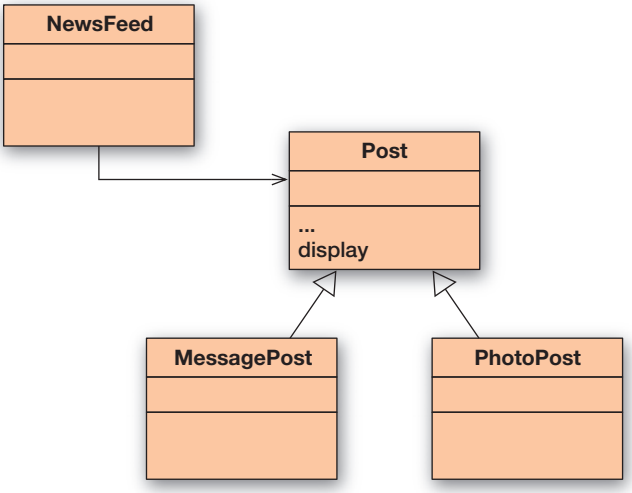

Figure 11.1
Display, version 1: **`display`** method in superclass

11.2 Static type and dynamic type

Trying to solve the problem of developing a complete polymorphic **`display`** method leads us into a discussion of *static* and *dynamic types* and *method lookup.* But let us start at the beginning.

A first attempt to solve the display problem might be to move the **`display`** method to the subclasses (Figure 11.2). That way, because the method would now belong to the **`MessagePost`** and **`PhotoPost`** classes, it could access the specific fields of **`MessagePost`** and **`PhotoPost`**. It could also access the inherited fields by calling accessor methods defined in the **`Post`** class. That should enable it to display a complete set of information again. Try out this approach by completing Exercise 11.1.

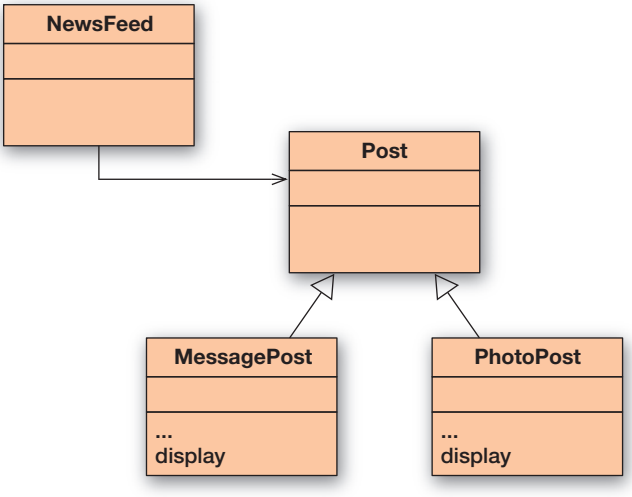

Figure 11.2
Display, version 2: **`display`** method in subclasses

Exercise 11.1 Open your last version of the *network* project. (You can use *network-v2* if you do not have your own version yet.) Remove the **display** method from class **Post** and move it into the **MessagePost** and **PhotoPost** classes. Compile. What do you observe?

When we try to move the **display** method from **Post** to the subclasses, we notice that the project does not compile any more. There are two fundamental issues:

- We get errors in the **MessagePost** and **PhotoPost** classes, because we cannot access the superclass fields.
- We get an error in the **NewsFeed** class, because it cannot find the **display** method.

The reason for the first sort of error is that the fields in **Post** have private access, and so are inaccessible to any other class—including subclasses. Because we do not wish to break encapsulation and make these fields public, as was suggested above, the easiest way to solve this is to define public accessor methods for them. However, in Section 11.9, we shall introduce a further type of access designed specifically to support the superclass–subclass relationship.

The reason for the second sort of error requires a more detailed explanation, and this is explored in the next section.

11.2.1 Calling display from NewsFeed

First, we investigate the problem of calling the **display** method from **NewsFeed**. The relevant lines of code in the **NewsFeed** class are:

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

The for-each statement retrieves each post from the collection; the first statement inside its body tries to invoke the **display** method on the post. The compiler informs us that it cannot find a **display** method for the post.

On the one hand, this seems logical; **Post** does not have a **display** method any more (see Figure 11.2).

On the other hand, it seems illogical and is annoying. We know that every **Post** object in the collection is in fact a **MessagePost** or a **PhotoPost** object, and both have **display** methods. This should mean that **post.display()** ought to work, because, whatever it is—**MessagePost** or **PhotoPost**—we know that it does have a **display** method.

To understand in detail why it does not work, we need to look more closely at types. Consider the following statement:

```
Car c1 = new Car();
```

We say that the type of **c1** is **Car**. Before we encountered inheritance, there was no need to distinguish whether by "type of **c1**" we meant "the type of the variable **c1**" or "the type of the object stored in **c1**." It did not matter, because the type of the variable and the type of the object were always the same.

Now that we know about subtyping, we need to be more precise. Consider the following statement:

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

Concept

The **static type** of a variable **v** is the type as declared in the source code in the variable declaration statement.

What is the type of **v1**? That depends on what precisely we mean by "type of **v1**." The type of the variable **v1** is **Vehicle**; the type of the object stored in **v1** is **Car**. Through subtyping and substitution rules, we now have situations where the type of the variable and the type of the object stored in it are not exactly the same.

Let us introduce some terminology to make it easier to talk about this issue:

- We call the declared type of the variable the *static type,* because it is declared in the source code—the static representation of the program.
- We call the type of the object stored in a variable the *dynamic type,* because it depends on assignments at runtime—the dynamic behavior of the program.

Concept

The **dynamic type** of a variable **v** is the type of the object that is currently stored in **v**.

Thus, looking at the explanations above, we can be more precise: the static type of **v1** is **Vehicle**, the dynamic type of **v1** is **Car**. We can now also rephrase our discussion about the call to the post's **display** method in the **NewsFeed** class. At the time of the call

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

the static type of **post** is **Post**, while the dynamic type is either **MessagePost** or **PhotoPost** (Figure 11.3). We do not know which one of these it is, assuming that we have entered both **MessagePost** and **PhotoPost** objects into the feed.

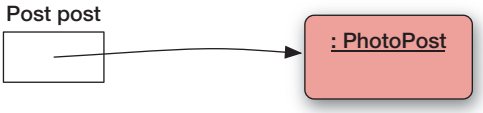

Figure 11.3
Variable of type **Post** containing an object of type **PhotoPost**

The compiler reports an error because, for type checking, the static type is used. The dynamic type is often only known at runtime, so the compiler has no other choice but to use the static type if it wants to do any checks at compile time. The static type of **post** is **Post**, and **Post** does not have a **display** method. It makes no difference that all known subtypes of **Post** do have a **display** method. The behavior of the compiler is reasonable in this respect, because it has no guarantee that *all* subclasses of **Post** will, indeed, define a **display** method, and this is impossible to check in practice.

In other words, to make this call work, class **Post** must have a **display** method, so we appear to be back to our original problem without having made any progress.