JAVA PROGRAMMING EXAMPLE: CARD, HAND, DECK

In this section, we look at some specific examples of object-oriented design in a domain that is simple enough that we have a chance of coming up with something reasonably reusable. Consider card games that are played with a standard deck of playing cards (a so-called "poker" deck, since it is used in the game of poker).

5.4.1 Designing the classes

In a typical card game, each player gets a hand of cards. The deck is shuffled and cards are dealt one at a time from the deck and added to the players' hands. In some games, cards can be removed from a hand, and new cards can be added. The game is won or lost depending on the value (ace, 2, ..., king) and suit (spades, diamonds, clubs, hearts) of the cards that a player receives. If we look for nouns in this description, there are several candidates for objects: game, player, hand, card, deck, value, and suit. Of these, the value and the suit of a card are simple values, and they might just be represented as instance variables in a *Card* object. In a complete program, the other five nouns might be represented by classes. But let's work on the ones that are most obviously reusable: card, hand, and deck.

If we look for verbs in the description of a card game, we see that we can shuffle a deck and deal a card from a deck. This gives use us two candidates for instance methods in a *Deck* class: `shuffle()` and`dealCard()`. Cards can be added to and removed from hands. This gives two candidates for instance methods in a *Hand* class: `addCard()` and `removeCard()`. Cards are relatively passive things, but we need to be able to determine their suits and values. We will discover more instance methods as we go along.

First, we'll design the deck class in detail. When a deck of cards is first created, it contains 52 cards in some standard order. The *Deck* class will need a constructor to create a new deck. The constructor needs no parameters because any new deck is the same as any other. There will be an instance method called `shuffle()` that will rearrange the 52 cards into a random order. The `dealCard()` instance method will get the next card from the deck. This will be a function with a return type of `Card`, since the caller needs to know what card is being dealt. It has no parameters -- when you deal the next card from the deck, you don't provide any information to the deck; you just get the next card, whatever it is. What will happen if there are no more cards in the deck when its `dealCard()` method is called? It should probably be considered an error to try to deal a card from an empty deck, so the deck can throw an exception in that case. But this raises another question: How will the rest of the program know whether the deck is empty? Of course, the program could keep track of how many cards it has used. But the deck itself should know how many cards it has left, so the program should just be able to ask the deck object. We can make this possible by specifying another instance method, `cardsLeft()`, that returns the number of cards remaining in the deck. This leads to a full specification of all the subroutines in the *Deck* class:

```
Constructor and instance methods in class Deck:

     /**
      * Constructor.  Create an unshuffled deck of cards.
      */
     public Deck()

     /**
      * Put all the used cards back into the deck,
      * and shuffle it into a random order.
      */
     public void shuffle()

     /**
      * As cards are dealt from the deck, the number of
      * cards left decreases.  This function returns the
      * number of cards that are still left in the deck.
      */
     public int cardsLeft()

     /**
      * Deals one card from the deck and returns it.
      *  @throws  IllegalStateException  if  no  more  cards  are
left.
      */
     public Card dealCard()
```

This is everything you need to know in order to use the `Deck` class. Of course, it doesn't tell us how to write the class. This has been an exercise in design, not in coding. In fact, writing the class involves a programming technique, arrays, which will not be covered until Chapter 7. Nevertheless, you can look at the source code, *Deck.java*, if you want. Even though you won't understand the implementation, the Javadoc comments give you all the information that you need to understand the interface. With this information, you can use the class in your programs without understanding the implementation.

We can do a similar analysis for the *Hand* class. When a hand object is first created, it has no cards in it. An `addCard()` instance method will add a card to the hand. This method needs a parameter of type *Card* to specify which card is being added. For the `removeCard()` method, a parameter is needed to specify which card to remove. But should we specify the card itself ("Remove the ace of spades"), or should we specify the card by its position in the hand ("Remove the third card in the hand")? Actually, we don't have to decide, since we can allow for both options. We'll have two `removeCard()` instance methods, one with a parameter of type *Card* specifying the card to be removed and one with a parameter of type int specifying the position of the card in the hand. (Remember that you can have two methods in a class with the same name, provided they have different numbers or types of parameters.) Since a hand can contain a variable number of cards, it's convenient to be able to ask a hand object how many cards it contains. So, we need an instance method `getCardCount()` that returns the number of cards in the hand. When I play cards, I like to arrange the cards in my hand so that cards of the same value are next to each other. Since this is a generally useful thing to be able to do, we can provide instance methods for sorting the cards in the hand. Here is a full specification for a reusable *Hand* class:

```
Constructor and instance methods in class Hand:

    /**
     * Constructor. Create a Hand object that is initially
empty.
     */
    public Hand()

    /**
     * Discard all cards from the hand, making the hand empty.
     */
    public void clear()

    /**
     * Add the card c to the hand.  c should be non-null.
     * @throws NullPointerException if c is null.
     */
    public void addCard(Card c)

    /**
     * If the specified card is in the hand, it is removed.
     */
    public void removeCard(Card c)

    /**
     * Remove the card in the specified position from the
     * hand.  Cards are numbered counting from zero.
     * @throws IllegalArgumentException if the specified
     *    position does not exist in the hand.
     */
    public void removeCard(int position)
```

```
        /**
         * Return the number of cards in the hand.
         */
        public int getCardCount()

        /**
         * Get the card from the hand in given position, where
         * positions are numbered starting from 0.
         * @throws IllegalArgumentException if the specified
         *    position does not exist in the hand.
         */
        public Card getCard(int position)

        /**
         * Sorts the cards in the hand so that cards of the same
         * suit are grouped together, and within a suit the cards
         * are sorted by value.
         */
        public void sortBySuit()

        /**
         *  Sorts  the  cards  in  the  hand  so  that  cards  are  sorted
    into
         * order of increasing value.  Cards with the same value
         * are sorted by suit. Note that aces are considered
         * to have the lowest value.
         */
        public void sortByValue()
```

Again, you don't yet know enough to implement this class. But given the source code, Hand.java, you can use the class in your own programming projects.

5.4.2 The Card Class

We **have** covered enough material to write a *Card* class. The class will have a constructor that specifies the value and suit of the card that is being created. There are four suits, which can be represented by the integers 0, 1, 2, and 3. It would be tough to remember which number represents which suit, so I've defined named constants in the *Card* class to represent the four possibilities. For example, `Card.SPADES` is a constant that represents the suit, spades. (These constants are declared to be `public` `final` `static` `ints`. It might be better to use an enumerated type, but for now we will stick to integer-valued constants. I'll return to the question of using enumerated types in this example at the end of the chapter.) The possible values of a card are the numbers 1, 2, ..., 13, with 1 standing for an ace, 11 for a jack, 12 for a queen, and 13 for a king. Again, I've defined some named constants to represent the values of aces and face cards. (When you read the *Card* class, you'll see that I've also added support for Jokers.)

A *Card* object can be constructed knowing the value and the suit of the card. For example, we can call the constructor with statements such as:

```
card1 = new Card( Card.ACE, Card.SPADES );  // Construct ace of
spades.
card2 = new Card( 10, Card.DIAMONDS );    // Construct 10 of
diamonds.
card3 = new Card( v, s );  // This is OK, as long as v and s
                           //                    are integer
expressions.
```

A *Card* object needs instance variables to represent its value and suit. I've made these `private` so that they cannot be changed from outside the class, and I've provided getter methods `getSuit()` and`getValue()` so that it will be possible to discover the suit and value from outside the class. The instance variables are initialized in the constructor, and are never changed after that. In fact, I've declared the instance variables `suit` and `value` to be `final`, since they are never changed after they are initialized. (An instance variable can be declared `final` provided it is either given an initial value in its declaration or is initialized in every constructor in the class.)

Finally, I've added a few convenience methods to the class to make it easier to print out cards in a human-readable form. For example, I want to be able to print out the suit of a card as the word "Diamonds", rather than as the meaningless code number 2, which is used in the class to represent diamonds. Since this is something that I'll probably have to do in many programs, it makes sense to include support for it in the class. So, I've provided instance methods `getSuitAsString()` and `getValueAsString()` to return string representations of the suit and value of a card. Finally, I've defined the instance method `toString()`to return a string with both the value and suit, such as "Queen of Hearts". Recall that this method will be used automatically whenever a *Card* needs to be converted into a *String*, such as when the card is concatenated onto a string with the + operator. Thus, the statement

```
System.out.println( "Your card is the " + card );
```

is equivalent to

```
System.out.println( "Your card is the " + card.toString() );
```

If the card is the queen of hearts, either of these will print out "Your card is the Queen of Hearts".

Here is the complete *Card* class. It is general enough to be highly reusable, so the work that went into designing, writing, and testing it pays off handsomely in the long run.

```
/**
 * An object of type Card represents a playing card from a
 * standard Poker deck, including Jokers.  The card has a suit,
which
 * can be spades, hearts, diamonds, clubs, or joker.  A spade,
heart,
 * diamond, or club has one of the 13 values: ace, 2, 3, 4, 5,
6, 7,
 * 8, 9, 10, jack, queen, or king.  Note that "ace" is
considered to be
 * the smallest value.  A joker can also have an associated
value;
 * this value can be anything and can be used to keep track of
several
 * different jokers.
 */

public class Card {

   public final static int SPADES = 0;    // Codes for the 4
suits, plus Joker.
   public final static int HEARTS = 1;
   public final static int DIAMONDS = 2;
   public final static int CLUBS = 3;
   public final static int JOKER = 4;

   public final static int ACE = 1;       // Codes for the non-
numeric cards.
   public final static int JACK = 11;     //   Cards 2 through
10 have their
   public final static int QUEEN = 12;    //   numerical values
for their codes.
   public final static int KING = 13;

   /**
    * This card's suit, one of the constants SPADES, HEARTS,
DIAMONDS,
    * CLUBS, or JOKER.  The suit cannot be changed after the
card is
    * constructed.
    */
   private final int suit;

   /**
    * The card's value.  For a normal card, this is one of the
values
    * 1 through 13, with 1 representing ACE.  For a JOKER, the
value
    * can be anything.  The value cannot be changed after the
card
    * is constructed.
    */
   private final int value;

   /**
    * Creates a Joker, with 1 as the associated value.  (Note
that
    * "new Card()" is equivalent to "new Card(1,Card.JOKER)".)
    */
   public Card() {
      suit = JOKER;
      value = 1;
   }

   /**
    * Creates a card with a specified suit and value.
    * @param theValue the value of the new card.  For a regular
card (non-joker),
    * the value must be in the range 1 through 13, with 1
representing an Ace.
    * You can use the constants Card.ACE, Card.JACK,
Card.QUEEN, and Card.KING.
    * For a Joker, the value can be anything.
    * @param theSuit the suit of the new card.  This must be
one of the values
    * Card.SPADES, Card.HEARTS, Card.DIAMONDS, Card.CLUBS, or
Card.JOKER.
    * @throws IllegalArgumentException if the parameter values
are not in the
    * permissible ranges
    */
   public Card(int theValue, int theSuit) {
      if (theSuit != SPADES && theSuit != HEARTS && theSuit !=
DIAMONDS &&
            theSuit != CLUBS && theSuit != JOKER)
         throw new IllegalArgumentException("Illegal playing
card suit");
      if (theSuit != JOKER && (theValue < 1 || theValue > 13))
         throw new IllegalArgumentException("Illegal playing
card value");
      value = theValue;
      suit = theSuit;
   }

   /**
    * Returns the suit of this card.
    * @returns the suit, which is one of the constants
Card.SPADES,
    * Card.HEARTS, Card.DIAMONDS, Card.CLUBS, or Card.JOKER
    */
   public int getSuit() {
      return suit;
   }

   /**
    * Returns the value of this card.
    * @return the value, which is one of the numbers 1 through
13, inclusive for
```

```
    * a regular card, and which can be any value for a Joker.
    */
   public int getValue() {
      return value;
   }

   /**
    * Returns a String representation of the card's suit.
    *   @return   one   of   the   strings   "Spades",   "Hearts",
"Diamonds", "Clubs"
    * or "Joker".
    */
   public String getSuitAsString() {
      switch ( suit ) {
      case SPADES:   return "Spades";
      case HEARTS:   return "Hearts";
      case DIAMONDS: return "Diamonds";
      case CLUBS:    return "Clubs";
      default:       return "Joker";
      }
   }

   /**
    * Returns a String representation of the card's value.
    *  @return  for  a  regular  card,  one  of  the  strings  "Ace",
"2",
    * "3", ..., "10", "Jack", "Queen", or "King".  For a Joker,
the
    * string is always numerical.
    */
   public String getValueAsString() {
      if (suit == JOKER)
         return "" + value;
      else {
         switch ( value ) {
         case 1:   return "Ace";
         case 2:   return "2";
         case 3:   return "3";
         case 4:   return "4";
         case 5:   return "5";
         case 6:   return "6";
         case 7:   return "7";
         case 8:   return "8";
         case 9:   return "9";
         case 10:  return "10";
         case 11:  return "Jack";
         case 12:  return "Queen";
         default:  return "King";
         }
      }
   }

   /**
    *  Returns  a  string  representation  of  this  card,  including
both
    *  its  suit  and  its  value  (except  that  for  a  Joker  with
value 1,
    * the return value is just "Joker").  Sample return values
    *  are:  "Queen  of  Hearts",  "10  of  Diamonds",  "Ace  of
Spades",
    * "Joker", "Joker #2"
    */
   public String toString() {
      if (suit == JOKER) {
         if (value == 1)
            return "Joker";
         else
            return "Joker #" + value;
      }
      else
         return    getValueAsString()    +    "    of    "    +
getSuitAsString();
   }


} // end class Card
```

5.4.3 Example: A Simple Card Game

I will finish this section by presenting a complete program that uses the *Card* and *Deck* classes. The program lets the user play a very simple card game called HighLow. A deck of cards is shuffled, and one card is dealt from the deck and shown to the user. The user predicts whether the next card from the deck will be higher or lower than the current card. If the user predicts correctly, then the next card from the deck becomes the current card, and the user makes another prediction. This continues until the user makes an incorrect prediction. The number of correct predictions is the user's score.

My program has a static method that plays one game of HighLow. This method has a return value that represents the user's score in the game. The `main()` routine lets the user play several games of HighLow. At the end, it reports the user's average score.

I won't go through the development of the algorithms used in this program, but I encourage you to read it carefully and make sure that you understand how it works. Note in particular that the subroutine that plays one game of HighLow returns the user's score in the game as its return value. This gets the score back to the main program, where it is needed. Here is the program:

```
/**
 * This program lets the user play HighLow, a simple card game
 * that is described in the output statements at the beginning
of
 * the main() routine.  After the user plays several games,
 * the user's average score is reported.
 */
```

```
public class HighLow {


   public static void main(String[] args) {

      System.out.println("This program lets you play the simple
card game,");
      System.out.println("HighLow.  A card is dealt from a deck
of cards.");
      System.out.println("You have to predict whether the next
card will be");
      System.out.println("higher or lower.  Your score in the
game is the");
      System.out.println("number of correct predictions you
make before");
      System.out.println("you guess wrong.");
      System.out.println();

      int gamesPlayed = 0;       // Number of games user has
played.
      int sumOfScores = 0;       // The sum of all the scores
from
                                 //      all the games played.
      double averageScore;       // Average score, computed by
dividing
                                 //          sumOfScores by
gamesPlayed.
      boolean playAgain;         // Record user's response when
user is
                                 //    asked whether he wants to
play
                                 //   another game.

      do {
         int scoreThisGame;        // Score for one game.
         scoreThisGame = play();   // Play the game and get the
score.
         sumOfScores += scoreThisGame;
         gamesPlayed++;
         TextIO.put("Play again? ");
         playAgain = TextIO.getlnBoolean();
      } while (playAgain);

      averageScore = ((double)sumOfScores) / gamesPlayed;

      System.out.println();
      System.out.println("You played " + gamesPlayed + "
games.");
      System.out.printf("Your average score was %1.3f.\n",
averageScore);

   }  // end main()


   /**
    * Lets the user play one game of HighLow, and returns the
    * user's score on that game.  The score is the number of
    * correct guesses that the user makes.
    */
   private static int play() {

      Deck deck = new Deck();  // Get a new deck of cards, and
                               //   store a reference to it in
                               //   the variable, deck.

      Card currentCard;   // The current card, which the user
sees.

      Card nextCard;   // The next card in the deck.  The user
tries
                      //     to predict whether this is higher
or lower
                      //    than the current card.

      int correctGuesses ;    // The number of correct
predictions the
                           //    user has made.  At the end of
the game,
                           //   this will be the user's score.

      char guess;    // The user's guess.  'H' if the user
predicts that
                    //    the next card will be higher, 'L' if
the user
                    //   predicts that it will be lower.

      deck.shuffle();  // Shuffle the deck into a random order
before
                      //    starting the game.

      correctGuesses = 0;
      currentCard = deck.dealCard();
      TextIO.putln("The first card is the " + currentCard);

      while (true) {   // Loop ends when user's prediction is
wrong.

         /* Get the user's prediction, 'H' or 'L' (or 'h' or
'l'). */

         TextIO.put("Will the next card be higher (H) or lower
(L)?  ");
         do {
             guess = TextIO.getlnChar();
             guess = Character.toUpperCase(guess);
             if (guess != 'H' && guess != 'L')
                TextIO.put("Please respond with H or L:  ");
         } while (guess != 'H' && guess != 'L');

         /* Get the next card and show it to the user. */

         nextCard = deck.dealCard();
         TextIO.putln("The next card is " + nextCard);
```

```
          /* Check the user's prediction. */

          if (nextCard.getValue() == currentCard.getValue()) {
             TextIO.putln("The value is the same as the previous
card.");
             TextIO.putln("You lose on ties.  Sorry!");
             break;  // End the game.
          }
          else if (nextCard.getValue() > currentCard.getValue())
{
             if (guess == 'H') {
                 TextIO.putln("Your prediction was correct.");
                 correctGuesses++;
             }
             else {
                 TextIO.putln("Your prediction was incorrect.");
                 break;  // End the game.
             }
          }
          else {  // nextCard is lower
             if (guess == 'L') {
                 TextIO.putln("Your prediction was correct.");
                 correctGuesses++;
             }
             else {
                 TextIO.putln("Your prediction was incorrect.");
                 break;  // End the game.
             }
          }

          /* To set up for the next iteration of the loop, the
nextCard
             becomes the currentCard, since the currentCard has
to be
             the card that the user sees, and the nextCard will
be
             set to the next card in the deck after the user
makes
             his prediction.  */

          currentCard = nextCard;
          TextIO.putln();
          TextIO.putln("The card is " + currentCard);

       } // end of while loop

       TextIO.putln();
       TextIO.putln("The game is over.");
       TextIO.putln("You made " + correctGuesses
                                                   +      "      correct
predictions.");
       TextIO.putln();

       return correctGuesses;

    }  // end play()
```

```
        } // end class
```

Source : http://math.hws.edu/javanotes/c5/s4.html