A Brief Tutorial in Traditional vs. OO Programming Using Java

Richard A. Johnson
CIS Dept., Southwest Missouri State University
Springfield, MO, 65804, USA
richardjohnson@smsu.edu

Abstract

Object-orientation (OO) is a relatively recent approach to addressing problems in systems development. However, OO is viewed by many as difficult to learn. This paper discusses how object-oriented programming is taught at one university and directly compares, through the use simple, straightforward examples, the traditional and OO methods of programming using the Java language. The paper demonstrates that in many important ways, OO is definitely superior to traditional methods, yet simple to understand.

Keywords: object-orientation, object-oriented programming, structured programming, systems development

1. INTRODUCTION

Information technology (IT) has long been recognized as crucial for creating and sustaining competitive advantage in business, and information systems development (ISD) is a critical element of IT. However, a "software crisis" has consistently plagued ISD efforts (Fayad, Tsai, and Fulghum, 1996). This crisis is fueled by user expectations for the rapid deployment of increasingly sophisticated systems of exceptional quality (Booch, 1994). The duration of this crisis has motivated some to rename it a "chronic affliction" (Pressman, 1996). A relatively recent approach to ISD, namely object-orientation, claims to be our best chance of successfully addressing this affliction (Johnson, 2000; Iivari, Hirschein, and Klein, 2000-2001).

A related problem currently exists in effectively teaching college students the basics of object-oriented (OO) systems development, particularly OO programming (OOP), the foundation of such development. These students, typically in computer science or computer information systems programs, often enter college with little or no formal training in programming. Some students do take a high school course or two in computer programming, or attempt to teach themselves some programming, but most have virtually no background in true OOP. Due to the ongoing software-crisis and the ever-increasing importance of OOP for both Internet and traditional business information systems, the question then becomes how best to teach such college students the important concepts and skills of OOP, given their limited backgrounds. The question is even more critical given the consensus view that learning OO can be very difficult (Sheetz, Irwin, Tegarden, Nelson, and Monarchi, 1997).

The purpose of this paper is to describe in some detail how one approach may be used to introduce the student to OO concepts and to effectively contrast traditional structured programming and OOP. The paper begins with a brief explanation of how OOP is introduced in the author's courses. Following is a concise introduction to Java syntax and how it incorporates object technology. Finally, a

simple application is developed using both traditional and OO approaches in order to directly compare the merits of each.

2. OOP AT SOUTHWEST MISSOURI STATE UNIVERSITY

OOP is taught at SMSU in two different departments, Computer Science (CSC), which is in the College of Applied and Natural Sciences, and Computer Information Systems (CIS), which is in the College of Business Administration. I teach primarily three courses in CIS: Web Application Development for Business I and II (CIS 275/375), and Object Technology I (CIS 260). There is also an Object Technology II (CIS 360) course offered. Although there is significant application of object-oriented concepts and techniques involved in web application development, this paper will focus on teaching students OOP in a beginning traditional programming course.

CIS 260 at SMSU has a prerequisite of Program Design and Development (CIS 202), a course that typically uses Visual Basic to teach introductory structured programming concepts with simple applications. Students who take CIS 260 almost always have also taken Windows Programming with Development Tools (CIS 224), which includes a more rigorous application of Visual Basic. So, the typical CIS 260 student has two semesters of Visual Basic, all taught strictly within the structured programming paradigm, before studying OOP. (In the fall of 2003, CIS 202 and CIS 224 will begin using VB.NET, which is now purely object-oriented—this fact further emphasizes the need for an adequate understanding of OO concepts and techniques.)

The language currently used in CIS 260 is Java. Since nearly all students in CIS 260 have never studied Java, it is necessary to first teach them the basic syntax. They should already understand the elements of structured programming: data types, the elementary structures (sequence, selection, and repetition), function calls, arrays, and data input/output. The challenge in CIS 260 is to teach the students a new language as well as the new concepts and techniques of the OOP paradigm.

3. UNDERSTANDING BOTH TRADITIONAL PROGRAMMING AND OOP

One school of thought in teaching OOP is to discard any reference at all to structured programming. In fact, many feel that learning structured programming interferes with one's ability to learn OOP. However, I disagree. I believe that students need to clearly understand both structured programming and OOP so that they appreciate the differences and can thereby make conscious efforts to pursue one or the other more effectively. To fail to understand the differences can lead to confusion and harmful intermingling of the two paradigms.

The approach I take is to first teach students the Java syntax by applying it to all the old familiar structured programming concepts. This firmly reinforces their knowledge of basic traditional programming. Then, I introduce how Java is used to create purely object oriented applications, starting with the simplest examples. This approach provides a stark contrast between structured programming and OOP. Clearly understanding the differences between the two should help students become better OO programmers. In fact, I complete the course by showing examples of creating identical applications with the two distinct methodologies. Then, I expect the students to be able to do the same, as demonstrated by their performance on a final exam.

4. TRADITIONAL PROGRAMMING USING JAVA

The text I currently use is *Murach's Beginning Java 2* (Steelman, 2002). While this may not be the best choice for this approach to teaching OOP, it performs sufficiently well. Some of the examples I use in this paper will be taken from this text. I will be using Sun's Java SDK version 1.4 (which may be downloaded from java.sun.com). Following, I will give a very condensed version of the approach that I take in teaching both structured and OO programming using the Java language.

Hello, World!

Of course, the simplest program ever written is the infamous "Hello, world!" (line numbers are added to program code only for discussion purposes):

```
1   public class FirstApp{
2     public static void main(String[] args){
3       System.out.println("Hello, world!");
4     }
5   }
```

Figure 1—"Hello, world!" (FirstApp.java)

Students are instructed that some things about the Java syntax may seem peculiar, primarily because it is new. First, program code is stored in a class (there is usually one class in a .java file). In Figure 1, the class name is FirstApp (line 1) and the class is public (so that other classes may have access to it as necessary). Everything contained in the class is enclosed in {}'s (lines 1 and 5). Line 2 declares a method called "main" (with the parameter "String[] args"), which is also public and static, with a return type of void. The term "static" implies that this method has essentially nothing to do with objects. (All this will seem strange to most students and is explained in detail to them later. For now, they are informed of what is going on only at a very high level.)

Everything within the main method is enclosed within {}'s (lines 2 and 4). Line 3 represents the basic program processing, which is simply to print the string "Hello, world!" It is at this time that students first hear about OO concepts. The method "println" is being called and belongs to the "out" object (the monitor), which belongs to the "System" class. Of course, this still won't mean much to students, although they may grasp that a class can be used to create objects and these objects have access to methods contained within the class. However, this can be confusing, so it is probably best for them to understand it only as Java's syntax for displaying output to the monitor. The essentials to derive from this simplest of examples is that a Java program contains a class, the class must be declared in the program, the class has a main method, and this main method contains code to perform processing. In the example of Figure 1, since the class name is FirstApp, the file is saved as FirstApp.java (note that Java is case sensitive).

If the Java SDK has been installed on the user's computer (see java.sun.com), then the program FirstApp.java (Figure 1) can be compiled and run. This can be done using DOS or some type of Java IDE. A simple IDE that I use is BlueJ, available free from www.bluej.org. The output from running FirstApp.java in BlueJ is shown in Figure 2.

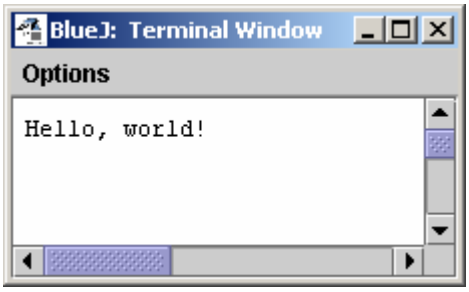


Figure 2—Output from FirstApp.java (see Figure 1 for code)

More Basic Java Syntax

Next, the student learns many of the intricate details of the Java syntax, such as how to declare and initialize primitive data types (int counter = 1; or double price = 14.95;), how to perform assignment (counter = counter + 1; or the equivalent counter++;), or how to create a string variable (String name = "Richard Johnson";). There is very little OO involved here (except for the fact that "name" is actually an object from the String class, which is the reason that "String" is capitalized, but that isn't extremely important to the student now either). However, the student will eventually need to learn some very rudimentary OO concepts and terminology to understand basic String operations. For example, Figure 3 contains code that compares Strings.

```
1   public class NameApp{
2     public static void main(String[] args){
3       String name1 = "Johnson";
4       String name2 = "Smith";
5         if (name1.equals(name2))
6           System.out.println("Same names!");
7         else
8           System.out.println("Different names!");
9         }
10  }
```

Figure 3—Comparing Strings (NameApp.java)

The key is line 5. In Java, one cannot write if(name1 == name2)… when name1 and name2 are Strings. (You can, however, write if(a == b)… when a and b are, say, integers.) Since name1 is a String object, one must use a method that belongs to the String class to compare it to another String, and the general syntax for calling a method of a class is ClassName.methodName().

Again, the student gets an indication that classes can have methods and learns how to call a method that is coded within a another class. However, this actually isn't a purely OO concept and shouldn't be particularly stressful to the student.

Method Calls

A method can be thought of as simply residing in a class. To call that method from another class requires the syntax ClassName.methodName(). (The student should note that class names are capitalized while method names are not, and that, by convention, new words within an identifier are capitalized.) Calling methods that belong to other classes has further application for integer and double data types. For example, suppose a user enters the number "12" into a Java program. Java stores all user input as a String object. If the user input was stored in a variable called quantityString, then the Java syntax to convert that input into an integer variable called quantity is

```
int quantity = Integer.parseInt (quantityString);
```

Notice that the method parseInt() belongs to the Integer class. So, the student is now familiar with the concept that in Java, built-in classes (such as System, String, and Integer) have special methods that can be called using the syntax ClassName.methodName() or ClassName.objectName.methodName().

Java Packages

Another important object-oriented concept that the beginning Java student must learn early on is that many Java classes must be imported to a program in order to use their methods. Also, the student must learn that Java classes are grouped together in packages, so one must reference the package name when importing classes. For example, the String and Integer classes belong to the java.lang package, which is automatically imported when the programmer creates his own Java class (program). But classes needed to provide a program with a GUI with which the user can provide input are not automatically imported. Figure 4 provides an example.

```
1  import javax.swing.JOptionPane;
2  public class EnterNameApp{
3     public static void main(String[] args){
4        String inputString =
            JOptionPane.showInputDialog(
            "Enter your first name: ");
6        String message = "Your first name is " +
            inputString;
7        JOptionPane.showMessageDialog(null,
            message);
8     }
9  }
```

Figure 4—Using a GUI in Java (EnterNameApp.java)

Line 1 imports the JOptionPane class that belongs to the Swing package (the Swing package contains many classes used for GUI's). Line 4 declares a variable called inputString into which the user will store a first name. This is accomplished using the showInputDialog() method of the JOptionPane class. The argument of showInputDialog() is a literal string, which will be displayed to the user in the GUI. Line 6 creates a String variable (i.e., object) called message using a literal string, a concatenation operator ("+") and the variable inputString. Line 7 uses another method of the JOptionPane class called showMessageDialog() with two arguments. The first argument (null) causes the GUI to be centered on the screen. The second argument is the String that the programmer wishes to display. The two windows displayed by this program are shown in Figure 5.

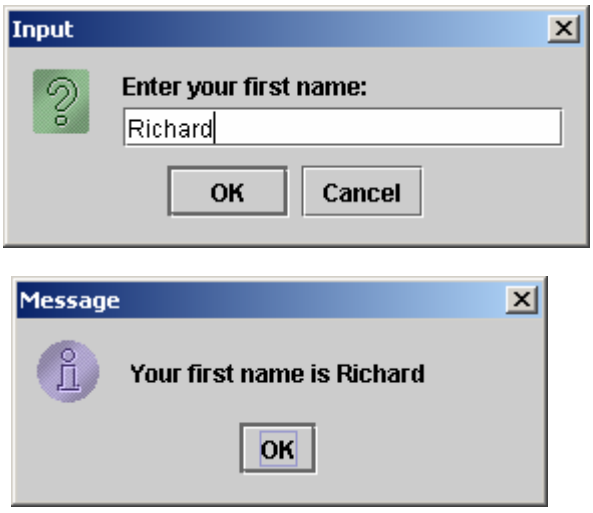


Figure 5—Output from EnterNameApp.java (see Figure 4 for code)

In summary, the student learns that Java methods belong to classes that in turn belong to packages, and that to use such methods requires the programmer to import those packages. In fact, to import all classes

in the Swing package, the programmer would begin his class with the code import javax.swing.*;. So, the student continues to learn a little about OO, but only as it applies to prewritten Java classes. Again, this really isn't at the heart of OO systems development (OOSD). At this point, the student is still just learning Java syntax—the Java way of doing things—but he hasn't learned how to create OO systems of his own.

Modularization

Of course, a key to creating any application, structured or OO, is to modularize the code, which means to organize code into smaller, logical units (called modules, procedures, methods, functions, etc.) that can be called by a program when needed. Modularization reduces complexity; it also speeds initial development and ongoing maintenance of applications. Modularization is even more critical to OOSD since a class is considered an essential module that "classifies" or defines real-world objects in the system. For example, an object-oriented bookseller application might have classes that are used to define real-world system objects such as books.

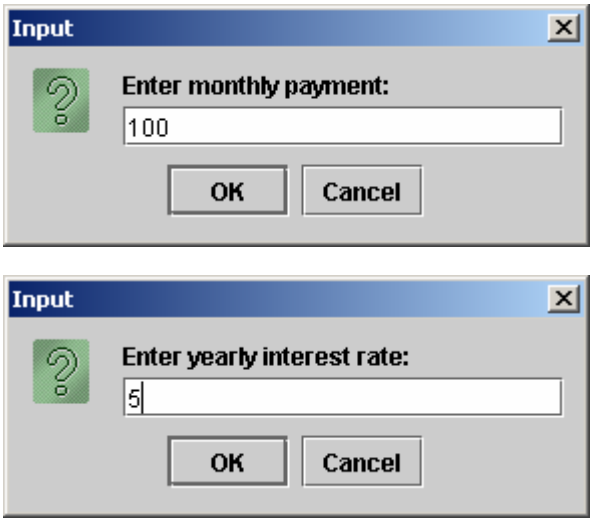
User-defined Methods

In Java, the method is the most basic type of module, and methods are contained in classes (whether the application is OO or not). In Java, a method that is used in a strictly traditional sense is called a static method (i.e., objects are not involved). The following short non-OO program demonstrates how Java handles methods (this is especially important for understanding how methods are used in OO programs later).

```
1   import javax.swing.*;

2   public class FutureValueApp{
3     public static void main(String[] args){

4       String paymentString =
          JOptioPane.showInputDialog(
            "Enter monthly payment: ");
5       double monthlyPayment =
            double.parseDouble(paymentString);
6       String rateString =
            JOptionPane.showInputDialog(
            "Enter yearly interest rate: ");
7       double interestRate = Double.parseDouble
                  (rateString);
8       double monthlyInterestRate =
            interestRate/12/100;
9       String yearsString =
          JOptionPane.showInputDialog(
            "Enter number of years:");
10      int years = Integer.parseInt
            (yearsString);
11      int months = years * 12;

12      double futureValue =
          calculateFutureValue(monthlyPayment,
          months,monthlyInterestRate);

13       String message = "Monthly payment: " +
         monthlyPayment + "\n" +
         "Yearly interest rate: " +
         interestRate/100 + "\n" +
         "Number of years: " +  years + "\n" +
         "Future value: " + futureValue;

14       OptionPane.showMessageDialog(null,
           message, "Future Value", JOption
           Pane.PLAIN_MESSAGE);
15  }

16    private static double calculateFutureValue(
        double monthlyPayment, int months,
        double interestRate){
17      int i = 1;
18      double fValue = 0;
19      while (i <= months) {
20        fValue = (fValue + monthlyPayment) *
              (1 + interestRate);
22        i++;
23       }

24      return fValue;
25    }
26  }
```

Figure 6—Demonstrating Java Methods (FutureValueApp.java)

Running the program in Figure 6 results in the following output.



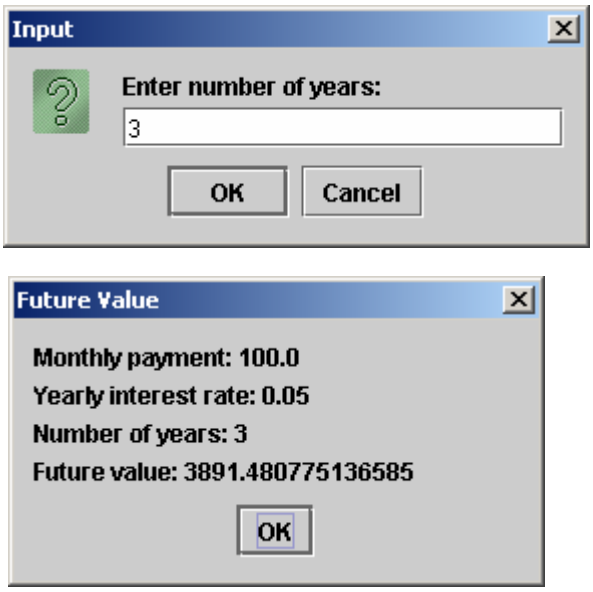


Figure 7—Output from FutureValueApp.java (see Figure 6 for code)

The first three windows are used to collect input from the user. The final output displays the parameters used in the calculation and the resulting future value. Note that the formatting of numbers (e.g., currency, percent, decimal places) has been omitted to keep the program simple. Special formatting and GUI design issues can be delayed until later in the course.

Line 1 in Figure 6 is used to import the Swing package to provide for GUI's. Lines 4-11 display the GUI's for user input and perform intermediate calculations to prepare for the future value calculation. Line 12 is critical. It serves two purposes: (1) it calls the calculateFutureValue() method with three arguments enclosed within the ()'s. After the method executes, the result is stored in the variable called futureValue. Lines 13-14 display the final output. Line 16 begins the code for the calculateFutureValue() method where the ()'s enclose the three parameters needed by the method. Then the method does its processing to calculate fValue (lines 17-23). Finally, the value stored in fValue is returned to the main() method (line 12) and stored in the variable futureValue.

Thus, Figure 6 illustrates how a Java program can be modularized to reduce complexity. The calculateFutureValue() method could be reused in many different financial programs if the programmer simply realizes that the three parameters (monthlyPayment, months, and interestRate) must be supplied to the method *in that order*. The code within the method really never needs changing. Of course, this kind of modularization is a cornerstone of traditional structured programming and is not unique to OOP. Remember, however, that in Java a method can exist in a separate class or file. If calculateFutureValue() were stored in a class named, for example, FinancialFormulas.java, the method call in line 12 of Figure 6 would read

```
double futureValue =
FinancialFormulas.calculateFutureValue
(monthlyPayment, months,
monthlyInterestRate);
```

Storing such reusable methods in separate Java classes is simply of way of becoming a more organized programmer.

5. A COMPLETE (BUT SIMPLE) TRADITIONAL APPLICATION

Armed with an adequate background in the Java syntax and an abbreviated understanding of how Java uses methods that are stored in classes, the student can begin to learn what is at the heart of OOSD and compare it to traditional structured systems development. The sample application will be a very simple one in order to convey the most essential points. The system under consideration will be that of a bookseller who sells books based on orders provided by customers. We will begin by creating a simple structured application using Java where the user can view all books that are available for ordering and then create an order. Remember that this application is extremely oversimplified to facilitate a direct comparison with OOSD.

Even when developing a traditional application, we must at some point think and talk about objects, whether we use the term or not. For a bookseller application, we are dealing ultimately with books, and books have certain characteristics, such as a code (id), a title, and a price. Books also have other characteristics that might be important for a shipping application, such as length, width, height, and weight. We could go on and on about the characteristics of books, but we are only concerned about those characteristics that are relevant to our book ordering system, so we will confine ourselves to code, title, and price.

It would make sense to store information about books in a file or database. For our purposes, a simple text file will suffice. Figure 8 shows a text file (comma delimited) with code, title, and price for four books.

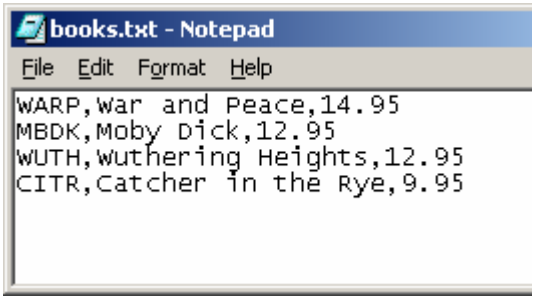


Figure 8—Text file with Book Code, Title, and Price (books.txt)

We will create an over-simplified structured application using Java that will present the user with five menu options: (1) display all book data, (2) add a book, (3) delete a book, (4) create a book order, and (5) quit. This kind of processing is quite basic and common. Figure 9 shows a simplified (no frills, no error routines, etc.) Java class that accomplishes this.

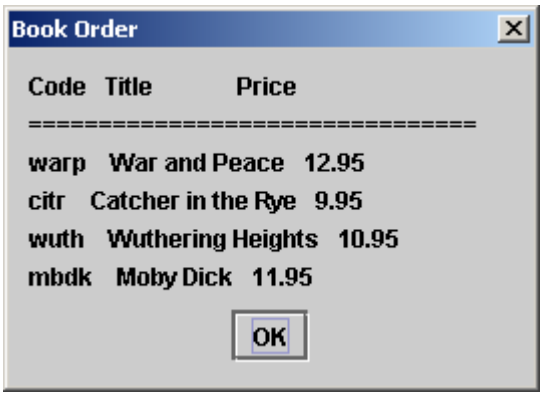
```
1  import javax.swing.*;
2  import java.io.*;
3  import java.util.*;

4  public class BookOrderApp{
5     public static void main(String[] args) throws
                  Exception {
6        String userInput = "";
7        int choice = 0;
8        String [][] books = readBookData();

9        while (choice != 5){
10          String menu =
11            "Enter                " + "\n" +
12            "     1 to display books  " + "\n" +
13            "     2 to add a book     " + "\n" +
14            "     3 to delete a book  " + "\n" +
15            "     4 to create an order" + "\n" +
16            "     5 to quit        ";
17          userInput =
          JOptionPane.showInputDialog(menu);
18          choice = Integer.parseInt(userInput);

19          switch(choice){
20            case 1:
21               displayBooks(books);
22               break;
23            case 2:
24               books = addABook(books);
25               break;
26            case 3:
27               books = deleteABook(books);
28               break;
29            case 4:
30               createBookOrder(books);
31               break;
32            case 5:
33              writeBookData(books);
34         } // end switch
35      } // end while
36      System.exit(0);
37   } // end main method


38   private static void displayBooks(String [][]
books) throws Exception {
39      String display =
40         "Code   Title            Price " + "\n" +
41   "==========================";
42      for(int i=0; i < books.length; i++){
43            String bookCode = books[i][0];
44            String bookTitle = books[i][1];
45            String priceString = books[i][2];
46            double bookPrice =
                 Double.parseDouble(priceString);
47            display += "\n" + bookCode + "   "
                + bookTitle + "   " + bookPrice;
48      }
49      JOptionPane.showMessageDialog(null,
           display, "Book Order",
50         JOptionPane.PLAIN_MESSAGE);
51   } // end displayBooks method


52   private static String [][] addABook(String
       [][] books) throws Exception {
53      String [][] newBooks = new String
                 [books.length+1][3];
54      System.arraycopy(books, 0, newBooks, 0,
                 books.length);

55      String code =
          JOptionPane.showInputDialog(
                 "Enter book code: ");
56      String title =
           JOptionPane.showInputDialog(
                 "Enter book title: ");
57      String price =
           JOptionPane.showInputDialog(
                 "Enter book price: ");

58      newBooks[books.length][0] = code;
59      newBooks[books.length][1] = title;
60      newBooks[books.length][2] = price;

61      return newBooks;
62   } // end addABook method


63   private static String [][] deleteABook(
        String [][] books) throws Exception {
64      String code =
           JOptionPane.showInputDialog(
    "Enter book code for book to be deleted: ");

65      String [][] newBooks = new String
                 [books.length-1][3];
66      int j = 0;
```

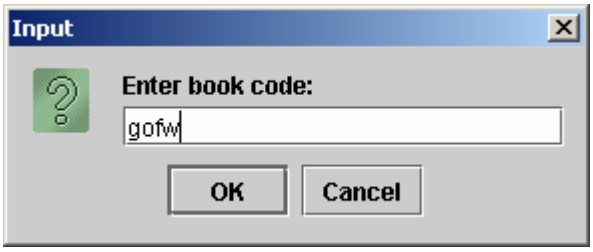
```
67      for(int i=0; i < books.length; i++){
68         String bookCode = books[i][0];

69       if(!(code.equalsIgnoreCase(bookCode))){
70            newBooks[j][0] = books[i][0];
71            newBooks[j][1] = books[i][1];
72            newBooks[j][2] = books[i][2];
73         }
74         else {
75            i++;
76            newBooks[j][0] = books[i][0];
77            newBooks[j][1] = books[i][1];
78            newBooks[j][2] = books[i][2];
79         }
80         j++;
81      }
82      return newBooks;
83   } // end deleteABook method


84   private static void createBookOrder(
        String [][] books) throws Exception {
85      String code = "", bookTitle = "Unknown",
                priceString = "0";
86      double bookPrice = 0, orderTotal = 0;
87      int orderQuantity = 0;

88      code =
           JOptionPane.showInputDialog(
                   "Enter book code: ");

89      for(int i=0; i < books.length; i++){
90         if(code.equalsIgnoreCase(books[i][0])){
91            bookTitle = books[i][1];
92            priceString = books[i][2];
93            bookPrice =
                 Double.parseDouble(priceString);
94            String orderQuantityString =
                    JOptionPane.showInputDialog(
                    "Enter order quantity: ");
95            orderQuantity =
                    Integer.parseInt(
                    orderQuantityString);
96            orderTotal =
                    orderQuantity * bookPrice;
97            break;
98         }
99      }

100     String orderOutput =
101        "Code: " + code + "\n" +
102        "Title: " + bookTitle + "\n" +
103        "Price: $" + priceString + "\n" +
104        "Quantity: " + orderQuantity + "\n" +
105        "Order Total: $" + orderTotal;

106     JOptionPane.showMessageDialog(
        null, orderOutput, "Book Order",
107     JOptionPane.PLAIN_MESSAGE);
108  } // end createBookOrder method


109  private static String [][] readBookData()
        throws Exception {
110     int numberOfRecords =
                0, numberOfFields = 3;
111     String fieldValue = "";

112     File bookData = new File("books.txt");
113     BufferedReader in = new
                BufferedReader(
                new FileReader(bookData));
114     String line = in.readLine();

115     while(line != null){
116        numberOfRecords++;
117        line = in.readLine();
118     }
119     in.close();

120     String [] [] books = new String
        [numberOfRecords] [numberOfFields];

121     bookData = new File("books.txt");
122     in = new BufferedReader(
                new FileReader(bookData));
123     line = in.readLine();

124     while(line != null){
125        for(int i=0; i <
                numberOfRecords; i++){
126           StringTokenizer t = new
                StringTokenizer(line, ",");
127           for(int j=0; j <
                numberOfFields; j++){
128              fieldValue = t.nextToken();
129              books[i][j] = fieldValue;
130           }
131           line = in.readLine();
132        }
133     }
134     in.close();
135     return books;
136  }// end readBookData method


137  private static void writeBookData(
        String [][] books) throws Exception {
138     File updatedBookData = new
                File("books.txt");
139     PrintWriter out = new PrintWriter(
                new BufferedWriter(
                new FileWriter(
                updatedBookData)));

140     for(int i=0; i < books.length; i++){
141        String bookCode = books[i][0];
142        String bookTitle = books[i][1];
143        String priceString = books[i][2];
144        String outputString = bookCode + ","
                + bookTitle + "," + priceString;
145        out.println(outputString);
146     }
147     out.close();
148  }// end writeBookData method
149 } // end class
```

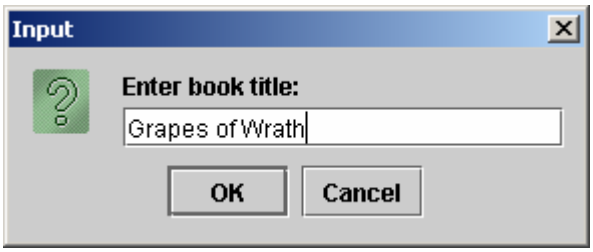
Figure 9—Structured Java Class for Book Orders (BookOrderApp.java)

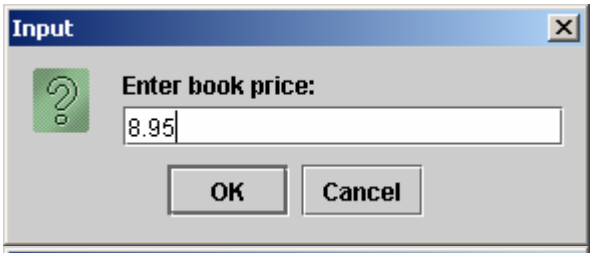
It is assumed here that the reader is fairly familiar with Java syntax, so an in-depth discussion of BookOrderApp.java will not be provided. Lines 1-3 import Java packages for using GUI's, file input/output, and various utilities (such as working with arrays and vectors), respectively. The "throws Exception" code (Line 5 and elsewhere) is required by Java to handle possible error conditions. The main method of BookOrderApp.java (Lines 5-37) essentially calls a method to read data from a text file into an array (Line 8). A menu is then presented (Line 17) to the user. Based on the user's input, various methods are called (Lines 19-34). The displayBooks method (Lines 38-51) accesses book data stored in the array named books and displays the available books to the user. The addABook method (Lines 52-62) allows the user to add a new book to the array. The deleteABook method (Lines 63-83) allows the user to delete an existing book from the array. The createBookOrder method (Lines 84-108) is used to gather input from the user to display book order information (title, price, order total). The writeBookData method (Lines 137-148) is called when the user exits the program (all additions and deletions of books are written to the text file books.txt).

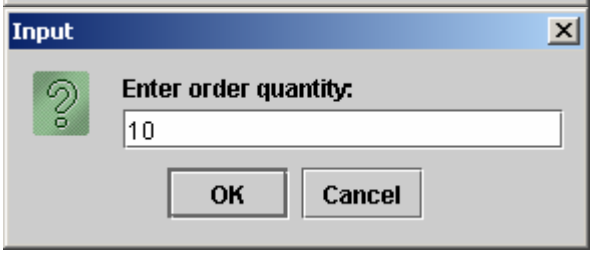
The class BookOrderApp.java is very traditional (non-OO) and structured. The text file books.txt stores the basic book data (code, title, and price), which is transferred to a two-dimensional array when the program runs. Adding, deleting, and processing data are the basic functions of this program. Figure 10 shows some of the windows that appear when various menu options are selected (note that special formatting is nonexistent to keep the code as simple as possible).

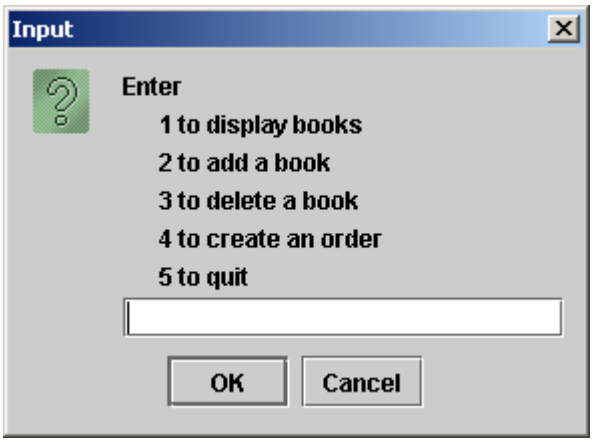












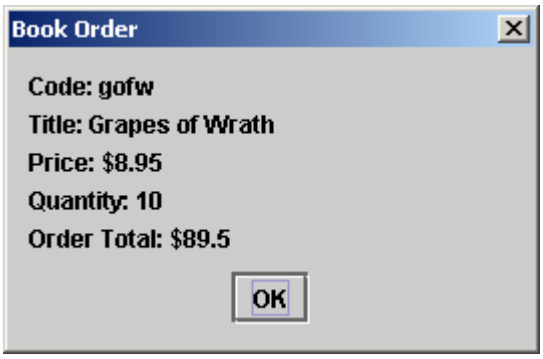


Figure 10—Various Windows from BookOrderApp.java

6. THE CORRESPONDING OO APPLICATION

Following is a completely identical application written using object-orientation. Again, this is an extremely simple application designed solely for highlighting the differences between OO and non-OO development and programming.

Although a formal analysis and design is not necessary with such a simple application, it should be noted that the central "object" of this system is the book. In this application, a book has three data attributes: code, title, and price. In Java, a class (file) must be created for the real-world book objects, which identifies book attributes and also contains a method to create books in the system as needed. Other methods for the book class may also be desired. In this case, the only other methods needed are three "get" methods designed to retrieve a book object's code, title, and price. The Java class for the book in this application is presented in Figure 11.

```
1   public class Book{
2     private String code;
3     private String title;
4     private double price;

5     public Book(String bookCode,
          String bookTitle, double bookPrice){
6       code = bookCode;
7       title = bookTitle;
8       price = bookPrice;
9     }

10    public String getCode(){
11    return code;
12    }

13    public String getTitle(){
14      return title;
15    }

16    public double getPrice(){
17      return price;
18    }
19  }
```

Figure 11—The Book Class (Book.java)

The class declaration is in Line 1—this class is public so that other classes can have access to it. Lines 2-4 identify the attributes (or instance variables) that all book objects will have. They are private variables meaning that other classes in the application will *not* have direct access to their values (access is only allowed through proper channels, as explained later). The book constructor method (Lines 5-9) will be called by another class in the application when it is necessary to create a new book object in the system. To create a book requires the input of code, title, and price. Lines 10-18 provide methods by which the values of book attributes may be accessed (read) when necessary (such as when creating a book order).

Following is the primary application file (Figure 12). This class corresponds directly to the traditional application file presented earlier in Figure 9. The reader should be able to directly compare each of the corresponding methods shown in Figures 9 and 12 (such as main(), displayBooks(), addABook(), etc.).

```
1  import javax.swing.*;
2  import java.io.*;
3  import java.util.*;

4  public class OOBookOrderApp{
5     public static void main(String[] args)
          throws Exception {
6       String userInput = "";
7       int choice = 0;
8       Vector books = readBookData();

9       while (choice != 5){
10        String menu =
11          "Enter                " + "\n" +
12          "     1 to display books  " + "\n" +
13          "     2 to add a book     " + "\n" +
14          "     3 to delete a book  " + "\n" +
15          "     4 to create an order" + "\n" +
16          "     5 to quit           ";
17        userInput =
        JOptionPane.showInputDialog(menu);
18        choice = Integer.parseInt(userInput);

19        switch(choice){
20          case 1:
21            displayBooks(books);
```

```
22              break;
23           case 2:
24              books = addABook(books);
25              break;
26           case 3:
27              books = deleteABook(books);
28              break;
29           case 4:
30              createBookOrder(books);
31              break;
32           case 5:
33             writeBookData(books);
34         } // end switch
35      } // end while
36      System.exit(0);
37   } // end main method


38   private static void displayBooks(
         Vector books) throws Exception {
39      String display =
40         "Code   Title           Price " + "\n" +
41      "=========================";

42      for(int i=0; i < books.size(); i++){
43       Book book = (Book) books.get(i);
44         display += "\n" + book.getCode() + "
           " + book.getTitle() + "   " +
           book.getPrice();
45      }
46      JOptionPane.showMessageDialog(
           null, display,  "Book Order",
47         JOptionPane.PLAIN_MESSAGE);
48   } // end displayBooks method


49   private static Vector addABook(
         Vector books) throws Exception {
50      String bookCode =
         JOptionPane.showInputDialog(
         "Enter book code: ");
51      String bookTitle =
         JOptionPane.showInputDialog(
         "Enter book title: ");
52      String bookPriceString =
         JOptionPane.showInputDialog(
         "Enter book price: ");
53      double bookPrice =
         Double.parseDouble(bookPriceString);

54      books.add(new Book(
         bookCode, bookTitle, bookPrice));
55      return books;
56   } // end addABook method

57   private static Vector deleteABook(
         Vector books) throws Exception {
58      String code =
         JOptionPane.showInputDialog(
         "Enter book code for book to be deleted:
");

59      for(int i=0; i < books.size(); i++){
60         Book book = (Book) books.get(i);
61         String bookCode = book.getCode();
62         if((code.equalsIgnoreCase(
         bookCode))){
63            books.remove(i);
64            break;
65         }
66      }
67      return books;
68   } // end deleteABook method


69   private static void createBookOrder(
         Vector books) throws Exception {
70      String code = "", bookTitle = "Unknown";
71      double bookPrice = 0, orderTotal = 0;
72      int orderQuantity = 0;

73      code =
         JOptionPane.showInputDialog(
         "Enter book code: ");

74
75      for(int i=0; i < books.size(); i++){
76         Book book = (Book) books.get(i);
77         if(code.equalsIgnoreCase(
         book.getCode())){
78            String orderQuantityString =
         JOptionPane.showInputDialog(
79             "Enter order quantity: ");
80            orderQuantity =
         Integer.parseInt(orderQuantityString);
81            bookTitle = book.getTitle();
82            bookPrice = book.getPrice();
83            orderTotal = orderQuantity
            *bookPrice;
84            }
85      }

86      String orderOutput =
87         "Code: " + code + "\n" +
88         "Title: " + bookTitle + "\n" +
89         "Price: $" + bookPrice + "\n" +
90         "Quantity: " + orderQuantity + "\n" +
91         "Order Total: $" + orderTotal;

92      JOptionPane.showMessageDialog(
         null, orderOutput, "Book Order",
93      JOptionPane.PLAIN_MESSAGE);
94   } // end createBookOrder method


95   private static Vector readBookData()
         throws Exception {
96      int numberOfRecords = 0,
             numberOfFields = 3;
97      String bookCode = "", bookTitle = "",
           bookPriceString = "";
98      double bookPrice = 0;
99      Vector books = new Vector();

100      File bookData = new File("books.txt");
101      BufferedReader in = new
         BufferedReader(
         new FileReader(bookData));
102      String line = in.readLine();

103      while(line != null){
104         numberOfRecords++;
105         line = in.readLine();
```

```
106     }
107     in.close();

108     bookData = new File("books.txt");
109     in = new BufferedReader(
          new FileReader(bookData));
110     line = in.readLine();

111     while(line != null){
112       for(int i=0; i <
              numberOfRecords; i++){
113         StringTokenizer t = new
              StringTokenizer(line, ",");
114         bookCode = t.nextToken();
115         bookTitle = t.nextToken();
116         bookPriceString = t.nextToken();
117         bookPrice =
              Double.parseDouble(
              bookPriceString);
118         Book book = new Book(
              bookCode, bookTitle, bookPrice);
119         books.add(book);
120         line = in.readLine();
121       }
122     }
123     in.close();

124     return books;
125   }// end readBookData method

126   private static void writeBookData(
        Vector books) throws Exception {
127     File updatedBookData = new
          File("books.txt");
128     PrintWriter out = new PrintWriter(
          new BufferedWriter(new
          FileWriter(updatedBookData)));

129     for(int i=0; i < books.size(); i++){
130       Book book = (Book) books.get(i);
131       String bookCode = book.getCode();
132       String bookTitle = book.getTitle();
133       double bookPrice = book.getPrice();
134       String priceString =
            Double.toString(bookPrice);
135       String outputString = bookCode + ","
            + bookTitle + "," + priceString;
136       out.println(outputString);
137     }
138     out.close();
139   }// end writeBooks method
140 } // end class
```

Figure 12—OO Java Class for Book Orders (OOBookOrderApp.java)

7. COMPARING TRADITIONAL AND OO APPLICATIONS

When one first compares the code in Figure 9 (traditional application) and Figure 12 (OO application), it appears that the OO application has nine fewer lines of code. Of course, the OO application won't work unless both its classes are present (Book.java and OOBookOrderApp.java), so there are actually a total of ten more lines of code in the complete OO application. However, the lines of code are less important than the actual complexity of each type of application.

Note first of all that the main methods of the traditional application and the OO application (Lines 5-37 in both) are virtually identical (with the only exception of Line 8 in each). The major differences in the two applications begin with the readBookData() methods of both applications (Lines 109-136 in Figure 9 and Lines 95-125 in Figure 12). In the traditional application, data from the text file are loaded in an array called books. In the OO application, a vector (basically an array of objects) called books is declared in Line 99 and data from the text file are used to create book objects (Line 118), one object for each record of data. In Line 119, the book object is added to the books vector. We will find that the handling of book objects stored in vectors is much more straightforward than the handling of book data stored in arrays, a major advantage of OOP.

Comparing the displayBooks() methods of each application (beginning with Line 38 in each), note that the traditional application uses basic array processing while the OO application retrieves each book from the vector of books and uses the get methods for each book object to retrieve the data and display it. Thus, there are actually fewer lines of code in the OO application. More importantly, if the data to be stored for books change (e.g., a field called numberOfPages is added), it is very likely that the OO code will be easier to modify (using objects and vectors) than the traditional code (using arrays).

A similar situation exists with the addABook() method (Line 52 in Figure 9, Line 49 in Figure 12). To add a new book to the vector in the OO application (Line 54) requires a simple command: books.add(). To do so with an array in the traditional application requires more complicated array processing. The differences are even more pronounced with the deleteABook() method (Line 63 in Figure 9, Line 57 in Figure 12) . The OO application uses a simple books.remove() command, but the traditional application requires fairly complex array manipulation. The createBookOrder() methods for both types of applications are very similar. Again, the OO application sim-

ply needs to call the get methods for each book object in the vector to display book data while the traditional application uses array processing. Finally, the writeBookData() methods are very similar with the OO application using book objects and the traditional application using arrays.

A major consideration here is that object processing is just inherently less complex than array processing. A major advantage of OOP is that many built-in Java methods exist to manipulate data stored in objects while such code doesn't exist (it must be written by the programmer) to manipulate data stored in arrays. Of course, the vector vs. array comparison is only the tip of the iceberg when it comes to evaluating the benefits of object-oriented analysis, design, and programming vis-à-vis traditional, structured analysis, design, and programming.

8. CONCLUSION

This paper has endeavored to present the basic OO features of the Java language and to briefly compare how a simple application would be created using traditional programming and OOP. The author is not aware of any papers or texts that directly compare OO with traditional applications as is done here. Even with the simple application used in this paper, it is apparent that handling data using OO can be much easier than handling data in a non-OO fashion (essentially storing data in objects rather than traditional arrays). The added simplicity of OO vs. traditional in the sample application is due primarily to the availability of built-in Java methods designed to manipulate objects in an easy, efficient manner. Organizing applications using objects simply reduces much of the programming overhead. It is also obvious from the example that the maintenance of OO applications can be easier. This was illustrated by considering the changes required in the sample application if just a single data field were added to a data record. While the differences in the examples presented here may seen somewhat trivial, many would argue that the OO approach becomes tremendously more advantageous as applications grow in size and complexity.

9. References

Booch, G. *Object-oriented analysis and design with applications*, 2nd ed., Redwood City, CA: Benjamin/Cummings, 1994.

Fayad, M., W. Tsai, and M. Fulghum, "Transition to object-oriented software development," *Communications of the ACM*, vol. 39, pp. 108-121, 1996.

Iivari, J., R. Hirschein, and H. Klein, "A dynamic framework for classifying information systems development methodologies and approaches," *Journal of Management Information Systems*, vol. 17, no.3, pp. 179-218, 2000-2001.

Johnson, R., "The ups and downs of object-oriented systems development," *Communications of the ACM*, vol. 43, pp. 69-73, 2000.

Steelman, A., *Murach's Beginning Java 2*, Fresno, CA: Mike Murach & Associates, 2002.

Pressman, R., *Software Engineering--A Practitioner's Approach*. 4th edition. New York, NY: McGraw-Hill, 1996.

Sheetz, S., G. Irwin, D. Tegarden, H. Nelson, and D. Monarchi, "Exploring the difficulties of learning object-oriented techniques," *Journal of Management Information Systems*, vol. 14, pp. 103-131, 1997.