It will not always be necessary to compare every field in two objects in order to establish that they are equal. For instance, if we know for certain that every **Student** is assigned a unique **id**, then we need not test the **name** and **credits** fields as well. It would then be possible to reduce the final statement above to

```
return id.equals(other.id);
```

Whenever the **equals** method is overridden, the **hashCode** method should also be overridden. The **hashCode** method is used by data structures such as **HashMap** and **HashSet** to provide efficient placement and lookup of objects in these collections. Essentially, the **hashCode** method returns an integer value that represents an object. From the default implementation in **Object**, distinct objects have distinct **hashCode** values.

There is an important link between the **equals** and **hashCode** methods in that two objects that are the same as determined by a call to **equals** must return identical values from **hashCode**. This stipulation, or contract, can be found in the description of **hashCode** in the API documentation of the **Object** class.[3] It is beyond the scope of this book to describe in detail a suitable technique for calculating hash codes, but we recommend that the interested reader see Joshua Bloch's *Effective Java*, whose technique we use here.[4] Essentially, an integer value should be computed making use of the values of the fields that are compared by the overridden **equals** method. Here is a hypothetical **hashCode** method that uses the values of an integer field called **count** and a **String** field called **name** to calculate the code:

```
public int hashCode()
{
    int result = 17; // An arbitrary starting value.
    // Make the computed value depend on the order in which
    // the fields are processed.
    result = 37 * result + count;
    result = 37 * result + name.hashCode();
    return result;
}
```

11.9 Protected access

In Chapter 10, we noted that the rules on public and private visibility of class members apply between a subclass and its superclass, as well as between classes in different inheritance hierarchies. This can be somewhat restrictive, because the relationship between a superclass and its subclasses is clearly closer than it is with other classes. For this reason, object-oriented languages often define a level of access that lies between the complete restriction of private access and the full availability of public access. In Java, this is called *protected access* and is provided by the **protected** keyword as an alternative to **public** and **private**. Code 11.5 shows an example of a protected accessor method, which we could add to class **Post**.

[3] Note that it is not essential that unequal objects always return distinct hash codes.

[4] Joshua Bloch: *Effective Java*, 2nd edition. Addison-Wesley. ISBN: 0-321-35668-3.

Code 11.5
An example of a `protected` method

```
protected long getTimeStamp()
{
    return timestamp;
}
```

Concepts

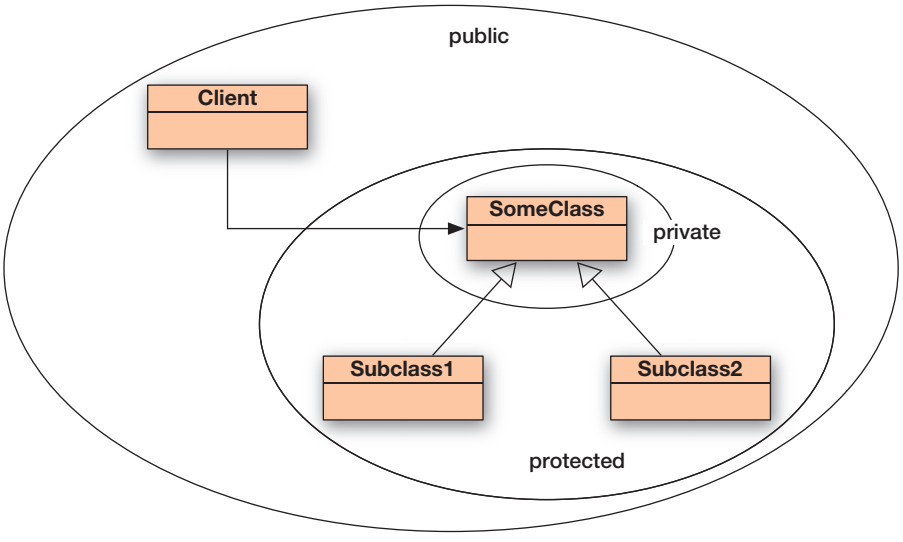
Declaring a field or a method **`protected`** allows direct access to it from (direct or indirect) subclasses.

Protected access allows access to the fields or methods within a class itself and from all its subclasses, but not from other classes.[5] The **`getTimeStamp`** method shown in Code 11.5 can be called from class **`Post`** or any subclasses, but not from other classes. Figure 11.8 illustrates this. The oval areas in the diagram show the group of classes that are able to access members in class **`SomeClass`**.

While protected access can be applied to any member of a class, it is usually reserved for methods and constructors. It is not usually applied to fields, because that would be a weakening of encapsulation. Wherever possible, mutable fields in superclasses should remain private. There are, however, occasional valid cases where direct access by subclasses is desirable. Inheritance represents a much closer form of coupling than does a normal client relationship.

Inheritance binds the classes closer together, and changing the superclass can more easily break the subclass. This should be taken into consideration when designing classes and their relationships.

Figure 11.8
Access levels: private, protected, and public



[5] In Java, this rule is not as clear-cut as described here, because Java includes an additional level of visibility, called *package level*, but with no associated keyword. We will not discuss this further, and it is more general to consider protected access as intended for the special relationship between superclass and subclass.