It is worth reiterating what was illustrated in Exercise 10.6: that in the absence of method overriding, the non-private methods of a superclass are directly accessible from its subclasses without any special syntax. A **super** call only has to be made when it is necessary to access the superclass version of an *overridden* method.

If you completed Exercise 11.3, you will have noticed that this solution works, but is not perfect yet. It prints out all details, but in a different order from what we wanted. We will fix this last problem later in the chapter.

11.6 Method polymorphism

Concept

Method polymorphism. Method calls in Java are polymorphic. The same method call may at different times invoke different methods, depending on the dynamic type of the variable used to make that call.

What we have just discussed in the previous sections (Sections 11.2–11.5) is yet another form of polymorphism. It is what is known as *polymorphic method dispatch* (or *method polymorphism* for short).

Remember that a polymorphic variable is one that can store objects of varying types (every object variable in Java is potentially polymorphic). In a similar manner, Java method calls are polymorphic, because they may invoke different methods at different times. For instance, the statement

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

could invoke the **MessagePost**'s **display** method at one time and the **PhotoPost**'s **display** method at another, depending on the dynamic type of the **post** variable.

11.7 Object methods: toString

Concept

Every object in Java has a **toString** method that can be used to return a string representation of itself. Typically, to make it useful, an object should override this method.

In Chapter 10, we mentioned that the universal superclass, **Object**, implements some methods that are then part of all objects. The most interesting of these methods is **toString**, which we introduce here (if you are interested in more detail, you can look up the interface for **Object** in the standard library documentation).

Exercise 11.4 Look up **toString** in the library documentation. What are its parameters? What is its return type?

The purpose of the **toString** method is to create a string representation of an object. This is useful for any objects that are ever to be textually represented in the user interface, but also helps for all other objects; they can then easily be printed out for debugging purposes, for instance.

The default implementation of **toString** in class **Object** cannot supply a great amount of detail. If, for example, we call **toString** on a **PhotoPost** object, we receive a string similar to this:

```
PhotoPost@65c221c0
```