Introduction to C#
Using .NET

Exercises and Case Study

Revision 1.0

Introduction to C# Using .NET
Exercises and Case Study
Rev. 1.0

These exercises are designed to accompany the book *Introduction to C# Using .NET* by Robert J. Oberg and published by Prentice Hall PTR, Upper Saddle River, NJ 07458.

ISBN 0-13-041801-3
www.phptr.com

Object Innovations, Inc.
4515 Emory Lane
Charlotte, NC 28211
877-558-7246
www.ObjectInnovations.com

Printed in the United States of America.

Table of Contents

Introduction to C# Using .NET
Exercises and Case Study

Introduction

Learning a programming language is much like learning any other skill. It requires lots of practice! These exercises are designed to give you practice with Microsoft's new programming language, C#. If you are an experienced programmer, you can probably skip most of the exercises in the early chapters and focus on the later chapters that discuss some features of C# that may be less familiar to you from earlier languages. If you are new to programming, do as many exercises as you can!

These exercises map to the chapters in the book *Introduction to C# Using .NET*. This write-up can be viewed as a reader's guide to the book. The focus is on the new programmer. If you already have another programming language under your belt, you can go directly to exercises in areas where you'd like a little reinforcement of your understanding. There is also a major case study.

C# as a First Programming Language

There have been many programming languages in vogue for learning programming. There is a tension between learning a language that will be "useful" in the real world, and one that is friendly to the newcomer. Happily, the popular languages for beginners are usually quite useful for practical applications. The first programming language I learned was FORTRAN (which is an acronym for FORmula TRANslation), a language designed originally for mathematical computations and widely used for a variety of other applications. My second language was BASIC (Beginners All-purpose Symbolic Instruction Code). I could not believe how easy BASIC was to learn! Bill Gates got his start writing a Basic interpreter for early personal computers. After dabbling in Algol I learned Pascal shortly after the language was released by its creator, Nicklaus Wirth. Like BASIC, Pascal was intended for learning programming. Unlike BASIC, it was designed with a view to teaching sound programming methodology, including disciplined use of data structures and control structures. After some experience with other languages, such as PL/I, I came to C, a very elegant language. After having generally experienced little difficulty in learning another language after my first, I found C to be a somewhat surprising hurdle. Maybe it was the quirky expressions one could write. Maybe it was the use of pointers. Whatever, it did present some hurdles, but once over them I enjoyed the language a lot.

While learning various languages I also became interested in programming methodology. "Structured programming," with the disciplined use of a limited number of control structures, seemed like a very natural concept, readily supported by languages such as Pascal, PL/I and C. Early FORTRAN did not have IF ... THEN ... ELSE or a WHILE loop, so you could not naturally write structured code in FORTRAN (you could write code in a structured way by using GOTO in a disciplined way, adhering to certain conventions, but this was somewhat artificial). There were a number of preprocessors

developed that would take “structured FORTRAN” and generate standard FORTRAN. In one of my programming classes, a team of students developed their own preprocessor called BIGFOR, which was actually used as the language of choice by one of my students in a later class! FORTRAN 77 added structured control to the standard language, and from then on most mainstream programming languages automatically supported structured programming.

Beyond disciplined control structures is structured handling of data. A key idea is “information hiding,” in which data would be encapsulated and only made accessible through functions. A useful program construct is an “informational-strength module” in which data is hidden, with access through only well-defined entry points. Informational-strength modules can be implemented in languages like C and C++.

I started to hear about “object-oriented programming,” and wondered how this went beyond programming concepts with which I was already familiar. I thought that I was already creating “objects” with encapsulated data and operations in the informational-strength modules that I was writing. I did not quite grasp the concept of “instantiation”, allowing many objects to be created from a given template or “class.” As early languages like FORTRAN did not directly support structured programming, languages such as C and PL/I did not directly support object-oriented programming. To cleanly implement classes you needed a new programming language construct. And the early history was similar to that of structured languages. Bjarne Stroustrup’s first implementation of an object-oriented language was to create a preprocessor that would transform code in a language called “C with Classes” to ordinary C. His new programming language quickly became C++, the first object-oriented language that I learned.

Learning C++ presented some hurdles for me, just like C had done earlier. By this time I was comfortable with the features of the language in common with C, but there were a surprising number of nuances in things like copy constructors, the use of **const** and the like.

As a teacher I have always been interested in how to teach programming languages, and in particular how to teach beginning programmers. I never considered C++ as a first language – it was just too complex.

Then came Java. By design, the language was simpler than C++ (for example, there is no multiple-inheritance). This time, I thought, surely the learning curve will be easier for me! I understood object-oriented programming, and the basic syntax of Java was like C, so it will surely be a snap. Not so! For one thing, even the programming model is a little different from conventional languages. You don’t run a compiler to create a machine-language program, which then gets executed. Instead, you generate “bytecode,” which gets executed by the Java Virtual Machine. “Running” the program involves invoking the JVM. The JVM may be invoked for you by a Web browser, which downloads and runs a Java “applet.” We’re not in Kansas anymore! But you can have a simpler environment, which is to write command-line programs, compile them to a Java class file, and then invoke the JVM yourself to execute the class file. Quite a bit is going on behind the

scenes, but operationally, at least, what you do is reasonably straightforward. So I was off and running to teach myself Java. I wrote "Hello, world", which was easy. Now for my second program, to echo back my own name. I stared and stared at the documentation and at introductory Java books and could not see how to do input from the command-line! It had to do with streams, and eventually I learned how to do it using several Java classes, but it was not easy. Bummer! Fortunately, once you understand the problem, the solution is reasonable—encapsulate the input operation through a class. A good friend of mine who knew Java well wrote an **InputWrapper** class for me, and I could then write an introduction to Java course, building up programming skills step-by-step, using object-oriented concepts quite early.

When Microsoft introduced C#, I was curious about how easy it would be to learn this new language. After my earlier experiences with C, C++ and Java, I wondered what the learning curve would be like. I was delighted at how easy the language was! For one thing, the conceptual foundation was a little cleaner. Just like a Java compiler generates bytecode, a C# compiler generates Intermediate Language or IL. But the packaging is a little bit easier. Instead of a class file that is interpreted by a JVM, IL is packaged inside an ordinary EXE file, which can be run at the command-line (provided the Common Language Runtime is installed on your machine). Next I wanted to write my little "echo" program, and, unlike Java, this was easy in C#. The same **Console** class that has the **WriteLine** method for output has an easy-to-use **ReadLine** method for input. So we are off and running with a simple testbed without any fuss. It turns out the rest of the language is quite regular. For example, "everything is an object," and simple data types can be treated like objects when needed, without resort to special wrapper classes.

My conclusion: C# is a wonderful first programming language. I hope you will enjoy it!

Case Study: The Electronic Commerce Game

There are many approaches to learning a programming language. You can read about it, and you can read code. To really learn it, you need to write programs—many programs. In the case of an object-oriented language, you should also gain some experience writing larger programs, because it is only in "programming in the large" that the benefit of the object-oriented approach really emerges. The book provides one case study, a banking system, that is developed incrementally in Chapters 12 – 18 with a componentized version in Chapter 21. These exercises provide a somewhat more elaborate case study, The Electronic Commerce Game. It is developed in ten steps, Chapters 12 – 19 and Chapters 21– 22. The final version is a Windows application, with the business logic encapsulated in a class library. Detailed instructions for developing the case study are provided in the chapter exercises, and an overview is presented in the appendix. Another good way to get an understanding of the game is to experiment with the final Windows program in Chapter 22. You can use the write-up in Chapter 22 as a player's guide.

Robert J. Oberg
Object Innovations

Chapter 1 – .NET Framework

The first chapter starts off with ".NET: What You Need to Know," which turns out not to be very much in order to compile and run your first program! The exercises for this chapter are essentially a few variations of the famous "Hello, world" program. We'll preview a few topics that you'll study in Chapter 2, and we invite you to begin looking at the .NET Framework online documentation.

Ex. 1.1 Type in and run the following program:

```
// Name.cs

class Name
{
   public static void Main()
   {
      string myName = "Bob";
      string greeting = "Hello, " + myName;
      System.Console.WriteLine(greeting);
   }
}
```

Note the use of the **string** variables **myName** and **greeting**. Note also the use of the + operator for concatenating (joining together) two strings. Modify the program to use your own name.

Ex 1.2 Create a program to display a 4x4 square using the asterisk symbol *. Thus your program should display:

```
****
*  *
*  *
****
```

Ex 1.3 Write a program to prompt the user to enter a name, read the name typed in at the keyboard, and print a greeting message welcoming the user by name. To do this you will need to do a little research on the **System.Console** class. Look up the online documentation for this class and see if you can figure out how to use the **Write** and **ReadLine** methods. You can base your solution on the **Name** program in Exercise 1.1.

Chapter 2 – First C# Programs

In this chapter you can begin writing C# programs in earnest. True, you don't know enough about C# yet to do anything complicated, but you can accomplish a surprising amount using very simple C#. As discussed in the book on page 28, you can use C# as a miniature calculator by writing little programs to do arithmetic. You can either hardcode the numbers used in the calculation into the program, or you can read the numbers in at runtime. For input, you can use the **InputWrapper** class discussed on pages 30 – 32. Using this class provides an example of creating an *object* using the **new** operator, our first taste of object-oriented programming! Using .NET Framework classes in your programs can be simplified somewhat by applying the **using** keyword to a namespace. Finally, you can control output more conveniently by using placeholders.

In these exercises you'll get practice with all these concepts, and we'll preview some other features of C#!

Ex. 2.1 Write a program to calculate the area of a triangle.
(a) Do this with hardcoded base of 15 inches and height of 5 inches.
(b) Write a general program that will prompt the user to enter a base and height and calculate the area of the corresponding triangle.

Ex. 2.2 Write a program to calculate the cube of a number entered by the user.
(a) Do this by multiplication.
(b) Do this by raising the number to the power 3. (Hint: look at the **Math** class for a method that you can use.)

Ex 2.3 Write a program to compute the volume of a sphere. The user should be prompted to enter the radius R, and your program should calculate the volume using the formula $V = \frac{4}{3}\pi R^3$. For pi you may use 3.1416. Or, try to find a more exact value using the **Math** class.

Ex. 2.4 Write a program to calculate the accumulation of $300 at 5% simple interest over a period of 8 years. (Answer: $420.00)

Ex. 2.5 Do the same problem, only using *compound interest*, where the interest is compounded annually. Your program should be general, allowing the user to enter the principal, interest rate, and number of years. You may use the compound interest formula $A = P(1+R)^N$, where P is the principal, R is the interest rate, N is the number of years, and A is the amount accumulated. (Answer for input data from **Ex. 2.4**: $443.24)

Ex. 2.6 Write a program to round an arbitrary **double** number to two decimal places. Can you think of two different ways to do this problem?

Chapter 3 – Visual Studio .NET

In this chapter you can begin using Visual Studio .NET to create projects, edit files, and run and debug programs. As practice you can create Visual Studio projects for all the previous exercises you have done and run them in Visual Studio. The supplied answers provide suitable project files. In this short problem set you will implement a new program, taking care to do everything in Visual Studio.

In an optional exercise you will preview creating Windows applications using Visual Studio. If you are familiar with Visual Basic, you should find creating Windows apps using Visual Studio .NET a snap. You may then wish to implement some Windows versions of exercises that follow, even though we won't officially come to Windows programming until Chapter 22!

Ex. 3.1 You own a small company for which you believe you can double the sales in the next year. Write a program to calculate by what rate your company will need to grow each quarter in order to achieve your goal.

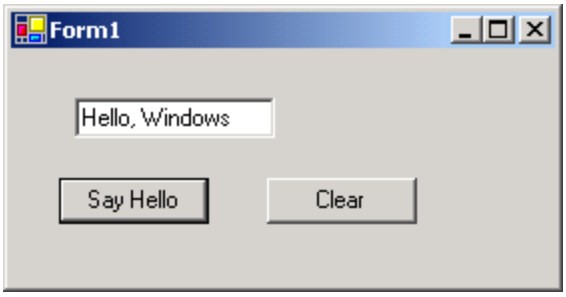
(a) Calculate the quarterly growth rate by solving the equation

$(1+r)^4 = 2$

(b) Check your work by multiplying by (1+r) four times.

(c) Single-step in the debugger and inspect the variable representing the current increase in sales each quarter. When you reach the fourth quarter, this value should be 2.0.

Ex. 3.2 Create a Windows application **SayHello** that has one textbox for displaying a greeting and two buttons. One button has the text Say Hello and will cause the greeting *Hello, Windows* to be displayed in the textbox. The second button has the text Clear and will cause the textbox to become blank.



(a) Create a C# project of type Windows Application with name **SayHello**.

(b) Use the Form Designer to drag one textbox and two buttons onto the form, as shown in the figure.

(c) Set the Name property of the textbox to **txtMessage** and of the buttons to **cmdSayHello** and **cmdClear**. Set the Text property of the textbox to blank, and of the buttons to the captions that are shown.

(d) Double-click each button to add a command handler in it.

(e) Add code to display an appropriate greeting, and to clear the greeting.

Chapter 4 – Simple Data Types

This chapter begins the systematic study of the C# programming language with a careful examination of the simple data types. Although this chapter does not immediately help you write more interesting programs, it contains fundamental information that you need to know as a C# programmer. Besides the textbook, it would be a good idea at this point to start becoming acquainted with the official reference to the C# language, the *C# Language Specification* published by ECMA International in December, 2002. The standard is available as a PDF file **Ecma-334.pdf** and can be downloaded from the Web site http://www.ecma-international.org.

Ex. 4.1 Download the ECMA C# specification and answer the following historical questions about C#.

(a) Who are the principal inventors of C#?
(b) What companies co-sponsored the submission of C# to ECMA along with Microsoft?
(c) When did Microsoft release the first widely available version of C#?
(d) Which of the following is *not* a design objective for C#?
    i. Be a simple, modern, general-purpose, object-oriented programming language.
    ii. Source code portability and programmer portability for programmers already knowing C and C++.
    iii. Runtime and size performance comparable to what can be obtained by C and C++.
    iv. Useful for developing software components suitable for deployment in distributed environments.
    v. Support for internationalization.

Ex. 4.2 Write a program to display the smallest and largest of each of the *signed* integer types, arranged from smallest to largest, each on a separate line.

Ex. 4.3 Study the section on literals in the ECMA *C# Language Specification* and answer the following questions:

(a) What are the possible boolean literal values?
(b) What data types can be stored in an integer literal?
(c) What data types can be stored in a real literal?
(d) Write a character literal to represent each of the following: slash, backslash, single quote, double quote.

Ex. 4.4 Write a program which will display, each on a separate line, a slash, backslash, single quote, and double quote. Show both the literal inside a single quote and not inside a single quote. The output should look like this:

```
'/' = /
'\' = \
''' = '
'"' = "
```

Ex 4.5 Write a program which will find the numerical representation of the following characters: ‘A’, ‘Z’, ‘a’, ‘z’. Also find out the character whose numeric representation is 91. For extra credit, display the numerical value in both decimal and hex.

Ex 4.6 Write a program to perform the following conversions:

(a) The value 3.14 to an integer.
(b) The number 1 to boolean.
(c) The number 2 to boolean.

Chapter 5 – Operators and Expressions

In this chapter we start working with the different operators provided by the C# language. As a member of the C family of languages, C# enjoys a very rich set of operators. Expressions can be built up by combining constants, variables and operators, and you can perform many useful calculations by evaluating expressions and printing out the results. These exercises should provide plenty of practice!

Ex. 5.1 Write a program to determine whether an integer read in at the console is divisible by four. Don't use any feature of C# that we have not discussed yet in the book! And don't even use the ternary **? :** operator! (This exercise is the beginning of an investigation of how to determine whether a year is a leap year—to be continued later.) Read in two numbers, so that you can test for both a number divisible by four and a number not divisible by four. Here is some sample output:

```
Number: 2003
Divisible by 4: False
Number: 2004
Divisible by 4: True
```

Ex. 5.2 Implement a different solution to the previous problem that will print a more elaborate message than "True" or "False" for the divisibility by 4. Here is some sample output:

```
Number: 2003
The number 2003 is not divisible by 4
Number: 2004
The number 2004 is divisible by 4
```

Ex. 5.3 Without using the computer, determine the values of the following expressions. Again write a program to check your results.

```
(a)     365 / 7
(b)     365 % 7
(c)     365 / 7.0
(d)     1E1 / (1E1 – 1E1)
(e)     1 / (1 – 1)
(f)     1m / (1m – 1m)
```

Ex. 5.4 Given the following assignments:

```
int x = 44;
int y = 45;
int z = 0;
```

determine the value of the following expressions, evaluated sequentially. Try to work this out with the computer. Then verify your work by writing a program.

```
(a)     x++ == --y
(b)     --x > y ? 5 : 10
(c)     z = (++x – y++)
(d)     z << 3
(e)     z < 3
(f)     z <<< 3
(g)     x
(h)     y
```

Ex. 5.5 Write a program to create a truth table for the bitwise AND operation.

Ex. 5.6 Write a program to assign a 32-bit integer and display the following 32-bit numbers:

(a) The leftmost byte of the number read in, with remaining bits 0.
(b) The rightmost byte of the number read in, with remaining bits 0.
(c) The middle two bytes of the number read in, with remaining bits 0.

Display both the input number and the results in. Also, display the three masks you use. For hex output, formatted to always show all 8 hex digits, you can use the following helper method:

```
private static void WriteHex(int x)
{
   Console.WriteLine("{0,8:X8}", x);
}
```

Ex. 5.7 Write a program to assign a 32-bit integer and swap the leftmost byte with the rightmost byte. The middle two bytes should be left as they are. Display both the original number and the swapped number in hex.

Ex. 5.8 The .NET Framework defines an enumeration type **ThreadState** to represent the possible states a thread can be in. This enumeration allows a bitwise combination of member values. Some combinations are valid and others are not. Without worrying right now about what a thread is, consider how to obtain a readable string representation of a numeric value representing a thread state. Here is a simplified table of member values of the enumeration.

0	ThreadState.Run
8	ThreadState.Unstarted
16	ThreadState.Stopped
32	ThreadState.WaitSleepJoin
64	ThreadState.Suspended
128	ThreadState.AbortRequested

Write a program which provides a loop for entering integers, –1 for end of file. For each number determine by masking which bit(s) are set and display the string(s) representing the state.

Chapter 6 – Control Structures

In this chapter we at last come to the control structures in C#. By using if tests and loops, you can implement programming logic that will enable you to solve many problems. In fact, the data types we have already discussed and the control structures covered in this chapter together are sufficient to solve almost any problem that can be implemented by a program in C#. Later concepts such as classes and methods will help us to write much better structured programs and enable us to build large programming systems. But the basic tools are already in place! There are more exercises than previously, because you can now do more interesting things!

Ex. 6.1 Write a program to calculate a worker's wage based on the number of hours worked in a week. Assume that the base pay rate is $12.00 an hour, with time and a half for overtime, which is defined as the number of hours worked beyond 40 hours.

Ex. 6.2 Write a program to determine whether a given year is a leap year or not. One solution to this problem was presented in the chapter, making use of a complex Boolean expression and a single if test. Implement a different solution using only simple Boolean expressions and multiple if tests.

Ex. 6.3 The previous solution is a little awkward to test, because only a single year is tested on any given run of the program. So to completely test the logic, you will need to run the program four times. Modify the program so that the processing will be done inside a loop. Use a year of 0 to terminate the loop. You should also maintain a count of the number of years processed.

Ex. 6.4 Write a program to evaluate π from the formula.

$$\frac{\pi^2}{6} = \sum_{1}^{\infty} \frac{1}{i^2}$$

Your program in a loop should read in an integer N and then calculate an approximate value of π by summing the series from 1 to N. Use a value of N = 0 to terminate the loop. Here is a sample run:

```
N: 100
Approximate value of pi is 3.1320765318
N: 1000
Approximate value of pi is 3.1406380562
N: 10000
Approximate value of pi is 3.1414971639
N: 100000
Approximate value of pi is 3.1415831043
N: 1000000
Approximate value of pi is 3.1415916987
N: 10000000
Approximate value of pi is 3.1415925581
N: 0
Math.PI = 3.1415926536
Difference = 0.0000000955
```

Ex. 6.5 Write a program to create a truth table for the bitwise AND operation. Instead of the brute force method used in Chapter 5, implement a solution using two loops, each of which goes from 0 to 1.

Ex. 6.6 Write a program to assign a 32-bit integer and swap the leftmost byte with the rightmost byte. The middle two bytes should be left as they are. Display both the original number and the swapped number in hex. The following code will display a 32-bit number in hex with all 8 hex digits shown:

```
Console.WriteLine("{0:X8}", x);
```

Ex. 6.7 Write a program to display the accumulation in an IRA account year by year. The program will accept the inputs:

- Annual Deposit = A
- Interest Rate = R
- Number of Years = N

Assume that a deposit is made at the end of each year and that interest is compounded annually. Test with values of A = $2500.00, R = 0.05, N = 10. As a check of your work, compare the total accumulation at the end with the amount calculated from the formula provided in Chapter 5.

Ex. 6.8 A number greater than 1 is prime if it has no factors other than 1 and itself. Write an interactive program to read a series of numbers and test each number for being prime.

Ex. 6.9 Write a program to determine all prime numbers less than 1000. Display the results neatly in rows of 10 with the numbers lined up in columns. Also, display a count of how many numbers less than 1000 are prime.

Ex. 6.10 Write an interactive program containing a command loop that lets the user enter simple commands, which will be carried out. The commands are:

a) "add" – prompt for two numbers, which will be added and the sum displayed
b) "subtract" – prompt for two numbers, which will be subtracted and the difference displayed
c) "quit" – exit the program

Any other command should cause a simple help message to be displayed that lists the legal commands. Make use of the **InputWrapper** class to simplify input. You can store a command in a **string** variable, and you can test for equality and inequality of strings using the == and **!=** operators. Here is a sample run of the program:

```
Enter command, 'quit' to exit
> help
legal commands are:
        add
        subtract
        quit
> add
first number: 5
second number: 7
sum = 12
> quit
```

Chapter 8 – Classes

We have been using classes in C# from the very beginning. Indeed, *every* C# program has at least one class. However, it is only in this chapter that we have studied classes systematically. Classes are the foundation for object-oriented programming in C#. They are also the foundation for components, which we will study later. From now on, classes will be very integral to all the C# programs we write.

It has been suggested that the traditional kind of "Hello World" program sets a poor example of how programs should be constructed. See Ralph Westfall, "Hello, World Considered Harmful", *Communications of the ACM*, October, 2001, p. 129. The basic argument is that the traditional program starts students thinking in terms of monolithic programs, a habit that is hard to shake. Indeed, a few of our exercises in Chapter 6 are a bit monolithic and might be clearer if they were broken down into smaller program units. From now on we will aim to structure programs by building them from smaller units.

These exercises will give you practice in working with classes in C#. The last exercise introduce the first of several classes that comprise The Electronic Commerce Game case study. This case study will be developed incrementally, beginning in earnest in Chapter 12.

Ex. 8.1 Implement an object-oriented version of a basic "Hello World" program. Your program should include a class that encapsulates a greeting string and provides the following features:

a) A default constructor, for which greeting will be "Hello, world."
b) A constructor taking a string parameter used to specify the greeting.
c) A method **SetGreeting()** which can be used to specify the greeting for an object after it has been constructed.
d) A method **DisplayGreeting()** which can be used to display the current greeting at the terminal.

Your program should also include a test program in a separate file that constructs several greeting objects, specifying various greetings, which are displayed.

Ex. 8.2 Remember the exercise from Chapter 1 in which you wrote out a square using the asterisk (*) character? Well, now you will create a **Square** class with various fields and methods, including displaying the square graphically using asterisks. To keep life simple, in this problem you can show the square solid. Your **Square** class should provide the following features:

a) Private fields specifying the size and origin of the square as integers. The origin (always zero or greater) specifies how many blank characters to display before showing the square.
b) A constructor to initialize the size of a new square, with the origin set to 0.
c) A method **Show()** to display the square.
d) Methods **SetSize()** and **GetSize()** to set and get the size of the square.

e) A method **Move()** to move the origin by a specified displacement. A positive displacement moves the origin to the right, and a negative displacement moves the origin to the left.
f) A method **GetOrigin()** to return the current origin.

Implement a test program that will instantiate a **Square** and perform various operations on the square. The square should be shown so that you can verify the results of the various operations.

Ex. 8.3 Enhance your **Square** class to add a private field specifying whether the square should be displayed solid or in outline form. Provide a public method to set this field, and modify the **Show()** method to display the square appropriately. You may find the code to be somewhat cleaner if you use helper methods **ShowSolid()** and **ShowOutline()**. Enhance your test program to exercise the new functionality. Be sure to test boundary conditions, such as a square of size 0 or 1.

Ex. 8.4 Implement a **Triangle** class with functionality similar to your **Square** class. You may limit yourself to solid triangles, but you should be able to specify the size and origin. A triangle should be displayed like this:

```
   *
  * *
 * * *
* * * *
size = 4, origin = 0
```

Ex. 8.5 Implement a **ProductItem** class to represent a product item. Your class should implement the following features:

(a) Private fields to hold an integer id, a string description, and a decimal price.
(b) A private static member that can be used to auto-assign ids starting at 1.
(c) A constructor to initialize a new item given a description and a price.
(d) Methods to get the id, the description, and the price.
(e) A method to set the price.
(f) A method to discount an item by a given percentage amount. Note that a discount must be between 0 and 100%. Display an error message if discount is outside this range.
(g) A method to return a string that shows the fields of an item.

Implement a test program that will instantiate two items. Show the first item by using the class method for showing an item. Show the second item that will use a private helper method that displays item data by calling the various get methods. Assign a new price to the first item and show it. Apply a discount to the second item and show it. Also try applying illegal discounts that are negative and greater than 100%.

Ex. 8.6 Implement a **Player** class to represent a player in a game. Your class should implement the following features:

(a) Public fields to hold a string name, a decimal balance, and a boolean flag indicating whether the player is active.
(b) A private field to hold a string password.

(c) A constructor to initialize a new player given a name, balance and password. The player should be initialized as inactive.
(d) A method to return a string that shows the public fields of a player separated by tabs.
(e) A method to check a string for being a valid password, returning a **bool**.

Implement a test program that will do the following:

(a) Instantiate a player object and show it
(b) Change the balance and active flag and show it again
(c) Give a user three chances to enter a correct password. If the user succeeds, the player should be welcomed by name, other an error message should be displayed.

Chapter 9 – The C# Type System

There are many different data types in C#. A fundamental distinction is between reference types and value types. In Chapter 8 we studied the **class** type, which is a reference types. Among the types studied in this chapter is **struct**, which is a value type. The simple data types studied in Chapter 4 are value types. Enumerations are another kind of value type. We will see a number of different reference types in the chapters ahead, including the built-in **object** and **string** types.

Ex. 9.1 In this exercise you will study the differences between classes and structs. Implement a class **NameC** which has the following features:

a) A public string field **Name**.
b) A default constructor which simply displays a message that a class object has been created.
c) A constructor taking a string parameter which initializes the **Name** field to this string and displays a message that a class object with specified name has been created.

Implement a struct **NameS** which replicates as much of this functionality as possible. Run your class and struct against the following test program. Make sure that you understand every nuance of the output.

```
// TestName.cs

using System;

public class TestName
{
   public static void Main()
   {
      // class
      NameC c1 = null;
      ShowNameC(c1);
      c1 = new NameC();
      ShowNameC(c1);
      NameC c2 = new NameC("Amy");
      ShowNameC(c2);
      c1 = c2;
      ShowNameC(c1);
      c1.Name = "Bob";
      ShowNameC(c1);
      ShowNameC(c2);

      // struct
      NameS s1;
      s1.Name = "";
      ShowNameS(s1);
      NameS s2 = new NameS("Carol");
      ShowNameS(s2);
      s1 = s2;
      ShowNameS(s1);
      s1.Name = "David";
      ShowNameS(s1);
```

```
      ShowNameS(s2);
   }
   private static void ShowNameC(NameC n)
   {
      if (n == null)
         Console.WriteLine("Name is null");
      else
         Console.WriteLine("name = {0}", n.Name);
   }
   private static void ShowNameS(NameS n)
   {
      Console.WriteLine("name = {0}", n.Name);
   }
}
```

Output:

```
Name is null
Class object created
name =
Class object Amy created
name = Amy
name = Amy
name = Bob
name = Bob
name =
Struct object Carol created
name = Carol
name = Carol
name = David
name = Carol
```

Ex. 9.2 Write an interactive program that will allow you to test the four different types supported by the **InputWrapper** class from Chapter 2. Your program should prompt for a data type ("quit" to exit the program) and then prompt for a number of that type and display the number read in.

Ex. 9.3 Create a new version of the **InputWrapper** class that will enable you to input signed or unsigned integers of all types of 32 bits or less. Write an interactive test program that will prompt for the data type and then prompt for a number of that type and display the number read in.

Ex. 9.4 Write a program that will exercise an enumeration type **Day** defined as follows:

```
public enum Day : byte
{
      SUN, MON, TUE, WED, THU, FRI, SAT
}
```

Your program should prompt for an integer between 0 and 6 and convert the integer to a value of type **Day**. Then, depending on the value, display a message showing the corresponding day of the week. An input of 99 is used for end-of-file.

Ex. 9.5 Enhance your **Player** class to keep track of a player's role through a public data member **RoleId**. This id should be of an enumeration type **PlayerRole** that has two values, Shopper = 1 and Vendor = 2. Your constructor should add a parameter to initialize the role id. The **Show** method should display a string representation of the role id. The test program should simply prompt for values of a new player object, create this object and then display it. Use an integer for inputting a role id and do the necessary conversion in passing this id to the constructor.

Chapter 10 – Methods, Properties and Operators

This chapter examines methods in detail. We study instance methods and static methods, parameter passing, method overloading, and variable length parameter lists. C# has a special *property* syntax that can be used to simplify the notation for get/set methods. Finally, C# allows you to overload certain operators, providing a notation that is sometimes a little more concise than method notation. These problems give you practice with these various concepts.

Ex. 10.1 Implement a struct **Vector** that is an ordered set of three integers. A constructor creates a vector given three integers. Define methods to implement vector addition, scalar multiplication, and dot product. Use method names **Sum** and **Product**. Note that **Product** is overloaded. Also implement a **Show** method that takes a string parameter that can be used as a label. Create a suitable test program to exercise your class. Here is some sample output from the test program:

```
a = (1, 2, 3)
b = (10, 20, 30)
sum = (11, 22, 33)
scalar product of a and 5 = (5, 10, 15)
dot product of a and b = 140
```

Ex. 10.2 Redo the previous exercise so that the **Sum** and the two **Product** methods are all static. Thus these methods will now take *two* parameters, and when you invoke a method you will qualify it by **Vector** rather than by an object instance.

Ex. 10.3 Now redo the exercise again, this time overloading the + operator for **Sum** and the * operator for **Product**.

Ex. 10.4 Write a program to compare the behavior of a vector implemented as a struct versus an implementation as a class. In each case the three integer data members will be public (so you can change them). A constructor creates a vector given three integers. The only method is **Show**. The struct version will be called **Vector** and the class version **CVector**. Write a test program that has the following features:

a) A method **Copy** that takes a **Vector** parameter and returns this same vector.
b) An overloaded method **Copy** that takes a **CVector** parameter and returns this same vector.
c) The **Main** method first creates and shows a **Vector** and a **CVector**.
d) It then copies the **Vector** and modifies the copy. Both the original and modified vectors are shown.
e) It then copies the **CVector** and modifies the copy. Both the original and modified vectors are shown.

Build and run. Make sure you understand clearly the reason for the different behavior of the struct and class versions.

Ex. 10.5 We have been working with three-dimensional vectors. Now let's create an N-dimensional vector of integers using an array. Please review the preview material on

arrays in Chapter 6, and also the discussion of variable length parameter lists in Chapter 10. Implement a class **Vector** in which a vector will be represented by an array of integers. Your class should have the following public members:

a) A constructor with a variable number of integer parameters .
b) A read-only property **Size** that returns the dimension of the vector.
c) A method **Product** that returns the scalar product of a vector and an integer scalar.
d) A method **Show** that displays to the console a vector identified by a string label.

Create a test program to create and show vectors of various dimensions and also the scalar product of a vector and an integer.

Ex. 10.6 Write a program to keep track of two player instances of the **Player** class that you implemented in Chapter 8 (Exercise 6). Your class **Players** should implement the following features:

a) A constructor to initialize two object instances.
b) A method **ShowPlayers** that will display the two players.
c) A method **FindPlayer** that will return a **Player** given a name, returning **null** if the name does not match either player.
d) A method that will login in a player given name and password. The method has an **out** parameter to pass back a reference to the **Player** object that has been logged in (**null** if login not successful). A string is returned that is set to "OK" in case of success and a descriptive error message otherwise. To succeed, the player must be found and not already active, and the password must match. As part of the login, the player is made active.
e) A method to logout a player given the name. Again a string is returned to indicate success or failure. Logout makes the player inactive.

Implement a suitable test program to exercise your **Players** class and the **Player** class.

Chapter 11 – Characters and Strings

String data is ubiquitous in programming. Strings are made up of individual characters. This chapter examines characters and strings in detail and looks at a few typical uses of strings in programs. A pattern for an interactive test program is introduced, consisting of a simple command processing loop and a "help" message that is displayed whenever an illegal command is introduced. The first two exercises involve implementing such interactive test programs for previously developed classes. The third exercise presents a slight refinement to this pattern, making the input command case insensitive. The remaining exercises give you further practice in working with strings.

Ex. 11.1 Implement an interactive test program for the **Players** class from Chapter 10, Exercise 6. Implement the following commands:

```
Enter command, quit to exit
: help
The following commands are available:
        list    -- list players
        find    -- find a player
        login   -- login a player
        logout  -- logout a player
        change  -- change balance
        quit    -- exit the program
:
```

Ex. 11.2 Implement an interactive test program for the shape classes from Chapter 8: **Square** (Exercise 2) and **Triangle** (Exercise 4). Your program should maintain both a **Square** object and a **Triangle** object, which can be manipulated by various commands:

```
Shape is square
> help
legal commands are:
        show     -- show the shape
        size     -- change the size
        move     -- move the shape (+ for right, - for left)
        mark     -- change the mark character
        triangle -- set shape to triangle
        square   -- set shape to square
        quit     -- exit the program
>
```

Ex. 11.3 Implement a case-insensitive version of the **StringDemo** example program from this chapter. The behavior of the program should be the same, only the user may enter commands in any case (all lower, all upper, or mixture of upper and lower).

Ex. 11.4 Write a program to read a series of strings from the console (empty string for end-of-file). For each string display a count of the number of vowels, the number of letters (alphabetic characters), and the total number of characters in the string.

Ex. 11.5 Write a program to read a series of strings from the console (empty string for end-of-file. For each string display the individual words in the string. A word is designed as a contiguous sequence of letters, delimited by non-letters.

Chapter 12 – Arrays and Indexers

Arrays are a very useful data type in computer programming. We previewed arrays in Chapter 6 and have made occasional use of simple arrays since then. This chapter discussed arrays in detail, and we will use arrays extensively from now on. With the help of arrays we can now create more interesting programs.

For the next several chapters our problems are going to be mainly focused on a case study, The Electronic Commerce Game, which you will build by incremental steps. Please read the introduction to this case study in the Appendix before beginning work on the exercises in this chapter.

Ex. 12.1 The most important class in the game is **Player**, which we implemented in Chapter 8, Exercise 6. Our game can have multiple players, so the very first step will be to create a new class **PlayerList** that can store a list of players. We will represent this list by an array (and later use other data representations, such as a .NET collection). To get started, implement the class with these features:

a) A constant MAXPLAYER to specify the size of the array. You can use a small value to make it easy to test for array index out of bounds.
b) A static data member to keep track of the next index at which to add an array element.
c) An array of type **Player[]** to hold the list of players.
d) A read-only property **Count** that will return the number of players.
e) A method **AddPlayer** that will create a new player given name, password and starting balance and add it to the list. This method should return the string "OK" if the operation was successful and a string with a descriptive error message if there was an error.
f) A method **Display** that will display a list of all the players at the console.

Implement a simple driver test program to exercise your class. To save typing while you test your program, you may wish to initialize **PlayerList** with some starting test data in your code.

```
Enter command, quit to exit
player> help
The following commands are available:
        add     -- add a new player
        list    -- list player information
        quit    -- exit the driver
player>
```

Ex. 12.2 The first version of our **PlayerList** class is rather primitive. A list should usually have a unique key, which in our case will be the player name. Add code to your class to ensure that player names are unique. If an attempt is made to add a player with the same name as a player already on the list, an error message should be returned.

Ex. 12.3 There is a fundamental defect in the first implementation of our **PlayerList** class—we do output in the class. Normally, that is not a good idea, because it makes the code too specific, tying it to a specific form of user interface. A class such as **PlayerList**

that is concerned with data should normally not do user interface. If we later do a different kind of user interface, for example a Windows graphical user interface, a data class should not have to be changed. Note that in the **Player** class we observe this principle; the **Show** method returns a string representation of a player rather than performs console output. Modify the **PlayerList** class so that it does not do output. You will also need to modify your driver test program.

Ex. 12.4 Now that we have a good, clean implementation of the basic functionality of **PlayerList**, we can start to add features. Add the following methods. Unless otherwise specified, the method should return a string that is "OK" if successful and otherwise is a descriptive error message.

a) A method **DeletePlayer** that will delete a player given the name.
b) A method **FindPlayer** that returns the corresponding **Player** object if the name is found and **null** otherwise.
c) A method **GetBalance** that will find the balance, returned as an **out** parameter, given a player name.
d) A method **SetBalance** that will set a new balance for a player.
e) A method **ChangeBalance** that will adjust the balance of a player by a specified amount (increasing the balance if the delta is positive, otherwise decreasing the balance).

Add code to the test driver to exercise these new features.

Ex. 12.5 Add login and logout methods:

a) A method **Login** that will login in a player given name and password. The method has an **out** parameter to pass back a reference to the **Player** object that has been logged in (**null** if login not successful). A string is returned that is set to "OK" in case of success and a descriptive error message otherwise. To succeed, the player must be found and not already active, and the password must match. As part of the login, the player is made active.
b) A method **Logout** to logout a player given the name. Again a string is returned to indicate success or failure. Logout makes the player inactive.

Add code to the test driver to exercise these new features.

Ex. 12.6 Implement a class **Item** that will keep track of items on a shopping list. An item consists of the name of a product and the quantity to be purchased. Provide a constructor and a suitable **Show** method. Implement a class **ItemList** with the features shown below. As usual, each method should return status in a string, unless otherwise specified.

a) A constant MAXITEM to specify the size of the array. You can use a small value to make it easy to test for array index out of bounds.
b) A static data member to keep track of the next index at which to add an array element.
c) An array of type **Item[]** to hold the list of items.
d) A read-only property **Count** that will return the number of players.
e) A method **AddItem** that will create a new item given name and quantity. Item names should be unique, so return an error on an attempt to add a duplicate name.
f) A method **GetItems** that will return an array consisting of all the items on the list.

g) A method **DeleteItem** that will delete an item given the name.
h) A method **FindItem** that returns the corresponding **Item** object if the name is found and **null** otherwise.
i) A method **ChangeQuantity** that changes the quantity of an item given the name and a delta.

Implement a suitable test driver program.

```
Enter command, quit to exit
item> help
The following commands are available:
        list    -- list items
        add     -- add an item
        delete  -- delete an item
        find    -- find an item
        change  -- change quantity
        quit    -- exit the driver
item>
```

Ex. 12.7 Implement a master driver program that will have commands that will bring up either the player driver program or the item driver program.

```
Enter command, quit to exit
: help
The following commands are available:
        player  -- player driver
        item    -- item driver
        quit    -- exit the program
:
```

Build and exercise the program thoroughly. You are now at Step 1 of The Electronic Commerce Game.

Chapter 13 – Inheritance

Inheritance is a key concept of object-oriented programming. It can be an effective technique for reusing code. The exercises of this chapter continue the implementation of The Electronic Commerce Game.

Ex. 13.1 Another important class for the commerce game is **Product**. A product has a name, a quantity, and a price. Implement this class by inheriting from **Item**. Test your class with a simple non-interactive test program that creates a product, displays it, changes some data, and shows it again.

Ex. 13.2 The next job is to create a class **ProductList** that can store a list of products. Again, you can use inheritance, this time inheriting from **ItemList**. Your class should have the following features:

a) A public field **VendorName** that is the name of the vendor that has this list of products.
b) A constructor that takes a string parameter that initializes **VendorName**.
c) A read-only property **Count** that will return the number of products.
d) A method **AddProduct** that will create a new product given name, quantity and price and add it to the list.
e) A method **FindProduct** that returns the corresponding **Product** object if the name is found and **null** otherwise.
f) A method **DeleteProduct** that will delete a product given its name.
g) A method **ChangeQuantity** that changes the quantity of a product given the name and a delta.
h) A method **ChangePrice** that changes the price of a product given the name and a delta.
i) A method **GetItems** that will return an array consisting of all the products on the list.

Implement a suitable test driver program.

```
Enter command, quit to exit
product> help
The following commands are available:
        list    -- list products
        add     -- add a product
        delete  -- delete a product
        find    -- find a product
        changeq -- change quantity
        changep -- change price
        quit    -- exit the driver
product>
```

Ex. 13.3 There are both wholesale and retail vendors in the commerce game. The first vendor we will work with is a wholesaler that maintains the master list of all products that are available in the game. Implement a class **Wholesaler** with the following features:

a) A private product list.
b) A constructor that initializes the product list with the vendor name "Wholesaler."

c) A method **Sell** that takes a product name and quantity as input parameters and returns a price as an output parameter. As usual, status is returned in a string. If the product is not found on the list, or the quantity available is insufficient, an error is returned. The available quantity is decremented, and the price of the item is passed back.
d) A property **Products** that gives access to the **ProductList** that is managed by the wholesaler.

Implement a simple non-interactive test program that initializes the wholesaler with some test data, shows the wholesaler's vendor name and product list, sells an item, and then shows the wholesaler's data again.

Ex. 13.4 Implement a game driver program that integrates everything you have done so far. The top-level driver will have two commands:

```
Enter command, quit to exit
: help
The following commands are available:
        player  -- player driver
        product -- product driver
        quit    -- exit the program
:
```

The "player" command is identical to what we implemented in Chapter 12. The "product" command is a replacement for the previous "item" command. A new **Game** class is introduced that has the following features:

a) A public static member to hold the wholesaler.
b) A read-only property of type **ProductList** that gives access to the product list maintained by the wholesaler.
c) A static constructor that initializes the wholesaler's product list.

The product driver is modified to work with the wholesaler's product list maintained in the **Game** class. Also, add a "sell" command. Build and exercise the program thoroughly. You are now at Step 2 of The Electronic Commerce Game.

Chapter 14 – Virtual Methods and Polymorphism

This chapter continues the story of inheritance with a discussion of virtual methods and polymorphism. We begin with some small exercises illustrating a polymorphic collection of shapes. The last of these shape exercises depends on a topic not mentioned in the chapter in the book but which is occasionally of importance—the **GetType()** method of **object** and the **typeof** operator. You may wish to consult the MSDN documentation. We then resume the implementation of The Electronic Commerce Game.

Ex. 14.1 Implement a tiny shape hierarchy consisting of squares and triangles. Create an abstract base class and two derived classes. There should be a single method **Display** in each class. The concrete classes should implement the method by displaying the string "Square" or "Rectangle". Write a simple test program that will create different shape objects and store them in an array. Show the shapes by iterating through this array.

Ex. 14.2 Elaborate your shape classes by storing the dimensions of the shapes. For a square you need only to store a single size. For a rectangle you need to store two sizes, the base and height. The **Display** method should display the size(s) along with the string description of the shape.

Ex. 14.3 Create a class **ShapeList** to hold a list of shapes in an array. Your class should support the following features:

a) A constant MAXSHAPE to specify the size of the array. You can use a small value to make it easy to test for array index out of bounds.
b) A static data member to keep track of the next index at which to add an array element.
c) An array of type **Shape[]** to hold the list of shapes.
d) A read-only property **Count** that will return the number of shapes.
e) A method **AddShape** that will add a new **Shape** object to the list. This method should return the string "OK" if the operation was successful and a string with a descriptive error message if there was an error.
f) A method **DisplayShapes** that will display a list of all the shapes. Each shape should be shown on a separate line along with the index showing position of the shape in the array.

Implement a simple driver test program to exercise your class. To save typing while you test your program, you may wish to initialize **ShapeList** with some starting test data in your code. In the "add" command, prompt for the type of shape (1 = Square, 2 = Rectangle).

```
> help
The following commands are available:
        display -- display shapes
        add     -- add a shape
        quit    -- exit the driver
>
```

Ex. 14.4 Add a method **ChangeSize** that will change the size(s) of a shape at a given index. The input parameters should be an index and a **Shape** object reference. Note that

this method is not allowed to change the *type* of shape, only its size. Thus a square can be changed into a square of a different size, and a rectangle into a rectangle with different sizes. You need to do several things to implement and test this functionality.

a) Implement the **ChangeSize** method. Include a check comparing the data type of the shape object reference being passed in with the data type of the existing shape object at the specified index. Use the **GetType** method that is one of the built-in methods of the **object** root class.
b) Add a method **GetShape** to the **ShapeList** class that will retrieve the shape object at a specified index. It should return **null** if the index is out of range.
c) Add a command to the driver program to test the **ChangeSize** method. Prompt for an index and retrieve the existing shape at that index. Depending on the shape type, prompt for either a single size (square) or else for a base and a height (rectangle). Invoke the **ChangeSize** method. You will need to use both the **GetType()** method and the **typeof** operator. You can read the MSDN documentation for a description of **GetType()** and **typeof**.

The next exercises continue the implementation of The Electronic Commerce Game, culminating in a full working version. What we are adding are two kinds of players, Vendor and Shopper. The program will exploit polymorphism, in a manner similar to its usage in the shape exercises you just completed.

Ex. 14.5 Create a class **Vendor** derived from **Player**. It will have a new field **Url**, a constructor, and overrides of the **Prompt** and **Show** methods. The prompt for a vendor is "V> ". The **Show** method shows the url along with the string from the base class. Several other changes will also be needed to the overall master driver program (pick up from Exercise 4 of Chapter 13).

a) The methods **Prompt** and **Show** in the base class must now be virtual. In the **Show** method include the prompt string.
b) Add an **enum** called **PlayerType** that can specify either Shopper (= 1) or Vendor (= 2).
c) Move the **players** array from the player driver to the **Game** class as a public static data member. Move the initialization code to the **Game** class also.
d) Implement an overloaded **AddPlayer** method in **Player** that takes as a parameter a **Player** object reference (which at runtime may in fact be a **Vendor**, and beginning with the next exercise, a **Shopper**).
e) Modify the "add" command in the player driver so that you will also prompt for a player type. If the type is Vendor, add a **Vendor** object, otherwise a vanilla **Player** object.

Build and run, exercising your new features thoroughly, and also checking that the old features still work (this will be the normal practice in all the incremental steps to the case study).

Ex. 14.6 Create a class **Shopper** derived from **Player**. It will have no new fields at this point, but it will have a constructor and an override of the **Prompt** method. The prompt for a shopper is "S> ". Modify the "add" command in the player driver so that you will now create either a **Vendor** or a **Shopper** depending on the player type—never a generic

Player. Also, modify the initialization code in the **Game** class so that you will create a few shoppers and vendors, not generic players. Build and exercise thoroughly.

Ex. 14.7 Modify the **Vendor** class so that it will now maintain a private **ProductList** representing the products that this vendor carries. Give access to this list through the property **Products**. Also provide a method **GetProducts** that will return an array of **Product**. The constructor should initialize the product list as well as assign the vendor name. Provide the following additional features in your overall program in order to implement the functionality of a vendor.

a) A public static member **CurrentVendor** of **Game** that is initially **null** and will be set to a vendor when a vendor logs in.
b) Initialization code in **Game** to set up several starting vendors and product lists for them.
c) A new command "vendor" in **GameDriver** that will login in a player. A test is made of the type of the player, and if a **Vendor**, then the **CurrentVendor** is set and a welcome message will be displayed showing player's name and balance. Then a vendor driver program will be invoked. The first thing that program does is to display the vendor's inventory, and then it enters a command processing loop.
d) A class **VendorDriver** with a **Loop** method that does the initial inventory display and provides the following commands:

```
V> help
The following commands are available:
        logout  -- logout current vendor
        buy     -- buy from wholesaler
        add     -- add a product to product list
        delete  -- delete a product from product list
        mylist  -- list products of current vendor
        wlist   -- list products of wholesaler
V>
```

e) The code for a vendor buying from the wholesaler will be placed in the **Game** class. The reason for this design decision is to keep the **Vendor** code as general as possible, basically managing a list of products, a balance, and a name. In the **Game** class implement a method **VendorBuy** which takes input parameters of a product name and quantity and returns status as a string. The wholesaler will sell this product, if available in the requested quantity. The product will be added to the vendor's inventory, and the vendor's balance will be decremented to represent the cost of this wholesale purchase.

Build and test thoroughly.

Ex. 14.8 Modify the **Shopper** class so that it will now maintain a private **ItemList** representing the items on the shopping list for this shopper. Give access to this list through the property **Items**. Also provide a method **GetShoppingList** that will return an array of **Item**. The constructor should initialize the shopping list, which will be identical for all shoppers. Provide the following additional features in your overall program in order to implement the functionality of a shopper.

a) A public static member **CurrentShopper** of **Game** that is initially **null** and will be set to a shopper when a shopper logs in.
b) A new command "shopper" in **GameDriver** that will login in a player. A test is made of the type of the player, and if a **Shopper**, then the **CurrentShopper** is set and a welcome message will be displayed showing player's name and balance. Then a shopper driver program will be invoked. The first thing that program does is to display the new player's shopping list, and then it enters a command processing loop.
c) A class **ShopperDriver** with a **Loop** method that does the initial shopping list display and provides the following commands:

```
S> help
The following commands are available:
        visit   -- visit a vendor
        logout  -- logout a player
        buy     -- buy from current vendor
        list    -- list products of current vendor
        mylist  -- show my shopping list
S>
```

d) Implement the "visit" command, which will prompt for a URL and then find the vendor with that URL. The **CurrentVendor** is set to the **Vendor** that is fouond. Provide a **FindVendor** method in the **Game** class that takes a string parameter for the URL and returns a **Vendor** object, **null** if not found.
e) Implement the "buy" command. Again, the code for a shopper buying from a vendor will be placed in the **Game** class. The reason for this design decision is to keep the **Shopper** code as general as possible, basically managing a shopping list of items and a balance. In the **Game** class implement a method **ShopperBuy** which takes input parameters of a product name and quantity and returns status as a string. The vendor will sell this product, if available in the requested quantity. The number of items bought will be decremented from the player's shopping list, and the player's balance will be decremented to represent the cost of this purchase.
f) To carry out the shopper's buy logic, you will also need to implement a suitable **Sell** method in the **Vendor** class.

Build and test thoroughly. The Electronic Commerce Game is now at Step 3.

Chapter 15 – Formatting and Conversion

This chapter covers formatting and conversion. Again, we illustrate with The Electronic Commerce Game, which provides an opportunity to practice formatting your output more attractively. Two kinds of formatting are illustrated. One case involves formatting strings, and the other formatting console output in **Write** and **WriteLine** statements. Our game has also illustrated conversions of various sorts, such as converting an integer to an enumeration type and type conversions associated with inheritance.

Ex. 15.1 Provide neat formatting in the **Player** class and in the **PlayerDriver**. As an example, here is some formatted output.

```
player> login
name: Petworld
password: ppp
Welcome, Petworld
Your balance is $15,000.00
player> list
Count = 6
S>    Ann          $5,000.00   Not Active
S>    Bob          $5,000.00   Not Active
S>    Carl         $5,000.00   Not Active
V>    Toyland     $15,000.00   Not Active       toyland.com
V>    Petworld    $15,000.00   Active           petworld.com
V>    Foodstore   $15,000.00   Not Active       foodstore.com
player>
```

Ex. 15.2 Format your output in **Product** and **ProductDriver**. Again, we provide some sample output.

```
product> list
Vendor name = Wholesaler
Count = 6
airplane toy       10000       $10.00
beanie baby        10000       $15.00
cat carrier        10000       $25.00
dog bone           10000        $5.00
elephant gun       10000       $55.00
fruit basket       10000       $10.00
product> sell
name: airplane toy
quantity: 1500
1500 sold at price of $10.00
product>
```

Ex. 15.3 Comb through the rest of the case study code and format any output you have not yet taken care of. Build and test thoroughly. This would be a good time to also check that there are not any lurking logic errors in your program. You are now at Step 4.

Chapter 16 – Exceptions

Error handling is a very important part of programming. There are two basic approaches to handling errors. One is to have your method calls return a status, which may either indicate success or a particular kind of error. The other approach is the use of exceptions, the subject of this chapter. After a small example, we will continue our illustrations using The Electronic Commerce Game case study.

Ex. 16.1 One source of errors concerns numbers exceeding the size of the data type that is being used to represent them. Study the use of the different integer data types in C# by writing a program that will allow the user for any of the integer data types up to 32 bits in size and receive the input the user types in. Catch any exceptions and display an appropriate error message. Here is a sample run:

```
> help
legal commands are:
        sbyte
        byte
        short
        ushort
        int
        uint
        quit
> sbyte
enter sbyte: 128
Value was either too large or too small for a signed byte.
>
```

Ex. 16.2 We designed The Electronic Commerce Game to pay attention to a number of error conditions. Method calls usually return a string status that is set to "OK" on success and to a descriptive error message otherwise. Still, it is possible for additional errors to occur that are not explicitly checked for. As an example of what can happen, run the product driver and try to change the quantity of a product by entering a non-integer quantity, such as 1.5, for the delta. You will encounter an exception, and the program will crash. Go through the program and put in exception handling code so that your program will never crash. This is not as difficult as it may appear, because you do not have to bracket every call that might cause an exception. It will suffice to enclose the entire body of a command-processing loop in a **try** block and drop through to reading the next command. Implement this exception-handling scheme. Build and test thoroughly, throwing all the bad data and other anomalies that you can think of at your program. You are now as Step 5 of the case study.

Ex. 16.3 The **ArrayDemo** program from Chapter 12 in the book can run across numerous exception. As an example, try creating a new array with various kinds of illegal data, such as a non-numeric string or a negative number. Implement exception handling for the command loop and provide two catch handlers. The first one should catch **FormatException** and display your own error message, "Format error. Try again." The second handler should catch the remaining exceptions and simply display whatever error message is stored in the exception.

Ex. 16.4 To illustrate some more error processing in the **ArrayDemo** program, implement a "set" command, which will call a helper method **set** that prompts for an array index and new value, and then sets the array at the index to the new value. We will certainly get an error if the index entered by the user is out of bounds. Create a user-defined exception class **MyArrayException** that has two public fields, **CurrentSize** and **NeededSize**. The constructor should take three parameters, the error message, the current size of the array, and the needed size of the array. In the **set** helper method throw a **MyArrayException** if the index is out of bound. In your call of the **set** method provide a catch handler that will catch **MyArrayException** and display the data members of this exception object. Here is some sample output.

```
Enter command, quit to exit
: show
5 2 11 7 3
: set
index: 8
Array is too small
Current size = 5
Needed size = 9
:
```

Ex. 16.5 Good error handling can sometimes go further than merely reporting an error. It some cases it may be possible to repair the error. In the error for the "set" command, it may be possible to grow the array to a larger size, and set the new array at the requested index. Implement this error handling strategy for your **ArrayDemo** program. Here is some sample output.

```
Enter command, quit to exit
: show
5 2 11 7 3
: set
index: 8
Array is too small
Current size = 5
Needed size = 9
new value: 77
: show
5 2 11 7 3 0 0 0 77
:
```

Chapter 17 – Interfaces

Interfaces are an important feature of C# and .NET, but their significance may be somewhat difficult to grasp. A good way to think about an interface is that it is a *contract*. The contract is very important in a world of components, where a large software system may be assembled by gluing together components from various sources. For the system to work, these components must fit with each other! If the components faithfully honor the contracts defined by interfaces, we can have confidence that the components will work together.

There is a discipline to working with interfaces. Consider the situation where you have a client program that makes use of a class, and you have some interfaces that implements certain interfaces. First you must make sure that you actually declare your class as implementing the interfaces by using the C# colon inheritance notation. Second, your client program should *only* use interface references in calling features provided by the class. Don't make use of some "extras" that the class provides, because then you may find that you can't just swap in another component that implements the same interfaces.

We will again make use of The Electronic Commerce Game case study as the source of exercises for this chapter. You are given a set of interfaces. Your challenge is restructure your program code so that your classes implement appropriate sets of these interfaces, and your driver programs use only interface references when they invoke functionality provided by your classes. Here is the complete set of interfaces. (They are also implemented by a set of classes that use a relational database for the storage medium.)

```
// Defs.cs

using System.Collections;

interface IProducts
{
      int Count {get;}
      string AddProduct(string name, int qty, decimal price);
      string DeleteProduct(string name);
      Product FindProduct(string name);
      string ChangeQuantity(string name, int delta);
      string ChangePrice(string name, decimal delta);
}

interface IProductList : IProducts
{
      IList GetProducts();
}

interface IItems
{
      int Count {get;}
      string AddItem(string name, int qty);
      string DeleteItem(string name);
      Item FindItem(string name);
      string ChangeQuantity(string name, int delta);
```

```
}

interface IItemList : IItems
{
      IList GetItems();
}

interface IVendorName
{
      string VendorName {get; set;}
}

public interface IPlayerAdmin
{
      int Count {get;}
      string AddPlayer(string name, string pwd, decimal bal, int
roleId);
      Player FindPlayer(string name);
      string DeletePlayer(string name);
      IList GetPlayers();
}

public interface IVendorAdmin
{
      string FindUrl(string name);
      string AddUrl(string name, string url);
      string DeleteUrl(string name);
}

interface IVendorInfo
{
      Vendor FindVendor(string url);
}

public interface IPlayer
{
      string Login(string name, string pwd, out Player play);
      string Logout(string name);
      string ChangeBalance(string name, decimal delta);
      string GetBalance(string name, out decimal bal);
      string SetBalance(string name, decimal bal);
}
```

Examining these interfaces, you should, on the whole, feel quite comfortable. Most of the signatures match exactly with various class methods from our implementation. There is one conspicuous exception, which is the fact that the methods that return multiple data items always return it as **IList**, which is an interface reference. This return type is much more satisfactory than various arrays that we have been using, because it is more general. Arrays implement **IList**, but so do many other .NET classes. Although we will discuss **IList** in Chapter 18, it is easy to start making use of **IList** immediately. As you work your way through the classes, remember one of the changes to make is to replace an array return (such as **Player[])** by **IList**. You will also have to have a **using** statement to bring in the namespace **System.Collections**.

Ex. 17.1 Let's begin with the key class, **PlayerList**. Looking at the interfaces, it is fairly evident that the class should implement the interfaces **IPlayer** and **IPlayerAdmin**. There is one fairly striking difference between the interfaces and our current implementation. The **AddPlayer** method takes a role id, while we have been using an **AddPlayer** method that takes a **Player** object as an input parameter. We instantiate either a **Shopper** object or a **Vendor** object to pass as the parameter. The constructor for **Vendor** takes a URL. In the scheme dictated by the interface, we must proceed somewhat differently. We need to add the appropriate kind of player first, and then afterwards add the URL information. Thus our **PlayerList** class must also implement the **IVendorAdmin** interface.

Restructure your program to do three things:

a) Your **PlayerList** class should implement the interfaces **IPlayer**, **IPlayerAdmin** and **IVendorAdmin**.
b) All the driver code should call through interface references, not object references.
c) Methods in **PlayerList** which are not implementations of interfaces should be either made private or deleted.

As you do this restructuring, also take the opportunity to make additional improvements to your program's structure. For example, in our implementation we had done initialization in a static constructor for the **Game** class. This was not a good idea, because exceptions will not be caught. We moved this code to an initialization method that is explicitly called inside a **try** block. When you are done, *test thoroughly*. (You will find that with the new **AddPlayer** method, the initialization of product data for a vendor has to be done a little differently.)

Ex. 17.2 The next set of interfaces to use are those for products, **IProductList** and **IVendorName**. Restructure your program so that the **ProductList** class supports these interfaces, and all client code calls through interface references and not object references. Again, test thoroughly.

Ex. 17. 3 The next step is to restructure your program so that the **ItemList** class supports the **IItemList** interface, and all client code calls through an interface reference not a class reference. As usual, test thoroughly.

Ex.17.4 The final step is to support the **IVendorInfo** interface. What class is a likely candidate? You may try the **Game** class and encounter a problem from **FindVendor** being a static method. Work around this issue. Build and test very thoroughly. Your case study is now as Step 6.

Chapter 18 – Interfaces and the .NET Framework

This chapter continues the study of interfaces, with a focus on some interfaces that are provided by the .NET Framework itself. We look at collection classes and also at features provided by the **object** base class. Again we base exercises on The Electronic Commerce Game case study. In contrast to the previous chapter where a substantial amount of restructuring of the program was required to accommodate the consistent use of interfaces, you should find the exercises pertaining to the case study in this chapter a comparative breeze. We can now start to reap some of the benefits of our object-oriented approach to the development of our program. Changing the private data structures while still preserving the external interfaces is comparatively easy. In this chapter you will replace the arrays by .NET collection classes. Although it is beyond the scope of this book, it is quite easy in .NET (using ADO.NET) to access data stored in relational databases. As a supplement you might be interested in checking out a database implementation that we provide for the case study.

Ex. 18.1 Change the implementation of the **PlayerList** class to use some .NET collection class in place of an array. You should find the code changes inside the class to be quite easy; in fact, collections are easier to use than arrays! You should find *no* required changes in any of the driver classes. Build and test.

Ex. 18.2 Change the implementation of the **ItemList** and **ProductList** classes to use .NET collections class in place of arrays. Again, you should find *no* required changes in any of the driver classes. Build and test.

Ex. 18.3 We have been using our own method **ToShow** to return a string representation of a number of objects, such as **Player**, **Item**, and **Vendor**. A better approach is to override the method **ToString** from the **object** root class. This is standard and so promotes consistency in code. Another advantage is that the .NET Framework knows about **ToString** and thus can call it for you under certain circumstances.

In this exercise you should go through your case study code and replace the **Show** method by overrides of **ToString**. Adjust your client code accordingly. As a test of the .NET Framework calling **ToString** on your behalf, in the **DisplayPlayers** helper method omit an explicit call to **ToString**. You will see that the code works just fine without it. (Although this code is succinct, it is probably clearer to explicitly call **ToString**, which is what we did in the rest of the client code.) Your case study is now at Step 7. Test thoroughly at this point.

Ex. 18.4 In the various lists in our case study, a name, such as a player name or a product name, is a unique key. In our implementations up to now we have written program code to enforce this uniqueness. We can have the behavior of a unique key provided for us automatically by .NET if we employ the right collection class. Study the documentation for **Hashtable**. This class makes use of a unique key, and it also does efficient searches. Our earlier implementations did linear searches, which would not be efficient for large amounts of data. Study the documentation of **Hashtable**, and then use this kind of

collection to implement a simplified version of our **PlayerList** class. Create a simple **Player** object with these features:

a) Public fields **Name** and **Balance**.
b) A constructor taking name and balance parameters.
c) An override of **ToString**.

Implement **PlayerList** with a **Hashtable** as the private data store . This class should have these features:

a) A property **Count** returning number of elements in the table.
b) **FindPlayer**, which returns the player with a given key, **null** if not found.
c) **AddPlayer**, which adds a player to the list, given a name and balance.
d) **DeletePlayer**, which deletes a player given a key.
e) **GetPlayers**, which returns a **IList**. The **Hashtable** class does not implement the **IList** interface, so you have to build up another kind of list, such as an **ArrayList**, to return. To iterate through a **Hashtable**, use a **IDictionaryEnumerator**.

Provide a suitable driver program. Build and test.

Ex. 18.5 The **ArrayList** class is very flexible. With very little code, you can do basic work of adding different kinds of objects to an **ArrayList** and extract them. As an illustration create a program to keep track of a list of first and last names. Your program should implement a class **Name** with the following features:

a) Public fields **First** and **Last** for first and last name respectively.
b) A constructor to initialize a **Name** object from first and last names that are passed as separate parameters.
c) An override of **ToString** that will return a complete name by concatenating the first and last names.

Provide a test driver program that implements the following commands.

a) An "addname" command that prompts for first and last name, creates a **Name** object and adds it to the list.
b) An "addstring" command that prompts for first and last name, concatenates these strings and adds the concatenated string to the list.
c) A "list" command that will iterate through all the objects in the list as **string** type and display the string
d) A "first" command that will iterate through all the objects in the list as **Name** type and display the first name.
e) An "index" command that will use indexing to iterate through the list and display the object at each index as a string along with the value of the index.
f) A "clear" command that will clear the list of all elements.

Enclose the body of your loop in a **try** block so that you can recover from exceptions. Build and test. Try entering all names. Then try entering all strings. Try a mixture. You can use the "clear" command between experiments. Here is a sample run.

```
Enter command, quit to exit
: addname
first name: Joan
last name: Smith
: addname
first name: Bill
last name: Jones
```

```
: index
0   Joan Smith
1   Bill Jones
: first
Joan
Bill
: list
Exception: Specified cast is not valid.
:
```

Ex. 18.6 The problem with this solution is that the **ArrayList** class is not type-safe by itself. You can put any kind of data type into an **ArrayList** (they are stored as **object**). When you extract the data, you perform a cast. You are in trouble if you cast to a different data type when you take the data out than you used when you put the data in. A standard solution to this issue is to create a type-safe wrapper class. The .NET Framework makes it easy to create type-safe collection classes by inheriting from the **CollectionBase** class. Study the documentation for **CollectionBase** and implement a class **NameList** that inherits from **CollectionBase**. Supply just one method, **AddName** that adds a name to the built in **List** object in the base class. Use the same driver program with only two changes:

a) Declare the list as a **NameList** rather than **ArrayList**.
b) In both the "list" and "first" commands iterate through **Name** objects.

You'll get two compiler errors—where you attempt to add a **string** to the list and where you do indexing. Comment out the offending code, build and run. You should now be able to work cleanly with **Name** objects, and you got a compile-type error when you attempted to add something other than a **Name** to your list.

Ex. 18.7 Add an indexer to your **NameList** class to access the indexer in the built-in **List** object. You should now be able to use the "index" command in the driver program. (You may wish to review the discussion of indexers from Chapter 12 in the book.)

Chapter 19 – Delegates and Events

Delegates in C# are somewhat analogous to function pointers in C, but they are much more powerful and robust. Delegates are like classes and thus can provide a type-safe implementation of callbacks. It is also possible to compose delegates, so a single call can result in multiple delegate methods being called. Delegates are also the foundation of events. Events were popularized in Visual Basic and are now a useful feature in all .NET languages.

When you first study delegates and events, it may not be entirely clear what additional features and benefits are provided by events as opposed to just using delegates. Events provide a useful syntax for expressing the event-handling paradigm and for placing reusable code for specifying and firing events right in a class. Events can even be part of a formal interface contract.

Ex. 19.1 As a warm-up exercise, write a program that sets up an "echo" delegate which will simply echo back any string to the console. Provide two static delegate methods, one which will echo the string verbatim, and another which will convert the string to upper case and then display it. Provide a driver that will allow you to enter strings and then invoke the delegate. Here is some sample output:

```
Enter string, quit to exit
: hello
hello
HELLO
: goodbye
goodbye
GOODBYE
: this is boring
this is boring
THIS IS BORING
: quit
```

Ex. 19.2 For our next example we will illustrate dynamically adding and subtracting delegate methods. Before setting up the delegates, first implement a program that can maintain a list of strings. Provide commands to add names, delete names, and list all the names on the list. If you try to add a duplicate name or delete a name not on the list, you should get an error message.

Ex. 19.3 Extend the previous example by declaring a delegate **HistoryDelegate**, whose method takes two string parameters, a name and an action. Create a class **History** with the following features:

a) A data structure to record a history list of names and actions.
b) A method **Announce** which will concatenate a name and an action and display the concatenated string at the console.
c) A method **Record** which will concatenate a name and an action and add the concatenated string to the history list.
d) A method **Display** which will display the history list at the console.

In the driver program create two delegate objects holding the **Announce** and **Record** methods respectively. Declare a third delegate object, initially **null**, which at run time can have the announce and record delegates added and subtracted. Depending on the setting, the name/action may be displayed at the console, written to the history list, or both, or neither. Provide commands to toggle the announce and record delegates, and also to display the history list. Here are the complete commands:

```
Enter command, quit to exit
: help
The following commands are available:
        add       -- add a name
        delete    -- delete a name
        list      -- list all names
        announce  -- toggle announce
        record    -- toggle record
        history   -- display the history list
        quit      -- exit the program
:
```

Build and test thoroughly.

Ex. 19.4 Our next goal is to get practice using events in classes in order to achieve greater code reuse. Before tackling events, let's make a type-safe string list class. You may wish to review Exercise 6 in Chapter 18 for an example of creating a type-safe collection by inheriting from **CollectionBase**. Implement a type-safe class **StringList** with the following features:

a) A method **Add** to add a string to the list. The method tests for duplicate strings and returns **true** is successful in adding a non-duplicate string, otherwise **false**.
b) A method **Delete** to remove a string from the list. If the string was not on the list, return **false**, otherwise **true**.

Modify the test driver program from Exercise 2 to use the **StringList** class in place of a generic **ArrayList**.

Ex. 19.5 We will now add code to the file containing the **StringList** class to define an event handler delegate, declare an event in the class, and fire the event in the class methods when appropriate. This structure will make it easier for clients of the class to work with the events. Basically, they will implement event handlers and hook the event handlers to the event. Begin by adding the following to the file containing the **StringList** class:

a) Definition of a delegate **HistoryEventHandler**. It takes a name and action **string** parameters and returns **void**. This is the same as the **HistoryDelegate** defined in Exercise 3, only changing the name to reflect usage associated with events.
b) In the class declare a public event **Changed** of type **HistoryEventHandler**.
c) In the **Add** method place a call to **Changed** when a new name is added. You should check that **Changed** is not **null** before calling it.

In the driver class do the following:

a) Remove the third delegate object, whose role is now filled by the public **Changed** event in the class we are calling.

b) Change the names of the first two delegates to **AnnounceHandler** and **RecordHandler**. They are of type **HistoryEventHandler**. Again, the name change is to reflect usage associated with events.
c) In the "add" and "delete" commands, remove the code that called the delegate. Event firing is now done in the class itself.
d) In the "announce" and "record" commands modify the += and -= code to reflect the new names.

Build and test. You should have the same behavior as in Exercise 3. Examine the driver code. Do you think it is simpler and more intuitive than the code from Exercise 3?

Ex. 19.6 Although events can have any signature, as illustrated in the preceding exercise, the .NET Framework prescribes a standard signature for events, as discussed in the book. The first parameter specifies the object that sent the notification, and the second parameter is of type **EventArgs** or of a class derived from **EventArgs**. If you do not provide any data to go along with the event, you can use a standard delegate data type defined in the .NET Framework, **EventHandler**. Otherwise, define your own delegate type and create your own event argument class, such as **HistoryEventArgs**, derived from **EventArgs**. Rework your solution to conform to this standard.

Ex. 19.7 One of the advantages of events over delegates is that events can be placed in an interface and thus become part of the formal contract that a class must fulfill. Create a file **Defs.cs** that has the definition of your **HistoryEventHandler** delegate and also the definition of an interface **IStringList** that specifies the methods and the event of the **StringList** class. Rework your solution to incorporate this definition file, and make your **StringList** class inherit from **IStringList**.

Ex. 19.8 Add an event interface to your **PlayerList** class in The Electronic Commerce Game case study. This interface should have a **Disqualified** event, which fires whenever a player's balance becomes negative. Provide an event handler which will display a message at the console giving the disqualified player's name and ending balance. Build and test thoroughly. Note that there are many places in the program which can trigger a negative event for a shopper or vendor player, but the logic for raising the event and handling the event can be quite localized. The case study is now at Step 8.

Chapter 20 – Advanced Features

This chapter takes up a few somewhat more advanced topics. We begin with a number of exercises illustrating multiple-thread programming in C#. This makes a nice follow-on to the previous chapter's discussion of delegates, because the .NET threading model is based on delegates. A sequence of exercises leads you to set up a small testbed for investigating multiple threads. You could easily do some further threading experiments at this point, but our exercises next turn to the use of files and streams in .NET. We conclude with an exercise on serialization, which is a nice way to persistently store object data.

Ex. 20.1 A good way to investigate a programming topic is to write interactive programs that allow you to explore the behavior of a program under varying inputs. To get started exploring threads, create an interactive version of the **ThreadDemo** program in the book. Rather than hardcoding some specific threads, provide a command loop that allows you to create an arbitrary number of threads, giving them whatever starting parameters you decide on at runtime. There are two commands, "new" and "quit." Here is some sample output.

```
Enter command, quit to exit
thread> help
The following commands are available:
        new       -- start a new thread
        quit      -- exit the program
thread> new
delta: 400
count: 5
name: aaa
thread> new
delta: 1000
count: 5
name: bbb
thread>
```

You will then see the output from the threads. Note that in this program we had the user assign a name for each thread. Note also that we had to make some other changes to the **ConsoleLog** class to get the output shown, avoiding the first thread starting up immediately and making it hard to enter data for the second thread.

Ex. 20.2 A general solution to the problem of threads starting up prematurely is to decouple the actual thread startup from creating the threads. Modify your program to maintain a list of threads, and also provide "list" and "start" commands. The "list" will command will display a list of threads by name along with the sleep interval specified. The "start" command will start all the threads on the list. Here is some sample output.

```
Enter command, quit to exit
thread> new
delta: 400
count: 5
name: aaa
```

```
thread> new
delta: 1000
count: 5
name: bbb
thread> list
aaa           400
bbb          1000
thread> start
thread> aaa: ticks = 0
bbb: ticks = 0
aaa: ticks = 400
...
```

Ex. 20.3 To explore further, investigate the **ThreadState** enumeration from the .NET Framework documentation. This enumeration provides masks for various thread states, some of which may occur in combination. In Chapter 5, Exercise 8, we wrote a program to display a string representation of several common thread states given the numeric representation. Use this program logic to enhance your "list" command, so that the thread state will be displayed along with the name and sleep interval. Here is a sample list before we have started the threads.

```
thread> list
aaa           400   Unstarted
bbb          1000   Unstarted
```

Ex. 20.4 A chronic problem with a program such as this one is the intermingling of output from various threads. While it is easy to input data for a number of threads before starting any of them, once some threads are started it is difficult to enter information for some additional threads. Another issue is that a simplistic "start" command will run into trouble trying to restart threads that have stopped. Enhance your program by putting into place locking behavior, so that once you have started entering data for a new thread, you will not be interrupted by output from running threads until you have finished your input. Also, address the issue of attempting to restart threads that have stopped.

Ex. 20.5 Another approach to the intermingling of output from various threads is to have different threads write to different output streams. Modify your solution for Exercise 3 to have the main program open up a **StreamWriter** that writes to a file. Replace the **ConsoleLog** class by a **FileLog** class. The **FileLog** constructor should take a **StreamWriter** object, which is saved away. All output is done to this writer. It is the responsibility of the main program to close the writer (which is important to ensure the buffer is flushed and the data is actually written to the file). Build and run. This should give you a more interesting view of the states of the threads at various point during program execution. Here is some sample output:

```
thread> list
aaa           400   Unstarted
bbb          1000   Unstarted
thread> start
thread> list
aaa           400   WaitSleepJoin
bbb          1000   WaitSleepJoin
```

```
thread> list
aaa           400   Stopped
bbb          1000   WaitSleepJoin
thread> list
aaa           400   Stopped
bbb          1000   Stopped
```

Ex. 20.6 In principle, we could do file input and output of complex data structures using files and streams, as illustrated (for output) in the preceding exercise. But traversing complex data structures and formatting appropriately for the stream we write to can quickly become rather tedious. Fortunately, the .NET Framework provides a powerful *serialization* mechanism to greatly simplify reading and writing of objects. Serialization was discussed in the chapter in the book. You should find it fairly easy to apply the technique to your own examples. We will illustrate with a simplified **PlayersList** class. For starters, create classes and a driver program with the following features (you can quickly implement by stripping down relevant code from the case study):

a) A **Player** class with a string **Name** and a decimal **Balance**. There should be a constructor taking a name and a balance, and an override of **ToString**.
b) A **PlayerList** class with the following features:
    i. A private array list for holding players
    ii. A read-only property **Count**
    iii. A **FindPlayer** method that will return a **Player** given a name, **null** if not found.
    iv. An **AddPlayer** method taking a name and a balance that will create a player and add it to the list. An error string is returned if the player is already on the list, otherwise "OK."
    v. A **GetPlayers** method that return an **IList**.
    vi. A **Clear** method that will empty the list.
c) A driver program that implements these command:

```
Enter command, quit to exit
player> help
The following commands are available:
        add     -- add a new player
        find    -- find a player
        list    -- list player information
        clear   -- clear players
        quit    -- exit the driver
player>
```

Ex. 20.7 Now add the capability to **PlayerList** to serialize itself. Add "save" and "load" commands to the driver program. In the book we illustrated use of a binary formatter. For variety, try a SOAP formatter. The file written will be a text file. If you are familiar with XML, you will recognize that the file is formatted as XML. Build and test.

Chapter 21 – Components and Assemblies

Components are a key concept of modern software engineering. Components represent a step beyond object-oriented programming in software reuse. In .NET components, are very easy to create via a special kind of project type, a class library. The exercises give you practice creating a number of class libraries and client programs that use the class library you have created. You will conclude by creating a componentized version of The Electronic Commerce Game case study, bringing you to Step 9 of this project.

Ex. 21.1 As a warm-up we'll create a component to do simple math, the operations of add, subtract, multiply and divide. When developing a new component, you may find it easiest to first create a monolithic program in which the component logic and the user interface test code are all in one project. This will make it easy to debug. When your logic is correct, you can then break the files into a class library project and an application project. So to get started, create a class **SimpleMath** that has static methods to perform the four elementary arithmetic operations on the type **double**. Implement a simple test driver program that will allow the user to interactively test the class by choosing a command and then entering test data. Place the body of your command loop inside a **try** block, so that you can catch exceptions such as bad input format or divide by zero.

Ex. 21.2 Create a class library **MathLib** that contains the **SimpleMath** class. Build the component, obtaining the dynamic link library **MathLib.dll**.

Ex. 21.3 Create a client console program **MathClient** that contains the test driver. Copy the DLL **MathLib.dll** to the source directory of **MathClient**. Add a reference to this DLL. Build and test.

Ex. 21.4 Now let's tackle componentizing the game cases study. One of the goals of the design was to create reusable code. We attempted to design the **Player** and **PlayerList** classes so that they would be generally useful in a number of games in which players are identified by name, hold a currency balance, and may be active or inactive, moving between these states by logging in and logging out. Adapt the code from the last version of the game (Exercise 8 in Chapter 19) to create a component **PlayerLib** that embodies these core features. You will find that in order to accomplish this task you will need to do a little redesign as well as stripping out other features of the game, such as items, products, and so on. We can keep the concept of a role ID, which can be useful in a number of different game scenarios, but specific references to the classes **Vendor** and **Shopper** have to go. Build your component, obtaining the DLL **PlayerLib.dll**.

Ex. 21.5 Create a client console program **PlayerClient** that contains a test driver. You can adapt the driver program from **PlayerDriver.cs** in the case study. Also had a handler for the **Disqualified** event, basing your code on the handler in the **Game** class of the case study. Copy the DLL **PlayerLib.dll** to the source directory of **PlayerClient**. Add a reference to this DLL. Build and test.

Ex. 21.6 We now wish to build a component containing the entire game logic, apart from the user interface driver code. One approach would be to use the **PlayerLib.dll** component already created and create a second component with the remaining functionality. Unfortunately, the current inheritance-based design create a tight coupling between the **Player** class and the derived classes **Vendor** and **Shopper**, which in turn entangles us with the **Item** and **Product** classes. So our design would have to be refactored. For the purpose of this exercise, don't worry about such an overhaul. Build a single component **GameLib.dll** that has all the game logic and none of the user interface driver code.

Ex. 21.7 Create a client console program **GameClient** that contains all the test drivers. Copy the DLL **GameLib.dll** to the source directory of **GameClient**. Add a reference to this DLL. Build and test thoroughly. You are now at Step 9 of the case study.

Chapter 22 – Introduction to Windows Forms

Although console programs like we have been using so far in the book are very convenient for learning a programming language, very few modern programs are of this type. Most programs are either Windows applications or Web applications. We conclude our coverage of C# with an introduction to creating Windows applications using C# and .NET, making use of the Windows Forms classes of the .NET Framework.

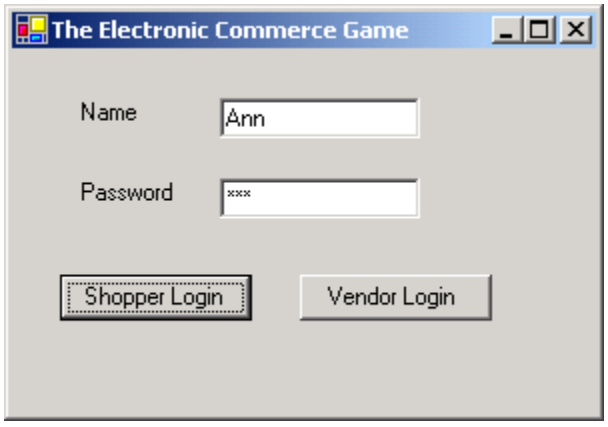
The format of the exercises for this last chapter is somewhat different. It is very easy to create exercises, because every program you have written so far using a console interface could be rewritten using a Windows graphical user interface (or GUI). The first thing you should do is to write a few simple Windows versions of programs you have already written.

The next step would be to write a larger Windows application, and the natural project to tackle is a Windows version of The Electronic Commerce Game. That is the major assignment for the last chapter, which will bring the project to Step 10. You should feel free to craft whatever user interface is most appealing to you. You can instantly gain access to all the backend logic you have created by adding a reference to the class library **GameLib.dll** that you created in Chapter 21.

In these notes we walk through one example of a user interface.

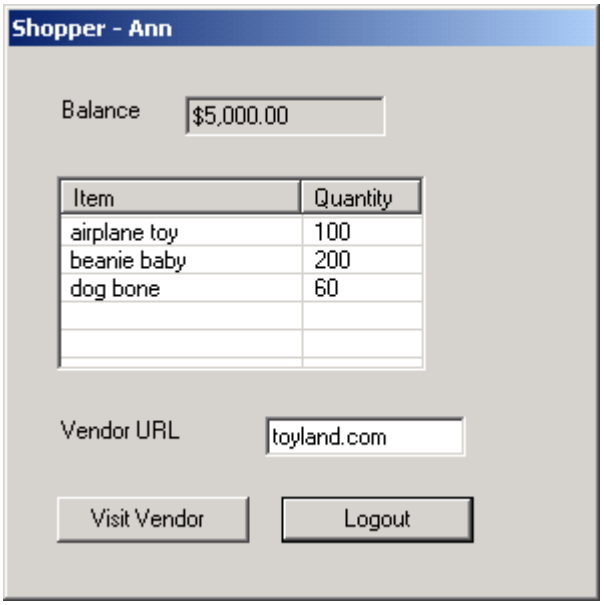
Top-Level Window

The top-level window provides a login screen for both shoppers and vendors. Note that in Windows it is very easy to conceal a password that is typed in by setting a password character property in the TextBox control.



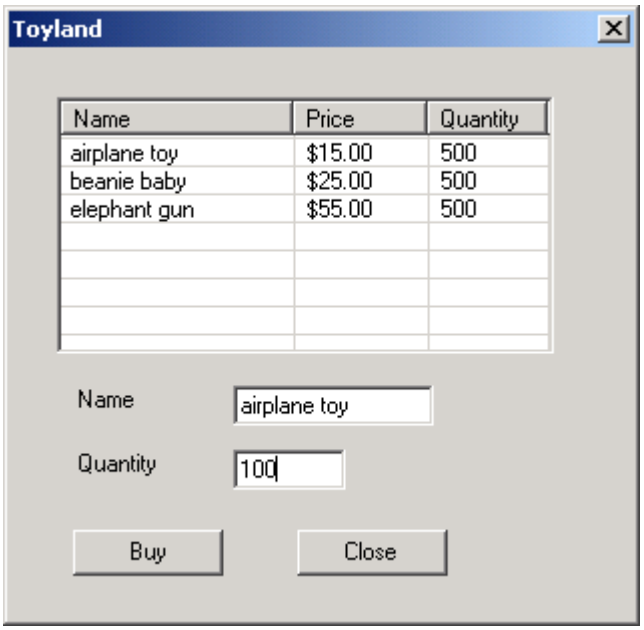
Shopper Window

When a shopper logs in, a shopper window is brought up that shows the shopper's name in the title bar. The shopper's balance and shopping list are shown, and buttons are provided to visit a vendor (whose URL is entered in a textbox) and to logout. The *only* way to exit this window is through the logout button.

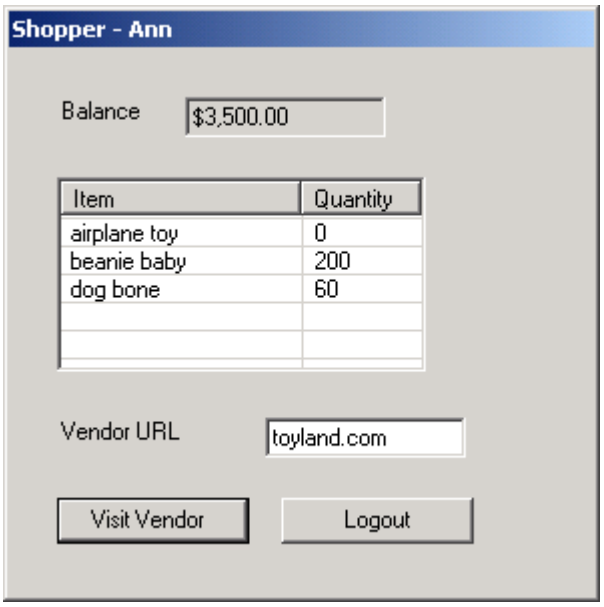


Visiting a Vendor

Clicking the Visit Vendor button brings up the window shown:

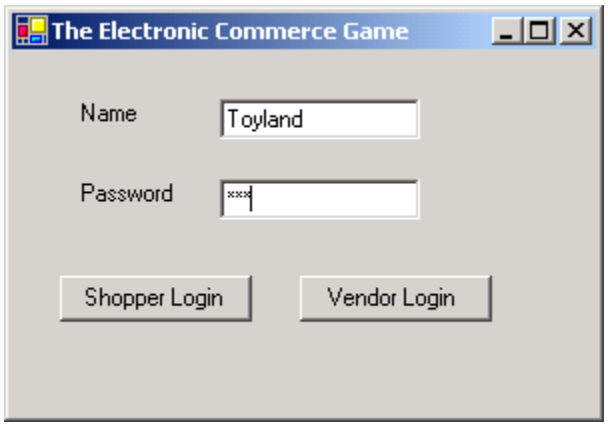


The name of the Vendor is shown in the title of the window, and the inventory in a list view control. Textboxes are provided for entering the name and quantity of a product to purchase. Here Ann is going to buy 100 airplane toys, fulfilling the first item on her shopping list. She clicks Buy, which will update the vendor quantity, showing a decrease in inventory. If she wants, she can buy more products while still at Toyland. When done, she closes this window. Back in the shopper window, the shopping list and balance have been updated:



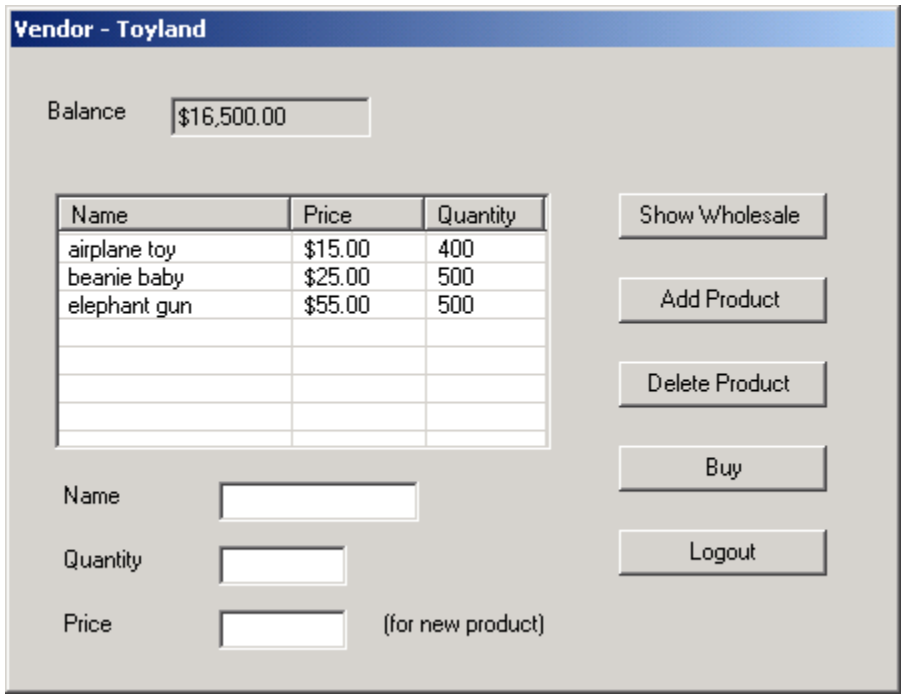
She can visit other vendors and try to fulfill all items on her shopping list. If she does so without her balance becoming negative, she wins. When she is done, she logs out.

Back in the top-level window, we can now explore how a vendor will play the game. Here Toyland is about to login:



Vendor Window

The vendor window shows name in the title. The balance and inventory are shown. In this case, the balance has been increased from a starting value of $15,000 reflecting a sale of 100 airplane toys to Ann. The quantity in inventory has been decreased.



Buttons are provided to add or delete a product, and to buy a product from the wholesaler. Another button will bring up the wholesale pricelist:

That's it! Try out the supplied version, and then build your own!

Appendix
Case Study: The Electronic Commerce Game

Introduction

Learning object-oriented programming involves more than learning the syntax of an object-oriented language such as C#. It is also more than learning how to write small programs with classes. To fully understand and internalize the object-oriented mindset, it is necessary to gain some experience with larger programs. It is only with "programming in the large" that object-oriented techniques really come into their own. For learning purposes, of course, the program cannot be *too* large. It needs to be large enough to illustrate the interactions of various objects, but still small enough to be manageable.

The book *Introduction to C# Using .NET* contains one such case study, a hypothetical banking system, that is elaborated in seven progressive steps in Chapters 12 through 18. From the standpoint of learning, it will also be helpful for you to progressively implement your own system. The Electronic Commerce Game is designed as a small system that you can implement using object-oriented programming in C#. Exercises in the chapters will lead you to one implementation, using first arrays and then .NET collections. Another approach would be for you to design your own implementation. You could use the supplied solution as a reference.

This appendix gives an overall introduction to the case study. It includes a player's guide and also an outline of the program design. It also sketches some further elaborations of the case study that could be implemented to illustrate various topics in the .NET Framework, including ADO.NET, ASP.NET and Web services. A database implementation using ADO.NET is also supplied with the source code accompanying this set of exercises.

Another version of this case study was used for illustrating COM+. See Appendix B in the book *Understanding and Programming COM+* by Robert J. Oberg.

Overview

The basic concept of the game is extremely simple—online buying and selling of products. It is structured as a game in which players compete to fulfill a shopping list by visiting various vendor sites and making purchases.

Players take on the role of Shopper (and may temporarily assume the role of a Vendor if they can discover a Vendor password). Vendors carry different products, which they buy at wholesale and sell at retail. Vendors decide what products to stock, what inventory level to maintain, and what retail prices to charge for their products. They may adjust their product list, prices and inventory during the game. All Vendors are given the same starting capital, and they earn more money by selling products to Shoppers. Their balance declines when they buy products at wholesale for their inventory. The winning Vendor is

the one with the most money when the game is over. A Vendor whose balance drops below zero is disqualified.

At the beginning of a game all Shoppers are supplied with an identical starting balance and an identical shopping list, specifying items to be purchased from Vendors. Shoppers pay for their purchases out of their balance. The winning Shopper is the one who has purchased all the items on the shopping list and has the largest balance. A Shopper whose balance drops below zero is disqualified. The ending balance gives the score for the game.

There are multiple Shoppers, and there are preassigned Vendors with various starting inventories. New Shoppers and Vendors can be dynamically created. A Shopper may temporarily assume the role of a Vendor by logging in as a Vendor. Shopper transactions are carried out through a simulated Web. To make a purchase, a Shopper visits a Vendor by specifying a URL.

There are two user interfaces to the game. The first is a command-line interface, and the second is a graphical user interface, implemented via Windows Forms. (A Web-based interface will also be developed. Check the Object Innovations website for information.)

Command-line Version of the Game

The command-line version discussed in this appendix uses an Access database for permanent storage. The advantage of this version is that you can look at the data as the game is played independently from the game interface. Thus as you enter various commands in the game, you may observe how the data changes in the database.

Database

The database is **Game.mdb**. There are four tables.

Products

The Products table has columns for VendorName, ProductName, Quantity, and Price. There are several retail vendors, and one wholesale supplier whose name is Wholesaler. The starting data is illustrated in the screenshot. Each retail vendor has a starting inventory of 500 of each item, and the Wholesaler a starting inventory of 10000. (It is assumed the wholesaler will never run out. Vendors can buy to restock their inventory, and can add or delete items that they carry.

Products : Table

VendorName	ProductName	Quantity	Price
Foodstore	dog bone	500	$10.00
Foodstore	fruit basket	500	$15.00
Petworld	cat carrier	500	$35.00
Petworld	dog bone	500	$15.00
Toyland	airplane toy	500	$15.00
Toyland	beanie baby	500	$25.00
Toyland	dog bone	500	$10.00
Wholesaler	airplane toy	10000	$10.00
Wholesaler	beanie baby	10000	$15.00
Wholesaler	cat carrier	10000	$25.00
Wholesaler	dog bone	10000	$5.00
Wholesaler	elephant gun	10000	$55.00
Wholesaler	fruit basket	10000	$10.00

Record: 14 of 14

Vendors

The Vendors table has columns Name and Url.

Vendors : Table

Name	Url
Petworld	petworld.com
Toyland	toyland.com
Foodstore	foodstore.com
Wholesaler	wholesaler.com

Record: 5 of 5

Players

The Players table has columns Name, Password, Balance, Active, and RoleId.

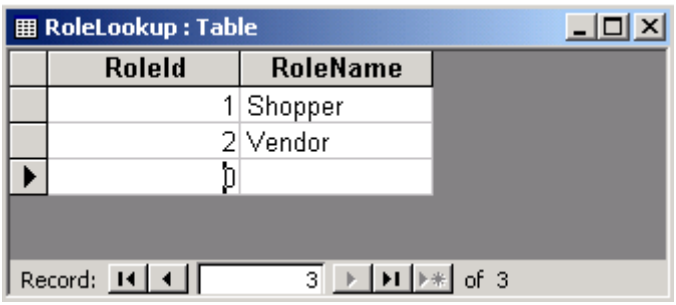
Players : Table

Name	Password	Balance	Active	RoleId
Ann	aaa	$5,000.00	☐	1
Bob	bbb	$5,000.00	☐	1
Carl	ccc	$5,000.00	☐	1
Petworld	ppp	$15,000.00	☐	2
Toyland	ttt	$15,000.00	☐	2
Foodstore	fff	$15,000.00	☐	2
		$0.00	☐	0

Record: 7 of 7

RoleLookup

The RoleLookup table has columns RoleId and RoleName.



GameDriver

The top-level class **GameDriver** has a loop offering a choice of four commands.

```
Enter command, quit to exit
: help
The following commands are available:
        shopper -- shopper driver
        vendor  -- vendor driver
        player  -- player admin
        product -- product admin
        quit    -- exit the program
```

ProductAdmin

The **ProductAdmin** class has a loop to exercises working with products in a stand-alone fashion.

```
: product
Enter command, quit to exit
prod> help
The following commands are available:
        list    -- list products
        add     -- add a product
        delete  -- delete a product
        find    -- find a product
        changeq -- change quantity
        changep -- change price
        vendor  -- new vendor
        quit    -- exit the program
prod>
```

PlayerAdmin

The **PlayerAdmin** class has a loop to exercise working with players in a stand-alone fashion.

```
: player
Enter command, quit to exit
player> help
The following commands are available:
        login   -- login a player
        logout  -- logout a player
        add     -- add a new player
        delete  -- delete a player
        list    -- list player information
        quit    -- exit the program
player>
```

ShopperDriver

The **ShopperDriver** class has a loop that lets a Shopper play the game. A player enters the loop via the "shopper" command. After logging in, the program verifies that the player has the role Shopper. Then a welcome message is displayed and the starting balance and shopping list are shown. Note that the shopping list only applies to this game and is not persistent.

```
: shopper
Please login
name: Bob
password: bbb
Welcome, Bob
Your balance is 5000
Your shopping list:
   airplane toy            100
   beanie baby             200
   dog bone                 60
Enter command, logout when done
S>
```

Commands are provided to allow the Shopper to visit various vendors and buy products. Both the list of products available from the current vendor and the player's own shopping list can be shown. When done, the player logs out.

```
S> help
The following commands are available:
        visit   -- visit a vendor
        logout  -- logout a player
        buy     -- buy from current vendor
        list    -- list products of current vendor
        mylist  -- show my shopping list
S>
```

VendorDriver

The **VendorDriver** class has a loop that lets a Vendor play the game. A player enters the loop via the "vendor" command. After logging in, the program verifies that the player has the role Vendor. Then a welcome message is displayed and the starting balance and inventory are shown.

```
: vendor
Please login
name: Toyland
password: ttt
Welcome, Toyland
Your balance is 15000
Your inventory:
airplane toy            500      $15.00
beanie baby             500      $25.00
dog bone                500      $10.00
Enter command, logout when done
V>
```

Commands are provided to allow the Vendor to buy from wholesale, add a product to the list of products carried, and list the vendor's own products and those carried by the wholesaler. When done, the player logs out.

```
Enter command, quit to exit
V> help
The following commands are available:
        logout  -- logout current vendor
        buy     -- buy from wholesaler
        add     -- add a product to product list
        delete  -- delete a product from product list
        mylist  -- list products of current vendor
        wlist   -- list products of wholesaler
V>
```

Design Notes

This section contains some informal notes describing the design. It is not a formal design specification.

Goals

The design is influenced by several goals.

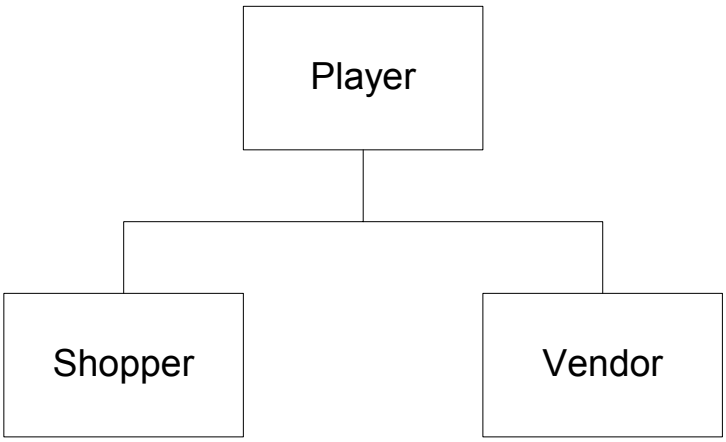
1. **Extensible**. We aim to develop a number of different versions of the game, i.e. we want to be able to extend our program in various directions.
    a. **Different data stores**. Data store may be arrays, .NET collections and database tables.
    b. **Different user interfaces**. Both a command-line interface and a Windows graphical user interface will be provided. Eventually we want to provide a Web-based interface.
    c. **Extensions to the game itself**. Electronic commerce is a big area, and our game captures only a small element of what is involved. We should be able to extend the game to incorporate additional elements (for example, shipping).
2. **Reusable code**. As we develop different versions of the game, we would like to reuse as much code as possible.

3. **Simple error processing**. Many different kinds of errors can occur. Both during development and when the game is being played we want reasonable error messages to be displayed. To minimize development effort, we want this error processing to be as simple as we can make it.

Class Hierarchies

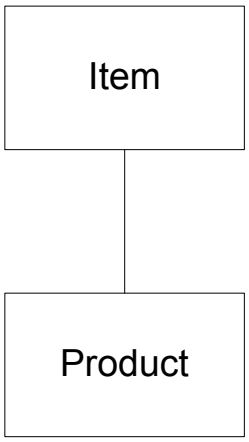
Player

One of the main component of the game is players of various sorts. A login/logout protocol is provided, which we would like to implement only once. A natural model is a base class **Player** and derived classes for each type of player.



Product

Another kind of component are products. As products are bought and sold, they need a name, quantity and price. But for a shopping list only a name and quantity is needed. This leads to another class hierarchy. An **Item** has a name and quantity, and a **Product** has a name, quantity, and price.



Error Processing

We implement a very simple, consistent error processing scheme. Most methods will have **string** as the return data type. For success, the return will be "OK." For failure, the return will be a descriptive error message. Thus we do not need a system of error codes and simply can use the descriptive string directly in error messages. **Find** methods that are expected to return an object reference will return **null** if the object is not found.

Interfaces

We specify key functionality through C# interfaces, which define various contracts. This use of interfaces will help us ensure that different implementations of data stores, such as array, collection and database, can be used interchangeably with the same client code.

For reference, here are the various interfaces:

```
interface IProducts
{
   int Count {get;}
   string AddProduct(string name, int qty, decimal price);
   string DeleteProduct(string name);
   Product FindProduct(string name);
   string ChangeQuantity(string name, int delta);
   string ChangePrice(string name, decimal delta);
}

interface IProductList : IProducts
{
   IList GetList();
}

interface IItems
{
   int Count {get;}
   string AddItem(string name, int qty);
   string DeleteItem(string name);
   Item FindItem(string name);
   string ChangeQuantity(string name, int delta);
}

interface IItemList : IItems
{
   IList GetList();
}

interface IVendorName
{
   string VendorName {get; set;}
}

interface IPlayerAdmin
{
   int Count {get;}
   string AddPlayer(string name, string pwd, decimal bal, int roleId);
```

```
   Player FindPlayer(string name);
   string DeletePlayer(string name);
   IList GetPlayers();
}

interface IVendorAdmin
{
   string FindUrl(string name);
   string AddUrl(string name, string url);
   string DeleteUrl(string name);
}

interface IVendorInfo
{
   Vendor FindVendor(string url);
}

public interface IPlayer
{
   string Login(string name, string pwd, out Player play);
   string Logout(string name);
   string ChangeBalance(string name, decimal delta);
   string GetBalance(string name, out decimal bal);
   string SetBalance(string name, decimal bal);
}
```

Note that none of the methods do input or output, which will be performed in separate user interface classes. For the purpose of obtaining a list of objects of various sorts, methods will consistently return an **IList**.