Chapter 6: Object-Oriented Design Lab Exercises

Topics	Lab Exercises
Parameter Passing	Changing People
Interfaces	Using the Comparable Interface
Method Decomposition	A Modified MiniQuiz Class
Overloading	A Flexible Account Class A Biased Coin
Static Variables and Methods	Opening and Closing Accounts Counting Transactions Transfering Funds
Overall class design	Random Walks
GUI Layouts	Telephone Keypad

Changing People

The file *ChangingPeople.java* contains a program that illustrates parameter passing. The program uses *Person* objects defined in the file *Person.java.* Do the following:

1. Trace the execution of the program using diagrams similar to those in Figure 6.5 of the text (which is a trace of the program in Listings 6.15 – 6.17). Also show what is printed by the program.
2. Compile and run the program to see if your trace was correct.
3. Modify the *changePeople* method so that it does what the documentation says it does, that is, the two Person objects passed in as actual parameters are actually changed.

```
// ********************************************************************
//   ChangingPeople.java
//
//   Demonstrates parameter passing -- contains a method that should
//   change to Person objects.
// ********************************************************************

public class ChangingPeople
{
    // -----------------------------------------------------------
    //  Sets up two person objects, one integer, and one String
    //  object.  These are sent to a method that should make
    //  some changes.
    // -----------------------------------------------------------
    public static void main (String[] args)
    {
      Person person1 = new Person ("Sally", 13);
      Person person2 = new Person ("Sam", 15);

      int age = 21;
      String name = "Jill";

      System.out.println ("\nParameter Passing... Original values...");
      System.out.println ("person1: " + person1);
      System.out.println ("person2: " + person2);
      System.out.println ("age: " + age + "\tname: " + name + "\n");

      changePeople (person1, person2, age, name);

      System.out.println ("\nValues after calling changePeople...");
      System.out.println ("person1: " + person1);
      System.out.println ("person2: " + person2);
      System.out.println ("age: " + age + "\tname: " + name + "\n");
    }

    // -----------------------------------------------------------------
    //  Change the first actual parameter to "Jack - Age 101" and change
    //  the second actual parameter to be a person with the age and
    //  name given in the third and fourth parameters.
    // -----------------------------------------------------------------
    public static void changePeople (Person p1, Person p2, int age, String name)
    {
      System.out.println ("\nInside changePeople... Original parameters...");
      System.out.println ("person1: " + p1);
      System.out.println ("person2: " + p2);
      System.out.println ("age: " + age + "\tname: " + name + "\n");
```

```
      // Make changes
      Person p3 = new Person (name, age);
      p2 = p3;
      name = "Jack";
      age = 101;
      p1.changeName (name);
      p1.changeAge (age);

      // Print changes
      System.out.println ("\nInside changePeople... Changed values...");
      System.out.println ("person1: " + p1);
      System.out.println ("person2: " + p2);
      System.out.println ("age: " + age + "\tname: " + name + "\n");
    }
}


// ****************************************************************
//   Person.java
//
//   A simple class representing a person.
// ****************************************************************
public class Person
{
    private String name;
    private int age;

    // -----------------------------------------------------------
    //  Sets up a Person object with the given name and age.
    // -----------------------------------------------------------
    public Person (String name, int age)
    {
      this.name = name;
      this.age = age;
    }

    // -----------------------------------------------------------
    //   Changes the name of the Person to the parameter newName.
    // -----------------------------------------------------------
    public void changeName(String newName)
    {
      name = newName;
    }

    // -----------------------------------------------------------
    //   Changes the age of the Person to the parameter newAge.
    // -----------------------------------------------------------
    public void changeAge (int newAge)
    {
      age = newAge;
    }

    // -----------------------------------------------------------
    //   Returns the person's name and age as a string.
    // -----------------------------------------------------------
    public String toString()
    {
      return name + " - Age " + age;
    }
}
```

Using the *Comparable* Interface

1. Write a class Compare3 that provides a static method *largest*. Method *largest* should take three *Comparable* parameters and return the largest of the three (so its return type will also be *Comparable*). Recall that method *compareTo* is part of the *Comparable* interface, so *largest* can use the *compareTo* method of its parameters to compare them.

2. Write a class Comparisons whose main method tests your *largest* method above.
   - First prompt the user for and read in three strings, use your *largest* method to find the largest of the three strings, and print it out. (It's easiest to put the call to *largest* directly in the call to println.) Note that since *largest* is a static method, you will call it through its class name, e.g., *Compare3.largest(val1, val2, val3)*.
   - Add code to also prompt the user for three integers and try to use your *largest* method to find the largest of the three integers. Does this work? If it does, it's thanks to *autoboxing*, which is Java 1.5's automatic conversion of ints to Integers. You may have to use the `-source 1.5` compiler option for this to work.

A Modified MiniQuiz Class

Files *Question.java*, *Complexity.java*, and *MiniQuiz.java* contain the classes in Listings 6.8-6.10of the text. These classes demonstrate the use of the Complexity interface; class Question implements the interface, and class MiniQuiz creates two Question objects and uses them to give the user a short quiz.

Save these three files to your directory and study the code in MiniQuiz.java. Notice that after the Question objects are created, almost exactly the same code appears twice, once to ask and grade the first question, and again to ask and grade the second question. Another approach is to write a method *askQuestion* that takes a Question object and does all the work of asking the user the question, getting the user's response, and determining whether the response is correct. You could then simply call this method twice, once for *q1* and once for *q2*. Modify the MiniQuiz class so that it has such an *askQuestion* method, and replace the code in *main* that asks and grades the questions with two calls to *askQuestion*. Some things to keep in mind:

- The definition of *askQuestion* should be inside the MiniQuiz class but after the *main* method.
- Since *main* is a static method, *askQuestion* must be static too. (A static method cannot call an instance method of the same class.) Also, *askQuestion* is for use only by this class, so it should be declared private. So the header for *askQuestion* should look like this:

```
private static void askQuestion(Question question)
```

- String *possible*, which is currently declared in *main*, will need to be defined in *askQuestion* instead.
- The Scanner object *scan* needs to be a static variable and moved outside of main (so it is available to askQuestion).
- You do not need to make any changes to Question.java or Complexity.java.

```
//********************************************************************
//  Question.java       Author: Lewis/Loftus
//
//  Represents a question (and its answer).
//********************************************************************

public class Question implements Complexity
{
   private String question, answer;
   private int complexityLevel;

   //-----------------------------------------------------------------
   //  Sets up the question with a default complexity.
   //-----------------------------------------------------------------
   public Question (String query, String result)
   {
      question = query;
      answer = result;
      complexityLevel = 1;
   }

   //-----------------------------------------------------------------
   //  Sets the complexity level for this question.
   //-----------------------------------------------------------------
   public void setComplexity (int level)
   {
      complexityLevel = level;
   }

   //-----------------------------------------------------------------
   //  Returns the complexity level for this question.
   //-----------------------------------------------------------------
   public int getComplexity()
```

```
   {
      return complexityLevel;
   }

   //-----------------------------------------------------------------
   //  Returns the question.
   //-----------------------------------------------------------------
   public String getQuestion()
   {
      return question;
   }

   //-----------------------------------------------------------------
   //  Returns the answer to this question.
   //-----------------------------------------------------------------
   public String getAnswer()
   {
      return answer;
   }

   //-----------------------------------------------------------------
   //  Returns true if the candidate answer matches the answer.
   //-----------------------------------------------------------------
   public boolean answerCorrect (String candidateAnswer)
   {
      return answer.equals(candidateAnswer);
   }

   //-----------------------------------------------------------------
   //  Returns this question (and its answer) as a string.
   //-----------------------------------------------------------------
   public String toString()
   {
      return question + "\n" + answer;
   }
}


//*****************************************************************
//  Complexity.java       Author: Lewis/Loftus
//
//  Represents the interface for an object that can be assigned an
//  explicit complexity.
//*****************************************************************

public interface Complexity
{
   public void setComplexity (int complexity);
   public int getComplexity();
}
```

```
//********************************************************************
//  MiniQuiz.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a class that implements an interface.
//********************************************************************

import java.util.Scanner;

public class MiniQuiz
{
   //-----------------------------------------------------------------
   //  Presents a short quiz.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Question q1, q2;
      String possible;

      Scanner scan = new Scanner(System.in);

      q1 = new Question ("What is the capital of Jamaica?",
                         "Kingston");
      q1.setComplexity (4);

      q2 = new Question ("Which is worse, ignorance or apathy?",
                         "I don't know and I don't care");
      q2.setComplexity (10);

      System.out.print (q1.getQuestion());
      System.out.println (" (Level: " + q1.getComplexity() + ")");
      possible = scan.nextLine();
      if (q1.answerCorrect(possible))
         System.out.println ("Correct");
      else
         System.out.println ("No, the answer is " + q1.getAnswer());

      System.out.println();
      System.out.print (q2.getQuestion());
      System.out.println (" (Level: " + q2.getComplexity() + ")");
      possible = scan.nextLine();
      if (q2.answerCorrect(possible))
         System.out.println ("Correct");
      else
         System.out.println ("No, the answer is " + q2.getAnswer());
   }
}
```

A Flexible Account Class

File *Account.java* contains a definition for a simple bank account class with methods to withdraw, deposit, get the balance and account number, and return a String representation. Note that the constructor for this class creates a random account number. Save this class to your directory and study it to see how it works. Then modify it as follows:

1. Overload the constructor as follows:

    - public Account (double initBal, String owner, long number) – initializes the balance, owner, and account number as specified
    - public Account (double initBal, String owner) – initializes the balance and owner as specified; randomly generates the account number.
    - public Account (String owner) – initializes the owner as specified; sets the initial balance to 0 and randomly generates the account number.

2. Overload the *withdraw* method with one that also takes a fee and deducts that fee from the account.

File *TestAccount.java* contains a simple program that exercises these methods. Save it to your directory, study it to see what it does, and use it to test your modified Account class.

```
//*******************************************************
// Account.java
//
// A bank account class with methods to deposit to, withdraw from,
// change the name on, and get a String representation
// of the account.
//*******************************************************

public class Account
{
  private double balance;
  private String name;
  private long acctNum;

  //-----------------------------------------------
  //Constructor -- initializes balance, owner, and account number
  //-----------------------------------------------
  public Account(double initBal, String owner, long number)
  {
    balance = initBal;
    name = owner;
    acctNum = number;
  }

  //-----------------------------------------------
  // Checks to see if balance is sufficient for withdrawal.
  // If so, decrements balance by amount; if not, prints message.
  //-----------------------------------------------
  public void withdraw(double amount)
  {
    if (balance >= amount)
       balance -= amount;
    else
       System.out.println("Insufficient funds");
  }

  //-----------------------------------------------
  // Adds deposit amount to balance.
  //-----------------------------------------------
```

```
  public void deposit(double amount)
  {
    balance += amount;
  }

  //-----------------------------------------------
  // Returns balance.
  //-----------------------------------------------
  public double getBalance()
  {
    return balance;
  }


  //-----------------------------------------------
  // Returns a string containing the name, account number, and balance.
  //-----------------------------------------------
  public String toString()
  {
      return "Name:" + name +
            "\nAccount Number: " + acctNum +
            "\nBalance: " + balance;
  }
}
```

```
//********************************************************
// TestAccount.java
//
// A simple driver to test the overloaded methods of
// the Account class.
//********************************************************

import java.util.Scanner;

public class TestAccount
{
    public static void main(String[] args)
    {
        String name;
        double balance;
        long acctNum;
        Account acct;

        Scanner scan = new Scanner(System.in);

        System.out.println("Enter account holder's first name");
        name = scan.next();
        acct = new Account(name);
        System.out.println("Account for " + name + ":");
        System.out.println(acct);

        System.out.println("\nEnter initial balance");
        balance = scan.nextDouble();
        acct = new Account(balance,name);
        System.out.println("Account for " + name + ":");
        System.out.println(acct);
```

```
        System.out.println("\nEnter account number");
        acctNum = scan.nextLong();
        acct = new Account(balance,name,acctNum);
        System.out.println("Account for " + name + ":");
        System.out.println(acct);

        System.out.print("\nDepositing 100 into account, balance is now ");
        acct.deposit(100);
        System.out.println(acct.getBalance());
        System.out.print("\nWithdrawing $25, balance is now ");
        acct.withdraw(25);
        System.out.println(acct.getBalance());
        System.out.print("\nWithdrawing $25 with $2 fee, balance is now ");
        acct.withdraw(25,2);
        System.out.println(acct.getBalance());

        System.out.println("\nBye!");
    }
}
```

Modifying the Coin Class

1. Create a new class named *BiasedCoin* that models a biased coin (heads and tails are not equally likely outcomes of a flip). To do this modify the coin class from the Listing 5.4 of text (in the file *Coin.java*) as follows:
   - Add a private data member *bias* of type double. This data member will be a number between 0 and 1 (inclusive) that represents the probability the coin will be HEADS when flipped. So, if bias is 0.5, the coin is an ordinary fair coin. If bias is 0.6, the coin has probability 0.6 of coming up heads (on average, it comes up heads 60% of the time).
   - Modify the default constructor by assigning the value 0.5 to bias *before* the call to flip. This will make the default coin a fair one.
   - Modify *flip* so that it generates a random number then assigns *face* a value of HEADS if the number is less than the bias; otherwise it assigns a value of TAILS.
   - Add a second constructor with a single double parameter—that parameter will be the bias. If the parameter is valid (a number between 0 and 1 inclusive) the constructor should assign the *bias* data member the value of the parameter; otherwise it should assign *bias* a value of 0.5. Call flip (as the other constructor does) to initialize the value of face.

2. Compile your class to make sure you have no syntax errors.

3. Write a program that uses three BiasedCoin objects. Instantiate one as a fair coin using the constructor with no parameter. Read in the biases for the other two coins and instantiate those coins using the constructor with the bias as a parameter. Your program should then have a loop that flips each coin 100 times and counts the number of times each is heads. After the loop print the number of heads for each coin. Run the program several times testing out different biases.

Opening and Closing Accounts

File *Account.java* (see previous exercise) contains a definition for a simple bank account class with methods to withdraw, deposit, get the balance and account number, and return a String representation. Note that the constructor for this class creates a random account number. Save this class to your directory and study it to see how it works. Then write the following additional code:

1. Suppose the bank wants to keep track of how many accounts exist.

    a. Declare a private static integer variable numAccounts to hold this value. Like all instance and static variables, it will be initialized (to 0, since it's an int) automatically.
    b. Add code to the constructor to increment this variable every time an account is created.
    c. Add a static method *getNumAccounts* that returns the total number of accounts. Think about why this method should be static – its information is not related to any particular account.
    d. File *TestAccounts1.java* contains a simple program that creates the specified number of bank accounts then uses the getNumAccounts method to find how many accounts were created. Save it to your directory, then use it to test your modified Account class.

2. Add a method void close() to your Account class. This method should close the current account by appending "CLOSED" to the account name and setting the balance to 0. (The account number should remain unchanged.) Also decrement the total number of accounts.

3. Add a static method *Account consolidate(Account acct1, Account acct2)* to your Account class that creates a new account whose balance is the sum of the balances in acct1 and acct2 and closes acct1 and acct2. The new account should be returned. Two important rules of consolidation:

    - Only accounts with the same name can be consolidated. The new account gets the name on the old accounts but a new account number.
    - Two accounts with the same number cannot be consolidated. Otherwise this would be an easy way to double your money!

    Check these conditions before creating the new account. If either condition fails, do not create the new account or close the old ones; print a useful message and return null.

4. Write a test program that prompts for and reads in three names and creates an account with an initial balance of $100 for each. Print the three accounts, then close the first account and try to consolidate the second and third into a new account. Now print the accounts again, including the consolidated one if it was created.

```
//********************************************************
// TestAccounts1
// A simple program to test the numAccts method of the
// Account class.
//********************************************************
import java.util.Scanner;

public class TestAccounts1
{
    public static void main(String[] args)
    {
        Account testAcct;

        Scanner scan = new Scanner(System.in);
```

```
        System.out.println("How many accounts would you like to create?");
        int num = scan.nextInt();

        for (int i=1; i<=num; i++)
            {
                testAcct = new Account(100, "Name" + i);
                System.out.println("\nCreated account " + testAcct);
                System.out.println("Now there are " + Account.numAccounts() +
                                   " accounts");
            }
    }
}
```

Counting Transactions

File *Account.java* (see **A Flexible Account Class** exercise) contains a definition for a simple bank account class with methods to withdraw, deposit, get the balance and account number, and return a String representation. Note that the constructor for this class creates a random account number. Save this class to your directory and study it to see how it works. Now modify it to keep track of the total number of deposits and withdrawals (separately) for each day, and the total amount deposited and withdrawn. Write code to do this as follows:

1. Add four private static variables to the Account class, one to keep track of each value above (number and total amount of deposits, number and total of withdrawals). Note that since these variables are static, all of the Account objects share them. This is in contrast to the instance variables that hold the balance, name, and account number; each Account has its own copy of these. Recall that numeric static and instance variables are initialized to 0 by default.

2. Add public methods to return the values of each of the variables you just added, e.g., *public static int getNumDeposits()*.

3. Modify the *withdraw* and *deposit* methods to update the appropriate static variables at each withdrawal and deposit

4. File *ProcessTransactions.java* contains a program that creates and initializes two Account objects and enters a loop that allows the user to enter transactions for either account until asking to quit. Modify this program as follows:
   - After the loop, print the total number of deposits and withdrawals and the total amount of each. You will need to use the Account methods that you wrote above. Test your program.
   - Imagine that this loop contains the transactions for a single day. Embed it in a loop that allows the transactions to be recorded and counted for many days. At the beginning of each day print the summary for each account, then have the user enter the transactions for the day. When all of the transactions have been entered, print the total numbers and amounts (as above), then reset these values to 0 and repeat for the next day. Note that you will need to add methods to reset the variables holding the numbers and amounts of withdrawals and deposits to the Account class. Think: should these be static or instance methods?

```
//*******************************************************
// ProcessTransactions.java
//
// A class to process deposits and withdrawals for two bank
// accounts for a single day.
//*******************************************************
import java.util.Scanner;

public class ProcessTransactions
{
    public static void main(String[] args){

        Account acct1, acct2;           //two test accounts
        String keepGoing = "y";         //more transactions?
        String action;                  //deposit or withdraw
        double amount;                  //how much to deposit or withdraw
        long acctNumber;                //which account to access

        Scanner scan = new Scanner(System.in);

        //Create two accounts
        acct1 = new Account(1000, "Sue", 123);
        acct2 = new Account(1000, "Joe", 456);

        System.out.println("The following accounts are available:\n");
        acct1.printSummary();

        System.out.println();
        acct2.printSummary();
```

```
        while (keepGoing.equals("y") || keepGoing.equals("y"))
            {
                //get account number, what to do, and amount
                System.out.print("\nEnter the number of the account you would like
to access: ");
                acctNumber = scan.nextLong();
                System.out.print("Would you like to make a deposit (D) or withdrawal
(W)? ");
                action = scan.next();
                System.out.print("Enter the amount: ");
                amount = scan.nextDouble();

                if (amount > 0)
                    if (acctNumber == acct1.getAcctNumber())
                        if (action.equals("w") || action.equals("W"))
                            acct1.withdraw(amount);
                        else if (action.equals("d") || action.equals("D"))
                            acct1.deposit(amount);
                        else
                            System.out.println("Sorry, invalid action.");
                    else if (acctNumber == acct2.getAcctNumber())
                        if (action.equals("w") || action.equals("W"))
                            acct1.withdraw(amount);
                        else if (action.equals("d") || action.equals("D"))
                            acct1.deposit(amount);
                        else
                            System.out.println("Sorry, invalid action.");
                    else
                        System.out.println("Sorry, invalid account number.");
                else
                        System.out.println("Sorry, amount must be > 0.");


                System.out.print("\nMore transactions? (y/n)");
                keepGoing = scan.next();
            }


        //Print number of deposits
        //Print number of withdrawals
        //Print total amount of deposits
        //Print total amount of withdrawals


    }
}
```

Transfering Funds

File *Account.java* (see **A Flexible Account Class** exercise) contains a definition for a simple bank account class with methods to withdraw, deposit, get the balance and account number, and print a summary. Save it to your directory and study it to see how it works. Then write the following additional code:

1. Add a method *public void transfer(Account acct, double amount)* to the Account class that allows the user to transfer funds from one bank account to another. If *acct1* and *acct2* are Account objects, then the call *acct1.transfer(acct2,957.80)* should transfer $957.80 from acct1 to acct2. Be sure to clearly document which way the transfer goes!

2. Write a class TransferTest with a main method that creates two bank account objects and enters a loop that does the following:
   - Asks if the user would like to transfer from account1 to account2, transfer from account2 to account1, or quit.
   - If a transfer is chosen, asks the amount of the transfer, carries out the operation, and prints the new balance for each account.
   - Repeats until the user asks to quit, then prints a summary for each account.

3. Add a static method to the Account class that lets the user transfer money between two accounts without going through either account. You can (and should) call the method transfer just like the other one – you are overloading this method. Your new method should take two Account objects and an amount and transfer the amount from the first account to the second account. The signature will look like this:

   ```
   public static void transfer(Account acct1, Account acct2, double amount)
   ```

   Modify your TransferTest class to use the static transfer instead of the instance version.

Random Walks

In this lab you will develop a class that models a random walk and write two client programs that use the class. A random walk is basically a sequence of steps in some enclosed space where the direction of each step is random. The walk terminates either when a maximum number of steps has been taken or a step goes outside of the boundary of the space. Random walks are used to model physical phenomena such as the motion of molecules and economic phenomena such as stock prices.

We will assume that the random walk takes place on a square grid with the point (0,0) at the center. The boundary of the square will be a single integer that represents the maximum x and y coordinate for the current position on the square (so for a boundary value of 10, both the x and y coordinates can vary from -10 to 10, inclusive). Each step will be one unit up, one unit down, one unit to the left, or one unit to the right. (No diagonal movement.)

The *RandomWalk* class will have the following instance data (all type int):

- the x coordinate of the current position
- the y coordinate of the current position
- the maximum number of steps in the walk
- the number of steps taken so far in the walk
- the boundary of the square (a positive integer -- the x and y coordinates of the position can vary between plus and minus this value)

Create a new file *RandomWalk.java*. You'll define the RandomWalk class incrementally testing each part as you go.

1. First declare the instance data (as described above) and add the following two constructors and toString method.
    - *RandomWalk (int max, int edge)* - Initializes the RandomWalk object. The maximum number of steps and the boundary are given by the parameters. The x and y coordinates and the number of steps taken should be set to 0.
    - *RandomWalk (int max, int edge, int startX, int startY)* -- Initializes the maximum number of steps, the boundary, and the starting position to those given by the parameters.
    - *String toString()* - returns a String containing the number of steps taken so far and the current position -- The string should look something like: Steps: 12; Position: (-3,5)

2. Compile what you have so far then open the file *TestWalk.java*. This file will be used to test your RandomWalk methods. So far it prompts the user to enter a boundary, a maximum number of steps, and the x and y coordinates of a position. Add the following:
    - Declare and instantiate two RandomWalk objects -- one with boundary 5, maximum steps 10, and centered at the origin (use the two parameter constructor) and the other with the values entered by the user.
    - Print out each object. Note that you won't get any information about the boundary or maximum number of steps (think about what your toString method does), but that's ok.

    Compile and run the program to make sure everything is correct so far.

3. Next add the following method to the RandomWalk class: *void takeStep()*. This method simulates taking a single step either up, down, left, or right. To "take a step" generate a random number with 4 values (say 0, 1, 2, 3) then use a switch statement to change the position (one random value will represent going right, one left, and so on). Your method should also increment the number of steps taken.

4. Add a for loop to TestWalk.java to have each of your RandomWalk objects take 5 steps. Print out each object after each step so you can see what is going on. Compile and run the program to make sure it is correct so far.

5. Now add to RandomWalk.java the following two methods. Each should be a single return statement that returns the value of a boolean expression.
    - *boolean moreSteps()* - returns true if the number of steps taken is less than the maximum number; returns false otherwise

- *boolean inBounds()* - returns true if the current position is on the square (include the boundary as part of the square); returns false otherwise.

6. Add to the RandomWalk class a method named *walk* that has no parameters and returns nothing. Its job is to simulate a complete random walk. That is, it should generate a sequence of steps as long the maximum number of steps has not been taken and it is still in bounds (inside the square). This should be a very simple loop (while or do... while) --- you will need to call the methods takeStep, moreSteps, and inBounds.

7. Add to TestWalk.java a statement to instantiate a RandomWalk object with a boundary of 10 and 200 as the maximum number of steps. (You may want to comment out most of the code currently in TestWalk -- especially the user input and the loop that takes five steps -- as the walk method will be easier to test on its own. The /* ... */ style of comment is useful for this. But don't delete that other code, as you'll need it later in the lab.) Then add a statement to have the object walk. Print the object after the walk. Compile and run the program. Run it more than once -- you should be able to tell by the value printed whether the object went out of bounds or whether it stopped because it reached the maximum number of steps.

8. Now write a client program in a file named *DrunkenWalk.java*. The program should simulate a drunk staggering randomly on some sort of platform (imagine a square dock in the middle of a lake). The goal of the program is to have the program simulate the walk many times (because of randomness each walk is different) and count the number of times the drunk falls off the platform (goes out of bounds). Your program should read in the boundary, the maximum number of steps, and the number of drunks to simulate. It should then have a loop (a for loop would be a good idea) that on each iteration instantiates a new RandomWalk object to represent a drunk, has the object walk, then determines whether or not the drunk fell off the platform (and updates a counter if it did). After the loop print out the number of times the drunk fell off. Compile and run your program. To see the "randomness" you should run it several times. Try input of 10 for the boundary and 200 for the number of steps first (sometimes the drunk falls off, sometimes not); try 10 for the boundary and 500 for the steps (you should see different behavior); try 50 for the boundary and 200 for the steps (again different behavior).

9. Now write a second client program in a file named *Collisions.java*. This program should simulate two particles moving in space. Its goal is to determine the number of times the two particles collide (occupy exactly the same position after the same number of steps -- the steps could be thought of as simulating time). We'll assume the particles are in a very large space so use a large number for the boundary (such as 2,000,000). Use 100,000 for the maximum number of steps. (Don't enter the commas.) Start one particle at (-3, 0) and the other at (3, 0). (You can hardcode these values into the program; no need to enter them.) Your program should contain a loop that has each particle take a step as long as the particles have not exceeded the maximum number of steps. The program then determines how often the particles have collided. Note that in order for your program to know whether or not the two different RandomWalk objects are in the same place it needs to be able to find out the position. Hence, you need to add the following two methods to the RandomWalk class.
   - *int getX()* - returns the x coordinate of the current position
   - *int getY()* - returns the y coordinate of the current position

   Compile and run your program to make sure it works. As before run it several times.

10. In your Collisions.java program the condition to determine if the points are at the same position is a bit cumbersome. This is something that would be best put in a separate method. Add a static method to Collisions.java (after the main method) with signature

    ```
    public static boolean samePosition (RandomWalk p1, RandomWalk p2)
    ```

    The method should return true if p1 and p2 are at the same position and return false otherwise. Modify your main method so it calls samePosition rather than directly testing to see if the objects are at the same position. Test the program.

11. In using random walks to simulate behavior it is often of interest to know how far away from the origin the object gets as it moves.

a. Add an instance variable *maxDistance* (type int) to the RandomWalk class. This should be set to 0 in each constructor.
b. Now the takeStep method needs to update this maximum when a step is taken. We'll add a support method to the class to do this. Add a *private* method named *max* that takes two integer parameters (say *num1* and *num2*) and returns the largest of the two.
c. Add code to takeStep to update maxDistance. This can be done in a single statement using the max method -- the new value of maxDistance should be the maximum of 1) the old value of maxDistance, and 2) the current distance to the origin. Note that if the current point is (-3, 15) the distance to the origin is 15; if the current point is (-10, 7) the distance to the origin is 10. Remember that Math.abs returns the absolute value of a number.
d. Finally add an accessor method to return that distance so a client program can access it:

   public int getMaxDistance()

e. Test the maximum by adding statements in TestWalk.java to get and print the maximum distance for each of the objects after the loop that had them take and print out 5 steps (this way you can see if the maximum is correct – each step is printed).

```
// ********************************************************
//  TestWalk.java
//
//  Program to test methods in the RandomWalk class.
// ********************************************************

import java.util.Scanner;

public class TestWalk
{
    public static void main (String[] args)
    {
        int maxSteps;   // maximum number of steps in a walk
        int maxCoord;   // the maximum x and y coordinate
        int x, y;       // starting x and y coordinates for a walk

        Scanner scan = new Scanner(System.in);

        System.out.println ("\nRandom Walk Test Program");
        System.out.println ();

        System.out.print ("Enter the boundary for the square: ");
        maxCoord = scan.nextInt();

        System.out.print ("Enter the maximum number of steps: ");
        maxSteps = scan.nextInt();

        System.out.print ("Enter the starting x and y coordinates with " +
                          "a space between: ");
        x = scan.nextInt();
        y = scan.nextInt();

    }
}
```

Telephone Keypad

Files *Telephone.java* and *TelephonePanel.java* contain the skeleton for a program to lay out a GUI that looks like telephone keypad with a title that says "Your Telephone!!". Save these files to your directory. Telephone.java is complete, but TelephonePanel.java is not.

1. Using the comments as a guide, add code to TelephonePanel.java to create the GUI. Some things to consider:
    a. TelephonePanel (the current object, which is a JPanel) should get a BorderLayout to make it easy to separate the title from the keypad. The title will go in the north area and the keypad will go in the center area. The other areas will be unused.
    b. You can create a JLabel containing the title and add it directly to the north section of the TelephonePanel. However, to put the keypad in the center you will first need to create a new JPanel and add the keys (each a button) to it, then add it to the center of the TelephonePanel. This new panel should have a 4x3 GridLayout.
    c. Your keypad should hold buttons containing 1 2 3, 4 5 6, 7 8 9, * 0 # in the four rows respectively. So you'll create a total of 12 buttons.

2. Compile and run Telephone.java. You should get a small keypad and title. Grow the window (just drag the corner) and see how the GUI changes – everything grows proportionately.

3. Note that the title is not centered, but it would look nicer if it were. One way to do this is to create a new JPanel , add the title label to it, then add the new JPanel to the north area of the TelephonePanel (instead of adding the label directly). This works because the default layout for a JPanel is a centered FlowLayout, and the JPanel itself will expand to fill the whole north area. Modify your program in this way so that the title is centered.

```
//*******************************************************
// Telephone.java
//
// Uses the TelephonePanel class to create a (functionless) GUI
// like a telephone keypad with a title.
// Illustrates use of BorderLayout and GridLayout.
//*******************************************************
import javax.swing.*;
public class Telephone
{
    public static void main(String[] args)
    {
        JFrame frame = new JFrame("Telephone");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(new TelephonePanel());
        frame.pack();
        frame.setVisible(true);
    }
}
```

```
//*******************************************************
// TelephonePanel.java
//
// Lays out a (functionless) GUI like a telephone keypad with a title.
// Illustrates use of BorderLayout and GridLayout.
//*******************************************************
import java.awt.*;
import javax.swing.*;

public class TelephonePanel extends JPanel
{
    public TelephonePanel()
    {
        //set BorderLayout for this panel

        //create a JLabel with "Your Telephone" title

        //add title label to north of this panel

        //create panel to hold keypad and give it a 4x3 GridLayout

        //add buttons representing keys to key panel

        //add key panel to center of this panel
    }
}
```