The keyword **`super`** is a call from the subclass constructor to the constructor of the superclass. Its effect is that the **`Post`** constructor is executed as part of the **`MessagePost`** constructor's execution. When we create a message post, the **`MessagePost`** constructor is called, which, in turn, as its first statement, calls the **`Post`** constructor. The **`Post`** constructor initializes the post's fields, and then returns to the **`MessagePost`** constructor, which initializes the remaining field defined in the **`MessagePost`** class. For this to work, those parameters needed for the initialization of the post fields are passed on to the superclass constructor as parameters to the **`super`** call.

Concept

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

In Java, a subclass constructor must always call the *superclass constructor* as its first statement. If you do not write a call to a superclass constructor, the Java compiler will insert a superclass call automatically, to ensure that the superclass fields are properly initialized. The inserted call is equivalent to writing

```
super();
```

Inserting this call automatically works only if the superclass has a constructor without parameters (because the compiler cannot guess what parameter values should be passed). Otherwise, an error will be reported.

In general, it is a good idea to always include explicit superclass calls in your constructors, even if it is one that the compiler could generate automatically. We consider this good style, because it avoids the possibility of misinterpretation and confusion in case a reader is not aware of the automatic code generation.

Exercise 10.7 Set a breakpoint in the first line of the **`MessagePost`** class's constructor. Then create a **`MessagePost`** object. When the debugger window pops up, use *Step Into* to step through the code. Observe the instance fields and their initialization. Describe your observations.

10.5 *Network*: adding other post types

Now that we have our inheritance hierarchy set up for the *network* project so that the common elements of the items are in the **`Post`** class, it becomes a lot easier to add other types of posts. For instance, we might want to add event posts, which consist of a description of a standard event (e.g., "Fred has joined the 'Neal Stephenson fans' group."). Standard events might be a user joining a group, a user becoming friends with another, or a user changing their profile picture. To achieve this, we can now define a new subclass of **`Post`** named **`EventPost`** (Figure 10.7). Because **`EventPost`** is a subclass of **`Post`**, it automatically inherits all fields and methods that we have already defined in **`Post`**. Thus, **`EventPost`** objects already have a username, a time stamp, a likes counter, and comments. We can then concentrate on adding attributes that are specific to event posts, such as the event type. The event type might be stored as an enumeration constant (see Chapter 8) or as a string describing the event.

This is an example of how inheritance enables us to *reuse* existing work. We can reuse the code that we have written for photo posts and message posts (in the **`Post`** class) so that it

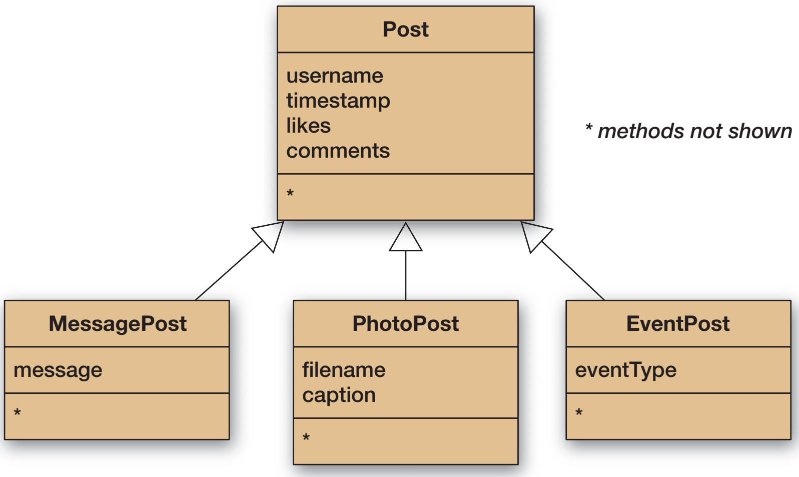

Figure 10.7
Network items with an **EventPost** class

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

also works for the **EventPost** class. The ability to reuse existing software components is one of the great benefits that we get from the inheritance facility. We will discuss this in more detail later.

This reuse has the effect that a lot less new code is needed when we now introduce additional post types. Because new post types can be defined as subclasses of **Post**, only the code that is actually different from **Post** has to be added.

Now imagine that we change the requirements a bit: event posts in our *network* application will not have a "Like" button or comments attached. They are for information only. How do we achieve this? Currently, because **EventPost** is a subclass of **Post**, it automatically inherits the **likes** and **comments** fields. Is this a problem?

We could leave everything as it is and decide to never display the likes count or comments for event posts—just ignore the fields. This does not feel right. Having the fields present but unused invites problems. Someday, a maintenance programmer will come along who does not realize that these fields should not be used and try to process them.

Or we could write **EventPost** without inheriting from **Post**. But then we are back to code duplication for the **username** and **timestamp** fields and their methods.

The solution is to refactor the class hierarchy. We can introduce a new superclass for all posts that have comments attached (named **CommentedPost**), which is a subclass of **Post** (Figure 10.8). We then shift the **likes** and **comments** fields from the **Post** class to this new class. **MessagePost** and **PhotoPost** are now subclasses of our new **CommentedPost** class, while **EventPost** inherits from **Posts** directly. **MessagePost** objects inherit everything from both superclasses and have the same fields and methods as before. Objects of class **EventPost** will inherit the **username** and **timestamp**, but not the comments.

This is a very common situation in designing class hierarchies. When the hierarchy does not seem to fit properly, we have to refactor the hierarchy.