Improving Structure with Inheritance

Java version

Objectives

To introduce two important object-oriented programming concepts: *inheritance* and *polymorphism*.

Main concepts discussed in this chapter

- inheritance
- substitution
- subtyping
- polymorphic variables

Resources

Classes needed for this lab - *chapter10.jar*.

To do

The Network example

The *Network* project is the start of a simple social network, a system for storing and displaying news in the form of textual message posts and photos. The system should include the following functionality:

- It should allow users to enter messages and upload photos (actually just an image file name and a photo caption).
- It should store this information permanently so that it can be used later.
- It should provide a search function that allows us to find, eg, all messages in the database by a certain user, or all photos within a given range of dates.
- It should allow us to print lists, eg, a list of all photos in the database, or a list of all messages.
- It should allow us to remove information.

For each message we need to store:

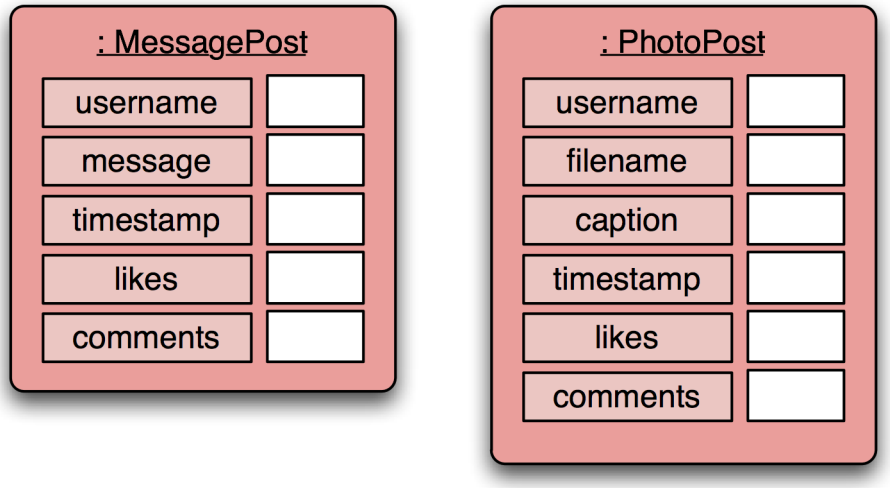
- the user name of the author;
- the message text;
- a time stamp;
- the number of like-its;
- comments by other users on the post.

For each photo we need to store:

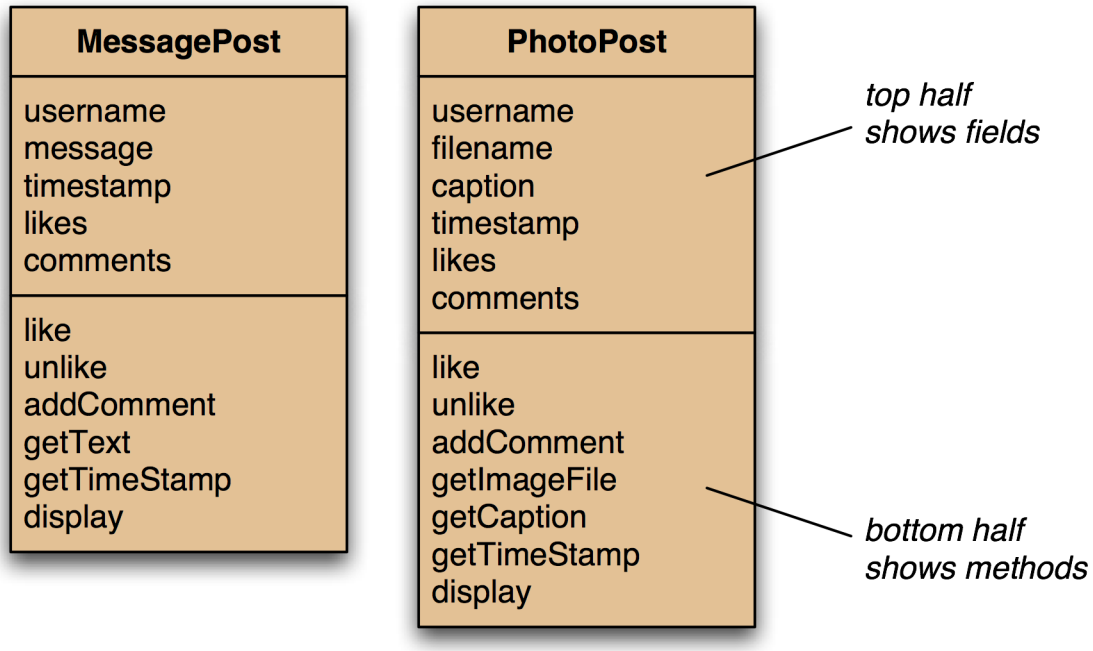
- the user name of the author;
- the file name of the image;
- the image caption;
- a time stamp;
- the number of like-its;
- comments by other users on the post.

The *network* project: classes and objects

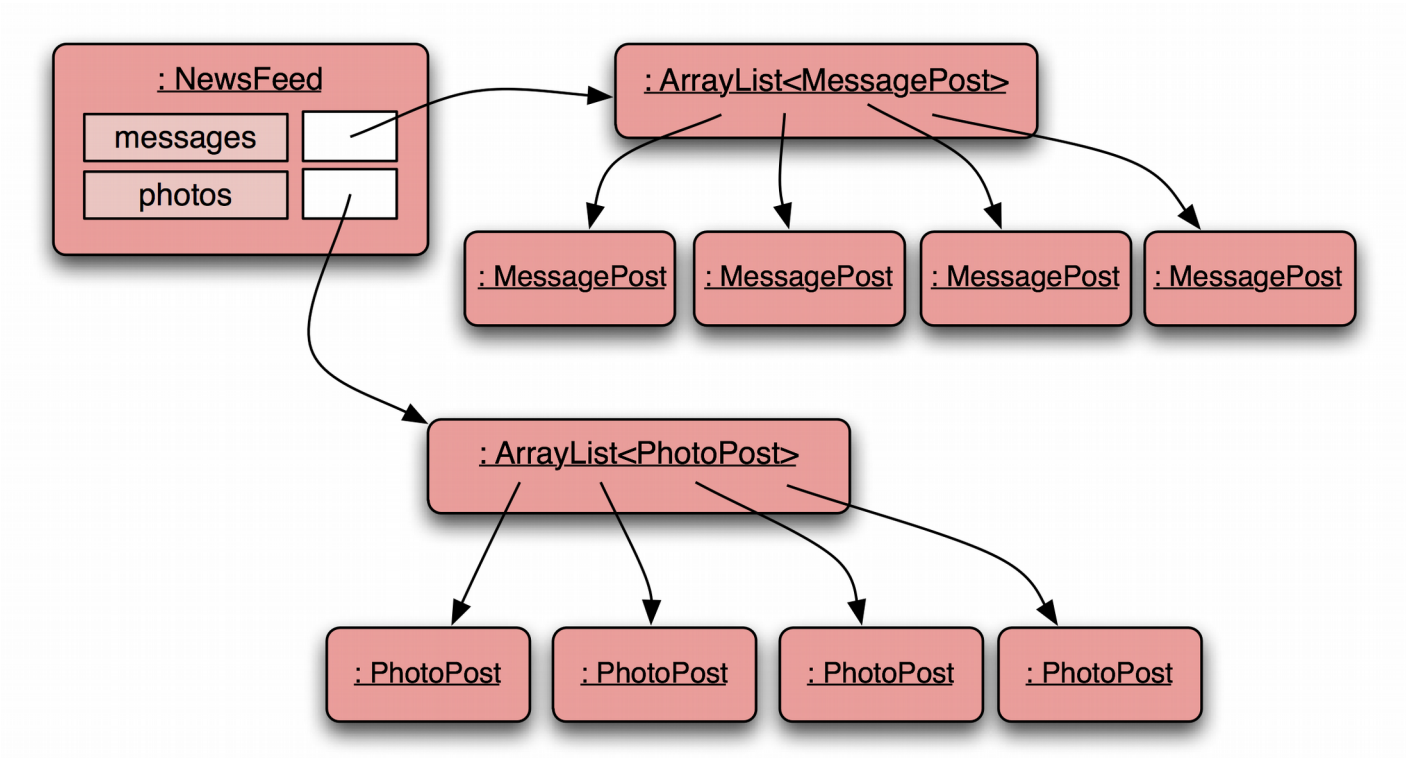
We first need to decide what classes to use to model the problem. It seems reasonable that we'll have at least two classes: **`MessagePost`** and **`PhotoPost`**. Objects of these classes will store the information from the lists above:



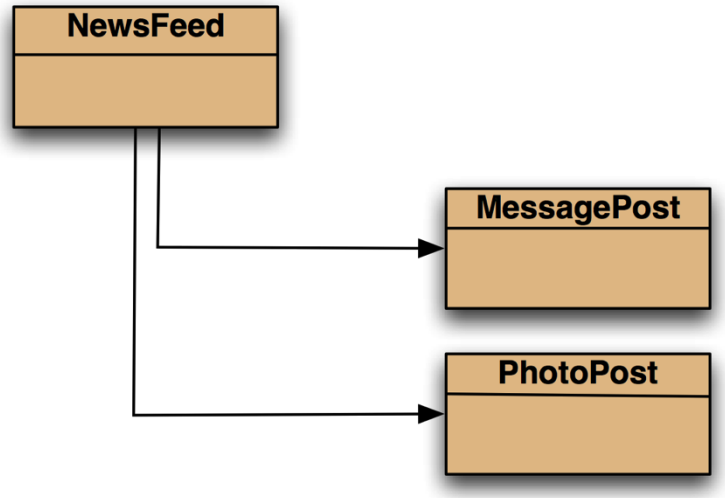
This information can be shown in a *UML class diagram*:



We have accessor and mutator methods (getter and setter methods) for the instance variables that are expected to change over time; the other instance variables are assumed to be set in the constructor. Once **`MessagePost`** and **`PhotoPost`** classes are defined, objects of that type can be created. However, we still need a **`NewsFeed`** class to store the collection of messages and the collection of photos that we create. If each collection is stored in an **`ArrayList`**, the object diagram could look like



and the UML class diagram



Classes from the standard Java library are omitted from the UML diagram, as are some classes that we'll only need later.

Network source code

The **`MessagePost`** and **`PhotoPost`** classes are straightforward, having getter and setter methods for those instance variables which are expected to change. The **`display`** method displays information about a stored object.

```
package network.v1;

import java.util.ArrayList;

/**
 * This class stores information about a post in a social network.
 * The main part of the post consists of a (possibly multi-line)
 * text message. Other data, such as author and time, are also stored.
 *
 * @author Michael Kölling and David J. Barnes
 * @version 0.1
 */
public class MessagePost {
    private final String username;  // username of the post's author
    private final String message;   // an arbitrarily long,
                                    //  multi-line message
    private final long timestamp;
    private int likes;
    private final ArrayList<String> comments;
```

```
/**
 * Constructor for objects of class MessagePost.
 *
 * @param author    The username of the author of this post.
 * @param text      The text of this post.
 */
public MessagePost(String author, String text) {
    username = author;
    message = text;
    timestamp = System.currentTimeMillis();
    likes = 0;
    comments = new ArrayList<>();
}

/**
 * Record one more 'Like' indication from a user.
 */
public void like() {
    likes++;
}

/**
 * Record that a user has withdrawn his/her 'Like' vote.
 */
private void unlike() {
    if (likes > 0) {
        likes--;
    }
}

/**
 * Add a comment to this post.
 *
 * @param text  The new comment to add.
 */
private void addComment(String text) {
    comments.add(text);
}

/**
 * Return the text of this post.
 *
 * @return The post's text.
 */
private String getText() {
    return message;
}

/**
 * Return the time of creation of this post.
 *
 * @return The post's creation time, as a system time value.
 */
```

```
    private long getTimeStamp() {
        return timestamp;
    }

    /**
     * Display the details of this post.
     *
     * (Currently: Print to the text terminal. This is simulating display
     * in a web browser for now.)
     */
    void display() {
        System.out.println(username);
        System.out.println(message);
        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).
Click here to view.");
        }
    }

    /**
     * Create a string describing a time point in the past in terms
     * relative to current time, such as "30 seconds ago"
     * or "7 minutes ago".
     * Currently, only seconds and minutes are used for the string.
     *
     * @param time  The time value to convert (in system milliseconds)
     * @return      A relative time string for the given time
     */

    private final String timeString(long time) {
        long current = System.currentTimeMillis();
        long pastMillis = current - time;  // time passed in milliseconds
        long seconds = pastMillis/1000;
        long minutes = seconds/60;
        if (minutes > 0) {
            return minutes + " minutes ago";
        } else {
            return seconds + " seconds ago";
        }
    }
}
```

The **PhotoPost** class below looks very very similar to the above **MessagePost** class, with

only minor differences in the details.

```
package network.v1;

import java.util.ArrayList;

/**
 * This class stores information about a post in a social network.
 * The main part of the post consists of a photo and a caption.
 * Other data, such as author and time, are also stored.
 *
 * @author Michael Kölling and David J. Barnes
 * @version 0.1
 */
public class PhotoPost {
    private final String username;  // username of the post's author
    private final String filename;  // the name of the image file
    private final String caption;   // a one line image caption
    private final long timestamp;
    private int likes;
    private final ArrayList<String> comments;

    /**
     * Constructor for objects of class PhotoPost.
     *
     * @param author    The username of the author of this post.
     * @param filename  The filename of the image in this post.
     * @param caption   A caption for the image.
     */
    public PhotoPost(String author, String filename, String caption) {
        username = author;
        this.filename = filename;
        this.caption = caption;
        timestamp = System.currentTimeMillis();
        likes = 0;
        comments = new ArrayList<>();
    }

    /**
     * Record one more 'Like' indication from a user.
     */
    public void like() {
        likes++;
    }

    /**
     * Record that a user has withdrawn his/her 'Like' vote.
     */
    private void unlike() {
        if (likes > 0) {
            likes--;
        }
    }
```

```
/**
 * Add a comment to this post.
 *
 * @param text  The new comment to add.
 */
private void addComment(String text) {
    comments.add(text);
}

/**
 * Return the file name of the image in this post.
 *
 * @return The post's image file name.
 */
private String getImageFile() {
    return filename;
}

/**
 * Return the caption of the image of this post.
 *
 * @return The image's caption.
 */
private String getCaption() {
    return caption;
}

/**
 * Return the time of creation of this post.
 *
 * @return The post's creation time, as a system time value.
 */
private long getTimeStamp() {
    return timestamp;
}

/**
 * Display the details of this post.
 *
 * (Currently: Print to the text terminal. This is simulating display
 * in a web browser for now.)
 */
void display() {
    System.out.println(username);
    System.out.println("  [" + filename + "]");
    System.out.println("  " + caption);
    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). Click here to view.");
        }
    }

    /**
     * Create a string describing a time point in the past in terms
     * relative to current time, such as "30 seconds ago"
     * or "7 minutes ago".
     * Currently, only seconds and minutes are used for the string.
     *
     * @param time  The time value to convert (in system milliseconds)
     * @return      A relative time string for the given time
     */

    private final String timeString(long time) {
        long current = System.currentTimeMillis();
        long pastMillis = current - time;  // time passed in milliseconds
        long seconds = pastMillis/1000;
        long minutes = seconds/60;
        if (minutes > 0) {
            return minutes + " minutes ago";
        } else {
            return seconds + " seconds ago";
        }
    }
}
```

The **NewsFeed** code is pretty simple as well.

```
package network.v1;

import java.util.ArrayList;

/**
 * The NewsFeed class stores news posts for the news feed in a social
 * network application.
 *
 * Display of the posts is currently simulated by printing the details
 * to the terminal. (Later, this should display in a browser.)
 *
 * This version does not save the data to disk, and it does not provide
 * any search or ordering functions.
 *
 * @author Michael Kölling and David J. Barnes
 * @version 0.1
 */
public class NewsFeed {
    private final ArrayList<MessagePost> messages;
    private final ArrayList<PhotoPost> photos;
```

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

    /**
     * Add a text post to the news feed.
     *
     * @param text  The text post to be added.
     */
    public void addMessagePost(MessagePost message) {
        messages.add(message);
    }

    /**
     * Add a photo post to the news feed.
     *
     * @param photo  The photo post to be added.
     */
    public void addPhotoPost(PhotoPost photo) {
        photos.add(photo);
    }

    /**
     * Show the news feed. Currently: print the news feed details to the
     * terminal. (To do: replace this later with display in web browser.)
     */
    public void show() {
        // display all text posts
        for (MessagePost message : messages) {
            message.display();
            System.out.println();   // empty line between posts
        }

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

There is still much missing from our application, eg, a decent user interface, and the ability to save the data somewhere permanently.

Exercises

1. Create a class to run everything in the *network.v1* package, eg,

```
package main.network.v1;

import network.v1.MessagePost;
import network.v1.NewsFeed;
import network.v1.PhotoPost;

public class Main {

    public static void main(String[] args) {
        NewsFeed nf = new NewsFeed();
        MessagePost mp = new MessagePost("Leonardo da Vinci",
              "Code this...!");
        mp.like();
        mp.like();
        nf.addMessagePost(mp);
        PhotoPost pp = new PhotoPost("Alexander Graham Bell",
              "handset.png", "Coming soon: the Samsung Galaxy S3.");
        pp.like();
        pp.like();
        nf.addPhotoPost(pp);
        pp.like();
        pp.like();
        nf.show();
    }
}
```

Note that **Main** is in a different package from the network code. Compile and run the application; make sure you can do this from the command line, eg,

```
javac -d build/classes `find src -name "*.java"`
java -cp build/classes/ main.network.v1.Main
```

2. Add a comment to one of the **MessagePost**s. Is it printed when you list the contents of the news feed database? Why / why not?

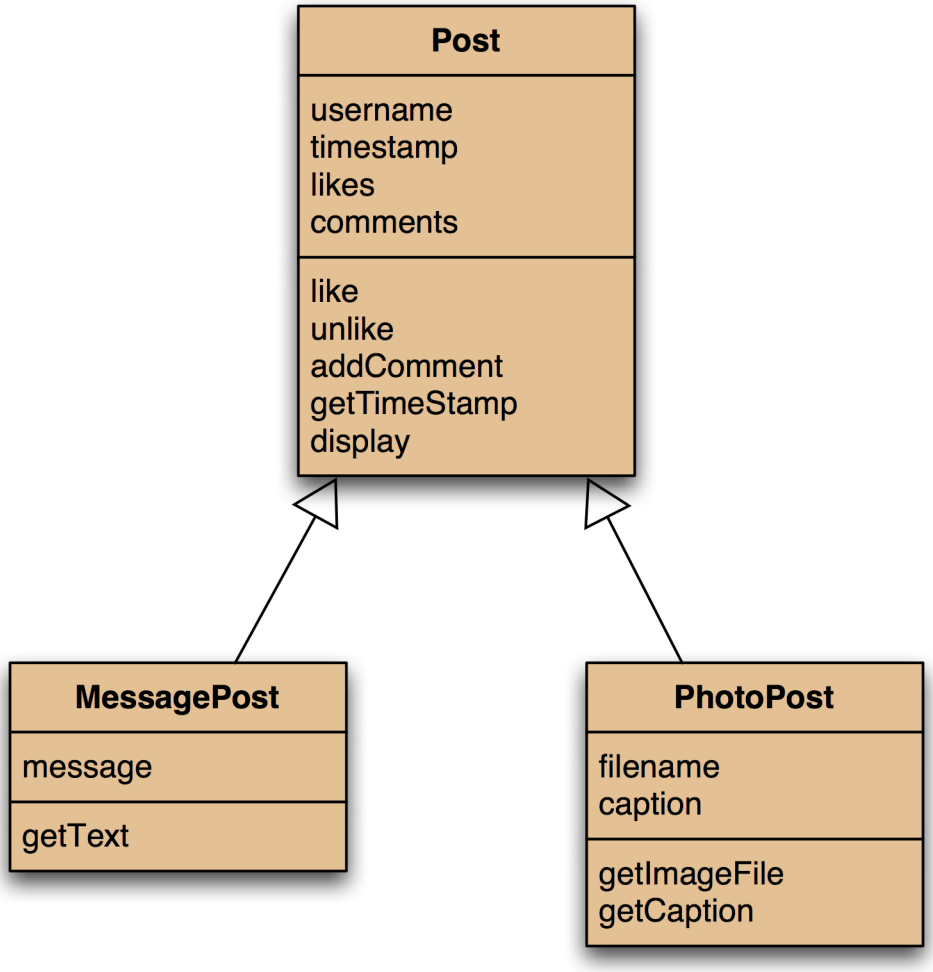
Discussion of the *network* application

The application above has several serious problems. First, the two classes **MessagePost** and **PhotoPost** are almost the same and suffer from *code duplication*, which is always a Bad Idea^{TM}. Code duplication makes code maintenance difficult, for example if we decide to change the type of the **likes** field from **int** to **long** (there was a lot of voting!) then we have to change it in both classes. The **NewsFeed** also suffers from code duplication - we need one method **addMessagePost** for adding **MessagePost**s and a completely different method **addPhotoPost** for adding **PhotoPost**s. In the future, if we add new types of objects to store in the database, eg, events, then we need to add new methods to **NewsFeed**, one for every new type. We can eliminate the code duplication by taking advantage of the similarities between classes and using the object oriented programming notion of *inheritance*.

Using inheritance

The code duplication comes about because **MessagePost** and **PhotoPost** were defined

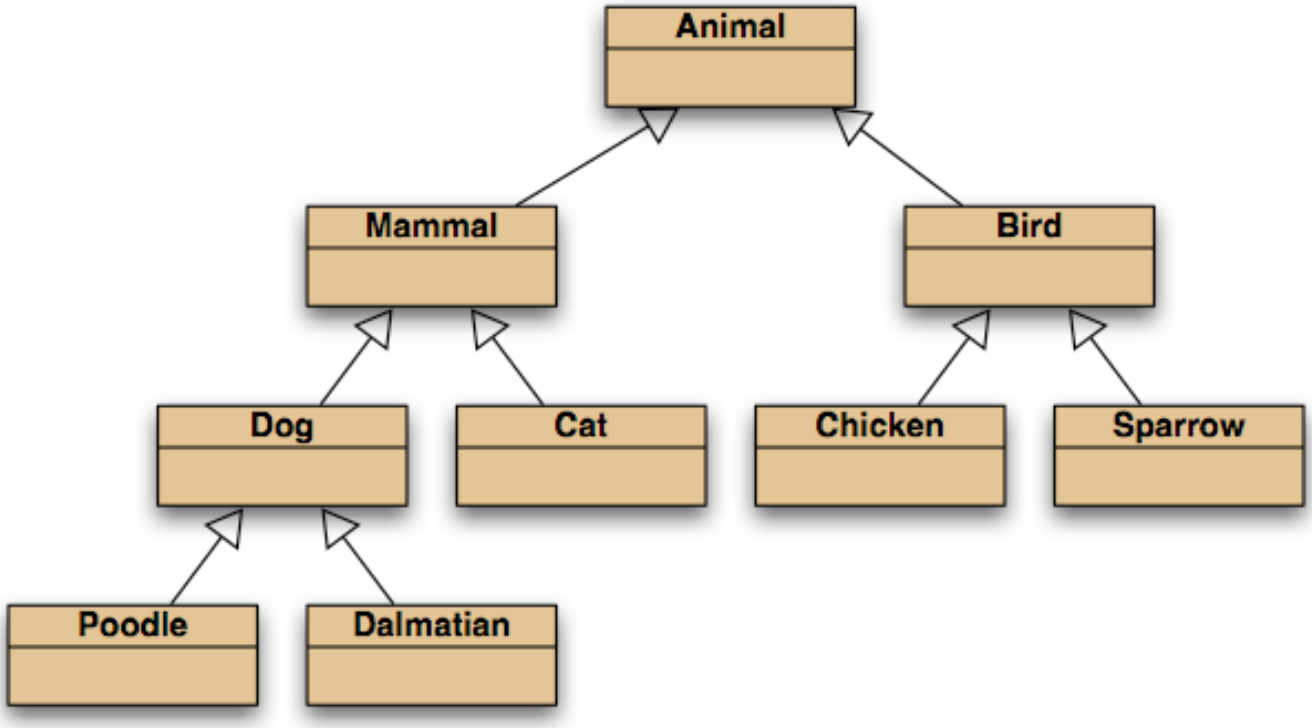
independently. Using inheritance to eliminate code duplication means defining a new class, call it **`Post`**, and moving everything that is in common into **`Post`**, and then saying that a **`MessagePost`** *is a* **`Post`** and **`PhotoPost`** *is a* **`Post`**. The UML class diagram looks like



Everything in common to **`MessagePost`** and **`PhotoPost`**, the fields **`username`**, **`timestamp`**, **`likes`**, and **`comments`**, the methods **`like`**, **`unlike`**, **`addComment`**, **`getTimestamp`**, and **`display`** now belong to **`Post`**. But since **`MessagePost`** and **`PhotoPost`** are both **`Posts`** as well, they also belong to them. On the other hand, the fields **`filename`** and **`caption`**, and the methods **`getImageFile`** and **`getcaption`** are specific to only **`PhotoPost`**. Similarly **`message`** and **`getText`** are specific to **`MessagePost`** and don't belong to **`PhotoPost`** nor **`Post`**.

`MessagePost` and **`PhotoPost`** *inherit from* **`Post`**. **`Post`** is called the *superclass* (or *base class*) and **`MessagePost`** is a *subclass*) (or *derived class*). In the UML diagram, the arrow points from the subclass to the superclass.

Inheritance hierarchies



Real-life hierarchies are common, eg, that used by biologists for classifying animals

Exercise

3. Draw an inheritance hierarchy for the people in your place of study or place of work. For example, if you are a university student, then your university probably has students (first-year students, second-year students,...), professors, tutors, office personnel, etc.

Inheritance in Java

The class **Post** starts with

```
class Post {
    private final String username; // username of the post's author
    private final long timestamp;
    private int likes;
    private final ArrayList<String> comments;

    Constructors and methods omitted.
}
```

Nothing unusual here.

MessagePost which inherits from **Post** starts with

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

    Constructors and methods omitted.
}
```

In Java, the keyword **extends** means *inherits from* so that **MessagePost extends Post** means that **MessagePost** inherits from **Post** as in the UML diagram.

Inheritance and access rights

A field declared with access level **private** can only be used *within the class where it is defined*. This applies even between super- and subclasses, For example, to make the **timestamp** field accessible to **MessagePost**, **Post** needs to have the appropriate getter method (in Java dogma, fields are **private** and if needed are made accessible outside their class by introducing the appropriate *accessor* methods).

Exercises

4. Look at the package *network.v2*. This code contains a version of the network application rewritten to use inheritance, as described above. Create the appropriate **Main** class.
5. For one of the **MessagePost** objects created in the **Main** class, call some of its methods. Can you call the inherited methods, eg, **addComment**? What do you observe about the inherited methods?
6. No special syntax is needed for a subclass to access public elements of its superclass. Create a method **printShortSummary** in the **MessagePost** class which prints "Message post from NAME" where NAME is the name of the author. However, **username** is a private field of **Post.** Without changing the access of **username,** make

this work.

Inheritance and initialization

The **MessagePost** constructor contains the arguments **String author, String text**. The variable **message** is an instance variable of **MessagePost** and can be initialized in **MessagePost**'s constructor. On the other hand, the variable **username** is an instance variable of **Post** and must be initialized in **Post**'s constructor. This is accomplished in the following

```
class Post {
    private final String username;  // username of the post's author
    private final long timestamp;
    private int likes;
    private ArrayList<String> comments;

    /**
     * Constructor for objects of class Post.
     *
     * @param author    The username of the author of this post.
     */
    Post(String author) {
        username = author;
        timestamp = System.currentTimeMillis();
        likes = 0;
        comments = new ArrayList<>();
    }

    Methods omitted.
}


public class MessagePost extends Post {
    private final String message;

    /**
     * Constructor for objects of class MessagePost.
     *
     * @param author
     *             The username of the author of this post.
     * @param text
     *             The text of this post.
     */
    public MessagePost(String author, String text) {
        super(author);
        message = text;
    }

    Methods omitted.
}
```

In

```
super(author);
```

super invokes the *constructor* of **MessagePost**'s superclass **Post** and passes it the value of **author**.

Every Java class's constructor must invoke its superclass's constructor *on the first line*. What??!!! We haven't been doing this and yet our code has worked... That's because Java automagically adds the invocation

```
super();
```

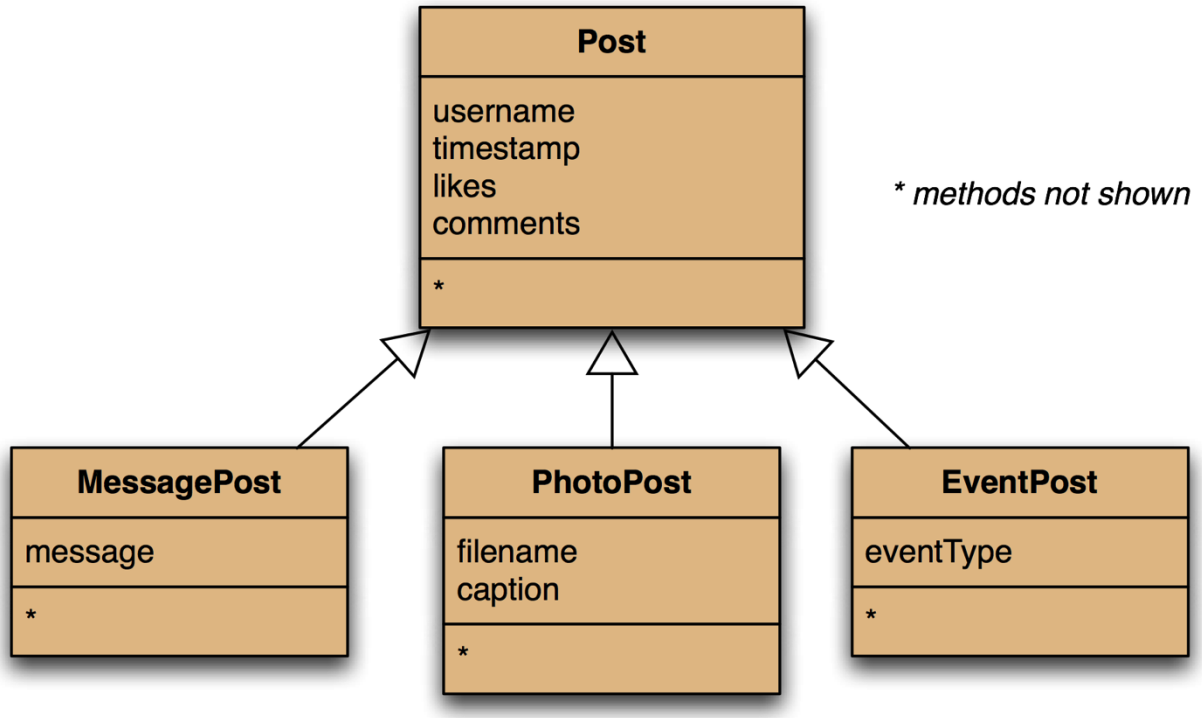
as the first line of the constructor if the developer leaves it out. This calls the *superclass's no-args constructor* because Java can't know by itself what arguments to supply.

Style

It is considered a Good Idea™ to always include explicit superclass calls in constructors. This makes the code clearer.

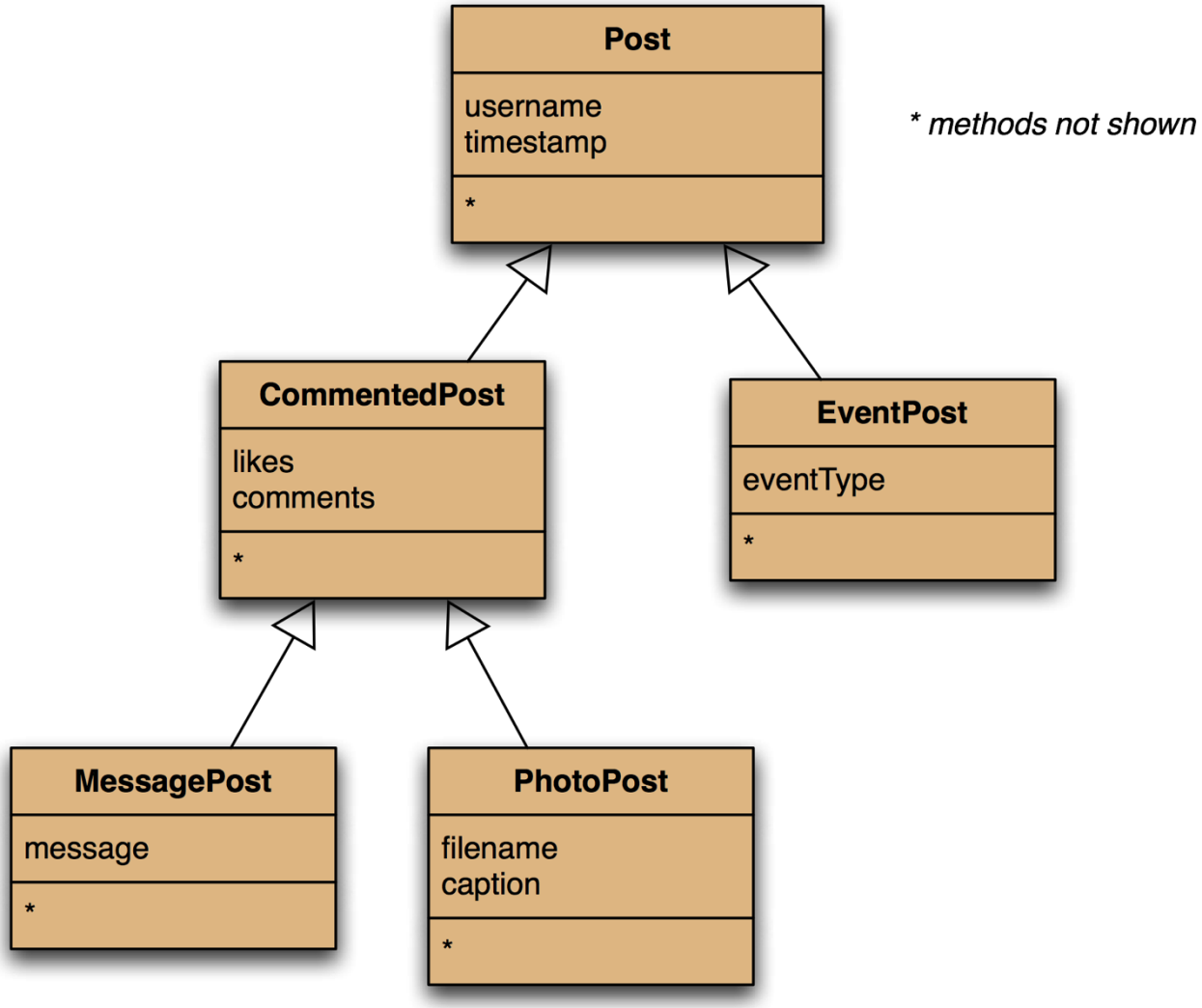
Network: adding other post types

Adding a third type of post to our application now becomes much easier, eg, **EventPost**



By inheritance, the class **EventPost** already has the methods that **Post** has, so we don't need to write them again. But we're going to change the project specifications a bit and say that **EventPost**s don't need likes and comments. Now, we could say that we'll just ignore them, but that's in bad taste since it violates the *is-a* principle which should guide inheritance – it would no longer be true that an **EventPost** is a **Post** **since the methods `like` and**

`unlike` no longer apply. We need to re-think the whole hierarchy. Introducing a new superclass **`CommentedPost`** for both **`MessagePost`** and **`PhotoPost`** allows us to move whatever is common to them but not to **`EventPost`** into this new class.



Exercise

8. Add a class for event posts to the *network.v2* package. Create some event post objects and test that all methods work as expected.

Advantages of inheritance (so far)

- **Avoiding code duplication** The use of inheritance avoids the need to write very similar copies of code more than once.
- **Code re-use** Existing code can be reused. If a class similar to the one we need already exists, we can subclass the existing class and reuse some of the existing code.
- **Easier maintenance** Maintaining applications becomes easier since the relationship between the classes is clearly expressed. A change to a field or a method that is shared between different types of subclasses needs to be made only once.
- **Extendibility** Using inheritance, it becomes much easier to extend an existing application in certain ways.

Exercises

9. Order these items into an inheritance hierarchy: apple, ice-cream, bread, fruit, food-item, cereal, orange, dessert, chocolate mousse, baguette.
10. What is the inheritance hierarchy between touchpad and (computer) mouse?
11. Inheritance sometimes can lead to headaches – think about the relationship between a **`Square`** class and a **`Rectangle`** class.

Subtyping

Inheritance also simplifies the **`NewsFeed`** class

```
package network.v2;

import java.util.ArrayList;

/**
 * The NewsFeed class stores news posts for the news feed in a
 * social network application.
 *
 * Display of the posts is currently simulated by printing the
 * details to the terminal. (Later, this should display in a browser.)
 *
 * This version does not save the data to disk, and it does not
 * provide any search or ordering functions.
 *
 * @author Michael Kölling and David J. Barnes
 * @version 0.2
 */
public class NewsFeed {
    private final ArrayList<Post> posts;

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

    /**
     * Add a post to the news feed.
     *
     * @param post  The post to be added.
     */
    public void addPost(Post post) {
        posts.add(post);
    }

    /**
     * Show the news feed. Currently: print the news feed details
     * to the terminal. (To do: replace this later with display
     * in web browser.)
```

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

This is considerably shorter than the original version. **NewsFeed** now knows nothing of **MessagePost**s or **PhotoPost**s, it only does **Posts**, so that the two methods

```
public void addMessagePost(MessagePost message)
public void addPhotoPost(PhotoPost photo)
```

can be replaced by the single method

```
public void addPost(Post post)
```

This works because both **MessagePost**s **and** **PhotoPost**s *are* **Post**s.

Substitution rule

Since an object of subclass type *is an* object of superclass type, a subclass object can be substituted for - used in place of - a superclass object. Hence the following lines are perfectly legal

```
addPost(new MessagePost(/* arguments */));
addPost(new PhotoPost(/* arguments */));
```

even though **addPost** expects to be passed a **Post** object.

Subclasses and subtypes

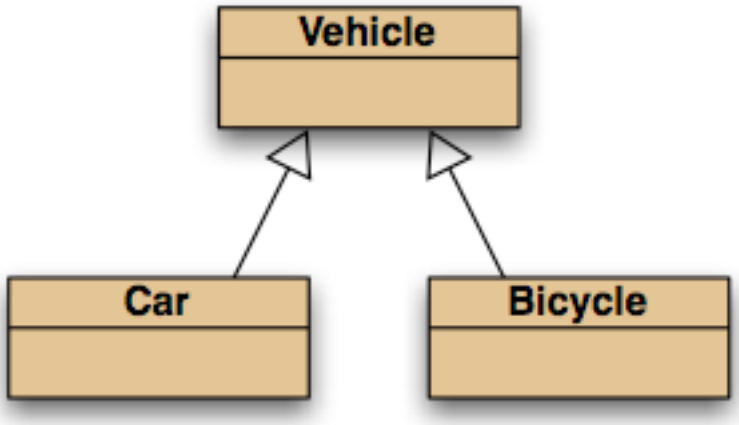
Classes determine types, hence **PhotoPost** is a subtype of **Post** since **PhotoPost** is a subclass of **Post**.

Subtyping and assignment

In an assignment statement, eg,

```
Car myCar = new Car();
```

the type of the object (on the right-hand side of **=**) must match the type of the variable (on the left-hand side of **=**). The type of **new Car()** must match Car. Now consider



By the substitution rule, the following are valid statements

```
Vehicle v1 = new Vehicle();
Vehicle v1 = new Car();
Vehicle v1 = new Bicycle();
```

However, these statements are not valid

```
Car c1 = new Vehicle();
Car c2 = new Bicycle();
```

Exercises

12. Assume we have four classes:

    Person, **Teacher**, **Student**, and **PhDStudent**. **Teacher** and **Student** are both subclasses of **Person**. **PhDStudent** is a subclass of **Student**.

    a. Which of the following statements are legal, and why (*use only pencil and paper*)?

    ```
    Person p1 = new Student();
    Person p2 = new PhDStudent();
    PhDStudent phd1 = new Student();
    Teacher t1 = new Person();
    Student s1 = new PhDStudent();
    ```

    b. Suppose that the following are all legal

    ```
    Person p1 = new Person();
    Person p2 = new Person();
    PhDStudent phd1 = new PhDStudent();
    Teacher t1 = new Teacher();
    Student s1 = new Student();
    ```

    Based on the above, which of the following assignments are legal, and why / why not?

    ```
    s1 = p1;
    s1 = p2;
    p1 = s1;
    t1 = s1;
    s1 = phd1;
    phd1 = s1;
    ```

13. Test your answers to the previous question by creating the classes mentioned in that exercise.

Subtyping and parameter passing

Given the following **addPost** method definition in the **NewsFeed** class

```
public class NewsFeed {
    public void addPost(Post post) {
        …
    }
}
```

Whatever gets passed as an argument is treated as of type Post. Hence the following statements:

```
NewsFeed feed = new NewsFeed();
MessagePost message = new MessagePost();
PhotoPost photo = new PhotoPost();

feed.addPost(message);
feed.addPost(photo);
```

These are all valid since the actual argument is *upcast* to a **Post** object.

Polymorphic variables

Polymorphic - meaning *many shapes*. Variables holding object types in Java are *polymorphic* variables, meaning they can hold objects of different types. Specifically, they can hold objects of the declared type and any subtype. The body of **NewsFeed**'s **show** method is

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

The **ArrayList posts** stores objects whose actual types are either **MessagePost** or **PhotoPost**, yet we assign them in turn to the **post** variable. This works because both are subtypes of **Post**.

Exercise

14. What has to change in the **NewsFeed** class when another **Post** subclass, eg, a class **VideoPost**, is added? Why?

Casting

The compiler would reject the last line of the following code

```
Vechicle v;
Car c = new Car();
v = c; // correct;
c = v; // compile-time error!
```

since a **Vehicle** object is not necessarily a **Car** object. However, the above example has been

contrived so that it *should* work, since the reference to **v** is actually a **Car** object, and we can in fact force the compiler to accept it:

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

In the above, the variable **v** has been *cast as* a **Car** object.

But now consider the following

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

Casting should be used with moderation, and generally indicates that your code might be better organized differently.

The Object class

All classes have a superclass...except for **java.lang.Object** which is the *cosmic superclass* since *every* class ultimately inherits from **Object**. When we write

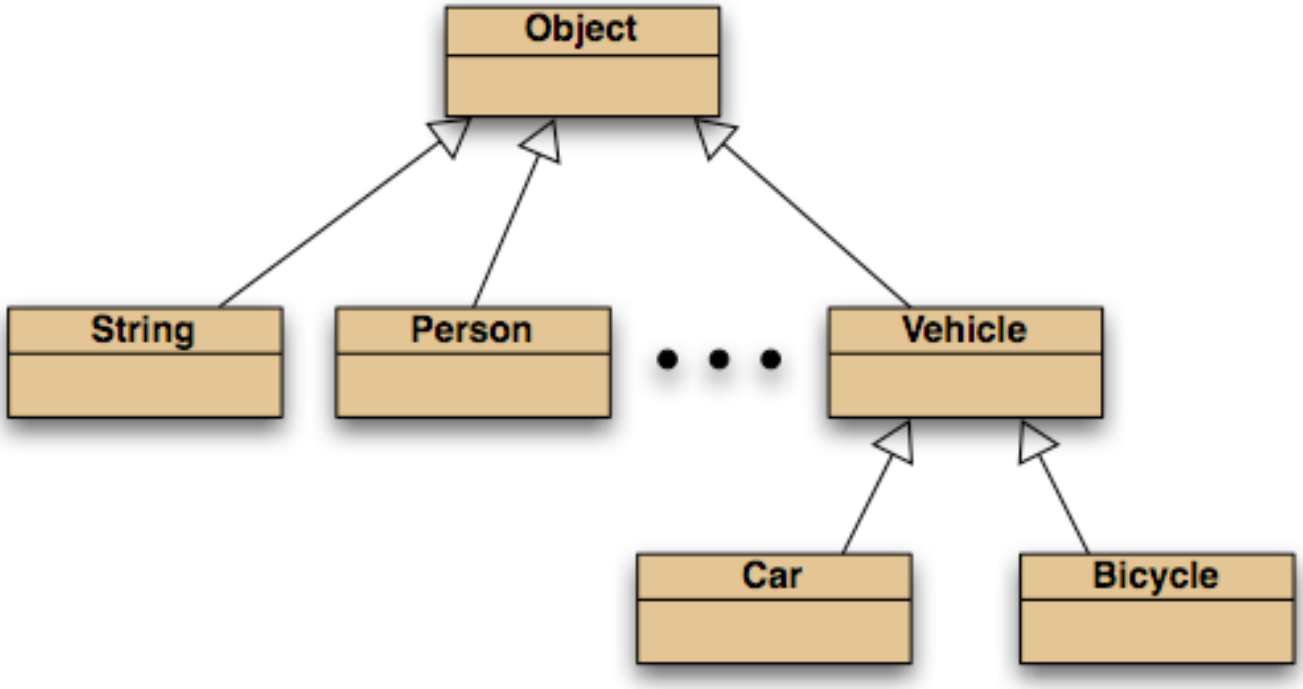
```
class Person {
     ...
}
```

it's as if we had written

```
class Person extends Object {
     ...
}
```

Whenever a class doesn't explicitly inherit from any other, it still inherits from **Object**.

Having every class inherit from **Object** has two advantages:



- every class can inherit common code from **`Object`**, eg, every class has a method **`toString`**;
- every object of any type can be upcast to **`Object`**, and then it can be treated polymorphically as of type **`Object`**.

Autoboxing and wrapper classes

We have seen that **`ArrayList`** can store any type of object, but what if we want to store **`int`**s? We can't, in fact we can't directly store any primitive types in collection objects. Java provides wrapper types for this situation

```
int i = 18; // the value to be entered
Integer iwrap = new Integer(i); // the wrapper around i
```

and **`iwrap`** could be stored in, eg, an **`ArrayList`** collection. *Autoboxing* makes the above a bit easier syntactically

```
private final ArrayList<Integer> markList;
      ...
private void storeMark(int mark) {
      markList.add(mark); // autoboxing
      ...
      int firstMark = markList.remove(0); // unboxing
}
```

The collection hierarchy

Java's collection framework makes use of polymorphism to work. It also uses subclassing extensively in its definition.

More information than you'll want to know, but that you'll need anyway, in *http://docs.oracle.com/javase/tutorial/collections/*.

Exercises

15. From the documentation for the Java standard collection classes, draw a diagram showing the whole hierarchy.

Summary

All Java classes are arranged in an inheritance hierarchy, with the **`Object`** class at the top.

Subclasses are specializations of superclasses. An object of subclass type *is an* object of its superclass type. Subclasses inherit methods and fields from superclasses. A subclass type object can be substituted for superclass objects.

Concept summary

inheritance Inheritance allows us to define one class as an extension of another.

superclass A superclass is a class that is extended by another class.

subclass A subclass is a class that extends (inherits from) another class. It inherits fields and methods from its superclass.

inheritance hierarchy Classes that are linked through inheritance relationships form an inheritance hierarchy.

superclass constructors The constructor of a subclass must always invoke the constructor of it superclass as its first statement. If the source code does not include such a call, Java will attempt to insert a call automagically.

reuse Inheritance allows us to reuse previously written classes in a new context.

subtypes As an analog to the class hierarchy, types form a type hierarchy. The type defined by a subclass definition is a subtype of the type of the superclass.

variables and subtypes Variables may hold objects of their declared type or of any subtype of their declared type.

substitution Subtype objects may be used wherever objects of a supertype are expected. This is known as substitution.

`Object` All classes with no explicit superclass have **`Object`** as their superclass.

autoboxing Autoboxing is performed automatically when a primitive-type value is used in a context requiring an object type.

Additional exercises

Exercises

16. Import the *labclasses.jar* archive into the project. Add an **`Instructor`** class to the project (a lab class has one instructor and can have many students). Use inheritance to avoid code duplication between **`Instructors`** and **`Students`** (both have a name, contact details, etc.).
17. Draw an inheritance hierarchy representing parts of a computer system (processor, memory, disk drive, CD drive, printer, scanner, keyboard, mouse, etc.).
18. Look at the code below. You have four classes (**`O`**, **`X`**, **`T`**, and **`M`**) and a variable of each of these.

```
O o;
X x;
T t;
M m;
```

The following assignments are all legal (assume that they all compile):

```
m = t;
m = x;
o = t;
```

The following assignments are all illegal:

```
o = m;
o = x;
x = o;
```

What can you say about the relationships of these classes?

19. Draw an inheritance hierarchy of **`AbstractList`** and all its (direct and indirect) subclasses, as they are defined in the Java standard library.
20. Retro-fit some unit tests onto the *network.v2* classes. Try to come up with *reasonable* tests that fail. Failing tests are actually good because they suggest what code has to be fixed up next. Now fix the code.