The technique we are using here is called *overriding* (sometimes it is also referred to as *redefinition*). Overriding is a situation where a method is defined in a superclass (method **display** in class **Post** in this example), and a method with exactly the same signature is defined in the subclass. The annotation **@Override** may be added before the version in the subclass to make it clear that a new version of an inherited method is being defined.

In this situation, objects of the subclass have two methods with the same name and header: one inherited from the superclass and one from the subclass. Which one will be executed when we call this method?

11.4 Dynamic method lookup

One surprising detail is what exactly is printed once we execute the news feed's **show** method. If we again create and enter the objects described in Section 11.1, the output of the **show** method in our new version of the program is

```
Had a great idea this morning.
But now I forgot what it was. Something to do with flying . . .

[experiment.jpg]
I think I might call this thing 'telephone'.
```

We can see from this output that the **display** methods in **MessagePost** and in **PhotoPost** were executed, but not the one in **Post**.

This may seem strange at first. Our investigation in Section 11.2 has shown that the compiler insisted on a **display** method in class **Post**—methods in the subclasses were not enough. This experiment now shows that the method in class **Post** is then not executed at all, but the subclass methods are. In short:

- Type checking uses the static type, but at runtime, the methods from the dynamic type are executed.

This is a fairly important statement. To understand it better, we look in more detail at how methods are invoked. This procedure is known as *method lookup*, *method binding*, or *method dispatch.* We will use the term "*method lookup*" in this book.

We start with a simple method-lookup scenario. Assume that we have an object of a class **PhotoPost** stored in a variable **v1** declared of type **PhotoPost** (Figure 11.5).

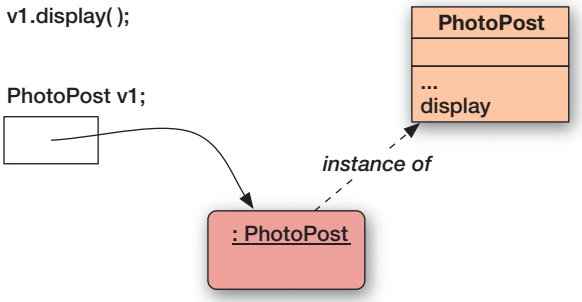


Figure 11.5
Method lookup with a simple object

The **PhotoPost** class has a **display** method and no declared superclass. This is a very simple situation—there is no inheritance or polymorphism involved here. We then execute the statement

```
v1.display();
```

When this statement executes, the **display** method is invoked in the following steps:

1. The variable **v1** is accessed.
2. The object stored in that variable is found (following the reference).
3. The class of the object is found (following the "instance of" reference).
4. The implementation of the **display** method is found in the class and executed.

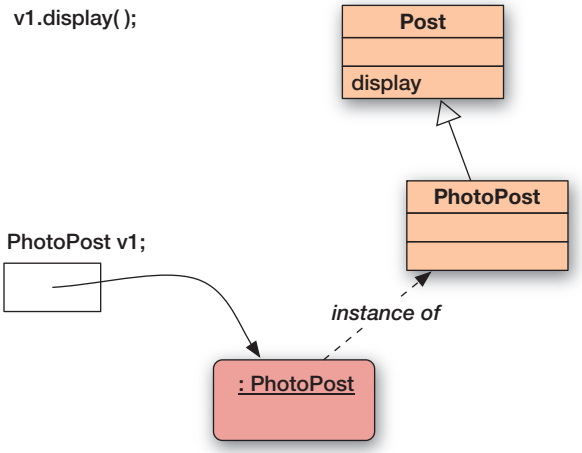
This is all very straightforward and not surprising.

Next, we look at method lookup with inheritance. This scenario is similar, but this time the **PhotoPost** class has a superclass **Post**, and the **display** method is defined only in the superclass (Figure 11.6).

We execute the same statement. The method invocation then starts in a similar way: steps 1 through 3 from the previous scenario are executed again, but then it continues differently:

4. No **display** method is found in class **PhotoPost**.
5. Because no matching method was found, the superclass is searched for a matching method. If no method is found in the superclass, the next superclass (if it exists) is searched. This continues all the way up the inheritance hierarchy to the **Object** class, until a method is found. Note that at runtime, a matching method should definitely be found, or else the class would not have compiled.
6. In our example, the **display** method is found in class **Post**, and will be executed.

Figure 11.6
Method lookup with inheritance



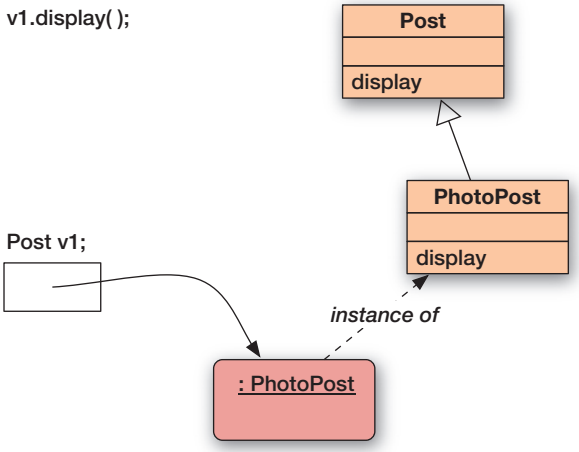


Figure 11.7
Method lookup with polymorphism and overriding

This scenario illustrates how objects inherit methods. Any method found in a superclass can be invoked on a subclass object and will correctly be found and executed.

Next, we come to the most interesting scenario: method lookup with a polymorphic variable and method overriding (Figure 11.7). The scenario is again similar to the one before, but there are two changes:

- The declared type of the variable **`v1`** is now **`Post`**, not **`PhotoPost`**.
- The **`display`** method is defined in class **`Post`** and then redefined (overridden) in class **`PhotoPost`**.

This scenario is the most important one for understanding the behavior of our *network* application, and in finding a solution to our **`display`** method problem.

The steps in which method execution takes place are exactly the same as steps 1 through 4 from scenario 1. Read them again.

Some observations are worth noting:

- No special lookup rules are used for method lookup in cases where the dynamic type is not equal to the static type. The behavior we observe is a result of the general rules.
- Which method is found first and executed is determined by the dynamic type, not the static type. In other words, the fact that the declared type of the variable **`v1`** is now **`Post`** does not have any effect. The instance we are dealing with is of class **`PhotoPost`**—that is all that matters.
- Overriding methods in subclasses take precedence over superclass methods. Because method lookup starts in the dynamic class of the instance (at the bottom of the inheritance hierarchy), the last redefinition of a method is found first, and this is the one that is executed.
- When a method is overridden, only the last version (the one lowest in the inheritance hierarchy) is executed. Versions of the same method in any superclasses are also not automatically executed.