Software Development

Part 2: Inheritance

Koen Pelsmaekers

Unit Informatie (GT 03.14.05)

email: koen.pelsmaekers@kuleuven.be

Part 2a: introduction to inheritance

Examples from:
Objects First with Java, 6th edition
Chapter 10 & 11

Object-oriented programming

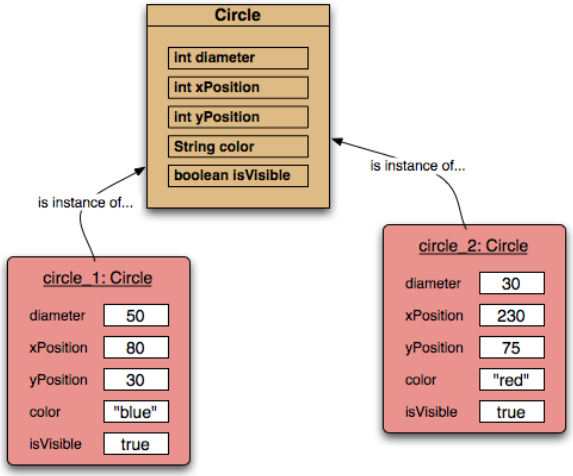
- Object-oriented programming pre-requisites:
  - Classes (and objects)
    - Properties (fields, *data*) and actions (methods, *behaviour*)
    - f.i. Person, Message, Contact, Circle, ...
  - Association/Aggregation/Composition
    - "has a" relationship
    - f.i. Person owns Cars
  - Inheritance and polymorphism
    - "is a" relationship, general or generic vs. specific
    - Polymorphic assignment
      - Subtype and Liskov's substitutability principle (polymorphic variable)
    - Polymorphic binding or dynamic binding or run-time binding or late binding
    - f.i. Student is a (special kind of) Person
      - Every Student is a Person, but not every Person is a Student

Classes and Objects

- Class: definition of properties and behaviour
  = (private) fields and methods
  + static fields and methods

  `public static void main(String[] args)`

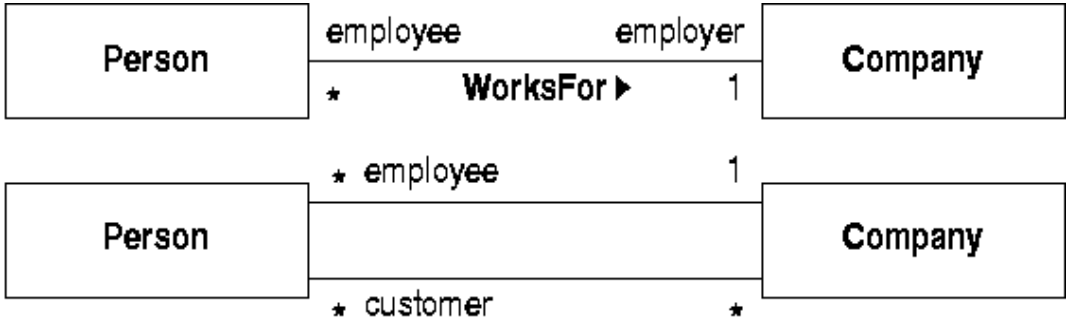
- Implements one abstraction
- Object: instance of class

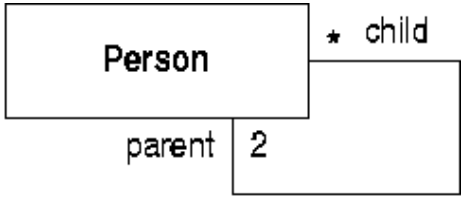



```
public class Circle
{
    private int diameter;
    private int xPosition;
    private int yPosition;
    private String color;
    private boolean isVisible;

    public Circle()
    {
        diameter = 68;
        xPosition = 230;
        yPosition = 90;
        color = "blue";
    }

    public void makeVisible()
    {
        isVisible = true;
        draw();
    }
...
}

...
Circle circle_1 = new Circle();
circle_1.draw();
...
```

Relations in UML class diagram: Association

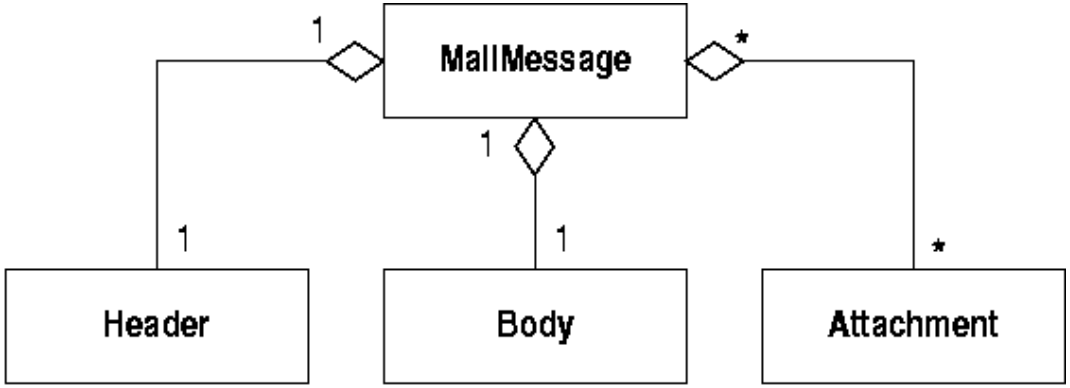
- labelled with a name (in the middle), f.i.: WorksFor
- association ends can be annotated with
  - a label, describing the role played by the class
  - multiplicity, showing how many instances an object at the other end can be linked to, f.i.:
    - a Person works for exactly one Company
    - a company has zero or more (*) people (= Person) working for it
  - an arrow, showing the direction or navigability
- most associations are binary
  - but: self-associations are possible too, f.i.: child – parent

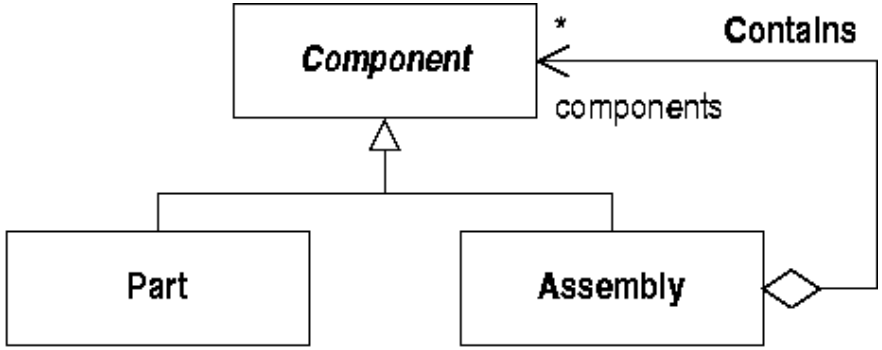




Relations in UML class diagram: Aggregation

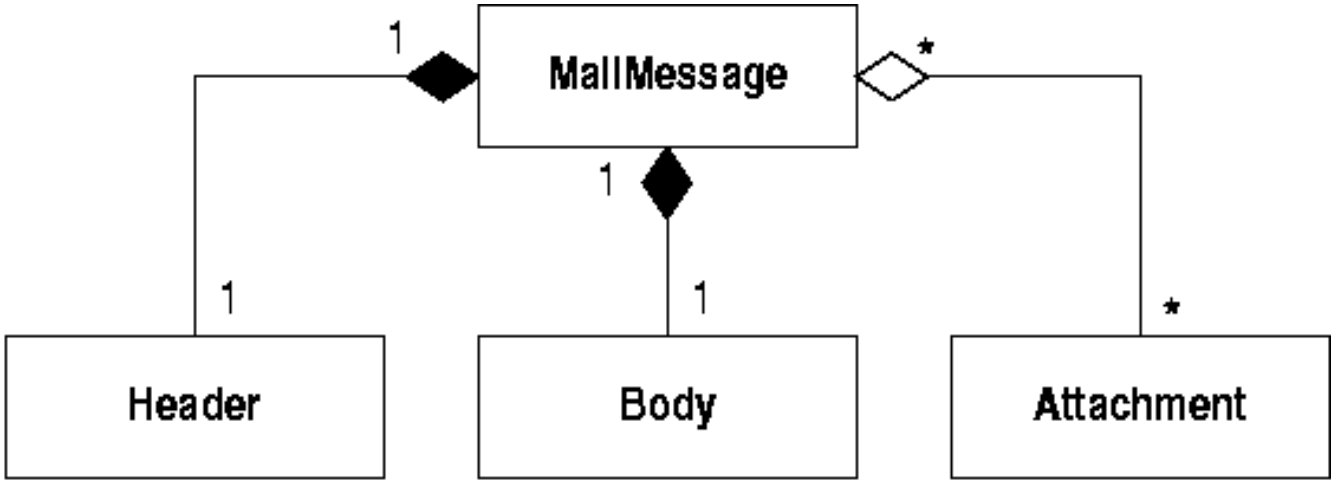
- "whole-part" relationship: Aggregation
  - a specialized form of an association
  - can have standard annotations on ends
  - f.i. composite design pattern



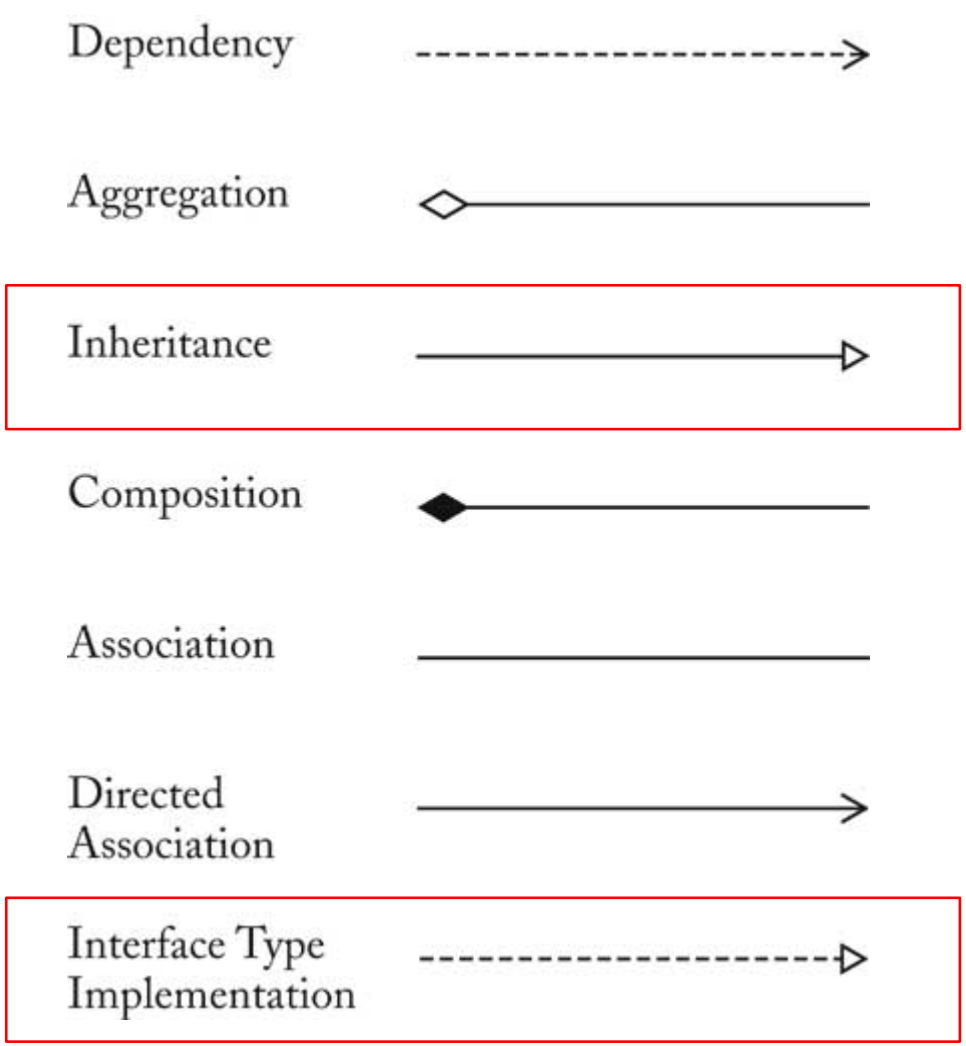


Relations in UML class diagram: Composition

- a strong form of aggregation: Composition
  - Parts can only belong to one composite at a time
  - Parts are destroyed when the composite is

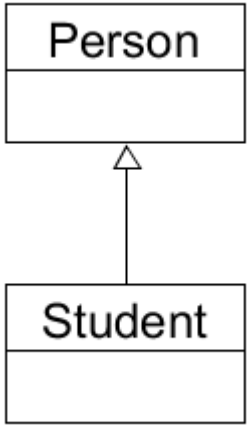


UML Class Diagram Connectors



Inheritance

- Generalization/Specialization
  - superclass (more generic) vs. subclass (more specific)
  - shared properties and behaviour
  - "is-a" relationship
  - Substitutability principle (Barbara Liskov)

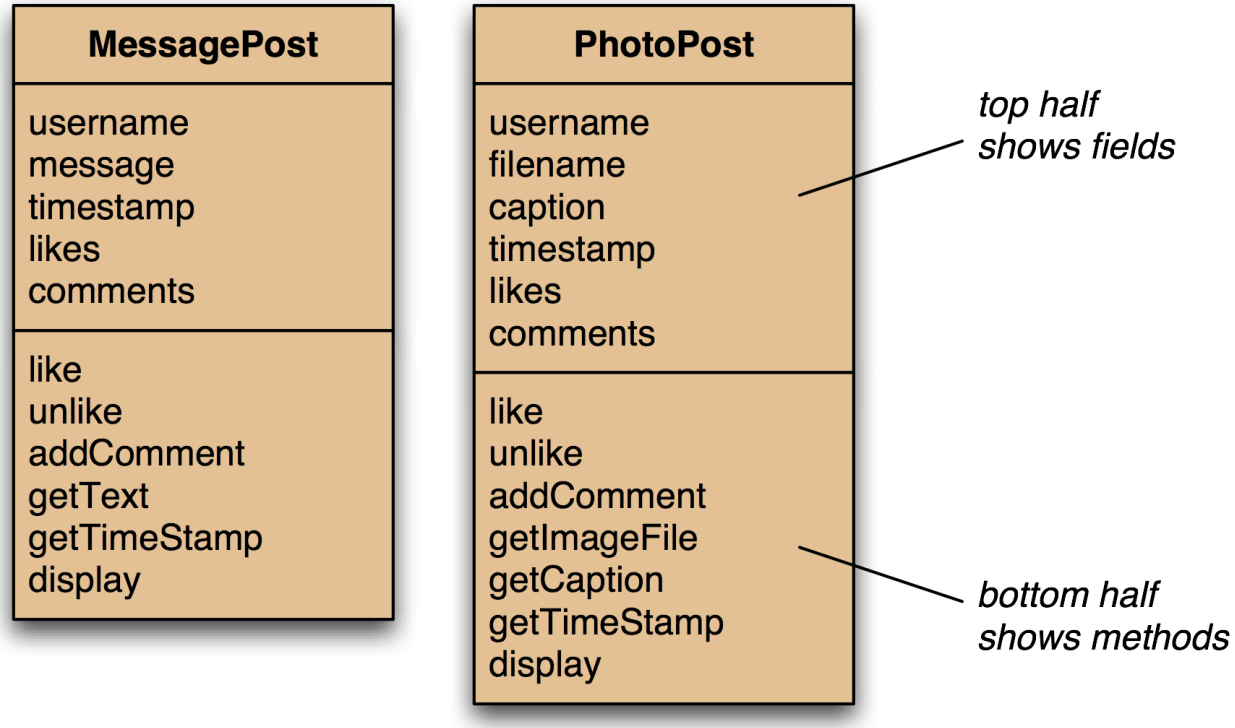



```
public class Person {
    ...
}

public class Student extends Person {
    ...
}

Person emp = new Person();
emp = new Student(); //substitutability
```

Network example (Objects First, chapter 10&11)

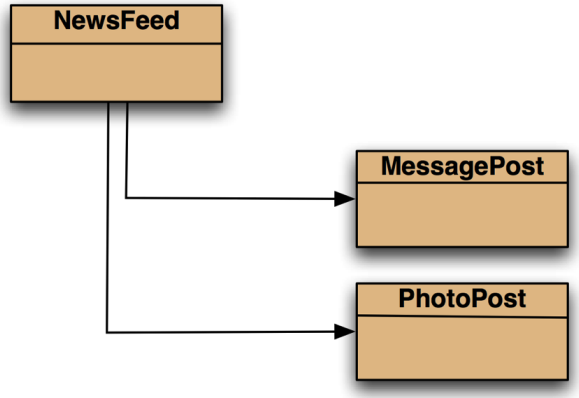
- News feeds with posts:
  - text posts
    - `MessagePost`: multi-line text message
  - photo posts
    - `PhotoPost`: photo and caption
- Operations on posts
  - add feed, show text and photo posts, search, ...

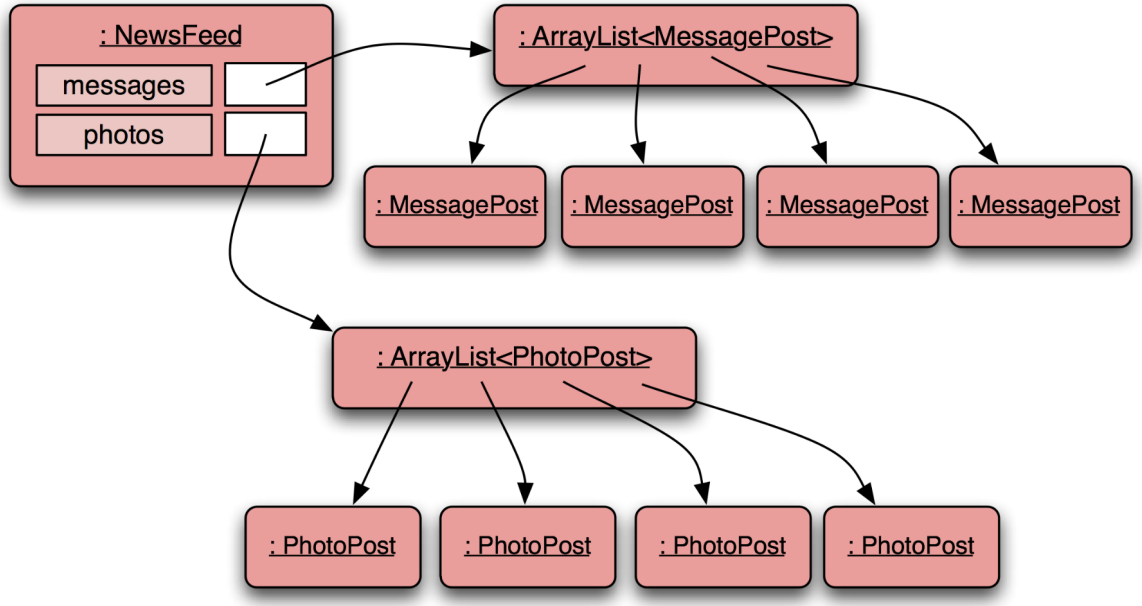
Network classes



Taken from: Objects First with Java, 6th edition

Network class diagram & object model





Taken from: Objects First with Java, 6th edition

MessagePost class – PhotoPost class

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

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

   public void addComment(String text) ...

   public void like() ...

   public void display() ...

   ...
}
```

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

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

   public void addComment(String text) ...
   public void like() …
   public void display() …
   ...
}
```

Taken from: Objects First with Java, 6th edition

NewsFeed class

```
public class NewsFeed
{
   private ArrayList<MessagePost> messages;
   private ArrayList<PhotoPost> photos;
   ...
   public void show()
   {
       for(MessagePost message : messages) {
          message.display();
          System.out.println();  // empty line between posts
       }

       for(PhotoPost photo : photos) {
          photo.display();
          System.out.println();  // empty line between posts
       }
   }
}
```

Taken from: Objects First with Java, 6th edition

Duplicate code...

- MessagePost and PhotoPost
  - maintenance problem
    - update algorithms twice
    - add new Post class (f.i. VideoPost)?
- NewsFeed
  - same code for the two collections

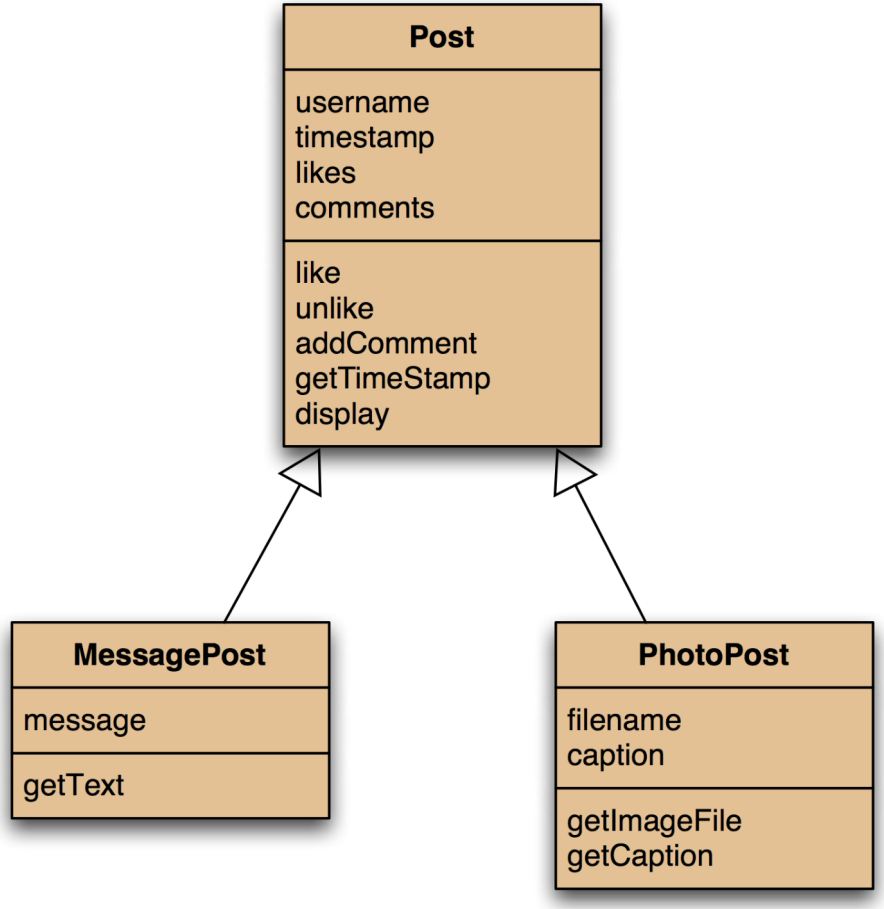
Solution: introduce superclass (inheritance)

superclass

more generic

more specific

subclass



Taken from: Objects First with Java, 6th edition

Inheritance

- Superclass: Post
  - common attributes: username, timestamp, likes, ...
  - common methods: like(), unlike(), addComment(), ...
- Subclasses: MessagePost and PhotoPost
  - inherits common attributes and common methods from superclass
  - adds specific methods and specific attributes
    - MessagePost: message, getText()
    - PhotoPost: filename, caption, getCaption(), ...

Superclass

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

    // constructor and methods omitted.
}
```

Taken from: Objects First with Java, 6th edition

Subclasses

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

    // constructor and methods omitted.
}
```

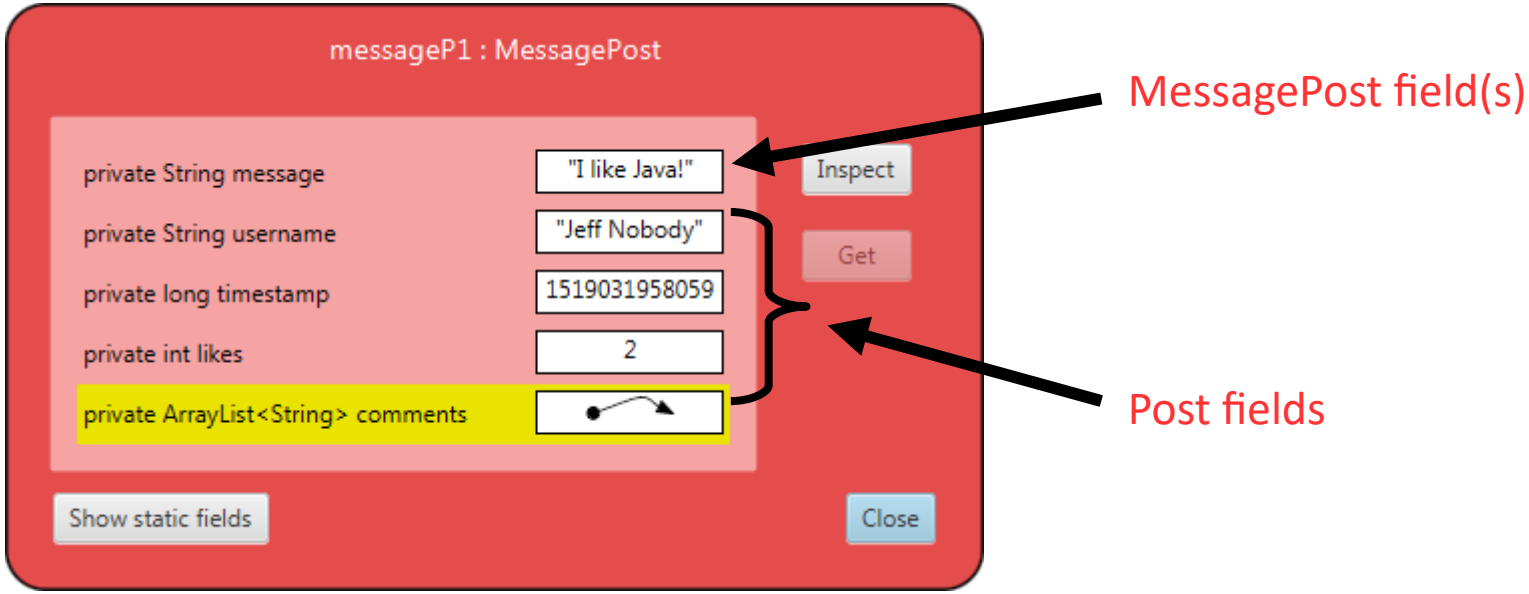
```
public class PhotoPost extends Post
{
    privat String filename;
    private String caption;

    // constructor and methods omitted.
}
```

Taken from: Objects First with Java, 6th edition

Part 2b: more about inheritance

Subclass has superclass fields/methods

- MessagePost has Post object (Post fields)
  - subclass constructor calls (implicitly) superclass constructor



Taken from: Objects First with Java, 6th edition

Inheritance and constructors

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

    /**
     * Initialise the fields of the post.
     */
    public Post(String author)
    {
        username = author;
        timestamp = System.currentTimeMillis();
        likes = 0;
        comments = new ArrayList<>();
    }

    // methods omitted
}
```

Taken from: Objects First with Java, 6th edition

Inheritance and constructors

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

    /**
     * Constructor for objects of class MessagePost
     */
    public MessagePost(String author, String text)
    {
        super(author);
        message = text;
    }

    // methods omitted
}
```

Superclass constructor call; must be first statement in subclass constructor

Taken from: Objects First with Java, 6th edition

Constructors in Java

- Implicit constructor (= no-args constructor)
  - in superclass
  - in subclass
- No implicit constructor if user-defined constructor exists
  - subclass constructor needed?
- Superclass constructor call in subclass <u>must be first statement</u> in subclass constructor

Inheritance: advantages

- Substitutability and dynamic method binding
  - See next slides
- Avoid code duplication
  - pull common code up to the super class
- Code reuse
  - reuse super class code
- Easier maintenance
- Extendibility
  - easy to add new post types

Substitutability

- Type and subtype
- Polymorphic assignment:

static type *dynamic type*

```
Post post = new MessagePost(...);
```

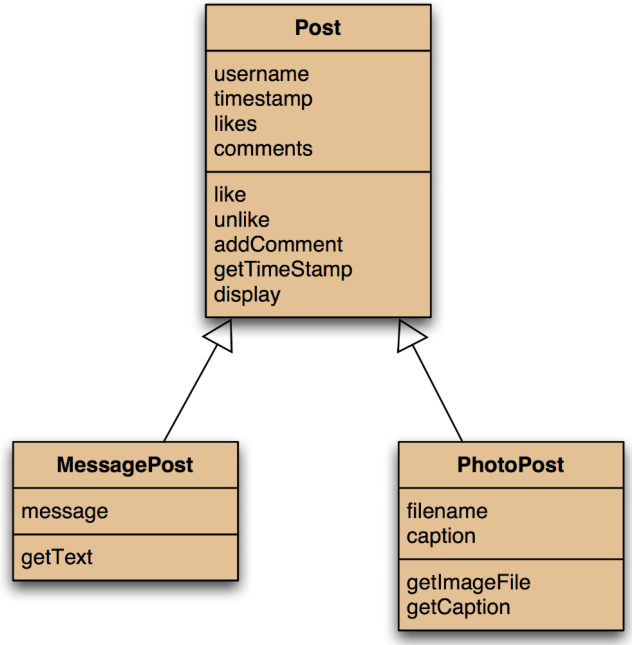
- Polymorphic or dynamic or run-time binding
  - uses dynamic type to bind method implementation:

```
post.display();
```

- Compile-time type or static type
  vs.
  Run-time type or dynamic type

Type casting

- Only necessary in rare cases
  - "down" cast
  - introduce inheritance?
- ClassCastException?
- instanceof operator




```
public void handlePost (Post p) {
    if (p instanceof MessagePost) {
        ((MessagePost)p).handleMessage();
    } else {
        ...
    }
}
```

Polymorphic collection/parameter

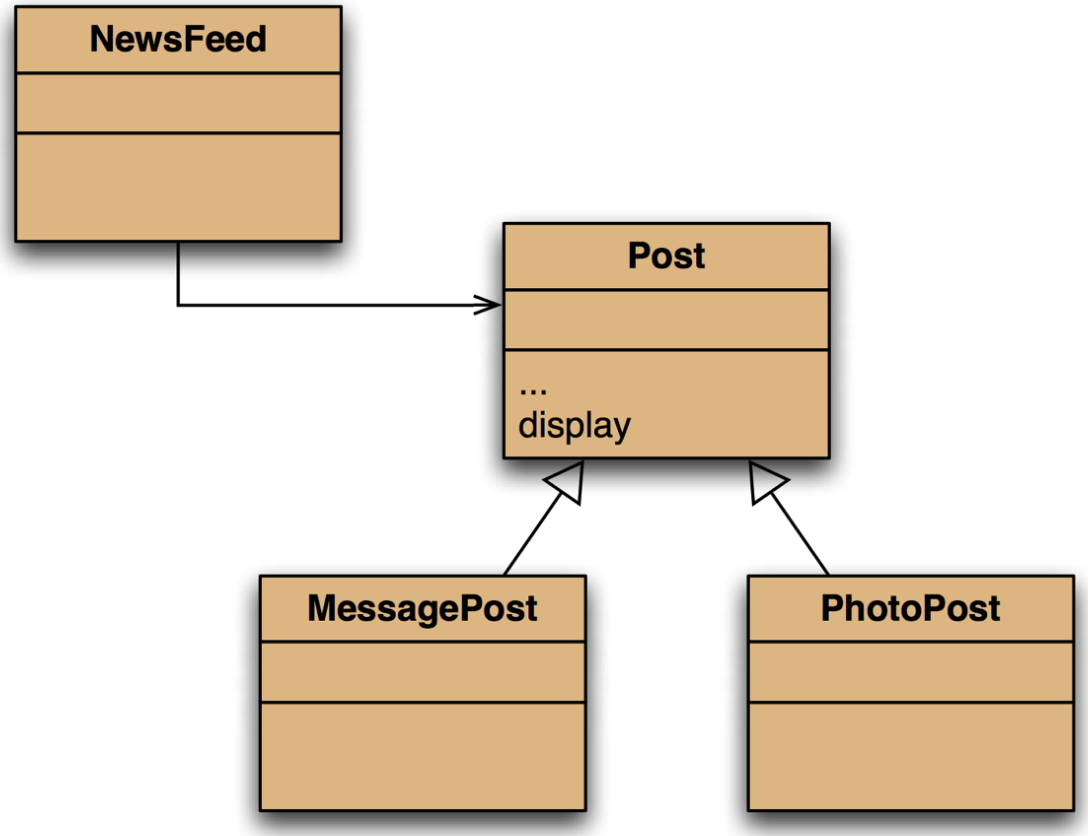
```
public class NewsFeed
{
    private ArrayList<Post> posts;

    /**
     * Construct an empty news feed.
     */
    public NewsFeed()
    {
        posts = new ArrayList<>();
    }

    /**
     * Add a post to the news feed.
     */
    public void addPost(Post post)
    {
        posts.add(post);
    }
    ...
}
```

polymorphic collection

polymorphic parameter

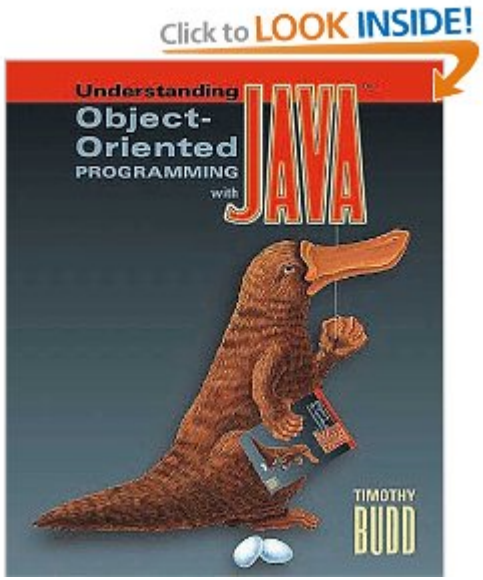
Taken from: Objects First with Java, 6th edition

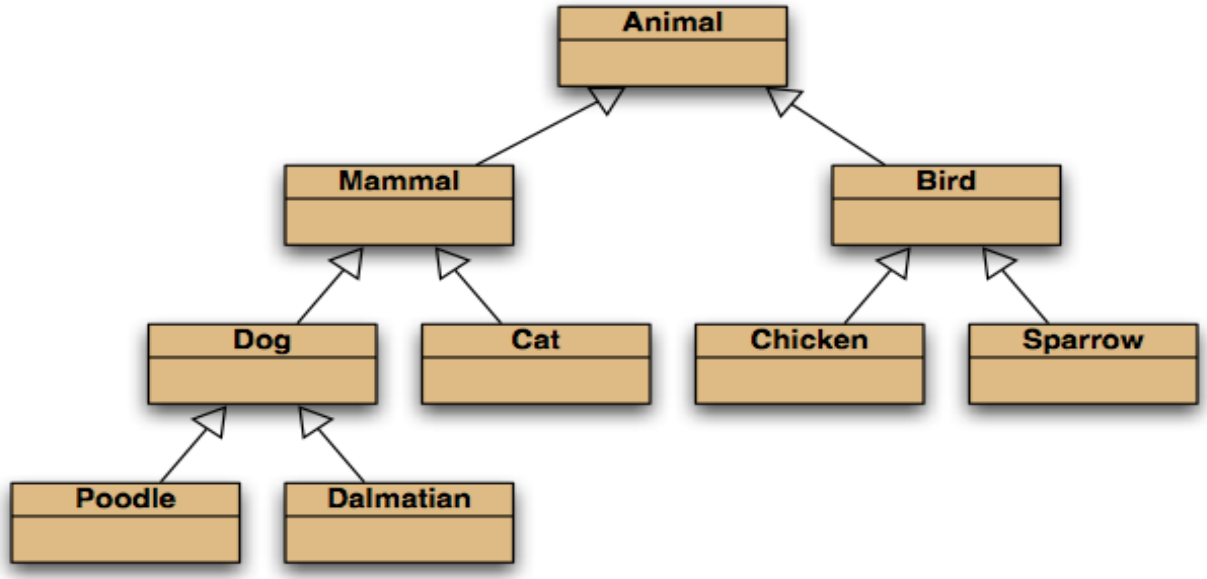
Newsfeed class diagram



Taken from: Objects First with Java, 6th edition

Method overriding



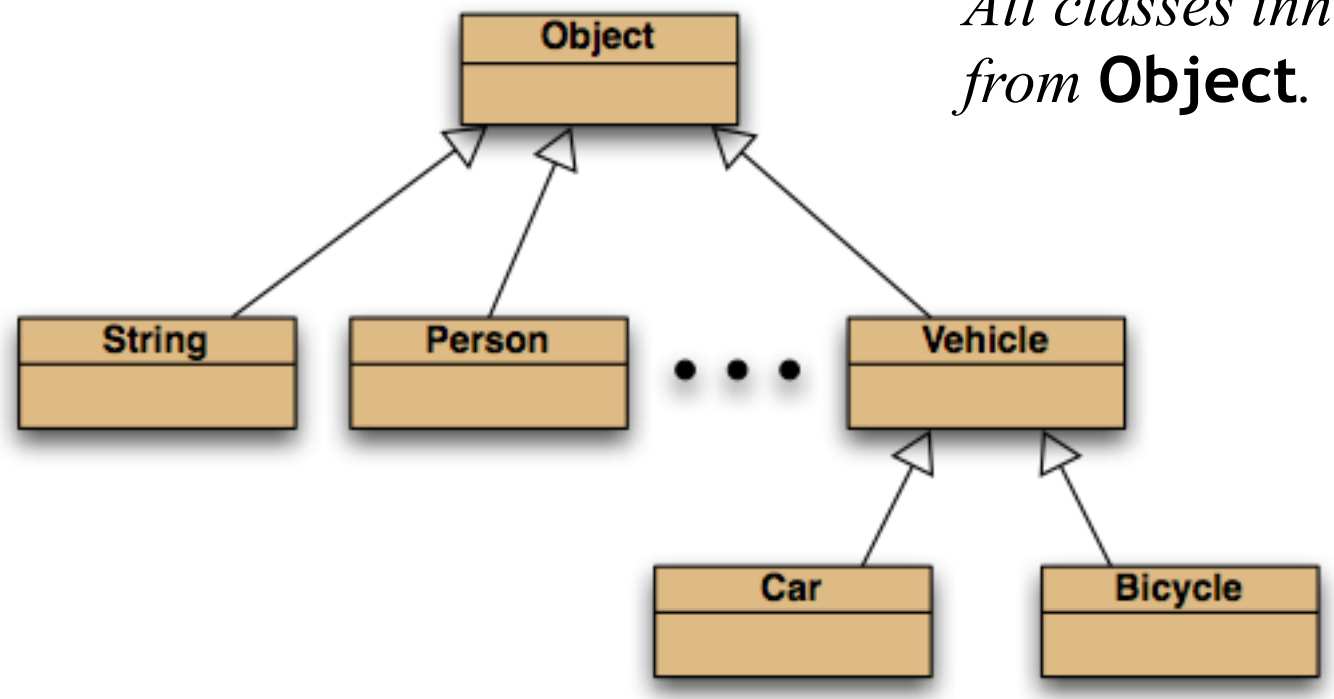


Taken from: Objects First with Java, 6th edition

class Object: superclass of all classes

- Java: single rooted hierarchy
- Single inheritance (<-> C++ multiple inheritance)

All classes inherit from **Object**.



Taken from: Objects First with Java, 6th edition

Method overriding

```
Leonardo da Vinci
Had a great idea this morning.
But now I forgot what it was. Something to do with flying ...
40 seconds ago - 2 people like this.
  No comments.

Alexander Graham Bell
[experiment.jpg]
I think I might call this thing 'telephone'.
12 minutes ago  -  4 people like this.
  No comments.
```

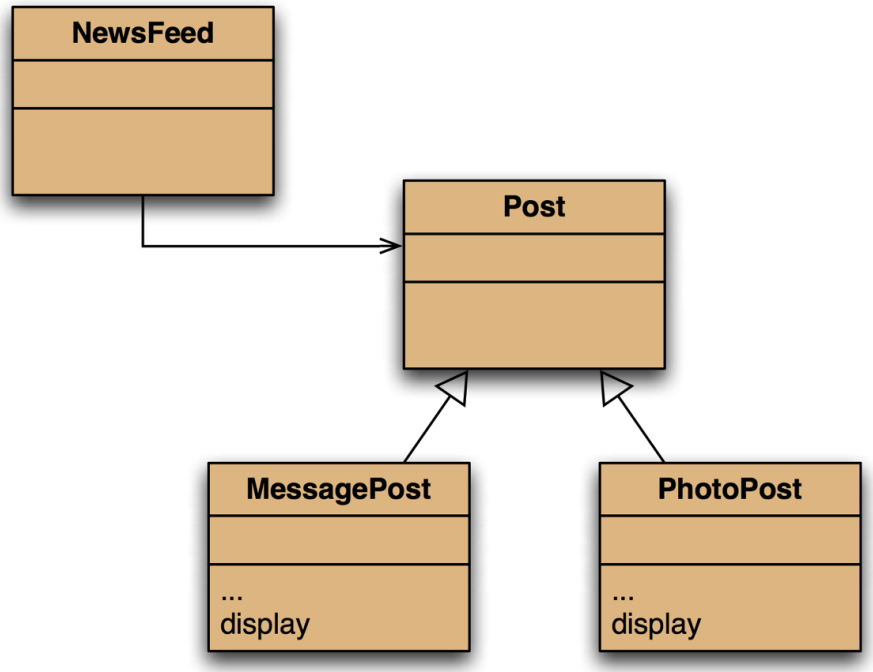
What we want

```
Leonardo da Vinci
40 seconds ago  -  2 people like this.
  No comments.

Alexander Graham Bell
12 minutes ago  -  4 people like this.
  No comments.
```

What we have

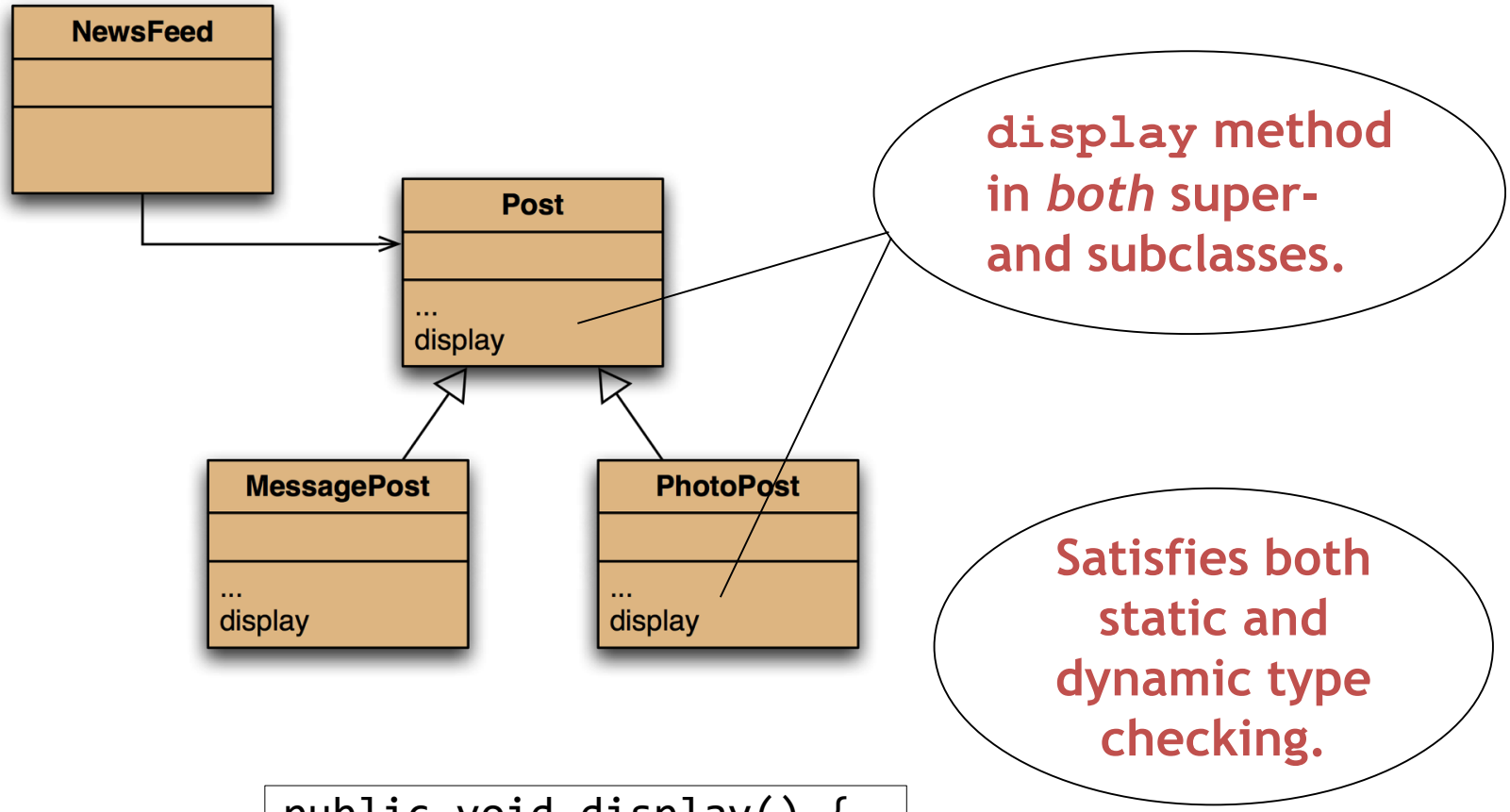
Taken from: Objects First with Java, 6th edition

Method overriding



Taken from: Objects First with Java, 6th edition

Method overriding




```
public void display() {
    ...
    super.display();
    ...
}
```

Taken from: Objects First with Java, 6th edition

Method overriding

- Superclass and subclass define methods with the same signature
- Each has access to the fields of its class
- Superclass satisfies static type check
- Subclass method is called at runtime – it overrides the superclass version

```
Post post = new MessagePost("Jeff", "I like Java");
post.display(); // dynamic binding
```

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

@Override annotation

- Compiler directive
  - To inform the compiler about your intent to override a method
  - not obligatory
- Goal
  - Compile-time check
  - Improve readability of your code

super call

- Overriding hides super class method
  - use "super" to call the super class method:

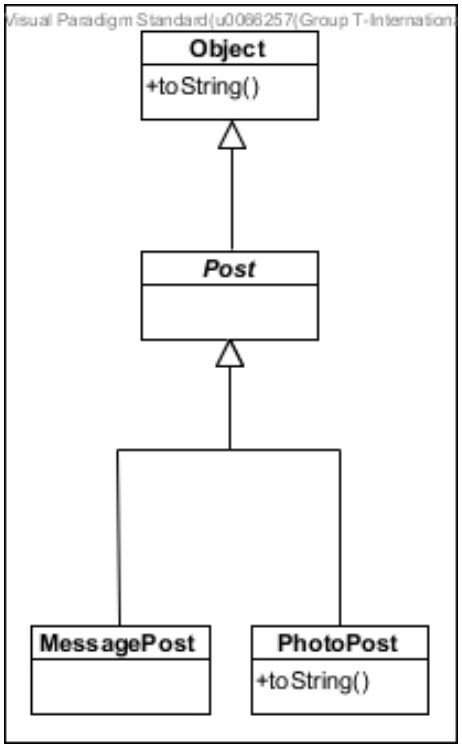
```
...
super.method();
...
```

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

Override Object class methods

- Useful methods in class Object
  - toString()
    - commonly overridden to return a String representation of an object
    - the default implementation (`"classname@hashCode()"`) is not particular useful
  - equals() & hashcode()
    - see later: hashtable based collections
  - clone()
    - see later: create a deep or shallow copy of an object

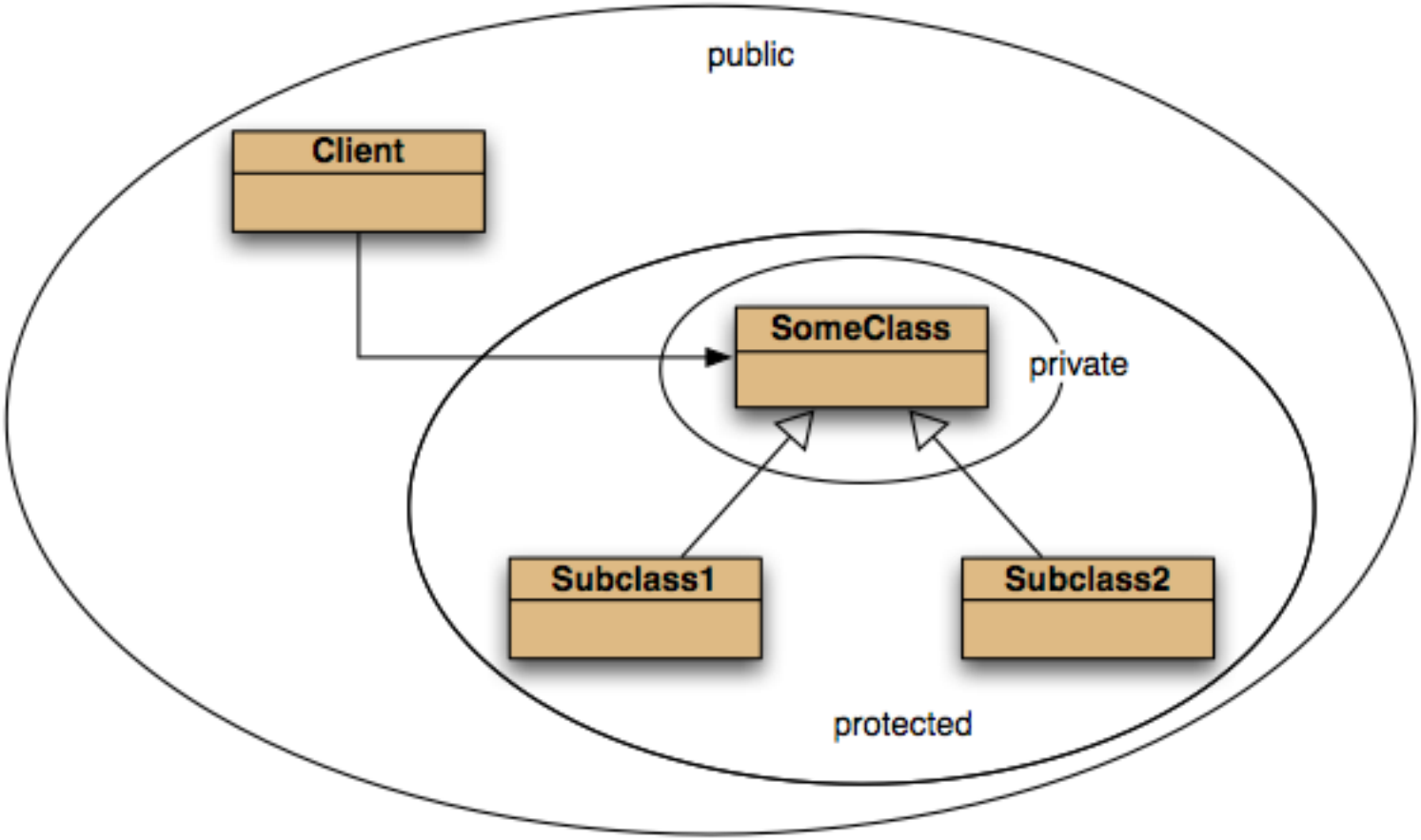
Exercise: compile-time vs. run-time




```
Post post = new MessagePost("Nobody","Java rules!");
System.out.println(post.toString()); //1
post = new PhotoPost("An", "world.jpg", "Hello world!");
System.out.println(post.toString()); //2
```

- Will this code compile?
- What is the output?
  - toString() is implemented in
    - Object (= the java.lang.Object)
    - PhotoPost
  - toString() is not implemented in
    - Post (= an abstract class)
    - MessagePost

public/private/protected



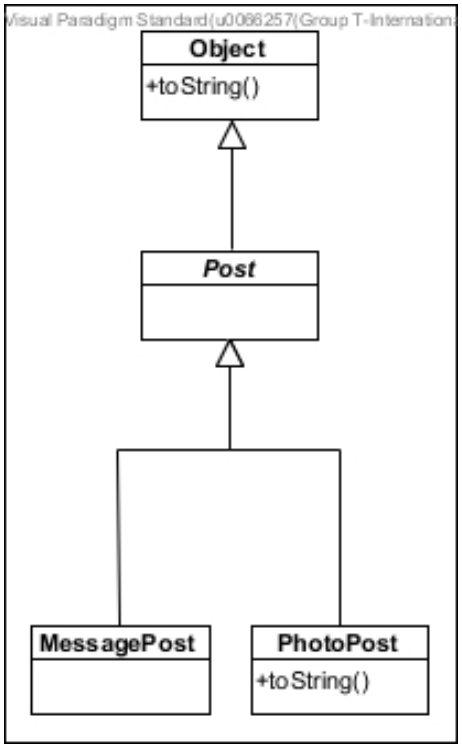
Taken from: Objects First with Java, 6th edition

Abstract class and methods

- abstract: useful for superclass
  - cannot be instantiated
  - can have abstract methods
- abstract method
  - no body
  - concrete subclass has to complete the implementation, otherwise the subclass has to be defined abstract
- used for keeping common fields and methods in class hierarchy (f.i. Post class: a (generic) Post object cannot exist – makes no sense)

```
public abstract Post {
   …
}
```

Exercise



- How to add a method "handlePost()" that acts different for both kind of concrete posts?
  - This method will print:
    - "I am handling a MessagePost" or
    - "I am handling a PhotoPost"
  - There is no "handlePost()" method in class Post nor in class Object
  - Class Post has no useful implementation for the method "handlePost()
- Use dynamic binding to avoid switch/case statements with type-checking