Chapter 11: Recursion
Lab Exercises

Topics	Lab Exercises
Basic Recursion	Computing Powers
	Counting and Summing Digits
	Base Conversion
	Efficient Computation of Fibonacci Numbers
Recursion on Strings	Palindromes
	Printing a String Backwards
Recursion on Arrays	Recursive Linear Search
	Recursive Binary Search
	A List of Employees
Fractals	Sierpinski Triangles
	Modifying the Koch Snowflake

Computing Powers

Computing a positive integer power of a number is easily seen as a recursive process. Consider a^n:

- If n = 0, a^n is 1 (by definition)
- If n > 0, a^n is a * a^{n-1}

File *Power.java* contains a main program that reads in integers *base* and *exp* and calls method *power* to compute $base^{exp}$. Fill in the code for *power* to make it a recursive method to do the power computation. The comments provide guidance.

```
// ****************************************************************
//   Power.java
//
//   Reads in two integers and uses a recursive power method
//   to compute the first raised to the second power.
// ****************************************************************

import java.util.Scanner;

public class Power
{
    public static void main(String[] args)
    {
      int base, exp;
      int answer;

      Scanner scan = new Scanner(System.in);

      System.out.print("Welcome to the power program! ");
      System.out.println("Please use integers only.");

      //get base
      System.out.print("Enter the base you would like raised to a power: ");
      base = scan.nextInt();

      //get exponent
      System.out.print("Enter the power you would like it raised to: ");
      exp = scan.nextInt();

      answer = power(base,exp);
      System.out.println(base + " raised to the " + exp + " is " + answer);
    }

    // -----------------------------------------------
    //   Computes and returns base^exp
    // -----------------------------------------------
    public static int power(int base, int exp)
    {
      int pow;

      //if the exponent is 0, set pow to 1

      //otherwise set pow to base*base^(exp-1)

      //return pow

    }
}
```

Counting and Summing Digits

The problem of counting the digits in a positive integer or summing those digits can be solved recursively. For example, to count the number of digits think as follows:

- If the integer is less than 10 there is only one digit (the base case).
- Otherwise, the number of digits is 1 (for the units digit) plus the number of digits in the rest of the integer (what's left after the units digit is taken off). For example, the number of digits in 3278 is 1 + the number of digits in 327.

The following is the recursive algorithm implemented in Java.

```
public int numDigits (int num)
{
    if (num < 10)
      return (1);   // a number < 10  has only one digit
    else
      return (1 + numDigits (num / 10));
}
```

Note that in the recursive step, the value returned is 1 (counts the units digit) + the result of the call to determine the number of digits in *num / 10*. Recall that *num/10* is the quotient when *num* is divided by 10 so it would be all the digits except the units digit.

The file *DigitPlay.java* contains the recursive method *numDigits* (note that the method is static—it must be since it is called by the static method main). Copy this file to your directory, compile it, and run it several times to see how it works. Modify the program as follows:

1. Add a static method named *sumDigits* that finds the *sum* of the digits in a positive integer. Also add code to main to test your method. The algorithm for *sumDigits* is very similar to *numDigits*; you only have to change two lines!

2. Most identification numbers, such as the ISBN number on books or the Universal Product Code (UPC) on grocery products or the identification number on a traveller's check, have at least one digit in the number that is a *check digit*. The check digit is used to detect errors in the number. The simplest check digit scheme is to add one digit to the identification number so that the sum of all the digits, including the check digit, is evenly divisible by some particular integer. For example, American Express Traveller's checks add a check digit so that the sum of the digits in the id number is evenly divisible by 9. United Parcel Service adds a check digit to its pick up numbers so that a weighted sum of the digits (some of the digits in the number are multiplied by numbers other than 1) is divisible by 7. Modify the main method that tests your sumDigits method to do the following: input an identification number (a positive integer), then determine if the sum of the digits in the identification number is divisible by 7 (use your sumDigits method but don't change it—the only changes should be in main). If the sum is not divisible by 7 print a message indicating the id number is in error; otherwise print an ok message. (FYI: If the sum is divisible by 7, the identification number could still be incorrect. For example, two digits could be transposed.) Test your program on the following input:
   - 3429072 --- error
   - 1800237 --- ok
   - 88231256 --- ok
   - 3180012 --- error

```
// ********************************************************************
//   DigitPlay.java
//
//   Finds the number of digits in a positive integer.
// ********************************************************************

import java.util.Scanner;

public class DigitPlay
{

    public static void main (String[] args)
    {
      int num;    //a number

      Scanner scan = new Scanner(System.in);

      System.out.println ();
      System.out.print ("Please enter a positive integer: ");
      num = scan.nextInt ();

      if (num <= 0)
          System.out.println ( num + " isn't positive -- start over!!");
      else
          {
            // Call numDigits to find the number of digits in the number
            // Print the number returned from numDigits
            System.out.println ("\nThe number " + num + " contains " +
                            + numDigits(num) + " digits.");
            System.out.println ();
          }
    }

    // -----------------------------------------------------------
    //  Recursively counts the digits in a positive integer
    // -----------------------------------------------------------
    public static int numDigits(int num)
    {
      if (num < 10)
          return (1);
      else
          return (1 + numDigits(num/10));
    }
}
```

Base Conversion

One algorithm for converting a base 10 number to base b involves repeated division by the base b. Initially one divides the number by b. The remainder from this division is the units digit (the rightmost digit) in the base b representation of the number (it is the part of the number that contains no powers of b). The quotient is then divided by b on the next iteration. The remainder from this division gives the next base b digit from the right. The quotient from this division is used in the next iteration. The algorithm stops when the quotient is 0. Note that at each iteration the remainder from the division is the next base b digit from the right—that is, this algorithm finds the digits for the base b number in reverse order.

Here is an example for converting 30 to base 4:

```
              quotient       remainder
              --------       ---------
 30/4 =           7              2
 7/4  =           1              3
 1/4  =           0              1
```

The answer is read bottom to top in the remainder column, so 30 (base 10) = 132 (base 4).

Think about how this is recursive in nature: If you want to convert x (30 in our example) to base b (4 in our example), the rightmost digit is the remainder x % b. To get the rest of the digits, you perform the same process on what is left; that is, you convert the quotient x / b to base b. If x / b is 0, there is no rest; x is a single base b digit and that digit is x % b (which also is just x).

The file *BaseConversion.java* contains the shell of a method *convert* to do the base conversion and a main method to test the conversion. The convert method returns a string representing the base b number, hence for example in the base case when the remainder is what is to be returned it must be converted to a String object. This is done by concatenating the remainder with a null string. The outline of the convert method is as follows:

```
public static String convert (int num, int b)
{
   int quotient;  // the quotient when num is divided by base b
   int remainder; // the remainder when num is divided by base b

   quotient = _____________________________;

   remainder = _____________________________;

   if ( ________________________________________ )  //fill in base case
   {
        return ("" + ____________________________________ );
   }
   else
   {
     // Recursive step: the number is the base b representation of
       // the quotient concatenated with the remainder

  return ( ____________________________________________________ );

   }
}
```

Fill in the blanks above (for now don't worry about bases greater than 10), then in BaseConversion.java complete the method and main. Main currently asks the user for the number and the base and reads these in. Add a statement to print the string returned by convert (appropriately labeled).

Test your function on the following input:

- Number: 89 Base: 2 ---> should print 1011001

- Number: 347 Base: 5 ---> should print 2342
- Number: 3289 Base: 8 ---> should print 6331

Improving the program: Currently the program doesn't print the correct digits for bases greater than 10. Add code to your convert method so the digits are correct for bases up to and including 16.

```
// ********************************************************************
//   BaseConversion.java
//
//   Recursively converts an integer from base 10 to another base
// ********************************************************************

import java.util.Scanner;

public class BaseConversion
{
    public static void main (String[] args)
    {
      int base10Num;
      int base;

      Scanner scan = new Scanner(System.in);

      System.out.println ();
      System.out.println ("Base Conversion Program");
      System.out.print ("Enter an integer: ");
      base10Num = scan.nextInt();
      System.out.print ("Enter the base: ");
      base = scan.nextInt();

      // Call convert and print the answer

    }
    // -----------------------------------------------
    //   Converts a base 10 number to another base.
    // -----------------------------------------------
    public static String convert (int num, int b)
    {
      int quotient;  // the quotient when num is divided by base b
      int remainder; // the remainder when num is divided by base b
    }

}
```

Efficient Computation of Fibonacci Numbers

The *Fibonacci* sequence is a well-known mathematical sequence in which each term is the sum of the two previous terms. More specifically, if fib(n) is the nth term of the sequence, then the sequence can be defined as follows:

```
fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2)  n>1
```

1. Because the Fibonacci sequence is defined recursively, it is natural to write a recursive method to determine the nth number in the sequence. File *Fib.java* contains the skeleton for a class containing a method to compute Fibonacci numbers. Save this file to your directory. Following the specification above, fill in the code for method *fib1* so that it recursively computes and returns the nth number in the sequence.

2. File *TestFib.java* contains a simple driver that asks the user for an integer and uses the *fib1* method to compute that element in the Fibonacci sequence. Save this file to your directory and use it to test your *fib1* method. First try small integers, then larger ones. You'll notice that the number doesn't have to get very big before the calculation takes a very long time. The problem is that the *fib1* method is making lots and lots of recursive calls. To see this, add a print statement at the beginning of your *fib1* method that indicates what call is being computed, e.g., "In fib1(3)" if the parameter is 3. Now run TestFib again and enter 5—you should get a number of messages from your print statement. Examine these messages and figure out the sequence of calls that generated them. (This is easiest if you first draw the call tree on paper.) . Since fib(5) is fib(4) + fib(3),you should not be surprised to find calls to fib(4) and fib(3) in the printout. But why are there two calls to fib(3)? Because both fib(4) and fib(5) need fib(3), so they both compute it—very inefficient. Run the program again with a slightly larger number and again note the repetition in the calls.

3. The fundamental source of the inefficiency is not the fact that recursive calls are being made, but that values are being recomputed. One way around this is to compute the values from the beginning of the sequence instead of from the end, saving them in an array as you go. Although this could be done recursively, it is more natural to do it iteratively. Proceed as follows:
   a. Add a method *fib2* to your Fib class. Like *fib1*, *fib2* should be static and should take an integer and return an integer.
   b. Inside *fib2*, create an array of integers the size of the value passed in.
   c. Initialize the first two elements of the array to 0 and 1, corresponding to the first two elements of the Fibonacci sequence. Then loop through the integers up to the value passed in, computing each element of the array as the sum of the two previous elements. When the array is full, its last element is the element requested. Return this value.
   d. Modify your TestFib class so that it calls *fib2* (first) and prints the result, then calls *fib1* and prints that result. You should get the same answers, but very different computation times.

```
// ****************************************************************
//   Fib.java
//
//   A utility class that provide methods to compute elements of the
//   Fibonacci sequence.
// ****************************************************************
public class Fib
{

    //-----------------------------------------------------------
    // Recursively computes fib(n)
    //-----------------------------------------------------------
    public static int fib1(int n)
    {
      //Fill in code -- this should look very much like the
      //mathematical specification
    }

}
```

```
// ********************************************************************
//   TestFib.java
//
//   A simple driver that uses the Fib class to compute the
//   nth element of the Fibonacci sequence.
// ********************************************************************

import java.util.Scanner;

public class TestFib
{
    public static void main(String[] args)
    {
      int n, fib;

      Scanner scan = new Scanner(System.in);

      System.out.print("Enter an integer: ");
      n = scan.nextInt();

      fib = Fib.fib1(n);
      System.out.println("Fib(" + n + ") is " + fib);
    }
}
```

Palindromes

A *palindrome* is a string that is the same forward and backward. In Chapter 5 you saw a program that uses a loop to determine whether a string is a palindrome. However, it is also easy to define a palindrome recursively as follows:

- A string containing fewer than 2 letters is always a palindrome.
- A string containing 2 or more letters is a palindrome if
  - its first and last letters are the same, and
  - the rest of the string (without the first and last letters) is also a palindrome.

Write a program that prompts for and reads in a string, then prints a message saying whether it is a palindrome. Your main method should read the string and call a recursive (static) method *palindrome* that takes a string and returns true if the string is a palindrome, false otherwise. Recall that for a string *s* in Java,

- *s.length()* returns the number of charaters in *s*
- *s.charAt(i)* returns the i^{th} character of *s*, 0-based
- *s.substring(i,j)* returns the substring that starts with the i^{th} character of *s* and ends with the $j–1^{st}$ character of *s* (not the j^{th}), both 0-based.

So if *s*="happy", *s.length*=5, *s.charAt(1)*=a, and *s.substring(2,4)* = "pp".

Printing a String Backwards

Printing a string backwards can be done iteratively or recursively. To do it recursively, think of the following specification:

If *s* contains any characters (i.e., is not the empty string)

- print the last character in *s*
- print *s′* backwards, where *s′* is *s* without its last character

File *Backwards.java* contains a program that prompts the user for a string, then calls method *printBackwards* to print the string backwards. Save this file to your directory and fill in the code for *printBackwards* using the recursive strategy outlined above.

```
// ********************************************************************
//   Backwards.java
//
//   Uses a recursive method to print a string backwards.
// ********************************************************************
import java.util.Scanner;

public class Backwards
{

    //-----------------------------------------------------------------
    // Reads a string from the user and prints it backwards.
    //-----------------------------------------------------------------
    public static void main(String[] args)
    {
      String msg;
      Scanner scan = new Scanner(System.in);

      System.out.print("Enter a string: ");
      msg = scan.nextLine();

      System.out.print("\nThe string backwards: ");
      printBackwards(msg);
      System.out.println();
    }

    //-----------------------------------------------------------------
    // Takes a string and recursively prints it backwards.
    //-----------------------------------------------------------------
    public static void printBackwards(String s)
    {

      // Fill in code

    }
}
```

Recursive Linear Search

File *IntegerListS.java* contains a class IntegerListS that represents a list of integers (you may have used a version of this in an earlier lab); *IntegerListSTest.java* contains a simple menu-driven test program that lets the user create, sort, and print a list and search for an element using a linear search.

Many list processing tasks, including searching, can be done recursively. The base case typically involves doing something with a limited number of elements in the list (say the first element), then the recursive step involves doing the task on the rest of the list. Think about how linear search can be viewed recursively; if you are looking for an item in a list starting at index i:

- If i exceeds the last index in the list, the item is not found (return -1).
- If the item is at list[i], return i.
- If the is not at list[i], do a linear search starting at index i+1.

Fill in the body of the method *linearSearchR* in the IntegerList class. The method should do a recursive linear search of a list starting with a given index (parameter *lo*). Note that the IntegerList class contains another method linearSearchRec that does nothing but call your method (linearSearchR). This is done because the recursive method (*linearSearchR*) needs more information (the index to start at) than you want to pass to the top-level search routine (*linearSearchRec*), which just needs the thing to look for.

Now change IntegerListTest.java so that it calls linearSearchRec instead of linearSearch when the user asks for a linear search. Thoroughly test the program.

```
// ****************************************************************
//   IntegerListS.java
//
//   Defines an IntegerListS class with methods to create, fill,
//   sort, and search in a list of integers. (Version S -
//   for use in the linear search exercise.)
//
// ****************************************************************


public class IntegerListS
{
    int[] list; //values in the list

    // ------------------------------------
    //   Creates a list of the given size
    // ------------------------------------
    public IntegerListS (int size)
    {
      list = new int[size];
    }

    // ------------------------------------------------------------
    //   Fills the array with integers between 1 and 100, inclusive
    // ------------------------------------------------------------
    public void randomize()
    {
      for (int i=0; i< list.length; i++)
          list[i] = (int)(Math.random() * 100) + 1;
    }

    // ----------------------------------------
    //   Prints array elements with indices
    // ----------------------------------------
    public void print()
```

```
    {
      for (int i=0; i<list.length; i++)
          System.out.println(i + ":\t" + list[i]);
    }

    // -----------------------------------------------------------------
    //   Returns the index of the first occurrence of target in the list.
    //   Returns -1 if target does not appear in the list.
    // -----------------------------------------------------------------
    public int linearSearch(int target)
    {
      int location = -1;
      for (int i=0; i<list.length && location == -1; i++)
          if (list[i] == target)
            location = i;
      return location;
    }

    // -----------------------------------------------------------------
    //   Returns the index of an occurrence of target in the list, -1
    //   if target does not appear in the list.
    // -----------------------------------------------------------------
    public int linearSearchRec(int target)
    {
      return linearSearchR (target, 0);
    }

    // -----------------------------------------------------------------
    //   Recursive implementation of the linear search - searches
    //   for target starting at index lo.
    // -----------------------------------------------------------------
    private int linearSearchR (int target, int lo)
    {
      return -1;
    }

    // -----------------------------------------------------------------------
    //  Sorts the list into ascending order using the selection sort algorithm.
    // -----------------------------------------------------------------------
    public void selectionSort()
    {
      int minIndex;
      for (int i=0; i < list.length-1; i++)
          {
            //find smallest element in list starting at location i
            minIndex = i;
            for (int j = i+1; j < list.length; j++)
                if (list[j] < list[minIndex])
                      minIndex = j;

            //swap list[i] with smallest element
            int temp = list[i];
            list[i] = list[minIndex];
            list[minIndex] = temp;
          }
    }
}
```

```
// ****************************************************************
//    IntegerListSTest.java
//
//    Provide a menu-driven tester for the IntegerList class.
//    (Version S - for use in the linear search lab exercise).
//
// ****************************************************************
import java.util.Scanner;

public class IntegerListSTest
{
    static IntegerListS list = new IntegerListS (10);
    static Scanner scan = new Scanner(System.in);

    // -----------------------------------------------------------------
    //   Creates a list, then repeatedly print the menu and do what the
    //   user asks until they quit.
    // -----------------------------------------------------------------
    public static void main(String[] args)
    {
      printMenu();
      int choice = scan.nextInt();
      while (choice != 0)
          {
            dispatch(choice);
            printMenu();
            choice = scan.nextInt();
          }
    }

    // -------------------------------------
    //  Does what the menu item calls for.
    // -------------------------------------
    public static void dispatch(int choice)
    {
      int loc;
      switch(choice)
          {
          case 0:
            System.out.println("Bye!");
            break;
          case 1:
            System.out.println("How big should the list be?");
            int size = scan.nextInt();
            list = new IntegerListS(size);
            list.randomize();
            break;
          case 2:
            list.selectionSort();
            break;
          case 3:
            System.out.print("Enter the value to look for: ");
            loc = list.linearSearch(scan.nextInt());
            if (loc != -1)
                System.out.println("Found at location " + loc);
            else
                System.out.println("Not in list");
            break;
          case 4:
            list.print();
```

```
            break;
          default:
            System.out.println("Sorry, invalid choice");
          }
    }

    // ------------------------------------
    //   Prints the menu of user's choices.
    // ------------------------------------
    public static void printMenu()
    {
      System.out.println("\n   Menu   ");
      System.out.println("   ====");
      System.out.println("0: Quit");
      System.out.println("1: Create new list elements (** do this first!! **)");
      System.out.println("2: Sort the list using selection sort");
      System.out.println("3: Find an element in the list using linear search");
      System.out.println("4: Print the list");
      System.out.print("\nEnter your choice: ");
    }

}
```

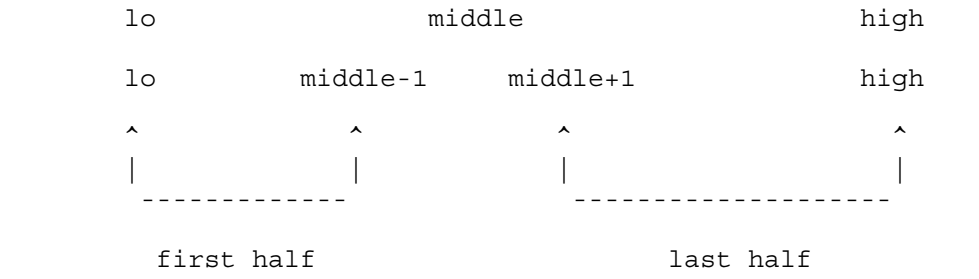
Recursive Binary Search

The binary search algorithm from Chapter 9 is a very efficient algorithm for searching an ordered list. The algorithm (in pseudocode) is as follows:

```
highIndex - the maximum index of the part of the list being searched
lowIndex - the minimum index of the part of the list being searched
target -- the item being searched for

//look in the middle
middleIndex = (highIndex + lowIndex) / 2
if the list element at the middleIndex is the target
   return the middleIndex
else
   if the list element in the middle is greater than the target
      search the first half of the list
   else
      search the second half of the list
```

Notice the recursive nature of the algorithm. It is easily implemented recursively. Note that three parameters are needed—the target and the indices of the first and last elements in the part of the list to be searched. To "search the first half of the list" the algorithm must be called with the high and low index parameters representing the first half of the list. Similarly, to search the second half the algorithm must be called with the high and low index parameters representing the second half of the list. The file *IntegerListB.java* contains a class representing a list of integers (the same class that has been used in a few other labs); the file *IntegerListBTest.java* contains a simple menu-driven test program that lets the user create, sort, and print a list and search for an item in the list using a linear search or a binary search. Your job is to complete the binary search algorithm (method binarySearchR). The basic algorithm is given above but it leaves out one thing: what happens if the target is not in the list? What condition will let the program know that the target has not been found? If the low and high indices are changed each time so that the middle item is NOT examined again (see the diagram of indices below) then the list is guaranteed to shrink each time and the indices "cross"—that is, the high index becomes less than the low index. That is the condition that indicates the target was not found.



Fill in the blanks below, then type your code in. Remember when you test the search to first sort the list.

```
private int binarySearchR (int target, int lo, int hi)
{
  int index;
  if ( _________________________________  ) // fill in the "not found" condition
      index = -1;
  else
      {
        int mid = (lo + hi)/2;
        if ( ___________________________________ ) // found it!
            index = mid;
        else if (target < list[mid])
                // fill in the recursive call to search the first half
                // of the list
            index = ____________________________________________________;
        else
```

```
                // search the last half of the list
            index = _____________________________________________________;
        }
      return index;
    }
```

Optional: The binary search algorithm "works" (as in does something) even on a list that is not in order. Use the algorithm on an unsorted list and show that it may not find an item that is in the list. Hand trace the algorithm to understand why.

```
// ****************************************************************
//   IntegerListB.java
//
//   Defines an IntegerList class with methods to create, fill,
//   sort, and search in a list of integers. (Version B - for use
//   in the binary search lab exercise)
//
// ****************************************************************

public class IntegerListB
{
    int[] list; //values in the list

    // ------------------------------------
    //   Creates a list of the given size
    // ------------------------------------
    public IntegerListB (int size)
    {
      list = new int[size];
    }

    // ----------------------------------------------------------------
    //   Fills the array with integers between 1 and 100, inclusive
    // ----------------------------------------------------------------
    public void randomize()
    {
      for (int i=0; i<list.length; i++)
          list[i] = (int)(Math.random() * 100) + 1;
    }

    // ---------------------------------------
    //   Prints array elements with indices
    // ---------------------------------------
    public void print()
    {
      for (int i=0; i<list.length; i++)
          System.out.println(i + ":\t" + list[i]);
    }

    // ----------------------------------------------------------------
    //   Returns the index of the first occurrence of target in the list.
    //   Returns -1 if target does not appear in the list.
    // ----------------------------------------------------------------
    public int linearSearch(int target)
    {
      int location = -1;
      for (int i=0; i<list.length && location == -1; i++)
          if (list[i] == target)
            location = i;
      return location;
    }
```

```
    // -----------------------------------------------------------------
    //   Returns the index of an occurrence of target in the list, -1
    //   if target does not appear in the list.
    // -----------------------------------------------------------------
    public int binarySearchRec(int target)
    {
      return binarySearchR (target, 0, list.length-1);
    }

    // -----------------------------------------------------------------
    //   Recursive implementation of the binary search algorithm.
    //   If the list is sorted the index of an occurrence of the
    //   target is returned (or -1 if the target is not in the list).
    // -----------------------------------------------------------------
    private int binarySearchR (int target, int lo, int hi)
    {
      int index;

      // fill in code for the search

      return index;
    }

    // -------------------------------------------------------------------------
    //  Sorts the list into ascending order using the selection sort algorithm.
    // -------------------------------------------------------------------------
    public void selectionSort()
    {
      int minIndex;
      for (int i=0; i < list.length-1; i++)
          {
            //find smallest element in list starting at location i
            minIndex = i;
            for (int j = i+1; j < list.length; j++)
                if (list[j] < list[minIndex])
                      minIndex = j;

            //swap list[i] with smallest element
            int temp = list[i];
            list[i] = list[minIndex];
            list[minIndex] = temp;
          }
    }

}
```

```
// ****************************************************************
//   IntegerListBTest.java
//
//   Provides a menu-driven tester for the IntegerList class.
//   (Version B - for use with the binary search lab exerice)
//
// ****************************************************************
import java.util.Scanner;

public class IntegerListBTest
{
    static IntegerListB list = new IntegerListB (10);
    static Scanner scan = new Scanner(System.in);

    // -----------------------------------------------------------------
    //  Create a list, then repeatedly print the menu and do what the
    //  user asks until they quit.
    // -----------------------------------------------------------------
    public static void main(String[] args)
    {
      printMenu();
      int choice = scan.nextInt();
      while (choice != 0)
          {
            dispatch(choice);
            printMenu();
            choice = scan.nextInt();
          }
    }

    // -----------------------------------------------------
    //  Does what the menu item calls for.
    // -----------------------------------------------------
    public static void dispatch(int choice)
    {
      int loc;
      switch(choice)
          {
          case 0:
            System.out.println("Bye!");
            break;
          case 1:
            System.out.println("How big should the list be?");
            int size = scan.nextInt();
            list = new IntegerListB(size);
            list.randomize();
            break;
          case 2:
            list.selectionSort();
            break;
          case 3:
            System.out.print("Enter the value to look for: ");
            loc = list.linearSearch(scan.nextInt());
            if (loc != -1)
                System.out.println("Found at location " + loc);
            else
                System.out.println("Not in list");
            break;
          case 4:
            System.out.print("Enter the value to look for: ");
```

```
            loc = list.binarySearchRec(scan.nextInt());
            if (loc != -1)
                System.out.println("Found at location " + loc);
            else
                System.out.println("Not in list");
            break;
          case 5:
            list.print();
            break;
          default:
            System.out.println("Sorry, invalid choice");
          }
    }

    // ----------------------------
    //  Prints the user's choices.
    // ----------------------------
    public static void printMenu()
    {
      System.out.println("\n   Menu   ");
      System.out.println("   ====");
      System.out.println("0: Quit");
      System.out.println("1: Create new list elements (** do this first!! **)");
      System.out.println("2: Sort the list using selection sort");
      System.out.println("3: Find an element in the list using linear search");
      System.out.println("4: Find an element in the list using binary search");
      System.out.println("5: Print the list");
      System.out.print("\nEnter your choice: ");
    }

}
```

A List of Employees

The files *Employee.java* and *Payroll.java* contain a definition of a simple list of hourly wage employees. An employee has a name, number of hours worked, and an hourly pay rate. The Payroll class is the list of employees. Currently there is a method in the class, *public void readPayrollInfo(String file)*, that reads in the employee information from a file and sets up the employee list. Your job is to add a recursive method that determines the number of employees who worked overtime (more than 40 hours). The method *numOvertime* is already defined. It is the public method that would be used by a program. It calls your method *int overtime (int start)* which will do all the work.

1. Complete the overtime method. The parameter *start* is the index of the first element in the part of the array being processed. Recall that to recursively process the elements in an array in sequential order the strategy is to "process" the first element (in this case count it if the number of hours worked is greater than 40), then call the method recursively to process the rest of the array. Don't forget the base case.

2. Complete the test program *Overtime.java* to test your method. The program currently has code to read in the name of the file that contains employee data. You need to add code to instantiate a Payroll object, read the data in, then call the *numOvertime* method to determine how many employees worked overtime.

3. Run the program at least twice using the files *payroll.dat* and *payroll2.dat* as input.

```
// ********************************************************
//   Employee.java
//
//   Represents an hourly wage worker.
// ********************************************************

public class Employee
{
    String name;
    int hours;        // hours worked
    double rate;      // hourly pay rate

    // -----------------------------------------------
    //  Sets up the Employee object with the given data.
    // -----------------------------------------------
    public Employee (String name, int hours, double rate)
    {
      this.name = name;
      this.hours = hours;
      this.rate = rate;
    }

    // -----------------------------------------------
    //  Returns the number of hours worked.
    // -----------------------------------------------
    public int getHours ()
    {
      return hours;
    }
}
```

```
// ********************************************************************
//   Payroll.java
//
//   Represents a list of employees.
// ********************************************************************

import java.util.Scanner;
import java.util.*;
import java.io.*;

public class Payroll
{
    final int MAX = 30;
    Employee[] payroll = new Employee[MAX];
    int numEmployees = 0;

    // -------------------------------------------------------
    //  Reads the list of employee wage data from the given
    //  file.
    // -------------------------------------------------------
    public void readPayrollInfo(String file)
    {
      String line;       // a line in the file
      String name;       // name of an employee
      int hours;         // hours worked
      double rate;       // hourly pay rate

      Scanner fileScan, lineScan;

      try
          {
            fileScan = new Scanner (new File(file));

            while (fileScan.hasNext())
                {
                  line = fileScan.nextLine();

                  lineScan = new Scanner(line);
                  name = lineScan.next ();

                  try
                      {
                        hours = lineScan.nextInt();
                        rate = lineScan.nextDouble();
                        payroll[numEmployees] = new Employee (name, hours, rate);
                        numEmployees++;
                      }
                  catch (InputMismatchException exception)
                      {
                        System.out.println ("Error in input. Line ignored.");
                        System.out.println (line);
                      }
                  catch {ArrayIndexOutOfBoundsException exception)
                      {
                        System.out.println ("Too many employees!");
                      }
                }
            fileScan.close();

          }
```

```
      catch (FileNotFoundException exception)
          {
            System.out.println ("The file " + file + " was not found.");
          }
      catch (IOException exception)
          {
            System.out.println (exception);
          }
    }


    // ----------------------------------------
    //   Returns the number of employees who
    //   worked over 40 hours; the helper method
    //   overtime is called to do all the work.
    // ----------------------------------------
    public int numOvertime ()
    {
      return overtime (0);
    }

    // ------------------------------------------------
    //   Returns the number of employees in the part
    //   of the list from index start to the end who
    //   worked more than 40 hours.
    // ------------------------------------------------
    private int overtime (int start)
    {

    }
}


// ****************************************************************
//   Overtime.java
//
//   Reads a file of employee payroll information and determines
//   how many employees worked more than 40 hours.
// ****************************************************************

import java.util.Scanner;

public class Overtime
{
    public static void main (String[] args)
    {
      String fileName;    // Name of the file containing employee data
      Scanner scan = new Scanner(System.in);

      System.out.println ("\nPayroll Program");
      System.out.print ("Enter the name of the file containing payroll data: ");
      fileName = scan.nextLine();

      // Instantiate a Payroll object and read in the data from the file


      // Print the number of workers who worked overtime.

    }
}
```

payroll.dat

```
Smith 45 13.50
Jones 39 23.75
Doe 40 17.80
Moe 30 21.90
Walker 41 14.60
Walton 57 8.95
Taylor 40 16.75
Lewis 40 35.50
Abbott 43 12.70
Who 39 33.95
Herrod 21 19.90
James 49 13.50
Summers 40 20.00
Winter 40 18.75
Farthington 38 24.50
Walsh 42 45.70
```

payroll2.dat

```
Jones 40 9.75
Ricardo 35 13.69
Smith 40 20.00
Smythe 40 17.80
```

Sierpinski Triangles

A Sierpinski triangle is a geometric figure that may be constructed as follows:

1. Draw a triangle.
2. Draw a new triangle by connecting the midpoints of the three sides of your original triangle. This should split your triangle into four smaller triangles, one in the center and three around the outside.
3. Repeat (2) for each of the outside triangles (not the center one). Each of them will split into four yet smaller triangles. Repeat for each of their outside triangles.. and for each of the new ones.. and so on, forever. Draw a few rounds of this on paper to see how it works. Check out the demo at http://cs.roanoke.edu/labs4e/demo.html to see how the program works on screen.

Your job is to write an applet that draws a Sierpinski triangle. Think about the following:

- A Sierpinski triangle is a recursively defined structure, since each of the three outer triangles formed by joining the midpoints is itself a Sierpinski triangle.
- In practice you don't want to go on with the process "forever" as suggested above, so we'll limit how deep it goes. Define the *depth* of a Sierpinski triangle as the number of directly nested triangles at the deepest point. So a Sierpinski triangle consisting of a single triangle has depth 0; when a triangle is drawn inside of it, the resulting Sierpinski triangle has depth 1; when the three outside triangles have triangles drawn inside of them, the resulting triangle is depth 2, and so on. A depth of 10 or 11 gives a nice looking triangle in a reasonable amount of time. (The demo uses depth 10.) Smaller depths are interesting in that you can see more of the construction; higher depths generally take too long for casual viewing.
- A triangle is a polygon, so you'll use the drawPolygon method. Remember that it takes an array containing the x coordinates, an array containing the y coordinates, and an integer indicating how many points should be drawn (3 for a triangle). Refer to Chapter 7 or the appendix (the Graphics class) to refresh your memory on this.
- Your initial triangle should look like the one in the demo—one point at the top center of the applet and one point in each lower corner.
- Your overall program is quite simple. The paint method will just call a recursive method sierpinski, passing it the Graphics object, the points of the initial triangle to be drawn, and the initial depth (0). Make the variables that hold the initial points instance variables and initialize them in the init method. The sierpinski method will then check to see if the desired depth has been exceeded (the depth can be a constant), and if not, draw the triangle, then call itself recursively to drawn the three Sierpinski triangles that will be embedded. Note that it will have to figure out what points to pass to each of these, and that each recursive call increases the depth by one.

When this works (that is, when your triangle looks like the one in the demo), modify it so that when the user clicks a new, random Sierpinski triangle is drawn. (The demo does this too.) This just means choosing three random points and repainting—nothing will change in your paint or sierpinski methods (if you followed the guidelines above). For fun you can choose a random color each time as well, as the demo does.

Modifying the Koch Snowflake

The Koch snowflake is a fractal generated by starting with 3 line sements forming an equilateral triangle (a Koch fractal of order 1). The algorithm for generating higher order Koch fractals involves splitting each line segment into three equal segments then replacing the middle segment by two line segments that protrude outward. The same algorithm is then recursively applied to each of the 4 new line segments. In the basic Koch snowflake the two protruding line segments meet at a 60 degree angle. These line segments form an equilateral triangle with the middle segment that is removed. (See the discussion in Chapter 11 for more details.) Files *KochSnowflake.java* and *KochPanel.java* contain slight modifications to the program from the text that generates Koch snowflakes (Listings 11.6 and 11.7). Copy these files to your directory, compile them, and run the program in the appletviewer to see how it works (you may use the file *Koch.html* to run the program). In this exercise you will generalize the pattern to allow for triangles other than equilateral ones to be built on the middle third segment. In the drawFractal method this involves changing the calculation of x3 and y3, the coordinates of the protrusion point. The following calculations are equivalent to those currently in the program:

```
x3 = (int) (x2 + (cosine * deltaX - sine * deltaY)/3);
y3 = (int) (y2 + (cosine * deltaY + sine * deltaX)/3);
```

where cosine is the cosine of 60 degrees (which is 1/2) and sine is the sine of 60 degrees (which is the square root of 3 over 2). These equations are generalizable to angles other than 60. In this exercise you will generalize the program to work for angles other than 60. The angle will be controlled by increase and decrease buttons in the same way that the order is currently controlled. Do the following:

1. In KochPanel.java,
   - Add instance variables *angle*, *sine*, and *cosine*. Angle will be an integer and sine and cosine type double.
   - In the constructor, set angle to 60 (this will be the default), and set sine to Math.sin (Math.PI / 3) and cosine to Math.cos (Math.PI / 3).
   - In drawFractal, replace the current calculations for x3 and y3 with those given above.

2. Compile and run the program. It should behave just as before.

3. To add controls to allow the angle to change, do the following:
   - In KochPanel, add two public methods *getAngle()* that returns the angle (type int), and *setAngle (int newAngle)* that sets the angle to be the value of *newAngle* and sets *sine* and *cosine* of that angle. Remember that the sin and cos methods in the Math class must have arguments that are in radians not degrees so you need to multiply the angle (which is in degrees) by Math.PI divided by 180 (pi/180 is the degree to radian conversion factor).
   - In KochSnowflake.java, add a new "tools" panel for the buttons to increase and decrease the angle. This panel will contain two buttons and a label giving the current angle. A horizontal box layout should be used. The applet should be added as a listener for the buttons.
   - The new tools panel should be added to the appletPanel. To accomodate it change APPLET_HEIGHT to 480.
   - The method actionPerformed must be modified to take action if the event source was one of the new buttons. If the source was the button to increase the angle, increase the angle by 10 degrees (you can get the current angle using the method you added to KochPanel); if the source was the button to decrease the angle, decrease it by 10 degrees. The new angle should be between 10 and 170 (inclusive)—you should add constants (similar to MIN and MAX) for the minimum and maximum angles.

4. Compile and run the program. Play with the angles and order to see what fractal patterns are generated.

```
// ********************************************************************
//  KochSnowflake.java               Author:  Lewis/Loftus
//
//  Demonstrates the use of recursion in graphics.
// ********************************************************************

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class KochSnowflake extends JApplet implements ActionListener
{
    private final int APPLET_WIDTH = 400;
    private final int APPLET_HEIGHT = 440;

    private final int MIN = 1, MAX = 9;

    private JButton increase, decrease;
    private JLabel titleLabel, orderLabel;
    private KochPanel drawing;
    private JPanel appletPanel, tools;

    // -----------------------------------------------------------
    //   Sets up the components for the applet.
    // -----------------------------------------------------------
    public void init()
    {
      tools = new JPanel ();
      tools.setLayout (new BoxLayout (tools, BoxLayout.X_AXIS));
      tools.setPreferredSize (new Dimension (APPLET_WIDTH, 40));
      tools.setBackground (Color.yellow);
      tools.setOpaque (true);

      titleLabel = new JLabel ("The Koch Snowflake");
      titleLabel.setForeground (Color.black);

      increase = new JButton ("Increase");
      increase.setMargin (new Insets (0, 0, 0, 0));
      increase.addActionListener (this);

      decrease = new JButton ("Decrease");
      decrease.setMargin (new Insets (0, 0, 0, 0));
      decrease.addActionListener (this);

      orderLabel = new JLabel ("Order: 1");
      orderLabel.setForeground (Color.black);

      tools.add (titleLabel);
      tools.add (Box.createHorizontalStrut (40));
      tools.add (decrease);
      tools.add (increase);
      tools.add (Box.createHorizontalStrut (20));
      tools.add (orderLabel);

      drawing = new KochPanel (1);

      appletPanel = new JPanel ();
      appletPanel.add (tools);
      appletPanel.add (drawing);
```

```
      getContentPane().add (appletPanel);

      setSize (APPLET_WIDTH, APPLET_HEIGHT);
    }

    // -----------------------------------------------------------
    //  Determines which button was pushed, and sets the new order
    //  if it is in range.
    // -----------------------------------------------------------
    public void actionPerformed (ActionEvent event)
    {
      int order = drawing.getOrder ();

      if (event.getSource() == increase)
          order++;
      else
          order--;

      if (order >= MIN && order <= MAX)
          {
            orderLabel.setText ("Order: " + order);
            drawing.setOrder (order);
            repaint();
          }
    }
}
```

```
// ********************************************************************
//  KochPanel.java         Author:  Lewis/Loftus
//
//  Represents a drawing surface on which to paint a Koch Snowflake.
// ********************************************************************

import java.awt.*;
import javax.swing.JPanel;

public class KochPanel extends JPanel
{
    private final int PANEL_WIDTH = 400;
    private final int PANEL_HEIGHT = 400;

    private final double SQ = Math.sqrt (3.0) / 6;

    private final int TOPX = 200, TOPY = 20;
    private final int LEFTX = 60, LEFTY = 300;
    private final int RIGHTX = 340, RIGHTY = 300;

    private int current;     // current order

    // -----------------------------------------------------------------
    //  Sets the initial fractal order to the value specified.
    // -----------------------------------------------------------------
    public KochPanel (int currentOrder)
    {
      current = currentOrder;
      setBackground (Color.black);
      setPreferredSize (new Dimension (PANEL_WIDTH, PANEL_HEIGHT));
    }

    // -----------------------------------------------------------------
    //  Draws the fractal recursively.  The base case is order 1 for
    //  which a simple straight line is drawn.  Otherwise three
    //  intermediate points arae computed, and each line segment is
    //  drawn as a fractal.
    // -----------------------------------------------------------------
    public void drawFractal (int order, int x1, int y1, int x5, int y5,
                      Graphics page)
    {
      int deltaX, deltaY, x2, y2, x3, y3, x4, y4;

      if (order ==1)
          page.drawLine (x1, y1, x5, y5);
      else
          {
            deltaX = x5 - x1;    // distance between end points
            deltaY = y5 - y1;

            x2 = x1 + deltaX /3;
            y2 = y1 + deltaY / 3;

            x3 = (int) ((x1 + x5)/2 + SQ * (y1 - y5));
            y3 = (int) ((y1 + y5)/2 + SQ * (x5 - x1));

            x4 = x1 + deltaX * 2 / 3;
            y4 = y1 + deltaY * 2 / 3;

            drawFractal (order - 1, x1, y1, x2, y2, page);
```

```
            drawFractal (order - 1, x2, y2, x3, y3, page);
            drawFractal (order - 1, x3, y3, x4, y4, page);
            drawFractal (order - 1, x4, y4, x5, y5, page);
          }
    }

    // -----------------------------------------------------------------
    //  Performs the initial calls to the drawFractal method.
    // -----------------------------------------------------------------
    public void paintComponent (Graphics page)
    {
      super.paintComponent (page);

      page.setColor (Color.green);

      drawFractal (current, TOPX, TOPY, LEFTX, LEFTY, page);
      drawFractal (current, LEFTX, LEFTY, RIGHTX, RIGHTY, page);
      drawFractal (current, RIGHTX, RIGHTY, TOPX, TOPY, page);
    }

    // -----------------------------------------------------------------
    //  Sets the fractal order to the specified value.
    // -----------------------------------------------------------------
    public void setOrder (int order)
    {
      current = order;
    }

    // -----------------------------------------------------------------
    //  Returns the current order.
    // -----------------------------------------------------------------
    public int getOrder ()
    {
      return current;
    }
}
```

koch.html

```
<html>
<title>Koch Snowflake</title>
<applet CODE="KochSnowflake.class" HEIGHT=400 WIDTH=440>
</applet>
</html>
```