Et objekt af en **subklasse** kan altid bruges hvor et objekt af superklassen forventes.*

-- Barbara Liskov, MIT, 1987

Subtyper og tildeling (=)




```
Vehicle v;
v = new Vehicle();
v = new Car();
v = new Bicycle();
```

OK som følge af substitutions-princippet

```
Car c = new Vehicle();  // error!
```

<u>Liskov Substitution Principle</u>:

*Et objekt af en **subklasse** kan altid bruges hvor et objekt af superklassen forventes.*

-- Barbara Liskov, MIT, 1987

Variable og subklasser

- Tilsvarende for en variabel af type **`Post`**:

```
NewsFeed newsfeed = new NewsFeed();
newsfeed.add(new MessagePost("Kim Kardashian", "I just bought a new purse"));
newsfeed.add(new PhotoPost("Kim Kardashian", "purse.jpg", "Me and my purse"));
newsfeed.show();
```

```
public class NewsFeed {
  private ArrayList<Post> posts;
  ...
  public void show() {
    for (Post post : posts) {
      post.display();
      System.out.println();
    }
  }
}
```

Et loop:
Peger på **`MessagePost`**
eller **`PhotoPost`** objekt
(aldrig blot **`Post`**)

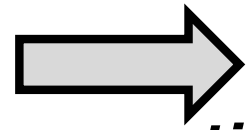
Et loop for **`MessagePost`**
og et loop for **`PhotoPost`**

```
public void show() {
  for (MessagePost message : messages) {
    message.display();
    System.out.println();
  }
  for (PhotoPost photo : photos) {
    photo.display();
    System.out.println();
  }
}
```

- Sammenlign med oprindelige version "**`network-v1`**":

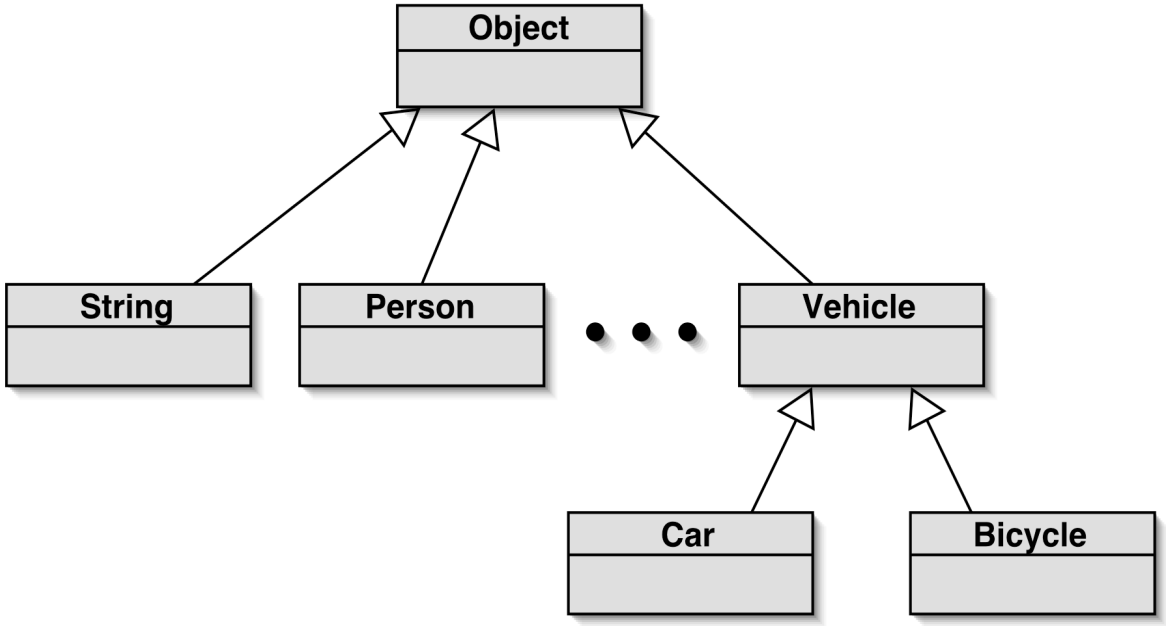
Klasse 'Object'
(er superklasse for alle klasser)

```
public class Vehicle {
   private String username;
   ...
}
```




```
public class Vehicle extends Object {
   private String username;
   ...
}
```

- Så klassehierarkiet ser faktisk således ud:



- Alle klasser (undtagen Object selv) nedarver fra Object

Klassetyper og primitive typer

- Alle *klassetyper* arver fra **`Object`**
- De *primitive* typer **`int`**, **`double`**, **`boolean`**, … arver ***ikke*** fra Objekt (er ikke klassetyper)

- Collections kan kun indeholde klassetyper:
  - Derfor har vi *wrapperklasser:*

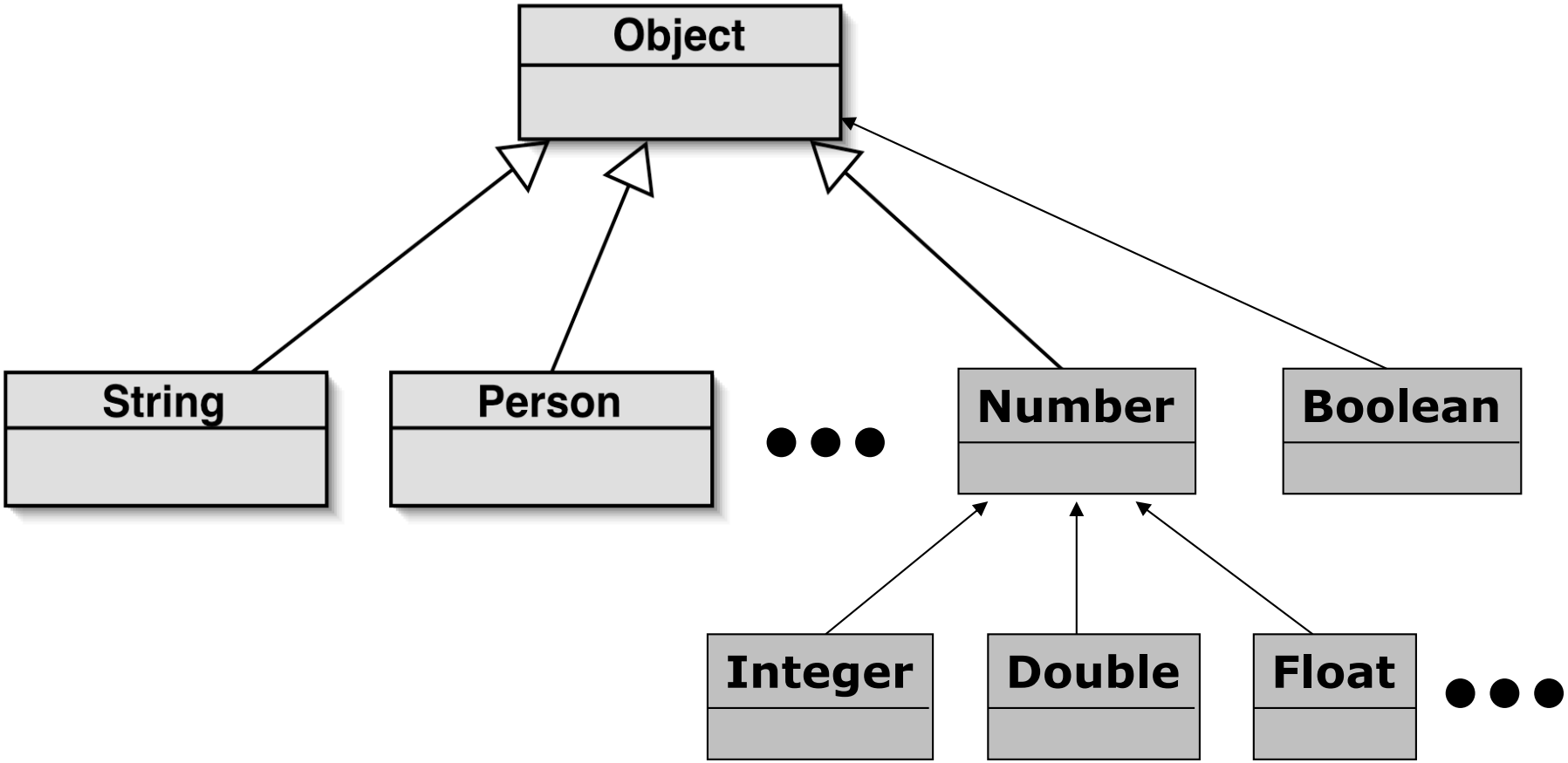
```
ArrayList<int> point;              // error!
point = new ArrayList<int>();      // error!
point.add(42);
```

Ulovligt

```
ArrayList<Integer> point;          // ok!
point = new ArrayList<Integer>(); // ok!
point.add(42);  // 42 gets "auto-boxed"!
```

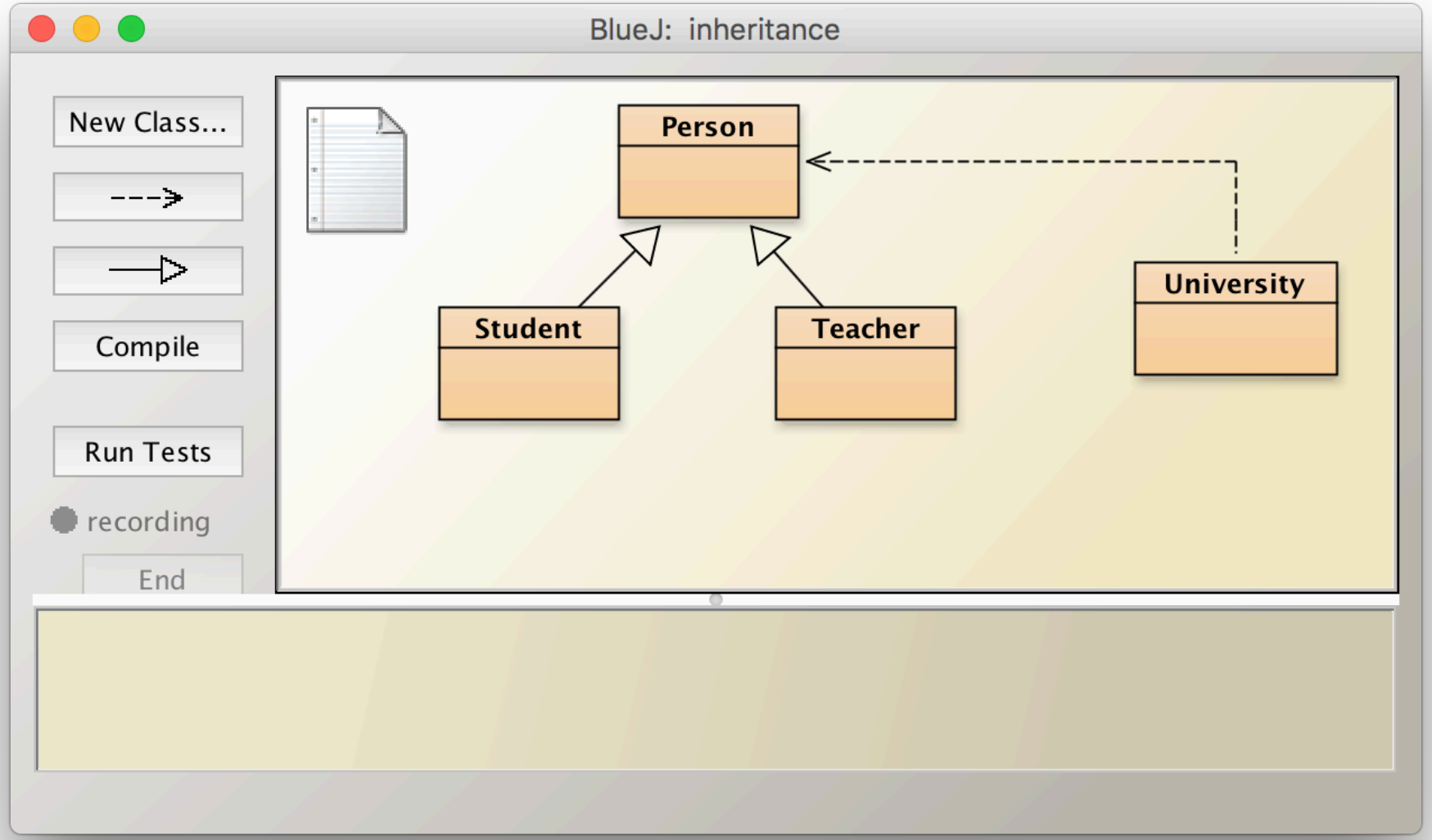
OK

Wrapperklasserne er klassetyper

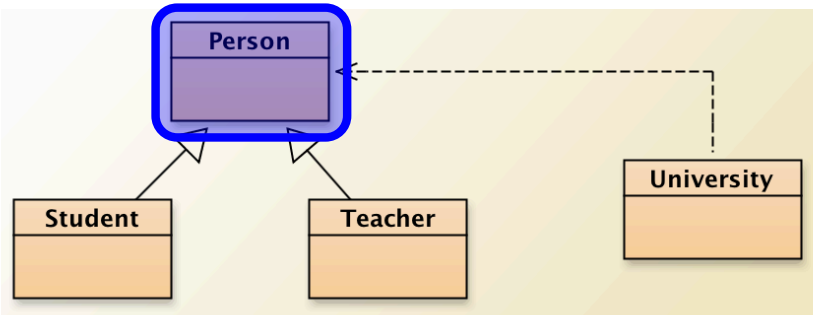


Minimalt Eksempel

Minimalt Eksempel

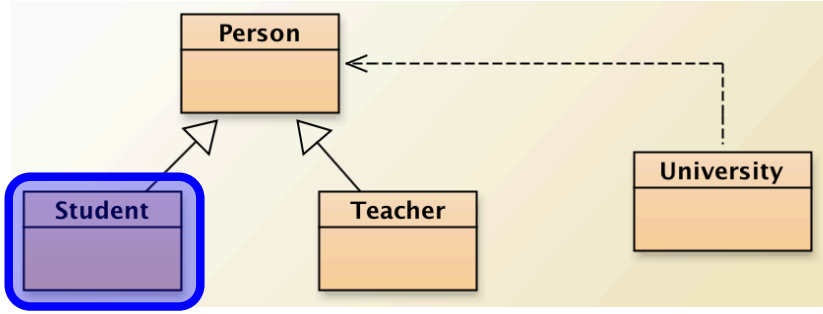


Person



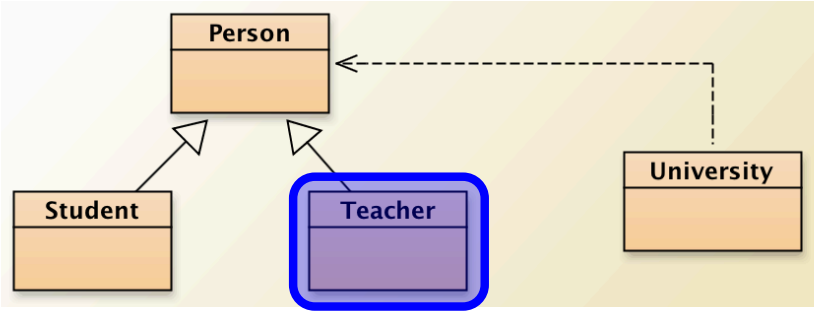

```
public class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }

    public void display() {
        System.out.println(name);
    }
}
```

Student



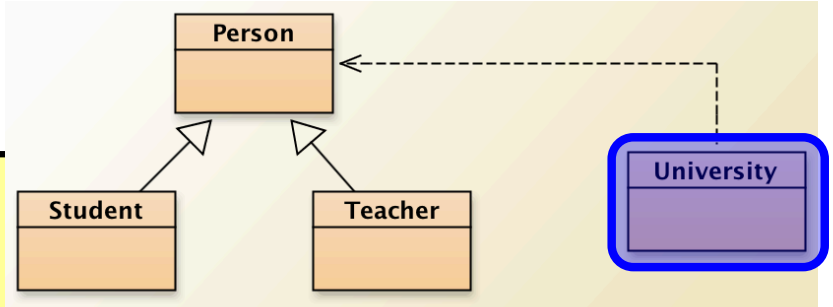

```
public class Student extends Person {
    private int id;

    public Student(String name, int id) {
        super(name);
        this.id = id;
    }
}
```

Teacher



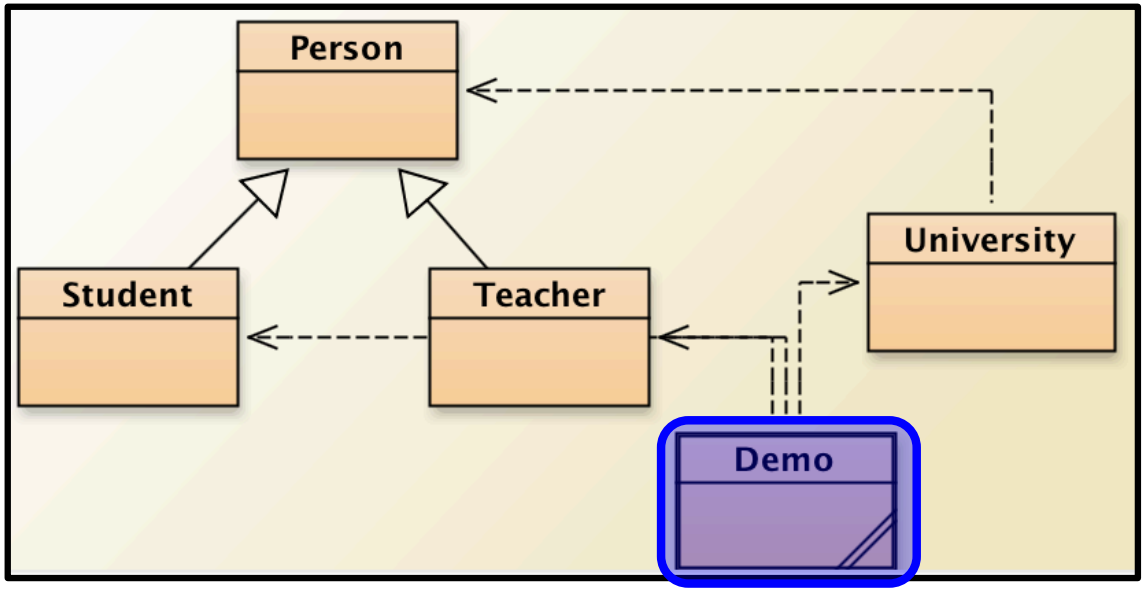

```
public class Teacher extends Person {
    private String office;

    public Teacher(String name, String office) {
        super(name);
        this.office = office;
    }
}
```

University




```
import java.util.*;

public class University {
    private ArrayList<Person> persons;

    public University() {
        persons = new ArrayList<Person>();
    }

    public void addPerson(Person p) {
        persons.add(p);
    }

    public void show() {
        for (Person person: persons) {
            person.display();
        }
    }
}
```

Demo




```
public class Demo {
    public static void demo() {
        University itu = new University();
        itu.addPerson(new Student("Alice", 1));
        itu.addPerson(new Student("Bob", 2));
        itu.addPerson(new Teacher("Claus", "4d12"));
        itu.show();
    }
}
```

```
Alice
Bob
Claus
```

På Fredag

- Mere om Arv / Inheritance
- Løse problem vedr display() i Post
- Virtual dispatching

Tak!

Spørgsmål? / Kommentarer? / Klager?