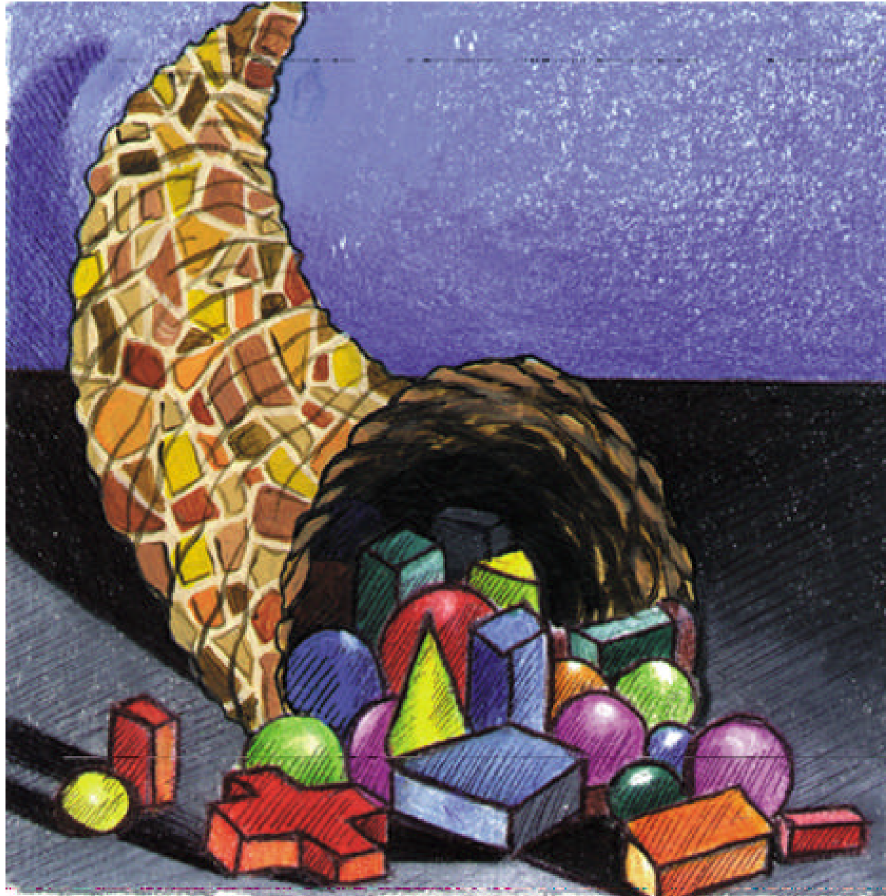
Introduction to Design Patterns in C#



Copyright © 2002 by James W. Cooper

IBM T J Watson Research Center

February 1, 2002

Copyright © , 2002 by James W Cooper

1. What are Design Patterns? 21

Defining Design Patterns 23

The Learning Process.. 25

Studying Design Patterns 26

Notes on Object-Oriented Approaches 26

C# Design Patterns.. 27

How This Book Is Organized 28

2. Syntax of the C# Language 29

Data Types 30

Converting Between Numbers and Strings 32

Declaring Multiple Variables.. 32

Numeric Constants.. 32

Character Constants 33

Variables 33

Declaring Variables as You Use Them.. 34

Multiple Equals Signs for Initialization.. 34

A Simple C# Program.. 34

Compiling & Running This Program.. 36

Arithmetic Operators.. 36

Increment and Decrement Operators 37

Combining Arithmetic and Assignment Statements.. 37

Making Decisions in C#.. 38

Comparison Operators 39

Copyright © , 2002 by James W Cooper

Combining Conditions 39
The Most Common Mistake 40
The switch Statement 41
C# Comments 41
The Ornery Ternary Operator 42
Looping Statements in C# 42
The while Loop 42
The do-while Statement 43
The for Loop 43
Declaring Variables as Needed in For Loops 44
Commas in for Loop Statements 44
How C# Differs From C 45
Summary 46
3. Writing Windows C# Programs 47
Objects in C# 47
Managed Languages and Garbage Collection 48
Classes and Namespaces in C# 48
Building a C# Application 49
The Simplest Window Program in C# 50
Windows Controls 54
Labels 55
TextBox 55
CheckBox 56

Copyright © , 2002 by James W Cooper

Buttons 56
Radio buttons 56
Listboxes and Combo Boxes 57
The Items Collection........ 57
Menus........ 58
ToolTips........ 58
Other Windows Controls 59
The Windows Controls Program 59
Summary........ 61
Programs on the CD-ROM 47
4. Using Classes and Objects in C# 62
What Do We Use Classes For? 62
A Simple Temperature Conversion Program........ 62
Building a Temperature Class........ 64
Converting to Kelvin........ 67
Putting the Decisions into the Temperature Class 67
Using Classes for Format and Value Conversion........ 68
Handling Unreasonable Values........ 71
A String Tokenizer Class 71
Classes as Objects 73
Class Containment 75
Initialization........ 76
Classes and Properties........ 77

Copyright © , 2002 by James W Cooper

Programming Style in C# 79
Summary 80
Programs on the CD-ROM 62
5. Inheritance 81
Constructors 81
Drawing and Graphics in C# 82
Using Inheritance 84
Namespaces 85
Creating a Square From a Rectangle 86
Public, Private and Protected 88
Overloading 89
Virtual and Override Keywords 89
Overriding Methods in Derived Classes 90
Replacing Methods Using New 91
Overriding Windows Controls 92
Interfaces 94
Abstract Classes 95
Comparing Interfaces and Abstract Classes 97
Summary 99
Programs on the CD-ROM 99
6. UML Diagrams 100
Inheritance 102
Interfaces 103

Copyright © , 2002 by James W Cooper

Composition.. 103
Annotation.. 105
WithClass UML Diagrams 106
C# Project Files 106
7. Arrays, Files and Exceptions in C# 107
Arrays.. 107
Collection Objects.. 108
ArrayLists.. 108
Hashtables 109
SortedLists 110
Exceptions 110
Multiple Exceptions 112
Throwing Exceptions 113
File Handling... 113
The File Object.. 113
Reading Text File... 114
Writing a Text File 114
Exceptions in File Handling.. 114
Testing for End of File 115
A csFile Class... 116
8. The Simple Factory Pattern... 121
How a Simple Factory Works 121
Sample Code 122

Copyright © , 2002 by James W Cooper

The Two Derived Classes 122
Building the Simple Factory 123
Using the Factory 124
Factory Patterns in Math Computation 125
Programs on the CD-ROM 128
Thought Questions 128
9. The Factory Method 129
The Swimmer Class 132
The Events Classes 132
Straight Seeding 133
Circle Seeding 134
Our Seeding Program 134
Other Factories 135
When to Use a Factory Method 136
Thought Question 136
Programs on the CD-ROM 136
10. The Abstract Factory Pattern 137
A GardenMaker Factory 137
The PictureBox 141
Handling the RadioButton and Button Events 142
Adding More Classes 143
Consequences of Abstract Factory 144
Thought Question 144

Copyright © , 2002 by James W Cooper

Programs on the CD-ROM 144

11. The Singleton Pattern 145

Creating Singleton Using a Static Method 145

Exceptions and Instances 146

Throwing the Exception........ 147

Creating an Instance of the Class 147

Providing a Global Point of Access to a Singleton........ 148

Other Consequences of the Singleton Pattern........ 149

Programs on Your CD-ROM 149

12. The Builder Pattern 150

An Investment Tracker 151

The Stock Factory 154

The CheckChoice Class 155

The ListboxChoice Class 156

Using the Items Collection in the ListBox Control 157

Plotting the Data 158

The Final Choice 159

Consequences of the Builder Pattern 160

Thought Questions 161

Programs on the CD-ROM 161

13. The Prototype Pattern 162

Cloning in C# 163

Using the Prototype 163

Copyright © , 2002 by James W Cooper

Cloning the Class 167
Using the Prototype Pattern 170
Dissimilar Classes with the Same Interface 172
Prototype Managers 176
Consequences of the Prototype Pattern 176
Thought Question 177
Programs on the CD-ROM 177
Summary of Creational Patterns 178
14. The Adapter Pattern 180
Moving Data Between Lists 180
Making an Adapter 182
Using the DataGrid 183
Detecting Row Selection 186
Using a TreeView 186
The Class Adapter 188
Two-Way Adapters 190
Object Versus Class Adapters in C# 190
Pluggable Adapters 191
Thought Question 191
Programs on the CD-ROM 191
15. The Bridge Pattern 192
The VisList Classes 195
The Class Diagram 196

Copyright © , 2002 by James W Cooper

Extending the Bridge 197
Windows Forms as Bridges 201
Consequences of the Bridge Pattern 202
Thought Question.. 203
Programs on the CD-ROM 203
16. The Composite Pattern... 204
An Implementation of a Composite 205
Computing Salaries.. 206
The Employee Classes 206
The Boss Class... 209
Building the Employee Tree 210
Self-Promotion... 213
Doubly Linked Lists 213
Consequences of the Composite Pattern.. 215
A Simple Composite 215
Composites in .NET... 216
Other Implementation Issues 216
Thought Questions 216
Programs on the CD-ROM 217
17. The Decorator Pattern.. 218
Decorating a CoolButton 218
Handling events in a Decorator... 220
Layout Considerations 221

Copyright © , 2002 by James W Cooper

Control Size and Position 221
Multiple Decorators 222
Nonvisual Decorators 225
Decorators, Adapters, and Composites 226
Consequences of the Decorator Pattern 226
Thought Questions 226
Programs on the CD-ROM 227
18. The Façade Pattern 228
What Is a Database? 228
Getting Data Out of Databases 230
Kinds of Databases 231
ODBC 232
Database Structure 232
Using ADO.NET 233
Connecting to a Database 233
Reading Data from a Database Table 234
`dtable = dset.Tables [0];` 235
Executing a Query 235
Deleting the Contents of a Table 235
Adding Rows to Database Tables Using ADO.NET 236
Building the Façade Classes 237
Building the Price Query 239
Making the ADO.NET Façade 239

Copyright © , 2002 by James W Cooper

The DBTable class 242
Creating Classes for Each Table 244
Building the Price Table 246
Loading the Database Tables 249
The Final Application 251
What Constitutes the Façade? 252
Consequences of the Façade 253
Thought Question.. 253
Programs on the CD-ROM 253
19. The Flyweight Pattern 254
Discussion.. 255
Example Code 256
The Class Diagram.. 261
Selecting a Folder.. 261
Handling the Mouse and Paint Events 263
Flyweight Uses in C# 264
Sharable Objects 265
Copy-on-Write Objects.. 265
Thought Question.. 266
Programs on the CD-ROM 266
20. The Proxy Pattern 267
Sample Code 268
Proxies in C# 270

Copyright © , 2002 by James W Cooper

Copy-on-Write 271
Comparison with Related Patterns 271
Thought Question... 271
Programs on the CD-ROM 271
21. Chain of Responsibility 274
Applicability.. 275
Sample Code 276
The List Boxes 280
Programming a Help System 282
Receiving the Help Command 286
A Chain or a Tree? 287
Kinds of Requests 289
Examples in C# 289
Consequences of the Chain of Responsibility 290
Thought Question... 290
Programs on the CD-ROM 291
22. The Command Pattern 292
Motivation.. 292
Command Objects 293
Building Command Objects... 294
Consequences of the Command Pattern 297
The CommandHolder Interface 297
Providing Undo 301

Copyright © , 2002 by James W Cooper

Thought Questions 309
Programs on the CD-ROM 310
23. The Interpreter Pattern.. 311
Motivation.. 311
Applicability... 311
A Simple Report Example 312
Interpreting the Language 314
Objects Used in Parsing... 315
Reducing the Parsed Stack... 319
Implementing the Interpreter Pattern.. 321
The Syntax Tree 322
Consequences of the Interpreter Pattern.. 326
Thought Question.. 327
Programs on the CD-ROM 327
24. The Iterator Pattern... 328
Motivation.. 328
Sample Iterator Code 329
Fetching an Iterator 330
Filtered Iterators.. 331
The Filtered Iterator.. 331
Keeping Track of the Clubs 334
Consequences of the Iterator Pattern... 335
Programs on the CD-ROM 336

Copyright © , 2002 by James W Cooper

25. The Mediator Pattern 337
An Example System.. 337
Interactions Between Controls 339
Sample Code 341
Initialization of the System 345
Mediators and Command Objects.. 345
Consequences of the Mediator Pattern.. 347
Single Interface Mediators 348
Implementation Issues 349
Programs on the CD-ROM 349
26. The Memento Pattern 350
Motivation... 350
Implementation 351
Sample Code 351
A Cautionary Note 358
Command Objects in the User Interface 358
Handling Mouse and Paint Events 360
Consequences of the Memento 361
Thought Question.. 361
Programs on the CD-ROM 362
27. The Observer Pattern 363
Watching Colors Change 364
The Message to the Media 367

Copyright © , 2002 by James W Cooper

Consequences of the Observer Pattern.. 368
Programs on the CD-ROM 369
28. The State Pattern 370
Sample Code 370
Switching Between States 376
How the Mediator Interacts with the State Manager 377
The ComdToolBarButton 378
Handling the Fill State 381
Handling the Undo List.. 382
The VisRectangle and VisCircle Classes... 385
Mediators and the God Class 387
Consequences of the State Pattern... 388
State Transitions 388
Thought Questions 389
Programs on the CD-ROM 389
29. The Strategy Pattern.. 390
Motivation.. 390
Sample Code 391
The Context.. 392
The Program Commands 393
The Line and Bar Graph Strategies.. 394
Drawing Plots in C#... 394
Making Bar Plots 395

Copyright © , 2002 by James W Cooper

Making Line Plots 396
Consequences of the Strategy Pattern.. 398
Programs on the CD-ROM 398
30. The Template Method Pattern 399
Motivation.. 399
Kinds of Methods in a Template Class 401
Sample Code 402
Drawing a Standard Triangle 404
Drawing an Isosceles Triangle 404
The Triangle Drawing Program... 405
Templates and Callbacks 406
Summary and Consequences 407
Programs on the CD-ROM 408
31. The Visitor Pattern 409
Motivation.. 409
When to Use the Visitor Pattern 411
Sample Code 411
Visiting the Classes 413
Visiting Several Classes... 414
Bosses Are Employees, Too 416
Catch-All Operations with Visitors 417
Double Dispatching.. 419
Why Are We Doing This? 419

Copyright © , 2002 by James W Cooper

Traversing a Series of Classes 419
Consequences of the Visitor Pattern.. 420
Thought Question... 420
Programs on the CD-ROM 421
32. Bibliography 422

Copyright © , 2002 by James W Cooper

Preface

This is a practical book that tells you how to write C# programs using some of the most common design patterns. It also serves as a quick introduction to programming in the new C# language. The pattern discussions are structured as a series of short chapters, each describing a design pattern and giving one or more complete working, visual example programs that use that pattern. Each chapter also includes UML diagrams illustrating how the classes interact.

This book is not a "companion" book to the well-known *Design Patterns* text. by the "Gang of Four." Instead, it is a tutorial for people who want to learn what design patterns are about and how to use them in their work. You do not have to have read *Design Patterns* to read this book, but when you are done here you may well want to read or reread it to gain additional insights.

In this book, you will learn that design patterns are frequently used ways of organizing objects in your programs to make them easier to write and modify. You'll also see that by familiarizing yourself with them, you've gained some valuable vocabulary for discussing how your programs are constructed.

People come to appreciate design patterns in different ways—from the highly theoretical to the intensely practical—and when they finally see the great power of these patterns, an "Aha!" moment occurs. Usually this moment means that you suddenly have an internal picture of how that pattern can help you in your work.

In this book, we try to help you form that conceptual idea, or *gestalt*, by describing the pattern in as many ways as possible. The book is organized into six main sections: an introductory description, an introduction to C#, and descriptions of patterns, grouped as creational, structural, and behavioral.

Copyright © , 2002 by James W Cooper

For each pattern, we start with a brief verbal description and then build simple example programs. Each of these examples is a visual program that you can run and examine to make the pattern as concrete a concept as possible. All of the example programs and their variations are on the companion CD-ROM, where you run them, change them, and see how the variations you create work.

Since each of the examples consists of a number of C# files for each of the classes we use in that example, we provide a C# project file for each example and place each example in a separate subdirectory to prevent any confusion. This book is based on the Beta-2 release of Visual Studio.Net. Any changes between this version and the final product will probably not be great. Consult the Addison-Wesley website for updates to any example code.

If you leaf through the book, you'll see screen shots of the programs we developed to illustrate the design patterns, providing yet another way to reinforce your learning of these patterns. In addition, you'll see UML diagrams of these programs, illustrating the interactions between classes in yet another way. UML diagrams are just simple box and arrow illustrations of classes and their inheritance structure, where arrows point to parent classes, and dotted arrows point to interfaces. And if you're not yet familiar with UML, we provide a simple introduction in the second chapter.

When you finish this book, you'll be comfortable with the basics of design patterns and will be able to start using them in your day-to-day C# programming work.

James W. Cooper

Nantucket, MA

Wilton, CT

Kona, HI

Copyright © , 2002 by James W Cooper

1. What are Design Patterns?

Sitting at your desk in front of your workstation, you stare into space, trying to figure out how to write a new program feature. You know intuitively what must be done, what data and what objects come into play, but you have this underlying feeling that there is a more elegant and general way to write this program.

In fact, you probably don't write any code until you can build a picture in your mind of what the code does and how the pieces of the code interact. The more that you can picture this "organic whole," or *gestalt*, the more likely you are to feel comfortable that you have developed the best solution to the problem. If you don't grasp this whole right away, you may keep staring out the window for a time, even though the basic solution to the problem is quite obvious.

In one sense you feel that the more elegant solution will be more reusable and more maintainable, but even if you are the sole likely programmer, you feel reassured once you have designed a solution that is relatively elegant and that doesn't expose too many internal inelegancies.

One of the main reasons that computer science researchers began to recognize design patterns is to satisfy this need for elegant, but simple, reusable solutions. The term "design patterns" sounds a bit formal to the uninitiated and can be somewhat offputting when you first encounter it. But, in fact, design patterns are just convenient ways of reusing object-oriented code between projects and between programmers. The idea behind design patterns is simple—write down and catalog common interactions between objects that programmers have frequently found useful.

One of the frequently cited patterns from early literature on programming frameworks is the Model-View-Controller framework for Smalltalk (Krasner and Pope 1988), which divided the user interface problem into three parts, as shown in Figure 1-1. The parts were referred to as a *data*

Copyright © , 2002 by James W Cooper

model, which contains the computational parts of the program; the *view,* which presented the user interface; and the c*ontroller*, which interacted between the user and the view.

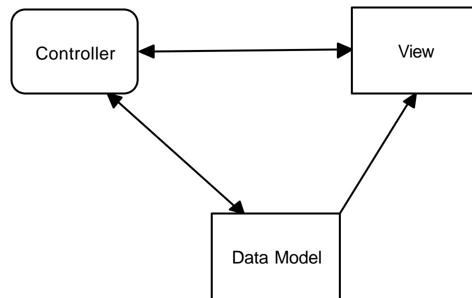


Figure 1-1 – The Model-View-Controller framework

Each of these aspects of the problem is a separate object, and each has its own rules for managing its data. Communication among the user, the GUI, and the data should be carefully controlled, and this separation of functions accomplished that very nicely. Three objects talking to each other using this restrained set of connections is an example of a powerful design pattern.

In other words, design patterns describe how objects communicate without become entangled in each other's data models and methods. Keeping this separation has always been an objective of good OO programming, and if you have been trying to keep objects minding their own business, you are probably using some of the common design patterns already.

Design patterns began to be recognized more formally in the early 1990s by Erich Gamma (1992), who described patterns incorporated in the GUI application framework, ET++. The culmination of these discussions and a number of technical meetings was the publication of the parent book in this series, *Design Patterns—Elements of Reusable Software,* by Gamma, Helm, Johnson, and Vlissides (1995). This book, commonly referred to as the Gang of Four, or "GoF," book, has had a powerful impact on those seeking to understand how to use design patterns and has become an all-

Copyright © , 2002 by James W Cooper

time bestseller. It describes 23 commonly occurring and generally useful patterns and comments on how and when you might apply them. We will refer to this groundbreaking book as *Design Patterns* throughout this book.

Since the publication of the original *Design Patterns* text, there have been a number of other useful books published. One closely related book is *The Design Patterns Smalltalk Companion (*Alpert, Brown, and Woolf 1998), which covers the same 23 patterns from the Smalltalk point of view. We'll refer to this book throughout as the *Smalltalk Companion.* Finally, we recently published *Java Design Patterns: a Tutorial,* and *Visual Basic Design Patterns,* which illustrate all of these patterns in those languages.

Defining Design Patterns

We all talk about the way we do things in our jobs, hobbies, and home life, and we recognize repeating patterns all the time.

- Sticky buns are like dinner rolls, but I add brown sugar and nut filling to them.
- Her front garden is like mine, but I grow astilbe in my garden.
- This end table is constructed like that one, but in this one, there are doors instead of drawers.

We see the same thing in programming when we tell a colleague how we accomplished a tricky bit of programming so he doesn't have to recreate it from scratch. We simply recognize effective ways for objects to communicate while maintaining their own separate existences.

Some useful definitions of design patterns have emerged as the literature in this field has expanded.

- "Design patterns are recurring solutions to design problems you see over and over." (*The Smalltalk Companion)*

Copyright © , 2002 by James W Cooper

- “Design patterns constitute a set of rules describing how to accomplish certain tasks in the realm of software development.” (Pree 1994)
- “Design patterns focus more on reuse of recurring architectural design themes, while frameworks focus on detailed design and implementation.” (Coplien and Schmidt 1995)
- “A pattern addresses a recurring design problem that arises in specific design situations and presents a solution to it.” (Buschmann et al. 1996)
- “Patterns identify and specify abstractions that are above the level of single classes and instances, or of components.” (Gamma et al., 1993)

But while it is helpful to draw analogies to architecture, cabinet making, and logic, design patterns are not just about the design of objects but about the *interaction* between objects. One possible view of some of these patterns is to consider them as *communication patterns*.

Some other patterns deal not just with object communication but with strategies for object inheritance and containment. It is the design of simple, but elegant, methods of interaction that makes many design patterns so important.

Design patterns can exist at many levels from very low-level specific solutions to broadly generalized system issues. There are now hundreds of patterns in the literature. They have been discussed in articles and at conferences of all levels of granularity. Some are examples that apply widely, and a few writers have ascribed pattern behavior to class groupings that apply to just a single problem (Kurata 1998).

It has become apparent that you don’t just *write* a design pattern off the top of your head. In fact, most such patterns are *discovered* rather than written. The process of looking for these patterns is called “pattern mining,” and it is worthy of a book of its own.

The 23 design patterns selected for inclusion in the original *Design Patterns* book were those that had several known applications and that

Copyright © , 2002 by James W Cooper

were on a middle level of generality, where they could easily cross application areas and encompass several objects.

The authors divided these patterns into three types: creational, structural, and behavioral.

- *Creational patterns* create objects for you rather than having you instantiate objects directly. This gives your program more flexibility in deciding which objects need to be created for a given case.
- *Structural patterns* help you compose groups of objects into larger structures, such as complex user interfaces or accounting data.
- *Behavioral patterns* help you define the communication between objects in your system and how the flow is controlled in a complex program.

We'll be looking at C# versions of these patterns in the chapters that follow, and we will provide at least one complete C# program for each of the 23 patterns. This way you can examine the code snippets we provide and also run, edit, and modify the complete working programs on the accompanying CD-ROM. You'll find a list of all the programs on the CD-ROM at the end of each pattern description.

The Learning Process

We have found that regardless of the language, learning design patterns is a multiple-step process.

1. Acceptance
2. Recognition
3. Internalization

First, you accept the premise that design patterns are important in your work. Then, you recognize that you need to read about design patterns in order to know when you might use them. Finally, you internalize the

Copyright © , 2002 by James W Cooper

patterns in sufficient detail that you know which ones might help you solve a given design problem.

For some lucky people, design patterns are obvious tools, and these people can grasp their essential utility just by reading summaries of the patterns. For many of the rest of us, there is a slow induction period after we've read about a pattern followed by the proverbial "Aha!" when we see how we can apply them in our work. This book helps to take you to that final stage of internalization by providing complete, working programs that you can try out for yourself.

The examples in *Design Patterns* are brief and are in C++ or, in some cases, Smalltalk. If you are working in another language, it is helpful to have the pattern examples in your language of choice. This book attempts to fill that need for C# programmers.

Studying Design Patterns

There are several alternate ways to become familiar with these patterns. In each approach, you should read this book and the parent *Design Patterns* book in one order or the other. We also strongly urge you to read the *Smalltalk Companion* for completeness, since it provides alternative descriptions of each of the patterns. Finally, there are a number of Web sites on learning and discussing design patterns for you to peruse.

Notes on Object-Oriented Approaches

The fundamental reason for using design patterns is to keep classes separated and prevent them from having to know too much about one another. Equally important, using these patterns helps you avoid reinventing the wheel and allows you to describe your programming approach succinctly in terms other programmers can easily understand.

There are a number of strategies that OO programmers use to achieve this separation, among them encapsulation and inheritance. Nearly all languages that have OO capabilities support inheritance. A class that inherits from a parent class has access to all of the methods of that parent

Copyright © , 2002 by James W Cooper

class. It also has access to all of its nonprivate variables. However, by starting your inheritance hierarchy with a complete, working class, you may be unduly restricting yourself as well as carrying along specific method implementation baggage. Instead, *Design Patterns* suggests that you always

Program to an interface and not to an implementation.

Putting this more succinctly, you should define the top of any class hierarchy with an *abstract* class or an *interface*, which implements no methods but simply defines the methods that class will support. Then in all of your derived classes you have more freedom to implement these methods as most suits your purposes. And since C#6 only supports interfaces and does not support inheritance, this is obviously very good advice in the C# context.

The other major concept you should recognize is that of *object composition.* This is simply the construction of objects that contain others: encapsulation of several objects inside another one. While many beginning OO programmers use inheritance to solve every problem, as you begin to write more elaborate programs, you will begin to appreciate the merits of object composition.Your new object can have the interface that is best for what you want to accomplish without having all the methods of the parent classes. Thus, the second major precept suggested by *Design Patterns* is

Favor object composition over inheritance.

C# Design Patterns

Each of the 23 patterns in *Design Patterns* is discussed, at least one working program example for that pattern is supplied. All of the programs have some sort of visual interface to make them that much more immediate to you. All of them also use class, interfaces, and object composition, but the programs themselves are of necessity quite simple so that the coding doesn’t obscure the fundamental elegance of the patterns we are describing.

Copyright © , 2002 by James W Cooper

However, even though C# is our target language, this isn't specifically a book on the C# language. There are lots of features in C# that we don't cover, but we do cover most of what is central to C#. You will find, however, that this is a fairly useful tutorial in object-oriented programming in C# and provides good overview of how to program in C#.NET.

How This Book Is Organized

We take up each of the 23 patterns, grouped into the general categories of creational, structural, and behavioral patterns. Many of the patterns stand more or less independently, but we do take advantage of already discussed patterns from time to time. For example, we use the Factory and Command patterns extensively after introducing them, and we use the Mediator pattern several times after we introduce it. We use the Memento again in the State pattern, the Chain of Responsibility in the Interpreter pattern discussion, and the Singleton pattern in the Flyweight pattern discussion. In no case do we use a pattern before we have introduced it formally.

We also take some advantage of the sophistication of later patterns to introduce new features of C#. For example, the Listbox, DataGrid, and TreeView are introduced in the Adapter and Bridge patterns. We show how to paint graphics objects in the Abstract Factory, We introduce the Enumeration interface in the Iterator and in the Composite, where we also take up formatting. We use exceptions in the Singleton pattern and discuss ADO.NET database connections in the Façade pattern. And we show how to use C# timers in the Proxy pattern.

The overall .NET system is designed for fairly elaborate web-based client-server interactions. However, in this book, concentrate on object-oriented programming issues in general rather than how to write Web-based systems. We cover the core issues of C# programming and show simple examples of how Design Patterns can help write better programs.

Copyright © , 2002 by James W Cooper

2. Syntax of the C# Language

C# has all the features of any powerful, modern language. If you are familiar with Java, C or C++, you'll find most of C#'s syntax very familiar. If you have been working in Visual Basic or related areas, you should read this chapter to see how C# differs from VB. You'll quickly see that every major operation you can carry out in Visual Basic.NET has a similar operation in C#.

The two major differences between C# and Visual Basic are that C# is *case sensitive* (most of its syntax is written in *lowercase)* and that every statement in C# is terminated with a semicolon (;). Thus C# statements are not constrained to a single line and there is no line continuation character.

In Visual Basic, we could write:

```
y = m * x + b       'compute y for given x
```

or we could write:

```
Y = M * X + b       'compute y for given x
```

and both would be treated as the same. The variables *Y*, *M*, and *X* are the same whether written in upper- or lowercase. In C#, however, case is significant, and if we write:

```
y = m * x + b;             //all lowercase
```

or:

```
Y = m * x + b;             //Y differs from y
```

we mean two different variables: *Y* and *y*. While this may seem awkward at first, having the ability to use case to make distinctions is sometimes very useful. For example, programmers often capitalize symbols referring to constants:

```
Const PI = 3.1416 As Single       ' in VB
const float PI = 3.1416;          // in C#
```

Copyright © , 2002 by James W Cooper

The *const* modifier in C# means that the named value is a constant and cannot be modified.

Programmers also sometimes define data types using mixed case and variables of that data type in lowercase:

```
class Temperature {         //begin definition of
                            //new data type
Temperature temp;           //temp is of this new type
```

We'll classes in much more detail in the chapters that follow.

Data Types

The major data types in C# are shown in Table 2-1.

Table 2-1 - Data types in C#

bool	true or false
byte	unsigned 8-bit value
short	16-bit integer
int	32-bit integer
long	64-bit integer
float	32-bit floating point
double	64-bit floating point
char	16-bit character
string	16-bit characters

Note that the lengths of these basic types are irrespective of the computer type or operating system. Characters and strings in C# are always 16 bits wide: to allow for representation of characters in non-Latin languages. It uses a character coding system called Unicode, in which thousands of characters for most major written languages have been defined. You can convert between variable types in the usual simple ways:

Copyright © , 2002 by James W Cooper

- Any wider data type can have a narrower data type (having fewer bytes) assigned directly to it, and the promotion to the new type will occur automatically. If *y* is of type float and *j* is of type int, then you can write:

```
float y = 7.0f;       //y is of type float
int j;                //j is of type int
y = j;                //convert int to float
```

to promote an integer to a float.

- You can reduce a wider type (more bytes) to a narrower type by *casting* it. You do this by putting the data type name in parentheses and putting it in front of the value you wish to convert:

```
    j = (int)y;       //convert float to integer
```

You can also write legal statements that contain casts that might fail, such as

```
float x = 1.0E45;
int k = (int) x;
```

If the cast fails, an exception error will occur when the program is executed.

Boolean variables can only take on the values represented by the reserved words *true* and *false*. Boolean variables also commonly receive values as a result of comparisons and other logical operations:

```
int k;
boolean gtnum;

gtnum = (k > 6);     //true if k is greater than 6
```

Unlike C or C++, you cannot assign numeric values to a boolean variable and you cannot convert between boolean and any other type.

Copyright © , 2002 by James W Cooper

Converting Between Numbers and Strings

To make a string from a number or a number from a string, you can use the Convert methods. You can usually find the right one by simply typing Convert and a dot in the development enviroment, and the system will provide you with a list of likely methods.

```
string s = Convert.ToString (x);
float y = Convert.ToSingle (s);
```

Note that "Single" means a single-precision floating point number.

Numeric objects also provide various kinds of formatting options to specify the number of decimal places:

```
float x = 12.341514325f;

string s =x.ToString ("###.###");        //gives 12.342
```

Declaring Multiple Variables

You should note that in C#, you can declare a number of variables of the same type in a single statement:

```
int i, j;
float x, y, z;
```

This is unlike VB6, where you had to specify the type of each variable as you declare it:

```
Dim i As Integer, j As Integer
Dim x As Single, y As Single, z As Single
```

Numeric Constants

Any number you type into your program is automatically of type int if it has no fractional part or type *double* if it does. If you want to indicate that it is a different type, you can use various suffix and prefix characters:

```
float loan = 1.23f;        //float
long pig   = 45L;          //long
int color = 0x12345;       //hexadecimal
```

Copyright © , 2002 by James W Cooper

C# also has three reserved word constants: *true, false,* and *null*, where *null* means an object variable that does not yet refer to any object. We'll learn more about objects in the following chapters

Character Constants

You can represent individual characters by enclosing them in single quotes:

```
char c = 'q';
```

C# follows the C convention that the *white space characters* (non printing characters that cause the printing position to change) can be represented by preceding special characters with a backslash, as shown in Table 2-2. Since the backslash itself is thus a special character, it can be represented by using a double backslash

`'\n'`	newline (line feed)
`'\r'`	carriage return
`'\t'`	tab character
`'\b'`	backspace
`'\f'`	form feed
`'\0'`	null character
`'\"'`	double quote
`'\''`	single quote
`'\\'`	backslash

Table 2-2 ***Representations of white space and special characters.***

Variables

Variable names in C# can be of any length and can be of any combination of upper- and lowercase letters and numbers, but like VB, the first character must be a letter. Note that since case is significant in C#, the following variable names all refer to different variables:

```
temperature
```

Copyright © , 2002 by James W Cooper

```
Temperature
TEMPERATURE
```

You must declare all C# variables that you use in a program before you use them:

```
int j;
float temperature;
boolean quit;
```

Declaring Variables as You Use Them

C# also allows you to declare variables just as you need them rather than requiring that they be declared at the top of a procedure:

```
int k = 5;
float x = k + 3 * y;
```

This is very common in the object-oriented programming style, where we might declare a variable inside a loop that has no existence or *scope* outside that local spot in the program.

Multiple Equals Signs for Initialization

C#, like C, allows you to initialize a series of variables to the same value in a single statement

```
i = j = k = 0;
```

This can be confusing, so don't overuse this feature. The compiler will generate the same code for:

```
i = 0; j = 0; k = 0;
```

whether the statements are on the same or successive lines.

A Simple C# Program

Now let's look at a very simple C# program for adding two numbers together. This program is a stand-alone program, or application.

```
using System;
class add2
       {
```

Copyright © , 2002 by James W Cooper

```
            static void Main(string[] args)
            {
                  double a, b, c;   //declare variables
                  a = 1.75;         //assign values
                  b = 3.46;
                  c = a + b;        //add together
                  //print out sum
                  Console.WriteLine ("sum = " + c);
            }
      }
```

This is a complete program as it stands, and if you compile it with the C# compiler and run it, it will print out the result:

```
sum = 5.21
```

Let's see what observations we can make about this simple program: This is the way I want it.

1. You must use the *using* statement to define libraries of C# code that you want to use in your program. This is similar to the imports statement in VB, and similar to the C and C++ #include directive.

2. The program starts from a function called main and it must have *exactly* the form shown here:

   ```
   static void Main(string[] args)
   ```

3. Every program module must contain one or more classes.

4. The class and each function within the class is surrounded by *braces* { }.

5. Every variable must be declared by type before or by the time it is used. You could just as well have written:

```
double a = 1.75;
double b = 3.46;
double c = a + b;
```

Copyright © , 2002 by James W Cooper

6. Every statement must terminate with a semicolon. Statements can go on for several lines but they must terminate with the semicolon.

7. Comments start with // and terminate at the end of the line.

8. Like most other languages (except Pascal), the equals sign is used to represent assignment of data.

9. You can use the + sign to combine two strings. The string "sum =" is concatenated with the string automatically converted from the double precision variable c.

10. The writeLine function, which is a member of the Console class in the System namespace, can be used to print values on the screen.

Compiling & Running This Program

This simple program is called add2.cs. You can compile and execute it by in the development enviroment by just pressing F5.

Arithmetic Operators

The fundamental operators in C# are much the same as they are in most other modern languages. Table 2-3 lists the fundamental operators in C#

+	addition
-	subtraction, unary minus
*	multiplication
/	division
%	modulo (remainder after integer division)

Table 2-3*: C# arithmetic operators*

The bitwise and logical operators are derived from C rather (see Table 2-4). *Bitwise operators* operate on individual bits of two words, producing a result based on an AND, OR or NOT operation. These are distinct from the Boolean operators, because they operate on a logical condition which evaluates to *true* or *false.*

Copyright © , 2002 by James W Cooper

&	bitwise And
|	bitwise Or
^	bitwise exclusive Or
~	one's complement
>> *n*	right shift *n* places
<< *n*	left shift *n* places

Table 2-4 ***Logical Operators in C#***

Increment and Decrement Operators

Like Java and C/C++ , C# allows you to express incrementing and decrementing of integer variables using the ++and -- operators. You can apply these to the variable before or after you use it:

```
i = 5;
j = 10;
x = i++;      //x = 5, then i = 6
y = --j;      //y = 9 and j = 9
z = ++i;      //z = 7 and i = 7
```

Combining Arithmetic and Assignment Statements

C# allows you to combine addition, subtraction, multiplication, and division with the assignment of the result to a new variable:

```
x = x + 3;          //can also be written as:
x += 3;             //add 3 to x; store result in x

//also with the other basic operations:
temp *= 1.80;       //mult temp by 1.80
z -= 7;             //subtract 7 from z
y /= 1.3;           //divide y by 1.3
```

This is used primarily to save typing; it is unlikely to generate any different code. Of course, these compound operators (as well as the ++ and – operators) cannot have spaces between them.

Copyright © , 2002 by James W Cooper

Making Decisions in C#

The familiar if-then-else of Visual Basic, Pascal and Fortran has its analog in C#. Note that in C#, however, we do not use the *then* keyword:

```
if ( y > 0 )
  z = x / y;
```

Parentheses around the condition are *required* in C#. This format can be somewhat deceptive; as written, only the single statement following the if is operated on by the if statement. If you want to have several statements as part of the condition, you must enclose them in braces:

```
if ( y > 0 )
  {
  z = x / y;
  Console.writeLine("z =  " + z);
  }
```

By contrast, if you write:

```
if ( y > 0 )
  z = x / y;
  Console.writeLine("z =  " + z);
```

the C# program will always print out z= and some number, because the if clause only operates on the single statement that follows. As you can see, indenting does not affect the program; it does what you say, not what you mean.

If you want to carry out either one set of statements or another depending on a single condition, you should use the else clause along with the if statement:

```
if ( y > 0 )
  z = x / y;
else
  z = 0;
```

and if the else clause contains multiple statements, they must be enclosed in braces, as in the code above.

Copyright © , 2002 by James W Cooper

There are two or more accepted indentation styles for braces in C# programs:

```
if (y >0 )
       {
       z = x / y;
       }
```

The other style, popular among C programmers, places the brace at the end of the if statement and the ending brace directly under the if:

```
if ( y > 0 ) {
  z = x / y;
  Console.writeLine("z=" + z);
}
```

You will see both styles widely used, and of course, they compile to produce the same result.

Comparison Operators

Above, we used the > operator to mean "greater than." Most of these operators are the same in C# as they are in C and other languages. In Table 2-5, note particularly that "is equal to" requires *two* equal signs and that "not equal" is different than in FORTRAN or VB.

Operator	Meaning
`>`	greater than
`<`	less than
`==`	is equal to
`!=`	is not equal to
`>=`	greater than or equal to
`<=`	less than or equal to

Table 2-5***: Comparison Operators in C#***

Combining Conditions

When you need to combine two or more conditions in a single if or other logical statement, you use the symbols for the logical and, or, and not operators (see Table 3-6). These are totally different than any other

Copyright © , 2002 by James W Cooper

languages except C/C++ and are confusingly like the bitwise operators shown in Table 2-6.

`&&`	logical And
`||`	logical Or
`~`	logical Not

Table 2-6 ***Boolean operators in C#***

So, while in VB.Net we would write:

```
If ( 0 < x) And (x <= 24) Then
  Console.writeLine ("Time is up")
```

in C# we would write:

```
if ( (0 < x) && ( x <= 24) )
  Console.writeLine("Time is up");
```

The Most Common Mistake

Since the is equal to operator is `==` and the assignment operator is `=` they can easily be misused. If you write

```
if (x = 0)
  Console.writeLine("x is zero");
```

instead of:

```
if (x == 0)
  Console.writeLine("x is zero");
```

you will get the confusing compilation error, "Cannot implcitly convert double to bool," because the result of the fragment:

```
 (x = 0)
```

is the double precision number 0, rather than a Boolean true or false. Of course, the result of the fragment:

```
(x == 0)
```

is indeed a Boolean quantity and the compiler does not print any error message.

Copyright © , 2002 by James W Cooper

The switch Statement

The switch statement allows you to provide a list of possible values for a variable and code to execute if each is true. In C#, however, the variable you compare in a switch statement must be either an integer or a character type and must be enclosed in parentheses:

```
       switch ( j ) {
  case 12:
    System.out.println("Noon");
    break;
  case 13:
    System.out.println("1 PM");   "
    break;
  default:
    System.out.println("some other time...");
}
```

Note particularly that a *break* statement *must* follow each case in the switch statement. This is very important, as it says "go to the end of the switch statement." If you leave out the break statement, the code in the next case statement is executed as well.

C# Comments

As you have already seen, comments in C# start with a double forward slash and continue to the end of the current line. C# also recognizes C-style comments which begin with /* and continue through any number of lines until the */ symbols are found.

```
//C# single-line comment
/*other C# comment style*/
/* also can go on
for any number of lines*/
```

You can't nest C# comments; once a comment begins in one style it continues until that style concludes.

Your initial reaction as you are learning a new language may be to ignore comments, but they are just as important at the outset as they are later. A program never gets commented at all unless you do it as you write it, and

Copyright © , 2002 by James W Cooper

if you ever want to use that code again, you'll find it very helpful to have some comments to help you in deciphering what you meant for it to do. For this reason, many programming instructors refuse to accept programs that are not thoroughly commented.

The Ornery Ternary Operator

C# has unfortunately inherited one of C/C++ and Java's most opaque constructions, the ternary operator. The statement:

```
if ( a > b )
  z = a;
else
  z = b;
```

can be written extremely compactly as:

```
z = (a > b) ? a : b;
```

The reason for the original introduction of this statement into the C language was, like the post-increment operators, to give hints to the compiler to allow it to produce more efficient code, and to reduce typing when terminals were very slow. Today, modern compilers produce identical code for both forms given above, and the necessity for this turgidity is long gone. Some C programmers coming to C# find this an "elegant" abbreviation, but we don't agree and will not be using it in this book.

Looping Statements in C#

C# has four looping statements: while, do-while, for and foreach. Each of them provides ways for you to specify that a group of statements should be executed until some condition is satisfied.

The while Loop

The while loop is easy to understand. All of the statements inside the braces are executed repeated as long as the condition is true.

```
i = 0;
while ( i < 100)
```

Copyright © , 2002 by James W Cooper

```
  {
   x = x + i++;
  }
```

Since the loop is executed as long as the condition is true, it is possible that such a loop may never be executed at all, and of course, if you are not careful, that such a while loop will never be completed.

The do-while Statement

The C# do-while statement is quite analogous, except that in this case the loop must always be executed at least once, since the test is at the bottom of the loop:

```
i = 0;
do  {
   x += i++;
}
while (i < 100);
```

The for Loop

The for loop is the most structured. It has three parts: an initializer, a condition, and an operation that takes place each time through the loop. Each of these sections are separated by semicolons:

```
for (i = 0; i< 100; i++)  {
   x += i;
  }
```

Let's take this statement apart:

```
for (i = 0;          //initialize i to 0
    i < 100 ; //continue as long as i < 100
    i++)             //increment i after every pass
```

In the loop above, *i* starts the first pass through the loop set to zero. A test is made to make sure that *i* is less than 100 and then the loop is executed. After the execution of the loop, the program returns to the top, increments *i* and again tests to see if it is less than 100. If it is, the loop is again executed.

Copyright © , 2002 by James W Cooper

Note that this for loop carries out exactly the same operations as the while loop illustrated above. It may never be executed and it is possible to write a for loop that never exits.

Declaring Variables as Needed in For Loops

One very common place to declare variables on the spot is when you need an iterator variable for a for loop. You can simply declare that variable right in the for statement, as follows:

```
for (int i = 0; i < 100; i++)
```

Such a loop variable exists or has *scope* only within the loop. It vanishes once the loop is complete. This is important because any attempt to reference such a variable once the loop is complete will lead to a compiler error message. The following code is incorrect:

```
for (int i =0; i< 5; i++) {
   x[i] = i;
}

//the following statement is in error
//because i is now out of scope
System.out.println("i=" + i);
```

Commas in for Loop Statements

You can initialize more than one variable in the initializer section of the C# for statement, and you can carry out more than one operation in the operation section of the statement. You separate these statements with commas:

```
for (x=0, y= 0, i =0; i < 100; i++, y +=2)
  {
   x = i + y;
  }
```

It has no effect on the loop's efficiency, and it is far clearer to write:

```
x = 0;
y = 0;
for ( i = 0; i < 100; i++)
  {
```

Copyright © , 2002 by James W Cooper

```
x = i + y;
y += 2;
}
```

It is possible to write entire programs inside an overstuffed for statement using these comma operators, but this is only a way of obfuscating the intent of your program.

How C# Differs From C

If you have been exposed to C, or if you are an experienced C programmer, you might be interested in the main differences between C# and C:

1. C# does not usually make use of pointers. You can only increment, or decrement a variable as if it were an actual memory pointer inside a special *unsafe* block.

2. You can declare variables anywhere inside a method you want to; they don't have to be at the beginning of the method.

3. You don't have to declare an object before you use it; you can define it just as you need it.

4. C# has a somewhat different definition of the struct types, and does not support the idea of a union at all.

5. C# has enumerated types, which allow a series of named values, such as colors or day names, to be assigned sequential numbers, but the syntax is rather different.

6. C# does not have bitfields: variables that take up less than a byte of storage.

7. C# does not allow variable length argument lists. You have to define a method for each number and type of argument. However

Copyright © , 2002 by James W Cooper

C# allows for the last argument of a function to be a variable parameter array.

Summary

In this brief chapter, we have seen the fundamental syntax elements of the C# language. Now that we understand the tools, we need to see how to use them. In the chapters that follow, we'll take up objects and show how to use them and how powerful they can be.

Copyright © , 2002 by James W Cooper

3. Writing Windows C# Programs

The C# language has its roots in C++, Visual Basic and Java. Both C# and VB.Net utilize the same libraries and compile to the same underlying code. Both are managed languages with garbage collection of unused variable space and both can be used interchangeably. Both also use classes with method names that are very similar to those in Java, so if you are familiar with Java, you will have no trouble with C#.

Objects in C#

In C#, everything is treated as an object. Objects contain data and have methods that operate on them. For example, strings are now objects. They have methods such as

```
Substring
ToLowerCase
ToUpperCase
IndexOf
Insert
```

and so forth.

Integers, float and double variables are also objects, and have methods.

```
string s;
float x;
x = 12.3;
s = x.ToString();
```

Note that conversion from numerical types is done using these methods rather than external functions. If you want to format a number as a particular kind of string, each numeric type has a Format method.

Copyright © , 2002 by James W Cooper

Managed Languages and Garbage Collection

C# and VB.Net are both *managed* languages. This has two major implications. First, both are compiled to an intermediate low-level language, and a common language runtime (CLR) is used to execute this compiled code, perhaps compiling it further first. So, not only do C# and VB.Net share the same runtime libraries, they are to a large degree two sides of the same coin and two aspects of the same language system. The differences are that VB7 is more Visual Basic like and a bit easier for VB programmers to learn and use. C# on the other hand is more C++ and Java-like, and may appeal more to programmers already experienced in those languages.

The other major implication is that managed languages are *garbage-collected*. Garbage collected languages take care of releasing unused memory: you never have to be concerned with this. As soon as the garbage collection system detects that there are no more active references to a variable, array or object, the memory is released back to the system. So you no longer need to worry as much about running out of memory because you allocated memory and never released it. Of course, it is still possible to write memory-eating code, but for the most part you do not have to worry about memory allocation and release problems.

Classes and Namespaces in C#

All C# programs are composed entirely of classes. Visual windows forms are a type of class, as we will see that all the program features we'll write are composed of classes. Since everything is a class, the number of names of class objects can get to be pretty overwhelming. They have therefore been grouped into various functional libraries that you must specifically mention in order to use the functions in these libraries.

Under the covers these libraries are each individual DLLs. However, you need only refer to them by their base names using the using statement, and the functions in that library are available to you.

```
using System;
```

Copyright © , 2002 by James W Cooper

```
using System.Drawing;
using System.Collections;
```

Logically, each of these libraries represents a different *namespace.* Each namespace is a separate group of class and method names which the compiler will recognize after you declare that name space. You can use namespaces that contain identically named classes or methods, but you will only be notified of a conflict if you try to use a class or method that is duplicated in more than one namespace.

The most common namespace is the System namespace, and it is imported by default without your needing to declare it. It contains many of the most fundamental classes and methods that C# uses for access to basic classes such as Application, Array, Console, Exceptions, Objects, and standard objects such as byte, bool, string. In the simplest C# program we can simply write a message out to the console without ever bringing up a window or form:

```
class Hello {
       static void Main(string[] args)   {
         Console.WriteLine ("Hello C# World");
       }
}
```

This program just writes the text "Hello C# World" to a command (DOS) window. The entry point of any program must be a Main method, and it must be declared as static.

Building a C# Application

Let's start by creating a simple console application: that is, one without any windows, that just runs from the command line. Start the Visual Studio.NET program, and select File |New Project. From the selection box, choose C# Console application as shown in Figure 3-1.

Copyright © , 2002 by James W Cooper

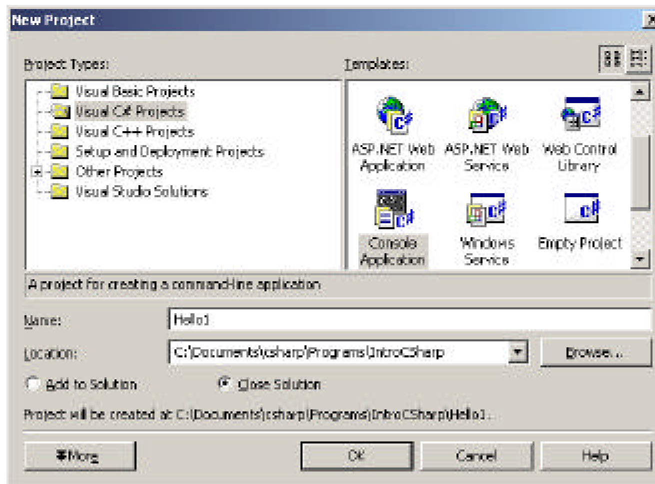


Figure 3-1 – The New Project selection window. Selecting a console application.

This will bring up a module, with Main already filled in. You can type in the rest of the code as follows:

```
Console.WriteLine ("Hello C# World");
```

You can compile this and run it by pressing F5.

When you compile and run the program by pressing F5, a DOS window will appear and print out the message “Hello C# World” and exit.

The Simplest Window Program in C#

C# makes it very easy to create Windows GUI programs. In fact, you can create most of it using the Windows Designer. To do this, start Visual Studio.NET and select File|New project, and select C# Windows Application. The default name (and filename) is WindowsApplication1, but you can change this before you close the New dialog box. This brings up a single form project, initially called Form1.cs. You can then use the Toolbox to insert controls, just as you can in Visual Basic.

Copyright © , 2002 by James W Cooper

The Windows Designer for a simple form with one text field and one button is shown in Figure 3-2.

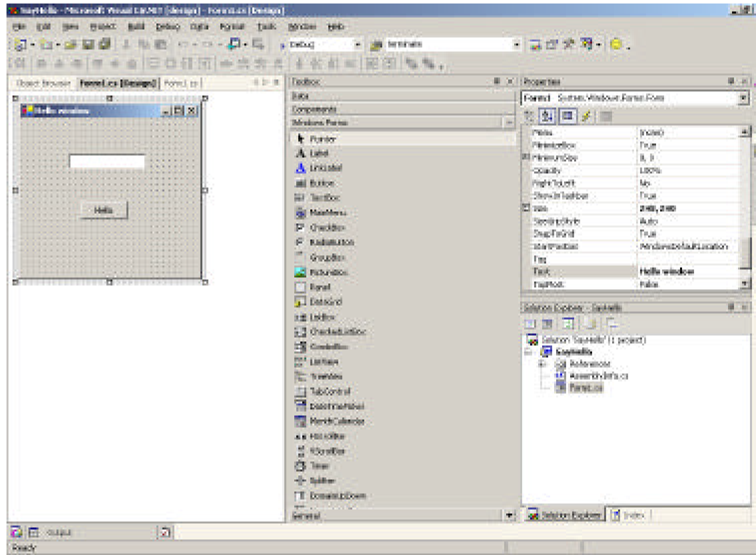


Figure 3-2 – The Windows designer in Visual Studio.NET

You can draw the controls on the form by selecting the TextBox from the Toolbox and dragging it onto the form, and then doing the same with the button. Then to create program code, we need only double click on the controls. In this simple form, we want to click on the "Hello" button and it copies the text from the text field to the textbox we called txHi, and clears the text field. So, in the designer, we double click on that button and the code below is automatically generated:

```
private void btHello_Click(object sender, EventArgs e) {
txHi.Text ="Hello there";
}
```

Note that the Click routine passes in a sender object and an event object that you can query for further information. Under the covers, it also connects the event to this method. The running program is shown in Figure 3-3.

Copyright © , 2002 by James W Cooper

Figure 3-3 – The SimpleHello form after clicking the Say Hello button.

While we only had to write one line of code inside the above subroutine, it is instructive to see how different the rest of the code is for this program. We first see that several libraries of classes are imported so the program can use them:

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
```

Most significant is the Windows.Forms library, which is common to all the .Net languages.

The code the designer generates for the controls is illuminating. And it is right there in the open for you to change if you want. Essentially, each control is declared as a variable and added to a container. Here are the control declarations. Note the event handler added to the btHello.Click event.

```
private System.Windows.Forms.TextBox txHi;
private System.Windows.Forms.Button btHello;

private void InitializeComponent()        {
      this.btHello = new System.Windows.Forms.Button();
      this.txHi = new System.Windows.Forms.TextBox();
```

Copyright © , 2002 by James W Cooper

```
      this.SuspendLayout();
      //
      // btHello
      //
      this.btHello.Location =  new System.Drawing.Point(80,
112);
      this.btHello.Name = "btHello";
      this.btHello.Size = new System.Drawing.Size(64, 24);
      this.btHello.TabIndex = 1;
      this.btHello.Text = "Hello";
      this.btHello.Click += new
EventHandler(this.btHello_Click);
      //
      // txHi
      //
      this.txHi.Location = new System.Drawing.Point(64,
48);
      this.txHi.Name = "txHi";
      this.txHi.Size = new System.Drawing.Size(104, 20);
      this.txHi.TabIndex = 0;
      this.txHi.Text = "";
      //
      // Form1
      //
      this.AutoScaleBaseSize = new System.Drawing.Size(5,
13);
      this.ClientSize = new System.Drawing.Size(240, 213);
      this.Controls.AddRange(
            new System.Windows.Forms.Control[] {
              this.btHello,
              this.txHi});
              this.Name = "Form1";
              this.Text = "Hello window";
              this.ResumeLayout(false);

            }
```

If you change this code manually instead of using the property page, the window designer may not work any more. We'll look more at the power of this system after we discuss objects and classes in the following chapter.

Copyright © , 2002 by James W Cooper

Windows Controls

All of the basic Windows controls work in much the same way as the TextBox and Button we have used so far. Many of the more common ones are shown in the Windows Controls program in Figure 3-4.

Figure 3-4 – A selection of basic Windows controls.

Each of these controls has properties such as Name, Text, Font, Forecolor and Borderstyle that you can change most conveniently using the properties window shown at the right of Figure 3-2. You can also change these properties in your program code as well. The Windows Form class that the designer generates always creates a Form1 constructor that calls an InitializeComponent method like the one above. One that method has been called, the rest of the controls have been created and you can change their properties in code. Generally, we will create a private *init()* method that is called right after the InitializeComponent method, in which we add any such additional initialization code.

Copyright © , 2002 by James W Cooper

Labels

A label is a field on the window form that simply displays text. Usually programmers use this to label the purpose of text boxes next to them. You can't click on a label or tab to it so it obtains the focus. However, if you want, you can change the major properties in Table 3-1 either in the designer or at runtime.

Property	Value
Name	At design time only
BackColor	A Color object
BorderStyle	None, FixedSingle or Fixed3D
Enabled	true or false. If false , grayed out.
Font	Set to a new Font object
ForeColor	A Color object
Image	An image to be displayed within the label
ImageAlign	Where in the label to place the image
Text	Text of the label
Visible	true or false

Table 3-1 –Properties for the Label Control

TextBox

The TextBox is a single line or multiline editable control. You can set or get the contents of that box using its Text property:

```
TextBox tbox = new TextBox();
tbox.Text = "Hello there";
```

In addition to the properties in Table 3-1, the TextBox also supports the properties in Table 3-2.

Property	Value
Lines	An array of strings, one per line
Locked	If true, you can't type into the text box
Multiline	true or false
ReadOnly	Same as locked. If true, you can still select the text and copy it, or set values

Copyright © , 2002 by James W Cooper

	from within code.
`WordWrap`	true or false

Table 3-2 – TextBox properties

CheckBox

A CheckBox can either be checked or not, depending on the value of the Checked property. You can set or interrogate this property in code as well as in the designer. You can create an event handler to catch the event when the box is checked or unchecked, by double clicking on the checkbox in the design mode.

CheckBoxes have a Appearance property which can be set to *Appearance.Normal* or *Appearance.Button*. When the appearance is set to the Button value, the control appears acts like a toggle button that stays depressed when you click on it and becomes raised when you click on it again. All the properties in Table 3-1 apply as well.

Buttons

A Button is usually used to send a command to a program. When you click on it, it causes an event that you usually catch with an event handler. Like the CheckBox, you create this event handler by double clicking on the button in the designer. All of the properties in Table 3-1 can be used as well.

Buttons are also frequently shown with images on them. You can set the button image in the designer or at run time. The images can be in bmp, gif, jpeg or icon files.

Radio buttons

Radio buttons or option buttons are round buttons that can be selected by clicking on them. Only one of a group of radio buttons can be selected at a time. If there is more than one group of radio buttons on a window form, you should put each set of buttons inside a Group box as we did in the program in Figure 3-1. As with checkboxes and buttons, you can attach

Copyright © , 2002 by James W Cooper

events to clicking on these buttons by double clicking on them in the designer. Radio buttons do not always have events associated with them. Instead, programmers check the Checked property of radio buttons when some other event, like an OK button click occurs.

Listboxes and Combo Boxes

Both list boxes and Combo boxes contain an Items array of the elements in that list. A ComboBox is a single line drop-down, that programmers use to save space when selections are changed less frequently. ListBoxes allow you to ser properties that allow multiple selections, but ComboBoxes do not. Some of their properties include those in Table 3-3.

Property	Value
Items	A collection of items in the list
MultiColumn	If true, the ColumnWidth property describes the width of each column.
SelectionMode	One, MultiSimple or MultiExtended. If set to MultiSimple, you can select or deselect multiple items with a mouse click. If set to MultiExtended, you can select groups of adjacent items with a mouse.
SelectedIndex	Index of selected item
SelectedIndices	Returns collection of selections when list box selection mode is multiple.
SelectedItem	Returns the item selected

Table 3-3 – The ListBox and ComboBox properties. SelectionMode and MultiColumn do not apply to combo boxes.

The Items Collection

You use the Items collection in the ListBox and ComboBox to add and remove elements in the displayed list. It is essentially an ArrayList, as we discuss in Chapter 8. The basic methods are shown in Table 3-4.

Method	Value

Copyright © , 2002 by James W Cooper

Add	Add object to list
Count	Number in list
Item[i]	Element in collection
RemoveAt(i)	Remove element i

Table 3-4 – Methods for the Items Collection

If you set a ListBox to a multiple selection mode, you can obtain a collection of the selected items or the selected indexes by

```
ListBox.SelectedIndexCollection it =
      new ListBox.SelectedIndexCollection (lsCommands);
ListBox.SelectedObjectCollection so =
      new ListBox.SelectedObjectCollection (lsCommands);
```

where *lsCommands* is the list box name.

Menus

You add menus to a window by adding a MainMenu controls to the window form. Then, you can the menu control and edit its drop-down names and new main item entries as you see in Figure 3-5.

Figure 3-5 – Adding a menu to a form.

As with other clickable controls, double clicking on one in the designer creates an event whose code you can fill in.

ToolTips

A ToolTip is a box that appears when your mouse pointer hovers over a control in a window. This feature is activated by adding an (invisible) ToolTip control to the form, and then adding specific tool tips control and

Copyright © , 2002 by James W Cooper

text combinations to the control. In our example in Figure 3-4, we add tooltips text to the button and list box using the *tips* control we have added to the window.

```
tips.SetToolTip (btPush, "Press to add text to list box");
tips.SetToolTip (lsCommands, "Click to copy to text box");
```

This is illustrated in Figure 3-6.

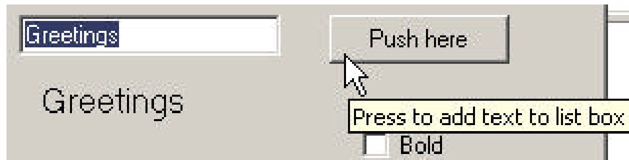


Figure 3-6 – A ToolTip over a button.

Other Windows Controls

We discuss how to use the Datagrid and TreeList in the Adapter and Bridge pattern chapters, and the Toolbar in the State and Strategey pattern chapters.

The Windows Controls Program

This program, shown in Figure 3-4, has the following features. The text in the label changes whenever you change the

- Font size from the combo box
- Font color from the radio buttons
- Font bold from the check box.

For the check box, we create a new font which is either bold or not depending on the state of the check box:

```
private void ckBold_CheckedChanged(object sender, EventArgs e) {
        if (ckBold.Checked ) {
                lbText.Font   =new Font ("Arial",
                                fontSize,FontStyle.Bold );
        }
```

Copyright © , 2002 by James W Cooper

```
        else {
               lbText.Font = new Font ("Arial", fontSize);
        }
}
```

When we create the form, we add the list of font sizes to the combo box:

```
private void init() {
        fontSize = 12;
        cbFont.Items.Add ("8");
        cbFont.Items.Add ("10");
        cbFont.Items.Add ("12");
        cbFont.Items.Add ("14");
        cbFont.Items.Add ("18");
        lbText.Text ="Greetings";
        tips.SetToolTip (btPush, "Press to add text to list box");
        tips.SetToolTip (lsCommands, "Click to copy to text box");
}
```

When someone clicks on a font size in the combo box, we convert that text to a number and create a font of that size. Note that we just call the check box changing code so we don't have to duplicate anything.

```
private void cbFont_SelectedIndexChanged(
               object sender, EventArgs e) {
        fontSize= Convert.ToInt16 (cbFont.SelectedItem );
        ckBold_CheckedChanged(null, null);
}
```

For each radio button, we click on it and insert color-changing code:

```
private void opGreen_CheckedChanged(object sender, EventArgs e) {
        lbText.ForeColor =Color.Green;
}

private void opRed_CheckedChanged(object sender, EventArgs e) {
        lbText.ForeColor =Color.Red ;
}

private void opBlack_CheckedChanged(object sender, EventArgs e) {
        lbText.ForeColor =Color.Black ;
}
```

When you click on the ListBox, it copies that text into the text box, by getting the selected item as an object and converting it to a string.

Copyright © , 2002 by James W Cooper

```
private void lsCommands_SelectedIndexChanged(
                object sender, EventArgs e) {
        txBox.Text = lsCommands.SelectedItem.ToString () ;
}
```

Finally, when you click on the File | Exit menu item, it closes the form, and hence the program:

```
private void menuItem2_Click(object sender, EventArgs e) {
        this.Close ();
}
```

Summary

Now that we've seen the basics of how you write programs in C#, we are ready to talk more about objects and OO programming in the chapters that follow.

Programs on the CD-ROM

Console Hello	`\IntroCSharp\Hello`
Windows hello	`\IntroCSharp\SayHello`
Windows controls	`\IntroCSharp\WinControls`

Copyright © , 2002 by James W Cooper

4. Using Classes and Objects in C#

What Do We Use Classes For?

All C# programs are composed of classes. The Windows forms we have just seen are classes, derived from the basic Form class and all the other programs we will be writing are made up exclusively of classes. C# does not have the concept of global data modules or shared data that is not part of classes.

Simply put, a class is a set of public and private *methods* and private data grouped inside named logical units. Usually, we write each class in a separate file, although this is not a hard and fast rule. We have already seen that these Windows forms are classes, and we will see how we can create other useful classes in this chapter.

When you create a class, it is not a single entity, but a master you can create copies or *instances* of, using the *new* keyword. When we create these instances, we pass some initializing data into the class using its *constructor*. A constructor is a method that has the same name as the class name, has no return type and can have zero or more parameters that get passed into each instance of the class. We refer to each of these instances as *objects.*

In the sections that follow we'll create some simple programs and use some instances of classes to simplify them.

A Simple Temperature Conversion Program

Suppose we wanted to write a visual program to convert temperatures between the Celsius and Fahrenheit temperature scales. You may remember that water freezes at zero on the Celsius scale and boils at 100 degrees, while on the Fahrenheit scale, water freezes at 32 and boils at 212. From these numbers you can quickly deduce the conversion formula that you may have forgotten.

Copyright © , 2002 by James W Cooper

The difference between freezing and boiling on once scale is 100 and on the other 180 degrees or 100/180 or 5/9. The Fahrenheit scale is "offset" by 32, since water freezes at 32 on its scale. Thus,

$$C = (F - 32)* 5/9$$

and

$$F = 9/5 * C + 32$$

In our visual program, we'll allow the user to enter a temperature and select the scale to convert it to as we see in Figure 4-1.

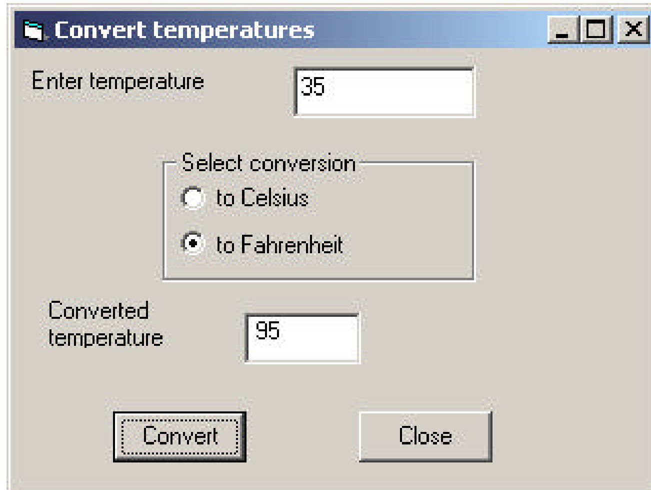


Figure 4-1– Converting 35 Celsius to 95 Fahrenheit with our visual interface.

Using the visual builder provided in Visual Studio.NET, we can draw the user interface in a few seconds and simply implement routines to be called when the two buttons are pressed. If we double click on the Convert button, the program generates the btConvert_Click method. You can fill it in to have it convert the values between temperature scales:

```
private void btCompute_Click(object sender,
              System.EventArgs e) {
       float temp, newTemp;
       //convert string to input value
```

Copyright © , 2002 by James W Cooper

```
        temp = Convert.ToSingle (txEntry.Text );
        //see which scale to convert to
        if(opFahr.Checked)
              newTemp = 9*temp/5 + 32;
        else
              newTemp = 5*(temp-32)/9;
        //put result in label text
        lbResult.Text =newTemp.ToString ();
        txEntry.Text ="";    //clear entry field
}
```

The above program is extremely straightforward and easy to understand, and is typical of how some simple C# programs operate. However, it has some disadvantages that we might want to improve on.

The most significant problem is that the user interface and the data handling are combined in a single program module, rather than being handled separately. It is usually a good idea to keep the data manipulation and the interface manipulation separate so that changing interface logic doesn't impact the computation logic and vice-versa.

Building a Temperature Class

A *class* in C# is a module that can contain both public and private functions and subroutines, and can hold private data values as well. These functions and subroutines in a class are frequently referred to collectively as *methods*.

Class modules allow you to keep a set of data values in a single named place and fetch those values using get and set functions, which we then refer to as *accessor methods*.

You create a class module from the C# integrated development environment (IDE) using the menu item Project | Add class module. When you specify a filename for each new class, the IDE assigns this name as the class name as well and generates an empty class with an empty constructor. For example, if we wanted to create a Temperature class, the IDE would generate the following code for us:

Copyright © , 2002 by James W Cooper

```
namespace CalcTemp
{
	/// <summary>
	/// Summary description for Temperatur.
	/// </summary>
	public class Temperature
	{
		public Temperature()
		{
			//
			// TODO: Add constructor logic here
			//
		}
	}
}
```

If you fill in the “summary description” special comment, that text will appear whenever your mouse hovers over an instance of that class. Note that the system generates the class and a blank constructor. If your class needs a constructor with parameters, you can just edit the code.

Now, what we want to do is with this class is to move all of the computation and conversion between temperature scales into this new Temperature class. One way to design this class is to rewrite the calling programs that will use the class module first. In the code sample below, we create an instance of the Temperature class and use it to do whatever conversions are needed:

```
private void btCompute_Click(object sender, System.EventArgs e) {
	string newTemp;
	//use input value to create instance of class
	Temperature temp = new Temperature (txEntry.Text );
	//use radio button to decide which conversion
	newTemp = temp.getConvTemp (opCels.Checked );

	//get result and put in label text
	lbResult.Text =newTemp.ToString ();
	txEntry.Text ="";    //clear entry field
}
```

The actual class is shown below. Note that we put the string value of the input temperature into the class in the constructor, and that inside the class it gets converted to a float. We do not need to know how the data are

Copyright © , 2002 by James W Cooper

represented internally, and we could change that internal representation at any time.

```
public class Temperature   {
       private float temp, newTemp;
       //-------------
       //constructor for class
       public Temperature(string thisTemp)           {
              temp = Convert.ToSingle(thisTemp);
       }
       //-------------
       public string getConvTemp(bool celsius){
       if (celsius)
              return getCels();
       else
              return getFahr();
       }
       //-------------
       private string getCels() {
              newTemp= 5*(temp-32)/9;
              return newTemp.ToString() ;
       }
       //-------------
       private string getFahr() {
              newTemp = 9*temp/5 + 32;
              return Convert.ToString(newTemp) ;
       }
}
```

Note that the temperature variable *temp* is declared as *private*, so it cannot be “seen” or accessed from outside the class. You can only put data into the class and get it back out using the constructor and the getConvTemp method. The main point to this code rearrangement is that the outer calling program does not have to know how the data are stored and how they are retrieved: that is only known inside the class.

The other important feature of the class is that it actually *holds data.* You can put data into it and it will return it at any later time. This class only holds the one temperature value, but classes can contain quite complex sets of data values.

Copyright © , 2002 by James W Cooper

We could easily modify this class to get temperature values out in other scales without still ever requiring that the user of the class know anything about how the data are stored, or how the conversions are performed

Converting to Kelvin

Absolute zero on the Celsius scale is defined as –273.16 degrees. This is the coldest possible temperature, since it is the point at which all molecular motion stops. The Kelvin scale is based on absolute zero, but the degrees are the same size as Celsius degrees. We can add a function

```
public string getKelvin() {
       newTemp = Convert.ToString (getCels() + 273.16)
}
```

What would the setKelvin method look like?

Putting the Decisions into the Temperature Class

Now we are still making decisions within the user interface about which methods of the temperature class. It would be even better if all that complexity could disappear into the Temperature class. It would be nice if we just could write our Conversion button click method as

```
private void btCompute_Click(object sender, System.EventArgs e) {
       Temperature temper =
              new Temperature(txEntry.Text , opCels.Checked);
       //put result in label text
       lbResult.Text = temper.getConvTemp();
       txEntry.Text ="";    //clear entry field
}
```

This removes the decision making process to the temperature class and reduces the calling interface program to just two lines of code.

The class that handles all this becomes somewhat more complex, however, but it then keeps track of what data as been passed in and what conversion must be done. We pass in the data and the state of the radio button in the constructor:

```
public Temperature(string sTemp, bool toCels)         {
       temp = Convert.ToSingle (sTemp);
       celsius =  toCels;
```

Copyright © , 2002 by James W Cooper

```
}
```

Now, the celsius boolean tells the class whether to convert or not and whether conversion is required on fetching the temperature value. The output routine is simply

```
public string getConvTemp(){
        if (celsius)
                return getCels();
        else
                return getFahr();
}
//-------------
private string getCels() {
        newTemp= 5*(temp-32)/9;
        return newTemp.ToString() ;
}
//-------------
private string getFahr() {
        newTemp = 9*temp/5 + 32;
        return Convert.ToString(newTemp) ;
}
```

In this class we have both public and private methods. The public ones are callable from other modules, such as the user interface form module. The private ones, getCels and getFahr, are used internally and operate on the temperature variable.

Note that we now also have the opportunity to return the output temperature as either a string or a single floating point value, and could thus vary the output format as needed.

Using Classes for Format and Value Conversion

It is convenient in many cases to have a method for converting between formats and representations of data. You can use a class to handle and hide the details of such conversions. For example, you might design a program where you can enter an elapsed time in minutes and seconds with or without the colon:

```
315.20
3:15.20
```

Copyright © , 2002 by James W Cooper

```
315.2
```

and so forth. Since all styles are likely, you'd like a class to parse the legal possibilities and keep the data in a standard format within. Figure 4-2 shows how the entries "112" and "102.3" are parsed.

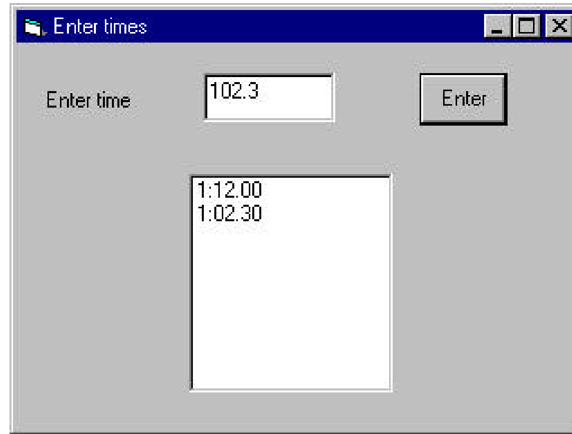


Figure 4-2 – A simple parsing program that uses the Times class.

Much of the parsing work takes place in the constructor for the class. Parsing depends primarily on looking for a colon. If there is no colon, then values greater than 99 are treated as minutes.

```
public FormatTime(string entry)         {
errflag = false;
if (! testCharVals(entry)) {
       int i = entry.IndexOf (":");
       if (i >= 0 ) {
              mins = Convert.ToInt32 (entry.Substring (0, i));
              secs = Convert.ToSingle (entry.Substring (i+1));
              if(secs >= 60.0F ) {
                     errflag = true;
                     t = NT;
              }
              t = mins *100 + secs;
       }
       else {
              float fmins = Convert.ToSingle (entry) / 100;
              mins = (int)fmins;
```

Copyright © , 2002 by James W Cooper

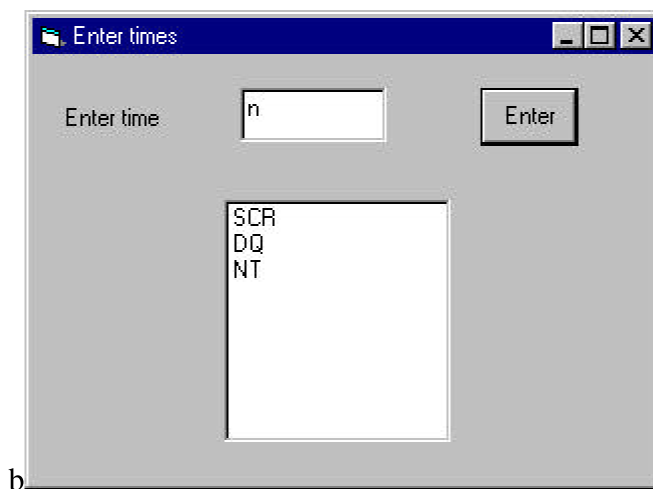
```
                secs = Convert.ToSingle (entry) - 100 * mins;
                if (secs >= 60) {
                       errflag = true;
                       t = NT;
                }
                else
                       t = Convert.ToSingle(entry);
           }
       }
}
```

Since illegal time values might also be entered, we test for cases like 89.22 and set an error flag.

Depending on the kind of time measurements these represent, you might also have some non-numeric entries such as NT for no time or in the case of athletic times, SC for scratch or DQ for disqualified. All of these are best managed inside the class. Thus, you never need to know what numeric representations of these values are used internally.

```
       static public int NT = 10000;
       static public int DQ = 20000;
```

Some of these are processed in the code represented by Figure 4-3.

b

Copyright © , 2002 by James W Cooper

Figure 4-3 – The time entry interface, showing the parsing of symbols for Scratch, Disqualification and No Time.

Handling Unreasonable Values

A class is also a good place to encapsulate error handling. For example, it might be that times greater than some threshold value are unlikely and might actually be times that were entered without a decimal point. If large times are unlikely, then a number such as 123473 could be assumed to be 12:34.73"

```
public void setSingle(float tm) {
        t = tm;
        if((tm > minVal) && (tm < NT)) {
                t = tm / 100.0f;
        }
}
```

The cutoff value minVal may vary with the domain of times being considered and thus should be a variable. You can also use the class constructor to set up default values for variables.

```
public class FormatTime {
  public FormatTime(string entry)        {
        errflag = false;
        minVal = 1000;
        t = 0;
```

A String Tokenizer Class

A number of languages provide a simple method for taking strings apart into tokens, separated by a specified character. While C# does not exactly provide a class for this feature, we can write one quite easily using the Split method of the string class. The goal of the Tokenizer class will be to pass in a string and obtain the successive string tokens back one at a time. For example, if we had the simple string

```
Now is the time
```

our tokenizer should return four tokens:

```
Now
```

Copyright © , 2002 by James W Cooper

```
is
the
time
```

The critical part of this class is that it holds the initial string and remembers which token is to be returned next.

We use the Split function, which approximates the Tokenizer but returns an array of substrings instead of having an object interface. The class we want to write will have a nextToken method that returns string tokens or a zero length string when we reach the end of the series of tokens.

The whole class is shown below.

```
//String Tokenizer class
public class StringTokenizer     {
  private string data, delimiter;
  private string[] tokens; //token array
  private int index;              //index to next token
//----------
public StringTokenizer(string dataLine)              {
       init(dataLine, " ");
}
//----------
//sets up initial values and splits string
private void init(String dataLine, string delim) {
       delimiter = delim;
       data = dataLine;
       tokens = data.Split (delimiter.ToCharArray() );
       index = 0;
}
//----------
public StringTokenizer(string dataLine, string delim) {
       init(dataLine, delim);
}
//----------
public bool hasMoreElements() {
       return (index < (tokens.Length));
}
//----------
public string nextElement() {
       //get the next token
       if( index < tokens.Length )
             return tokens[index++];
```

Copyright © , 2002 by James W Cooper

```
        else
                return "";    //or none
        }
}
```

The class is illustrated in use in Figure 4-4.

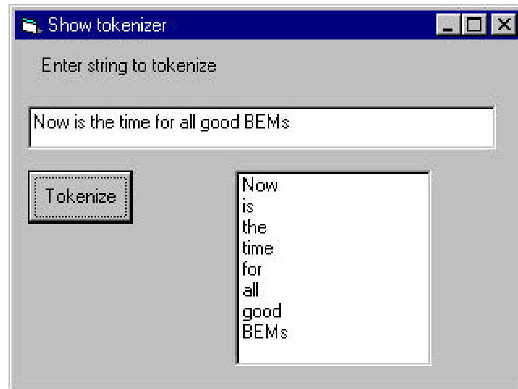


Figure 4-4– The tokenizer in use.

The code that uses the Tokenizer class is just:

```
//call tokenizer when button is clicked
private void btToken_Click(object sender,
                System.EventArgs e) {
        StringTokenizer tok =
                new StringTokenizer (txEntry.Text );
        while(tok.hasMoreElements () ) {
                lsTokens.Items.Add (tok.nextElement());
        }
  }
```

Classes as Objects

The primary difference between ordinary procedural programming and object-oriented (OO) programming is the presence of classes. A class is just a module as we have shown above, which has both public and private methods and which can contain data. However, classes are also unique in that there can be any number of *instances* of a class, each containing

Copyright © , 2002 by James W Cooper

different data. We frequently refer to these instances as objects. We'll see some examples of single and multiple instances below.

Suppose as have a file of results from a swimming event stored in a text data file. Such a file might look, in part, like this:

```
1 Emily Fenn                    17   WRAT        4:59.54
2 Kathryn Miller                16   WYW         5:01.35
3 Melissa Sckolnik              17   WYW         5:01.58
4 Sarah Bowman                  16   CDEV        5:02.44
5 Caitlin Klick                 17   MBM         5:02.59
6 Caitlin Healey                16   MBM         5:03.62
```

where the columns represent place, names, age, club and time. If we wrote a program to display these swimmers and their times, we'd need to read in and parse this file. For each swimmer, we'd have a first and last name, an age, a club and a time. An efficient way to keep the data for each swimmer grouped together is to design a Swimmer class and create an instance for each swimmer.

Here is how we read the file and create these instances. As each instance is created we add it into an ArrayList object:

```
private void init() {
       ar = new ArrayList ();     //create array list
       csFile fl = new csFile ("500free.txt");
       //read in liens
       string s =  fl.readLine ();
       while (s != null) {
              //convert to tokens in swimmer object
              Swimmer swm = new Swimmer(s);
              ar.Add (swm);
              s= fl.readLine ();
       }
       fl.close();
       //add names to list box
       for(int i=0; i < ar.Count ; i++) {
              Swimmer swm = (Swimmer)ar[i];
              lsSwimmers.Items.Add (swm.getName ());
       }
}
```

Copyright © , 2002 by James W Cooper

The Swimmer class itself parses each line of data from the file and stores it for retrieval using getXXX accessor functions:

```
public class Swimmer {
        private string frName, lName;
        private string club;
        private int age;
        private int place;
        private FormatTime tms;
//-----------
 public Swimmer(String dataLine) {
        StringTokenizer tok = new StringTokenizer (dataLine);
        place =  Convert.ToInt32 (tok.nextElement());
        frName = tok.nextElement ();
        lName = tok.nextElement ();
        string s = tok.nextElement ();
        age = Convert.ToInt32 (s);
        club = tok.nextElement ();
        tms = new FormatTime (tok.nextElement ());

 }
//-----------
 public string getName() {
        return frName+" "+lName;
 }
//-----------
public string getTime() {
        return tms.getTime();
 }
}
```

Class Containment

Each instance of the Swimmer class contains an instance of the StringTokenizer class that it uses to parse the input string and an instance of the Times class we wrote above to parse the time and return it in formatted form to the calling program. Having a class contain other classes is a very common ploy in OO programming and is one of the main ways we can build up more complicated programs from rather simple components.

The program that displays these swimmers is shown in Figure 4-5.

Copyright © , 2002 by James W Cooper

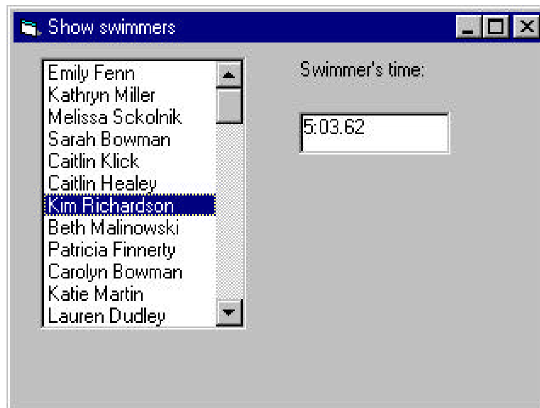


Figure 4-5 – A list of swimmers and their times, using containment.

When you click on any swimmer, her time is shown in the box on the right. The code for showing that time is extremely easy to write since all the data are in the swimmer class:

```
private void lsSwimmers_SelectedIndexChanged(
              object sender, System.EventArgs e) {
  //get index of selected swimmer
      int i = lsSwimmers.SelectedIndex ;
      //get that swimmer
      Swimmer swm = (Swimmer)ar[i];
      //display her time
      txTime.Text =swm.getTime ();
}
```

Initialization

In our Swimmer class above, note that the constructor in turn calls the constructor of the StringTokenizer class:

```
public Swimmer(String dataLine)         {
      StringTokenizer tok =
            new StringTokenizer (dataLine);
```

Copyright © , 2002 by James W Cooper

Classes and Properties

Classes in C# can have Property methods as well as public and private functions and subs. These correspond to the kinds of properties you associate with Forms, but they can store and fetch any kinds of values you care to use. For example, rather than having methods called getAge and setAge, you could have a single age property which then corresponds to a get and a set method:

```
private int Age;
//age property
public int age {
        get {
              return Age;
        }
        set {
              Age = value;
        }
}
```

Note that a property declaration does *not* contain parentheses after the property name, and that the special keyword *value* is used to obtain the data to be stored.

To use these properties, you refer to the age property on the left side of an equals sign to set the value, and refer to the age property on the right side to get the value back.

```
age = sw.Age;      //Get this swimmer's age
sw.Age = 12;       //Set a new age for this swimmer
```

Properties are somewhat vestigial, since they originally applied more to Forms in the Vb language, but many programmers find them quite useful. They do not provide any features not already available using get and set methods and both generate equally efficient code.

In the revised version of our SwimmerTimes display program, we convert all of the get and set methods to properties, and then allow users to vary the times of each swimmer by typing in new ones. Here is the Swimmer class

Copyright © , 2002 by James W Cooper

```
public class Swimmer
{
        private string frName, lName;
        private string club;
        private int Age;
        private int place;
        private FormatTime tms;
//-----------
public Swimmer(String dataLine)         {
        StringTokenizer tok = new StringTokenizer (dataLine);
        place =  Convert.ToInt32 (tok.nextElement());
        frName = tok.nextElement ();
        lName = tok.nextElement ();
        string s = tok.nextElement ();
        Age = Convert.ToInt32 (s);
        club = tok.nextElement ();
        tms = new FormatTime (tok.nextElement ());
}
//-----------
public string name {
        get{
        return frName+" "+lName;
        }
}
//-----------
public string time {
        get{
        return tms.getTime();
        }
        set  {
                tms = new FormatTime (value);
        }
}
//-------------------
//age property
public int age {
        get {
                return Age;
        }
        set {
                Age = value;
        }
}
        }
}
```

Copyright © , 2002 by James W Cooper

Then we can type a new time in for any swimmer, and when the txTime text entry field loses focus, we can store a new time as follows:

```
private void txTime_OnLostFocus(
       object sender, System.EventArgs e) {
       //get index of selected swimmer
       int i = lsSwimmers.SelectedIndex ;
       //get that swimmer
       Swimmer swm = (Swimmer)ar[i];
       swm.time =txTime.Text ;
}
```

Programming Style in C#

You can develop any of a number of readable programming styles for C#. The one we use here is partly influenced by Microsoft's Hungarian notation (named after its originator, Charles Simonyi) and partly on styles developed for Java.

We favor using names for C# controls such as buttons and list boxes that have prefixes that make their purpose clear, and will use them whenever there is more than one of them on a single form:

Control name	Prefix	Example
Buttons	bt	btCompute
List boxes	ls	lsSwimmers
Radio (option buttons)	op	opFSex
Combo boxes	cb	cbCountry
Menus	mnu	mnuFile
Text boxes	tx	txTime

We will not generally create new names for labels, frames and forms when they are never referred to directly in the code. We will begin class names with capital letters and instances of classes with lowercase letters. We will also spell instances and classes with a mixture of lowercase and capital letters to make their purpose clearer:

```
swimmerTime
```

Copyright © , 2002 by James W Cooper

Summary

In this chapter, we've introduced C# classes and shown how they can contain public and private methods and can contain data. Each class can have many instances and each could contain different data values. Classes can also have Property methods for setting and fetching data. These Property methods provide a simpler syntax over the usual getXXX and setXX accessor methods but have no other substantial advantages.m

Programs on the CD-ROM

Termperature conversion	\UsingClasses\CalcTemp
Temperature conversion using classes	\UsingClasses\ClsCalcTemp
Temperature conversion using classes	\UsingClasses\AllClsCalcTemp
Time conversion	\UsingClasses\Formatvalue
String tokenizer	\UsingClasses\TokenDemo
Swimmer times	\UsingClasses\SwimmerTokenizer

Copyright © , 2002 by James W Cooper

5. Inheritance

Now we will take up the most important feature of OO languages like C# (and VB.NET): inheritance. When we create a Windows form, such as our Hello form, the IDE (VS.NET Integrated Development Environment) creates a declaration of the following type:

```
public class Form1 : System.Windows.Forms.Form
```

This says that the form we create is a child class of the Form class, rather than being an instance of it. This has some very powerful implications. You can create visual objects and override some of their properties so that each behaves a little differently. We'll see some examples of this below.

Constructors

All classes have specific *constructors* that are called when you create an instance of a class. These constructors always have the same name as the class. This applies to form classes as well as non-visual classes. Here is the constructor the system generates for our simple hello window in the class Form1:

```
public class Form1 {
	public Form1(){			//constructor
		InitializeComponent();
	}
```

When you create your own classes, you must create constructor methods to initialize them, and can pass arguments into the class to initialize class parameters to specific values. If you do not specifically include a constructor in any class you write, a constructor having no arguments is generated for you under the covers.

The InitializeComponent method is generated by the IDE as well, and contains code that creates and positions all the visual controls in that window. If we need to set up additional code as part of the initialization of

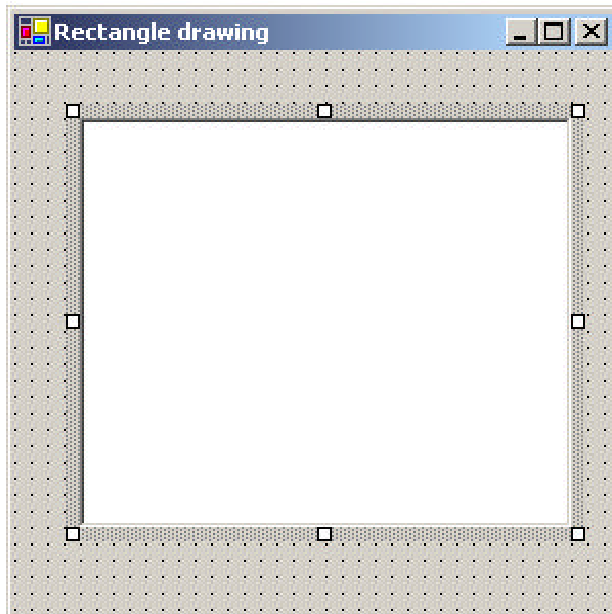
Copyright © , 2002 by James W Cooper

a Form class, we will always write a private *init* method that we call *after* the InitializeComponent method call.

```
public Form1(){
       InitializeComponent();
       init();
}
private void init() {
       x = 12.5f;         //set initial value of x
}
```

Drawing and Graphics in C#

In out first example, we'll write a program to draw a rectangle in a PictureBox on a form. In C#, controls are repainted by the Windows system and you can connect to the paint event to do your own drawing whenever a paint event occurs. Such a paint event occurs whenever the window is resize, uncovered or refreshed. To illustrate this, we'll create a Form containing a PictureBox, as shown in Figure 5-1.



Copyright © , 2002 by James W Cooper

Figure 5-1 – Inserting a PictureBox on a Form

Then, we'll select the PictureBox in the designer, and select the Events button (with the lightning icon) in the Properties window. This brings up a list of all the events that can occur on a PictureBox as shown in Figure 5-2.

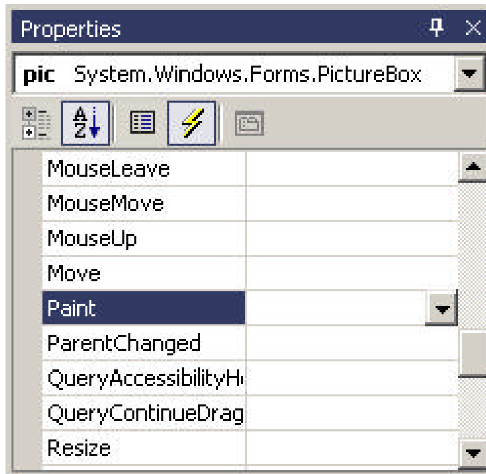


Figure 5-2 – Selecting the Paint Event for the PictureBox window.

Double clicking on the Paint event creates the following empty method in the Form's code:

```
private void pic_Paint(object sender, PaintEventArgs e) {
}
```

It also generates code that connects this method to the Paint event for that picture box, inside the *InitializeComponents* method.

```
this.pic.Paint += new PaintEventHandler(this.pic_Paint);
```

The PaintEventArgs object is passed into the subroutine by the underlying system, and you can obtain the graphics surface to draw on from that object. To do drawing, you must create an instance of a Pen object and define its color and, optionally its width. This is illustrated below for a black pen with a default width of 1.

Copyright © , 2002 by James W Cooper

```
private void pic_Paint(object sender, PaintEventArgs e) {
  Graphics g = e.Graphics;            //get Graphics surface
  Pen rpen = new Pen(Color.Black);   //create a Pen
  g.drawLine(rpen, 10,20,70,80);     //draw the line
}
```

In this example, we show the Pen object being created each time a paint event occurs. We might also create the pen once in the window's constructor or in the init method we usually call from within it.

Using Inheritance

Inheritance in C# gives us the ability to create classes which are derived from existing classes. In new derived classes, we only have to specify the methods that are new or changed. All the others are provided automatically from the base class we inherit from. To see how this works, lets consider writing a simple Rectangle class that draws itself on a form window. This class has only two methods, the constructor and the draw method.

```
namespace CsharpPats
{
public class Rectangle     {
       private int x, y, w, h;
       protected Pen rpen;
       public Rectangle(int x_, int y_, int w_, int h_)
       {
              x = x_;       //save coordinates
              y = y_;
              w = w_;
              h = h_;
              //create a pen
              rpen = new Pen(Color.Black);
       }
       //-----------------
       public void draw(Graphics g) {
              //draw the rectangle
              g.DrawRectangle (rpen, x, y, w, h);
       }
 }
}
```

Copyright © , 2002 by James W Cooper

Namespaces

We mentioned the System namespaces above. Visual Studio.Net also creates a namespace for each project equal to the name of the project itself. You can change this namespace on the property page, or make it blank so that the project is not in a namespace. However, you can create namespaces of your own, and the Rectangle class provides a good example of a reason for doing so. The System.Drawing namespace that this program requires to use the Graphics object also contains a Rectangle class. Rather than renaming our new Rectangle class to avoid this name overlap or "collision," we can just put the whole Rectangle class in its own namespace as we show above.

Then, when we declare the variable in the main Form window, we can declare it as a member of that namespace.

```
CsharpPats.Rectangle rec;
```

In this main Form window, we create an instance of our Rectangle class.

```
private void init() {
       rect = new CsharpPats.Rectangle (10, 20, 70, 100);
}
//---------------
public Form1() {
       InitializeComponent();
       init();
}
```

Then we add the drawing code to our Paint event handler to do the drawing and pass the graphics surface on to the Rectangle instance.

```
private void pic_Paint(object sender, PaintEventArgs e) {
       Graphics g =  e.Graphics;
       rect.draw (g);
}
```

This gives us the display we see in Figure 5-3.

Copyright © , 2002 by James W Cooper

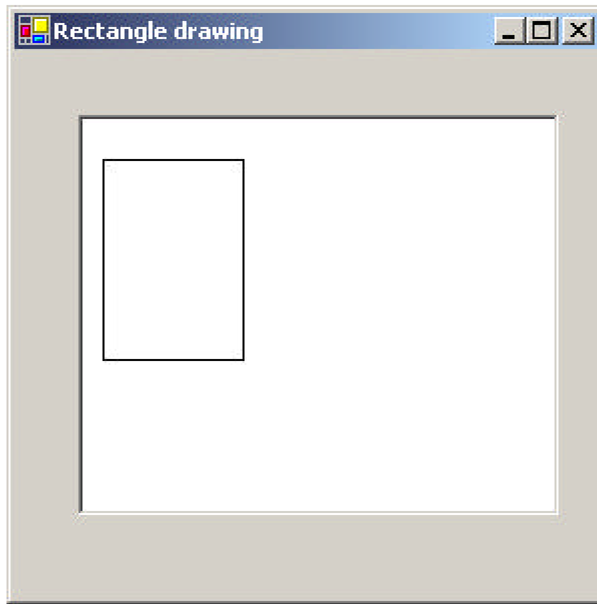


Figure 5-3 The Rectangle drawing program.

Creating a Square From a Rectangle

A square is just a special case of a rectangle, and we can derive a square class from the rectangle class without writing much new code. Here is the entire class:

```
namespace CsharpPats {
	public class Square : Rectangle  {
	public Square(int x, int y, int w):base(x, y, w, w) {
		}
	}
}
```

This Square class contains only a constructor, which passes the square dimensions on to the underlying Rectangle class by calling the constructor of the parent Rectangle class as part of the Square constructor.

```
base(x, y, w, w)
```

Copyright © , 2002 by James W Cooper

Note the unusual syntax: the call to the parent class's constructor follows a colon and is *before* the opening brace of the constructor itself.

The Rectangle class creates the pen and does the actual drawing. Note that there is no draw method at all for the Square class. If you don't specify a new method the parent class's method is used automatically, and this is what we want to have happen, here.

The program that draws both a rectangle and a square has a simple constructor where instances of these objects are created:

```
private void init() {
       rect = new Rectangle (10, 20, 70, 100);
       sq = new Square (150,100,70);
}
```

and a paint routine where they are drawn.

```
private void pic_Paint(object sender, PaintEventArgs e) {
       Graphics g =  e.Graphics;
       rect.draw (g);
       sq.draw (g);
}
```

The display is shown in Figure 5-4 for the square and rectangle:

Copyright © , 2002 by James W Cooper

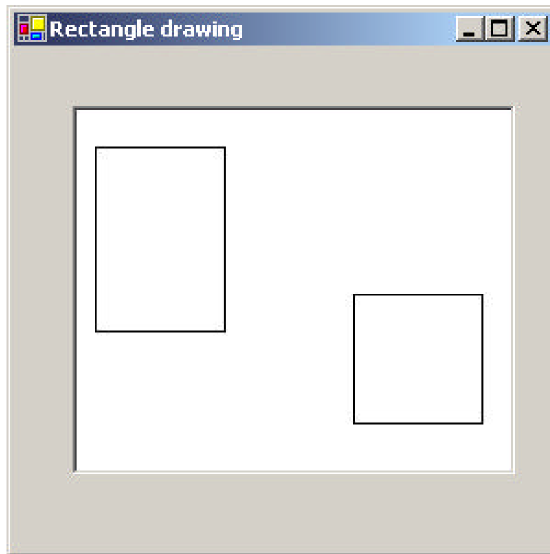


Figure 5-4 – The rectangle class and the square class derived from it.

Public, Private and Protected

In C#, you can declare both variables and class methods as public, private or protected. A public method is accessible from other classes and a private method is accessible only inside that class. Usually, you make all class variables private and write getXxx and seXxx accessor functions to set or obtain their values. It is generally a bad idea to allow variables inside a class to be accessed directly from outside the class, since this violates the principle of *encapsulation*. In other words, the class is the only place where the actual data representation should be known, and you should be able to change the algorithms inside a class without anyone outside the class being any the wiser.

C# introduces the *protected* keyword as well. Both variables and methods can be protected. Protected variables can be accessed within the class and from any subclasses you derive from it. Similarly, protected methods are only accessible from that class and its derived classes. They are not

Copyright © , 2002 by James W Cooper

publicly accessible from outside the class. If you do not declare any level of accessibility, private accessibility is assumed.

Overloading

In C# as well as other object oriented languages, you can have several class methods with the same name as long as they have different calling arguments or *signatures*. For example we might want to create an instance of a StringTokenizer class where we define both the string and the separator.

```
tok = new StringTokenizer("apples, pears", ",");
```

By declaring constructors with different numbers of arguments we say we are *overloading* the constructor. Here are the two constructors.

```
public StringTokenizer(string dataLine)              {
       init(dataLine, " ");
}
//----------
public StringTokenizer(string dataLine, string delim) {
       init(dataLine, delim);
}
private void init(string data, string delim) {
       //…
}
```

Of course C# allows us to overload any method as long as we provide arguments that allow the compiler can distinguish between the various overloaded (or *polymorphic*) methods.

Virtual and Override Keywords

If you have a method in a base class that you want to allow derived classes to override, you must declare it as *virtual*. This means that a method of the same name and argument signature in a derived class will be called rather than the one in the base class. Then, you must declare the method in the derived class using the *override* keyword.

Copyright © , 2002 by James W Cooper

If you use the *override* keyword in a derived class without declaring the base class's method as *virtual* the compiler will flag this as an error. If you create a method in a derived class that is identical in name and argument signature to one in the base class and do not declare it as *overload*, this also is an error. If you create a method in the derived class and do *not* declare it as *override* and also do *not* declare the base class's method as *virtual* the code will compile with a warning but will work correctly, with the derived class's method called as you intended.

Overriding Methods in Derived Classes

Suppose we want to derive a new class called DoubleRect from Rectangle, which draws a rectangle in two colors offset by a few pixels. We must declare the base class draw method as virtual:

```
public virtual void draw(Graphics g) {
  g.DrawRectangle (rpen, x, y, w, h);
}
```

In the derived DoubleRect constructor, we will create a red pen in the constructor for doing the additional drawing:

```
public class DoubleRect:Rectangle       {
private Pen rdPen;
public DoubleRect(int x, int y, int w, int h):
             base(x,y,w,h) {
       rdPen  = new Pen (Color.Red, 2);
}
```

This means that our new class DoubleRect will have to have its own draw method. However, this draw method will use the parent class's draw method but add more drawing of its own.

```
public override void draw(Graphics g) {
       base.draw (g);       //draw one rectangle using parent class
       g.DrawRectangle (rdPen, x +5, y+5, w, h);
}
```

Note that we want to use the coordinates and size of the rectangle that was specified in the constructor. We could keep our own copy of these parameters in the DoubleRect class, or we could change the protection mode of these variables in the base Rectangle class to protected from private.

Copyright © , 2002 by James W Cooper

```
protected int x, y, w, h;
```

The final rectangle drawing window is shown in Figure 5-5.

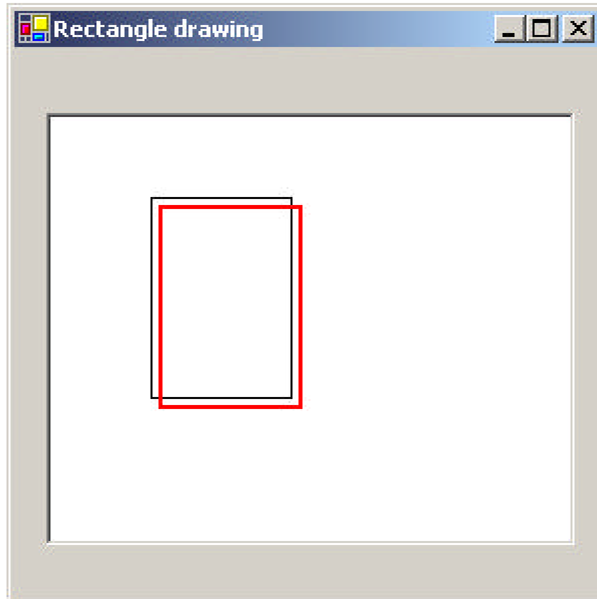


Figure 5-5 - The DoubleRect classes.

Replacing Methods Using New

Another way to replace a method in a base class when you cannot declare the base class method as *virtual* is to use the *new* keyword in declaring the method in the derived class. If you do this, it effectively hides any methods of that name (regardless of signature) in the base class. In that case, you cannot make calls to the base method of that name from the derived class, and must put all the code in the replacement method.

```
public new void draw(Graphics g) {
        g.DrawRectangle (rpen, x, y, w, h);
        g.DrawRectangle (rdPen, x +5, y+5, w, h);
```

Copyright © , 2002 by James W Cooper

```
}
```

Overriding Windows Controls

In C# we can easily make new Windows controls based on existing ones using inheritance. We'll create a Textbox control that highlights all the text when you tab into it. In C#, we can create that new control by just deriving a new class from the Textbox class.

We'll start by using the Windows Designer to create a window with two text boxes on it. Then we'll go to the Project|Add User Control menu and add an object called HiTextBox. We'll change this to inherit from TextBox instead of UserControl.

```
public class HiTextBox : Textbox {
```

Then, before we make further changes, we compile the program. The new HiTextBox control will appear at the bottom of the Toolbox on the left of the development environment. You can create visual instances of the HtextBox on any windows form you create. This is shown in Figure 5-6.

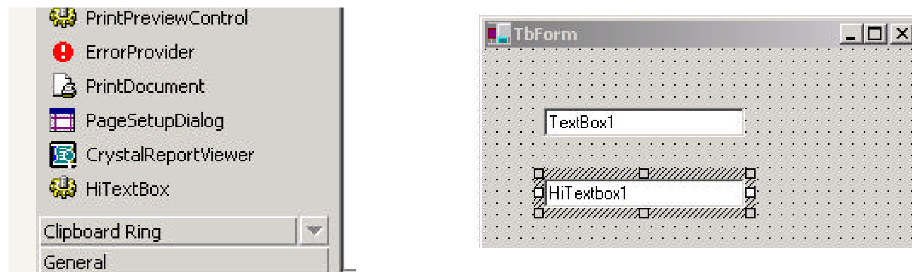


Figure 5-6 -The Toolbox, showing the new control we created and an instance of the HiTextBox on the Windows Designer pane of a new form.

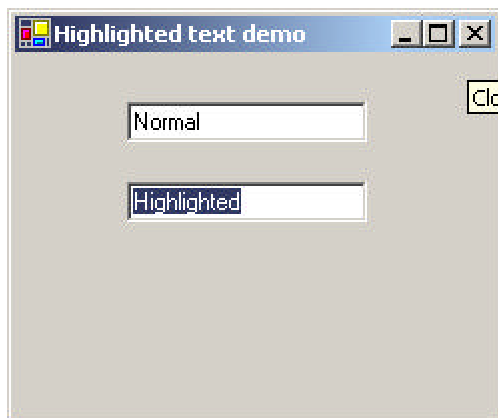
Now we can modify this class and insert the code to do the highlighting.

```
public class HiTextBox : System.Windows.Forms.TextBox
{
```

Copyright © , 2002 by James W Cooper

```
        private Container components = null;
//-------------
private void init() {
        //add event handler to Enter event
        this.Enter += new System.EventHandler (highlight);
}
//-------------
//event handler for highlight event
private void highlight(object obj, System.EventArgs e) {
        this.SelectionStart =0;
        this.SelectionLength =this.Text.Length ;
}
//-------------
public HiTextBox()          {
        InitializeComponent();
        init();
}
```

And that's the whole process. We have derived a new Windows control in about 10 lines of code. That's pretty powerful. You can see the resulting program in Figure Figure 5-6. If you run this program, you might at first think that the ordinary TextBox and the HiTextBox behave the same, because tabbing between them makes them both highlight. This is the "autohighlight" feature of the C# textbox. However, if you *click* inside the Textbox and the HiTextBox and tab back and forth, you will see in Figure 5-7 that only our derived HiTextBox continues to highlight.



Copyright © , 2002 by James W Cooper

Figure 5-7 A new derive d HiTextbox control and a regular Textbox control.

Interfaces

An interface is a declaration that a class will contain a specific set of methods with specific arguments. If a class has those methods, it is said to *implement* that interface. It is essentially a contract or promise that a class will contain all the methods described by that interface. Interfaces declare the signatures of public methods, but do not contain method bodies.

If a class implements an interface called Xyz, you can refer to that class as if it was of type Xyz as well as by its own type. Since C# only allows a single tree of inheritance, this is the only way for a class to be a member of two or more base classes.

Let's take the example of a class that provides an interface to a multiple select list like a list box or a series of check boxes.

```
//an interface to any group of components
//that can return zero or more selected items
//the names  are returned in an Arraylist
       public interface Multisel  {
              void clear();
              ArrayList getSelected();
              Panel getWindow();
       }
```

When you implement the methods of an interface in concrete classes, you must declare that the class uses that interface, and, you must provide an implementation of each method in that interface as well, as we illustrate below.

```
/// ListSel  class implements MultiSel interface
       public class ListSel : Multisel {
              public ListSel() {
              }
              public void clear() {
              }
              public ArrayList getSelected() {
                     return new ArrayList ();
              }
```

Copyright © , 2002 by James W Cooper

```
        public Panel getWindow() {
              return new Panel ();
        }
  }
```

We'll show how to use this interface when we discuss the Builder pattern.

Abstract Classes

An *abstract* class declares one or more methods but leaves them unimplemented. If you declare a method as abstract, you must also declare the class as abstract. Suppose, for example, that we define a base class called Shape. It will save some parameters and create a Pen object to draw with. However, we'll leave the actual *draw* method unimplemented, since every different kind of shape will need a different kind of drawing procedure:

```
public abstract class Shape      {
      protected int height, width;
      protected int xpos, ypos;
      protected Pen bPen;
      //-----
      public Shape(int x, int y, int h, int w)       {
            width = w;
            height = h;
            xpos = x;
            ypos = y;
            bPen = new Pen(Color.Black );
      }
      //-----
      public abstract void draw(Graphics g);
      //-----
      public virtual float getArea() {
            return height * width;
      }
}
```

Note that we declare the *draw* method as *abstract* and end it with a semicolon rather than including any code between braces. We also declare the overall class as abstract.

Copyright © , 2002 by James W Cooper

You can't create an instance of an abstract class like Shape, though. You can only create instances of derived classes in which the abstract methods are filled in. So, lets create a Rectangle class that does just that:

```
public class Rectangle:Shape      {
       public Rectangle(int x, int y,int h, int w):
                     base(x,y,h,w) {}
       //-----
       public override void draw(Graphics g) {
              g.DrawRectangle (bPen, xpos, ypos, width, height);
       }
}
```

This is a complete class that you can instantiate. It has a real draw method.

In the same way, we could create a Circle class which has its own draw method:

```
public class Circle :Shape        {
       public Circle(int x, int y, int r):
              base(x,y,r,r){      }
       //-----
       public override void draw(Graphics g) {
              g.DrawEllipse (bPen, xpos, ypos, width, height);
       }
}
```

Now, if we want to draw the circle and rectangle, we just create instances of them in the init method we call from our constructor. Note that since they are both of base type Shape we can treat them as Shape objects:

```
public class Form1 : System.Windows.Forms.Form       {
       private PictureBox pictureBox1;
       private Container components = null;
       private Shape rect, circ;
       //-----
       public Form1()             {
              InitializeComponent();
              init();
       }
       //-----
       private void init() {
              rect = new CsharpPats.Rectangle (50, 60, 70, 100);
              circ = new Circle (100,60, 50);
       }
```

Copyright © , 2002 by James W Cooper

Finally, we draw the two objects by calling their draw methods from the paint event handler we create as we did above:

```
private void pictureBox1_Paint(object sender, PaintEventArgs e) {
        Graphics g = e.Graphics ;
        rect.draw (g);
        circ.draw (g);
}
```

We see this program executing in Figure 5-8

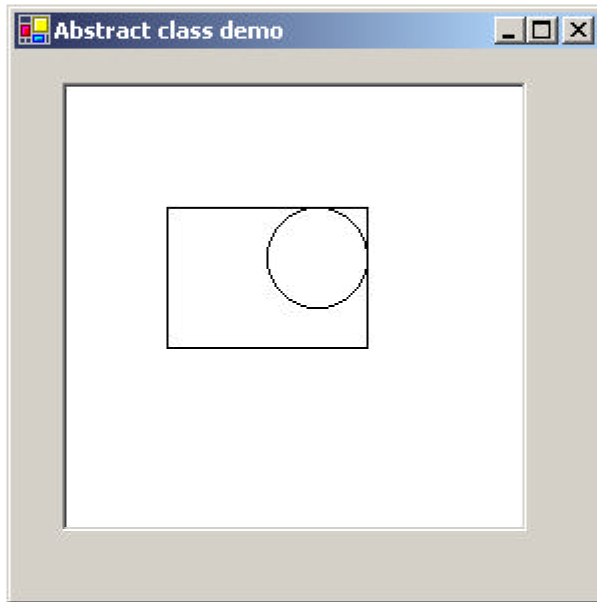


Figure 5-8 – An abstract class system drawing a Rectangle and Circle

Comparing Interfaces and Abstract Classes

When you create an *interface,* you are creating a set of one or more method definitions that you must write in each class that implements that interface. There is no default method code generated: you must include it yourself. The advantage of interfaces is that they provide a way for a class to appear to be part of two classes: one inheritance hierarchy and one from

Copyright © , 2002 by James W Cooper

the interface. If you leave an interface method out of a class that is supposed to implement that interface, the compiler will generate an error.

When you create an abstract class, you are creating a base class that might have one or more complete, working methods, but at least one that is left unimplemented, and declared abstract. You can't instantiate an abstract class, but must derive classes from it that do contain implementations of the abstract methods. If all the methods of an abstract class are unimplemented in the base class, it is essentially the same as an interface, but with the restriction that you can't make a class inherit from it as well as from another class hierarchy as you could with an interface. The purpose of abstract classes is to provide a base class definition for how a set of derived classes will work, and then allow the programmer to fill these implementations in differently in the various derived classes.

Another related approach is to create base classes with empty methods. These guarantee that all the derived classes will compile, but that the default action for each event is to do nothing at all. Here is a Shape class like that:

```
public class NullShape     {
             protected int height, width;
             protected int xpos, ypos;
             protected Pen bPen;
             //-----
             public Shape(int x, int y, int h, int w)       {
                    width = w;
                    height = h;
                    xpos = x;
                    ypos = y;
                    bPen = new Pen(Color.Black );
             }
             //-----
             public void draw(Graphics g){}
             //-----
             public virtual float getArea() {
                    return height * width;
             }
      }
```

Copyright © , 2002 by James W Cooper

Note that the *draw* method is now an empty method. Derived classes will compile without error, but they won't do anything much. And there will be no hint what method you are supposed to override, as you would get from using an abstract class.

Summary

We've seen the shape of most of the important features in C# in this chapter. C# provides inheritance, constructors and the ability to overload methods to provide alternate versions. This leads to the ability to create new derived versions even of Windows controls. In the chapters that follow, we'll show you how you can write design patterns in C#.

Programs on the CD-ROM

\Inheritance\RectDraw	Rectangle and Square
\Inheritance\DoubleRect	DoubleRect
\Inhertance\Hitext	A highlighted textbox
\Inheritance\abstract	Abstract Shape

Copyright © , 2002 by James W Cooper

6. UML Diagrams

We have illustrated the patterns in this book with diagrams drawn using Unified Modeling Language (UML). This simple diagramming style was developed from work done by Grady Booch, James Rumbaugh, and Ivar Jacobson, which resulted in a merging of ideas into a single specification and, eventually, a standard. You can read details of how to use UML in any number of books such as those by Booch et al. (1998), Fowler and Scott (1997), and Grand (1998). We'll outline the basics you'll need in this introduction.

Basic UML diagrams consist of boxes representing classes. Let's consider the following class (which has very little actual function).

```
public class Person         {
       private string name;
       private int age;
       //-----
       public Person(string nm, int ag)        {
              name = nm;
              age = ag;
       }
       public string makeJob() {
              return "hired";
       }
       public int getAge() {
              return age;
       }
       public void splitNames() {
       }
}
```

We can represent this class in UML, as shown in Figure 6-1.

Copyright © , 2002 by James W Cooper