EXTENDING EXISTING CLASSES IN JAVA

The topics covered in later subsections of this section are relatively advanced aspects of object-oriented programming. Any programmer should know what is meant by subclass, inheritance, and polymorphism. However, it will probably be a while before you actually do anything with inheritance except for extending classes that already exist. In the first part of this section, we look at how that is done.

In day-to-day programming, especially for programmers who are just beginning to work with objects, subclassing is used mainly in one situation: There is an existing class that can be adapted with a few changes or additions. This is much more common than designing groups of classes and subclasses from scratch. The existing class can be extended to make a subclass. The syntax for this is

```
public class subclass-name extends existing-class-name {
   .
   .   // Changes and additions.
   .
}
```

As an example, suppose you want to write a program that plays the card game, Blackjack. You can use the *Card*, *Hand*, and *Deck* classes developed in Section 5.4. However, a hand in the game of Blackjack is a little different from a hand of cards in general, since it must be possible to compute the "value" of a Blackjack hand according to the rules of the game. The rules are as follows: The value of a hand is obtained by adding up the values of the cards in the hand. The value of a numeric card such as a three or a ten is its numerical value. The value of a Jack, Queen, or King is 10. The value of an Ace can be either 1 or 11. An Ace should be counted as 11 unless doing so would put the total value of the hand over 21. Note that this means that the second, third, or fourth Ace in the hand will always be counted as 1.

One way to handle this is to extend the existing *Hand* class by adding a method that computes the Blackjack value of the hand. Here's the definition of such a class:

```
public class BlackjackHand extends Hand {

    /**
     * Computes and returns the value of this hand in the game
     * of Blackjack.
     */
    public int getBlackjackValue() {

        int val;      // The value computed for the hand.
        boolean ace;  // This will be set to true if the
                      //   hand contains an ace.
        int cards;    // Number of cards in the hand.

        val = 0;
        ace = false;
        cards = getCardCount();   // (method defined in class
Hand.)

        for ( int i = 0;  i < cards;  i++ ) {
                // Add the value of the i-th card in the hand.
            Card card;    // The i-th card;
            int cardVal;  // The blackjack value of the i-th
card.
            card = getCard(i);
            cardVal = card.getValue();  // The normal value, 1
to 13.
            if (cardVal > 10) {
                cardVal = 10;   // For a Jack, Queen, or King.
            }
            if (cardVal == 1) {
                ace = true;     // There is at least one ace.
            }
            val = val + cardVal;
         }
         // Now, val is the value of the hand, counting any ace
as 1.
         // If there is an ace, and if changing its value from
1 to
         // 11 would leave the score less than or equal to 21,
         // then do so by adding the extra 10 points to val.

         if ( ace == true  &&  val + 10 <= 21 )
             val = val + 10;

         return val;

    }  // end getBlackjackValue()

} // end class BlackjackHand
```

Since *BlackjackHand* is a subclass of *Hand*, an object of type *BlackjackHand* contains all the instance variables and instance methods defined in *Hand*, plus the new instance method named`getBlackjackValue()`. For example, if `bjh` is a variable of type `BlackjackHand`, then the following are all legal: `bjh.getCardCount()`, `bjh.removeCard(0)`, and `bjh.getBlackjackValue()`. The first two methods are defined in *Hand*, but are inherited by *BlackjackHand*.

Variables and methods from the *Hand* class are inherited by *BlackjackHand*, and they can be used in the definition of *BlackjackHand* just as if they were actually defined in that class (except for any that are declared to be `private`, which prevents access even by subclasses). The statement "`cards = getCardCount();`" in the above definition of `getBlackjackValue()` calls the instance method`getCardCount()`, which was defined in *Hand*.

Extending existing classes is an easy way to build on previous work. We'll see that many standard classes have been written specifically to be used as the basis for making subclasses.

Access modifiers such as `public` and `private` are used to control access to members of a class. There is one more access modifier, protected, that comes into the picture when subclasses are taken into consideration. When `protected` is applied as an access modifier to a method or member variable in a class, that member can be used in subclasses -- direct or indirect -- of the class in which it is defined, but it cannot be used in non-subclasses. (There is an exception: A `protected` member can also be accessed by any class in the same package as the class that contains the protected member. Recall that using no access modifier makes a member accessible to classes in the same package, and nowhere else. Using the `protected` modifier is strictly more liberal than using no modifier at all: It allows access from classes in the same package and from **subclasses** that are not in the same package.)

When you declare a method or member variable to be `protected`, you are saying that it is part of the implementation of the class, rather than part of the public interface of the class. However, you are allowing subclasses to use and modify that part of the implementation.

For example, consider a *PairOfDice* class that has instance variables `die1` and `die2` to represent the numbers appearing on the two dice. We could make those variables `private` to make it impossible to change their values from outside the class, while still allowing read access through getter methods. However, if we think it possible that *PairOfDice* will be used to create subclasses, we might want to make it possible for subclasses to change the numbers on the dice. For example, a *GraphicalDice* subclass that draws the dice might want to change the numbers at other times besides when the dice are rolled. In that case, we could make `die1` and `die2` `protected`, which would allow the subclass to change their values without making them public to the rest of the world. (An even better idea would be to define `protected`setter methods for the variables. A setter method could, for example, ensure that the value that is being assigned to the variable is in the legal range 1 through 6.)

`Source : http://math.hws.edu/javanotes/c5/s5.html`