Java Inheritance

Abstract Classes: Inheritance in Java

While the `getRank` and `getSuit` methods are different for each class that implements `Card` notice that the `toString` method is identical; in this case, because it only uses methods defined in the interface. A good coding practice is to always attempt to *factor out* common code and put it in one place; this avoids having to maintain the same code in multiple places. Java provides a way to do this: *abstract classes*.

An abstract class is merely one which is declared with the Java keyword `abstract`. Like an interface, an abstract class cannot instantiate objects, but unlike an interface, an abstract class can have its own instance variables, and implement methods. These features make abstract classes good intermediates between interfaces and complete (non-abstract) class declarations: Any method that has the same implementation across all classes that implement the interface can be put in the abstract class*.

Below we show an abstract base class for our card example, and describe how its presence changes the classes that fully implement `Card`.

```
public abstract class CardAbstract implements Card
{
    public String toString() {
        return getRank() + " of " + getSuit();
    }
}
```

We want to position this class "between" the `Card` interface and the full implementations: `CardRankSuit`, `Card52`, etc.. We do this by indicating that `CardAbstract` implements `Card`†. The only method that can be factored out in this example is the `toString` method, so we include it here and remove it from the individual implementing classes. We also need to indicate that the implementing classes `CardRankSuit, Card52`, etc. know about and can use the code in `CardAbstract`. We do this by saying that these classes *extend* `CardAbstract`. We no longer have to state that `CardRankSuit, Card52`, etc. `implement Card`; because they are extending a class that implements `Card`, they themselves automatically are classes that implement `Card`. Other than removing the `toString` method and altering the "class" statement in each of `CardRankSuit, Card52`, etc., no further modifications are needed; the code for these classes is available on the course web site.

This decomposition of our code into an interface, one (or more) abstract classes, and one or more fully implemented classes will appear repeatedly throughout the semester as we design our data structures.

Extending Non-Abstract Classes

Continuing our playing card example, supposed we decided that for some applications we wanted each of our cards to be able to store a point value. How might we take advantage of our previous coding work? Java allow us to extend classes, adding new instance variables and methods. Here is an implementation of a class `CardRankSuitPoints` that extends `CardRankSuit` so that each card now has a point value.

```
public class CardRankSuitPoints extends CardRankSuit
{
    // "protected" means other classes can't access them
    //  (data hiding)
```

*A class playing this role is sometimes called an *abstract base class*.

†Clearly it only partially implements `Card`.

```
    // instance variables
    protected int points;

    // Constructors

    // Constructs a card of the given type
    public CardRankSuitPoints( Rank theRank, Suit theSuit,
                               int pointVal) {
        super(theRank, theSuit);
        points = pointVal;
    }

    // Constructs a card of the given type
    public CardRankSuitPoints( Rank theRank, Suit theSuit ) {
        super(theRank, theSuit);
        // Default point value is "face" value
        points = 2 + theRank.ordinal();
    }

    // returns point value of card
    public int getPoints() {
        return points;
    }

    // Note: Probably don't want to add point value
    // to String representation of card, but it's done
    // to illustrate the overriding of a method
    public String toString() {
        return super.toString() + " (" + points + " points)";
    }

    public static void main(String s[]) {
        Card ace = new CardRankSuitPoints( Rank.ACE, Suit.SPADES, 20 );
        Card three = new CardRankSuitPoints( Rank.THREE, Suit.DIAMONDS );

        System.out.println(ace);
        System.out.println(three);
    }
}
```

Here are the salient features of this code:

- The phrase `extends CardRankSuit` is included in the `class` statement,
- A new protected instance variable `points` is added,
- The features of `CardRankSuit`—instance variables, methods—are *inherited* from `CardRankSuit` and so do not need to be rewritten here (unless it is desired to change their behavior),
- The constructor is modified to allow a parameter to pass in a point value for a card; that constructor, through the keyword `super` begins by calling the constructor for `CardRankSuit`,
- A second constructor allows for a default point setting, so no point value parameter needed,
- A new method `getPoints` is added, and the method `toString` is *overridden* so that when an object of type `CardRankSuitPoints` is converted to a `String`, the point value is included.

Hopefully this extended example has given you a sense of how combining interfaces, inheritance, and abstract classes provides support for flexible and efficient development of modular and reusable code. Before we end, let's put the code to use by creating a rudimentary deck of cards class. An object created by the `CardDeck` class provides a deck of 52 standard playing cards. When created, the deck is sorted from the 2 of clubs to the ace of spades. There is also a method, `shuffle` that randomly permutes the order of the cards in the deck, as well as a `toString` method. Finally, the method `main` creates a deck of cards, prints it out, then repeatedly shuffles it until an ace appears as the "top" card on the deck, reporting how many shuffles were needed and printing the deck in order.

```
public class CardDeck {

    static protected final int NUM_CARDS = 52;

    protected Card[] cards;
    protected Random gen;
    /*
     * Create a new random deck
     */
    public CardDeck() {
        // allocate array and create random number generator
        cards = new Card[NUM_CARDS];
        gen = new Random();

        int count = 0;
        for( Suit s : Suit.values() )
            for( Rank r : Rank.values() ) {
                cards[count] = new Card413( r, s );
                count++;
            }
        shuffle();
    }

   public void shuffle() {

        for (int remaining = cards.length; remaining > 1; remaining--) {
            int i = gen.nextInt(remaining);
            Card toMove = cards[i];
            cards[i] = cards[remaining-1];
            cards[remaining-1] = toMove;
        }
    }

    /*
     * Returns a string representation of the deck
     */
    public String toString() {
        String result = "";
        for (int i = 0; i < cards.length; i++) {
            result = result + cards[i] + "\n";
        }
        return result;
    }

    /*
     * Return true when top card is an ace
     */
```

```
    protected boolean isAceOnTop() {
        return Rank.ACE == cards[0].getRank();
    }

    public static void main(String s[]) {
        CardDeck deck = new CardDeck();
        System.out.println();
        System.out.println(deck);

        int count = 0;
        while(!deck.isAceOnTop()) {
            System.out.println("Not yet...");
            count++;
            deck.shuffle();
        }
        System.out.println("Deck #:" + count + " has an ace on top!");
        System.out.println(deck);
    }
}
```

Aspects of `CardDeck` worth noting

- The deck is held as an array of `Card`s; only when each card is created do we need to commit to a particular implementation,
- The shuffle method is cute: It picks a random position from 0 to 51 and swaps the card in that position with the card in position 51, then it picks a random position from 0 to 50 and swaps the card in that position with the card in position 50, and so forth. This is much more efficient than, say, generating new cards at random, while making sure that the same card is not generated more than once!

Some Final Notes: Testing Object Equality and the Object Class

Consider the following code fragment:

```
        Card a = new Card(Rank.KING, Suit.DIAMONDS);
        Card b = new Card(Rank.KING, Suit.DIAMONDS);
        Card c = a;
        System.out.println(a == c);
        System.out.println(a == b);
```

What does this code print? Not surprisingly, the first `println` will produce `true`; the second, however, will produce `false`. Why? While cards `a` and `b` both represent the king of diamonds, they are two different objects and the equality operator, when applied to class types, is checking equality of the *references*, not the actual contents of the objects. But we would often like to be able to determine whether two different objects of the same type have the same value (e.g., whether two different card objects both represent the king of diamonds). Java provides a method for this: the `equals` method. Adding the `equals` method to, say, `CardRankSuit` as illustrated below, allows us to test for the kind of equality (equivalent values in different cards) that we desire

```
public boolean equals(Object other) {
        Card c = (Card) other;
        return c.getRank() == this.rank && c.getSuit() == this.suit;
}
```

Now we can check equality of our cards by writing, say,

```
        if ( myCard.equals(yourCard) ) \{ ... \}
```

In fact, if we rewrote the code as follows

```
public boolean equals(Object other) {
        Card c = (Card) other;
        return c.getRank() == this.getRank() && c.getSuit() == this.getSuit();
}
```

we could then move the code into the abstract base class `CardAbstract` and wouldn't need to include it in all of the separate classes that implement the `Card` interface.

There are a number of situations in which some Java library class, for example the Java Collections classes, will use the `equals` method when comparing objects, so it is a good idea to add this method to any classes you write for which reference equality is *not* the appropriate version of equality for your class.

You might wonder why we made the type of the parameter `other` be `Object` rather than `Card`. The reason is the following. All class types in Java extend by default the class `Object`; it is the simplest reference type. The class `Object` includes a handful of methods, among them `toString` and `equals` that are therefore inherited by all class types. Because these methods have to work for every class-based type and yet still have the same method signature, the parameter to `equals` must be of type `Object`. In order to use the `Card` methods `getRank` and `getSuit` on the parameter, it must be cast from type `Object` to type `Card`.

What would happen if someone passed some object that was not of type `Card` to the `equals` method, say `myCard.equals(someStudent)`? The attempt to cast `other` of type `Student` to type `Card` would produce a run-time error. Java is a *type-safe* language and will only permit the casting of a value to its actual type or of a type that it extends. To avoid the run-time error, we could modify `equals` as follows:

```
public boolean equals(Object other) {
        if( other instanceof Card ) {
                Card c = (Card) other;
                return c.getRank() == this.getRank() &&
                        c.getSuit() == this.getSuit();
        }
        else return false;
}
```

Clearly if other is not even a `Card`, it can't be equal to a value that is a `Card`!