Placing Exception Handlers

File *ParseInts.java* contains a program that does the following:

- Prompts for and reads in a line of input
- Uses a second Scanner to take the input line one token at a time and parses an integer from each token as it is extracted.
- Sums the integers.
- Prints the sum.

Save ParseInts to your directory and compile and run it. If you give it the input

```
10 20 30 40
```

it should print

```
The sum of the integers on the line is 100.
```

Try some other inputs as well. Now try a line that contains both integers and other values, e.g.,

```
We have 2 dogs and 1 cat.
```

You should get a NumberFormatException when it tries to call *Integer.parseInt* on “We”, which is not an integer. One way around this is to put the loop that reads inside a *try* and catch the NumberFormatException but not do anything with it. This way if it’s not an integer it doesn’t cause an error; it goes to the exception handler, which does nothing. Do this as follows:

- Modify the program to add a try statement that encompasses the entire *while* loop. The *try* and opening { should go before the *while,* and the *catch* after the loop body. Catch a NumberFormatException and have an empty body for the catch.
- Compile and run the program and enter a line with mixed integers and other values. You should find that it stops summing at the first non-integer, so the line above will produce a sum of 0, and the line “1 fish 2 fish” will produce a sum of 1. This is because the entire loop is inside the *try,* so when an exception is thrown the loop is terminated. To make it continue, move the *try* and *catch* inside the loop. Now when an exception is thrown, the next statement is the next iteration of the loop, so the entire line is processed. The dogs-and-cats input should now give a sum of 3, as should the fish input.

```
// ****************************************************************
// ParseInts.java
//
// Reads a line of text and prints the integers in the line.
//
// ****************************************************************

import java.util.Scanner;

public class ParseInts
{
    public static void main(String[] args)
    {
      int val, sum=0;
      Scanner scan = new Scanner(System.in);
      String line;

      System.out.println("Enter a line of text");
      Scanner scanLine = new Scanner(scan.nextLine());

      while (scanLine.hasNext())
          {
            val = Integer.parseInt(scanLine.next());
            sum += val;
          }
      System.out.println("The sum of the integers on this line is " +
      sum);
    }

}
```

Reading from and Writing to Text Files

Write a program that will read in a file of student academic credit data and create a list of students on academic warning. The list of students on warning will be written to a file. Each line of the input file will contain the student name (a single String with no spaces), the number of semester hours earned (an integer), the total quality points earned (a double). The following shows part of a typical data file:

```
Smith 27  83.7
Jones 21  28.35
Walker 96 182.4
Doe 60 150
```

The program should compute the GPA (grade point or quality point average) for each student (the total quality points divided by the number of semester hours) then write the student information to the output file if that student should be put on academic warning. A student will be on warning if he/she has a GPA less than 1.5 for students with fewer than 30 semester hours credit, 1.75 for students with fewer than 60 semester hours credit, and 2.0 for all other students. The file *Warning.java* contains a skeleton of the program. Do the following:

1. Set up a Scanner object *scan* from the input file and a PrintWriter *outFile* to the output file inside the try clause (see the comments in the program). Note that you'll have to create the PrintWriter from a FileWriter, but you can still do it in a single statement.
2. Inside the while loop add code to read and parse the input—get the name, the number of credit hours, and the number of quality points. Compute the GPA, determine if the student is on academic warning, and if so write the name, credit hours, and GPA (separated by spaces) to the output file.
3. After the loop close the PrintWriter.
4. Think about the exceptions that could be thrown by this program:
   - A FileNotFoundException if the input file does not exist
   - A NumberFormatException if it can't parse an int or double when it tries to - this indicates an error in the input file format
   - An IOException if something else goes wrong with the input or output stream

   Add a catch for each of these situations, and in each case give as specific a message as you can. The program will terminate if any of these exceptions is thrown, but at least you can supply the user with useful information.
5. Test the program. Test data is in the file *students.dat.* Be sure to test each of the exceptions as well.

```
// **********************************************************************
//   Warning.java
//
//   Reads student data from a text file and writes data to another text file.
// **********************************************************************

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

public class Warning
{
    // -------------------------------------------------------------------
    //   Reads student data (name, semester hours, quality points) from a
    //   text file, computes the GPA, then writes data to another file
    //   if the student is placed on academic warning.
    // -------------------------------------------------------------------
    public static void main (String[] args)
    {
      int creditHrs;          // number of semester hours earned
      double qualityPts;      // number of quality points earned
      double gpa;             // grade point (quality point) average

      String line, name, inputName = "students.dat";
      String outputName = "warning.dat";
      try
          {
            // Set up scanner to input file

            // Set up the output file stream

            // Print a header to the output file
            outFile.println ();
            outFile.println ("Students on Academic Warning");
            outFile.println ();

            // Process the input file, one token at a time

            while ()
                {
                  // Get the credit hours and quality points and
                  // determine if the student is on warning. If so,
                  // write the student data to the output file.
                }

            // Close output file
          }
      catch (FileNotFoundException exception)
          {
            System.out.println ("The file " + inputName + " was not
            found.");
          }
      catch (IOException exception)
          {
            System.out.println (exception);
          }
      catch (NumberFormatException e)
          {
            System.out.println ("Format error in input file: " + e);
          }

    }
}
```

students.dat

```
Smith 27  83.7
Jones 21  28.35
Walker 96 182.4
Doe 60 150
Wood 100 400
Street 33 57.4
Taylor 83 190
Davis 110 198
Smart 75 2 92.5
Bird 84 168
Summers 52 83.2
```

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 $j\text{-}1^{st}$ 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".

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.

```
lo                  middle                     high

lo        middle-1     middle+1                high

^            ^            ^                      ^
|            |            |                      |
 ------------              ----------------------

  first half                     last half
```

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: ");
    }
}
```