There are several fundamental problems with our current solution. The most obvious one is *code duplication*.

We have noticed above that the **MessagePost** and **PhotoPost** classes are very similar. In fact, the majority of the classes' source code is identical, with only a few differences. We have already discussed the problems associated with code duplication in Chapter 8. Apart from the annoying fact that we have to write everything twice (or copy and paste, then go through and fix all the differences), there are often problems associated with maintaining duplicated code. Many possible changes would have to be done twice. If, for example, the type of the comment list is changed from **ArrayList<String>** to **ArrayList<Comment>** (so that more details can be stored), this change has to be made once in the **MessagePost** class and again in the **PhotoPost** class. In addition, associated with maintenance of code duplication is always the danger of introducing errors, because the maintenance programmer might not realize that an identical change is needed at a second (or third) location.

There is another spot where we have code duplication: in the **NewsFeed** class. We can see that everything in that class is done twice—once for message posts, and once for photo posts. The class defines two list variables, then creates two list objects, defines two **add** methods, and has two almost-identical blocks of code in the **show** method to print out the lists.

The problems with this duplication become clear when we analyze what we would have to do to add another type of post to this program. Imagine that we want to store not only text messages and photo posts, but also activity posts. Activity posts can be automatically generated and inform us about an activity of one of our contacts, such as "Fred has changed his profile picture" or "Jacob is now friends with Feena." Activity posts seem similar enough that it should be easy to modify our application to do this. We would introduce another class, **ActivityPost**, and essentially write a third version of the source code that we already have in the **MessagePost** and **PhotoPost** classes. Then we have to work through the **NewsFeed** class and add another list variable, another list object, another **add** method, and another loop in the **show** method.

We would have to do the same for a fourth type of post. The more we do this, the more the code-duplication problem increases, and the harder it becomes to make changes later. When we feel uncomfortable about a situation such as this one, it is often a good indicator that there may be a better alternative approach. For this particular case, the solution is found in object-oriented languages. They provide a distinctive feature that has a big impact on programs involving sets of similar classes. In the following sections, we will introduce this feature, which is called *inheritance*.

10.2 Using inheritance

Concept

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

Inheritance is a mechanism that provides us with a solution to our problem of duplication. The idea is simple: instead of defining the **MessagePost** and **PhotoPost** classes completely independently, we first define a class that contains everything these two have in common. We shall call this class **Post**. Then we can declare that a **MessagePost** is a **Post** and a **PhotoPost** is a **Post**. Finally, we add those extra details needed for a message post to the **MessagePost** class, and those for a photo post to the **PhotoPost** class. The essential feature of this technique is that we need to describe the common features only once.

Figure 10.5 shows a class diagram for this new structure. At the top, it shows the class **Post**, which defines all fields and methods that are common to all posts (messages and photos). Below the **Post** class, it shows the **MessagePost** and **PhotoPost** classes, which hold only those fields and methods that are unique to each particular class.

This new feature of object-oriented programming requires some new terminology. In a situation such as this one, we say that the class **MessagePost** *inherits from* class **Post**. Class **PhotoPost** also inherits from **Post**. In the vernacular of Java programs, the expression "class **MessagePost** *extends* class **Post**" could be used, because Java uses an **extends** keyword to define the inheritance relationship (as we shall see shortly). The arrows in the class diagram (usually drawn with hollow arrow heads) represent the inheritance relationship.

Concept

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

Class **Post** (the class that the others inherit from) is called the *parent class* or *superclass*. The inheriting classes (**MessagePost** and **PhotoPost** in this example) are referred to as *child classes* or *subclasses*. In this book, we will use the terms "superclass" and "subclass" to refer to the classes in an inheritance relationship.

Concept

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

Inheritance is sometimes also called an *is-a* relationship. The reason is that a subclass is a specialization of a superclass. We can say that "a message post *is a* post" and "a photo post *is a* post."

The purpose of using inheritance is now fairly obvious. Instances of class **MessagePost** will have all fields defined in class **MessagePost** *and* in class **Post**. (**MessagePost** inherits the fields from **Post**.) Instances of **PhotoPost** will have all fields defined in **PhotoPost** and in **Post**. Thus, we achieve the same as before, but we need to define the fields **username**, **timestamp**, **likes**, and **comments** only once, while being able to use them in two different places.

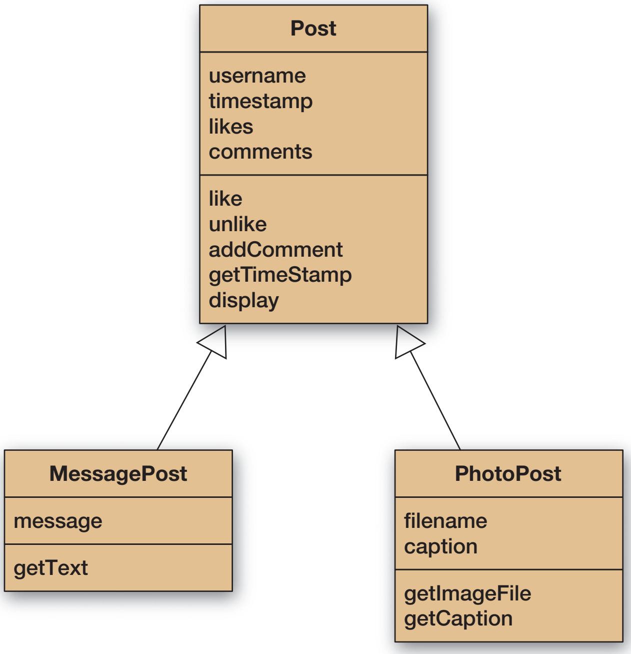

Figure 10.5
MessagePost and **PhotoPost** inheriting from **Post**