In fact, the method *does* work as expected. If you can explain this example in detail, then you probably already have a good understanding of most of the concepts that we have introduced in this and the previous chapter! Here is a detailed explanation of the single **println** statement inside the loop.

- The for-each loop iterates through all posts and places them in a variable with the static type **Post**. The dynamic type is either **MessagePost** or **PhotoPost**.
- Because this object is being printed to **System.out** and it is not a **String**, its **toString** method is automatically invoked.
- Invoking this method is valid only because the class **Post** (the static type!) has a **toString** method. (Remember: Type checking is done with the static type. This call would not be allowed if class **Post** had no **toString** method. However, the **toString** method in class **Object** guarantees that this method is always available for any class.)
- The output appears properly with all details, because each possible dynamic type (**MessagePost** and **PhotoPost**) overrides the **toString** method and the dynamic method lookup ensures that the redefined method is executed.

The **toString** method is generally useful for debugging purposes. Often, it is very convenient if objects can easily be printed out in a sensible format. Most of the Java library classes override **toString** (all collections, for instance, can be printed out like this), and often it is a good idea to override this method for our classes as well.

11.8 Object equality: equals and hashCode

It is often necessary to determine whether two objects are "the same." The **Object** class defines two methods, **equals** and **hashCode**, that have a close link with determining similarity. We actually have to be careful when using phrases such as "the same"; this is because it can mean two quite different things when talking about objects. Sometimes we wish to know whether two different variables are referring to the same object. This is exactly what happens when an object variable is passed as a parameter to a method: there is only one object, but both the original variable and the parameter variable refer to it. The same thing happens when any object variable is assigned to another. These situations produce what is called *reference equality.* Reference equality is tested for using the **==** operator. So the following test will return **true** if both **var1** and **var2** are referring to the same object (or are both **null**), and **false** if they are referring to anything else:

```
var1 == var2
```

Reference equality takes no account at all of the *contents* of the objects referred to, just whether there is one object referred to by two different variables or two distinct objects. That is why we also define *content equality*, as distinct from reference equality. A test for content equality asks whether two objects are the same internally—that is, whether the internal states of two objects are the same. This is why we rejected using reference equality for making string comparisons in Chapter 6.

What content equality between two particular objects means is something that is defined by the objects' class. This is where we make use of the **equals** method that every class inherits from the **Object** superclass. If we need to define what it means for two objects to be equal according to their internal states, then we must override the **equals** method, which then allows us to write tests such as

```
var1.equals(var2)
```

This is because the **equals** method inherited from the **Object** class actually makes a test for reference equality. It looks something like this:

```
public boolean equals(Object obj)
{
    return this == obj;
}
```

Because the **Object** class has no fields, there is no state to compare, and this method obviously cannot anticipate fields that might be present in subclasses.

The way to test for content equality between two objects is to test whether the values of their two sets of fields are equal. Notice, however, that the parameter of the **equals** method is of type **Object**, so a test of the fields will make sense only if we are comparing fields of the same type. This means that we first have to establish that the type of the object passed as a parameter is the same as that of the object it is being compared with. Here is how we might think of writing the method in the **Student** class of the *lab-classes* project from Chapter 1:

```
public boolean equals(Object obj)
{
    if(this == obj) {
        return true; // Reference equality.
    }
    if(!(obj instanceof Student)) {
        return false; // Not the same type.
    }
    // Gain access to the other student's fields.
    Student other = (Student) obj;
    return name.equals(other.name) &&
           id.equals(other.id) &&
           credits == other.credits;
}
```

The first test is just an efficiency improvement; if the object has been passed a reference to itself to compare against, then we know that content equality must be true. The second test makes sure that we are comparing two students. If not, then we decide that the two objects cannot be equal. Having established that we have another student, we use a cast and another variable of the right type so that we can access its details properly. Finally, we make use of the fact that the private elements of an object are directly accessible to an instance of the same class; this is essential in situations such as this one, because there will not necessarily be accessor methods defined for every private field in a class. Notice that we have consistently used content-equality tests rather than reference-equality tests on the object fields **name** and **id**.