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
```

The return value simply shows the object's class name and a magic number.[2]

Exercise 11.5 You can easily try this out. Create an object of class **PhotoPost** in your project, and then invoke the **toString** method from the **Object** sub-menu in the object's pop-up menu.

To make this method more useful, we would typically override it in our own classes. We can, for example, define the **Post**'s **display** method in terms of a call to its **toString** method. In this case, the **toString** method would not print out the details, but just create a string with the text. Code 11.3 shows the changed source code.

Code 11.3 **toString** method for **Post** and **MessagePost**

```
public class Post
{
    ...
    public String toString()
    {
        String text = username + "\n" + timeString(timestamp);
        if(likes > 0) {
            text += "  -  " + likes + " people like this.\n";
        }
        else {
            text += "\n";
        }

        if(comments.isEmpty()) {
            return text + "   No comments.\n";
        }
        else {
            return text + "    " + comments.size() +
                   " comment(s). Click here to view.\n";
        }
    }

    public void display()
    {
        System.out.println(toString());
    }
}
```

[2] The magic number is in fact the memory address where the object is stored. It is not very useful except to establish identity. If this number is the same in two calls, we are looking at the same object. If it is different, we have two distinct objects.

Code 11.3 continued
toString method for **Post** and **MessagePost**

```
public class MessagePost extends Post
{
    ...
    public String toString()
    {
        return super.toString() + message + "\n";
    }

    public void display()
    {
        System.out.println(toString());
    }
}
```

Ultimately, we would plan on removing the **display** methods completely from these classes. A great benefit of defining just a **toString** method is that we do not mandate in the **Post** classes what exactly is done with the description text. The original version always printed the text to the output terminal. Now, any client (e.g., the **NewsFeed** class) is free to do whatever it chooses with this text. It might show the text in a text area in a graphical user interface; save it to a file; send it over a network; show it in a web browser; or, as before, print it to the terminal.

The statement used in the client to print the post could now look as follows:

```
System.out.println(post.toString());
```

In fact, the **System.out.print** and **System.out.println** methods are special in this respect: if the parameter to one of the methods is not a **String** object, then the method automatically invokes the object's **toString** method. Thus we do not need to write the call explicitly and could instead write

```
System.out.println(post);
```

Now consider the modified version of the **show** method of class **NewsFeed** shown in Code 11.4. In this version, we have removed the **toString** call. Would it compile and run correctly?

Code 11.4
New version of **NewsFeed show** method

```
public class NewsFeed
{
    Fields, constructors, and other methods omitted.

    /**
     * Show the news feed. Currently: print the news feed details to the
     * terminal. (To do: replace this later with display in web browser.)
     */
    public void show()
    {
        for(Post post : posts) {
            System.out.println(post);
        }
    }
}
```