Chapter 6: Arrays
Lab Exercises

Tracking Sales

Copy the code for the `Sales` class below into a file `Sales.java`. `Sales.java` contains a Java program that prompts for and reads in the sales for each of 5 salespeople in a company. Sales are entered as integer values. It then prints out the id and amount of sales for each salesperson and the total sales. Study the code, then compile and run the program to see how it works. Now modify the program as follows:

1. Compute and print the average sales. (You can compute this directly from the `sum` total; no loop is necessary.) Instead of dividing by the number of salespeople, use the `length` instance variable of the array: `sales.length`

2. Find and print the maximum sale. Print both the id of the salesperson with the max sale and the amount of the sale, e.g., "Salesperson 3 had the highest sale with $4500." Note that you don't need another loop for this; you can do it in the same loop where the values are read and the sum is computed.

3. Do the same for the minimum sale.

4. Instead of always reading in 5 sales amounts, at the beginning ask the user for the number of sales people and then create an array that is just the right size. The program can then proceed as before.

```
// ********************************************************************
// Sales.java
//
// Reads in and stores sales for each of 5 salespeople.  Displays
// sales entered by salesperson id and total sales for all salespeople.
//
// ********************************************************************
import java.util.Scanner;

public class Sales
{
    public static void main(String[] args)
    {
      Scanner scan = new Scanner(System.in);

      final int SALESPEOPLE = 5;
      int[] sales = new int[SALESPEOPLE];
      int sum;

      for (int i=0; i < sales.length; i++)
          {
            System.out.print("Enter sales for salesperson " + i + ": ");
            sales[i] = scan.nextInt();
          }

      System.out.println("\nSalesperson   Sales");
      System.out.println("--------------------");
      sum = 0;
      for (int i=0; i < sales.length; i++)
          {
            System.out.println("\t " + i + "\t\t\t" + sales[i]);
            sum += sales[i];
          }

      System.out.println("\nTotal sales: " + sum);
    }
}
```

A Shopping Cart

In this exercise you will complete a class that implements a shopping cart as an array of items. Copy the code below for the `Item` class and `ShoppingCart` class. The file `Item.java` contains the definition of a class named `Item` that models an item one would purchase. An item has a name, price, and quantity (the quantity purchased). The file `ShoppingCart.java` implements the shopping cart as an array of `Item` objects.

1. Complete the `ShoppingCart` class by doing the following:

   a. Declare an instance variable `cart` to be an array of `Items` and instantiate `cart` in the constructor to be an array holding `capacity Items`. (Note: `capacity` is an instance variable initialized to 5)

   b. Fill in the code for the `addToCart` method. This method should add the item to the cart and update the `totalPrice` instance variable (note this variable takes into account the quantity).

   c. Compile your class. (Note: No tester or driver class has been written yet. You are checking for syntax errors in your `ShoppingCart` class.)

2. Write a program `ShopTest` that simulates shopping. The program should have a loop that continues as long as the user wants to shop. Each time through the loop read in the name, price, and quantity of the item the user wants to add to the cart. After adding an item to the cart, the cart contents should be printed. Be sure not to add more than 5 items to your cart.

   a) Add a method `getTotalPrice` to the `ShoppingCart` class which returns the `totalPrice` of the cart. After the loop print a "Please pay ..." message with the total price of the items in the cart.

3. Use the following test data to test your program. Note the following example has **4 items** (quantity of each items is a different value):

Item	**Quantity**	**Price**
milk	2	3.56
donuts	12	0.47
bread	1	4.24
oranges	5	0.62

```
Please pay $20.10
```

```
// ************************************************************
//   Item.java
//
//   Represents an item in a shopping cart.
// ************************************************************

import java.text.NumberFormat;

public class Item
{
    private String name;
    private double price;
    private int quantity;

    // -----------------------------------------------------
    //  Create a new item with the given attributes.
    // -----------------------------------------------------
    public Item (String itemName, double itemPrice, int numPurchased)
    {
      name = itemName;
```

```
    price = itemPrice;
    quantity = numPurchased;
  }

  // ------------------------------------------------------
  //   Return a string with the information about the item
  // ------------------------------------------------------
  public String toString ()
  {
    NumberFormat fmt = NumberFormat.getCurrencyInstance();

    return (name + "\t" + fmt.format(price) + "\t" + quantity + "\t"
          + fmt.format(price*quantity));
  }

  // ------------------------------------------------
  //   Returns the unit price of the item
  // ------------------------------------------------
  public double getPrice()
  {
    return price;
  }

  // ------------------------------------------------
  //   Returns the name of the item
  // ------------------------------------------------
  public String getName()
  {
    return name;
  }

  // ------------------------------------------------
  //   Returns the quantity of the item
  // ------------------------------------------------
  public int getQuantity()
  {
    return quantity;
  }
}
```

```
// ********************************************************************
//   ShoppingCart.java
//
//   Represents a shopping cart as an array of items
// ********************************************************************

import java.text.NumberFormat;

public class ShoppingCart
{
    private int itemCount;      // total number of items in the cart
    private double totalPrice;  // total price of items in the cart
    private int capacity;       // current cart capacity

    // -----------------------------------------------------------
    //  Creates an empty shopping cart with a capacity of 5 items.
    // -----------------------------------------------------------
    public ShoppingCart()
    {
      capacity = 5;
      itemCount = 0;
      totalPrice = 0.0;
    }

    // -----------------------------------------------------------
    //  Adds an item to the shopping cart.
    // -----------------------------------------------------------
    public void addToCart(String itemName, double price, int quantity)
    {
    }

    // -----------------------------------------------------------
    //  Returns the contents of the cart together with
    //  summary information.
    // -----------------------------------------------------------
    public String toString()
    {
      NumberFormat fmt = NumberFormat.getCurrencyInstance();

      String contents = "\nShopping Cart\n";
      contents += "\nItem\t\tUnit Price\tQuantity\tTotal\n";

      for (int i = 0; i < itemCount; i++)
          contents += cart[i].toString() + "\n";

      contents += "\nTotal Price: " + fmt.format(totalPrice);
      contents += "\n";

      return contents;
    }

}
```

A Flexible Shopping Cart

In the previous exercise, your `ShoppingCart` was limited to 5 (capacity) items. Copy `ShoppingCart.java` into `ShoppingCart2.java`. In this exercise you will add a method to the `ShoppingCart2` class to increase the capacity.

1. Complete the `ShoppingCart2` class by doing the following:

   a. Add a method `increaseSize`. This method will increase the size of cart by 3.
      - Create a temporary cart that is 3 items bigger than `cart`.
      - Write a for loop to loop through the `cart` array, adding the items to the temporary cart.
      - Write a statement to change the address of `cart` to the address of the temporary cart. Now `cart` is 3 items bigger.

   b. Add logic to the `addToCart` method so that if the user adds an item to a full cart, the `increaseSize` method is called. The item is then added.

2. Test your changes. Try adding more than 5 items to your cart. Use the following test data:

Item	Quantity	Price
milk	2	3.56
donuts	12	0.47
bread	1	4.24
oranges	5	0.62
butter	1	3.24
yogurt	6	0.82
pepsi	1	2.45

```
Please pay $30.71
```

```
// ---------------------------------------------------------
//  Increases the capacity of the shopping cart by 3
// ---------------------------------------------------------
private void increaseSize()
{



}
```

A Shopping Cart Using the ArrayList Class

In this exercise you will modify `ShoppingCart` to use the `ArrayList` class. Create a new project and copy the file `Item.java` from the previous lab. The class named `Item` models an item one would purchase. An item has a name, price, and quantity (the quantity purchased). Copy `ShoppingCart.java` into `ShoppingCart3.java`. Copy `ShopTest.java` into `ShopTest3.java` and modify to use the `ShoppingCart3` class.

`ShopTest3.java` will behave exactly like `ShopTest2.java`. The difference is the `ShoppingCart3` class will store `Item` objects in an `ArrayList`, rather than an `Array`. Modify all of the methods to use the `ArrayList` syntax for adding and displaying items from the `Cart ArrayList`. Do not forget to import the `ArrayList` class at the top of your `ShoppingCart3` class.

Test your code with the following items:

Item	**Quantity**	**Price**
milk	2	3.56
donuts	12	0.47
bread	1	4.24
oranges	5	0.62
butter	1	3.24
yogurt	6	0.82
pepsi	1	2.45

```
Please pay $30.71
```

Card Game

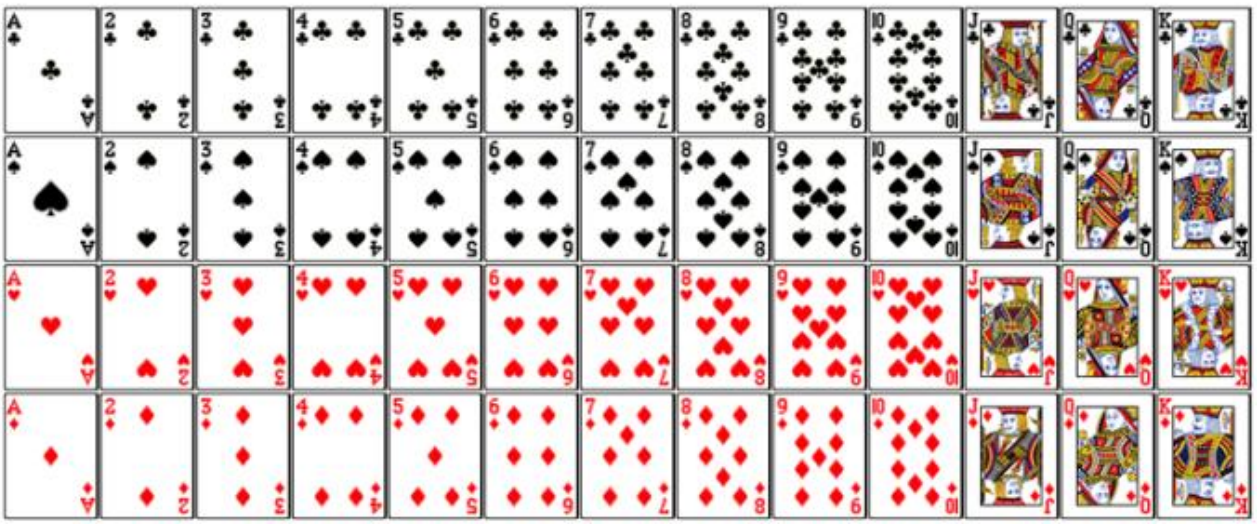
Copy the code for the `Cards` class, `DeckOfCards` class and `DeckOfCardsTest` class. `Cards.java` contains a Java program that contains two `String` variables – `face` and `suit` – to represent a specific `Card`. `DeckOfCards` contains an array of `Card` objects (52 cards in the deck). This class has two methods `shuffle()` which randomly mixes the deck of cards and `dealCard()` which returns the next `Card` to be dealt. The driver program `DeckOfCardsTest` shuffles the deck of cards and prints out each `Card` as it is dealt. Study the code, then compile and run the program to see how it works.

Choose one of the three following card games to implement (poker, blackjack, high-low). Deck of Cards and one of the games will count as one lab.

```
// ********************************************************************
//   Card.java
//
//   Represents a Card with a face and a suit.
// ********************************************************************
public class Card
{
      private String face;
      private String suit;

      public Card (String cardFace, String cardSuit)
      {
            face = cardFace;
            suit = cardSuit;
      }

      public String toString()
      {
            return face + " of " + suit;
      }
}
```

```
// ************************************************************************
//   DeckOfCards.java
//
//   Contains a deck of Card objects.  Methods to shuffle and deal Cards.
// ************************************************************************
public class DeckOfCards {

    private Card deck[];
    private int currentCard;
    private final int NUMBER_OF_CARDS = 52;
    private String faces[] = {"Ace", "Deuce", "Three", "Four", "Five", "Six",
                        "Seven", "Eight", "Nine", "Ten", "Jack", "Queen", "King"};
    private String suits[] = { "Hearts", "Diamonds", "Clubs", "Spades" };

    // -------------------------------------------------------------
    //  Constructor fills a deck array with Card objects.
    // -------------------------------------------------------------

    public DeckOfCards()
    {

      deck = new Card[ NUMBER_OF_CARDS ];
      currentCard = 0; // set currentCard so first Card dealt is deck[0]

      // populate deck with Card objects
      for (int count = 0; count < deck.length; count ++)
            deck [ count ] = new Card(faces[count % 13], suits [count / 13]);

    } // end DeckOfCards constructor

    // -------------------------------------------------------------
    //  Shuffle deck of Cards by randomly switching all cards in deck.
    // -------------------------------------------------------------
    public void shuffle()
    {
      for (int first = 0; first < deck.length; first++)
      {
            // select a random number between 0 and 51
            int second = (int)(Math.random() * 52);

            // swap current Card with randomly selected Card
            Card temp = deck[ first ];
            deck[ first ] = deck[ second ];
            deck[ second ] = temp;
      }
    } // end method shuffle

    // -------------------------------------------------------------
    //  Deals one Card.
    // -------------------------------------------------------------
    public Card dealCard()
    {
      // determine whether Cards remain to be dealt
      if (currentCard < deck.length)
            return deck [ currentCard++ ];
      else
            return null; // return null to indicate no more cards
    }

}
```

```
// ************************************************************************
//   DeckOfCardsTest.java
//
//   Tester program to shuffle and deal a deck of Card objects
// ************************************************************************

public class DeckOfCardsTest
{
      public static void main (String[] args)
      {
            DeckOfCards myDeckOfCards = new DeckOfCards();
            myDeckOfCards.shuffle(); // put Card objects in random order

    // --------------------------------------------------------
    //  print all 52 Cards in the order in which they are dealt
    // --------------------------------------------------------
            for (int i = 0; i < 13; i++)
            {
                  // printf method used for formatting output
                  // print string (%) in a space of 20 characters (-20s)
                  System.out.printf("%-20s%-20s%-20s%-20s\n",
                  myDeckOfCards.dealCard(), myDeckOfCards.dealCard(),
                  myDeckOfCards.dealCard(), myDeckOfCards.dealCard());

            }
      }
}
```

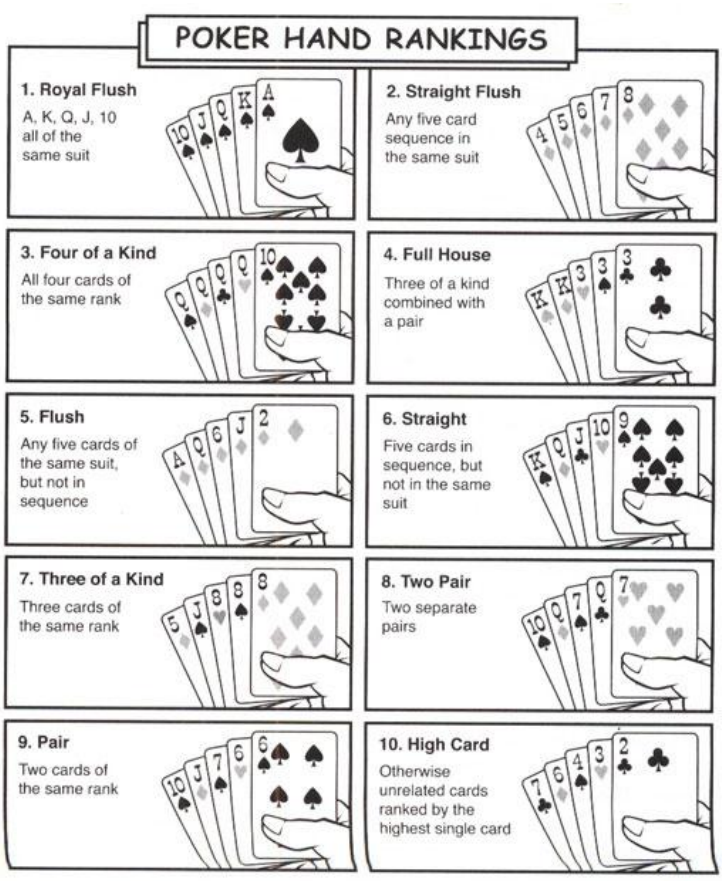


Card Game #1: Five-Card Poker

Write a program using the classes you have already created that deals two five-card poker hands, evaluates each hand and determines which is better. Add the remaining methods to the `DeckOfCard` class or play poker with the hands already defined in the Deck of Cards lab.
Modify the program as follows:

1. Modify the program `DeckOfCardsTest` to deal a five card poker hand into an array `hand[]`. Then modify class `DeckOfCards` to include methods that determine whether a hand contains (pass the `hand[]` array to the method):

   a. a pair
   b. two pairs
   c. three of a kind (e.g., three jacks)
   d. four of a kind (e.g., four aces)
   e. a straight (i.e., five cards of consecutive face values – can have different suits)
   f. a full house (i.e., two cards of one face value and three cards of another face value)



Hint 1: Add methods `getFace` and `getSuit` to class `Card`

Hint 2: Create a method in class `DeckOfCards` to total the hand (`totalHand`). Pass the `hand` array to the method and tally the number of each face in an integer array `numbers`. For example, if the hand contains 2 Queens, and 3 4's, the `numbers` array will contain:

`numbers`

0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	3	0	0	0	0	0	0	2	0

Card Game #2: Blackjack

Write a program using the classes you have already created to play Blackjack. The basic premise of the game is that you want to have a hand value that is closer to 21 than that of the dealer, without going over 21.

In blackjack, the cards are valued as follows:

- An Ace can count as either 1 or 11.
- The cards from 2 through 9 are valued at their face value.
- The 10, Jack, Queen, and King are all valued at 10.

The suits of the cards do not have any meaning in the game. The value of a hand is simply the sum of the point counts of each card in the hand. For example, a hand containing (5,7,9) has the value of 21. The Ace can be counted as either 1 or 11. It's assumed to always have the value that makes the best hand. You may want to modify the `Card` class to contain an `int` variable `value` to hold the card's point value. Add a method `getValue()` to return the `Card's` point value.

Here is how the game is played:

1. The dealer deals himself and the player two cards.
2. Print only one of the dealer's cards. The other is face down. If the dealer has blackjack (21) the dealer wins.
3. Print the players two cards and the total value. If the player has blackjack (21), he wins.
4. Ask the player if he wants to "hit" or "stay". If he chooses "hit", deal another card and display the total value. Keep prompting the user until they select "stay" or their total goes over 21, which is a "bust".
5. The dealer must continue to deal himself cards UNTIL the total is 17 or over. Once the total is 17 or over, he stops dealing cards. Display the total value of the hand or "bust" if the hand is over 21.
6. Compare the values of the dealer and player's hand and display the winner.

Card Game #3: High-Low

Write a program using the classes you have already created to play a game of High-Low. The basic premise of the game is that the highest value card wins. The dealer gets a card and the player gets a card. Display the cards and announce the winner.

In High-Low, the cards are valued as follows:

- An Ace is valued at 11.
- The cards from 2 through 9 are valued at their face value.
- The 10, Jack, Queen, and King are all valued at 10.

For ties in the face of the card, example 2 Kings, compare the suits for a winner in the order from **diamonds** (lowest), followed by **clubs**, **hearts**, and **spades** (highest). Therefore, a King of Spades would beat a King of Clubs.

Finch Plays Simon

Program your Finch to play a modified version of the memory game Simon. In the game you will write, the Finch will print and say an orientation, either Up (for beak up), Down (for beak down), Left (for left wing down), or Right (for right wing down). You will then need to move the Finch to that orientation. If you do this successfully, the Finch will print a new orientation – you‟ll have to move the Finch to the first orientation, then to the new one.

This game goes on for as many moves as you can remember. When you finally mess up, the Finch will tell you how many moves in a row you got right, and either compliment or insult your performance.

Here's the details:

Start a loop that ends when the player makes an incorrect move. The loop should:

- Use the `Math.random()` method to generate a random number from 0 to 3
- Use a control structure to add to an ArrayList (of type String) a String containing the value “Up” for 0, “Down” for 1, “Left” for 2, and “Right” for 3
- Print and say what the move is
- Print and say that the user needs to repeat all of the moves done so far.

At this point, you should create a loop that goes through every move so far, starting with the first one created. This loop checks that a move is correct by calling a second method, `moveCorrect()`.

`moveCorrect()` returns a `Boolean` value, and is passed a `String`. For each of the four possible moves, check if the Finch is in the orientation suggested by the `String` (either Up, Down, Left, or Right). Give the player five seconds to get to the correct move – you can continue checking throughout the five seconds by constantly reading the Finch methods, like `myFinch.isBeakUp()`.

If the player gets the correct move, the Finch‟s beak should flash green for 1 second, the Finch should beep happily, and the `moveCorrect()` method should return true. If they don‟t get it within five seconds, `moveCorrect()` should return false.

If the player correctly gets the whole sequence, your overall loop should continue and generate a new move for them to remember. If they mess up, the program should tell them the maximum number of moves they reached.

The Finch should also render judgment – if the total moves are less than 4, insult them for having a bad memory, if moves are 4-8, tell them to try harder next time, and if it‟s 9 or more, congratulate them.

Notes:

The hardest part of this assignment is the `moveCorrect()` method . You should continuously check the sensors and maybe use some method to check the system time to see if 5 seconds has elapsed.

For example an elapsed time of 5000 would be 5 seconds:

```
long startTime = System.currentTimeMillis();

// do “something” here

long stopTime = System.currentTimeMillis();
long elapsedTime = stopTime - startTime;   // time to execute “something”
```