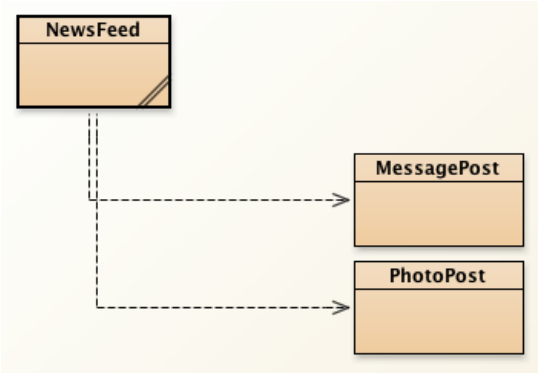
Arv, superklasser & subklasser
(Generalisering & Specialisering)

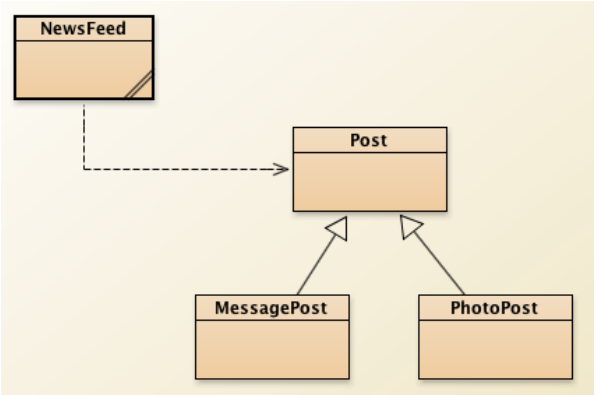
GRPRO: "Grundlæggende Programmering"

A G E N D A

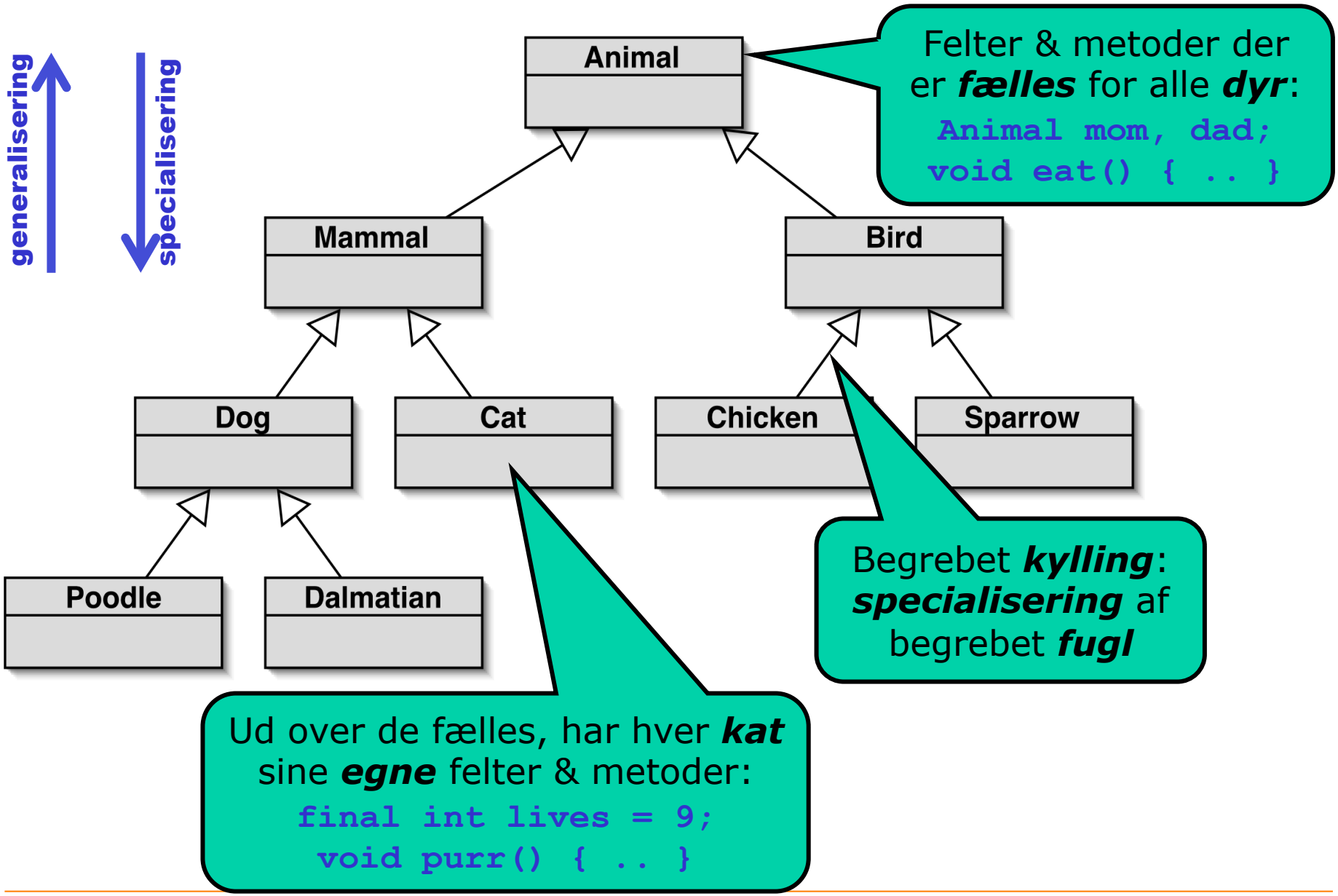
- Nedarvning
- Klassehierakier
- Substitutionsprincippet og tildeling
- Casting (Dynamisk cast)

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

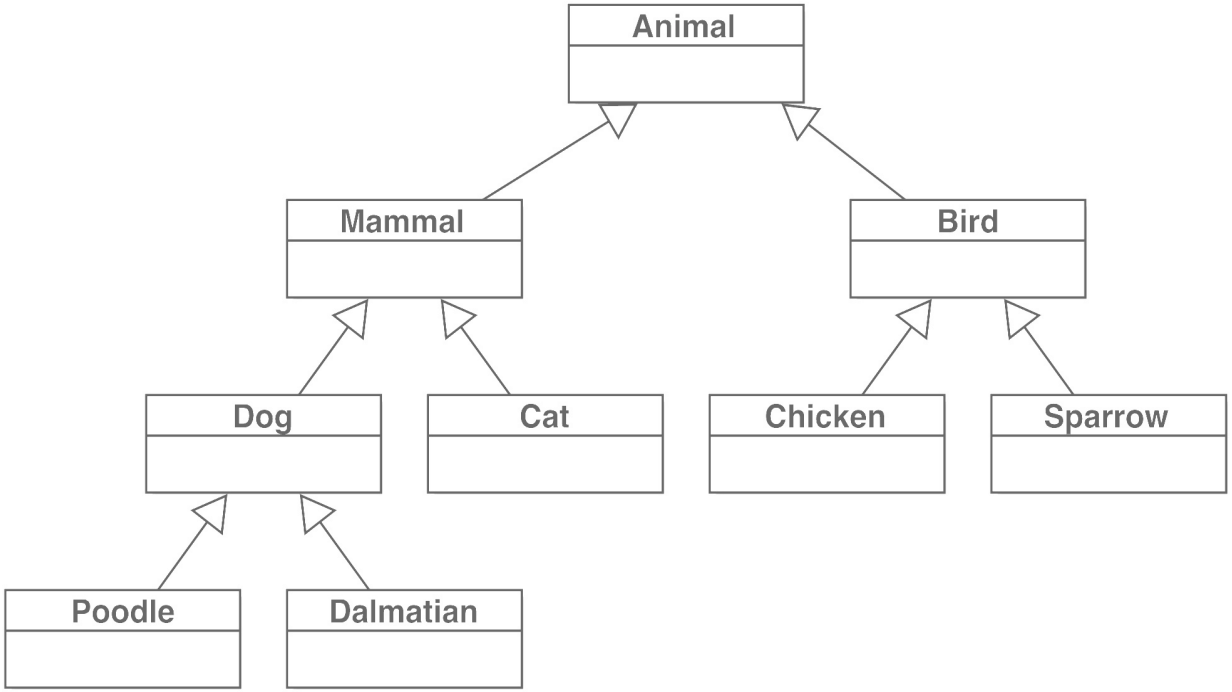




Klassehierarkier er en velkendt ide



OPGAVE



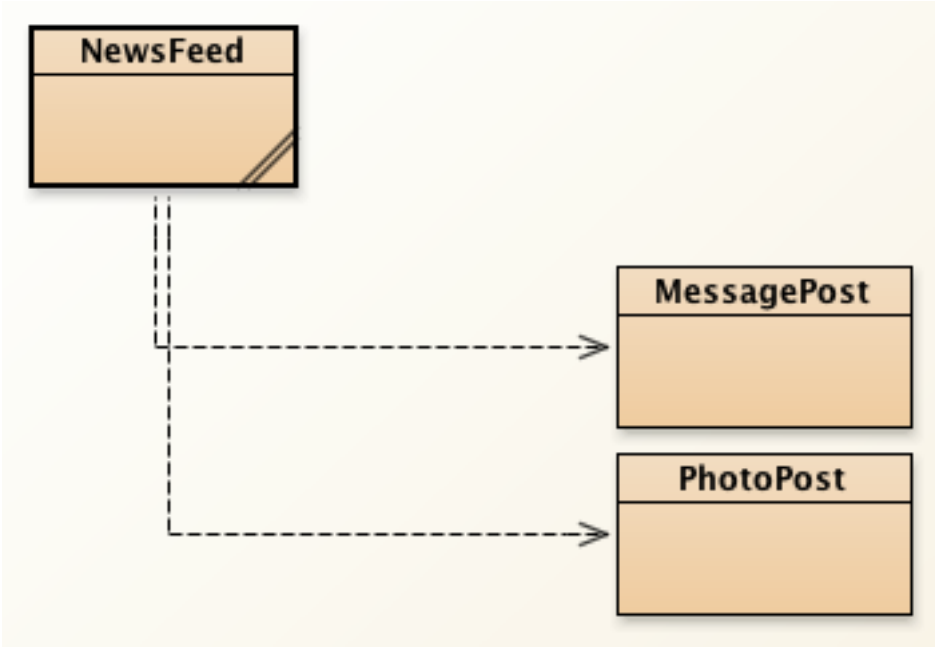
- **Find og "tegn”:**
  To eksempler på klassehierarkier.
  (Lav et af disse kan have en højde på 5.)

Helt centrale OO begreber!

- **Arv: *subklasse* og *superklasse***
  - ***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 bruges hvor et Person behøves
- **Virtual dispatching:**
  - I kaldet `person.printInfo()` bestemmer ***person objektet*** hvilken metode der kaldes! [næste uge]

NewsFeed (à la Facebook)

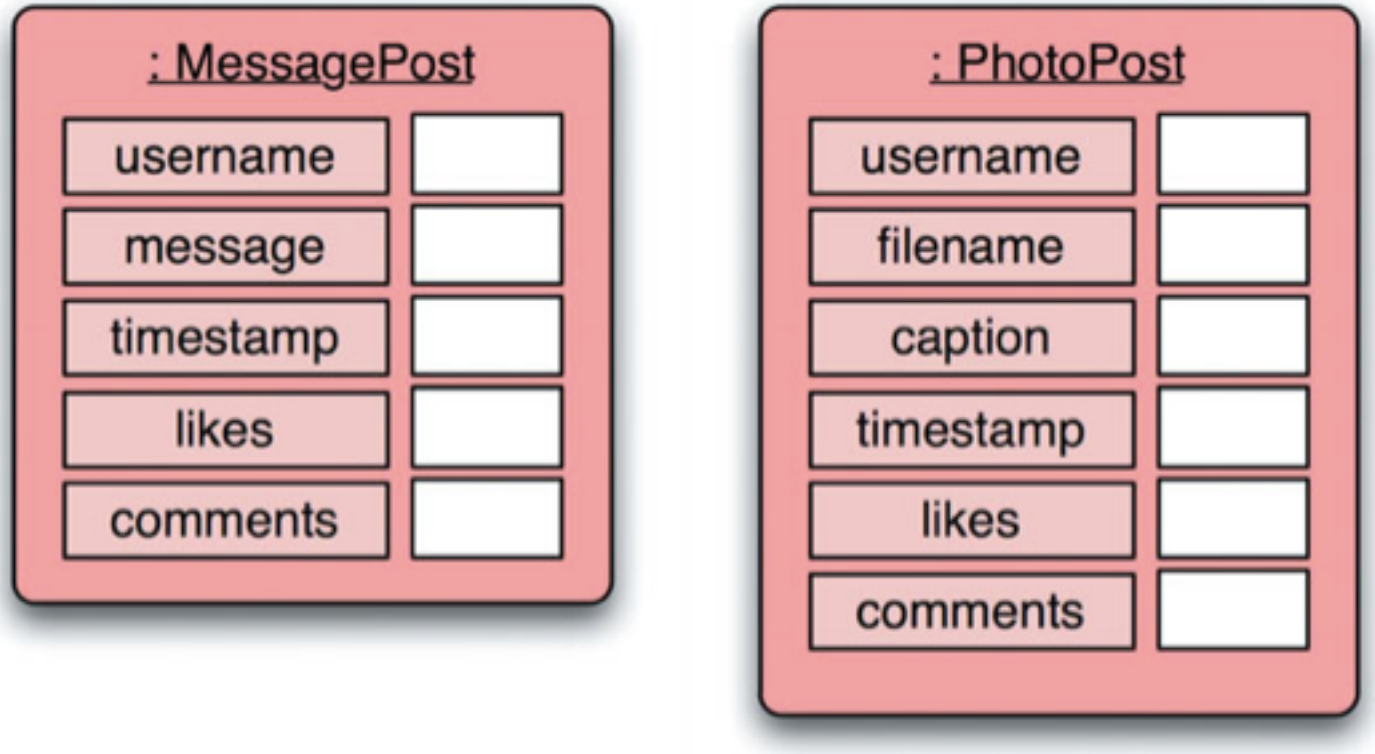
- **Example**: "`network-v1`":



NewsFeed

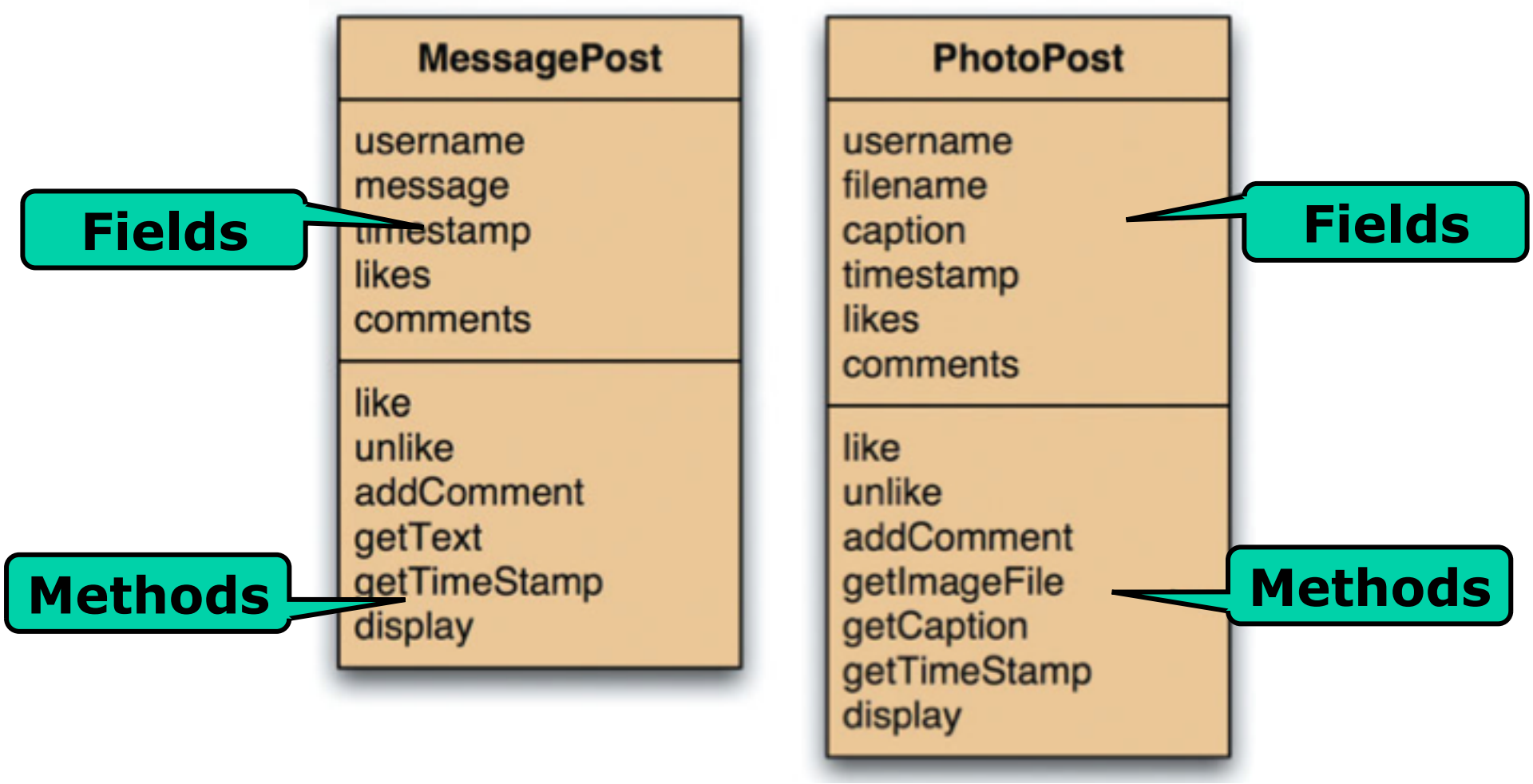
- Hvad har **`MessagePost`** og **`PhotoPost`** til fælles og hvordan adskiller de sig fra hinanden?
- Oplysninger om **`MessagePost`**:
  - username, message, timestamp, likes, comments
- Oplysninger om **`PhotoPost`**:
  - username, filename, caption, timestamp, likes, comments
- Kan udskrive **`MessagePost`** og **`PhotoPost`**
- Kan (senere) udvides:
  - søge efter comment
  - ...

MessagePost og PhotoPost

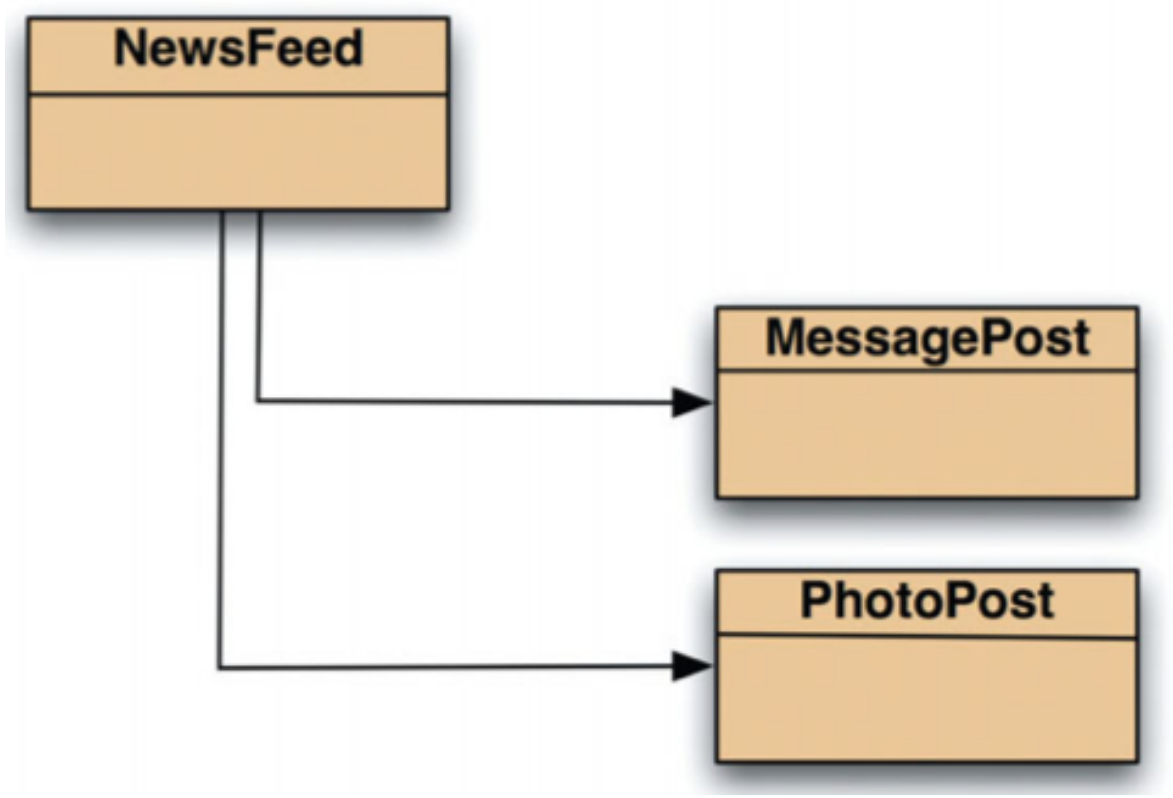


NB: nogle felter findes i ***begge klasser:***
(`username`, `likes`, `comments`)

MessagePost og PhotoPost

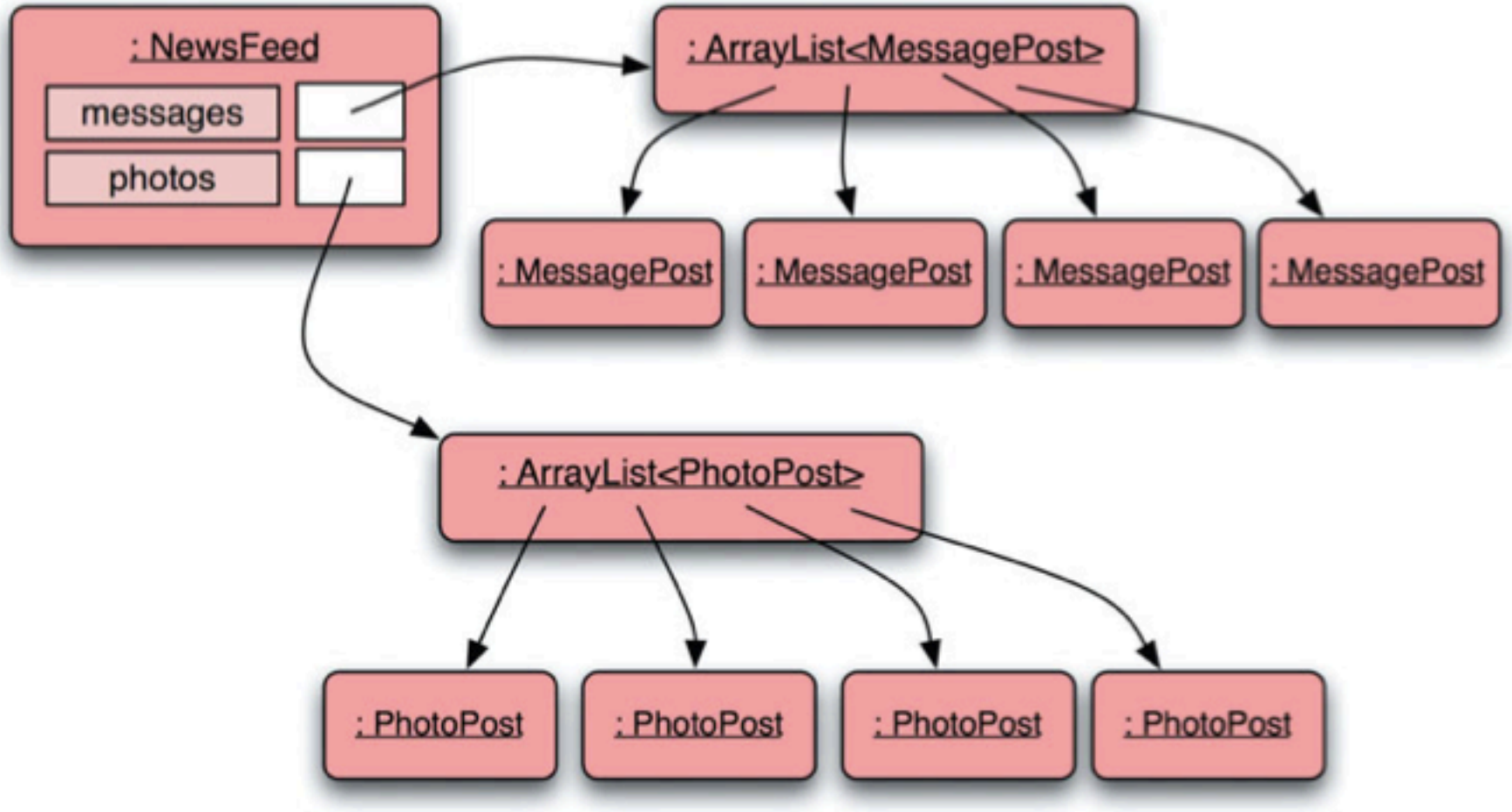


Klassediagram



- NewsFeed bruger MessagePost og PhotoPost

Objektdiagram
(4 MessagePosts & 4 PhotoPosts)



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

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>();
  }

  ...
```

MessagePost.java

```
...

public void like() {
  likes++;
}

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



public String getText() {
  return message;
}



public long getTimeStamp() {
  return ts;
}

...
```

PhotoPost.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";
    }
  }
}
```

MessagePost

Class MessagePost

java.lang.Object
  └ **MessagePost**

public class **MessagePost**extends 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 **PhotoPost**extends 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 post 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."*

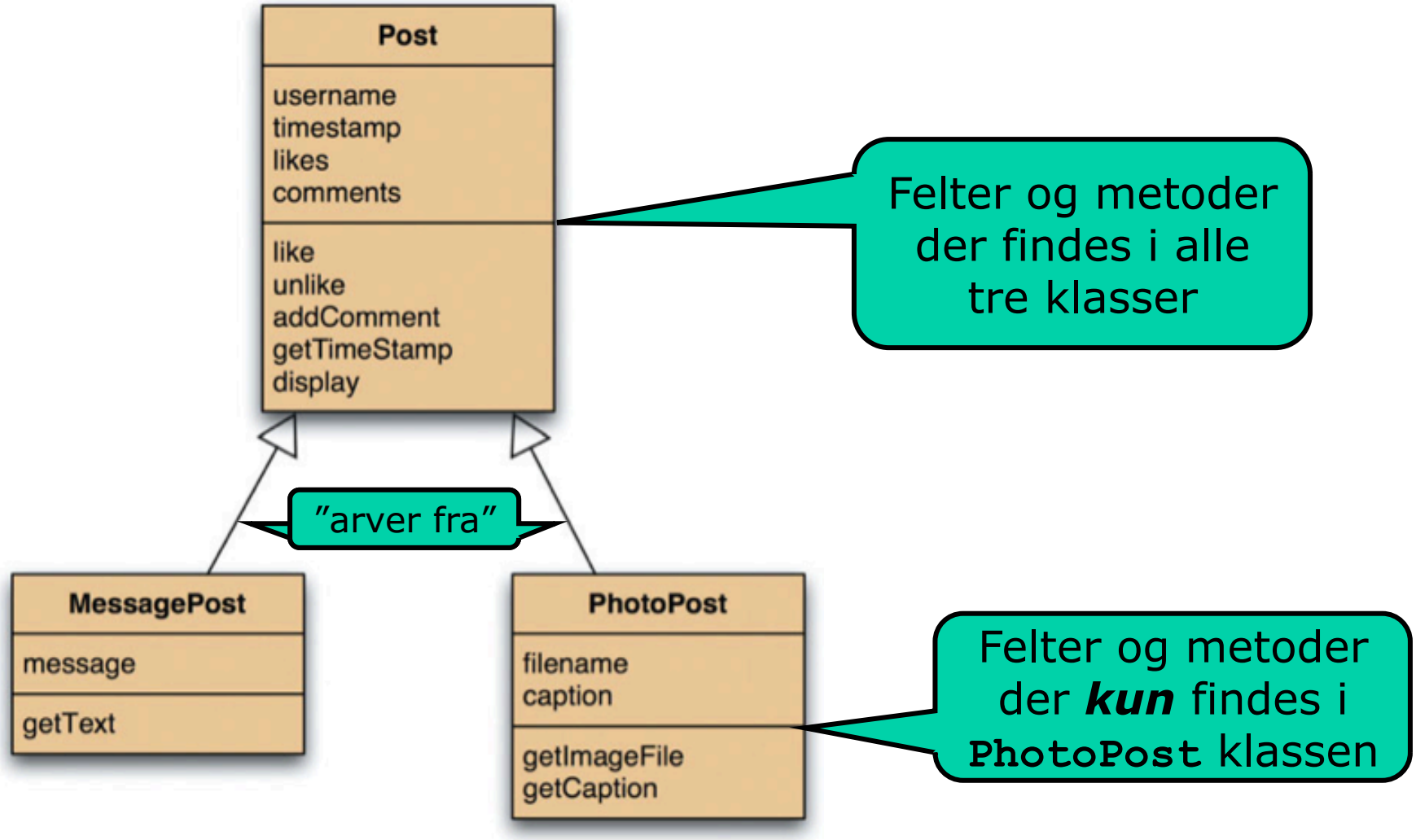
-- [Barnes & Kölling, p. 276]

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!)

OO Løsning: Arv (inheritance)

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



Ord og begreber

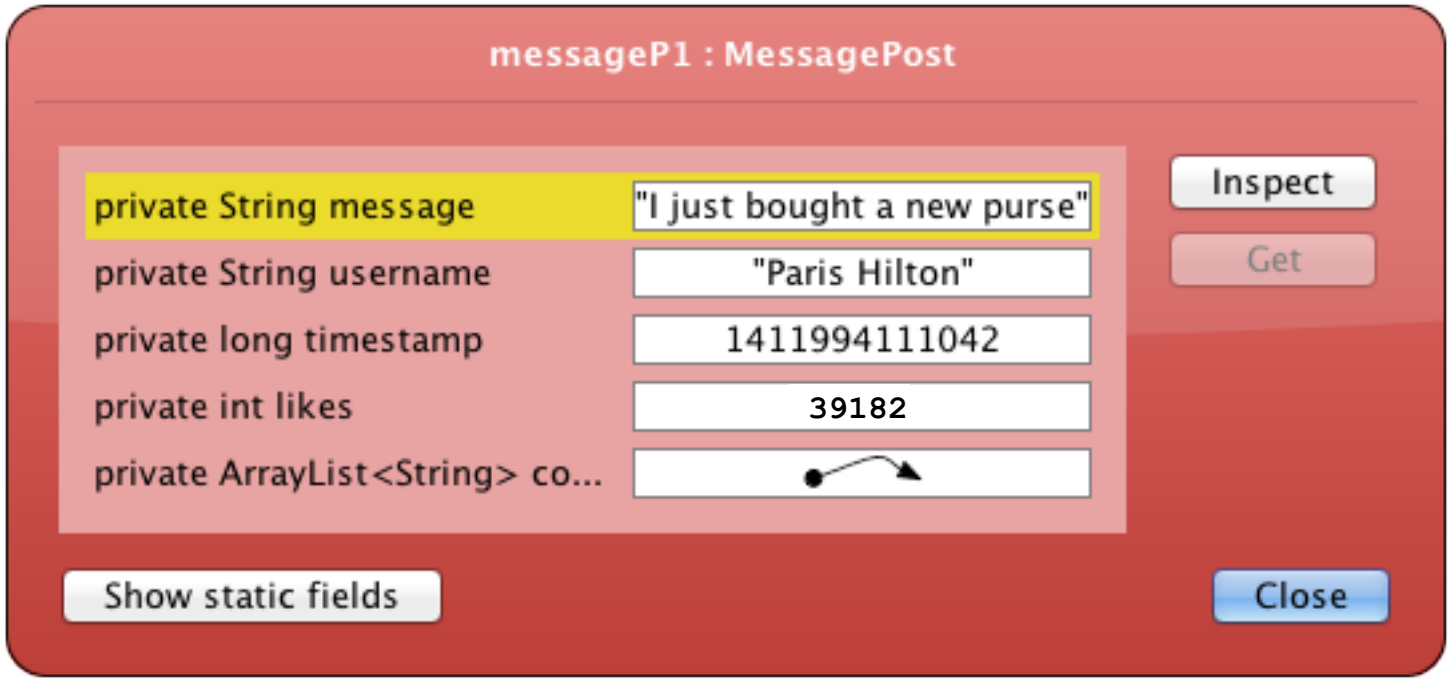
- Superklassen `Post`:
  - erklærer ***fælles*** felter og metoder
- Subklasserne `MessagePost` og `PhotoPost`:
  - ***arver alle*** superklassens felter og metoder
  - og ***erklærer egne*** særlige felter og metoder

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
- Superklasse = superclass = baseklasse
- Subklasse = subclass = derived class

OPGAVER

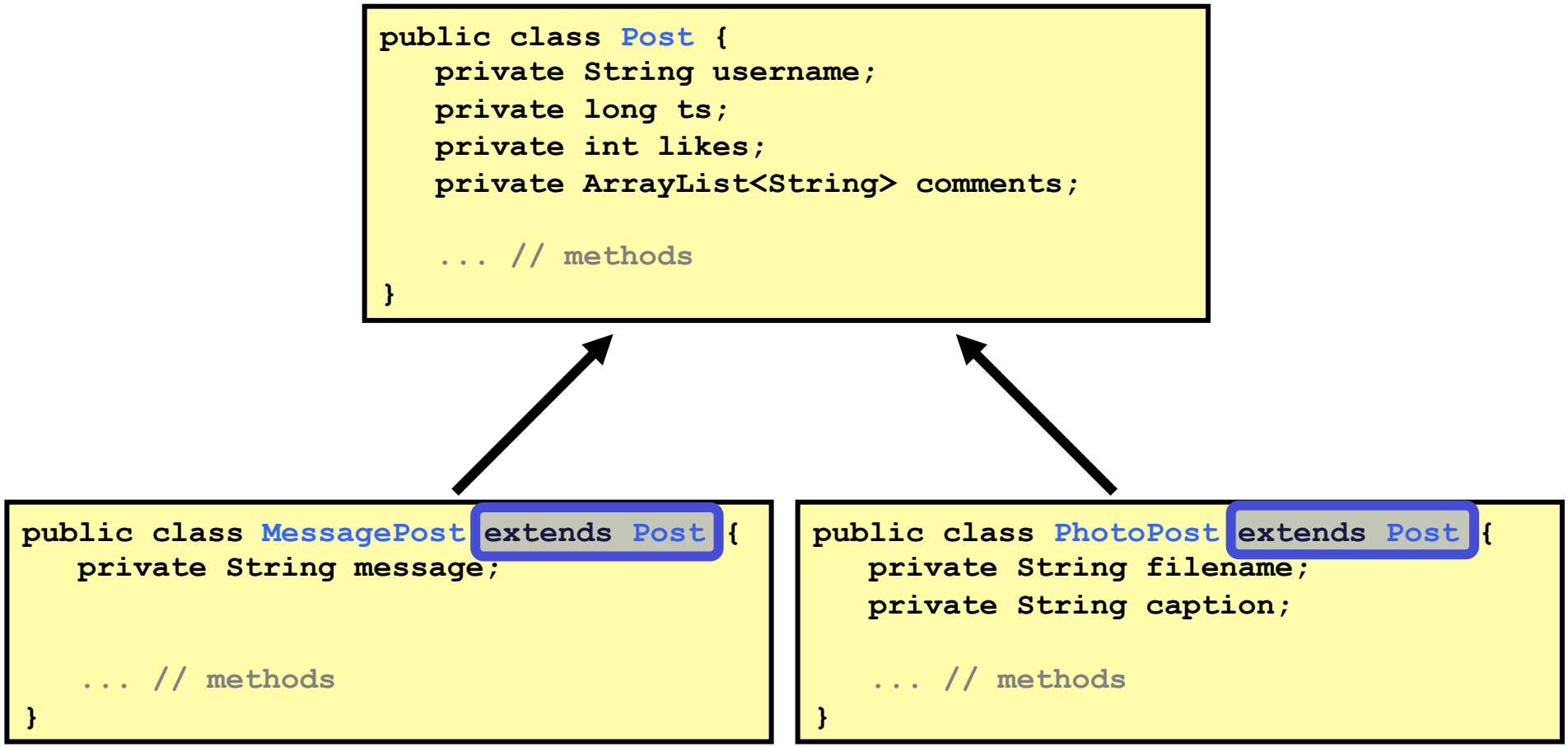
[B&K 8.9]:

- Order these concepts into an inheritance hierarchy:
  - apple, ice cream, bread, fruit, food-item, cereal, orange, dessert, chocolate mouse, baguette

[B&K 8.11]:

- Consider this: **`Rectangle -vs- Square`** (Sometimes things are more difficult than they first seem.) What are the arguments? Discuss.

Hvordan udtrykkes arv i Java



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

```
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;
  }
}
```

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!`

Bemærk: 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>();
  }
  ...
}
```

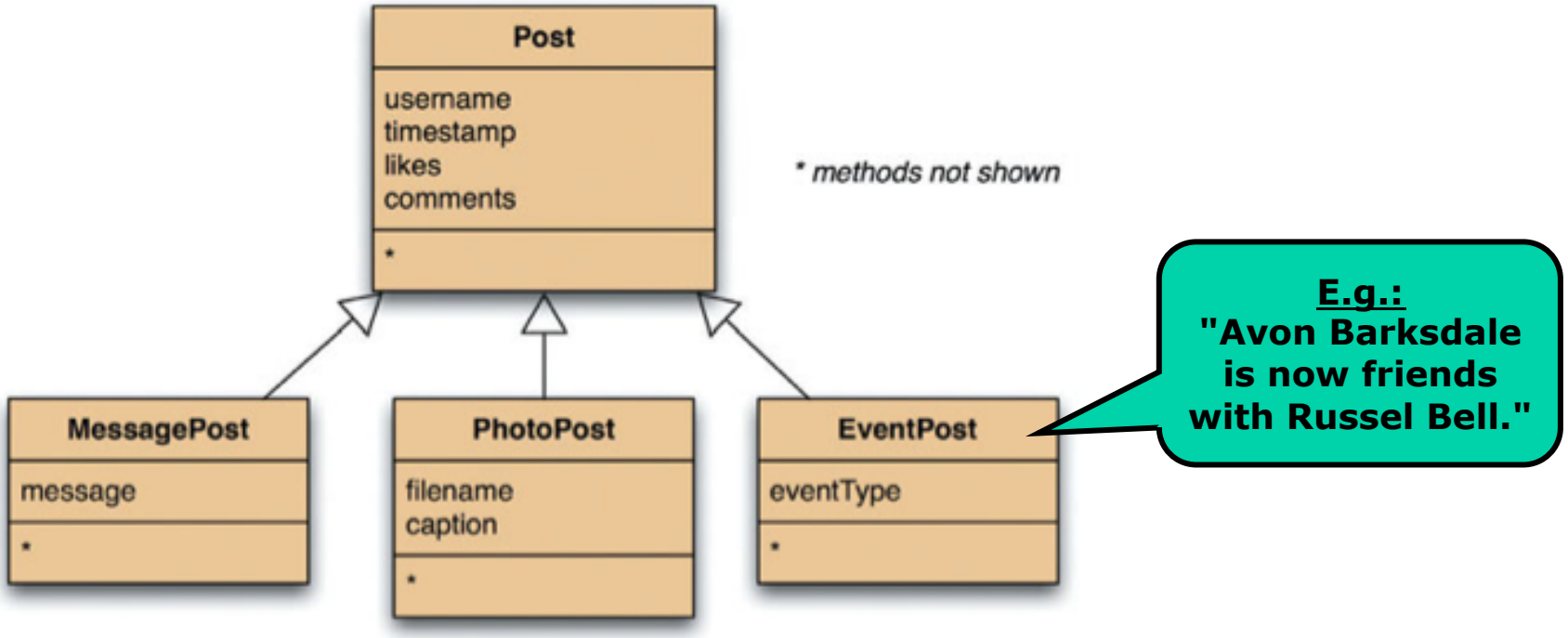
```
public class MessagePost extends Post {
  private String message;

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

Kalder konstruktor i superklassen

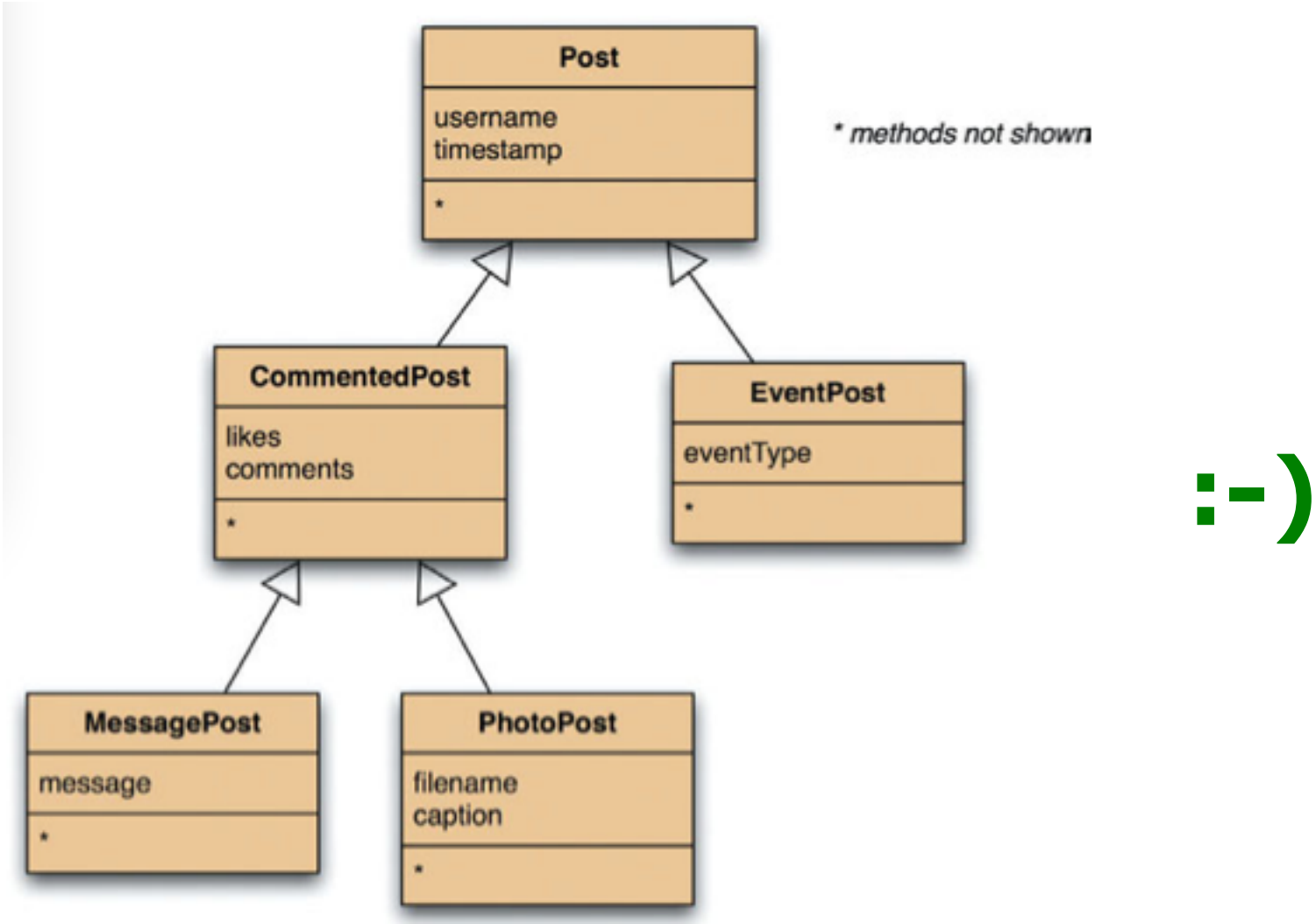
- Superklassens konstruktor skal kaldes
- Ellers indsættes automatisk **`super();`**

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

- **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)

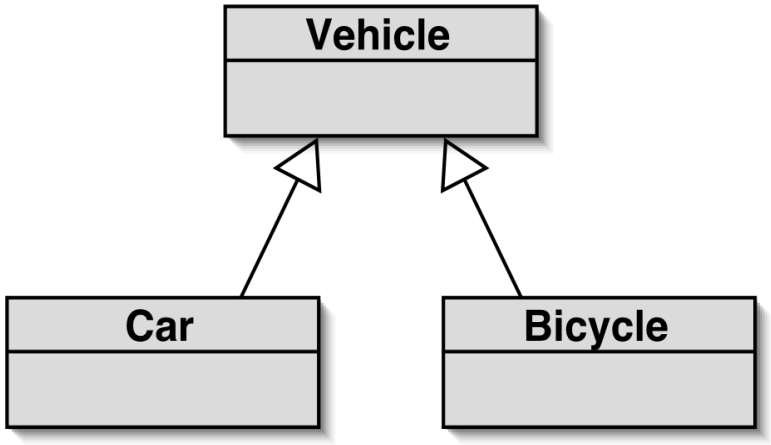
```
NewsFeed newsfeed = new NewsFeed();
MessagePost p1 = new MessagePost("Paris Hilton",              // author
                                 "I just bought a new purse"); // msg
newsfeed.addPost(p1); // add MessagePost!
PhotoPost p2 = new PhotoPost("Paris Hilton",                  // author
    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

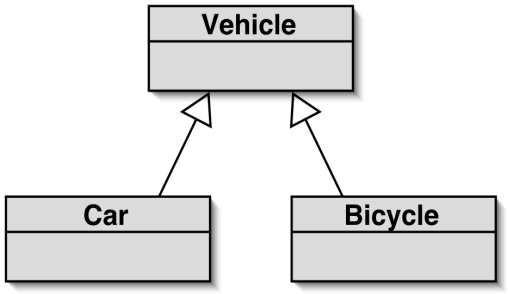
Liskov Substitution Principle:

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

-- Barbara Liskov, MIT, 1987

Variable og subklasser

- En variabel af type **`Vehicle`** kan:
  - være **`null`**
  - pege på et **`Vehicle`**-objekt
  - pege på et **`Bicycle`**-objekt
  - pege på et **`Car`**-objekt




```
Vehicle v;
Vehicle vb = new Bicycle();
Vehicle vc = new Car();
Car c = new Car();
v = null;
v = new Vehicle();
v = vb;
v = c;
```

- Følgende er ***ikke*** okay:

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

  - En bil ***er et*** køretøj (men et køretøj ***er ikke en*** bil)
  - Bilen '`c`' ville mangle noget (felter & metoder)

Variable og subklasser

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

```
NewsFeed newsfeed = new NewsFeed();
newsfeed.add(new MessagePost("Paris Hilton", "I just bought a new purse"));
newsfeed.add(new PhotoPost("Paris Hilton", "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`**":

Dynamisk cast

- Hvad hvis man har brug for det modsatte?

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

OK

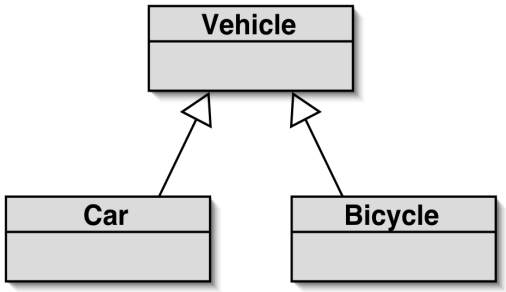
Ikke OK, compilefejl!

- Løsning: casting af **`v`** til type **`Car`**:

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

OK. Det tjekkes på køretid at **`v`** faktisk er af klasse **`Car`**

Casting (eksempler)



- Casting eksempler:

```
Vehicle v;
Car c;
Bicycle b;
c = new Car();
v = c;              // ok (substitutionsprincippet)!
b = (Bicycle) c; // compile-time error!
b = (Bicycle) v; // runtime error!
```

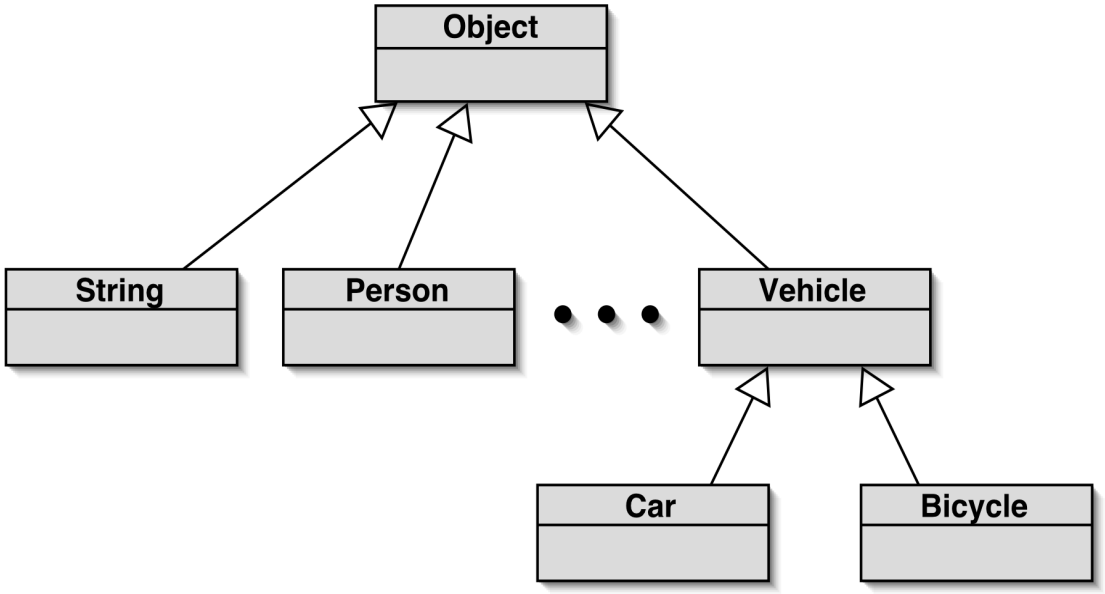
Klasse 'Object' (er superklasse for alle klasser)

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

svarer til

```
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:*

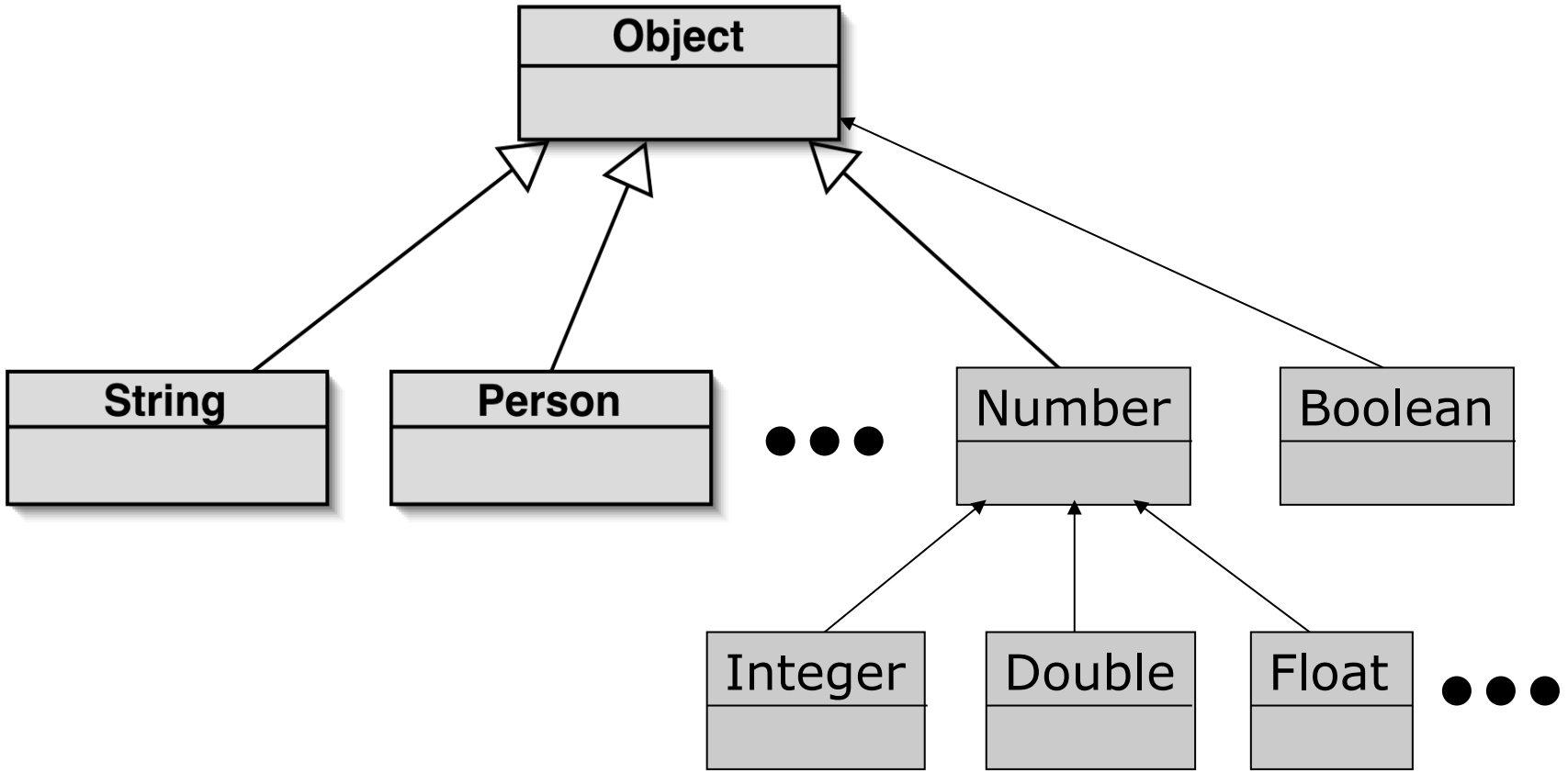
```
ArrayList<int> point;
point = new ArrayList<int>();
point.add(42);
```

Ulovligt

```
ArrayList<Integer> point;
point = new ArrayList<Integer>();
point.add(42);
```

OK

Wrapperklasserne er klassetyper



Tak!

Spørgsmål?