THỰC HÀNH CÔNG CỤ VÀ MÔI TRƯỜNG LẬP TRÌNH 2

Lab 2.2 (4 tiết): Lập tình hướng đối tượng trong Java.

TS. Võ Phương Bình – Email: binhvp@dlu.edu.vn
Information Technology Faculty - Dalat University
Website: http://it.dlu.edu.vn/ivp-lab

A. Mục tiêu:

- Sinh viên tìm hiểu cách chạy ứng dụng Java với công cụ NetBean
- Sinh viên hiểu được các cấu trúc cơ bản nhất của ngôn ngữ Java.
- Sinh viên hiểu thêm được lập trình hướng đối tượng trên ngôn ngữ Java.

B. Kết quả sau khi hoàn thành:

- Hiểu được cách thức và phương pháp sử dụng NetBean.
- Hiểu thêm về lập trình hướng đối tượng với Java.

C. Luyện tập:

Sử dụng môi trường Console như Lab1 để làm các bài tập.

D. Bài tập.

Exercise 2.1:

Write a version of the *PairOfDice* class in which the instance variables `die1` and `die2` are `private`. Your class will need "getter" methods that can be used to find out the values of `die1` and `die2`. (The idea is to protect their values from being changed from outside the class, but still to allow the values to be read.) Include other improvements in the class, including at least a `toString()` method. Test your class with a short program that counts how many times a pair of dice is rolled, before the total of the two dice is equal to two.

Discussion

The versions of the *PairOfDice* class in Section 2 differ in how the dice are initialized. I like the idea of initializing the dice to random values, so I will work with the following version:

```
public class PairOfDice {

    public int die1;   // Number showing on the first die.
    public int die2;   // Number showing on the second die.

    /**
     * Constructor creates a pair of dice and rolls them so that
     * they initially show some random value.
     */
    public PairOfDice() {
        roll();  // Call the roll() method to roll the dice.
    }

    /**
     * Roll the dice by setting each die to be a random number
between 1 and 6.
     */
    public void roll() {
        die1 = (int)(Math.random()*6) + 1;
        die2 = (int)(Math.random()*6) + 1;
    }

} // end class PairOfDice
```

After a *PairOfDice* object has just been created, the number on each die is definitely between 1 and 6, just like the number on a real die. Can we be sure that this will always be true? Not if the instance variables `die1` and `die2` are `public`, since they can be changed from outside the class. There is nothing to stop someone from changing them to 42 and -17 or anything else. It's not good enough to say that you're not supposed to do that with dice. I want an absolute guarantee that my dice objects can only have the values that real dice could have. By making `die1` and `die2` `private`, I can have that guarantee, because the code that I write in the *PairOfDice* class is the only code that will ever affect the values of the variables.

The main program is easy, especially since we've done the same problem before without using objects. Note how the pair of dice object is used. To test whether or not the total on the dice is 2, I use the test "`while (dice.getTotal() != 2)`". Because of the `toString()` method, I just have to say say

```
System.out.println("The dice come up " + dice );
```

to show the numbers on the two dice.

The Solution

The Modified PairOfDice Class

```
     /**
      * An object of class PairOfDice represents a pair of dice,
      * where each die shows a number between 1 and 6.  The dice
      * can be rolled, which randomizes the numbers showing on the
      * dice.
      */
     public class PairOfDice {

        private int die1;   // Number showing on the first die.
        private int die2;   // Number showing on the second die.

        /**
         * Constructor creates a pair of dice and rolls them so that
         * they initially show some random value.
         */
        public PairOfDice() {
            roll();  // Call the roll() method to roll the dice.
        }

        /**
         * Roll the dice by setting each die to be a random number
between 1 and 6.
         */
        public void roll() {
            die1 = (int)(Math.random()*6) + 1;
            die2 = (int)(Math.random()*6) + 1;
        }

        /**
         * Return the number showing on the first die.
         */
        public int getDie1() {
           return die1;
        }

        /**
         * Set the value of the first die.  Throws an
IllegalArgumentException
         * if the value is not in the range 1 to 6.
         */
        public void setDie1( int value ) {
           if ( value < 1 || value > 6 )
              throw new IllegalArgumentException("Illegal dice value " +
value);
           die1 = value;
        }

        /**
```

```
          * Return the number showing on the second die.
          */
         public int getDie2() {
            return die2;
         }

         /**
          * Set the value of the second die.  Throws an
IllegalArgumentException
          * if the value is not in the range 1 to 6.
          */
         public void setDie2( int value ) {
            if ( value < 1 || value > 6 )
               throw new IllegalArgumentException("Illegal dice value " +
value);
            die2 = value;
         }

         /**
          * Return the total showing on the two dice.
          */
         public int getTotal() {
            return die1 + die2;
         }

         /**
          * Return a String representation of a pair of dice, where die1
          * and die2 are instance variables containing the numbers that
are
          * showing on the two dice.
          */
         public String toString() {
            if (die1 == die2)
               return "double " + die1;
            else
               return die1 + " and " + die2;
         }

      }  // end class PairOfDice
```

The Main Program

```
      /**
       * Rolls a pair of dice until the dice come up snake eyes
       * (with a total value of 2).  Counts and reports the
       * number of rolls.
       */
      public class RollFor2 {

         public static void main(String[] args) {

            PairOfDice dice;  // A variable that will refer to the dice.
            int rollCount;    // Number of times the dice have been
rolled.
```

```
           dice = new PairOfDice();  // Create the PairOfDice object.
           rollCount = 0;

           /* Roll the dice until they come up snake eyes. */

           do {
               dice.roll();
               System.out.println("The dice come up " + dice );
               rollCount++;
           } while (dice.getTotal() != 2);

           /* Report the number of rolls. */

           System.out.println("\nIt took " + rollCount + " rolls to get a
2.");

           /* Now, generate an exception. */

           System.out.println();
           System.out.println("This program will now crash with an
error");
           System.out.println("when it tries to set the value of a die to
42.");
           System.out.println();

           dice.setDie1(42);
           System.out.println(dice);  // This statement will not be
executed!

        }

     }  // end class RollFor2
```

Exercise 2.2:

A common programming task is computing statistics of a set of numbers. (A statistic is a number that summarizes some property of a set of data.) Common statistics include the mean (also known as the average) and the standard deviation (which tells how spread out the data are from the mean). I have written a little class called *StatCalc* that can be used to compute these statistics, as well as the sum of the items in the dataset and the number of items in the dataset. You can read the source code for this class in the file *StatCalc.java*. If `calc` is a variable of type *StatCalc*, then the following instance methods are available:

- `calc.enter(item)` where `item` is a number, adds the item to the dataset.
- `calc.getCount()` is a function that returns the number of items that have been added to the dataset.

- `calc.getSum()` is a function that returns the sum of all the items that have been added to the dataset.
- `calc.getMean()` is a function that returns the average of all the items.
- `calc.getStandardDeviation()` is a function that returns the standard deviation of the items.

Typically, all the data are added one after the other by calling the `enter()` method over and over, as the data become available. After all the data have been entered, any of the other methods can be called to get statistical information about the data. The methods `getMean()` and `getStandardDeviation()` should only be called if the number of items is greater than zero.

Modify the current source code, *StatCalc.java*, to add instance methods `getMax()` and `getMin()`. The `getMax()` method should return the largest of all the items that have been added to the dataset, and `getMin()` should return the smallest. You will need to add two new instance variables to keep track of the largest and smallest items that have been seen so far.

Test your new class by using it in a program to compute statistics for a set of non-zero numbers entered by the user. Start by creating an object of type *StatCalc*:

```
StatCalc  calc;   // Object to be used to process the data.
calc = new StatCalc();
```

Read numbers from the user and add them to the dataset. Use 0 as a sentinel value (that is, stop reading numbers when the user enters 0). After all the user's non-zero numbers have been entered, print out each of the six statistics that are available from `calc`.

Discussion

For the *StatCalc* class to handle minimums and maximums, some of what must be added to the class is obvious. We need two new instance variables, `min` and `max`, and two getter methods to return the values of those instance variables. So, we can add these lines to the class definition:

```
private double min;   // Smallest item that has been entered.
private double max;   // Largest item that has been entered.

public double getMin() {
   return min;
}
```

```
public double getMax() {
   return max;
}
```

But then there is the problem of making sure that `min` and `max` have the right values. `min` records the smallest number seen so far. Every time we have a new number to add to the dataset there is a possibility that `min` will change, so we have to compare `min` with the newly added number. If the new number is smaller than the current `min`, then the number becomes the new value of `min` (since the new number is now the smallest number we have seen so far). We do something similar for `max`. This has to be done whenever a number is entered into the dataset, so it has to be added to the `enter()` method, giving:

```
public void enter(double num) {
      // This is not completely correct!
   count++;
   sum += num;
   squareSum += num*num;
   if (num > max)           // We have a new maximum.
      max = num;
   if (num < min)           // We have a new minimum.
      min = num;
}
```

Unfortunately, if this is all we do, there is a **bug in our program**! For example, if the dataset consists of the numbers 21, 17, and 4, the computer will insist that the minimum is 0, rather than 4. The problem is that the variables `min` and `max` are initialized to zero. (If no initial value is provided for a numerical instance variable, it gets the default initial value, zero.) Since `min` is 0, none of the numbers in the dataset pass the test "`if (num < min)`", and therefore the value of `min` never changes. A similar problem holds for `max`, but it will only show up if all the numbers in the dataset are less than zero. For the other instance variables, `count`, `sum`, and `squareSum`, the default initial value of zero is correct. For `min` and `max`, we have to do something different.

One possible way to fix the problem is to treat the first number entered as a special case. When only one number has been entered, it's certainly the largest number so far and also the smallest number so far, so it should be assigned to both `min` and `max`. This can be handled in the `enter()` method:

```
public void enter(double num) {
      // (This is NOT the version I used in my final answer.)
   count++;
   sum += num;
   squareSum += num*num;
   if (count == 1) {  // This is the first number.
```

```
      max = num;
      min = num;
   }
   else {
      if (num > max)      // We have a new maximum.
         max = num;
      if (num < min)      // We have a new minimum.
         min = num;
   }
}
```

This works fine. However, I decided to use an alternative approach. We would be OK if we could initialize `min` to have a value that is bigger than any possible number. Then, when the first number is entered, it definitely will satisfy the test "`if (num < min)`", and therefore the first number will become the value of `min`. But to be "bigger than any possible number," `min` would have to be infinity. The initial value for `max` has to be smaller than any possible number, so `max` has to be initialized to negative infinity. And that's what we'll do!

Recall from Section 2.5 that the standard class *Double* contains constants `Double.POSITIVE_INFINITY` and `Double.NEGATIVE_INFINITY` that represent positive and negative infinity. We can use these named constants to provide initial values for the instance variables `min` and `max`. So, the declarations become:

```
private double max = Double.NEGATIVE_INFINITY;  // Largest item
seen.
private double min = Double.POSITIVE_INFINITY;  // Smallest item
seen.
```

With this change, the *StatCalc* class works correctly. The complete class is shown below. (By the way, you might think about what happens if `getMin()` or `getMax()` is called before any data has been entered. What actually happens? What should happen? What is the minimum or maximum of a set of numbers that contains no numbers at all?)

The main program is fairly straightforward, so just for fun I decided to use a *Scanner* instead of *TextIO* to read the user's input (see Subsection 2.4.6. The user's data are read and entered into the *StatCalc* object in a loop:

```
do {
   System.out.print("? ");
   item = in.nextDouble();  // (in is the Scanner.)
   if (item != 0)
      calc.enter(item);
} while ( item != 0 );
```

The subroutine call "`calc.enter(item);`" enters the user's item. That is, it does all the processing necessary to include this data item in the statistics it is computing. After all the data have been entered, the statistics can be obtained by using function calls such as "`calc.getMean()`". The statistics are output in statements such as:

```
System.out.println("   Average:             " + calc.getMean());
```

Note that a function call represents a value, and so can be used anyplace where a variable or literal value could be used. I don't have to assign the value of the function to a variable. I can use the function call directly in the output statement. Another note: In this program, I decided not to use formatted output, since it seems appropriate to print the answers with as much accuracy as possible. For formatted output, the statement used to print the mean could be something like:

```
System.out.printf("    Average:             %10.3f\n",
calc.getMean());
```

The complete main program is shown below.

Although that completes the exercise, one might wonder: Instead of modifying the source code of *StatCalc*, could we make a subclass of *StatCalc* and put the modifications in that? The answer is yes. We will need need to use the slightly obscure special variable `super` that was discussed in Subsection 5.6.2.

The new instance variables and instance methods can simply be put into the subclass. The problem arises with the `enter()` method. We have to redefine this method so that it will update the values of `min` and `max`. But it also has to do all the processing that is done by the original `enter()` method in the *StatCalc* class. This is what `super` is for. It lets us call a method from the superclass of the class we are writing. So, the subclass can be written:

```
class StatCalcWithMinMax extends StatCalc {

   private double max = Double.NEGATIVE_INFINITY;  // Largest item
seen.
   private double min = Double.POSITIVE_INFINITY;  // Smallest item
seen.

   public void enter(double num) {
      super.enter(num);  // Call the enter method from the StatCalc
class.
      if (num > max)  // Then do the extra processing for min and
max.
         max = num;
      if (num < min)
```

```
         min = num;
   }

   public double getMin() {
      return min;
   }

   public double getMax() {
      return max;
   }

}  // end class StatCalcWithMinMax
```

The Solution

Revised StatCalc Class. Changes from the original are shown in red.

```
     /*
      * An object of class StatCalc can be used to compute several simple
statistics
      * for a set of numbers.  Numbers are entered into the dataset using
      * the enter(double) method.  Methods are provided to return the
following
      * statistics for the set of numbers that have been entered: The
number
      * of items, the sum of the items, the average, the standard
deviation,
      * the maximum, and the minimum.
      */
     public class StatCalc {

        private int count;   // Number of numbers that have been entered.
        private double sum;  // The sum of all the items that have been
entered.
        private double squareSum;  // The sum of the squares of all the
items.
        private double max = Double.NEGATIVE_INFINITY;  // Largest item
seen.
        private double min = Double.POSITIVE_INFINITY;  // Smallest item
seen.

        /**
         * Add a number to the dataset.  The statistics will be computed
for all
         * the numbers that have been added to the dataset using this
method.
         */
        public void enter(double num) {
           count++;
           sum += num;
           squareSum += num*num;
           if (num > max)
              max = num;
           if (num < min)
              min = num;
        }
        /**
         * Return the number of items that have been entered into the
dataset.
         */
        public int getCount() {
           return count;
        }

        /**
         * Return the sum of all the numbers that have been entered.
         */
        public double getSum() {
           return sum;
        }

        /**
         * Return the average of all the items that have been entered.
         * The return value is Double.NaN if no numbers have been
entered.
         */
        public double getMean() {
           return sum / count;
        }

        /**
         * Return the standard deviation of all the items that have been
entered.
         * The return value is Double.NaN if no numbers have been
entered.
         */
        public double getStandardDeviation() {
           double mean = getMean();
           return Math.sqrt( squareSum/count - mean*mean );
        }

        /**
         * Return the smallest item that has been entered.
         * The return value will be infinity if no numbers have been
entered.
         */
        public double getMin() {
           return min;
        }

        /**
         * Return the largest item that has been entered.
         * The return value will be -infinity if no numbers have been
entered.
         */
        public double getMax() {
           return max;
        }

     }  // end class StatCalc
```

Main Program

```
/**
 * Computes and display several statistics for a set of non-zero
 * numbers entered by the user.  (Input ends when user enters 0.)
 * This program uses StatCalc.java.
 */

import java.util.Scanner;

public class SimpleStats {

  public static void main(String[] args) {

     Scanner in = new Scanner(System.in);

     StatCalc calc; // Computes stats for numbers entered by user.
     calc = new StatCalc();

     double item; // One number entered by the user.

     System.out.println("Enter your numbers.  Enter 0 to end.");
     System.out.println();

     do {
        System.out.print("? ");
        item = in.nextDouble();
        if (item != 0)
           calc.enter(item);
     } while (item != 0);

     System.out.println("\nStatistics about your calc:\n");
     System.out.println("   Count:              " +
calc.getCount());
     System.out.println("   Sum:                " + calc.getSum());
     System.out.println("   Minimum:            " + calc.getMin());
     System.out.println("   Maximum:            " + calc.getMax());
     System.out.println("   Average:            " + calc.getMean());
     System.out.println("   Standard Deviation: "
           + calc.getStandardDeviation());

  } // end main()

} // end SimpleStats
```

Exercise 2.3:

A Blackjack hand typically contains from two to six cards. Write a program to test the *BlackjackHand* class. You should create a *BlackjackHand* object and a *Deck* object. Pick a random number between 2 and 6. Deal that many cards from the deck and add them to the hand. Print out all the cards in the hand, and then print out the value computed for the hand by `getBlackjackValue()`. Repeat this as long as the user wants to continue.

Your program will depend on *Card.java*, *Deck.java*, *Hand.java*, and *BlackjackHand.java*.

```
/**
 * 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
```

```
/**
 *  An object of type Deck represents a deck of playing cards.  The deck
 *  is a regular poker deck that contains 52 regular cards and that can
 *  also optionally include two Jokers.
 */
public class Deck {

    /**
     * An array of 52 or 54 cards.  A 54-card deck contains two Jokers,
     * in addition to the 52 cards of a regular poker deck.
     */
    private Card[] deck;

    /**
     * Keeps track of the number of cards that have been dealt from
     * the deck so far.
     */
    private int cardsUsed;

    /**
     * Constructs a regular 52-card poker deck.  Initially, the cards
     * are in a sorted order.  The shuffle() method can be called to
     * randomize the order.  (Note that "new Deck()" is equivalent
     * to "new Deck(false)".)
     */
    public Deck() {
        this(false);  // Just call the other constructor in this class.
    }

    /**
     * Constructs a poker deck of playing cards, The deck contains
     * the usual 52 cards and can optionally contain two Jokers
     * in addition, for a total of 54 cards.   Initially the cards
     * are in a sorted order.  The shuffle() method can be called to
     * randomize the order.
     * @param includeJokers if true, two Jokers are included in the deck; if
false,
     * there are no Jokers in the deck.
     */
    public Deck(boolean includeJokers) {
        if (includeJokers)
            deck = new Card[54];
        else
            deck = new Card[52];
        int cardCt = 0; // How many cards have been created so far.
        for ( int suit = 0; suit <= 3; suit++ ) {
            for ( int value = 1; value <= 13; value++ ) {
                deck[cardCt] = new Card(value,suit);
                cardCt++;
            }
        }
        if (includeJokers) {
            deck[52] = new Card(1,Card.JOKER);
            deck[53] = new Card(2,Card.JOKER);
        }
        cardsUsed = 0;
    }

    /**
```

```
     * Put all the used cards back into the deck (if any), and
     * shuffle the deck into a random order.
     */
    public void shuffle() {
        for ( int i = deck.length-1; i > 0; i-- ) {
            int rand = (int)(Math.random()*(i+1));
            Card temp = deck[i];
            deck[i] = deck[rand];
            deck[rand] = temp;
        }
        cardsUsed = 0;
    }

    /**
     * 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.  The return value would be
     * 52 or 54 (depending on whether the deck includes Jokers)
     * when the deck is first created or after the deck has been
     * shuffled.  It decreases by 1 each time the dealCard() method
     * is called.
     */
    public int cardsLeft() {
        return deck.length - cardsUsed;
    }

    /**
     * Removes the next card from the deck and return it.  It is illegal
     * to call this method if there are no more cards in the deck.  You can
     * check the number of cards remaining by calling the cardsLeft()
function.
     * @return the card which is removed from the deck.
     * @throws IllegalStateException if there are no cards left in the deck
     */
    public Card dealCard() {
        if (cardsUsed == deck.length)
            throw new IllegalStateException("No cards are left in the
deck.");
        cardsUsed++;
        return deck[cardsUsed - 1];
        // Programming note:  Cards are not literally removed from the array
        // that represents the deck.  We just keep track of how many cards
        // have been used.
    }

    /**
     * Test whether the deck contains Jokers.
     * @return true, if this is a 54-card deck containing two jokers, or
false if
     * this is a 52 card deck that contains no jokers.
     */
    public boolean hasJokers() {
        return (deck.length == 54);
    }

} // end class Deck
```

```
/**
 * An object of type Hand represents a hand of cards.  The
 * cards belong to the class Card.  A hand is empty when it
 * is created, and any number of cards can be added to it.
 */

import java.util.ArrayList;

public class Hand {

    private ArrayList<Card> hand;   // The cards in the hand.

    /**
     * Create a hand that is initially empty.
     */
    public Hand() {
        hand = new ArrayList<Card>();
    }

    /**
     * Remove all cards from the hand, leaving it empty.
     */
    public void clear() {
        hand.clear();
    }

    /**
     * Add a card to the hand.  It is added at the end of the current hand.
     * @param c the non-null card to be added.
     * @throws NullPointerException if the parameter c is null.
     */
    public void addCard(Card c) {
        if (c == null)
            throw new NullPointerException("Can't add a null card to a
hand.");
        hand.add(c);
    }

    /**
     * Remove a card from the hand, if present.
     * @param c the card to be removed.  If c is null or if the card is not
in
     * the hand, then nothing is done.
     */
    public void removeCard(Card c) {
        hand.remove(c);
    }

    /**
     * Remove the card in a specified position from the hand.
     * @param position the position of the card that is to be removed, where
     * positions are starting from zero.
     * @throws IllegalArgumentException if the position does not exist in
     * the hand, that is if the position is less than 0 or greater than
     * or equal to the number of cards in the hand.
     */
    public void removeCard(int position) {
        if (position < 0 || position >= hand.size())
            throw new IllegalArgumentException("Position does not exist in
hand: "
                    + position);
        hand.remove(position);
    }

    /**
     * Returns the number of cards in the hand.
     */
    public int getCardCount() {
        return hand.size();
    }

    /**
     * Gets the card in a specified position in the hand.  (Note that this
card
     * is not removed from the hand!)
     * @param position the position of the card that is to be returned
     * @throws IllegalArgumentException if position does not exist in the
hand
     */
    public Card getCard(int position) {
        if (position < 0 || position >= hand.size())
            throw new IllegalArgumentException("Position does not exist in
hand: "
                    + position);
        return hand.get(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.
     * Note that aces are considered to have the lowest value, 1.
     */
    public void sortBySuit() {
        ArrayList<Card> newHand = new ArrayList<Card>();
        while (hand.size() > 0) {
            int pos = 0;  // Position of minimal card.
            Card c = hand.get(0);  // Minimal card.
            for (int i = 1; i < hand.size(); i++) {
                Card c1 = hand.get(i);
                if ( c1.getSuit() < c.getSuit() ||
                        (c1.getSuit() == c.getSuit() && c1.getValue() <
c.getValue()) ) {
                    pos = i;
                    c = c1;
                }
            }
            hand.remove(pos);
            newHand.add(c);
        }
        hand = newHand;
    }
    /**
     * Sorts the cards in the hand so that cards of the same value are
     * grouped together.  Cards with the same value are sorted by suit.
     * Note that aces are considered to have the lowest value, 1.
     */
    public void sortByValue() {
        ArrayList<Card> newHand = new ArrayList<Card>();
        while (hand.size() > 0) {
            int pos = 0;  // Position of minimal card.
            Card c = hand.get(0);  // Minimal card.
            for (int i = 1; i < hand.size(); i++) {
                Card c1 = hand.get(i);
                if ( c1.getValue() < c.getValue() ||
                        (c1.getValue() == c.getValue() && c1.getSuit() <
c.getSuit()) ) {
                    pos = i;
                    c = c1;
                }
            }
            hand.remove(pos);
            newHand.add(c);
        }
        hand = newHand;
    }

}
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
```

Discussion

This problem is mostly a warm-up for the next one. It uses objects of three different types, *Card*, *Deck*, and *BlackjackHand*. The *Hand* class is used indirectly, as the superclass of *BlackjackHand*. To use these objects, you need to know what methods are available in each class, so you should review the information that you have about the classes before beginning the program.

An algorithm for the program is

```
Create a deck
repeat while user wants to continue:
    Shuffle the deck
    Create a new BlackjackHand
    Decide the number of cards in the hand
    Deal cards from the deck into the hand, and print them out
    Display the value of the hand
```

Some variation is possible. You could use just one *BlackjackHand* object, and remove all the cards from it between hands. The *Hand* class includes an instance method, `clear()`, that could be used for this purpose. Similarly, you could create a new *Deck* object each time through the loop. Or, you might want to use one deck and shuffle it only when the number of cards in the deck gets too small. You could say:

```
if (deck.cardsLeft() < 6)
      deck.shuffle();
```

Since we always want to do at least one hand, we can use a `do..while` statement for the loop. Putting in some variable names, we can refine the algorithm to

```
deck  =  new Deck();
do:
    deck.shuffle();
    hand  =  new BlackjackHand();
    cardsInHand = a random number between 2 and 6
    Deal cards from deck into hand, and print them out.
    Display hand.getBlackjackValue()
    Ask if user wants to go again
while user wants to go again
```

The number of cards in the hand is supposed to be a random number between 2 and 6. There are five possible values. The expression "`(int)(Math.random()*5)`" has one of the 5 possible values 0, 1, 2, 3, or 4. Adding 2 to the result gives one of the values 2, 3, 4, 5, or 6. So, the number of cards can be computed as "`2 + (int)(Math.random()*5)`".

Once we know the number of cards, we can use a `for` loop to deal cards into the hand, one at a time. The function call `deck.dealCard()` gets a card from the deck. Once we have a card, we can add it to the hand with the subroutine call `hand.addCard(card)`. This allows us to refine the algorithm to

```
deck  =  new Deck();
do:
    deck.shuffle();
    hand  =  new BlackjackHand();
    cardsInHand = 2 + (int)(Math.random()*5)
    for i = 0 to cardsInHand:
        card  =  deck.dealCard()
        hand.addCard(card)
        Display the card
    Display hand.getBlackjackValue()
    Ask if user wants to go again
while user wants to go again
```

Alternatively, dealing the cards and displaying them could be done in separate `for` loops.

This algorithm can be translated pretty directly into the `main()` routine of the program, which is shown below.

The Solution

```
/**
 * Creates random blackjack hands, with 2 to 6 cards,
 * and prints out the blackjack value of each hand.
 * The user decides when to stop.
 */
public class TestBlackjackHand {
   public static void main(String[] args) {

      Deck deck;              // A deck of cards.
      Card card;              // A card dealt from the deck.
      BlackjackHand hand;     // A hand of from two to six cards.
      int cardsInHand;        // Number or cards in the hand.
      boolean again;          // Set to true if user wants to continue.

      deck = new Deck();      // Create the deck.

      do {
         deck.shuffle();
         hand = new BlackjackHand();
         cardsInHand = 2 + (int)(Math.random()*5);
         System.out.println();
         System.out.println();
         System.out.println("Hand contains:");
         for ( int i = 1; i <= cardsInHand; i++ ) {
                // Get a card from the deck, print it out,
                //   and add it to the hand.
            card = deck.dealCard();
            hand.addCard(card);
            System.out.println("    " + card);
         }
         System.out.println("Value of hand is " +
hand.getBlackjackValue());
         System.out.println();
         System.out.print("Again? ");
         again = TextIO.getlnBoolean();
      } while (again == true);

   }

}  // end class TestBlackjackHand
```

E. Kết quả thực hành.

- Sinh viên thực hành ứng dụng trên Console.
- Thời gian thực hành: 4 tiết.
- Thực hiện xong 2 bài tập với kết quả nhập từ bàn phím.

F. Đánh giá:

- Kiểm tra lại chương trình, thử các kết quả.
- Bắt các lỗi bằng cách sử dụng các phần bắt lỗi: try – catch.

-------------------Hết-------------------