

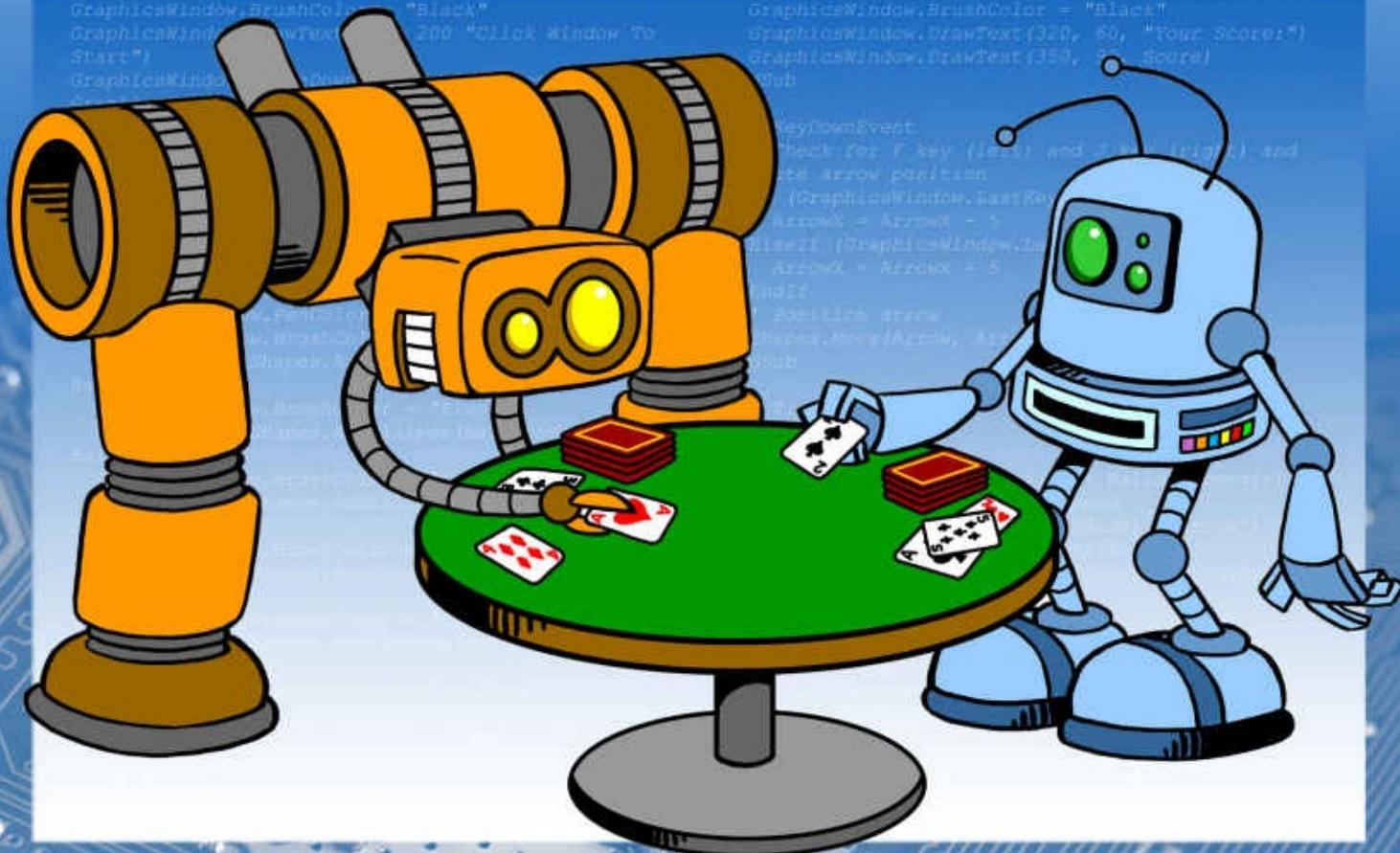
# VISUAL C#<sup>®</sup>

## HOMEWORK PROJECTS

### AN INTERMEDIATE STEP-BY-STEP TUTORIAL

```
GraphicsWindow.Width = 400  
GraphicsWindow.Height = 400  
/* start message  
GraphicsWindow.BrushColor = "Black"  
GraphicsWindow.DrawText(200, 60, "Click Window To Start")  
GraphicsWindow.BrushColor = "White"  
GraphicsWindow.FillRectangle(340, 80, 400, 120)  
GraphicsWindow.BrushColor = "Black"  
GraphicsWindow.DrawText(320, 60, "Your Score:")  
GraphicsWindow.DrawText(350, 80, score)  
Sub
```

```
KeydownEvent  
Check for F key (Left) and Z key (Right) and  
the arrow position  
(GraphicsWindow.GetKeyDown(ArrowKey = ArrowKey - 5  
Or If (GraphicsWindow.GetKeyDown(ArrowKey = ArrowKey + 5  
EndIf  
ElseIf arrow  
ElseIf MoreArrowKey, ArrowKey = ArrowKey + 5  
EndIf
```



PHILIP CONROD  
LOU TYLEE

# **Visual C#®**

## **Homework Projects**

An Intermediate Step-By-Step Tutorial

By  
Philip Conrod & Lou Tylee

©2017 Kidware Software LLC



<http://www.computerscienceforkids.com>  
<http://www.kidwaresoftware.com>

Copyright © 2017 by Kidware Software LLC. All rights reserved

Kidware Software LLC  
PO Box 701  
Maple Valley, Washington 98038  
1.425.413.1185  
[www.kidwaresoftware.com](http://www.kidwaresoftware.com)  
[www.computerscienceforkids.com](http://www.computerscienceforkids.com)

All Rights Reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Printed in the United States of America

ISBN-13: 978-1-937161-95-8 (Printed Edition)  
ISBN-13: 978-1-937161-96-5 (Electronic Edition)

Previous edition published as “Programming Home Projects with Visual C# Express - 2012 Edition”

Illustrations by Kevin Brockschmidt  
Copy Edit by Stephane and Jessica Conrod

This copy of “Visual C# Homework Projects” and the associated software is licensed to a single user. Copies of the course are not to be distributed or provided to any other user. Multiple copy licenses are available for educational institutions. Please contact Kidware Software for school site license information.

This guide was developed for the course, “Visual C# Homework Projects,” produced by Kidware Software, Maple Valley, Washington. It is not intended to be a complete reference to the Visual C# language. Please consult the Microsoft website for detailed reference information.

This guide refers to several software and hardware products by their trade names. These references are for informational purposes only and all trademarks are the property of their respective companies and owners. Microsoft, Visual Studio, Small Basic, Visual Basic, Visual J#, and Visual C#, IntelliSense, Word, Excel, MSDN, and Windows are all trademark products of the Microsoft Corporation. Java is a trademark product of the Oracle Corporation.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

This book expresses the author’s views and opinions. The information in this book is distributed on an "as is" basis, without and expresses, statutory, or implied warranties.

Neither the author(s) nor Kidware Software LLC shall have any liability to any person or entity with respect to any loss nor damage caused or alleged to be caused directly or indirectly by the information contained in this book.

## About The Authors

**Philip Conrod** has authored, co-authored and edited numerous computer programming books for kids, teens and adults. Philip holds a BS in Computer Information Systems and a Master's certificate in the Essentials of Business Development from Regis University. He also holds a Certificate in Programming for Business from WarrenTech. Philip has been programming computers since 1977. He has also held various Information Technology leadership roles in companies like Sundstrand Aerospace, Safeco Insurance Companies, FamilyLife, Kenworth Truck Company, PACCAR and Darigold Inc. In his spare time, Philip serves as the President & Publisher of Kidware Software, LLC. He is the proud father of three "techie" daughters and he and his beautiful family live in Maple Valley, Washington.

**Lou Tylee** holds BS and MS degrees in Mechanical Engineering and a PhD in Electrical Engineering. Lou has been programming computers since 1969 when he took his first Fortran course in college. He has written software to control suspensions for high speed ground vehicles, monitor nuclear power plants, lower noise levels in commercial jetliners, compute takeoff speeds for jetliners, locate and identify air and ground traffic and to let kids count bunnies, learn how to spell and do math problems. He has written several online texts teaching Visual Basic, Visual C# and Java to thousands of people. He taught a beginning Visual Basic course for over 15 years at a major university. Currently, Lou works as an engineer at a major Seattle aerospace firm. He is the proud father of five children and proud husband of his special wife. Lou and his family live in Seattle, Washington.

# Acknowledgements

I want to thank my three wonderful daughters - Stephanie, Jessica and Chloe, who helped with various aspects of the book publishing process including software testing, book editing, creative design and many other more tedious tasks like finding errors and typos. I could not have accomplished this without all your hard work, love and support. I want to also thank my best friend Jesus, who has always been there by my side giving me wisdom and guidance. Without you, this book would have never been printed and published.

I also want to thank my multi-talented co-author, Lou Tylee, for doing all the real hard work necessary to develop, test, debug, and keep current all the 'beginner-friendly' applications, games and base tutorial text found in this book. Lou has tirelessly poured his heart and soul into so many previous versions of this tutorial and there are so many beginners who have benefited from his work over the years. Lou is by far one of the best application developers and tutorial writers I have ever worked with. Thank you Lou for collaborating with me on this book project.

# **Contents**

Course Description  
Course Prerequisites  
Software Requirements  
Installing and Using the Downloadable Solution Files  
Using Visual C# Homework Projects  
Forward by Alan Payne, A Computer Science Teacher

## **1. Building Projects Using Visual C#**

Preview  
Requirements for Visual C# Homework Projects  
Introducing Visual C#  
Structure of a Visual C# Project  
Steps in Developing a Visual C# Project  
Starting Visual C#  
Visual C# Integrated Development Environment (IDE)  
Drawing the User Interface  
Setting Properties of Controls at Design Time  
Setting Properties at Run Time  
How Names are Used in Control Events  
Use of the Form Name Property  
Writing Code  
Saving a Visual C# Project  
Running a Visual C# Project  
Chapter Review

## **2. Overview of Visual C# Programming**

Review and Preview  
A Brief History of C#  
Rules of C# Programming  
Variables  
Visual C# Data Types  
Variable Declaration  
Arrays  
Constants

Variable Initialization  
Intellisense Feature  
Visual C# Statements and Expressions  
Type Casting  
Visual C# Arithmetic Operators  
Comparison and Logical Operators  
Concatenation Operators  
String Functions  
Dates and Times  
Random Number Object  
Math Functions  
Visual C# Decisions - If Statements  
Select Case - Another Way to Branch  
Visual C# Looping  
Visual C# Counting  
Chapter Review

### **3. Debugging a Visual C# Project**

Review and Preview  
Errors in Visual C# Projects  
Debugging Visual C# Projects  
Opening a Visual C# Project  
Debugging Example  
Using the Debugging Tools  
Breakpoints  
Viewing Variables in the Locals Window  
Viewing Variables in the Watch Window  
Call Stack Window  
Single Stepping (Step Into) an Application  
Method Stepping (Step Over)  
Method Exit (Step Out)  
Debugging Strategies  
Chapter Review

### **4. Dual-Mode Stopwatch Project**

Review and Preview  
Project Design Considerations

Dual-Mode Stopwatch Project Preview  
Form Object  
Button Control  
Label Control  
Timer Control  
Stopwatch Form Design  
Code Design – Initial to Running State  
Code Design – Timer Control  
Using General Methods in Projects  
Code Design – Update Display  
Code Design – Running to Stopped State  
Code Design – Stopped State  
Dual-Mode Stopwatch Project Review  
Dual-Mode Stopwatch Project Enhancements

## 5. Consumer Loan Assistant Project

Review and Preview  
Consumer Loan Assistant Project Preview  
TextBox Control  
Loan Assistant Form Design  
Code Design – Switching Modes  
Form Design – Tab Order and Focus  
Code Design – Computing Monthly Payment  
Code Design – Computing Number of Payments  
Code Design – Loan Analysis  
Code Design – New Loan Analysis  
Improving a Visual C# Project  
Code Design – Zero Interest  
Key Trapping  
Code Design – Key Trapping  
Message Box Dialog  
Code Design – Input Validation  
Consumer Loan Assistant Project Review  
Consumer Loan Assistant Project Enhancements

## 6. Flash Card Math Quiz Project

Review and Preview

Flash Card Math Quiz Project Preview  
CheckBox Control  
RadioButton Control  
GroupBox Control  
ScrollBar Controls  
Flash Card Math Form Design  
Code Design – Start Practice  
Code Design – Problem Generation  
Code Design – Obtaining Answer  
Handling Multiple Events in a Single Event Method  
Code Design – Choosing Problem Type  
Code Design – Timing Options  
Code Design – Presenting Results  
Flash Card Math Quiz Project Review  
Flash Card Math Quiz Project Enhancements

## 7. Multiple Choice Exam Project

Review and Preview  
Multiple Choice Exam Project Preview  
MenuStrip Control  
OpenFileDialog Control  
Multiple Choice Exam Form Design  
Form Design – Initialization  
Code Design – Exam File Format  
Code Design – Generating Exam Files  
Code Design – Opening an Exam File  
Code Design – Reading an Exam File  
Code Design – Error Trapping and Handling  
Form Design – Selecting Options  
Code Design – Start Exam  
Code Design – Question Generation  
Code Design – Checking Multiple Choice Answers  
Code Design – Checking Type In Answers  
Code Design – Checking Spelling  
Code Design – Presenting Results  
Multiple Choice Exam Project Review  
Multiple Choice Exam Project Enhancements

## **8. Blackjack Card Game Project**

Review and Preview  
Blackjack Card Game Project Preview  
PictureBox Control  
PictureBox Examples  
Blackjack Form Design  
Code Design – Card Definition  
Code Design – Card Shuffle  
Code Design – Start New Game  
Code Design – Start New Hand  
Code Design – End Hand  
Code Design – Display Dealer Card  
Code Design – Display Player Card  
Code Design – Deal New Hand  
Code Design – Player ‘Hit’  
Code Design – Player ‘Stay’  
Blackjack Card Game Project Review  
Blackjack Card Game Project Enhancements

## **9. Weight Monitor Project**

Review and Preview  
Weight Monitor Project Preview  
TabControl Control  
ListBox Control  
SaveFileDialog Control  
Weight Monitor Form Design  
Form Design – Weight Editor Tab  
Code Design – New Weight File  
Code Design – Entering Weights  
Code Design – Editing Weights  
Sequential File Output (Variables)  
Sequential File Input (Variables)  
Code Design – Saving Weight Files  
Code Design – Opening Weight Files  
Configuration Files  
Code Design – Configuration File  
Form Design – Weight Plot

Code Design – Weight Plot  
Code Design – Grid Lines  
Brush Object  
DrawString Method  
Code Design – Plot Labels  
Code Design – Weight Plot Trend  
Weight Monitor Project Review  
Weight Monitor Project Enhancements

## 10. Home Inventory Manager Project

Review and Preview  
Home Inventory Manager Project Preview  
ToolStrip Control  
ComboBox Control  
Home Inventory Manager Form Design  
Form Design – Toolbar  
Form Design – Tab Order and Focus  
Introduction to Object-Oriented Programming  
Code Design – InventoryItem Class  
Code Design – Inventory File Input  
Code Design – Viewing Inventory Item  
Code Design – Inventory File Output  
Code Design – Input Validation  
Code Design – New Inventory Item  
Code Design – Deleting Inventory Items  
Code Design – Editing Inventory Items  
Form Design – Inventory Item Search  
Code Design – Inventory Item Search  
Printing with Visual C#  
Printing Document Pages  
PrintDialog Control  
PrintPreviewDialog Control  
Code Design – Printing the Inventory  
Home Inventory Manager Project Preview  
Home Inventory Manager Project Enhancements

## 11. Snowball Toss Game Project

## **Review and Preview**

Snowball Toss Game Project Preview  
Snowball Toss Game Form Design  
Form Design – Choosing Options  
Code Design – Configuration Files  
Animation with Visual C#  
Drawing Images with Paint  
Code Design – Sprite Class  
Code Design – Start/Stop Game  
Code Design – Moving the Tossers  
Code Design – MovingSprite Class  
Code Design – Throwing Snowballs  
Code Design – Collision Detection  
Code Design – Zombie Snowmen  
Code Design – Playing Sounds  
Code Design – One Player Game  
Snowball Toss Game Project Review  
Snowball Toss Game Project Enhancements

## **Appendix. Distributing a Visual C# Project**

Preview

Distribution of Files

Application Icons

Setup Wizard

Building the Setup Program

Installing a Visual C# Application

## **More Self-Study or Instructor-Led Computer Programming Tutorials by Kidware Software**

## Course Description

**VISUAL C# HOMEWORK PROJECTS** teaches Visual C# programming concepts while providing detailed step-by-step instructions in building many fun and useful projects. **VISUAL C# HOMEWORK PROJECTS** explains (in simple, easy-to-follow terms) how to build a Visual C# Windows project. Students learn about project design, the Visual C# toolbox, many elements of the Visual C# language, and how to debug and distribute finished projects. The projects built include:

- **Dual-Mode Stopwatch** - Allows you to time tasks you may be doing.
- **Consumer Loan Assistant** - Helps you see just how much those credit cards are costing you.
- **Flash Card Math Quiz** - Lets you practice basic addition, subtraction, multiplication and division skills.
- **Multiple Choice Exam** - Quizzes a user on matching pairs of items, like countries/capitals, words/meanings, books/authors.
- **Blackjack Card Game** - Play the classic card game against the computer to see how risky gambling really is!
- **Weight Monitor** - Track your weight each day and monitor your progress toward established goals.
- **Home Inventory Manager** - Helps you keep track of all your belongings - even includes photographs.
- **Snowball Toss Game** - Lets you throw snowballs at another player or against the computer - has varying difficulties.

The product includes over 600 pages of self-study notes, all Visual C# source code and all needed graphics and sound files.

## **Course Prerequisites**

To grasp the concepts presented in **VISUAL C# HOMEWORK PROJECTS**, you should possess a working knowledge of the Windows operating system. You should know how to use Windows Explorer to locate, copy, move and delete files. You should be familiar with the simple tasks of using menus, toolbars, resizing windows, and moving windows around.

You should have had some exposure to Visual C# programming (or some other programming language). We offer two beginning programming tutorials (**VISUAL C# FOR KIDS** and **BEGINNING VISUAL C#**) that would help you gain this needed exposure. You will also need the ability to view and print documents saved in an Acrobat PDF format.

## **Software Requirements**

To use Visual C#, you (and your potential users) must be using Windows 7 or higher. And, of course, you need to have the Visual Studio Community Edition product installed on your computer. It is available for free download from Microsoft. Follow this link for complete instructions for downloading and installing it on your computer:

<https://www.visualstudio.com/products/free-developer-offers-vs>

## Installing and Using the Downloadable Solution Files

If you purchased this directly from our website you received an email with a special and individualized internet download link where you could download the compressed Program Solution Files. If you purchased this book through a 3rd Party Book Store like [Amazon.com](http://Amazon.com), the solutions files for this tutorial are included in a compressed ZIP file that is available for download directly from our website at:

<http://www.kidwaresoftware.com/phpVCS2015-registration.html>

Complete the online web form at the webpage above with your name, shipping address, email address, the exact title of this book, date of purchase, online or physical store name, and your order confirmation number from that store. After we receive all this information we will email you a download link for the Source Code Solution Files associated with this book.

**Warning:** If you purchased this book “used” or “second hand” you are not licensed or entitled to download the Program Solution Files. However, you can purchase the Digital Download Version of this book at a discounted price which allows you access to the digital source code solutions files required for completing this tutorial.

## Using Visual C# Homework Projects

The course notes and code for **VISUAL C# HOMEWORK PROJECTS** are included in one or more ZIP file(s). Use your favorite ‘unzipping’ application to write all files to your computer. The course is included in the folder entitled **HomeVCS**. This folder contains two other folders: **HomeVCS Notes** and **HomeVCS Projects**. There’s a chance when you copy the files to your computer, they will be written as ‘**Read-Only**.’ To correct this (in **Windows Explorer** or **My Computer**), right-click the **HomeVCS** folder and remove the check next to **Read only**. Make sure to choose the option to apply this change to all sub-folders and files. The **HomeVCS Projects** folder includes all projects developed during the course. Work through the notes and projects at your leisure.

## **Forward by Alan Payne, A Computer Science Teacher**

### **What is "Visual C# Homework Projects" and how it works.**

These lessons are a highly organized and well-indexed set of lessons in the Visual C# programming environment. They are written for the initiated programmer: the college or university student seeking to advance their computer science repertoire on their own, or the enlightened professional who wishes to embark on Visual C# coding for the first time. Skilled programmers and beginners alike benefit from the style of presentation.

While full solutions are provided, practical projects are presented in an easy-to-follow set of lessons explaining the rational for the solution - the form layout, coding design and conventions, and specific code related to the problem. The learner may follow the tutorials at their own pace while focusing upon context relevant information.

The finished product is the reward, but the adult student is fully engaged and enriched by the process. This kind of learning is often the focus of teacher training at the highest level. Every Computer Science teacher and self-taught learner knows what a great deal of work is required for projects to work in this manner, and with these tutorials, the work is done by an author who understands the adult need for streamlined learning.

### **Graduated Lessons for Every Project. Graduated Learning. Increasing and appropriate difficulty. Great results.**

By presenting Home Projects in this graduated manner, adult students are fully engaged and appropriately challenged to become independent thinkers who can come up with their own project ideas and design their own forms and do their own coding. Once the problem-solving process is learned, then student engagement is unlimited! Students literally cannot get enough of what is being presented.

These projects encourage accelerated learning - in the sense that they provide an

enriched environment to learn Computer Science, but they also encourage accelerating learning because students cannot put the lessons away once they start! Computer Science provides this unique opportunity to challenge students, and it is a great testament to the authors that they are successful in achieving such levels of engagement with consistency.

## **My history with the Kidware Software products.**

As a learner who just wants to get down to business, these lessons match my learning style. I do not waste valuable time ensconced in language reference libraries for programming environments and help screens which can never be fully remembered! With every Home Project, the pathway to learning is clear and immediate, though the topics in Computer Science remain current, relevant and challenging.

Some of the topics covered in these tutorials include:

- \* Getting to know the Visual C# Environment
- \* Overview of Visual C# Programming, including...
- \* Data Types and Ranges
- \* Scope of Variables
- \* Naming Conventions
- \* Arithmetic, Comparison and Logical Operators
- \* String Functions, Dates and Times, Random Numbers,
- \* Decision Making (Selections)
- \* Looping
- \* Language Functions - String, Date, Numerical
- \* Arrays, Control Arrays
- \* Writing Your own Methods and Classes
- \* Sequential File Access, Error-Handling and Debugging techniques
- \* Distributing a Visual C# Express Project (in the Appendix)
- \* and more... it's all integrated into the Homework Projects.

The specific Home Projects include:

- \* Dual-Mode Stopwatch

- \* Consumer Loan Assistant
- \* Flash Card Math Quiz
- \* Multiple Choice Exam Project
- \* Black Jack Card Game
- \* Weight Monitor Project
- \* Home Inventory Manager
- \* Snowball Toss Game

### **Quick learning curve by Contextualized Learning**

"Visual C# Homework Projects" encourages contextualized, self-guided learning.

Once a project idea is introduced, then the process of form-design, naming controls and coding is mastered for a given set of Visual C# controls. Then, it is much more likely that students create their own projects and solutions from scratch. This is the pattern of learning for any language!

Students may trust the order of presentation in order to have sufficient background information for every project. But the lessons are also highly indexed, so that students may pick and choose projects if limited by time.

Materials already condense what is available from MSDN so that students remember what they learn.

### **Meet Different State and Provincial Curriculum Expectations and More**

Different states and provinces have their own curriculum requirements for Computer Science. With the Kidware Software products, you may pick and choose from Home Projects which best suit your learning needs. Learners focus upon design stages and sound problem-solving techniques from a Computer Science perspective. In doing so, they become independent problem-solvers, and will exceed the curricular requirements of secondary and post-secondary schools everywhere.

Computer Science topics not explicitly covered in tutorials can be added at the

learner's discretion. The language - whether it is Visual Basic, Visual C#, Visual C++, or Console Java, Java GUI, etc... is really up to the individual learner !

### **Lessons encourage your own programming extensions.**

Once Computer Science concepts are learned, it is difficult to NOT know how to extend the learning to your own Home Projects and beyond!

Having my own projects in one language, such as Visual C#, I know that I could easily adapt them to other languages once I have studied the Kidware Software tutorials. I do not believe there is any other reference material out there which would cause me to make the same claim! In fact, I know there is not as I have spent over a decade looking!

Having used Kidware Software tutorials for the past decade, I have been successful at the expansion of my own learning to other platforms such as XNA for the Xbox, or the latest developer suites for tablets and phones. I thank Kidware Software and its authors for continuing to stand for what is right in the teaching methodologies which not only inspire, but propel the self-guided learner through what can be a highly intelligible landscape of opportunities.

Regards,

Alan Payne, B.A.H., B.Ed.  
Computer Science Teacher  
T.A. Blakelock High School  
Oakville, Ontario  
<http://chatt.hdsb.ca/~paynea>

**1**

# **Building Projects Using Visual C#**

# Preview

In this first chapter, we will do an overview of how to build a Windows project using Visual C#. You'll get a brief history of Visual C#, review the parts of a Visual C# project and delve into use of the Integrated Development environment (IDE).

# Requirements for Homework Projects With Visual C#

Before starting, let's examine what you need to successfully build the projects included with **Homework Projects With Visual C#**. As far as computer skills, you should be comfortable working within the Windows environment. You should know how to run programs, find and create folders, and move and resize windows. As far as programming skills, we assume you have had some exposure to computer programming using some language. If that language is **Visual C#**, great!! We offer two tutorials **Visual C# for Kids** and **Beginning Visual C#**, that could help you gain that exposure (see our website for details). But, if you've ever programmed in any language (Visual C#, C, C++, C#, Java, J#, Ada, even FORTRAN), you should be able to follow what's going on. Even if you are a veteran programmer, we suggest you go through the first three chapters before attacking the projects. This review will give you some idea of the terminology we use in referring to different parts of a Visual C# project.

Regarding software requirements, to use Visual C#, you (and your potential users) must be using Windows 7 or Windows 10. And, of course, you need to have the Visual C# (part of Visual Studio Community Edition) product installed on your computer. It is a free download from Microsoft. Follow this link for complete instructions for downloading and installing Visual C# on your computer:

<https://www.visualstudio.com/free-developer-offers/>

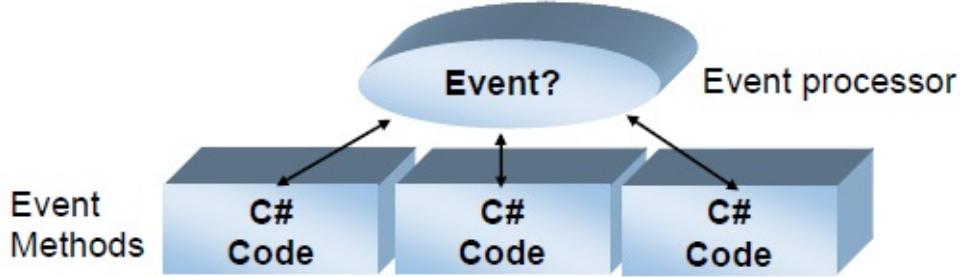
# Introducing Visual C#

In the early-1980's, there were many 'toy' computers being introduced to the computing world. Names like VIC-20, Commodore 64, Apple II, Texas Instruments 99/4A, Atari 400, Coleco Adam, Sinclair Timex, and the IBM PC Jr., were finding their way to market. This market created many home and hobbyist programmers who enjoyed writing computer programs they could use at home. These programs were written in BASIC, a language included for free with the computers being sold. Since that time, programming languages have become more elaborate and more expensive and the needs and desires of the hobbyist programmer have shifted to the background.

With Visual C#, you are able to quickly build Windows-based applications (the emphasis in these notes), database applications and projects that run on the web. Windows applications built using Visual C# feature a Graphical User Interface (GUI). Users interact with a set of visual tools (buttons, text boxes, tool bars, menu items) to make an application do its required tasks. The applications have a familiar appearance to the user.

As you develop as a Visual C# programmer, you will begin to look at Windows applications in a different light. You will recognize and understand how various elements of Word, Excel, Access and other applications work. You will develop a new vocabulary to describe the elements of Windows applications.

Visual C# Windows applications are **event-driven**, meaning nothing happens until an application is called upon to respond to some event (button pressing, menu selection, ...). Visual C# is governed by an event processor. As mentioned, nothing happens until an event is detected. Once an event is detected, a corresponding event method is located and the instructions provided by that method are executed. Those instructions are the actual code written by the programmer. In Visual C#, that code is written using a version of the C# programming language. Once an event method is completed, program control is then returned to the event processor.



All Windows applications are event-driven. For example, nothing happens in Word until you click on a button, select a menu option, or type some text. Each of these actions is an event.

The event-driven nature of applications developed with Visual C# makes it very easy to work with. As you develop a Visual C# application, event methods can be built and tested individually, saving development time. And, often event methods are similar in their coding, allowing re-use (and lots of copy and paste).

Visual C# possesses many features of more powerful (and more expensive) programming languages:

- All new, easy-to-use, powerful Integrated Development Environment (IDE)
- Full set of controls - you 'draw' the application
- Response to mouse and keyboard actions
- Printer access
- Full array of mathematical, string handling, and graphics functions
- Can easily work with arrays of variables and objects
- Sequential file support
- Useful debugger and structured error-handling facilities
- Easy-to-use graphic tools

In these notes, we will use Visual C# to build many fun projects you can use around your home. The projects you will build are:

- **Dual-Mode Stopwatch** – Measures total and elapsed time.
- **Consumer Loan Assistant** – Helps you determine just how much those loans cost you.

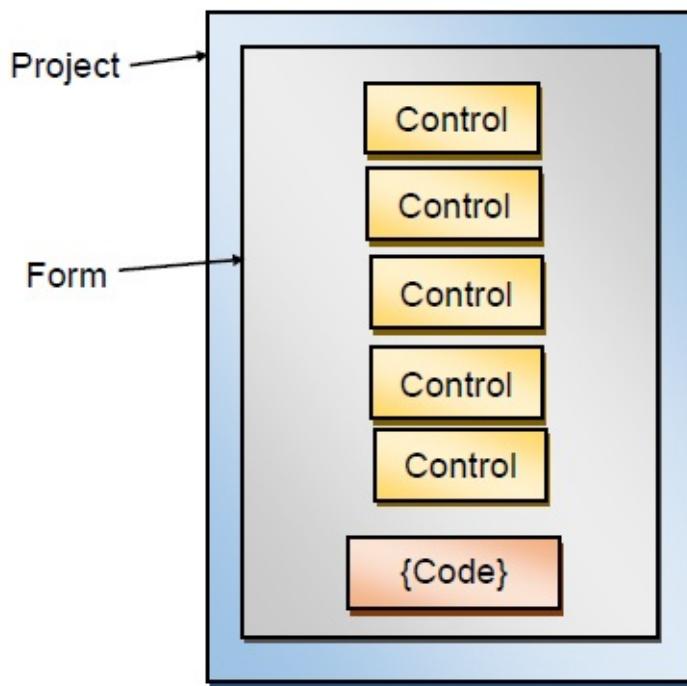
- **Flash Card Math Quiz** – Practice basic math skills with timed drills.
- **Multiple Choice Exam** – Set up exams matching like terms.
- **Blackjack Card Game** – The classic casino game.
- **Biorhythm Tracker** – Does your body follow natural rhythms?
- **Weight Monitor** – Tool to aid in your weight management.
- **Home Inventory Manager** – Keep track of all the stuff you own for insurance purposes.
- **Snowball Toss Game** – A little game using sounds and animation.

Each project will be addressed in a single chapter. Complete step-by-step instructions covering every project detail will be provided. Before beginning the projects, however, we will review the Visual C# development environment in the remainder of this chapter. Then, we provide an overview of the Visual C# programming language (Chapter 2) and instructions on using the Visual C# debugger tools (Chapter 3). The projects will begin with Chapter 4.

# Structure of a Visual C# Project

We want to get started building our first Visual C# Windows project. But, first we need to define some of the terminology we will be using. In Visual C#, a Windows **application** is defined as a **solution**. A solution is made up of one or more **projects**. Projects are groups of forms and code that make up some application. In these notes, our applications (solutions) will be made up of a single project. Because of this, we will usually use the terms application, solution and project synonymously.

As mentioned, a project (application) is made up of forms and code. In these notes, our projects will only use a single form. Pictorially, this is:



This **Project** (application) is made up of:

- **Form** - Window that you create for user interface
- **Controls** - Graphical features drawn on forms to allow user interaction (text boxes, labels, scroll bars, buttons, etc.) Forms and Controls are also called **objects**. Visual C# is known as an **object-oriented language**.
- **Properties** - Every characteristic of a form or control is specified by a

property. Example properties include names, captions, size, color, position, and contents. Visual C# applies default properties. You can change properties when designing the application or even when an application is executing.

- **Methods** - Built-in procedures that can be invoked to impart some action to a particular object.
- **Event methods** - **Code** related to some object or control. This is the code that is executed when a certain event occurs. In our applications, this code will be written in the C# language (covered in detail in Chapter 2 of these notes).
- **General Methods** - **Code** not related to objects. This code must be invoked or called in the application.

Visual C# uses a very specific directory structure for saving all of the components for a particular application. When you start a new project (solution), you will be asked for a **Name** and **Location** (directory). A folder named **Name** will be established in the selected **Location**. That folder will be used to store all solution files, project files, form files (cs extension) and other files needed by the project. Two subfolders will be established within the Name folder: **Bin** and **Obj**. The **Obj** folder contains files used for debugging your application as it is being developed. The **Bin** folder contains two other folders: **Debug** and **Release**. The **Bin\Debug** folder holds your compiled application (the actual executable code or **exe** file). Later, you will see that this folder is considered the ‘application path’ when ancillary data, graphics and sound files are needed by an application.

In a project folder, you will see these files (and possibly more):

<b>SolutionName.sln</b>	Solution file for solution named SolutionName
<b>SolutionName.suo</b>	Solution options file
<b>ProjectName.csproj</b>	Project file – one for each project in solution
<b>ProjectName.csproj.user</b>	Another Project file – one for each project in solution
<b>FormName.resx</b>	Form resources file – one for each form
<b>FormName.cs</b>	Form code file – one for each form
<b>App.ico</b>	Icon used to represent the application

# Steps in Developing a Visual C# Project

The Visual C# Integrated Development Environment (IDE) makes building an application a straightforward process. There are three primary steps involved in building a Visual C# application:

1. **Draw** the user **interface** by placing controls on a Windows form
2. **Assign properties** to controls
3. **Write code** for control events (and perhaps write other methods)

These same steps are followed whether you are building a very simple application or one involving many controls and many lines of code.

The event-driven nature of Visual C# applications allows you to build your application in stages and test it at each stage. You can build one method, or part of a method, at a time and try it until it works as desired. This minimizes errors and gives you, the programmer, confidence as your application takes shape.

As you progress in your programming skills, always remember to take this sequential approach to building a Visual C# project. Build a little, test a little, modify a little and test again. You'll quickly have a completed application. This ability to quickly build something and try it makes working with Visual C# fun – not a quality found in some programming environments! Now, we'll start Visual C# and look at each step in the application development process.

# Starting Visual C#

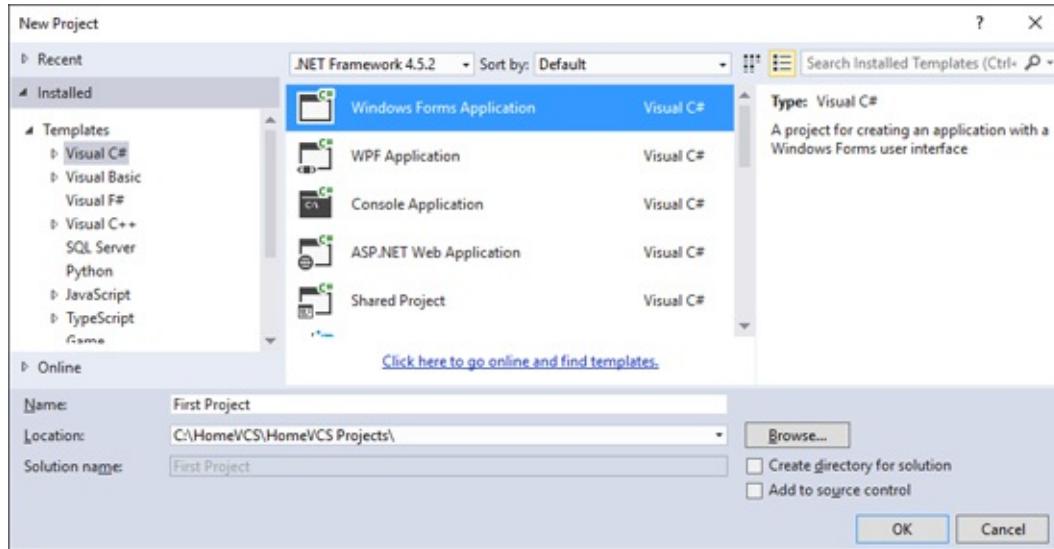
We assume you have Visual C# installed and operational on your computer. **Visual C#** is included as a part of **Microsoft Visual Studio Community Edition**. Visual Studio includes not only Visual C#, but also Visual C++ and Visual Basic. All three languages use the same development environment.

Once installed, to start Visual C#:

- Click on the **Start** button on the Windows task bar.
- Select **All apps**, then **Visual Studio**

The Visual C# program should start. Several windows will appear on the screen, with the layout depending on settings within your product.

We want to start a new project. Select the **File** menu option, then click **New**, then **Project**. The following dialog box should appear:



Under **Installed Templates**, make sure **Visual C#** is selected. We will always be building windows applications, so select **Windows Form Application**.

This window also asks where you want to save your project. In the **Name** box, enter the name (I used **First Project**) of the folder to save your project in.

**Location** should show the directory your project folder will be in. You can **Browse** to an existing location or create a new directory by checking the indicated box. For these notes, we suggest saving each of your project folders in the same directory. For the course notes, all project folders are saved in the **\HomeVCS\HomeVCS Projects** folder.

Once done, click **OK**. The Visual C# development environment will appear. Your project will consist of a single Windows form. We will use this simple little project to demonstrate the steps in building a project using Visual C#.

# Visual C# Integrated Development Environment (IDE)

The **Visual C# IDE** is where we build and test our application via implementation of the three steps of application development (draw controls, assign properties, write code). As you progress through this course, you will learn how easy-to-use and helpful the IDE is. There are many features in the IDE and many ways to use these features. Here, we will introduce the IDE and some of its features. You must realize, however, that its true utility will become apparent as you use it yourself to build your own applications.

Several windows appear when you start Visual C#. Each window can be viewed (made visible or active) by selecting menu options, depressing function keys or using the displayed toolbar. As you use the IDE, you will find the method you feel most comfortable with. Also, windows can ‘float’ or be ‘docked’. Right-clicking a window can change this option. We’ll leave it to you to configure the development environment to your liking.

The title bar area shows you the name of your project:



When your project is running, the title bar will also show the mode Visual C# is operating in. Visual C# operates in three modes.

- **Design** mode - used to build application
- **Run** mode - used to run the application
- **Debug** (or **Break**) mode - application halted and debugger is available

We focus here on the **design** mode. You should, however, always be aware of what mode you are working in.

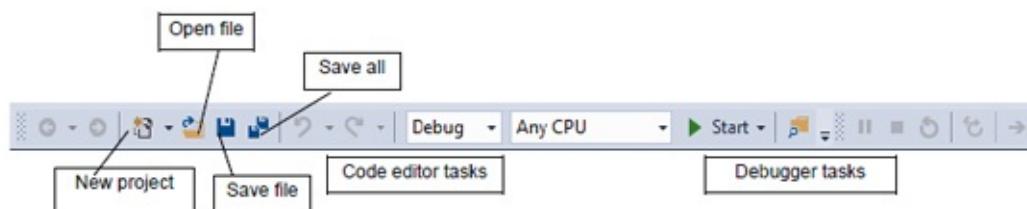
Under the title bar is the **Menu**. This menu is dynamic, changing as you try to do different things in Visual C#. When you start working on a project, it should look like this:



You will become familiar with each menu topic as you work through the course. Briefly, they are:

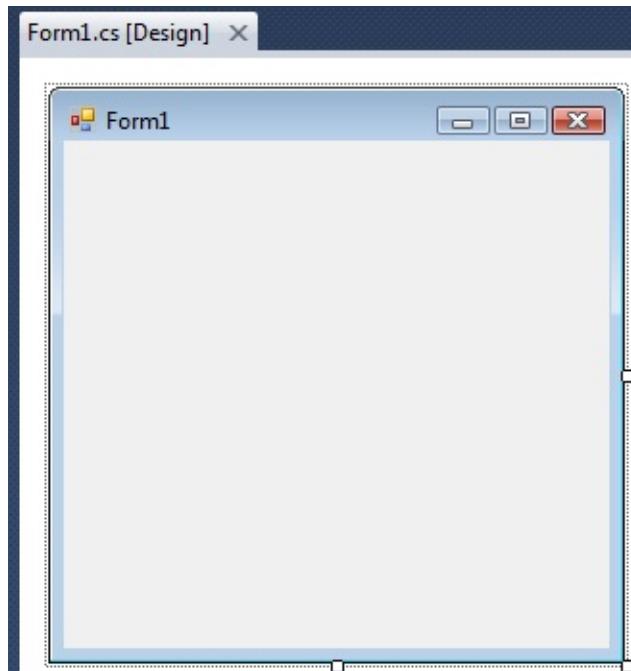
<b>File</b>	Use to open/close projects and files. Use to exit Visual C#.
<b>Edit</b>	Used when writing code to do the usual editing tasks of cutting, pasting, copying and deleting text
<b>View</b>	Provides access to most of the windows in the IDE
<b>Project</b>	Allows you to add/delete components to your project
<b>Build</b>	Controls the compiling process
<b>Debug</b>	Comes in handy to help track down errors in your code (works when Visual C# is in <b>Debugging</b> mode)
<b>Team</b>	Used when several people work on one project.
<b>Format</b>	Used to modify appearance of your graphic interface.
<b>Tools</b>	Allows custom configuration of the IDE. Be particularly aware of the <b>Options</b> choice under this menu item. This choice allows you to modify the IDE to meet any personal requirements.
<b>Test</b>	Allows you to compile and run your completed application (go to <b>Running</b> mode)
<b>Analyze</b>	Performs analytics on your code.
<b>Window</b>	Lets you change the layout of windows in the IDE
<b>Help</b>	Perhaps, the most important item in the Menu. Provides access to the Visual C# on-line documentation via help contents, index or search. Get used to this item!

The **View** menu also allows you to choose from a myriad of toolbars available in the Visual C# IDE. Toolbars provide quick access to many features. The **Standard** (default) toolbar appears below the Menu:



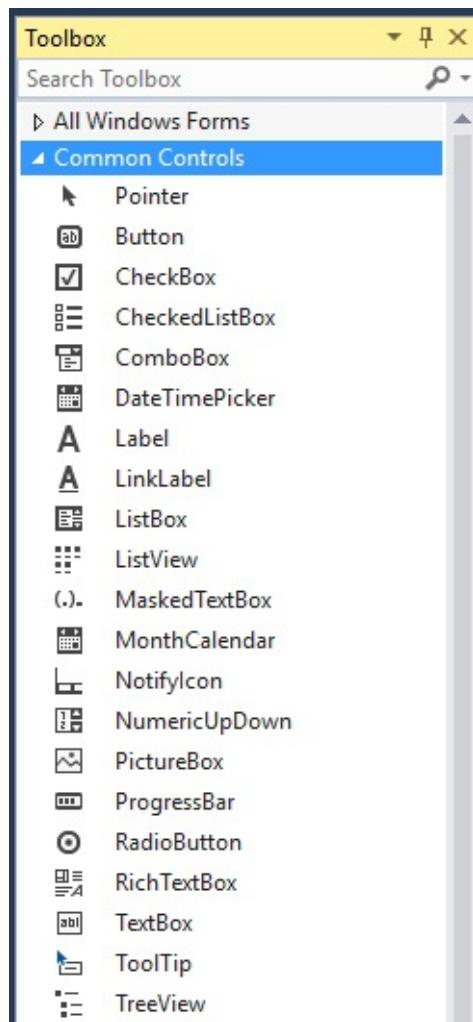
If you forget what a toolbar button does, hover your mouse cursor over the button until a descriptive tooltip appears. We will discuss most of these toolbar functions in the remainder of the IDE information.

In the middle of the Visual C# IDE is the **Design Window**. This window is central to developing Visual C# applications. Various items are available by selecting tabs at the top of the window. The primary use is to draw your application on a form and, later, to write code. Forms are selected using the tab named **FormName.cs [Design]**; our example form has the default name **Form1**:



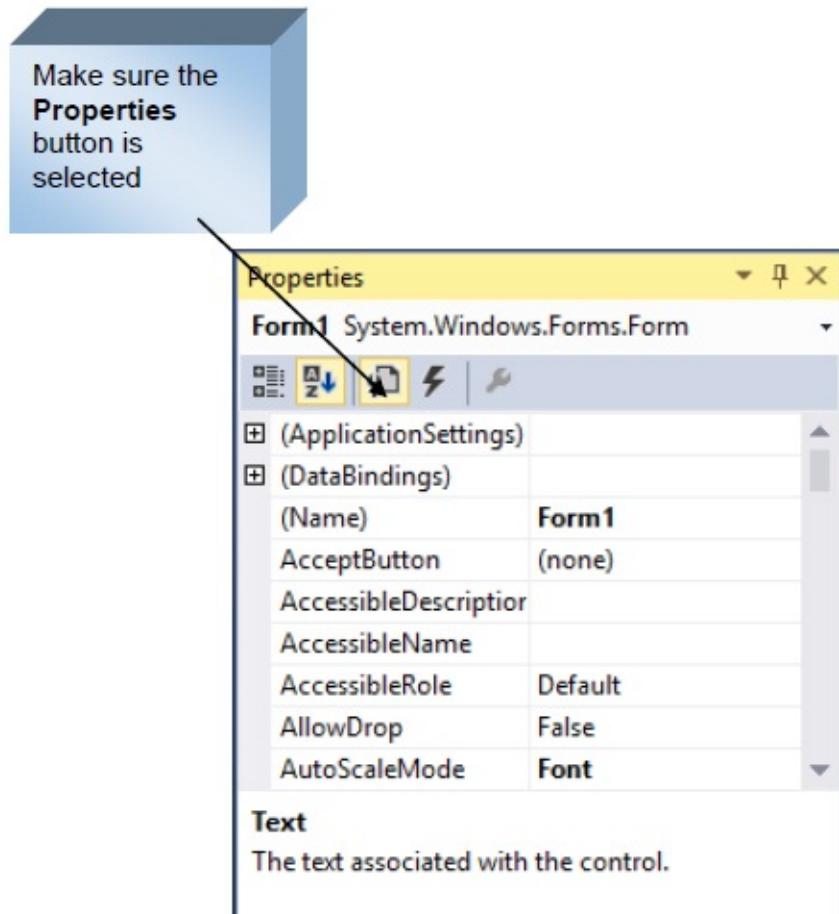
The corresponding code for a form is found using the **FormName.cs** tab (not shown yet). The design window will also display help topics and other information you may choose.

The **Toolbox** is the selection menu for controls used in your application. It can be viewed by clicking the **Toolbox** button in the toolbar. It is active when a form is shown in the design window:



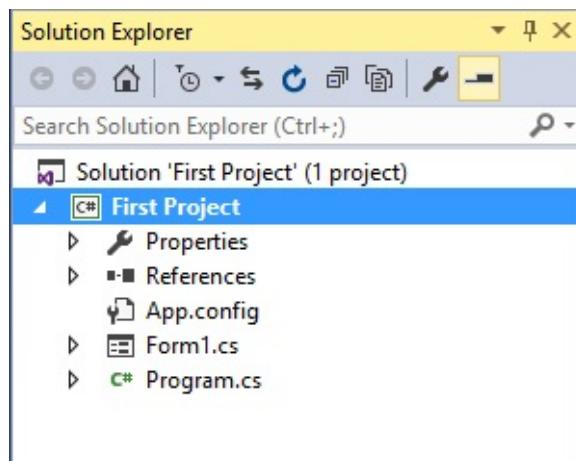
Make sure you are viewing the **Common Controls**. Many of these tools will look familiar to you. They are used in a wide variety of Windows applications you have used (the Visual C# IDE even uses them!) We will soon look at how to move a control from the toolbox to a form.

The **Properties Window** serves two purposes. Its primary purpose is to establish design mode (initial) property values for objects (controls). It can also be used to establish event methods for controls. Here, we just look at how to work with properties. To do this, click the **Properties** button in the task bar:



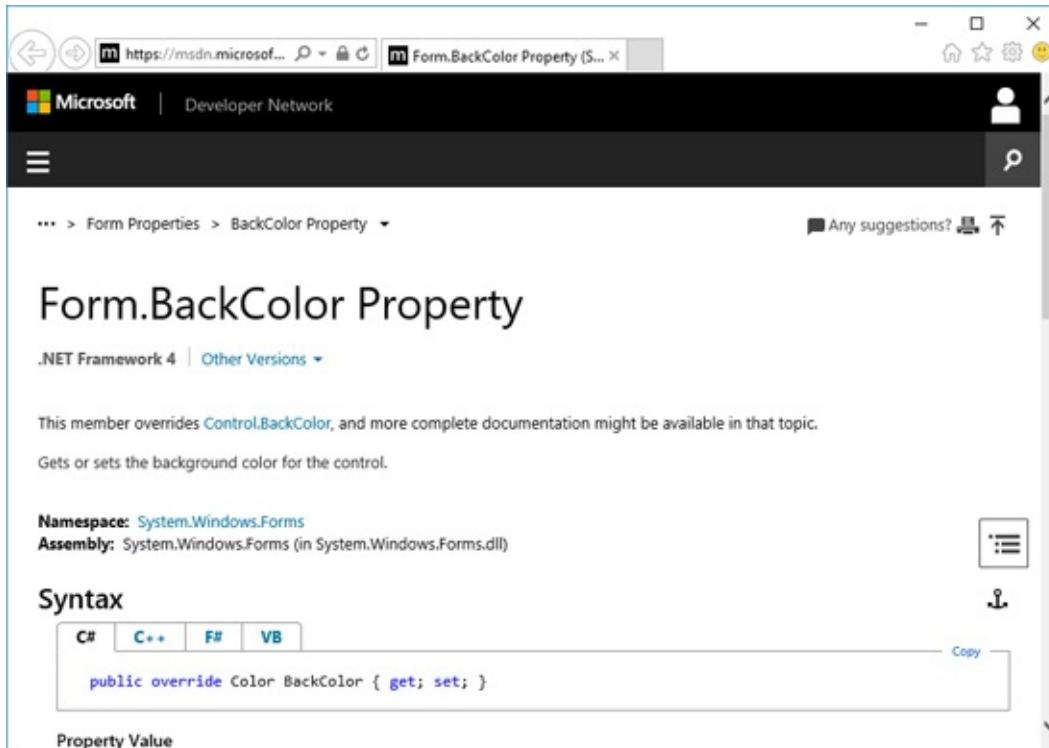
The drop-down box at the top of the window lists all objects in the current form. Under this box are the available properties for the active (currently selected) object. Two property views are available: **Alphabetic** and **Categorized** (selection is made using menu bar under drop-down box). Help with any property can be obtained by highlighting the property of interest and pressing <F1>. We will examine how to assign properties after placing some controls on a form.

The **Solution Explorer Window** displays a list of all forms and other files making up your application. To view a form listed in this window, simply double-click the file name. Or, highlight the file and press <Shift>-<F7>. Or, you can obtain a view of the **Form** or **Code** windows (window containing the actual C# coding) from the Project window, using the toolbar near the top of the window. As we mentioned, there are many ways to do things using the Visual C# IDE.



The help facilities of Visual C# are vast and very useful. As you work more and more with Visual C#, you will become very dependent on these facilities. Help is obtained using three different windows (all accessed using the **Help** menu item).

A great feature about the Visual C# on-line help system is that it is ‘context sensitive.’ What does this mean? Well, let’s try it. Start Visual C# and start a new project. Go to the properties window. Scroll down the window displaying the form properties and click on the word **BackColor**. The word is highlighted. Press the <F1> key. A screen of information about the **Form.BackColor** property appears:



The help system has intelligence. It knows that since you highlighted the word BackColor, then pressed <F1> (<F1> has always been the key to press when you need help), you are asking for help about BackColor. Anytime you press <F1> while working in Visual C#, the program will look at where you are working and try to determine, based on context, what you are asking for help about. It looks at things like highlighted words in the properties window or position of the cursor in the code window.

As you work with Visual C#, you will find you will use ‘context-sensitive’ help a lot. Many times, you can get quick answers to questions you might have. Get used to relying on the Visual C# on-line help system for assistance.

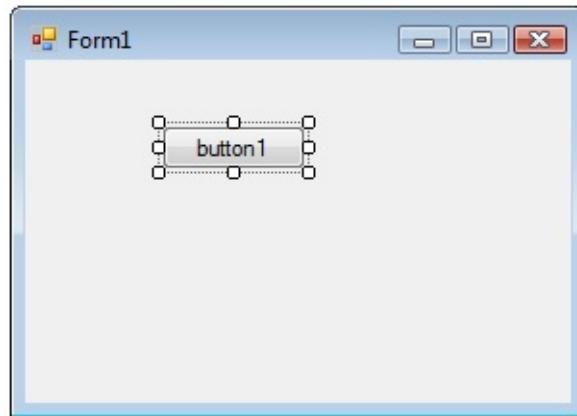
This completes our quick tour of the Visual C# IDE. We now turn our attention to building our first application using these three steps:

- Draw the user interface (place controls on the form).
- Set control properties
- Write code

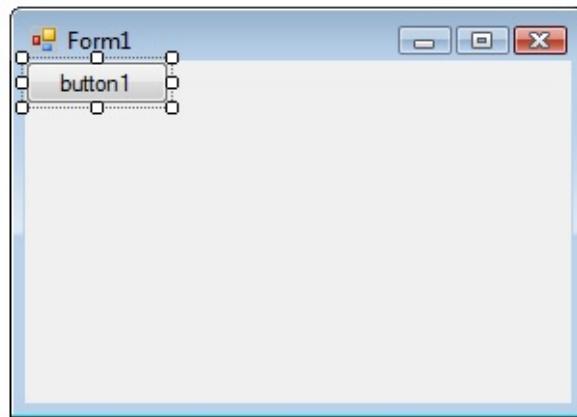
# Drawing the User Interface

The first step in developing a Visual C# Windows application is to draw the user interface on the form. Make sure the Form appears in the Design window. There are four ways to place controls on a form:

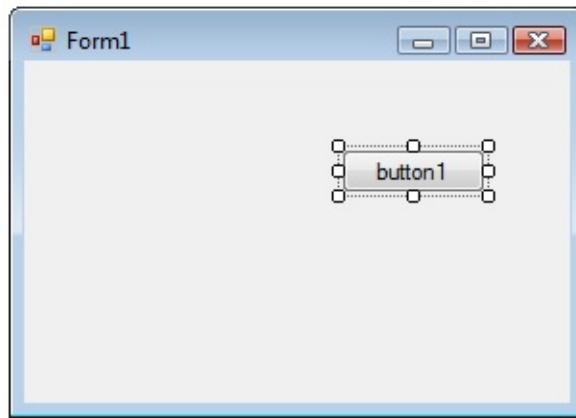
1. Click the tool in the toolbox and hold the mouse button down. Drag the selected tool over the form. When the cursor pointer is at the desired upper left corner, release the mouse button and the default size control will appear. This is the classic “drag and drop” operation. Here is a button control “dropped” on the form:



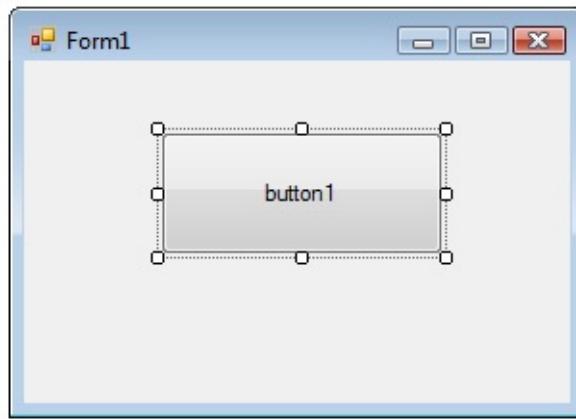
2. Double-click the tool in the toolbox and it is created with a default size on the form. You can then move it or resize it. Here is a button control placed on the form using this method:



3. Click the tool in the toolbox, then move the mouse pointer to the form window. The cursor changes to a crosshair. Place the crosshair at the upper left corner of where you want the control to be and click the left mouse button. The control will appear at the clicked point. Here is a default size button placed on the form using this technique:

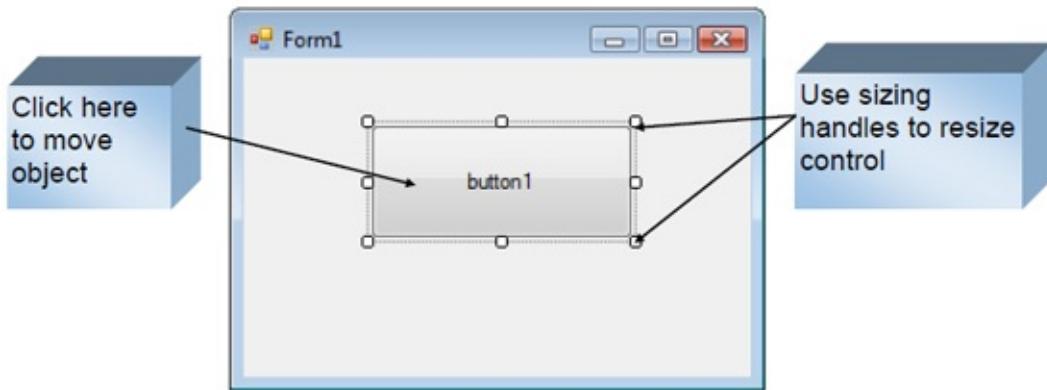


4. Click the tool in the toolbox, then move the mouse pointer to the form window. The cursor changes to a crosshair. Place the crosshair at the upper left corner of where you want the control to be, press the left mouse button and hold it down while dragging the cursor toward the lower right corner. A rectangle will be drawn. When you release the mouse button, the control is drawn in the rectangle. A big button drawn using this method:



To **move** a control you have drawn, click the object in the form (a cross with arrows will appear). Now, drag the control to the new location. Release the mouse button.

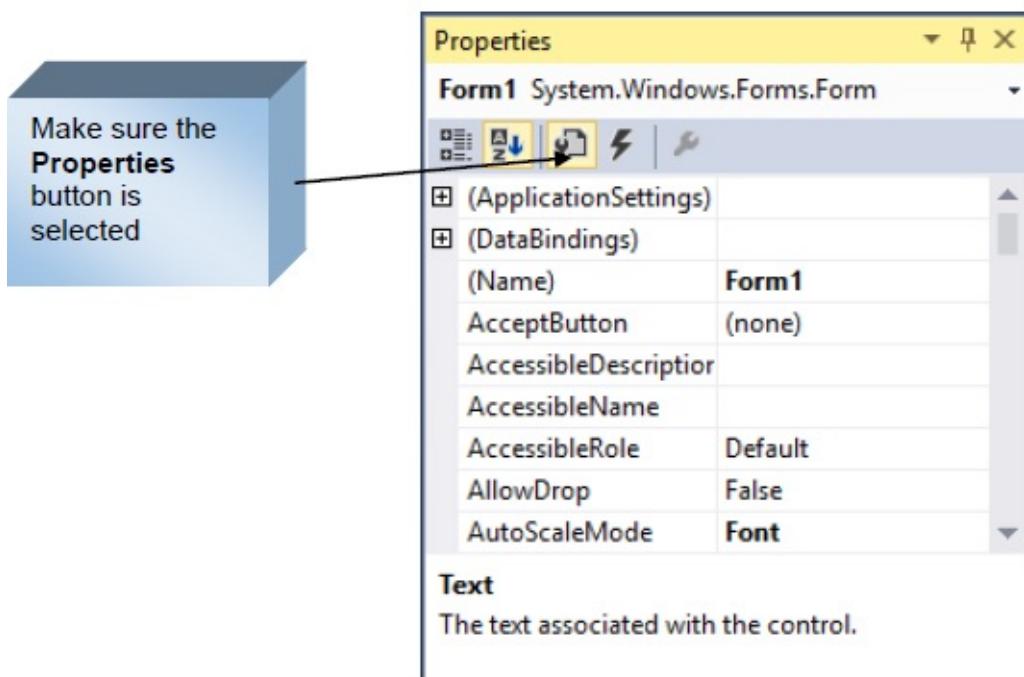
To **resize** a control, click the control so that it is selected (active) and sizing handles appear. Use these handles to resize the object.



To delete a control, select that control so it is active (sizing handles will appear). Then, press <**Delete**> on the keyboard. Or, right-click the control. A menu will appear. Choose the **Delete** option. You can change your mind immediately after deleting a control by choosing the **Undo** option under the **Edit** menu.

# Setting Properties of Controls at Design

Each form and control has **properties** assigned to it by default when you start up a new project. There are two ways to display the properties of an object. The first way is to click on the object (form or control) in the form window. Sizing handles will appear on that control. When a control has sizing handles, we say it is the **active** control. Now, click on the Properties window or the Properties window button in the tool bar. The second way is to first click on the Properties window. Then, select the object from the drop-down box at the top of the Properties window. When you do this, the selected object (control) will now be active (have sizing handles). Shown is the **Properties** window (make sure the **Properties** button, not the **Events** button is selected in the toolbar) for the Form object:



The drop-down box at the top of the Properties Window is the **Object** box. It displays the name of each object in the application as well as its type. This display shows the **Form** object. The **Properties** list is directly below this box. In this list, you can scroll through the list of properties for the selected object. You select a property by clicking on it. Properties can be changed by typing a new value or choosing from a list of predefined settings (available as a drop down

list). Properties can be viewed in two ways: **Alphabetic** and **Categorized** (selected using the menu bar under the Object box). We always use the Alphabetic view.

A very important property for each control is its **Name**. The name is used by Visual C# to refer to a particular object or control in code. A convention has been established for naming Visual C# controls. This convention is to use a three letter (lower case) prefix (identifying the type of control) followed by a name you assign. A few of the prefixes are (we'll see more as we progress):

Control	Prefix	Example
Form	frm	frmWatch
Button	btn	btnExit, btnStart
Label	lbl	lblStart, lblEnd
Text Box	txt	txtTime, txtName
Menu	mnu	mnuExit, mnuSave
Check box	chk	chkChoice

It is suggested to use a mixture of upper and lower case letters for better readability. But, be aware that Visual C# is a case-sensitive language, meaning the names **frmWatch** and **FRMWATCH** would be not assumed to be the same name.

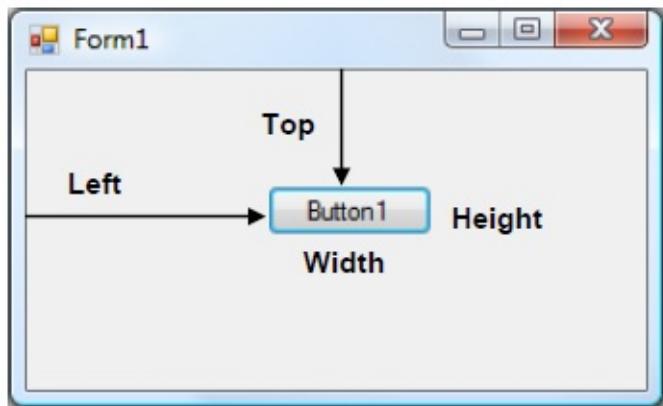
Control (object) names can be up to 40 characters long, must start with a letter, must contain only letters, numbers, and the underscore (\_) character. Names are used in setting properties at run-time and also in establishing method names for control events. Use meaningful names that help you (or another programmer) understand the type and purpose of the respective controls.

Another set of important properties for each control are those that dictate position and size. There will be times you want to refer to (or change) these properties. There are four properties:

- Left** Distance from left side of form to left edge of control, in pixels
- Top** Distance from title bar of form to top edge of control, in pixels

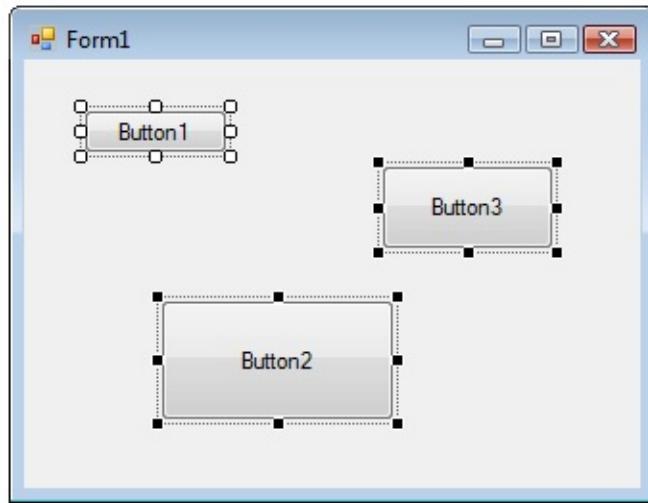
- Width**    Width of control, in pixels  
**Height**    Height of control, in pixels

Pictorially, these properties for a button control are:



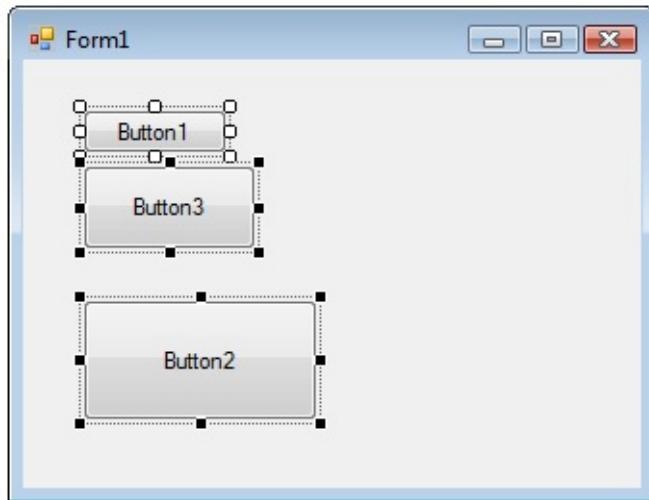
Finding these properties in the Properties window is a bit tricky. To find the Left and Top properties, expand the **Location** property. The displayed X value is the Left property, while Y is the Top property. To find the Width and Height properties, expand the **Size** property – the width and height are displayed and can be modified.

It is possible to multi-select several controls and set common properties. To select multiple controls, first click one control to make it active. Then, for all subsequent choices, hold down the <Ctrl> key while clicking the desired controls. For example, here is a form with all three button controls selected (**button1** selected first):

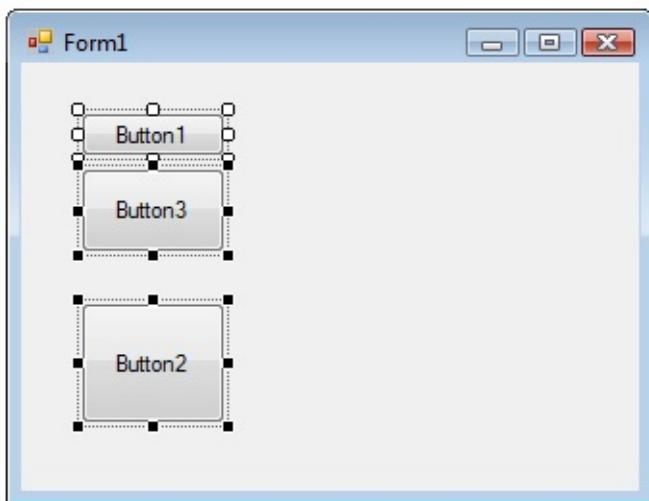


When you go to the **Properties** window with multiple controls selected, no control name will be listed in the drop-down box. Rather, a set of common properties for the selected controls will be listed. Any properties changed in this situation will affect all selected controls. This approach to setting properties is useful for such properties as colors and fonts. You'll find you use it often in projects using many similar controls.

It is also possible to “group align” and “group size” controls. The **Format** item in the main menu provides such capability. Open that item in the menu. You can align left, top, right or bottom properties. You can make controls the same height and/or width. And, there are many other format possibilities. One control in a group is the reference control that all other selected controls will match. The reference control is selected by clicking on a single control in a selected group (the reference control will have white sizing handles). For example, with the three buttons above, select **button1** as the reference. Then, choose **Format**, then **Align**, then **Lefts**, the buttons will adjust as so:



Now choose **Format**, then **Make Same Size**, then **Width**, you obtain:



You'll find the **Format** menu option is very handy at times.

# Setting Properties at Run Time

In addition to setting properties at design time, you can set or modify properties while your application is running. To do this, you must write some code. The code format is:

```
objectName.PropertyName = newValue;
```

Such a format is referred to as dot notation. For example, to change the **BackColor** property of a button named **btnStart**, we'd type:

```
btnStart.BackColor = Color.Blue;
```

You've just seen your first line of code in Visual C#. Good naming conventions make it easy to understand what's going on here. The button named **btnStart** will now have a blue background. We won't learn much code in this first chapter (you'll learn a lot in Chapter 2), but you should see that the code that is used is straightforward and easy to understand.

# How Names are Used in Control Events

The names you assign to controls are also used by Visual C# to set up a framework of event-driven methods for you to add code to. Hence, proper naming makes these methods easier to understand.

The format for each of these methods is:

```
private void controlName_Event(Arguments)  
{  
}  
}
```

where **Arguments** provides information needed by the method to do its work.

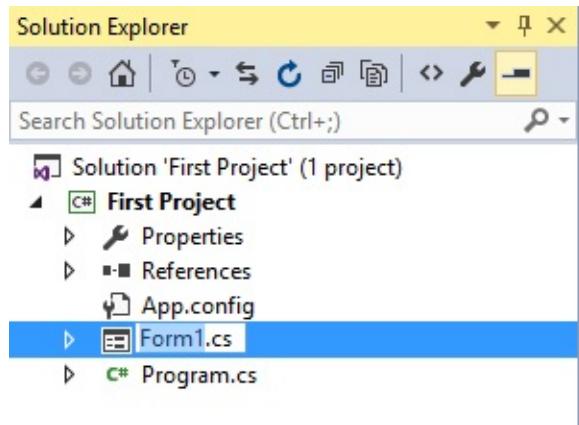
Visual C# provides the header line with its information and arguments and left and right curly braces (all code goes between these two braces) delineating the start and end of the method. Don't worry about how to provide any code right now. Just notice that with proper naming convention, it is easy to identify what tasks are associated with a particular event of a particular control.

# Use of Form Name Property

When setting run-time properties or accessing events for the **Form** object, the **Name** property is not used in the same manner as it is for other controls. To set properties, rather than use the name property, the keyword **this** is used. Hence, to set a form property at run-time, use:

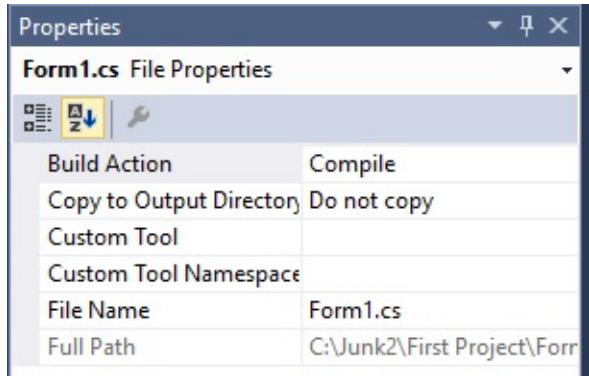
```
this.PropertyName = value;
```

We also need to distinguish between the form **Name** property and the **File Name** used within Visual C# to save the form file (which includes control descriptions and all code associated with a particular form). The Name property and File Name are two different entities. We have seen how the form Name property is used. A look at the Solution Explorer window for a sample application shows how the form File Name is used:



The name preceding the **cs** extension (**Form1**) is the form file name. Again, this is not the same as the name assigned in the Form Properties window.

To change the form file name, click the file in the Solution Explorer window to see the **File Properties** window (this is not the same as the Form Properties window):

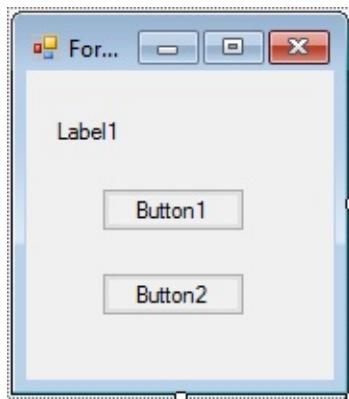


In this window, you change the File Name by simply typing another name (with the **.cs** extension). Visual C# will then automatically save the form file with this new name. This name is then used to refer to the form file. This is the name you would use if you want to load a particular form into a Visual C# solution or project.

# Writing Code

The last step in building a Visual C# application is to write code using the **C#** language. This is the most time consuming task in any Visual C# application. It is also the most fun and most rewarding task.

We will write a little code here using the example we started. Clear any controls you may have placed on the example's form. Place a label control (displays text) and two button controls on the form. Make it look something like this:



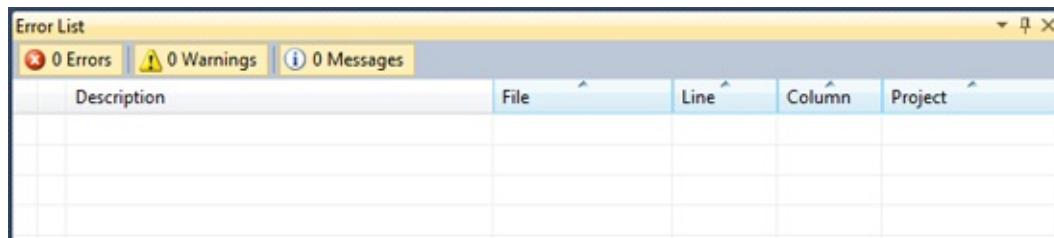
Now, set the **Name** property of **button1** to **btnHello** and the **Text** property to **Hello**. For **button2**, use a **Name** property of **btnGoodbye** and a **Text** property of **Goodbye**, so the finished form is like this:



As controls are added to a form, Visual C# automatically builds a framework of all event methods. We simply add code to the event methods we want our application to respond to. And, if needed, we write general methods. For those

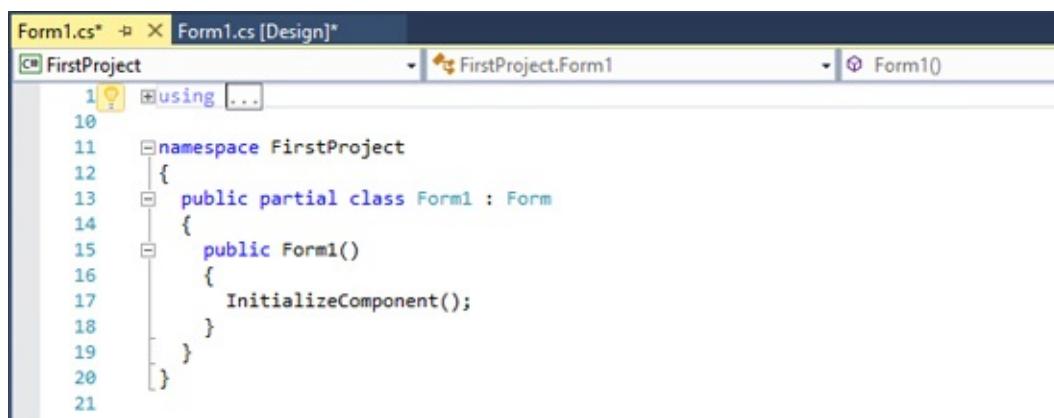
who may have never programmed before, the code in these methods is simply a line by line list of instructions for the computer to follow

As you write code, any errors you make are identified in the **Error List Window**. To view this window, choose the **View** menu option. Then, select **Other Windows**, then **Error List**. The **Error List** window will appear:



Errors are listed under **Description**. You will use this window a lot while writing code.

Code is placed in the **Code Window**. Typing code in the code window is just like using any word processor. You can cut, copy, paste and delete text (use the **Edit** menu or the toolbar). Learn how to access the code window using the menu (**View**), toolbar, or by pressing <F7> (and there are still other ways) while the form is active. Here is the Code window (**Form1.cs**) for the example project we have been playing with:



The header line (**namespace First\_Project**) starts the code. In Visual C#, everything that makes up your project is called a **namespace**. Your form is called a **class**. The line **public Form1()** begins the form **constructor**. The constructor consists of a single line saying **InitializeComponent();**. This code accesses a routine written by the Visual C# environment to set up the form that

you designed. Notice again the use of curly braces to start and end code segments. You don't have to worry much about any of this code – just don't change any of it. All the code we write will go after this constructor. We need to write event methods for the two button controls.

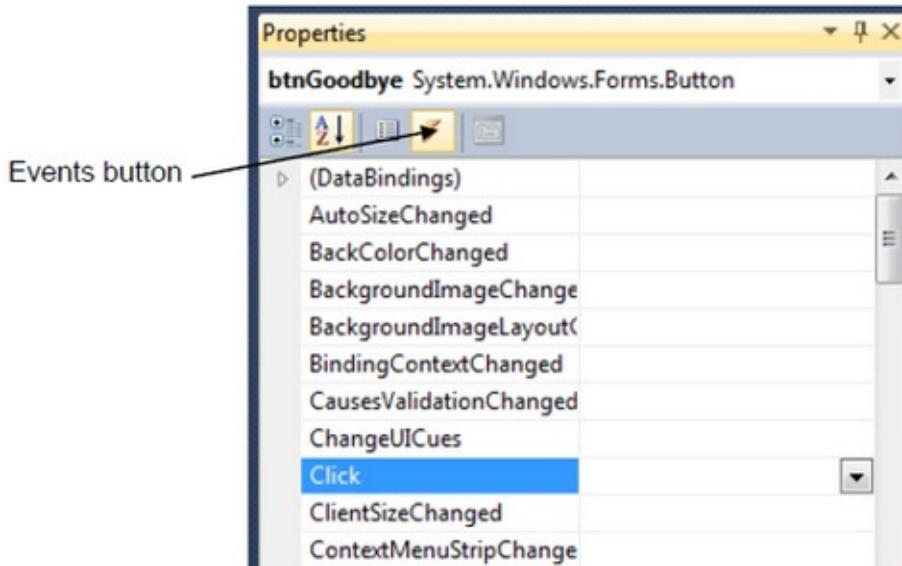
There are two ways to establish event methods for controls – one directly from the form and one using the properties window. Let's look at both. Every control has a **default event method**. This is the method most often used by the control. For example, a button control's default event method is the **Click** event, since this is most often what we do to a button. To access a control's default event method, you simply double-click the control on the form. For example, if you double-click the button named **btnHello** in this example, the bottom of the code window will display:

```
private void btnHello_Click(object sender, EventArgs e)
{
}
```

Let's spend some time looking at this method – all event methods used in Visual C# look just like this. The time spent clarifying things now will be well worth it.

The assigned name for the event method is seen to be **btnHello\_Click**. Our naming convention tells us this is the method executed when the user clicks on the **btnHello** button. The method has two arguments: **sender** and **e**. **Sender** tells the method what control caused the **Click** event to occur and **e** gives us some information about the event. Following the header line are a matched set of curly braces (**{ }**). The code for the event method will go between these two braces.****

Though simple and quick, double-clicking a control to establish a control event method will not work if you are not interested in the default event. In that case, you need to use the properties window. Recall we mentioned that the properties window is not only used to establish properties, but also event methods. To establish an event method using the properties window, click on the **Events** button (appears as a lightning bolt) in the properties window toolbar:



The active control's events will be listed (the default event will be highlighted). To establish an event method, scroll down the listed events and double-click the one of interest. The selected event method will appear in the code window. Add the **Click** event for **btnGoodbye**.

There are several ways to locate a previously established method. One is to double-click the event name in the properties window, just as we did to create the method. Once you double-click the name, the code window will display the selected method. If the desired method is a default method, double-clicking the control on the form will take you to that method in the code window.

Another way to locate an event method is via the code window. At the top of the code window are two dropdown boxes, the **method list** is on the right. Selecting this box will provide a list of all methods in the code window. Find the event method of interest and select it. The selected event will appear in the code window:

```
Form1.Designer.cs* Form1.cs* & X Form1.cs [Design]*  
FirstProject FirstProject.Form1  
1  using ...  
10  
11  namespace FirstProject  
12  {  
13      public partial class Form1 : Form  
14      {  
15          public Form1()  
16          {  
17              InitializeComponent();  
18          }  
19  
20          private void btnHello_Click(object sender, EventArgs e)  
21          {  
22          }  
23  
24          private void btnGoodbye_Click(object sender, EventArgs e)  
25          {  
26          }  
27  
28      }  
29  }  
30 }  
31
```

Deleting an existing event method can be tricky. Simply deleting the event method in the code window is not sufficient. When you add an event method to your application, Visual C# writes a line of code to connect the method to your control. That line of code remains after you delete the method. If you delete the method and try to run your application, an error will occur. In the Task Window will be a message saying your application does not contain a definition for a specific event method. Double-click the message and C# will take you to the offending statement (the one that connects the control and the method). Delete that line and your code will be fixed.

The preferred way to delete a method is to select the desired method using the properties window (make sure the control of interest is the active control). Once the event is selected, highlight the event method name on the right side and press the **Delete** key. This process deletes the event from the code window and deletes the line of code connecting the method and control.

In the little example here, we will just be giving you code to type in the code window. There are a few rules to pay attention to as you type Visual C# code:

- Visual C# code requires perfection. All words must be spelled correctly.
- Visual C# is case-sensitive, meaning upper and lower case letters are considered to be different characters. When typing code, make sure you use upper and lower case letters properly

- Visual C# ignores any “**white space**” such as blanks. We will often use white space to make our code more readable.
- Curly **braces** are used for grouping. They mark the beginning and end of programming sections. Make sure your Visual C# programs have an equal number of left and right braces. We call the section of code between matching braces a **block**.
- It is good coding practice to **indent** code within a block. This makes code easier to follow. The Visual C# environment automatically indents code in blocks for you.
- Every Visual C# statement will end with a semicolon. A **statement** is a program expression that generates some result. Note that not all Visual C#ions are statements (for example, the line defining the Main method has no semicolon).

Go to the code window and add a single line to the **btnHello\_Click** event:

```
label1.Text = "Hello World!!";
```

At this point, the code window should appear as:

```

Form1.Designer.cs* Form1.cs* X Form1.cs [Design]*

FirstProject FirstProject.Form1
    btnHello_Click(object sender, EventArgs e)
        label1.Text = "Hello World!";
    }
}

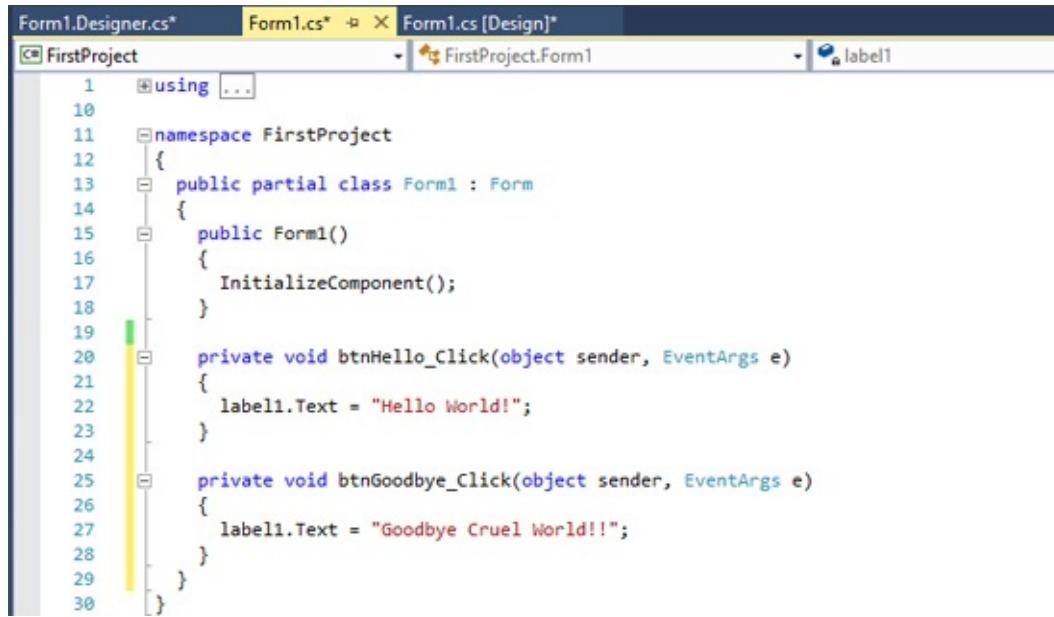
```

This code says when the user clicks **btnHello**, have the label control display Hello World!.

Next, locate the **btnGoodbye\_Click** event and add this single line of code::

```
label1.Text = "Goodbye Cruel World!!";
```

A different message will appear when the other button is clicked. Now, the code window should appear as:

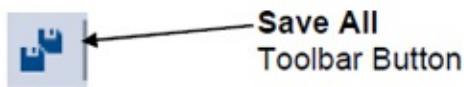


```
Form1.Designer.cs*  Form1.cs*  X  Form1.cs [Design]*  
FirstProject  FirstProject.Form1  label1  
1  using ...  
10  
11  namespace FirstProject  
12  {  
13      public partial class Form1 : Form  
14      {  
15          public Form1()  
16          {  
17              InitializeComponent();  
18          }  
19  
20          private void btnHello_Click(object sender, EventArgs e)  
21          {  
22              label1.Text = "Hello World!";  
23          }  
24  
25          private void btnGoodbye_Click(object sender, EventArgs e)  
26          {  
27              label1.Text = "Goodbye Cruel World!!";  
28          }  
29      }  
30  }
```

We're almost ready to run our first little project.

# Saving a Visual C# Project

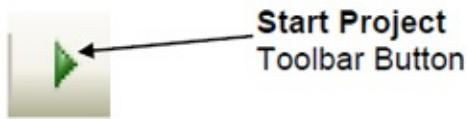
When a project is created in Visual C#, it is automatically saved in the location you specify. If you are making lots of changes, you might occasionally like to save your work prior to running the project. Do this by clicking the **Save All** button in the Visual C# toolbar. Look for a button that looks like several floppy disks. (With writeable CD's so cheap, how much longer do you think people will know what a floppy disk looks like? – many new machines don't even have a floppy disk drive!)



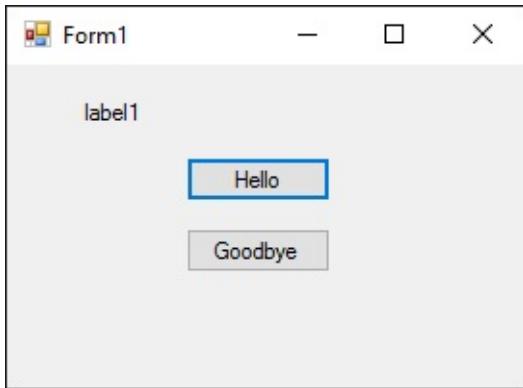
Always make sure to save your project before running it or before leaving Visual C#.

# Running a Visual C# Project

Run your project by clicking the **Start** button on the toolbar:

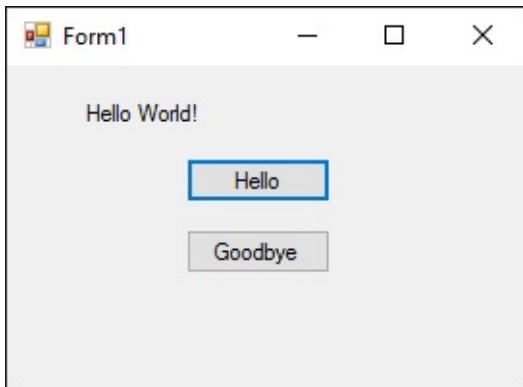


or by pressing <F5>. The form should appear as:



Note, too, the words (**Running**) in the IDE title bar, indicating the project is now in run mode, as opposed to design mode.

Click the **Hello** button to see the classic “Hello World” message seen in every programming book:



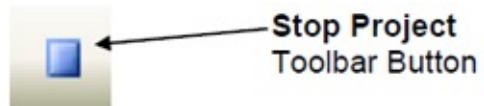
You generated a **Click** event which processed the code in the **btnHello\_Click** event method.

Click the **Goodbye** button to see the other message:



You generated a **Click** event which processed the code in the **btnGoodbye\_Click** event method.

You can now stop the project by clicking the **Stop** button:



# Chapter Review

This completes our overview of the Visual C# IDE and our review of the steps in building a Visual C# project. If you've used Visual C# before, this material should be familiar. If you've programmed with other languages, you should have a fairly clear understanding of what's going on.

After completing this chapter, you should understand:

- A little of the history of Visual C#.
- The concept of an event-driven application
- The parts of a Visual C# application (form, control, property, event, ...)
- The various windows of the Visual C# integrated development environment
- How to use the Visual C# help system
- The three steps in building a Visual C# project
- Different ways to place controls on a form
- Methods to set properties for controls
- Proper control naming convention
- How to add event methods and code using the code window

Before starting the Homework Projects in Chapter 4, we need to review the language of Visual C# (Chapter 2) and learn how to debug projects (Chapter 3).

2

## Overview of Visual C# Programming

# Review and Preview

In the first chapter, we found there were three primary steps involved in developing a Windows application using Visual C#:

1. Draw the user interface
2. Assign properties to controls
3. Write code for events

In this chapter, we are primarily concerned with Step 3, writing code. We will provide an overview of many of the elements of the language used in Visual C#—known as C#. This chapter is essentially a self-contained guide to the C# language. This will give us the foundation needed to begin building some Homework Projects.

# A Brief History of C#

It's interesting to see just where the C# language fits in the history of some other computer languages. You will see just how new Visual C# is! In the early 1950's most computers were used for scientific and engineering calculations. The programming language of choice in those days was called **FORTRAN**. FORTRAN was the first modern language and is still in use to this day (after going through several updates). In the late 1950's, bankers and other business people got into the computer business using a language called **COBOL**. Within a few years after its development, COBOL became the most widely used data processing language. And, like FORTRAN, it is still being used today.

In the 1960's, two professors at Dartmouth College decided that "everyday" people needed to have a language they could use to learn programming. They developed **BASIC** (**B**eginner's **A**ll-Purpose **S**ymbolic **I**nstruction **C**ode). BASIC (and its successors, GW-Basic, Visual Basic, Visual Basic Express) is probably the most widely used programming language. Many dismiss it as a "toy language," but BASIC was the first product developed by a company you may have heard of – Microsoft! And, BASIC has been used to develop thousands of commercial applications.

**C#** had its beginnings in 1972, when AT&T Bell Labs developed the **C** programming language. It was the first, new scientific type language since FORTRAN. If you've ever seen a C program, you will notice many similarities between C# and C. Then, with object-oriented capabilities added, came **C++** in 1986 (also from Bell Labs). This was a big step.

In 2001, Microsoft introduced its .NET platform and a new language was introduced – C#. It represented a streamlined version of C and C++, with a little Java thrown in. This chapter provides an overview of the C# language used in the Visual C# environment. If you've ever used another programming language, you will see equivalent structures in the language of Visual C#.

# Rules of C# Programming

Before starting our review of the C# language, let's review some of the rules of C# programming seen in the first class:

- C# code requires perfection. All words must be spelled correctly.
- C# is case-sensitive, meaning upper and lower case letters are considered to be different characters. When typing code, make sure you use upper and lower case letters properly.
- C# ignores any “**white space**” such as blanks. We will often use white space to make our code more readable.
- Curly **braces** are used for grouping. They mark the beginning and end of programming sections. Make sure your C# programs have an equal number of left and right braces. We call the section of code between matching braces a **block**.
- It is good coding practice to **indent** code within a block. This makes code easier to follow. The Visual C# environment automatically indents code in blocks for you.
- Every C# statement will end with a semicolon. A **statement** is a program expression that generates some. Note that not all C# expressions are statements (for example, the line defining the main method has no semicolon).

# Variables

Variables are used by Visual C# to hold information needed by an application. Variables must be properly named. Rules used in naming variables:

- No more than 40 characters
- They may include letters, numbers, and underscore (\_)
- The first character must be a letter which, by convention, is usually lower case
- You cannot use a reserved word (keywords used by Visual C#)

Use meaningful variable names that help you (or other programmers) understand the purpose of the information stored by the variable.

**Examples** of acceptable variable names:

startingTime	interest_Value	letter05
johnsAge	number_of_Days	timeOfDay

# Visual C# Data Types

Each variable is used to store information of a particular **type**. Visual C# has a wide range of data types. You must always know the type of information stored in a particular variable.

**bool** (short for Boolean) variables can have one of two different values: **true** or **false** (reserved words in Visual C#). Boolean variables are helpful in making decisions. There is one possible point of confusion when working with Boolean values. Note some control properties are Boolean, for example, the Enabled property. When setting such a property at design time, the choices are **True** or **False** (using upper case letters). When setting Boolean values in code (either setting control properties or some other variable), the choices are **true** or **false** (using lower case letters). Be aware of this when writing code. A common error is to use upper case values, rather than the proper lower case values.

If a variable stores a whole number (no decimal), there are three data types available: **short**, **int** or **long**. Which type you select depends on the range of the value stored by the variable:

Data Type	Range
short	-32,678 to 32,767
int	-2,147,483,648 to 2,147,483,647
long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

We will almost always use the **int** type in our work.

If a variable stores a decimal number, there are two data types: **float** or **double**. The double uses twice as much storage as float, providing more precision and a wider range. Examples:

Data Type	Value
float	3.14
double	3.14159265359

Visual C# is a popular language for performing string manipulations. These manipulations are performed on variables of type **string**. A string variable is just that - a string (list) of various characters. In Visual C#, string variable values are enclosed in quotes. Examples of string variables:

“Visual C#”      “012345”      “Title Author”

Single character string variables have a special type, type **char**, for character type. Examples of character variables (enclosed in single quotes):

‘a’      ‘1’      ‘V’      ‘\*’

Visual C# also has great facilities for handling dates and times. The **DateTime** variable type stores both dates and times. Using different formatting techniques (consult the Visual C# help system) allow us to display dates and times in any format desired. The **TimeSpan** variable type helps in doing date math; for example, the difference between two dates.

A last data type is type **Object**. That is, we can actually define a variable to represent any Visual C# object, like a button or form. We will see the utility of the Object type as we progress in the course.

# Variable Declaration

Once we have decided on a variable name and the type of variable, we must tell our Visual C# application what that name and type are. We say, we must **explicitly declare** the variable.

There are many advantages to **explicitly** typing variables. Primarily, we insure all computations are properly done, mistyped variable names are easily spotted, and Visual C# will take care of insuring consistency in variable names. Because of these advantages, and because it is good programming practice, we will always explicitly type variables.

To **explicitly** type a variable, you must first determine its **scope**. Scope identifies how widely disseminated we want the variable value to be. We will use three levels of scope:

- Block level
- Method level
- Form level

**Block level** variables are only usable within a single block of code. We will see these when we discuss loops.

The value of **method level** variables are only available within a method. Such variables are declared within a method, using the variable type as a declarer:

```
int myInt;  
double myDouble;  
string myString, yourString;
```

These declarations are usually placed after the opening left curly brace of a method.

**Form level** variables retain their value and are available to all methods within that form. Form level variables are declared in the code window right after the

**method** with the form constructor (generated automatically by Visual C#), outside of any other method:

```
Form1.cs*  Form1.cs [Design]*  
WindowsFormsApplication1  WindowsFormsApplication1.Form1  
1  using ...  
10  
11  namespace WindowsFormsApplication1  
12  {  
13  public partial class Form1 : Form  
14  {  
15  public Form1()  
16  {  
17  InitializeComponent();  
18  }  
19  
20  ←  
21  }  
22  }  
23  }  
24  }
```

Place form level variable declarations here, before any methods.

Form level variables are declared just like method level variables:

```
int myInt;  
DateTime myDate;
```

# Arrays

Visual C# has powerful facilities for handling arrays, which provide a way to store a large number of variables under the same name. Each variable, called an element, in an array must have the same data type, and they are distinguished from each other by an array index. In these notes, we work with one-dimensional arrays, although multi-dimensional arrays are possible.

Arrays are declared in a manner similar to that used for regular variables. For example, to declare an integer array named '**item**', with dimension **9**, we use:

```
int[] item = new int[9];
```

The index on an array variable begins at 0 and ends at the dimensioned value minus one. Hence, the **item** array in the above examples has **nine** elements, ranging from **item[0]** to **item[8]**. You use array variables just like any other variable - just remember to include its name and its index.

It is also possible to have arrays of controls. For example, to have 20 button types available use:

```
Button[] myButtons = new Button[20];
```

The utility of such a declaration will become apparent in later classes.

# Constants

You can also define constants for use in Visual C#. The format for defining an **int** type constant named **numberOfUses** with a value **200** is:

```
const int numberOfUses = 200;
```

The scope of user-defined constants is established the same way a variables' scope is. That is, if defined within a method, they are local to the method. If defined in the top region of a form's code window, they are global to the form.

If you attempt to change the value of a defined constant, your program will stop with an error message.

# Variable Initialization

By default, any declared numeric variables are initialized at zero. String variables are initialized at an empty string. If desired, Visual C# lets you initialize variables at the same time you declare them. Just insure that the type of initial value matches the variable type (i.e. don't assign a string value to an integer variable).

Examples of variable initialization:

```
int myInt = 23;
string myString = "Visual C#";
double myDouble = 7.28474746464;
bool isLeap = false;
```

You can even initialize arrays with this technique. With this technique, you can (optionally) delete the explicit dimension and let Visual C# figure it out by counting the number of elements used to initialize the array. An example is:

```
int[] item = new int[] {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

Visual C# will know this array has 10 elements (a dimension of 9).

# Intellisense Feature

Working within the code window is easy. You will see that typing code is just like using any word processor. The usual navigation and editing features are all there.

One feature that you will become comfortable with and amazed with is called **Intellisense**. As you type code, the Intellisense feature will, at times, provide assistance in completing lines of code. For example, once you type a control name and a dot (.), a drop-down list of possible properties and methods will appear. When we use methods, suggested values for arguments will be provided. Syntax errors will be identified. And, potential errors with running an application will be pointed out.

Intellisense is a very useful part of Visual C#. You should become acquainted with its use and how to select suggested values. We tell you about now so you won't be surprised when little boxes start popping up as you type code.

# Visual C# Statements and Expressions

The simplest (and most common) statement in C# is the **assignment** statement. It consists of a variable name, followed by the assignment operator (=), followed by some sort of **expression**, followed by a semicolon (;). The expression on the right hand side is evaluated, then the variable on the left hand side of the assignment operator is **replaced** by that value of the expression.

## Examples:

```
startTime = now;  
explorer = "Captain Spaulding";  
bitCount = byteCount * 8;  
energy = mass * lightSpeed * lightSpeed;  
netWorth = assets - liabilities;
```

The assignment statement stores information.

Statements normally take up a single line. Since C# ignores white space, statements can be **stacked** using a semicolon (;) to separate them. Example:

```
startTime = now; endTime = startTime + 10;
```

The above code is the same as if the second statement followed the first statement. The only place we tend to use stacking is for quick initialization of like variables.

If a statement is very long, it may be continued to the next line without any kind of continuation character. Again, this is because C# ignores white space. It keeps processing a statement until it finally sees a semicolon. Example:

```
months = Math.Log(final * intRate / deposit + 1) / Math.Log(1 + intRate);
```

This statement, though on two lines, will be recognized as a single line. We usually write each statement on a single line. Be aware that long lines of code in

the notes many times wrap around to the next line (due to page margins).

Comment statements begin with the two slashes (//). For example:

```
// This is a comment  
x = 2 * y; // another way to write a comment
```

You, as a programmer, should decide how much to comment your code. Consider such factors as reuse, your audience, and the legacy of your code. In our notes and examples, we try to insert comment statements when necessary to explain some detail. You can also have a multiple line comment. Begin the comment with / \* and end it with \*/. Example:

```
/*  
This is a very long  
comment over  
a few lines  
*/
```

# Type Casting

In each assignment statement, it is important that the type of data on both sides of the operator (=) is the same. That is, if the variable on the left side of the operator is an **int**, the result of the expression on the right side should be **int**.

C# (by default) will try to do any conversions for you. When it can't, an error message will be printed. In those cases, you need to explicitly **cast** the result. This means convert the right side to the same side as the left side. Assuming the desired type is **type**, the casting statement is:

**leftSide = (type) rightSide;**

You can cast from any basic type (decimal and integer numbers) to any other basic type. Be careful when casting from higher precision numbers to lower precision numbers. Problems arise when you are outside the range of numbers.

There are times we need to convert one data type to another. Visual C# offers a wealth of functions that perform these conversions. Some of these functions are:

<b>Convert.ToBoolean(Expression)</b>	Converts a numerical <i>Expression</i> to a <b>bool</b> data type (if <i>Expression</i> is nonzero, the result is true, otherwise the result is false)
<b>Convert.ToChar(Expression)</b>	Converts any valid single character string to a <b>char</b> data type
<b>Convert.ToDateTime(Expression)</b>	Converts any valid date or time <i>Expression</i> to a <b>DateTime</b> data type
<b>Convert.ToSingle(Expression)</b>	Converts an <i>Expression</i> to a <b>float</b> data type
<b>Convert.ToDouble(Expression)</b>	Converts an <i>Expression</i> to a <b>double</b> data type
<b>Convert.ToInt16(Expression)</b>	Converts an <i>Expression</i> to a <b>short</b> data type (any fractional part is rounded)

<b>Convert.ToInt32(Expression)</b>	Converts an <i>Expression</i> to an <b>int</b> data type (any fractional part is rounded)
<b>Convert.ToInt64(Expression)</b>	Converts an <i>Expression</i> to a <b>long</b> data type (any fractional part is rounded)

We will use many of these conversion functions as we write code for our applications and examples. The Intellisense feature of the code window will usually direct us to the function we need.

# Visual C# Arithmetic Operators

Operators modify values of variables. The simplest **operators** carry out **arithmetic** operations. There are five **arithmetic operators** in C#.

**Addition** is done using the plus (+) sign and **subtraction** is done using the minus (-) sign. Simple examples are:

Operation	Example	Result
Addition	$7 + 2$	9
Addition	$3.4 + 8.1$	11.5
Subtraction	$6 - 4$	2
Subtraction	$11.1 - 7.6$	3.5

**Multiplication** is done using the asterisk (\*) and **division** is done using the slash (/). Simple examples are:

Operation	Example	Result
Multiplication	$8 * 4$	32
Multiplication	$2.3 * 12.2$	28.06
Division	$12 / 2$	6
Division	$45.26 / 6.2$	7.3

The last operator is the **remainder** operator represented by a percent symbol (%). This operator divides the whole number on its left side by the whole number on its right side, ignores the main part of the answer, and just gives you the remainder.

It may not be obvious now, but the remainder operator is used a lot in computer programming. Examples are:

Operation	Example	Division Result	Operation Result
Remainder	$7 \% 4$	1 Remainder 3	3
Remainder	$14 \% 3$	4 Remainder 2	2
Remainder	$25 \% 5$	5 Remainder 0	0

The mathematical operators have the following **precedence** indicating the order they are evaluated without specific groupings:

1. Multiplication (\*) and division (/)
2. Remainder (%)
3. Addition (+) and subtraction (-)

If multiplications and divisions or additions and subtractions are in the same expression, they are performed in left-to-right order. **Parentheses** around expressions are used to force some desired precedence.

# Comparison and Logical Operators

There are six **comparison** operators in Visual C# used to compare the value of two expressions (the expressions must be of the same data type). These are the basis for making decisions:

Operator	Comparison
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to

It should be obvious that the result of a comparison operation is a **bool** value (**true** or **false**). **Examples:**

a = 9.6, b = 8.1, a > b returns true  
a = 14, b = 14, a < b returns false  
a = 14, b = 14, a >= b returns true  
a = 7, b = 11, a <= b returns true  
a = 7, b = 7, a == b returns true  
a = 7, b = 7, a != b returns false

Logical operators operate on **bool** data types, providing a Boolean result. They are also used in decision making. We will use three **logical** operators

Operator	Operation
!	Logical Not
&&	Logical And
	Logical Or

The Not (!) operator simply negates a Boolean value. It is very useful for ‘toggling’ Boolean variables. Examples:

If  $a = \text{true}$ , then  $!a = \text{false}$

If  $a = \text{false}$ , then  $!a = \text{true}$

The And (**&&**) operator checks to see if two different bool data types are both true. If both are true, the operator returns a true. Otherwise, it returns a false value. Examples:

$a = \text{true}, b = \text{true}$ , then  $a \&\& b = \text{true}$

$a = \text{true}, b = \text{false}$ , then  $a \&\& b = \text{false}$

$a = \text{false}, b = \text{true}$ , then  $a \&\& b = \text{false}$

$a = \text{false}, b = \text{false}$ , then  $a \&\& b = \text{false}$

The Or (**||**) operator (typed as two “pipe” symbols) checks to see if either of two bool data types is true. If either is true, the operator returns a true. Otherwise, it returns a false value. Examples:

$a = \text{true}, b = \text{true}$ , then  $a || b = \text{true}$

$a = \text{true}, b = \text{false}$ , then  $a || b = \text{true}$

$a = \text{false}, b = \text{true}$ , then  $a || b = \text{true}$

$a = \text{false}, b = \text{false}$ , then  $a || b = \text{false}$

Logical operators follow arithmetic operators in precedence. Use of these operators will become obvious as we delve further into coding.

# Concatenation Operators

To **concatenate** two string data types (tie them together), use the + symbol, the string concatenation operator:

```
currentTime = "The current time is" + "9:30";  
textSample = "Hook this " + "to this";
```

Visual C# offers other concatenation operators that perform an operation on a variable and assign the resulting value back to the variable. Hence, the operation

```
a = a + 2;
```

Can be written using the addition concatenation operator (+=) as:

```
a += 2;
```

This says a is incremented by 2.

Other concatenation operators and their symbols are:

Operator Name	Operator Symbol	Operator Task
String	<b>a += b;</b>	<b>a = a + b;</b>
Addition	<b>a += b;</b>	<b>a = a + b;</b>
Subtraction	<b>a -= b;</b>	<b>a = a - b;</b>
Multiplication	<b>a *= b;</b>	<b>a = a * b;</b>
Division	<b>a /= b;</b>	<b>a = a / b;</b>

We often increment and decrement by one. There are operators for this also. The increment operator:

**a++;**      is equivalent to:      **a = a + 1;**

Similarly, the decrement operator:

**a--;**      is equivalent to:      **a = a - 1;**

# String Functions

Visual C# offers a powerful set of functions to work with string type variables, which are very important in Visual C#. The **Text** property of the label control and the text box control are string types. You will find you are constantly converting string types to numeric data types to do some math and then converting back to strings to display the information.

To convert a string type to a numeric value, use one of the **Convert** methods seen in the **Type Casting** section. As an example, to convert the **Text** property of a text box control named **txtExample** to an **int** type, use:

```
Convert.ToInt32(txtExample.Text)
```

This result can then be used with the various mathematical operators. You need to be careful that the string you are converting to a number is a valid representation of a number. If it is not, an error will occur.

The **ToString** method can be used to convert a numeric variable to a string. This bit of code can be used to display the numeric variable **myNumber** in a text box control:

```
myNumber = 3.14159;  
txtExample.Text = myNumber.ToString();
```

Some quantities do not support a **ToString** method. In such cases, you can use the **Convert.ToString** method. This code:

```
myNumber = 3.14159;  
txtExample.Text = Convert.ToString(myNumber);
```

will also do a conversion.

If you need to control the number of decimal points (or other display features), you can use the **String.Format** method. This method has two arguments, the first is a **format string**, the second is the **number** to be formatted. The format

string is enclosed in two curly braces and consists of the variable number (0 in this case, since we only have one variable – the method can also be used to format multiple variables), a colon followed by the letter f (floating point) and the number of decimals. Sounds complicated doesn't it? An example will clear things up. To format a floating point number (**myNumber**) to 2 decimal points, convert it to a string and display it in a text box (**txtExample**), we use:

```
txtExample.Text = String.Format("{0:f2}", myNumber);
```

This statement says take the first (the **0** term in the format string) floating point number **myNumber**, round it to 2 decimals (the **f2** term following the colon in the format string) and convert it to a string. If you apply this code to our sample number, the text box control in this example will display **3.14**.

# Dates and Times

Working with dates and times in computer applications is a common task. The **DateTime** and **TimeSpan** data types are used to specify and determine dates, times and the difference between dates and times.

The **DateTime** data type is used to hold a date and a time. And, even though that's the case, you're usually only interested in the date or the time. To initialize a **DateTime** variable (**myDateTime**) to a specific date, use:

```
DateTime myDateTime = new DateTime(year, month, day);
```

where **year** is the desired year (**int** type), **month** the desired month (**int** type), and **day** the desired day (**int** type). The month 'numbers' run from 1 (January) to 12 (December), not 0 to 11. As an example, if you use:

```
DateTime myDateTime = new DateTime(1950, 7, 19);
```

then, display the result (after converting it to a string) in some control using:

```
myDateTime.ToString()
```

you would get:

**7/19/1950 12:00:00 AM**

This is my birthday, by the way. The time is set to a default value since only a date was specified.

Other **DateTime** formats are available. You will see some of them in further examples, problems and exercises in this course, or consult the on-line help facilities of Visual C#. Some examples using **myDateTime** are:

```
myDateTime.ToString("dddd, MMMM dd, yyyy"); // returns Wednesday, July 19, 1950  
myDateTime.ToString("dd/MM/yyyy"); // returns 19/07/1950
```

```
myDateTime.ToString("T"); // returns 12:00:00 AM
myDateTime.ToString("t"); // returns 12:00 AM
```

Individual parts of a **DateTime** object can be retrieved. Examples include:

```
myDateTime.Month // returns 7
myDateTime.Day // returns 19
myDateTime.DayOfWeek // returns DayOfWeek.Wednesday
```

Notice the **DayOfWeek** property yields a **DayOfWeek** object.

Visual C# also allows you to retrieve the current date and time using the **DateTime** data type. To place the current date in a variable use the **Today** property:

```
DateTime myToday = DateTime.Today;
```

The variable **myToday** will hold today's date with the default time. To place the current date and time in a variable, use the **Now** property:

```
DateTime myToday = DateTime.Now;
```

Doing that at this moment, I get:

```
myToday.ToString() // returns 2/2/2005 12:51:55 PM
```

Many times, you want to know the difference between two dates and/or times, that is, determine some time span. The Visual C# **TimeSpan** data type does just that – it allows you to find the difference between any two **DateTime** variables. The syntax is simple. To compute the difference between two **DateTime** variables (**dateTime1** and **dateTime2**), use:

```
TimeSpan diff = dateTime1 - dateTime2;
```

The computed difference will be in the format:

**days.hours:minutes:seconds**

The **Days**, **Hours**, **Minutes** or **Seconds** property of the **diff** variable will provide each component of the time span.

As an example, let's use today's date (**myToday**) and the example date (**myDateTime**, my birthday) to see how long I've been alive. First, compute the difference between the two dates:

```
DateTime myToday = DateTime.Today;  
DateTime myDateTime = new DateTime(1950, 7, 19);  
TimeSpan diff = myToday - myDateTime;
```

Then, if we examine:

```
diff.ToString()
```

the result is:

**19922.15:39:43.9750720**

The tells me I've been alive 19922 days, 15 hours, 39 minutes, and 43.975 seconds. Looks like I should be getting ready to celebrate my 20,000 days birthday!!

We have only introduced the **DateTime** and **TimeSpan** data types. You will find them very useful as you progress in your programming studies. Do some research on your own to determine how best to use dates and times in applications you build.

# Random Number Object

In writing games and learning software, we use a random number generator to introduce unpredictability. This insures different results each time you try a program. Visual C# has a few ways to generate random numbers. We will use one of them – a random generator of integers. The generator uses the Visual C# **Random** object.

To use the Random object (named **MyRandom** here), it is first declared and created using:

**Random myRandom = new Random();**

This statement is placed with the variable declaration statements.

Once created, when you need a random integer value, use the **Next** method of this Random object:

**myRandom.Next(Limit)**

This statement generates a random integer value that is greater than or equal to 0 and less than **Limit**. Note it is less than limit, not equal to. For example, the method:

**myRandom.Next(5)**

will generate random integers from 0 to 4. The possible values will be 0, 1, 2, 3 and 4.

As other examples, to roll a six-sided die, the number of spots would be computed using:

**numberSpots = myRandom.Next(6) + 1;**

To randomly choose a number between 100 and 200, use:

```
number = myRandom.Next(101) + 100;
```

# Math Functions

A last set of functions we need are mathematical functions (yes, programming involves math!) Visual C# provides a set of functions that perform tasks such as square roots, trigonometric relationships, and exponential functions.

Each of the Visual C# math functions comes from the **Math** class of the Visual C# framework (an object-oriented programming concept). All this means is that each function name must be preceded by **Math.** (say Math-dot) to work properly. The functions and the returned values are:

Math Function	Value Returned
Math.Abs	Returns the absolute value of a specified number
Math.Acos	Returns a double value containing the angle whose cosine is the specified number
Math.Asin	Returns a double value containing the angle whose sine is the specified number
Math.Atan	Returns a double value containing the angle whose tangent is the specified number
Math.Cos	Returns a double value containing the cosine of the specified angle
Math.E	A constant, the natural logarithm base
Math.Exp	Returns a double value containing e (the base of natural logarithms) raised to the specified power
Math.Log	Returns a double value containing the natural logarithm of a specified number
Math.Log10	Returns a double value containing the base 10 logarithm of a specified number
Math.Max	Returns the larger of two numbers
Math.Min	Returns the smaller of two numbers
Math.PI	A constant that specifies the ratio of the circumference of a circle to its diameter
Math.Pow	Used to raise a number to a specified power – an exponentiation operator

Math.Round	Returns the number nearest the specified number of decimal points
Math.Sign	Returns an Integer value indicating the sign of a number
Math.Sin	Returns a Double value containing the sine of the specified angle
Math.Sqrt	Returns a double value specifying the square root of a number
Math.Tan	Returns a double value containing the tangent of an angle

### **Examples:**

Math.Abs(-5.4) returns the absolute value of -5.4 (returns 5.4)

Math.Cos(2.3) returns the cosine of an angle of 2.3 radians

Math.Max(7, 10) returns the larger of the two numbers (returns 10)

Math.Sign(-3) returns the sign on -3 (returns a -1)

Math.Sqrt(4.5) returns the square root of 4.5

# Visual C# Decisions - if Statements

The concept of an **if** statement for making a decision is very simple. We check to see if a particular **Boolean** condition is true. If so, we take a certain action. If not, we do something else. **if** statements are also called **branching** statements. **Branching** statements are used to cause certain actions within a program if a certain condition is met.

The simplest branching statement is this form of the **if** structure:

**if (condition)**

[process this line of code]

In this statement, if **condition** is **true**, then the single statement following the statement will be executed. If condition is false, nothing is done and program execution continues following this line of code.

**Example:**

**if (balance – check < 0)**

**trouble = true;**

Here, if and only if balance - check is less than zero, the **trouble** variable is set to **true**. Notice this form of the **if** statement has no curly braces.

If there is more than one line of code to process following an **if** check, we use another form for the Visual C# **if** statement:

**if (condition)**

**{**

[process this code]

**}**

Here, if **condition** is **true**, the code bounded by the two braces is executed. If condition is false, nothing happens and code execution continues after the

closing right brace.

**Example:**

```
if (balance - check < 0)
{
    trouble = true;
    SendLettertoAccount();
}
```

In this case, if balance - check is less than zero, two lines of information are processed: trouble is set to true and a method sending a letter to the account holder is executed. Notice the indentation of the code between the two braces. The Visual C# environment automatically supplies this indentation. It makes understanding (and debugging) your code much easier. You can adjust the amount of indentation the IDE uses if you like.

What if you want to do one thing if a condition is true and another if it is false? Use an **if/else** block:

```
if (condition)
{
    [process this code]
}
else {
    [process this code]
}
```

In this block, if condition is true, the code between the first two braces is executed. If condition is false, the code between the second set of braces is processed.

**Example:**

```
if (balance - check < 0)
{
```

```
trouble = true;
SendLettertoAccount();
}
else
{
    trouble = false;
}
```

Here, the same two lines are executed if you are overdrawn ( $\text{balance} - \text{check} < 0$ ), but if you are not overdrawn (**else**), the trouble flag is turned off.

Lastly, we can test multiple conditions by adding the **else if** statement:

```
if (condition1)
{
    [process this code]
}
else if (condition2)
{
    [process this code]
}
else if (condition3)
{
    [process this code]
}
else
{
    [process this code]
}
```

In this block, if condition1 is true, the code between the if and first else if line is executed. If condition1 is false, condition2 is checked. If condition2 is true, the indicated code is executed. If condition2 is not true, condition3 is checked. Each subsequent condition in the structure is checked until a true condition is found,

an else statement is reached or the last closing brace is reached.

### Example:

```
if (balance - check < 0)
{
    trouble = true;
    SendLettertoAccount();
}

else if (balance - check == 0)
{
    trouble = false;
    SendWarningLetter();
}

else
{
    trouble = false;
}
```

Now, one more condition is added. If your balance equals the check amount [**else if** (balance - check == 0)], you're still not in trouble, but a warning is mailed.

In using branching statements, make sure you consider all viable possibilities in the if/else if structure. Also, be aware that each if and else if in a block is tested sequentially. The first time an if test is met, the code block associated with that condition is executed and the if block is exited. If a later condition is also true, it will never be considered.

# Switch - Another Way to Branch

In addition to if/then/else type statements, the **switch** structure can be used when there are multiple selection possibilities. switch is used to make decisions based on the value of a single variable. The structure is:

```
switch (variable)
{
    case [variable has this value]:
        [process this code]
        break;
    case [variable has this value]:
        [process this code]
        break;
    case [variable has this value]:
        [process this code]
        break;
    default:
        [process this code]
        break;
}
```

The way this works is that the value of **variable** is examined. Each **case** statement is then sequentially examined until the value matches one of the specified cases. Once found, the corresponding code is executed. If no case match is found, the code in the default segment (if there) is executed. The **break** statements transfer program execution to the line following the closing right brace. These statements are optional, but will almost always be there. If a break is not executed, all code following the case processed will also be processed (until a break is seen or the end of the structure is reached). This is different behavior than **if** statements where only one ‘case’ could be executed.

As an example, say we've written this code using the **if** statement:

```
if (age == 5)
{
    category = "Kindergarten";
}
else if (age == 6)
{
    category = "First Grade";
}
else if (age == 7)
{
    category = "Second Grade";
}
else if (age == 8)
{
    category = "Third Grade";
}
else if (age == 9)
{
    category = "Fourth Grade";
}
else
{
    category = "Older Child";
}
```

This will work, but it is ugly code and difficult to maintain.

The corresponding code with **switch** is ‘cleaner’:

```
switch (age)
{
    case 5:
        category = "Kindergarten";
```

```
break;
case 6:
    category = "First Grade";
    break;
case 7:
    category = "Second Grade";
    break;
case 8:
    category = "Third Grade";
    break;
case 9:
    category = "Fourth Grade";
    break;
default:
    category = “Older Child”;
    break;
}
```

# Visual C# Looping

Many applications require repetition of certain code segments. For example, you may want to roll a die (simulated die of course) until it shows a six. Or, you might generate financial results until a certain sum of returns has been achieved. This idea of repeating code is called iteration or **looping**.

In Visual C#, looping is done with one of two formats. The first is the **while** loop:

```
while (condition)
{
    [process this code]
}
```

In this structure, the code block in braces is repeated ‘as long as’ the Boolean expression **condition** is true. Note a while loop structure will not execute even once if the while condition is false the first time through. If we do enter the loop, it is assumed at some point condition will become false to allow exiting. Notice there is no semicolon after the while statement.

This brings up a very important point – if you use a loop, make sure you can get out of the loop!! It is especially important in the event-driven environment of Visual C# event-driven applications. As long as your code is operating in some loop, no events can be processed. You can also exit a loop using the **break** statement. This will get you out of a loop and transfer program control to the statement following the loop’s closing brace. Of course, you need logic in a loop to decide when a break is appropriate.

You can also use a **continue** statement within a loop. When a continue is encountered, all further steps in the loop are skipped and program operation is transferred to the top of the loop.

## Example:

```
counter = 1;
```

```
while (counter <= 1000)
{
    counter += 1;
}
```

This loop repeats as long as (**while**) the variable counter is less than or equal to 1000.

**Another example:**

```
roll = 0;
counter = 0;
while (counter < 10)
{
    // Roll a simulated die
    roll += 1;
    if (myRandom.Next(6) + 1 == 6)
    {
        counter += 1;
    }
}
```

This loop repeats **while** the counter variable is less than 10. The counter variable is incremented each time a simulated die rolls a 6. The roll variable tells you how many rolls of the die were needed to reach 10 sixes.

The second looping structure in Visual C# is a **do/while** structure:

```
do
{
    [process this code]
}
while (condition);
```

This loop repeats ‘as long as’ the Boolean expression condition is true. The loop is always executed at least once. Somewhere in the loop, condition must be changed to false to allow exiting. Notice there is a semicolon after the while statement.

### **Examples:**

```
sum = 0;  
do  
{  
    sum += 3;  
}  
while (sum <= 50);
```

In this example, we increment a sum by 3 until that sum exceeds 50 (or **while** the sum is less than or equal to 50).

### **Another example:**

```
sum = 0;  
counter = 0;  
do  
{  
    // Roll a simulated die  
    sum += myRandom.Next(6) + 1;  
    counter += 1;  
}  
while (sum <= 30);
```

This loop rolls a simulated die **while** the sum of the rolls does not exceed 30. It also keeps track of the number of rolls (counter) needed to achieve this sum.

Again, make sure you can always get out of a loop! Infinite loops are never nice. Sometimes the only way out is rebooting your machine!

# Visual C# Counting

With **while** and **do/while** structures, we usually didn't know, ahead of time, how many times we execute a loop or iterate. If you know how many times you need to iterate on some code, you want to use Visual C# **counting**. Counting is useful for adding items to a list or perhaps summing a known number of values to find an average.

Visual C# counting is accomplished using the **for** loop:

```
for (initialization; expression; update)
{
    [process this code]
}
```

The **initialization** step is executed once and is used to initialize a counter variable. The **expression** step is executed before each repetition of the loop. If expression is true, the code is executed; if false, the loop is exited. The **update** step is executed after each loop iteration. It is used to update the counter variable.

## Example:

```
for (degrees = 0; degrees <= 360; degrees += 10)
{
    // convert to radians
    r = degrees * Math.PI / 180;
    a = Math.Sin(r);
    b = Math.Cos(r);
    c = Math.Tan(r);
}
```

In this example, we compute trigonometric functions for angles from 0 to 360 degrees in increments of 10 degrees. It is assumed that all variables have been

properly declared.

### **Another Example:**

```
for (countdown = 10; countdown <= 0; countdown--)
{
    timeTextField.setText(String.valueOf(countdown));
}
```

NASA called and asked us to format a text field control to count down from 10 to 0. The loop above accomplishes the task. Note the use of the **decrement** operator.

### **And, another Example:**

```
double[] myValues = new double[100];
sum = 0;
for (int i = 0; i < 100; i++)
{
    sum += myValues[i];
}
average = sum / 100;
```

This code finds the average value of 100 numbers stored in the array **myValues**. It first sums each of the values in a **for** loop. That sum is then divided by the number of terms (100) to yield the average. Note the use of the **increment** operator. Also, notice the counter is declared in the initialization step. This is a common declaration in a loop or block of code. Such **loop** or **block level variables** lose their values once the loop is completed.

You may exit a for loop early using a **break** statement. This will transfer program control to the statement following the closing brace. Use of a **continue** statement will skip all statements remaining in the loop and return program control to the **for** statement.

# Chapter Review

After completing this chapter, you should understand:

- How to declare and properly use variables
- Visual C# statements
- The assignment operator, mathematics operators, comparison and logic operators and concatenation operators
- The wide variety of built-in Visual C# methods, especially string methods, the random number generator, and mathematics functions
- The **if/else if** structure used for branching and decisions
- The **switch** decision structure
- How the **do** structure is used in conjunction with the **while** statements
- How the **for** loop is used for counting

We now have the foundation needed to begin building projects. We will begin our first project in Chapter 4, once we have a brief discussion of debugging projects.

3

## Debugging a Visual C# Project

# Review and Preview

As you begin building projects using **Visual C# Homework Projects**, you will undoubtedly encounter errors in your code. In this chapter, we look at handling errors in projects, using the Visual C# debugger. After this, we begin building projects (at last).

# Errors in Visual C# Projects

No matter how hard we try, **errors** do creep into our projects. These errors can be grouped into three categories:

1. **Syntax** errors
2. **Run-time** errors
3. **Logic** errors

**Syntax errors** occur when you mistype a command, leave out an expected phrase or argument, or omit needed punctuation. Visual C# and the Intellisense feature detects these errors as they occur and even provides help in correcting them. You cannot run a Visual C# program until all syntax errors have been corrected. The **Task** window in the Visual C# IDE lists all syntax errors and identifies the line they occur in.

**Run-time errors** are usually beyond your program's control. Examples include: when a variable takes on an unexpected value (divide by zero), when a drive door is left open, or when a file is not found. Visual C# lets us trap such errors and make attempts to correct them. Doing so precludes our program from unceremoniously stopping. User's do not like programs that stop unexpectedly! We will discuss handling run-time errors in later chapters.

**Logic errors** are the most difficult to find. With logic errors, the program will usually run, but will produce incorrect or unexpected results. The Visual C# debugger is an aid in detecting logic errors.

Some ways to minimize errors are:

- Design your application carefully. More design time means less debugging time.
- Use comments where applicable to help you remember what you were trying to do.
- Use consistent and meaningful naming conventions for your variables, controls, objects, and methods.

# Debugging Visual C# Projects

We now consider the search for, and elimination of, **logic errors**. These are errors that don't prevent an application from running, but cause incorrect or unexpected results. Logic errors are sometimes difficult to find; they may be very subtle. Visual C# provides an excellent set of **debugging** tools to aid in this search.

A typical logic error could involve an **If/End If** structure. Look at this example:

```
if (a > 5 && b < 4)
{
..'do this code
}
else if (a == 6)
{
    'do this code
}
```

In this example, if  $a = 6$  and  $b = 2$ , the **else if** statement (which you wanted executed if  $a = 6$ ) will never be seen. In this case, swap the two **if** clauses to get the desired behavior. Or, another possible source of a logic error:

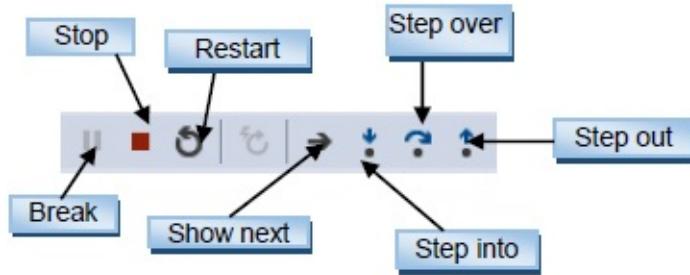
```
StreamReader inputFile = new StreamReader(dlgOpen.FileName);
inputFile.ReadLine(myLine);
```

In this little ‘snippet,’ a file the user selected using an file open dialog control (**dlgOpen**) is opened and the first variable read. It looks okay, but what if the user selected a file that really wasn’t meant to be used by your application. This would be a classic case of GIGO (garbage in – garbage out).

Debugging code is an art, not a science. There are no prescribed processes that you can follow to eliminate all logic errors in your program. The usual approach is to eliminate them as they are discovered.

What we'll do here is present the debugging tools available in the Visual C# environment and describe their use with an example. You, as the program designer, should select the debugging approach and tools you feel most comfortable with. The more you use the debugger, the more you will learn about it. Fortunately, the simpler tools will accomplish the tasks for most debugging applications.

The interface between your application and the debugging tools is via several different windows in the Visual C# IDE: the **Locals** window, the **Watch** window, and the **Output** window.. These windows are accessed using the **View** and/or **Debug** menu options. Debugging tasks can be accessed using this menu or they can be selected from the 'debug' region of the IDE toolbar (appears when program is in **break** mode):



We will examine buttons on this toolbar as we continue (you should already recognize the buttons to **Continue** (Start), **Break** and **Stop** the application).

All debugging using the debug windows is done when your application is in **debug** (or **break**) mode. Recall the application mode (while running) is always displayed in the title bar of Visual C#. You usually enter break mode by setting **breakpoints** (we'll look at this in a bit), pressing the **Break** button on the toolbar or when your program encounters an untrapped run-time error, one of the options provided is to go into break mode (click the button marked **Debug**).

Once in break mode, the debug windows and other tools can be used to:

- Determine values of variables
- Set breakpoints
- Set watch variables and expressions
- Manually control the application

- Determine which methods have been called
- Change the values of variables and properties

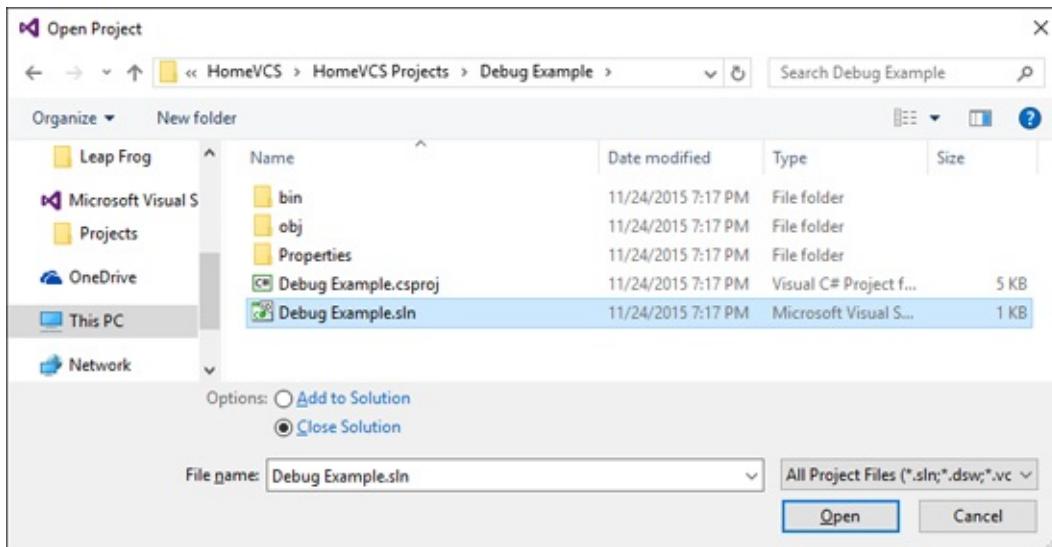
The best way to learn proper debugging is do an example. We'll open that example now, then learn how to use the Visual C# debugger.

# Opening a Visual C# Project

As mentioned, we will use an example project to illustrate use of the Visual C# debugger. So, let's review the steps in opening a previously saved Visual C# Project. We open a project by selecting **File** from the menu, then clicking **Open Project**.

Follow these steps:

- Click **File**, then **Open**, then **Project/Solution** in the menu. An **Open Project** window will appear:



- Find the folder named **HomeVCS**. This is the folder that holds the notes and projects for these notes. Open that folder.
- Find and open the folder named **HomeVCS Projects**. This folder holds all the projects for these notes.

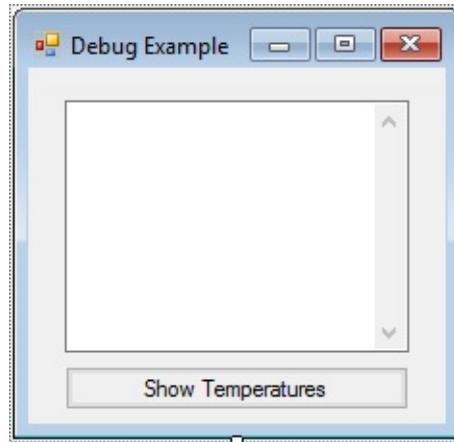
Remember how you got to this folder. In subsequent chapters, you will go to this folder to open projects you will need. Open the project folder named **Debug Example**.

In this project folder, among other things is a **Visual Studio Solution** file named **Debug Example** and a **Visual C# Project** file named **Debug Example**. Open

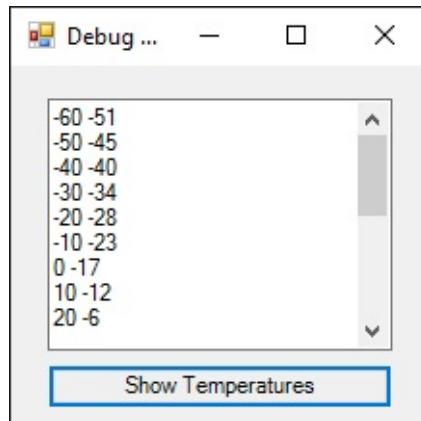
the **Debug Example** solution file (as shown in the example Open Project window). Since there is only one project in this solution, you could also open the project file and get the same results, but it is better to always open the solution file.

# Debugging Example

Once opened, note the project in **Debug Example** is simply a form with a button control (named **btnShow**) and a text box (named **txtTemps**):



Run the project (press <F5>) and click the button marked **Show Temperatures**. In the text box, you will see several lines, each line containing two numbers. The first number is a temperature in degrees Fahrenheit. The second number is the corresponding temperature in degrees C:



You should note in the above results that 0 degrees Fahrenheit is equivalent to negative 17 degrees Celsius. Let's look at the code that produced these results.

The **btnShow\_Click** event is a simple loop that generates temperature in degrees Fahrenheit (**tempF**) and uses a method (**FtoC**) to convert these values to degrees

Celsius (**tempC**):

```
private void btnShow_Click(object sender, EventArgs e)
{
    int tempF;
    int tempC;
    tempF = -60;
    do
    {
        tempC = FtoC(tempF);
        txtTemps.Text += tempF.ToString() + " " + tempC.ToString()+
"\r\n";
        tempF += 10;
    }
    while (tempF <= 120);
}
```

This code begins with a tempF value of -60 and computes the tempC value using the general method **FtoC**. The results are output to the text box control. tempF is then incremented by 10 and the loop is repeated. It continues looping while tempF is less than or equal to 120.

The temperature conversion method **FtoC** is computed using:

```
private int FtoC(int f)
{
    int c;
    c = (int) ((f - 32) * 5 / 9);
    numberCalls++;
    return (c);
}
```

This routine uses a form level variable (**numberCalls**) that keeps track of how many times this function is used. This variable is declared and initialized in the

“general declarations” area using:

```
int numberCalls = 0;
```

Make sure you can find all the code listed above in the code window for the example. Admittedly, this code doesn’t do much, but it makes a good example for looking at debugger use. So, get ready to try debugging.

# Using the Debugging Tools

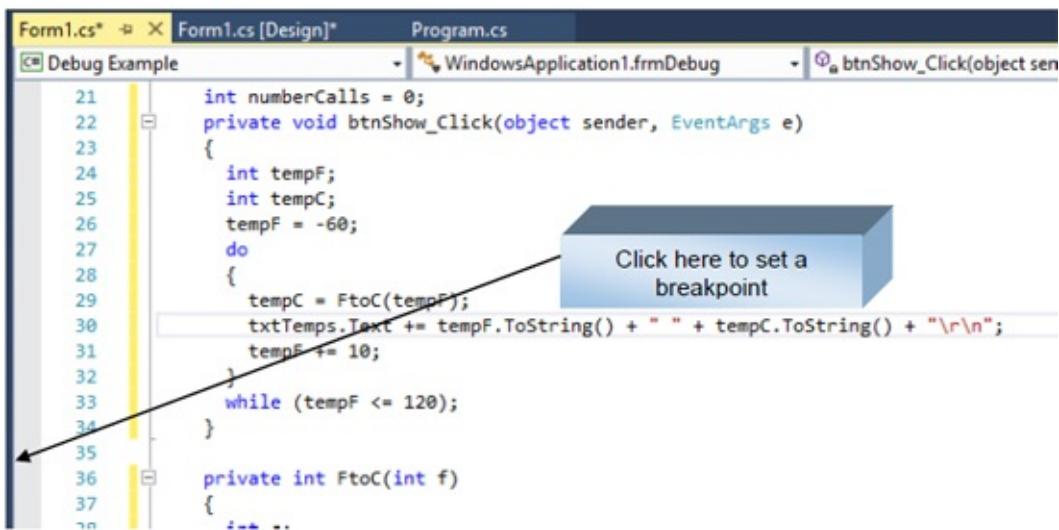
There are several **debugging tools** available for use in Visual C#. Access to these tools is provided via both menu options and buttons on the toolbar. Some of the tools we will examine are:

- **Breakpoints** which let us stop our application.
- **Locals** and **Watch windows** which let us examine variable values.
- **Call stack** which let us determine how we got to a certain point in code.
- **Step into, step over** and **step out** which provide manual execution of our code.

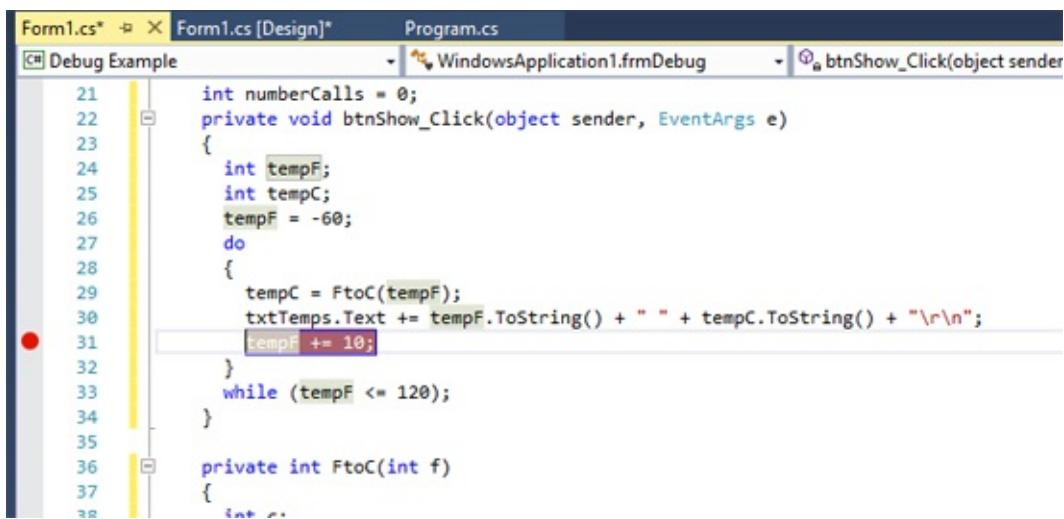
These tools work in conjunction with the various debugger windows.

# Breakpoints

To debug a program, we want to stop the application while it is running, examine variables and then continue running. This can be done with **breakpoints**. A breakpoint marks a line in code where you want to stop (temporarily) program execution, that is force the program into **break (debug)** mode. One way to set a breakpoint is to put the cursor in the line of code you want to break at and press <F9>. Or, a simpler way is to click next to the desired line of code in the vertical shaded bar at the left of the code window:



Once set, a large red dot marks the line along with shading:

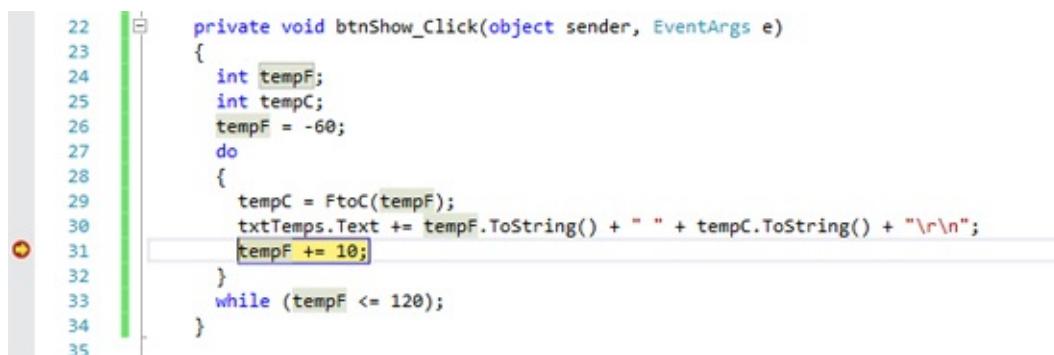


To remove a breakpoint, repeat the above process. Pressing <F9> or clicking the line at the left will ‘toggle’ the breakpoint. Breakpoints can be added/deleted at **design** time or in **break** mode.

When you run your application, Visual C# will stop when it reaches lines with breakpoints and allow you to check variables and expressions. To continue program operation after a breakpoint, press <F5>, click the **Continue** button on the toolbar, or choose **Continue** from the **Debug** menu.

### **Breakpoint Example:**

Set a breakpoint on the **tempF += 10** line in the **btnShow Click** event (as demonstrated above). Run the program and click the **Show Temperatures** button. The program will stop at the desired line (it will be highlighted). Hold the cursor over **tempF** two lines above this line and you should see:



```
22
23
24
25
26
27
28
29
30
31
32
33
34
35
```

```
private void btnShow_Click(object sender, EventArgs e)
{
    int tempF;
    int tempC;
    tempF = -60;
    do
    {
        tempC = FtoC(tempF);
        txtTemps.Text += tempF.ToString() + " " + tempC.ToString() + "\r\n";
        tempF += 10;
    }
    while (tempF <= 120);
}
```

A screenshot of a Visual Studio code editor showing a C# method named `btnShow_Click`. A red circular breakpoint icon is positioned on the left margin next to line 31. The line of code containing the breakpoint, `tempF += 10;`, is highlighted with a yellow background. A tooltip window is open over the variable `tempF` on line 30, displaying its current value as `-60`.

Notice a tooltip appears displaying the current value of this variable (**tempF = -60**). You can move the cursor onto any variable in this method to see its value. If you check values on lines prior to the line with the breakpoint, you will be given the current value. If you check values on lines at or after the breakpoint, you will get their last computed value.

Continue running the program (click the **Continue** button or press <F5>). The program will again stop at this line, but **tempF** will now be equal to **-50**. Check it. Continue running the program, examining **tempF** and **tempC**.

Try other breakpoints in the application if you have time.

# Viewing Variables in the Locals Window

In the breakpoint example, we used tooltips to examine variable values. We could see variable values, no matter what their scope. The **locals** window can also be used to view variables, but only those with method level scope. As execution switches from method to method, the contents of the local window change to reflect only the variables applicable to the current method. The locals window can be viewed using the dropdown button on the debug toolbar or select the **Debug** menu option, choose **Windows**, then **Locals**.

## Locals Window Example:

Make sure there is still a breakpoint on the **tempF += 10** line in the **btnShow Click** event. Also place a breakpoint on the **numberCalls++** line in the **FtoC** function. Run the program and click the **Show Temperatures** button. The program will stop at the desired line in the **FtoC** function. View the locals window and you should see:

Locals	
Name	Value
▶ <input checked="" type="checkbox"/> this	{WindowsApplication1.frmDebug, Text: Debug Example}
▶ <input checked="" type="checkbox"/> f	-60
▶ <input checked="" type="checkbox"/> c	-51

All the variables local to this method (**f** and **c**) are seen.

Continue running the application (click **Continue** button or press **<F5>**). The program will now stop at the marked line in the **btnShow Click** event. Viewing the locals window shows:

Locals	
Name	Value
▶ <input checked="" type="checkbox"/> this	{WindowsApplication1.frmDebug, Text: Debug Example}
▶ <input checked="" type="checkbox"/> sender	{Text = "Show Temperatures"}
▶ <input checked="" type="checkbox"/> e	(X = 110 Y = 15 Button = Left)
▶ <input checked="" type="checkbox"/> tempF	-60
▶ <input checked="" type="checkbox"/> tempC	-51

which are initial values for the local variables (**tempF** and **tempC**) in this

method. Continue running, watching the values change as the program move from one method to another. Stop the example whenever you want. Remove the breakpoint in the **FtoC** function.

# Viewing Variables in the Watch Window

**Watch** windows can also be used to examine variables. They can maintain values for both method level and form level variables. And, you can even change the values of variables in the watch window (be careful if you do this; strange things can happen). The watch window can only be accessed in **break** mode. At that time, you simply right-click on any variables you want to add to the watch window and select the **Add Watch** option. To view a watch window, select the **Debug** menu option, choose **Windows**, then **Watch**, then one of available watch windows.

## Watch Window Example:

Make sure there is still a breakpoint on the **tempF += 10** line in the **btnShow Click** event. Run the program and click the **Show Temperatures** button. The program will stop at the desired line. Open a watch window. Right-click on the **tempF** variable and select **Add Watch**. Scroll down into the **FtoC** function and do the same for the **numberCalls** variable. You should now see:

Watch 1	
Name	Value
tempF	-60
numberCalls	1

The current value of **tempF** (a method level variable) and **numberCalls** (a form level variable) are seen. They are both zero. Continue running the application (click **Continue** button or press **<F5>**), watching **tempF** and **numberCalls** change with each loop execution.

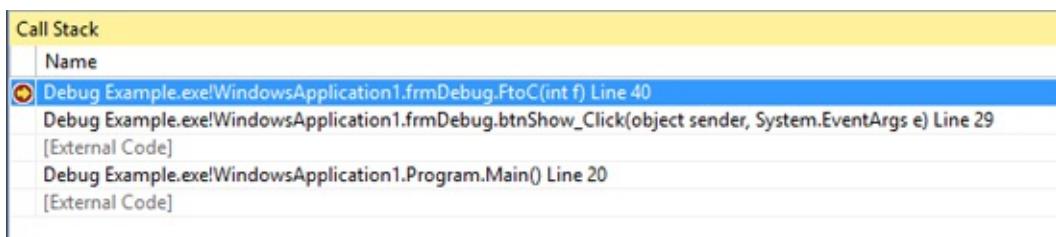
The two watch variables can be deleted by right-clicking the variable name in the watch window and selecting **Delete Watch**. Again, you can only do this in **break** mode.

# Call Stack Window

General methods can be called from many places in an application. That's the idea of using a general method! If an error occurs in such a method, it is helpful to know which method was calling it. While in break mode, the **Call Stack** window will provide will display all active methods, that is those that have not been exited. The call stack window provides a road map of how you got to the point you are in the code. The call stack window can be viewed using the dropdown button on the debug toolbar or select the **Debug** menu option, choose **Windows**, then **Call Stack**.

## Call Stack Window Example:

Set a breakpoint on the **numberCalls++** line in the general method (**FtoC**). Run the application. Click **Show Temperatures**. The program will break at the marked line. Open the call stack window and you will see:



The information of use to us is the top two lines. These lines tell us we are in the **FtoC** function and it was called by the **btnShow\_Click** method on a form named **frmDebug**. This would be very useful information if there were an error occurring in the function.

Stop the application and remove all breakpoints.

# Single Stepping (Step Into) an Application

A powerful feature of the Visual C# debugger is the ability to manually control execution of the code in your application. The **Step Into** option lets you execute your program one line at a time. It lets you watch how variables change (in the locals window) or how your form changes, one step at a time. Single stepping is especially useful to check that decision structures and loops you use in your code are executing correctly.

Once in break mode (at a breakpoint), you can use **Step Into** by pressing <F11>, choosing the **Step Into** option in the **Debug** menu, or by clicking the **Step Into** button on the toolbar:



Each time you click this button, the single highlighted line of code will be executed.

## Step Into Example:

Set a breakpoint at the `tempF = -60` line in the **btnShow Click** event. Run the application. Click **Show Temperatures**. The program will stop and the marked line will be highlighted in the code window:

```
21 int numberCalls = 0;
22 private void btnShow_Click(object sender, EventArgs e)
23 {
24     int tempF;
25     int tempC;
26     tempF = -60;
27     do
28     {
29         tempC = FtoC(tempF);
30         txtTemps.Text += tempF.ToString() + " " + tempC.ToString() + "\r\n";
31         tempF += 10;
32     }
33     while (tempF <= 120);
34 }
35
36 private int FtoC(int f)
37 {
38     int c;
```

Open the locals window and use the **Step Into** button to single step through the program. It's fun to see the program logic being followed. Notice how the contents of the locals window change as program control moves from the **btnShow Click** event to the general function **FtoC**. When you are in the **btnShow Click** method, values for **tempF** and **tempC** are listed. When in the function, values for **C** and **F** are given.

# Method Stepping (Step Over)

Did you notice in the example just studied that, after a while, it became annoying to have to single step through the function evaluation at every step of the do loop? While single stepping your program, if you come to a method call that you know operates properly, you can perform **method stepping**. This simply executes the entire method at once, treating as a single line of code, rather than one step at a time.

To move through a method in this manner, while in break mode, press <F10>, choose **Step Over** from the **Debug** menu, or press the **Step Over** button on the toolbar:



## Step Over Example:

Run the previous example. Single step through it a couple of times.

One time through, when you are at the line calling the **FtoC** method, press the **Step Over** button. Notice how the program did not single step through the method as it did previously.

# Method Exit (Step Out)

While stepping through your program, if you wish to complete the execution of a method (or function) you are in, without stepping through it line-by-line, choose the **Step Out** option. The method will be completed and you will be returned to the method accessing that function.

To perform this step out, press **<Shift>+<F11>**, choose **Step Out** from the **Debug** menu, or press the **Step Out** button on the toolbar



## Step Out Example:

Run the previous example. Single step through it a couple of times. Also, try stepping over the function.

At some point, while single stepping through the function, press the **Step Out** button. Notice how control is immediately returned to the calling method (**btnShow Click** event).

At some point, while in the **btnShow Click** event, press the **Step Out** button. The method will be completed and the application form will reappear with the completed text box displayed.

# Debugging Strategies

We've looked at each debugging tool briefly. Be aware this is a cursory introduction. Use the on-line help to delve into the details of each tool described.

Only through lots of use and practice can you become a proficient debugger. You'll get this practice, too, while building the projects in these notes. Every time your application encounters a run-time error, the dialog box describing the error will have a **Debug** button. Click that button to start using your new found debugging skills.

There are some common sense guidelines to follow when debugging. My first suggestion is: keep it **simple**. Many times, you only have one or two bad lines of code. And you, knowing your code best, can usually quickly narrow down the areas with bad lines. Don't set up some elaborate debugging method if you haven't tried a simple approach to find your error(s) first.

A tried and true approach to debugging can be called **Divide and Conquer**. If you're not sure where your error is, guess somewhere in the middle of your application code. Set a breakpoint there. If the error hasn't shown up by then, you know it's in the second half of your code. If it has shown up, it's in the first half. Repeat this division process until you've narrowed your search.

And, of course, the best debugging strategy is to be careful when you first design and write your application to minimize searching for errors later.

# Chapter Review

After completing this class, you should understand:

- The different types of errors you can encounter in a Visual C# project.
- How to use the various capabilities of the Visual C# debugger to find and eliminate logic errors.
- Differences between the locals and watch windows in the debugger.
- How to set breakpoints in the code window.
- How to view variables using tooltips, the locals window and the watch window.
- How to single step, step over and step out of methods while debugging.

# 4

## Dual-Mode Stopwatch Project

# Review and Preview

We've completed our review of the Visual C# IDE, the language used to program using Visual C# and techniques for debugging projects. We can finally start building some Homework Projects. For each project built, we provide step-by-step instructions in designing and building the form's graphic interface and detailed explanations of the code behind the projects.

The first project we build is a **Dual-Mode Stopwatch** that allows you to time tasks you may be doing. Controls used include the form, the label control, the button control and the timer. We also look at how to subtract **DateTime** data types to obtain elapsed times and how to use general methods.

# Project Design Considerations

Before building this first project by drawing the Visual C# interface, setting the control properties, and writing the C# code, let's look at some of the things that should be considered to make a useful project. A first consideration should be to determine what processes and methods you want your application to perform. What are the inputs and outputs? Develop a framework or flow chart of all your application's processes.

Decide what controls you need. Do the built-in Visual C# controls and methods meet your needs? Do you need to develop some controls or methods of your own? You can design and build your own controls using Visual C#, but that topic is beyond the scope of this course. The skills gained in this course, however, will be invaluable if you want to tackle such a task.

Design your user interface. What do you want your form to look like? Consider appearance and ease of use. Make the interface consistent with other Windows applications. Familiarity is good in program design.

Write your code. Make your code readable and traceable - future code modifiers (including yourself) will thank you. Consider developing reusable code - modules with utility outside your current development. This will save you time in future developments.

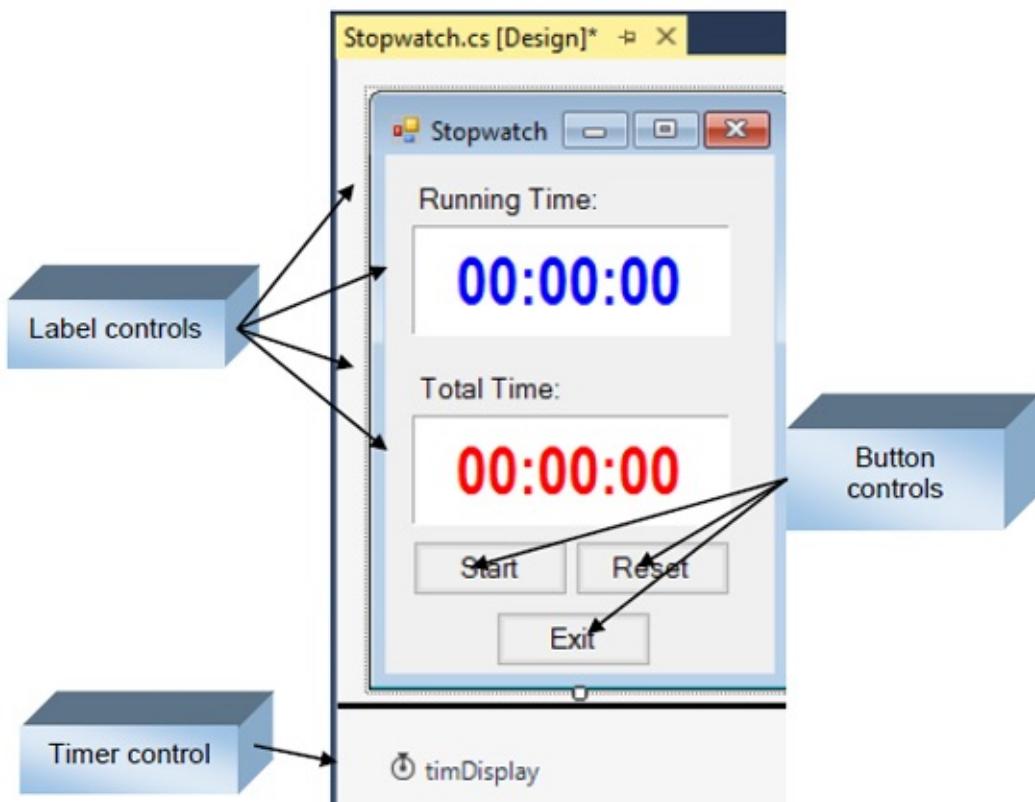
Make your code 'user-friendly.' Make operation of your application obvious to the user. Step the user through its use. Try to anticipate all possible ways a user can mess up in using your application. It's fairly easy to write an application that works properly when the user does everything correctly. It's difficult to write an application that can handle all the possible wrong things a user can do and still not bomb out.

Debug your code completely before giving it to others. There's nothing worse than having a user call you to point out flaws in your application. A good way to find all the bugs is to let several people try the code - a mini beta-testing program.

# Dual-Mode Stopwatch Project Preview

In this chapter, we will build a **dual-mode stopwatch**. The stopwatch can be started and stopped when desired. Two times are tracked: the time that elapses while the stopwatch is active (the running time) and the total time elapsed between first starting and finally stopping the stopwatch.

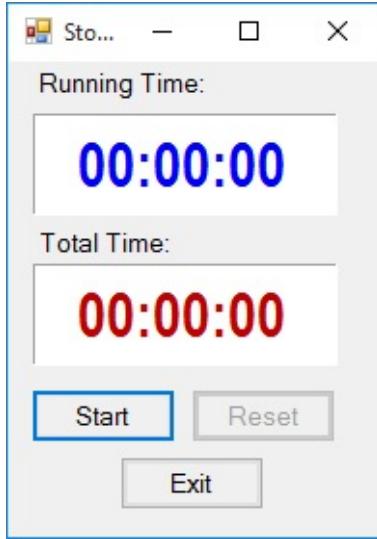
The finished project is saved as **Stopwatch** in the **HomeVCS\HomeVCS Projects** folder. Start Visual C# and open the finished project. Open the form (double-click **Stopwatch.cs** in Solution Explorer) and you will see:



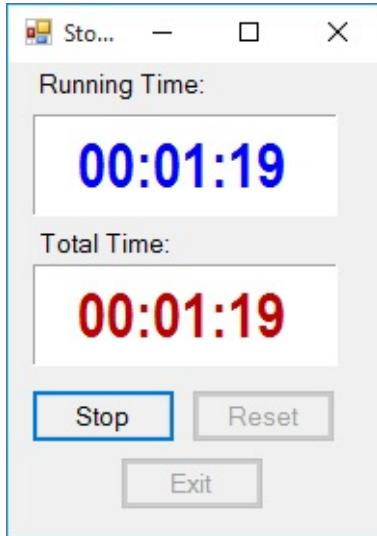
Two label controls are used for time displays (two additional labels provide titling information). Three button controls start, stop and reset the stopwatch and one stops the application. A timer control is used to periodically (every second) update the displayed times.

Run the project (press <F5>). The stopwatch will appear in its ‘initial’ state, with

the displayed times set at zero and the **Reset** button disabled:



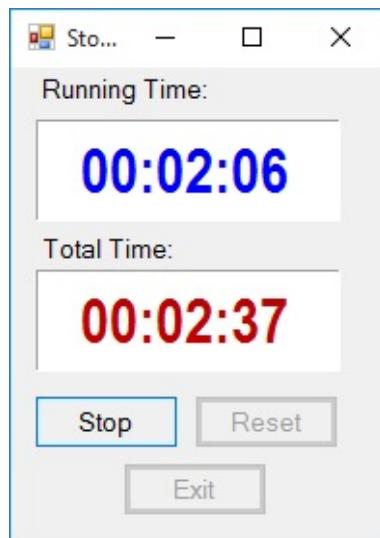
Click the **Start** button to start the stopwatch. Its caption will change (now reading **Stop**) and the **Exit** button will become disabled - the two displayed times will be the same (updating every second). We call this the ‘running’ state:



At some point, click **Stop**. When I did, the form appears as:



At this point ('stopped' state), all buttons become enabled and you have three options. You can click **Exit** to stop the project. You can click **Reset** to set both times back to zero and return the project to its initial state. Or, you can click the button now labeled **Restart** to restart the timer. When I do this, after a short wait, I get:



The stopwatch is running again, but the two displayed times are different. The **Total Time** is the amount of time elapsed since we first started the stopwatch. The **Running Time** is the total time less time when the stopwatch is in stopped mode. Based on these values, this stopwatch has spent 0:31 waiting around.

Continue starting, stopping, restarting and resetting the stopwatch to understand

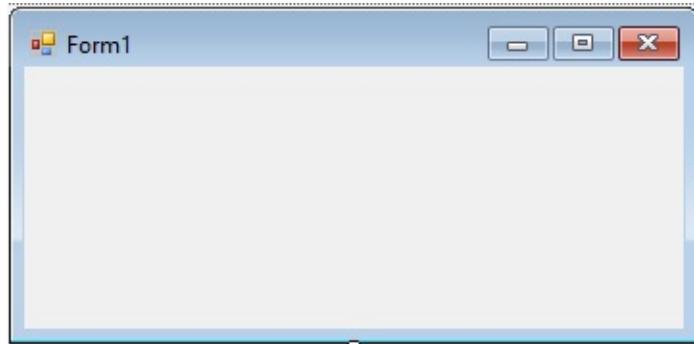
its operation. Click **Exit** when you're done to stop the project. Open the code window and skim over the code, if you like.

You will now build this project in stages. As you build Visual C# projects, we always recommend taking a slow, step-by-step process. It minimizes programming errors and helps build your confidence as things come together in a complete project. This is the approach we will take on all projects in these notes.

We address **form design**. We discuss the controls needed to build the form, establish initial control properties and discuss how to change the state of the controls. And, we address **code design**. We discuss how to do the necessary mathematics to determine the various displayed times. We also discuss use of general methods in Visual C# projects.

We see that the stopwatch project uses a form, several label controls, three button controls and a timer control. Since this is the first time we've used these controls in these notes, we will first review their use in Visual C# projects.

# Form Object



The **Form** is the object where the user interface is drawn. It is central to the development of Visual C# applications. The form is known as a **container** object, since it ‘holds’ other controls. One implication of this distinction is that controls placed on the form will share **BackColor**, **ForeColor** and **Font** properties. To change this, select the desired control (after it is placed on the form) and change the desired properties. Another feature of a container control is that when its **Enabled** property is False, all controls in the container are disabled. Likewise, when the container control **Visible** property is False, all controls in the container are not visible.

Here, we present some of the more widely used **Properties**, **Methods** and **Events** for the form. Recall **properties** described the appearance and value of a control, **methods** are actions you can impose on controls and **events** occur when something is done to the control (usually by a user). This is not an exhaustive list – consult on-line help for such a list. You may not recognize all of these terms now. They will be clearer as you progress in the course. The same is true for the remaining controls presented in this chapter.

## Form Properties:

<b>Name</b>	Gets or sets the name of the form (three letter prefix for form name is <b>frm</b> ).
<b>AcceptButton</b>	Gets or sets the button on the form that is clicked when the user presses the <Enter> key.
<b>BackColor</b>	Get or sets the form background color.

<b>CancelButton</b>	Gets or sets the button control that is clicked when the user presses the <Esc> key.
<b>ControlBox</b>	Gets or sets a value indicating whether a control box is displayed in the caption bar of the form.
<b>Enabled</b>	If False, all controls on form are disabled.
<b>Font</b>	Gets or sets font name, style, size.
<b>ForeColor</b>	Gets or sets color of text or graphics.
<b>FormBorderStyle</b>	Sets the form border to be fixed or sizeable.
<b>Height</b>	Height of form in pixels.
<b>Help</b>	Gets or sets a value indicating whether a Help button should be displayed in the caption box of the form.
<b>Icon</b>	Gets or sets the icon for the form.
<b>Left</b>	Distance from left of screen to left edge of form, in pixels.
<b>MaximizeButton</b>	Gets or sets a value indicating whether the maximize button is displayed in the caption bar of the form.
<b>MinimizeButton</b>	Gets or sets a value indicating whether the minimize button is displayed in the caption bar of the form.
<b>StartPosition</b>	Gets or sets the starting position of the form when the application is running.
<b>Text</b>	Gets or sets the form window title.
<b>Top</b>	Distance from top of screen to top edge of form, in pixels.
<b>Width</b>	Width of form in pixels.

### Form Methods:

<b>Close</b>	Closes the form.
<b>Focus</b>	Sets focus to the form.
<b>Hide</b>	Hides the form.
<b>Refresh</b>	Forces the form to immediately repaint itself.
<b>Show</b>	Makes the form display by setting the Visible

property to True.

The normal syntax for invoking a method is to type the control name, a dot, then the method name. For form methods, the name to use is **this**. This is a Visual C# keyword used to refer to a form. Hence, to close a form, use:

```
this.Close();
```

### Form Events:

<b>Activated</b>	Occurs when the form is activated in code or by the user.
<b>Click</b>	Occurs when the form is clicked by the user.
<b>Closing</b>	Occurs when the form is closing.
<b>DoubleClick</b>	Occurs when the form is double clicked.
<b>Load</b>	Occurs before a form is displayed for the first time (the Form object <b>default</b> event).
<b>Paint</b>	Occurs when the form is redrawn.

Typical use of **Form** object (for each control in this, and following chapters, we will provide information for how that control is typically used):

- Set the **Name** and **Text** properties
- Set the **StartPosition** property (in this course, this property will almost always be set to **CenterScreen**)
- Set the **FormBorderStyle** to some value. In this course, we will mostly use **FixedSingle** forms. You can have resizable forms in Visual C# (and there are useful properties that help with this task), but we will not use resizable forms in this course.
- Write any needed initialization code in the form's **Load** event. To access this event, double-click the form or select it using the properties window.

# Button Control

## In Toolbox:



## On Form (Default Properties):



The **Button** control probably the most widely used Visual C# control. It is used to begin, interrupt, or end a particular process. Here, we provide some of the more widely used properties, methods and events for the button control.

### Button Properties:

<b>Name</b>	Gets or sets the name of the button (three letter prefix for button name is <b>btn</b> ).
<b>BackColor</b>	Get or sets the button background color.
<b>Enabled</b>	If False, button is visible, but cannot accept clicks.
<b>Font</b>	Gets or sets font name, style, size.
<b>ForeColor</b>	Gets or sets color of text or graphics.
<b>Image</b>	Gets or sets the image that is displayed on a button control.
<b>Text</b>	Gets or sets string displayed on button.
<b> TextAlign</b>	Gets or sets the alignment of the text on the button control.

### Button Methods:

<b>Focus</b>	Sets focus to the button.
<b>PerformClick</b>	Generates a Click event for a button.

### Button Events:

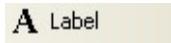
<b>Click</b>	Event triggered when button is selected either by clicking on it or by pressing the access key (the Button control <b>default</b> event).
--------------	---

Typical use of **Button** control:

- Set the **Name** and **Text** property.
- Write code in the button's **Click** event.
- You may also want to change the **Font**, **Backcolor** and **Forecolor** properties.

# Label Control

## In Toolbox:



## On Form (Default Properties):



A **Label** control is used to display text that a user can't edit directly. The text of a label box can be changed at run-time in response to events.

### Label Properties:

<b>Name</b>	Gets or sets the name of the label (three letter prefix for label name is <b>lbl</b> ).
<b>AutoSize</b>	Gets or sets a value indicating whether the label is automatically resized to display its entire contents.
<b>BackColor</b>	Get or sets the label background color.
<b>BorderStyle</b>	Gets or sets the border style for the label.
<b>Font</b>	Gets or sets font name, style, size.
<b>ForeColor</b>	Gets or sets color of text or graphics.
<b>Text</b>	Gets or sets string displayed on label.
<b> TextAlign</b>	Gets or sets the alignment of text in the label.

### Label Methods:

<b>Refresh</b>	Forces an update of the label control contents.
----------------	---

### Label Events:

<b>Click</b>	Event triggered when user clicks on a label (the Label control's <b>default</b> event).
--------------	---

<b>DblClick</b>	Event triggered when user double-clicks on a label.
-----------------	---

Typical use of **Label** control for static, unchanging display:

- Set the **Name** (though not really necessary for static display) and **Text** property.
- You may also want to change the **Font**, **Backcolor** and **Forecolor** properties.

Typical use of **Label** control for changing display:

- Set the **Name** property. Initialize **Text** to desired string.
- Assign **Text** property (string type) in code where needed.
- You may also want to change the **Font**, **Backcolor** and **Forecolor** properties.

# Timer Control

**In Toolbox:**



**Below Form (Default Properties):**



The Visual C# **Timer** control can generate events without any input from the user. Timer controls work in your project's background, generating events at time intervals you specify. This event generation feature comes in handy when screen displays need to be updated at regular intervals. Other control events can be detected while the timer control processes events in the background. This multi-tasking allows more than one thing to be happening in your application.

**Timer Properties:**

<b>Name</b>	Gets or sets the name of the timer control (three letter prefix is <b>tim</b> ).
<b>Enabled</b>	Used to turn the timer on and off. When True, timer control continues to operate until the Enabled property is set to False.
<b>Interval</b>	Number of milliseconds (there are 1000 milliseconds in one second) between each invocation of the timer control's <b>Tick</b> event.

**Timer Events:**

<b>Tick</b>	Event method invoked every <b>Interval</b> milliseconds while timer control's <b>Enabled</b> property is <b>True</b> .
-------------	--

To use the **Timer** control, we add it to our application the same as any control. There is no user interface, so it will appear in the tray area below the form in the

design window. You write code in the timer control's **Tick** event. This is the code you want to repeat every **Interval** milliseconds.

The timer control's **Enabled** property is **False** at design time. You 'turn on' a timer in code by changing this property to **True**. Usually, you will have some control that toggles the timer control's **Enabled** property. That is, you might have a button that turns a timer on and off. The logical **Not** operator is very useful for this toggling operation. If you have a timer control named **timExample**, this line of code will turn it on (set Enabled to True) if it is off (Enabled is False). It will do the reverse if the timer is already on:

```
timExample.Enabled = !timExample.Enabled;
```

It is best to start and end applications with the timer controls off (**Enabled** set to **False**).

Applications can (and many times do) have multiple timer controls. You need separate timer controls if you have events that occur with different regularity (different **Interval** values). Timer controls are used for two primary purposes. First, you use timer controls to periodically repeat some code segment. This is like our stopwatch project. Second, you can use a timer control to implement some 'wait time' established by the **Interval** property. In this case, you simply start the timer and when the Interval is reached, have the **Tick** event turn its corresponding timer off.

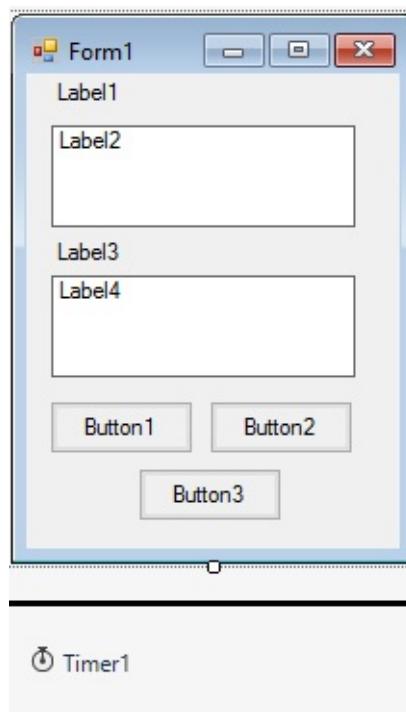
Typical use of **Timer** control:

- Set the **Name** property and **Interval** property.
- Write code in **Tick** event.
- At some point in your application, set **Enabled** to **True** to start timer. Also, have capability to reset **Enabled** to **False**, when desired.

# Stopwatch Form Design

Having reviewed the necessary controls, we can begin building the stopwatch project. Before starting, make sure you have established a folder on your computer for building Visual C# projects. Always save your projects in this folder. Do not save them in the folder used in these notes (**HomeVCS\HomeVCS Projects**). Leave this folder intact so you can always reference the finished projects, if needed.

Let's build the form. Start a new project in Visual C#. Place four label controls (set **AutoSize** to **False** for two of them to allow resizing) and three button controls on your form. Add a timer control to the project. The form should look similar to this:



The label controls are used to display times and their title information. Note **label2** and **label4** have been resized to display the stopwatch times (the **FormBorder** property is temporarily set to **FixedSingle** to show the label size). The buttons (one to start/stop/restart, one to reset and one to exit the project) are used to control operation of the stopwatch. The timer control is used to update the displayed times every second.

Set the control properties using the properties window:

**Form1** Form:

<b>Property Name</b>	<b>Property Value</b>
Name	frmStopwatch
Text	Stopwatch
FormBorderStyle	Fixed Single
StartPosition	CenterScreen

**label1** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Running Time:
Font Size	10

**label2** Label:

<b>Property Name</b>	<b>Property Value</b>
Name	lblRunning
Text	00:00:00
BackColor	White
ForeColor	Blue
BorderStyle	Fixed3D
Font	Arial Narrow
Font Size	24
Font Style	Bold
TextAlign	MiddleCenter

**label3** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Total Time:
Font Size	10

**label4** Label:

<b>Property Name</b>	<b>Property Value</b>
Name	lblTotal
Text	00:00:00
BackColor	White
ForeColor	Red
BorderStyle	Fixed3D
Font	Arial Narrow
Font Size	24
Font Style	Bold
TextAlign	MiddleCenter

**button1** Button:

<b>Property Name</b>	<b>Property Value</b>
Name	btnStartStop
Text	Start
Font Size	10

**button2** Button:

<b>Property Name</b>	<b>Property Value</b>
Name	btnReset
Text	Reset
Enabled	False
Font Size	10

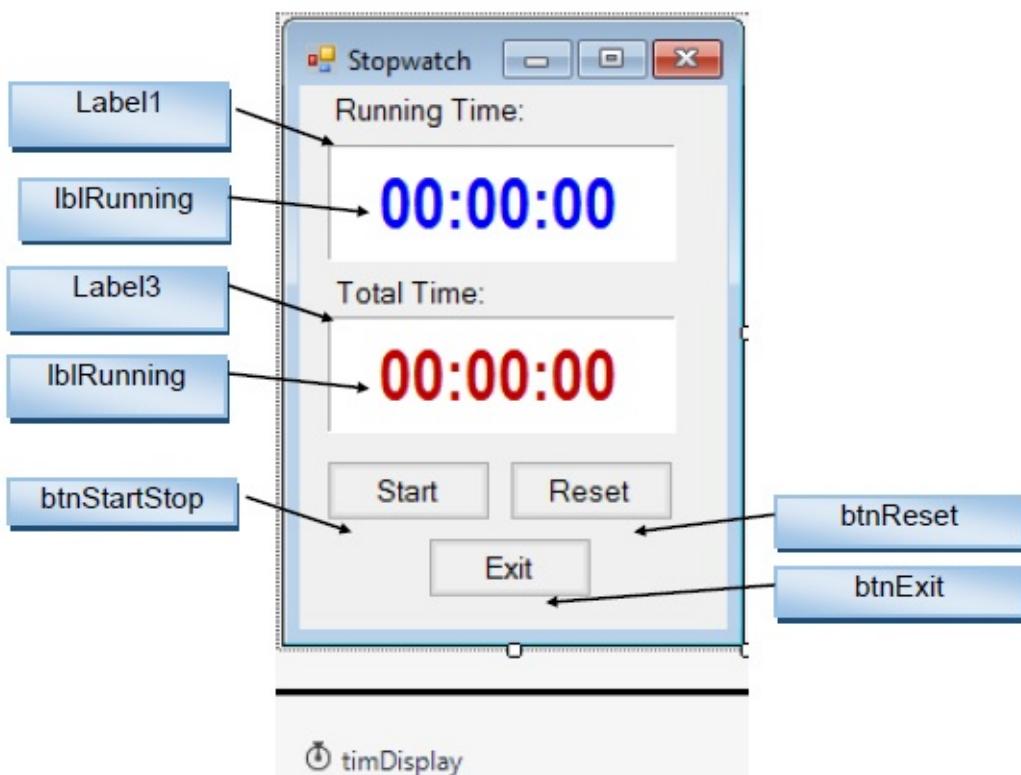
**button3** Button:

<b>Property Name</b>	<b>Property Value</b>
Name	btnExit
Text	Exit
Font Size	10

**timer1** Timer:

Property Name	Property Value
Name	timDisplay
Interval	1000

When done setting properties, my form looks like this:

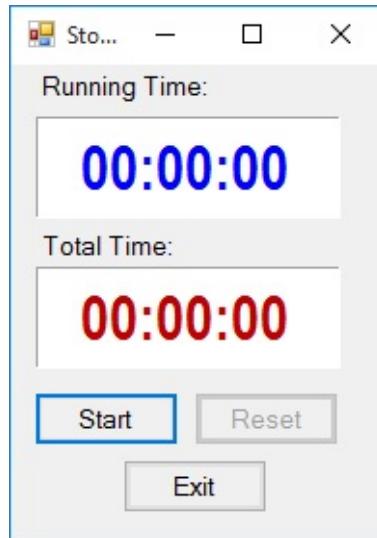


This completes the form design.

We will begin writing code for the application. We will write the code in several steps. As a first step, we will write the code that takes the stopwatch from this 'initial' state to its 'running' state, following clicking of the **Start** button. During the code development process, recognize you may modify a particular method several times before arriving at the finished product.

# Code Design – Initial to Running State

Even though we have yet to write any code, you can run the stopwatch project to make sure the form is properly initialized. Do this and you'll see in the ‘initial’ state (using default properties), the stopwatch looks like this:



We have two options at this point – either click **btnStartStop** (the button with **Start**) or click **btnExit** (the button with **Exit**). (Note the **Reset** button is initially disabled – you can't click this button until the stopwatch has been running.) We write code for both options.

First, the **btnExit Click** event is simply:

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

This simply says whenever the **Exit** button is clicked, the project closes. Add this code to the code window.

When the user clicks the **Start** button in ‘initial’ state, several things must happen to switch the stopwatch to ‘running’ state:

- Determine the starting time.
- Initialize the stopped time to zero.
- Start the timer control to begin updating the displays.
- Change the **Text** property of **btnStartStop** to **Stop**.
- Disable **btnExit**.

We will define two form level variables to track the starting time (a **DateTime** type) and the stopped time (a **TimeSpan** type):

```
DateTime startTime;
TimeSpan stoppedTime;
```

Place these statements in the general declarations area of the code window – after the method with the form constructor.

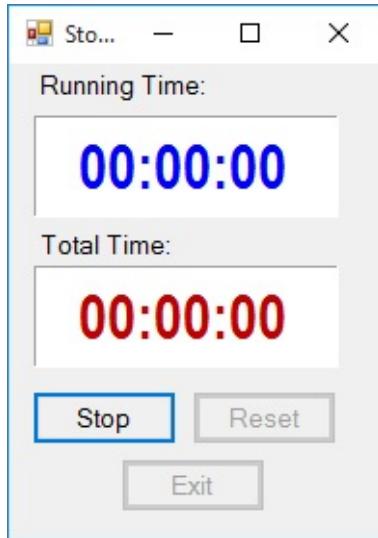
The code for the **btnStartStop Click** event that implements the listed steps is then:

```
private void btnStartStop_Click(object sender, EventArgs e)
{
    // initial to running state
    startTime = DateTime.Now;
    stoppedTime = new TimeSpan(0);
    timDisplay.Enabled = true;
    btnStartStop.Text = "Stop";
    btnExit.Enabled = false;
}
```

Note use of the **Now** method (a **DateTime** type) to obtain the starting time.

# Code Design – Timer Control

Save and run the project. Click the **Start** button and you should see:



The project is now in ‘running’ state. However, nothing is seen in the displays. We need to write code for the timer control **Tick** event to update the displayed times.

We set the timer control’s **Interval** property to 1000 milliseconds, or 1 second. Hence, every second, the timer control’s **Tick** event is invoked. Each time this event is invoked, we need to:

- Determine the current time.
- Subtract the current time from the start time to obtain the total time.
- Subtract the stopped time from the total time to get the running time.
- Display the total and running times in the appropriate label controls.

We find the current time using the **Now** method. The differences in times can be generated using a **TimeSpan** object:

```
TimeSpan diff = DateTime.Now - startTime;
```

The value **diff.Hours** returns the number of hours, **diff.Minutes** returns the

number of minutes and **diff.Seconds** returns the number of second in (**DateTime.Now – StartTime**).

Once we have a time difference, how do we display such a time in the desired format of **hours:minutes:seconds**? We need to do this twice in each **Tick** event, once for the running time and once for the total time. Whenever we need to repeat a certain task, something called a general method comes in very handy. We will use general methods many times in the projects in these notes, so we'll spend a little time looking at such methods.

# Using General Methods in Projects

Many projects have tasks that need to be repeated, such as the display of times with our stopwatch. Such tasks are usually coded in a general **method**. A method performs a specific task, using provided information to return some value.

Using general methods can help divide a complex application into more manageable units of code. This helps meet the earlier stated goals of readability and reusability. As you build applications, it will be obvious where such a method is needed. Look for areas in your application where code is repeated in different places. It would be best (shorter code and easier maintenance) to put this repeated code in a method. And, look for places in your application where you want to do some long, detailed task – this is another great use for a general method. It makes your code much easier to follow.

The form for a general method named **MyMethod** is:

```
private type MyMethod(arguments) // definition header
{
    [Method code]
    return(returnedValue);
}
```

The definition header names the **method**, specifies its **type** (the type of the returned value – if no value is returned, use the keyword **void**) and defines any input **arguments** passed to the method. The keyword **private** indicates the method will have form level scope.

**Arguments** are a comma-delimited list of variables passed to the method. If there are arguments, we need to take care in how they are declared in the header statement. In particular, we need to be concerned with:

- Number of arguments
- Order of arguments
- Type of arguments

We will address each point separately.

The **number** of arguments is dictated by how many variables the method needs to do its job. You need a variable for each piece of input information. You then place these variables in a particular **order** for the argument list.

Each variable in the argument list will be a particular **data type**. This must be known for each variable. In Visual C#, all variables are passed by value, meaning their value cannot be changed in the method. Variables are declared in the argument list using standard notation:

**type variableName**

The variable name (**variableName**) is treated as a local variable in the method.

Arrays can also be used as input arguments. To declare an array as an argument, use:

**type[] arrayName**

The brackets indicate an array is being passed.

To use a general method, simply refer to it, by name, in code (with appropriate arguments). Wherever it is used, it will be replaced by the computed value. A function can be used to return a value:

**rtnValue = MyMethod(arguments);**

or in an expression:

**thisNumber = 7 \* MyMethod(arguments) / anotherNumber;**

Let's build a quick example that converts Fahrenheit temperatures to Celsius. Here's such a function:

```
public double DegFTodegC(double tempF)
{
```

```
    double tempC;  
    tempC = (tempF - 32) * 5 / 9;  
    return(tempC);  
}
```

The method is named **DegFTodegC**. It has a single argument, **tempF**, of type **double**. It returns a **double** data type. This code segment converts 45.7 degrees Fahrenheit to the corresponding Celsius value:

```
double t;  
.  
. .  
t = DegFTodegC(45.7);
```

After this, **t** will have a value of **7.61 degrees C**.

To put a general method in a Visual C# application, simply type it among the event methods associated with controls. I usually put general methods below the last event method. Just make sure it is before the final closing brace for the Visual C# class defining the project. Type the header, the opening left brace, the code and the closing right brace.

# Code Design – Update Display

The job of the timer control in the stopwatch project is to update the displays of running and total times (values in seconds). The **Tick** event of that control will use a general method to generate these displays. The general method (named **HMS**) that converts a time value in seconds to the desired **hours:minutes:seconds** display is:

```
private string HMS(TimeSpan t)
{
    string hs, ms, ss;
    // Break time down into hours, minutes, and seconds
    hs = t.Hours.ToString();
    ms = t.Minutes.ToString();
    ss = t.Seconds.ToString();
    if (t.Hours < 10)
        hs = "0" + hs;
    if (t.Minutes < 10)
        ms = "0" + ms;
    if (t.Seconds < 10)
        ss = "0" + ss;
    return (hs + ":" + ms + ":" + ss );
}
```

In this method, the time span (**t**) is input as an argument. String representations of the hours (**hs**), minutes (**ms**) and seconds (**ss**) are determined. The returned value is a **string** type in the desired **hs:ms:ss** format for display. Work through this method with an example to convince yourself it works. Notice how we see if a leading zero is needed. Type the method into the code window.

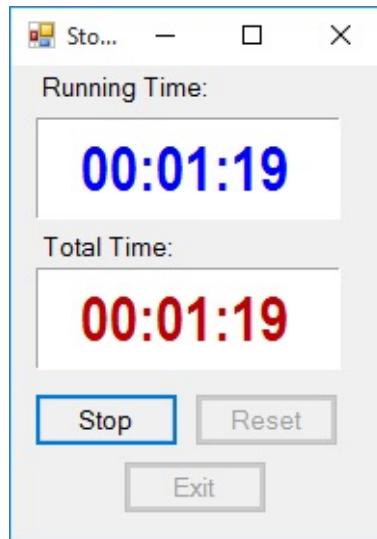
The **HMS** method is used in the timer control's **Tick** event, which has the steps outlined earlier. The **timDisplay Tick** event code is:

```
private void timDisplay_Tick(object sender, EventArgs e)
{
    TimeSpan totalTime;
    TimeSpan runningTime;
    // Determine total and running times in seconds
    totalTime = DateTime.Now - startTime;
    runningTime = totalTime - stoppedTime;
    // Display times
    lblTotal.Text = HMS(totalTime);
    lblRunning.Text = HMS(runningTime);
}
```

You should be able to see how this method computes the needed times and displays them using the **HMS** method. Add the method to the stopwatch project code window.

# Code Design – Running to Stopped State

Save and run the project. Click the **Start** button. The times should now be updating every second:



The project is now in the ‘running’ state. Only one option exists at this point – click **Stop** to put the stopwatch in ‘stopped’ state.

When a user clicks **Stop** (the **btnStartStop** button), the following things need to happen:

- Determine the stop time (not to be confused with the stopped time).
- Stop the timer control to stop updating the displays.
- Change the **Text** property of **btnStartStop** to **Restart**
- Enable **btnReset**.
- Enable **btnExit**.

We define another form level variable to store the stop time (a **DateTime** type):

```
DateTime stopTime;
```

Add this statement with the other general declarations.

The button now marked **Stop** is the **btnStartStop** button. We have already added some code to its **Click** event (when the button is used to start the stopwatch). It is common practice to have one button control have multiple purposes - we just need to have some way to distinguish which “mode” the button is in when it is clicked. In this project, we use the **Text** property of the button. If the **Text** property is **Start**, we switch to ‘running’ mode. If the **Text** property is **Stop**, we switch to ‘stopped’ mode. The code that does this is (modifications to the current **Click** event code are shaded):

```
private void btnStartStop_Click(object sender, EventArgs e)
{
    if (btnStartStop.Text == "Start")
    {
        // initial to running state
        startTime = DateTime.Now;
        stoppedTime = new TimeSpan(0);
        timDisplay.Enabled = true;
        btnStartStop.Text = "Stop";
        btnExit.Enabled = false;
    }
    else if (btnStartStop.Text == "Stop")
    {
        // running to stopped state
        stopTime = DateTime.Now;
        timDisplay.Enabled = false;
        btnStartStop.Text = "Restart";
        btnReset.Enabled = true;
        btnExit.Enabled = true;
    }
}
```

Make the noted modifications to the code.

# Code Design – Stopped State

Save and run the project. Click the **Start** button. Let the stopwatch run for a while, then click **Stop**. The stopwatch will go to ‘stopped’ state:



In this state, there are three possible options – clicking **Restart** (**btnStartStop**), clicking **Reset** (**btnReset**) or clicking **Exit** (**btnExit**). We’ll address each possibility in reverse order.

If **Exit** is clicked, the project ends. We have already coded the **btnExit Click** event.

If **Reset** is clicked, we want to return the stopwatch to its ‘initial’ state. The steps to do this are:

- Reset the displayed times to **00:00:00**
- Change the **Text** property of **btnStartStop** to **Start**
- Disable **btnReset**.

The **btnReset Click** event is thus:

```
private void btnReset_Click(object sender, EventArgs e)
{
```

```

// return to initial state
lblRunning.Text = "00:00:00";
lblTotal.Text = "00:00:00";
btnStartStop.Text = "Start";
btnReset.Enabled = false;
}

```

If **Restart** is clicked while in ‘stopped’ state, the **btnStartStop Click** event is processed. This is another use for the **btnStartStop** button. We need to modify the code already in that method to handle such an event

When a user clicks **Restart** (the **btnStartStop** button), the following things need to happen:

- Update (increment) the stopped time – add in the difference between the current time and the **StopTime**, the time when the **Stop** button was clicked.
- Start the timer control.
- Change the **Text** property of **btnStartStop** to **Stop**
- Disable **btnReset**.
- Disable **btnExit**.

The modified **btnStartStop Click** event that does implements these new steps (changes are shaded) is:

```

private void btnStartStop_Click(object sender, EventArgs e)
{
    if (btnStartStop.Text == "Start")
    {
        // initial to running state
        startTime = DateTime.Now;
        stoppedTime = new TimeSpan(0);
        timDisplay.Enabled = true;
        btnStartStop.Text = "Stop";
        btnExit.Enabled = false;
    }
}

```

```
}

else if (btnStartStop.Text == "Stop")
{
    // running to stopped state
    stopTime = DateTime.Now;
    timDisplay.Enabled = false;
    btnStartStop.Text = "Restart";
    btnReset.Enabled = true;
    btnExit.Enabled = true;
}

else if (btnStartStop.Text == "Restart")
{
    // stopped to running state
    stoppedTime += DateTime.Now - stopTime;
    timDisplay.Enabled = true;
    btnStartStop.Text = "Stop";
    btnReset.Enabled = false;
    btnExit.Enabled = false;
}

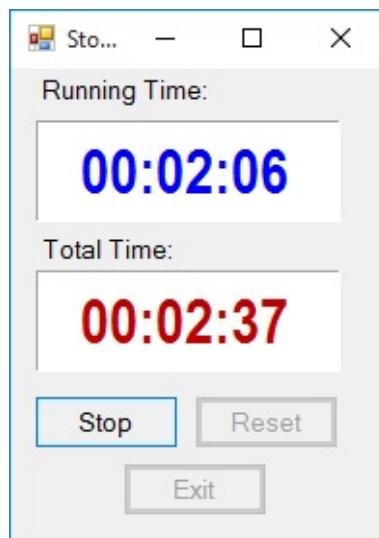
}
```

Notice how **stoppedTime** is updated. Implement the noted changes.

Save and run the project. At some point, click **Stop**. When I did, the form appears as:



After a wait, click **Restart**. When I do this, I get:



The stopwatch is running again, but the two displayed times are different. The **Total Time** is the amount of time elapsed since we first started the stopwatch. The **Running Time** is the total time less time when the stopwatch is in stopped mode. Based on these values, this stopwatch has spent 0:31 waiting around. Click **Stop** – make sure the **Reset** option works.

# Dual-Mode Stopwatch Project Review

The **Dual-Mode Stopwatch** project is now complete. Save and run the project and make sure it works as promised. Check that you can move from state to state correctly. Use it to time tasks as you work on your computer.

If there are errors in your implementation, go back over the steps of form and code design. Go over the developed code – make sure you understand how different parts of the project were coded. As mentioned in the beginning of this chapter, the completed project is saved as **Stopwatch** in the **HomeVCS\HomeVCS Projects** folder.

While completing this project, new concepts and skills you should have gained include:

- Proper steps in project design.
- Capabilities and use of the form, label, button and timer controls.
- How to use **DateTime** and **TimeSpan** types to find differences between two times.
- How to develop and use general methods.

# Dual-Mode Stopwatch Project Enhancements

There are always things you can do to improve a project. At the end of each chapter, we will give you some ideas for the current project. For the dual-mode stopwatch, some possibilities are:

- Whenever you stop the stopwatch, save that time. Then, when you ultimately stop the watch, provide a review mode. In review, you can see how much time was spent running the stopwatch and how much time was spent stopped.
- Provide an immediate feedback on each segment of elapsed time – a lap timing feature.

# 5

## **Consumer Loan Assistant Project**

# Review and Preview

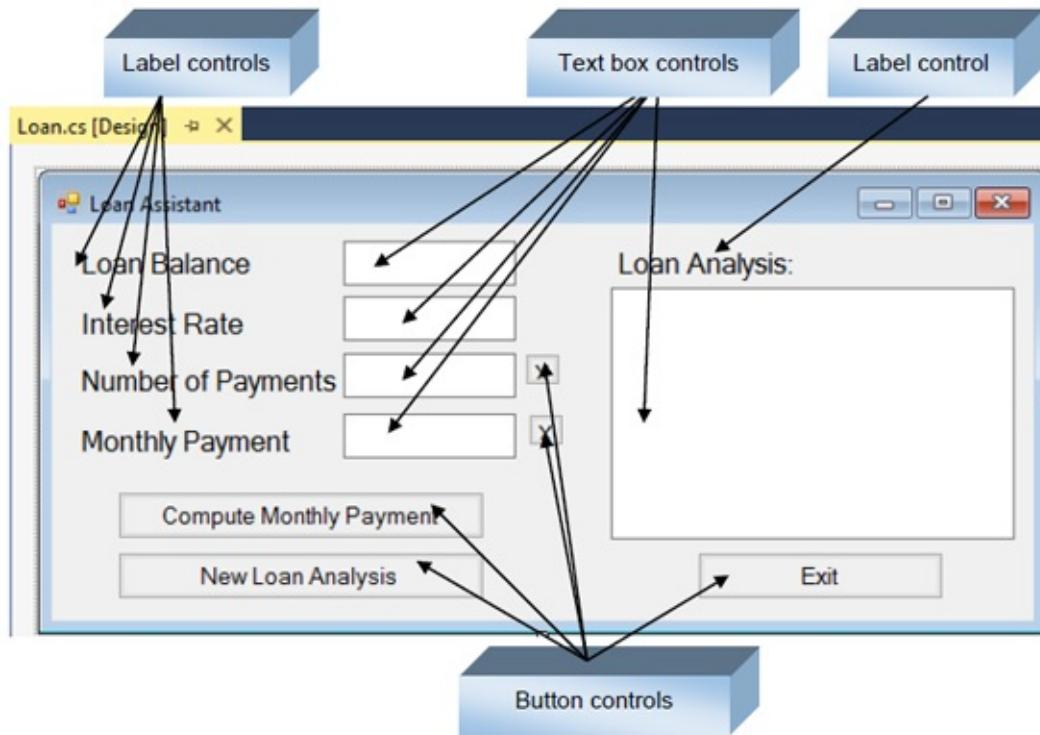
Ever wonder just how much those credit card accounts are costing you? This project will help you get a handle on consumer debt. The **Consumer Loan Assistant Project** we build computes payments and loan terms given balance and interest information.

We review use of the text box, a new control for this project. We also look at tab order among controls, techniques for intercepting user keystrokes, how to do input validation, and the message box for user feedback.

# Consumer Loan Assistant Project Preview

In this chapter, we will build a **consumer loan assistant**. You input a loan balance and yearly interest rate. You then have two options: (1) enter the desired number of payments and the loan assistant computes the monthly payment, or (2) enter the desired monthly payment and the loan assistant determines the number of payments you will make. An analysis of your loan, including total of payments and interest paid is also provided.

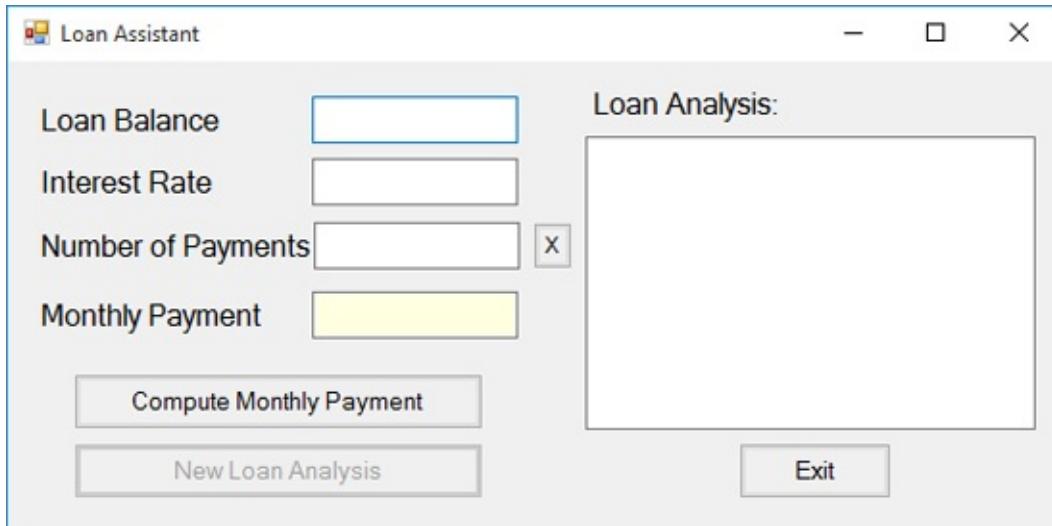
The finished project is saved as **Loan Assistant** in the **HomeVCS\HomeVCS Projects** folder. Start Visual C# and open the finished project. Open the form (double-click **Loan.cs** in Solution Explorer) and you will see:



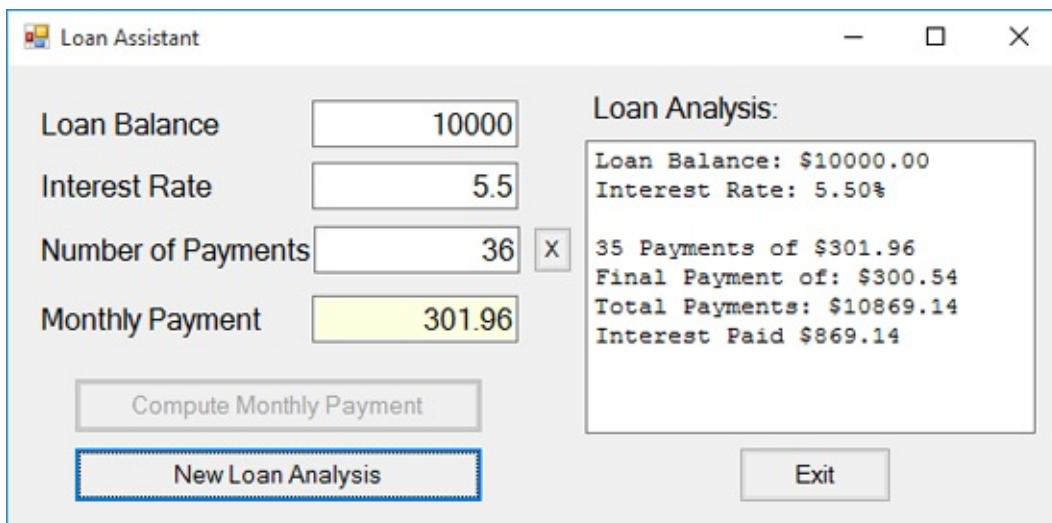
All label controls are used for title information. Two button controls are used to compute results and to start a new analysis. Two small button controls (marked with X) control whether you compute the number of payments or the payment amount. One button exits the project. Four text box controls are used for inputs

and a large text box is used to present the loan analysis results.

Run the project (press <F5>). The loan assistant will appear as:

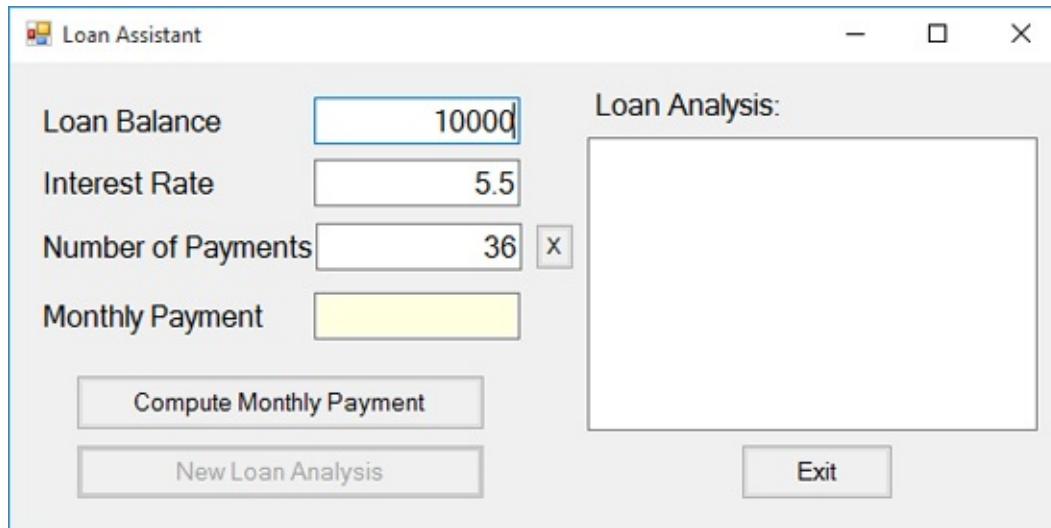


In this initial configuration, you enter a **Loan Balance**, an **Interest Rate** (annual rate as a percentage) and a **Number of Payments** value. Click **Compute Monthly Payment**. The payment will appear in the ‘yellow’ text box and a complete loan analysis will appear in the large text box. Here are some numbers I tried:



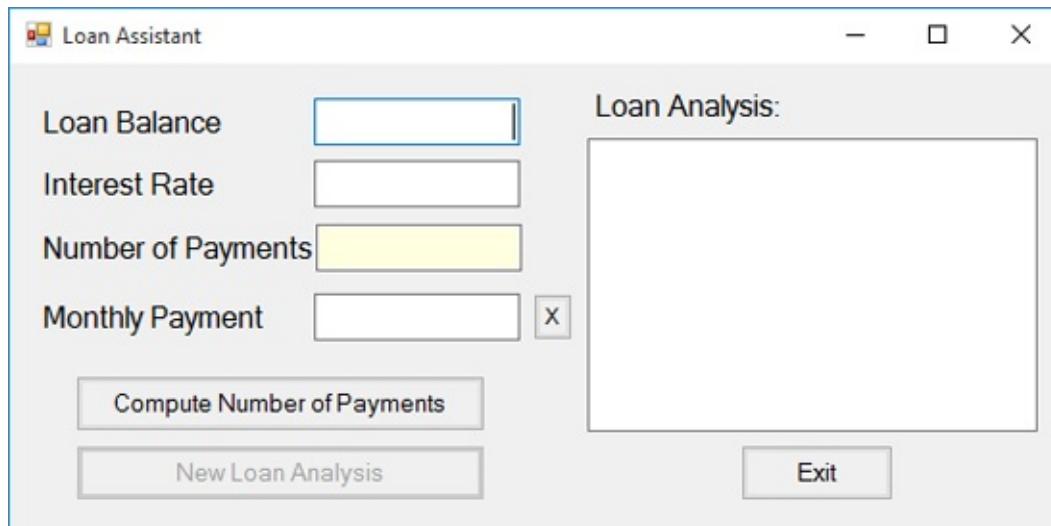
So, if I borrow \$10,000 at 5.5% interest, I will pay \$301.96 for three years (36 months). More specific details on exact payment amounts, including total interest paid, is shown under **Loan Analysis**.

At this point, you can click **New Loan Analysis** to try some new values:



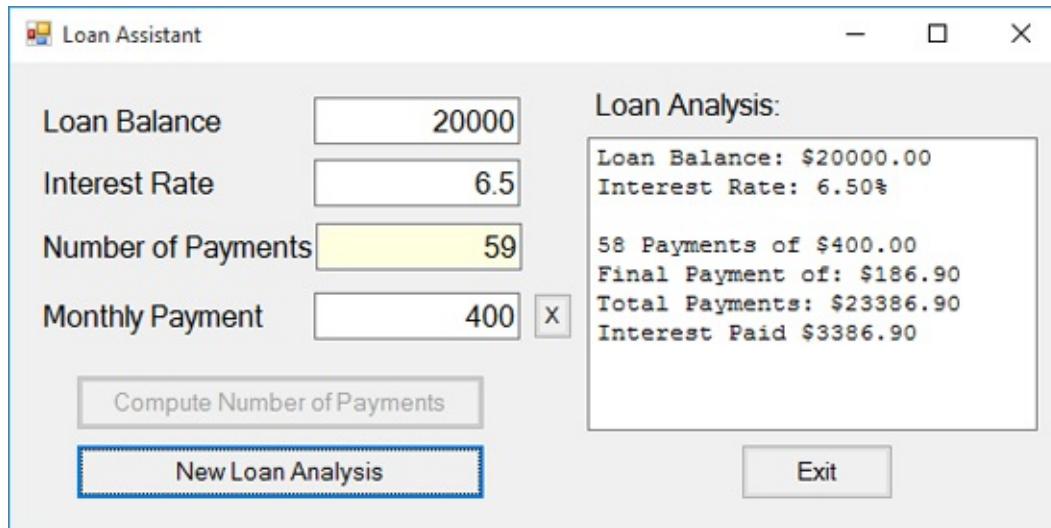
Note the **Loan Balance**, **Interest Rate**, and **Number of Payments** entries remain. Only the **Monthly Payment** and the **Loan Analysis** have been cleared. This lets you try different values with minimal typing of new entries. Change any entry you like to see different results – or even change them all. Try as many combinations as you like.

At some point, clear the text boxes and click the button with an X next to the **Number of Payments** text box. You will see:



Notice the **Number of Payments** box is now yellow. The button with an X has moved to the **Monthly Payment** text box. In this configuration, you enter a

**Loan Balance**, an **Interest Rate** and a **Monthly Payment**. The loan assistant will determine how many payments you need to pay off the loan. Here are some numbers I tried:



It will take 59 payments (the last one is smaller) to pay off this particular loan. Again, you can click **New Loan Analysis** to try other values and see the results.

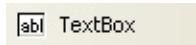
That's all you do with the loan assistant project – there's a lot going on behind the scenes though. The loan assistant has two modes of operation. It can compute the monthly payment, given the balance, interest and number of payments. Or, it can compute the number of payments, given the balance, interest, and payment. The text box representing the computed value is yellow. The button marked X is used to switch from one mode to the next. To exit the project, click the **Exit** button.

You will now build this project in several stages. We first address **form design**. We discuss the controls used to build the form, establish initial properties, and discuss switching from one mode to the next. And, we address **code design** in detail. We cover the mathematics behind the financial computations. We also discuss validation of the input values, making sure the user only types valid entries.

This is the first project in these notes where we have used a text box control. Let's review its use first.

# TextBox Control

## In Toolbox:



## On Form (Default Properties):



A **TextBox** control is used to display information entered at design time, by a user at run-time, or assigned within code. The displayed text may be edited.

## TextBox Properties:

<b>Name</b>	Gets or sets the name of the text box (three letter prefix for text box name is <b>txt</b> ).
<b>AutoSize</b>	Gets or sets a value indicating whether the height of the text box automatically adjusts when the font assigned to the control is changed.
<b>BackColor</b>	Get or sets the text box background color.
<b>BorderStyle</b>	Gets or sets the border style for the text box.
<b>Font</b>	Gets or sets font name, style, size.
<b>ForeColor</b>	Gets or sets color of text or graphics.
<b>HideSelection</b>	Gets or sets a value indicating whether the selected text in the text box control remains highlighted when the control loses focus.
<b>Lines</b>	Gets or sets the lines of text in a text box control.
<b>MaxLength</b>	Gets or sets the maximum number of characters the user can type into the text box control.
<b>MultiLine</b>	Gets or sets a value indicating whether this is a multiline text box control.
<b>PasswordChar</b>	Gets or sets the character used to mask characters of a password in a single-line TextBox control.
<b>ReadOnly</b>	Gets or sets a value indicating whether text in the

	text box is read-only.
<b>ScrollBars</b>	Gets or sets which scroll bars should appear in a multiline TextBox control.
<b>SelectedText</b>	Gets or sets a value indicating the currently selected text in the control.
<b>SelectionLength</b>	Gets or sets the number of characters selected in the text box.
<b>SelectionStart</b>	Gets or sets the starting point of text selected in the text box.
<b>Tag</b>	Stores a string expression.
<b>Text</b>	Gets or sets the current text in the text box.
<b>TextAlign</b>	Gets or sets the alignment of text in the text box.
<b>TextLength</b>	Gets length of text in text box.

### TextBox Methods:

<b>AppendText</b>	Appends text to the current text of text box.
<b>Clear</b>	Clears all text in text box.
<b>Copy</b>	Copies selected text to clipboard.
<b>Cut</b>	Moves selected text to clipboard.
<b>Focus</b>	Places the cursor in a specified text box.
<b>Paste</b>	Replaces the current selection in the text box with the contents of the Clipboard.
<b>SelectAll</b>	Selects all text in text box.
<b>Undo</b>	Undoes the last edit operation in the text box.

### TextBox Events:

<b>Click</b>	Occurs when the user clicks the text box.
<b>Enter</b>	Occurs when the control receives focus.
<b>KeyDown</b>	Occurs when a key is pressed down while the control has focus.
<b>KeyPress</b>	Occurs when a key is pressed while the control has focus.
<b>Leave</b>	Triggered when the user leaves the text box. This

is a good place to examine the contents of a text box after editing.

<b>TextChanged</b>	Occurs when the <b>Text</b> property value has changed.
--------------------	---

Typical use of **TextBox** control as display control:

- Set the **Name** property. Initialize **Text** property to desired string.
- Set **ReadOnly** property to **True**.
- If displaying more than one line, set **MultiLine** property to **True**.
- Assign **Text** property in code where needed.
- You may also want to change the **Font**, **Backcolor** and **Forecolor** properties.

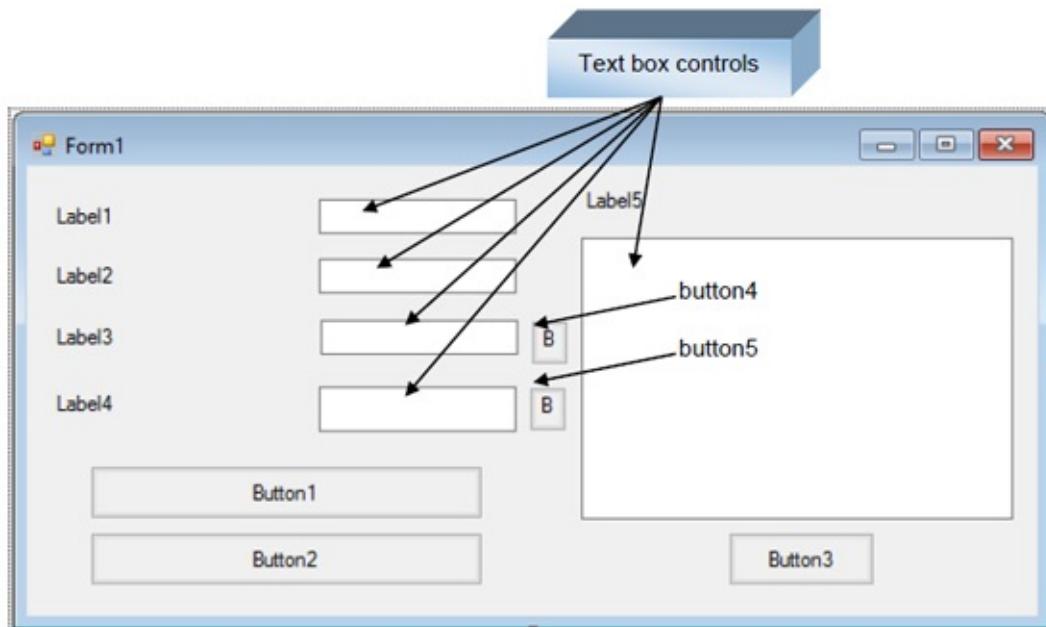
Typical use of **TextBox** control as input device:

- Set the **Name** property. Initialize **Text** property to desired string.
- If it is possible to input multiple lines, set **MultiLine** property to **True**. Set **ScrollBar** property.
- In code, give **Focus** to control when needed. Provide key trapping code in **KeyPress** event (discussed later in this chapter). Read **Text** property when **Leave** event occurs.
- You may also want to change the **Font**, **Backcolor** and **Forecolor** properties.

Use of the **TextBox** control should be minimized if possible. Whenever you give a user the option to type something, it makes your job as a programmer more difficult. You need to validate the information they type to make sure it will work with your code (we will do this with the loan assistant). There are many controls in Visual C# that are ‘point and click,’ that is, the user can make a choice simply by clicking with the mouse. We’ll look at such controls through these notes. Whenever these ‘point and click’ controls can be used to replace a text box, do it!

# Loan Assistant Form Design

Having reviewed the text box control, we can begin building the loan assistant project. Let's build the form. Start a new project in Visual C#. Place five label controls, five text box controls and five button controls on your form. Resize and position controls so the form looks similar to this:



The label controls are used for title information. The four smaller text boxes are for user input. The large text box will display the loan analysis. The larger buttons are used to compute loan results, redo analysis, and/or exit the project. The two small button controls are used to switch from one calculation mode to the next.

Set the control properties using the properties window:

**Form1** Form:

Property Name	Property Value
Name	frmLoan
Text	Loan Assistant

FormBorderStyle Fixed Single  
StartPosition CenterScreen

**label1** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Loan Balance
Font Size	12

**label2** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Interest Rate
Font Size	12

**label3** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Number of Payments
Font Size	12

**label4** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Payment Amount
Font Size	12

**label5** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Loan Analysis:
Font Size	12

**textBox1** Text Box:

<b>Property Name</b>	<b>Property Value</b>
Name	txtBalance

TextAlign	Right
Font Size	12

**textBox2** Text Box:

<b>Property Name</b>	<b>Property Value</b>
Name	txtInterest
TextAlign	Right
Font Size	12

**textBox3** Text Box:

<b>Property Name</b>	<b>Property Value</b>
Name	txtMonths
TextAlign	Right
Font Size	12

**textBox4** Text Box:

<b>Property Name</b>	<b>Property Value</b>
Name	txtPayment
TextAlign	Right
Font Size	12

**textBox5** Text Box:

<b>Property Name</b>	<b>Property Value</b>
Name	txtAnalysis
Font Name	Courier New
Font Size	10
ReadOnly	True
BackColor	White

**button1** Button:

<b>Property Name</b>	<b>Property Value</b>

Name	btnCompute
Text	Compute Monthly Payment
Font Size	10

**button2** Button:

<b>Property Name</b>	<b>Property Value</b>
Name	btnNewLoan
Text	New Loan Analysis
Enabled	False
Font Size	10

**button3** Button:

<b>Property Name</b>	<b>Property Value</b>
Name	btnExit
Text	Exit
Font Size	10

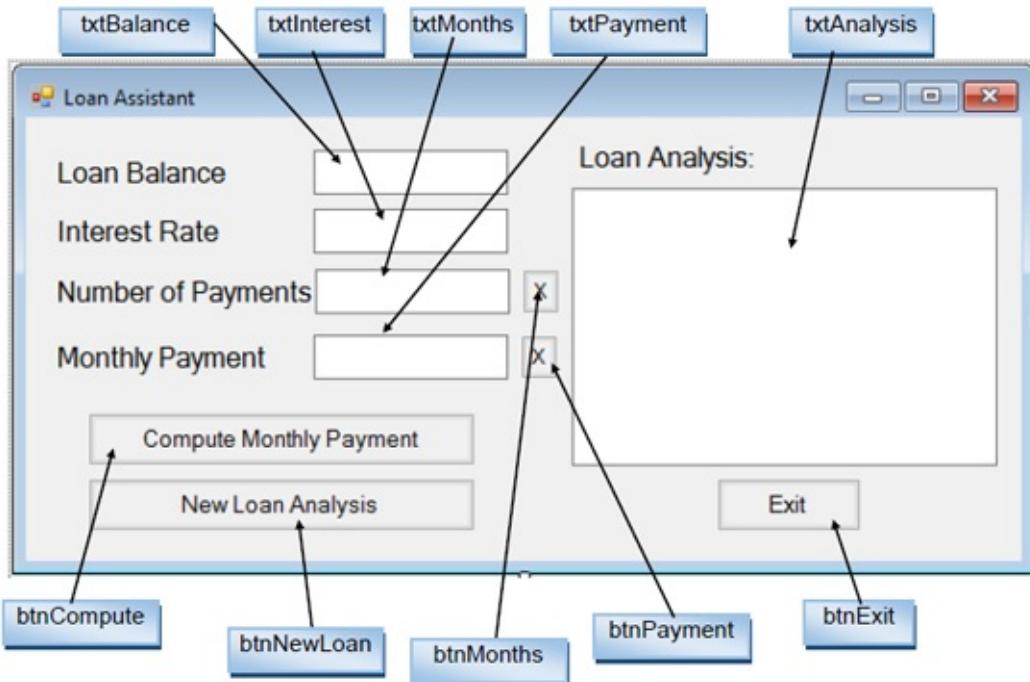
**button4** Button:

<b>Property Name</b>	<b>Property Value</b>
Name	btnMonths
Text	X
Font Size	10

**button5** Button:

<b>Property Name</b>	<b>Property Value</b>
Name	btnPayment
Text	X
Font Size	10

When done setting properties, my form looks like this:



This completes the initial form design (we have not identified the label controls – they should be obvious).

We will begin writing code for the application. We will write the code in several steps. As a first step, we write the code that switches the application between its two possible modes of operation: (1) compute monthly payment, or (2) compute number of payments.

# Code Design – Switching Modes

There are two modes the loan assistant can operate in. In the first mode, you enter a loan balance, an interest rate and a number of payments. The assistant then computes the monthly payment. In the second mode, you enter a loan balance, an interest rate and a monthly payment. The assistant computes the number of payments. The buttons with X control which mode the assistant operates in. Click the X (**btnPayment**) next to the payment text box and you switch to the first mode (compute monthly payment). Click the X (**btnMonths**) next to the number of payments text box and you switch to the second mode (compute number of payments). Let's look at the steps for each operation.

When the user clicks the X next to the monthly payment text box (**btnPayment** button), we want to make **txtPayment** available for user input and **txtMonths** available for output. The steps are taken:

- Make **btnPayment** disappear.
- Make **btnMonths** appear.
- Set **ReadOnly** property of **txtMonths** to **False**.
- Set **txtMonths** background to **White**.
- Blank out the **txtPayment** text box.
- Set **ReadOnly** property of **txtPayment** to **True**.
- Set **txtPayment** background to **LightYellow**.
- Set **Text** property of **btnCompute** to **Compute Monthly Payment**.

When you click the X next to the number of payments text box (**btnMonths** button), we essentially ‘reverse’ the steps just listed:

- Make **btnPayment** appear.
- Make **btnMonths** disappear.
- Set **ReadOnly** property of **txtMonths** to **True**.
- Set **txtMonths** background to **LightYellow**.
- Blank out the **txtMonths** text box.
- Set **ReadOnly** property of **txtPayment** to **False**.

- Set **txtPayment** background to **White**.
- Set **Text** property of **btnCompute** to **Compute Number of Payments**.

Define a form level variable to keep track of what mode we are working in:

```
bool computePayment;
```

If **computePayment** is **true**, we are computing the payment, otherwise we are computing the number of payments.

The code for the **btnPayment Click** event that implements the listed steps is then:

```
private void btnPayment_Click(object sender, EventArgs e)
{
    // will compute payment
    computePayment = true;
    btnPayment.Visible = false;
    btnMonths.Visible = true;
    txtMonths.ReadOnly = false;
    txtMonths.BackColor = Color.White;
    txtPayment.Text = "";
    txtPayment.ReadOnly = true;
    txtPayment.BackColor = Color.LightYellow;
    btnCompute.Text = "Compute Monthly Payment";
}
```

The code for the **btnMonths Click** is:

```
private void btnMonths_Click(object sender, EventArgs e)
{
    // will compute months
    computePayment = false;
    btnPayment.Visible = true;
```

```

btnMonths.Visible = false;
txtMonths.Text = "";
txtMonths.ReadOnly = true;
txtMonths.BackColor = Color.LightYellow;
txtPayment.ReadOnly = false;
txtPayment.BackColor = Color.White;
btnCompute.Text = "Compute Number of Payments";
}

```

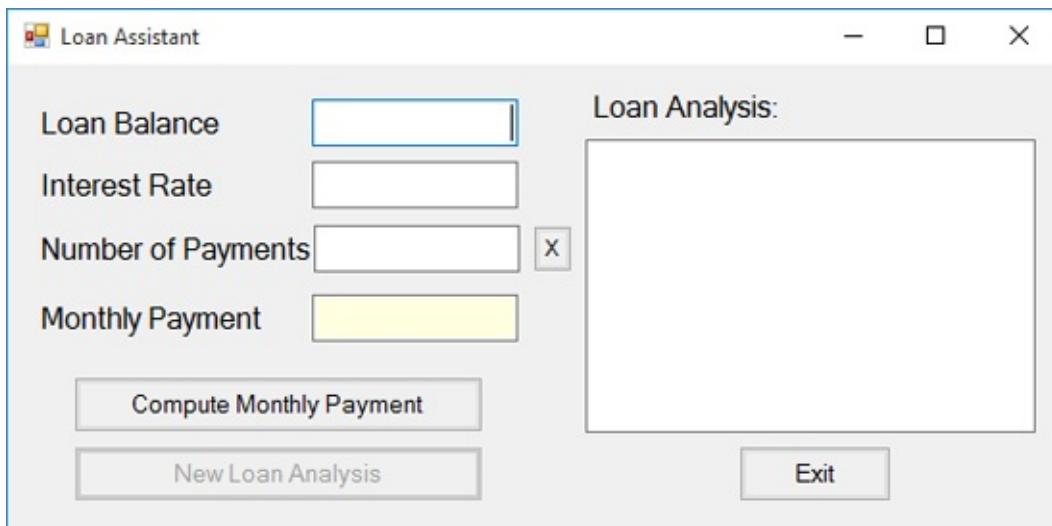
We would like the application to begin in the mode where the monthly payment is computed. One way we could do this is by setting properties in design mode that correspond to the properties listed in the **btnPayment Click** method. But an easier approach is to have the application ‘simulate’ clicking on the **btnPayment** button when the application begins. This can be done in the **frmLoan Load** event using this code:

```

private void frmLoan_Load(object sender, EventArgs e)
{
    btnPayment.PerformClick();
}

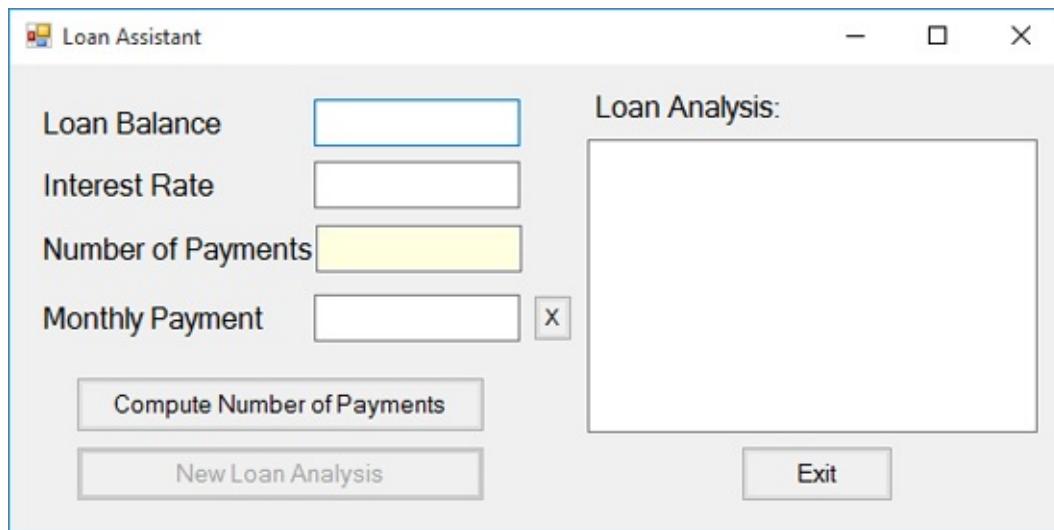
```

Save and run the project. If the code is entered correctly, the form should appear in the ‘compute payment’ mode:



Note the **Monthly Payment** box is yellow, as desired. The **btnCompute** caption is **Compute Monthly Payment**.

Click the **X** next to **Number of Payments** and you switch to the ‘compute number of payments’ mode:



Now, the **Number of Payments** box is yellow and the **btnCompute** caption is **Compute Number of Payments**.

The mode switching should be working correctly. Before writing the code behind the actual computations, let's address interface issues of tab ordering and control focus.

# Form Design - Tab Order and Focus

When you run the loan assistant application, the cursor may not necessarily start out at the top text box where you enter the **Loan Balance**. And, if you try to move from text box to text box using the <Tab> key, there may be no predictable order in how the cursor moves. Or, you may move to controls you don't want to move to (for example, a read-only text box). To enter values, you have to make sure you first click in the text box. To make this process more orderly, we need to understand two properties of controls: **TabStop** and **TabIndex**.

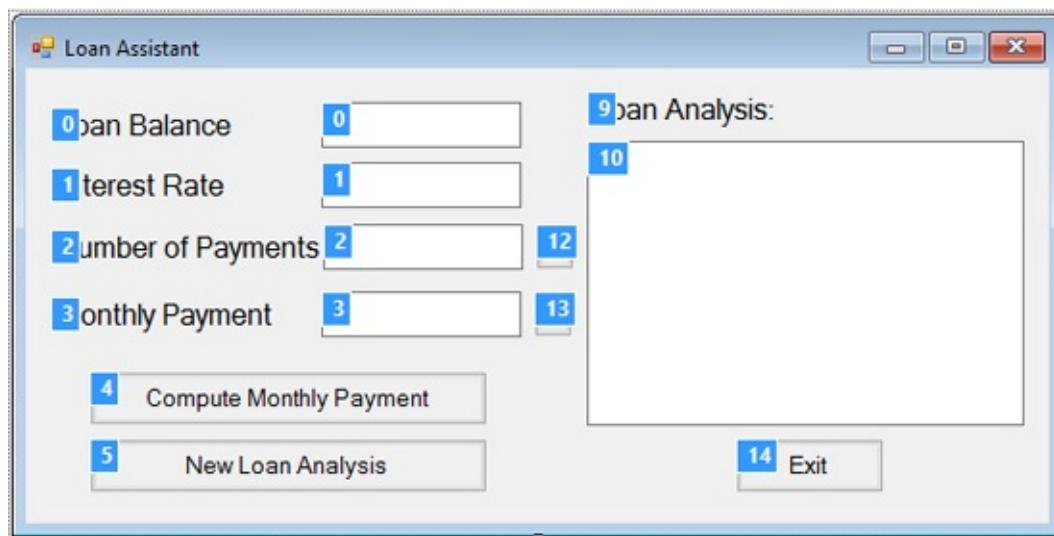
When interacting with a Windows application, we can work with a single control at a time. That is, we can click on a single button or type in a single text box. We can't be doing two things at once. The control we are working with is known as the **active** control or we say the control has **focus**. In our loan assistant project, when the cursor is in a particular text box, we say that text box has focus. If the user begins typing, the typed characters go in that text box. If a button control has focus, that button can be 'clicked' by simply pressing the <Enter> key. In a properly designed application, focus is shifted from one control to another (in a predictable, orderly fashion) using the <Tab> key.

To define an orderly Tab response, we do two things. First, for each control you want accessible via the <Tab> key, make sure its **TabStop** property is **True** (the default value). All non-accessible controls should have the TabStop property set to False. In the loan assistant project, the only controls we want to have tab stops are the text box controls used for entries and the **btnCompute** and **btnNewLoan** button controls. Return to the loan assistant project and set the **TabStop** property of **txtAnalysis**, **btnMonths**, **btnPayment**, and **btnExit** to **False**. If you rerun the project at this point, you should see the focus is never given to these controls using the <Tab> key.

Second, the order in which the controls will be accessed (given focus) is set by the **TabIndex** property. The control with the lowest TabIndex property will be the first control with focus. Higher subsequent TabIndex properties will be followed with each touch of the <Tab> key. The process can be reversed using <Tab> in combination with the <Shift> key. Once the highest TabIndex property is found, the Tab process restarts at the lowest value.

There are two ways to set **TabIndex** values. You can set the property of each control using the Properties window. This is tedious especially when new controls are introduced to the form. The Visual C# IDE offers a great tool for setting the **TabIndex** property of all controls. Let's look at that tool.

Return to your loan assistant project. With the form selected (has resizing handles), choose the **View** menu option, and choose **Tab Order**. You should see something like this:



The little blue number on each control is its corresponding **TabIndex** property. To set (or reset) these values, simply click on each control in the sequence you want the Tab ordering to occur in. We chose to click on the four smaller text boxes (in order, TabIndex 0 - 3) and the two large button controls (TabIndex 4 and 5).

If you click on a control with a False **TabStop** property, the index value will be ignored in the Tab sequence. To turn off the tab order display, return to the **TabIndex** View menu and choose **TabOrder** again (make sure the form is active while doing this or that menu item does not appear in the View menu).

Notice when the loan assistant project is in the 'compute payment' mode, we don't want to be able to tab to the **Monthly Payment** text box. Likewise, when in 'compute number of payments' mode, we don't want to be able to tab to the **Number of Payments** text box. This adjustment of the **TabStop** must be done in

code.

With tab ordering, the <Tab> key is used to move from control to control, shifting the focus. Many times, you might like to move focus from one control to another in code, or programmatically. For example, in our loan assistant, once a user changes the computation mode, it would be nice if focus would be moved to the **Loan Balance** text box so the user can type an entry there. To programmatically assign focus to a control, apply the **Focus** method to the control using this dot-notation:

```
controlName.Focus();
```

Modifications to the **btnPayment** and **btnMonths Click** events to account for tab ordering and focus are shown as shaded lines:

```
private void btnPayment_Click(object sender, EventArgs e)
{
    // will compute payment
    computePayment = true;
    btnPayment.Visible = false;
    btnMonths.Visible = true;
    txtMonths.ReadOnly = false;
    txtMonths.TabStop = true;
    txtMonths.BackColor = Color.White;
    txtPayment.Text = "";
    txtPayment.ReadOnly = true;
    txtPayment.TabStop = false;
    txtPayment.BackColor = Color.LightYellow;
    btnCompute.Text = "Compute Monthly Payment";
    txtBalance.Focus();
}

private void btnMonths_Click(object sender, EventArgs e)
{
```

```
// will compute months
computePayment = false;
btnPayment.Visible = true;
btnMonths.Visible = false;
txtMonths.Text = "";
txtMonths.ReadOnly = true;
txtMonths.TabStop = false;
txtMonths.BackColor = Color.LightYellow;
txtPayment.ReadOnly = false;
txtPayment.TabStop = true;
txtPayment.BackColor = Color.White;
btnCompute.Text = "Compute Number of Payments";
txtBalance.Focus();
}
```

Make these modifications to the code.

Save and run the project. Switch from mode to mode. Notice how in each mode, the tab ordering is now predictable and as desired. Notice how the focus is always on the desired control.

# Code Design – Computing Monthly Payment

Let's develop the code to run the loan assistant in its initial 'compute payment' mode. We need an equation that computes the payment, knowing the loan balance, the interest rate and the number of payments. Computer programming is many times mathematical in nature. I recognize different people have different comfort levels with math. For those "math-phobes" out there, I'll just give you the code. For those interested, I'll show you the math behind the code.

Here's the code that does the necessary computations. In these lines, **balance** (**double** type) is the entered loan balance, **interest** (**double** type) is the entered interest rate and **months** (**int** type) is the entered number of payments (each of these values will come from the text box controls):

```
multiplier = Math.Pow(1 + monthlyInterest, months); payment = balance  
* monthlyInterest * multiplier / (multiplier - 1);
```

In this code, the input interest (a yearly percentage) is converted to a monthly interest (**monthlyInterest**). This conversion is done by dividing by 12 (the number of months in a year) times 100 (to convert percentage to a decimal number). A **multiplier** term is formed using the mathematical power (exponentiation) method (**Pow**). These values are then used to compute **payment** (**double** type).

If you don't want to see mathematics, **stop now!!** Skip ahead to the code steps for the **btnCompute Click** event. If you're still with me, I'll go over the steps that derive the code above. Let  $B$  represent the initial loan balance,  $i$  the monthly interest and  $P$  the monthly payment (we'll be solving an equation for this value). With this notation, the product of  $B$  times  $i$  ( $Bi$ ) represents one month's interest on the existing balance. We add this interest to the balance then subtract the payment to obtain the balance after one payment,  $B_1$ :

$$B_1 = B + Bi - P = B(1 + i) - P$$

Using the same approach, the balance after two payments ( $B_2$ ) would be:

$$B_2 = B_1 + B_1i - P = B_1(1 + i) - P$$

Substituting the previous equation for  $B_1$  into this equation gets things in terms of the original balance:

$$B_2 = [B(1 + i) - P](1 + i) - P = B(1 + i)^2 - P(1 + i) - P$$

Doing the same for  $B_3$ , we can show:

$$B_3 = B(1 + i)^3 - P(1 + i)^2 - P(1 + i) - P$$

Noting the trend in this relation, we can obtain an expression for  $B_N$  (the balance after  $N$  payments, when the loan is finally paid off):

$$B_N = B(1 + i)^N - P \sum_{k=0}^{N-1} (1 + i)^k$$

The Greek sigma in the above equation simply indicates that you add up all the corresponding elements next to the sigma.

After  $N$  payments, we want the balance of the loan to be zero. If we set  $B_N$  to zero in the above equation, we obtain a value for  $P$ , the payment:

$$P = B(1 + i)^N / \sum_{k=0}^{N-1} (1 + i)^k$$

This is the desired result and we could easily code it using a for loop to evaluate the summation in the denominator. We can avoid this step by consulting a handbook on “finite series.” The denominator term actually has a “closed-form” (one not requiring the summation). It is (trust me on this):

$$\sum_{k=0}^{N-1} (1 + i)^k = [(1 + i)^N - 1]/i$$

Try a few values of  $i$  and  $N$  to convince yourself this works (if you need convincing). Substituting this into the equation for  $P$  and flipping a few terms around gives us the final equation for computing  $P$ :

$$P = Bi(1 + i)^N / [(1 + i)^N - 1]$$

Compare this equation to the code we gave you. You should see the code matches this equation (**B** is **balance**, **i** is **monthlyInterest**, **N** is **months** and **P** is **payment**).

When the user clicks **Compute Monthly Payment** (**btnCompute**), the following steps are taken:

- Obtain the **balance** value from user input.
- Obtain the **interest** value from user input.
- Determine monthly interest.
- Obtain the **months** value from user input.
- Compute **payment** using given code.
- Display **payment** in **txtPayment**.

The **btnCompute Click** event that implements these steps are (note we have declared all variables to have method level scope):

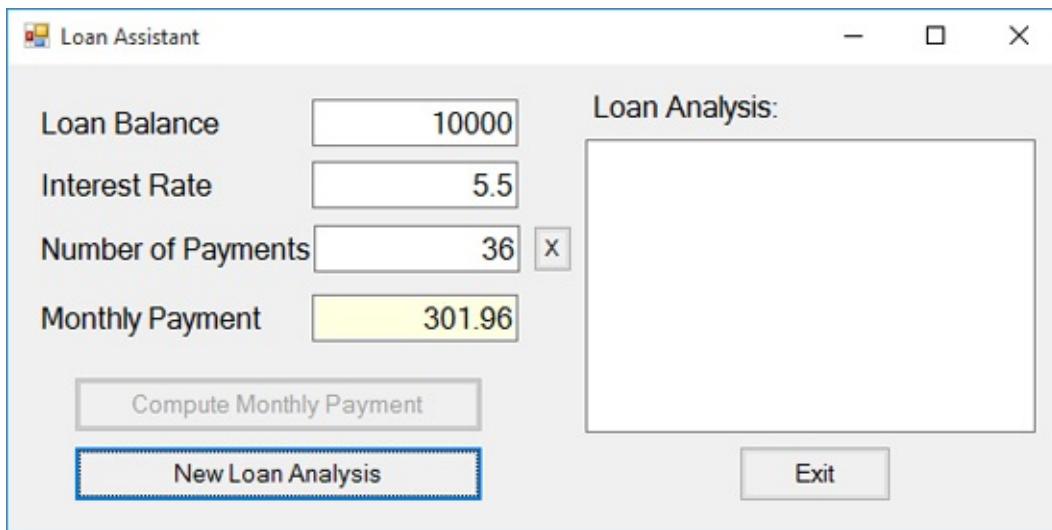
```
private void btnCompute_Click(object sender, EventArgs e)
{
    double balance, interest, payment;
    int months;
    double monthlyInterest, multiplier;
    balance = Convert.ToDouble(txtBalance.Text);
    interest = Convert.ToDouble(txtInterest.Text);
    monthlyInterest = interest / 1200;
    // Compute loan payment
    months = Convert.ToInt32(txtMonths.Text);
    multiplier = Math.Pow(1 + monthlyInterest, months);
    payment = balance * monthlyInterest * multiplier / (multiplier - 1);
```

```
txtPayment.Text = String.Format("{0:f2}", payment);
}
```

While we're at it, let's take care of the **Exit** button. Its **Click** event is simply:

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

Save and run the project. Enter some numbers for balance, interest and number of payments, then click **Compute Monthly Payment**. Here's a run I made:



A \$10,000 loan at 5.5% yearly interest has a monthly payment of \$301.96. Try as many possibilities as you'd like. Make sure **Exit** works.

# Code Design – Computing Number of Payments

The second mode of operation for the loan assistant is ‘compute number of payments’ mode. We need an equation the computes the number of payments, knowing the loan balance, the interest rate and the monthly payment. Again, a bit of math is involved. And, again, for those interested, I’ll show you the math behind the code.

Here’s the code that does the necessary computations. In these lines, **balance** (**double** type) is the entered loan balance, **interest** (**double** type) is the entered interest rate and **payment** (**double** type) is the entered monthly payment (each of these values will come from the text box controls):

```
monthlyInterest = interest / 1200;  
months = (int)((Math.Log(payment) - Math.Log(payment - balance *  
monthlyInterest)) / Math.Log(1 + monthlyInterest));
```

In this code, we again use the **monthlyInterest** value. The number of payments (**months**, an **int** type) is computed using the **Math.Log** function. This is a mathematical logarithm.

All “math-phobes,” skip ahead to the code to modify the **btnCompute Click** event. For those interested, let’s see where that logarithm comes from. The equation we derived for the **Payment (P)** was:

$$P = Bi(1 + i)^N / [(1 + i)^N - 1]$$

where **B** is **Balance**, **i** is **MonthlyInterest**, and **N** is **Months**. In the current mode, we want to solve for N, given B, i, and P. Multiply both sides of the equation by the denominator on the right side to get:

$$[(1 + i)^N - 1]P = Bi(1 + i)^N$$

Multiply out the left side:

$$(1 + i)^N P - P = Bi(1 + i)^N$$

Then collect terms:

$$(P - Bi)(1 + i)^N = P$$

or

$$(1 + i)^N = P / (P - Bi)$$

Now, take the logarithm (hopefully you remember how these work) of both sides to yield:

$$N \log(1 + i) = \log(P) - \log(P - Bi)$$

Or, solving for N, our desired result:

$$N = [\log(P) - \log(P - Bi)] / \log(1 + i)$$

Look back at the code and you should see this equation. In the code, we cast the result to an **int** type (we can't make a fractional payment).

So when the user clicks **Compute Number of Payments** (**btnCompute** when **ComputePayment** is **False**), the following steps are taken:

- Obtain the **balance** value from user input.
- Obtain the **interest** value from user input.
- Determine monthly interest.
- Obtain the **payment** value from user input.
- Compute **months** using given code.
- Display **months** in **txtMonths**.

The modified **btnCompute Click** event that implements these steps are (new code is shaded, notice we now look **ComputePayment** to see what 'mode' we are in):

```
private void btnCompute_Click(object sender, EventArgs e)
```

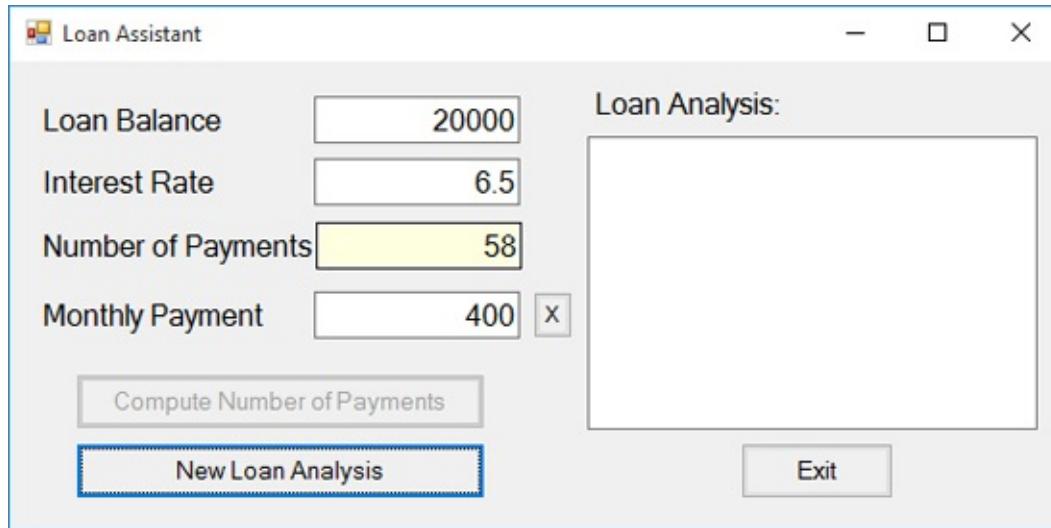
```

{
    double balance, interest, payment;
    int months;
    double monthlyInterest, multiplier;
    balance = Convert.ToDouble(txtBalance.Text);
    interest = Convert.ToDouble(txtInterest.Text);
    monthlyInterest = interest / 1200;
    if (computePayment)
    {
        // Compute loan payment
        months = Convert.ToInt32(txtMonths.Text);
        multiplier = Math.Pow(1 + monthlyInterest, months);
        payment = balance * monthlyInterest * multiplier / (multiplier -
1);
        txtPayment.Text = String.Format("{0:f2}", payment);
    }
    else
    {
        // Compute number of payments
        payment = Convert.ToDouble(txtPayment.Text);
        months = (int)((Math.Log(payment) - Math.Log(payment -
balance * monthlyInterest)) / Math.Log(1 + monthlyInterest));
        txtMonths.Text = months.ToString();
    }
}

```

Save and run the application. Make sure it still works in the initial mode for computing the monthly payment. When you're sure this is working okay, click the X next to the number of payment text box to switch to ‘compute number of payments’ mode.

Type in some values for balance, interest and payment. Click **Compute Number of Payments**. Here’s a run I made:



This tells me if I borrow \$20, 000 at 6.5% interest, I would need to make 58 monthly payments of \$400 to pay the loan back. If you have a good memory (or look back earlier in this chapter), you'll remember we tried this when demonstrating the loan assistant project. In that earlier run, we obtained a value of 59 monthly payments. Is this a mistake? No – you'll see why next.

# Code Design – Loan Analysis

Another desired feature of the consumer loan assistant project is to provide an analysis of the loan, once computations are done. The information this analysis should include is:

- Loan Balance
- Interest Rate
- Number of Payments
- Amount of Each Payment
- Total of Payments Made
- Total Interest Paid

Such information is very useful in analyzing how effective and economical a loan payoff plan is. Our form has a text box control (**txtAnalysis**) available to provide these results. The analysis is generated after **btnCompute** is clicked and the number of payments or payment amount have been computed.

At first, generating a loan analysis seems like a simple task. The balance (**balance**) and interest rate (**interest**) are input numbers. The number of payments (**months**) and monthly payment (**payment**) are either input or computed. So, it seems the total of payments would be given by:

**totalPayments = months \* payment;**

while the interest paid would be:

**interestPaid = totalPayments – balance;**

The second equation is correct (assuming **totalPayments** is correct). But, the first equation (for **totalPayments**) doesn't quite apply. It's not that simple.

The code used for computing the payment amount (**computePayment** is **true**) and the number of payments (**computePayment** is **false**) is not exact. Truncation errors (making sure payments only have two decimal places) can affect the final

payment amount. And, forcing the number of payments to be an integer value can result in significant errors in the final payment, perhaps even necessitating a final payment (remember the example we just ran?). We need to develop an analysis that recognizes the possibility of such errors and make necessary adjustments.

Here's the approach we will take. If the loan has N payments of P dollars, we will process all but the last payment and see what the remaining balance is at that point. If that balance is less than P, that will become the final payment. If that balance is greater than P, a payment of P will be applied and an additional payment of the final balance will be created. The displayed loan analysis will then show the final payment and the associated total of payments and interest paid.

For those of you who have avoided all the mathematical derivations up to this point, you need to know how to process a single payment to reduce the loan balance. If B is the current loan balance, i the monthly interest rate and P the payment. The balance ( $B_{\text{after}}$ ) after the payment is:

$$B_{\text{after}} = B + Bi - P$$

This equation simply says the new balance is the old balance incremented by interest owed ( $Bi$ ), then decreased by the payment amount ( $P$ ). We use this equation to compute the final payment in the loan analysis.

The steps behind generating the loan analysis are:

- Display **balance**.
- Display **interest**.
- Compute **finalPayment** (adding a payment, if necessary).
- Compute and display total of payments.
- Compute and display interest paid.
- Disable **btnCompute**.
- Enable **btnNewLoan**.
- Set focus on **btnNewLoan**.

Each of these steps is performed in the **btnCompute Click** event. The modified

method is (changes are shaded):

```
private void btnCompute_Click(object sender, EventArgs e)
{
    double balance, interest, payment;
    int months;
    double monthlyInterest, multiplier;
    double loanBalance, finalPayment;

    balance = Convert.ToDouble(txtBalance.Text);
    interest = Convert.ToDouble(txtInterest.Text);
    monthlyInterest = interest / 1200;
    if (computePayment)
    {
        // Compute loan payment
        months = Convert.ToInt32(txtMonths.Text);
        multiplier = Math.Pow(1 + monthlyInterest, months);
        payment = balance * monthlyInterest * multiplier / (multiplier -
1);
        txtPayment.Text = String.Format("{0:f2}", payment);
    }
    else
    {
        // Compute number of payments
        payment = Convert.ToDouble(txtPayment.Text);
        months = (int)((Math.Log(payment) - Math.Log(payment -
balance * monthlyInterest)) / Math.Log(1 + monthlyInterest));
        txtMonths.Text = months.ToString();
    }
    // reset payment prior to analysis to fix at two decimals
    payment = Convert.ToDouble(txtPayment.Text);
    // show analysis
    txtAnalysis.Text = "Loan Balance: $" + String.Format("{0:f2}",

```

```

balance);

txtAnalysis.Text += "\r\n" + "Interest Rate: " + String.Format(
{0:f2}, interest) + "%";

// process all but last payment
loanBalance = balance;

for (int paymentNumber = 1; paymentNumber <= months - 1;
paymentNumber++)
{
    loanBalance += loanBalance * monthlyInterest - payment;
}

// find final payment
finalPayment = loanBalance;
if (finalPayment > payment)
{
    // apply one more payment
    loanBalance += loanBalance * monthlyInterest - payment;
    finalPayment = loanBalance;
    months++;
    txtMonths.Text = months.ToString();
}

txtAnalysis.Text += "\r\n\r\n" + Convert.ToString(months - 1) +
Payments of $" + String.Format("{0:f2}", payment);

txtAnalysis.Text += "\r\n" + "Final Payment of: $" +
String.Format("{0:f2}", finalPayment);

txtAnalysis.Text += "\r\n" + "Total Payments: $" + String.Format(
{0:f2}, (months - 1) * payment + finalPayment);

txtAnalysis.Text += "\r\n" + "Interest Paid $" + String.Format(
{0:f2}, (months - 1) * payment + finalPayment - balance);

btnCompute.Enabled = false;
btnNewLoan.Enabled = true;
btnNewLoan.Focus();

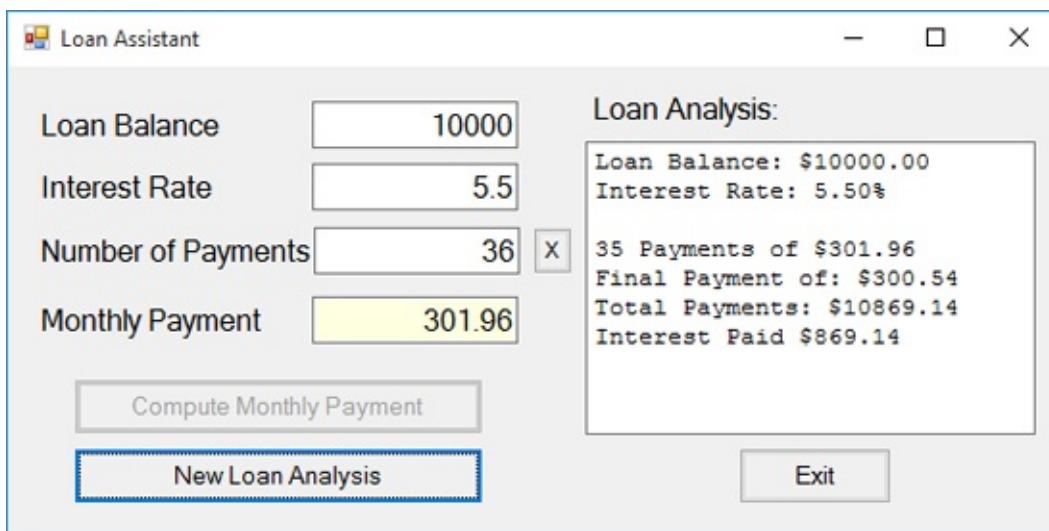
}

```

You should be able to identify all the steps of the loan analysis, especially the final payment adjustment.

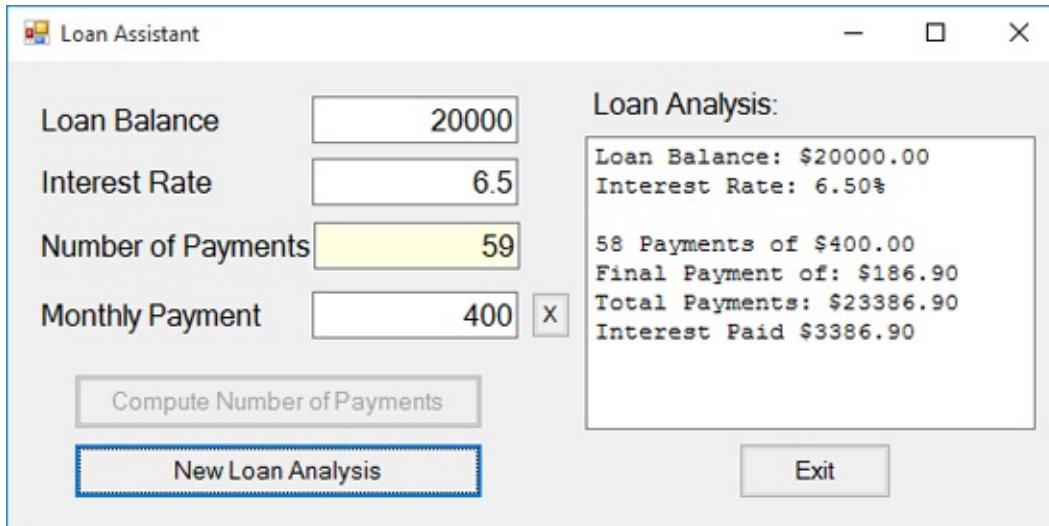
A couple of comments. In the first line of the analysis code, we reassign the **payment** value to the displayed value in the **txtPayment** text box. The displayed value is formatted to two decimal places. Through this reassignment, we make sure **payment** is just two decimal places. Second, note the analysis in the text box is essentially just one long **Text** property. To start a new line, we use the control string **\r\n** which stands for a carriage return, new line (a throwback to typewriter days).

Save and run the project. Enter values for balance, interest and number of payments. Click **Compute Monthly Payment**. Here are the results for the example I've been using:



Note the slight adjustment to the final payment amount. Note the focus on **New Loan Analysis**. You can't do another analysis at this point since **btnCompute** is disabled – we'll fix that in the next section. Click **Exit**.

Run the project again, this time clicking the X next to the **Number of Payments** text box. Enter values for balance, interest and payment, then click **Compute Number of Payments**. Continuing with the example I've been using (remember we got 58 payments before?) shows:



We now get 59 rather than 58 payments, the same result we saw earlier in the chapter. It was determined that once 58 payments of \$400.00 per month were applied, there was still a balance over \$400, necessitating a 59<sup>th</sup> payment of \$400.00 plus an additional payment of \$186.90. Click **Exit**, since you can't do anything else at this point.

# Code Design – New Loan Analysis

Following an analysis, we would like the capability of performing a new analysis. When a user clicks the **New Loan Analysis** button (**btnNewLoan**), the following things should happen:

- If computing payment, clear **txtPayment**, else clear **txtMonths**.
- Clear **txtAnalysis**.
- Enable **btnCompute**.
- Disable **btnNewLoan**.
- Set focus on **txtBalance**.

We do not clear the **txtBalance** or **txtInterest** boxes. If computing the payment, we do not clear the **txtMonths** box. If computing the number of months, we do not clear the **txtPayment** box. This allows a user to try different things with a specific loan. Individual boxes can be cleared by the user, if desired.

The **btnNewLoan Click** event is:

```
private void btnNewLoan_Click(object sender, EventArgs e)
{
    // clear computed value and analysis
    if (computePayment)
    {
        txtPayment.Text = "";
    }
    else
    {
        txtMonths.Text = "";
    }
    txtAnalysis.Text = "";
    btnCompute.Enabled = true;
    btnNewLoan.Enabled = false;
```

```
txtBalance.Focus();  
}
```

Enter this code into the code window.

Save and run the project. The project should now have total ability to compute monthly payments or number of payments, providing complete loan analysis results. Play with the project as much as you'd like.

# Improving a Visual C# Project

The consumer loan assistant project works fine in its current configuration, but there are some hidden problems. You may have uncovered some of them already. Earlier, we saw the possibility of unpredictable tab ordering and fixed the problem, improving the performance of our project. This is something you, as a programmer, will do a lot. You will build a project and, while running it and testing it, will uncover weaknesses that need to be eliminated. These weaknesses could be actual errors in the application or just things that, if eliminated, make your application easier to use. Some weaknesses are easy to find, some more subtle.

You will find, as you progress as a programmer, that you will spend much of your time improving your projects. You will always find ways to add features to a project and to make it more appealing to your user base. You should never be satisfied with your first solution to a problem. There will always be room for improvement. And Visual C# provides a perfect platform for adding improvements to an application. You can easily add features and test them to see if the desired performance enhancements are attained.

If you run the loan assistant project a few more times, you can identify some weaknesses:

- For example, what happens if you input a zero interest? The program will stop with an error message because the formulas implemented in code will not work with zero interest.
- Notice you can type any characters you want in the text boxes when you should just be limited to numbers and a single decimal point – any other characters will cause the program to work incorrectly.
- As a convenience, it would be nice that when you hit the <Enter> key after typing a number, the focus would move to the next control in the tab sequence.
- What happens if you forget to input a value (leaving a text box empty)? You could get unpredictable results.
- A subtle problem arises when using the ‘compute number of months’ mode. In this configuration, the minimum desired payment must exceed

the loan balance times the monthly interest. If it doesn't, you will never get the loan paid off – your balance will just keep growing (!), something called negative amortization.

We can (and will) address each of these points as we improve the loan assistant project. As we do, we'll need to look at some other Visual C# features.

# Code Design – Zero Interest

If you are lucky enough to find a bank or someone to give you a loan at zero percent interest, congratulations!! However, you can't use the current code to compute payment information. Try it if you like – you'll receive an error message.

The formulas used in the code assume a non-zero interest rate. If **interest** is zero, we can use much simpler formulas. For the ‘compute payment’ mode, the code is simply:

```
payment = balance / months;
```

While for the ‘compute number of payments’ mode, the code is:

```
months = (int)(balance / payment);
```

The modified **btnCompute Click** event method (changes are shaded, some unmodified code is not shown for brevity):

```
private void btnCompute_Click(object sender, EventArgs e)
{
    .
    .
    if (computePayment)
    {
        // Compute loan payment
        months = Convert.ToInt32(txtMonths.Text);
        if (interest == 0)
        {
            payment = balance / months;
        }
        else
```

```

    {
        multiplier = Math.Pow(1 + monthlyInterest, months);
        payment = balance * monthlyInterest * multiplier / (multiplier -
1);
    }
    txtPayment.Text = String.Format("{0:f2}", payment);
}
else
{
    // Compute number of payments
    payment = Convert.ToDouble(txtPayment.Text);
    if (interest == 0)
    {
        months = (int)(balance / payment);
    }
    else
    {
        months = (int)((Math.Log(payment) - Math.Log(payment -
balance * monthlyInterest)) / Math.Log(1 + monthlyInterest));
    }
    txtMonths.Text = months.ToString();
}
.
.
}

```

Save and run the application, making sure zero interest works under each mode. Notice adjustments to the final payment are only made when the balance is not an exact multiple of the payment. Make sure the non-zero interest options still work, too. Always make sure when you make changes to your code that you haven't disturbed portions that are working satisfactorily.

# Key Trapping

When entering values in the loan assistant project text boxes, there is nothing to prevent the user from typing in meaningless (non-numerical) characters. We want to keep this from happening. Whenever getting input from a user using a text box control, we want to limit the available keys they can press. This process of intercepting and eliminating unacceptable keystrokes is called **key trapping**.

Key trapping is done in the **KeyPress** event method of a text box control. Such a method has the form (for a text box named **txtText**):

```
private void txtText_KeyPress(object sender,  
System.Windows.Forms.KeyPressEventArgs e)  
{  
}  
}
```

What happens in this method is that every time a key is pressed in the corresponding text box, the **KeyPressEventArgs** class passes the key that has been pressed into the method via the **char** type **e.KeyChar** property. Recall the **char** type is used to represent a single character. We can thus examine this key. If it is an acceptable key, we set the **e.Handled** property to **false**. This tells Visual C# .NET that this method has not been handled and the KeyPress should be allowed. If an unacceptable key is detected, we set **e.Handled** to **true**. This ‘tricks’ Visual C# .NET into thinking the KeyPress event has already been handled and the pressed key is ignored.

We need someone way of distinguishing what keys are pressed. The usual alphabetic, numeric and character keys are fairly simple to detect. To help detect non-readable keys, we can examine the key’s corresponding Unicode value. Two values we will use are:

Definition	Value
Backspace	8
Carriage return (<Enter> key)	13

As an example, let's build a snippet of code to use for our loan assistant application example. We'll work with the **txtBalance** control, knowing the KeyPress events for the other text box controls will be similar. There are several ways to build a key trapping routine. I suggest an if/else structure that, based on different values of **e.KeyChar**, takes different steps. If **e.KeyChar** represents a number, a decimal point or a backspace key (always include backspace or the user won't be able to edit the text box properly), we will allow the keypress (**e.Handled = false**). Otherwise, we will set **e.Handled = true** to ignore the keypress. The code to do this is:

```
if (e.KeyChar >= '0' && e.KeyChar <= '9')
{
    // number values
    e.Handled = false;
}
else if ((int) e.KeyChar == 8)
{
    // backspace
    e.Handled = false;
}
else if (e.KeyChar == '.')
{
    // decimal point
    e.Handled = false;
}
else
{
    // any other character
    e.Handled = true;
}
```

Note the use of single quotes to signify **char** types, the same type as **e.KeyChar**.

Note, in the code above, there is nothing to keep the user from typing multiple

decimal points. To solve this, we let the ‘decimal point’ have its own case. To check if a decimal point already exists, we use the **IndexOf** method. The format for such a method is:

```
string1.IndexOf(string2)
```

If **string2** is found in **string1**, a non-zero value the starting position of **string2** is returned (the first character in **string1** is numbered 0). If -1 is returned, **string2** is not found in **string1**. So the snippet of code for the decimal point case is:

```
else if (e.KeyChar == '.')
{
    // decimal point
    if (txtBalance.Text.IndexOf(".") == -1)
    {
        // no decimal yet
        e.Handled = false;
    }
    else
    {
        // has decimal
        e.Handled = true;
    }
}
```

In this new case, if there is no decimal point (**IndexOf** returns a negative one), the key press is allowed. If a decimal point is already there (**IndexOf** returns a nonnegative value), the key press is disallowed.

Once the user types in a **Loan Balance**, it would be nice if focus would be moved to the **Interest Rate** text box if the user presses <**Enter**>. We can do that with another ‘else if’ in our KeyPress event. The **Case** to do this in our **txtBalance** text box key trapping routine would be:

```
else if ((int) e.KeyChar == 13)
```

```
{  
    // enter key – move to next box  
    txtInterest.Focus();  
    e.Handled = false;  
}
```

Here, if the <Enter> key is pressed (a UniCode value of 13), focus is shifted to the **txtInterest** text box control. We'll now add complete key trapping capabilities to our loan project.

# Code Design – Key Trapping

Each text box control used for input requires a key trapping routine in its **KeyPress** event. Using the snippets just developed as a guideline, here is the **txtBalance\_KeyPress** event:

```
private void txtBalance_KeyPress(object sender, KeyPressEventArgs e)
{
    // only allow numbers, backspace, decimal point, enter
    if (e.KeyChar >= '0' && e.KeyChar <= '9')
    {
        e.Handled = false;
    }
    else if ((int)e.KeyChar == 8)
    {
        e.Handled = false;
    }
    else if (e.KeyChar == '.')
    {
        if (txtBalance.Text.IndexOf(".") == -1)
        {
            e.Handled = false;
        }
        else
        {
            e.Handled = true;
        }
    }
    else if ((int)e.KeyChar == 13)
    {
        txtInterest.Focus();
    }
}
```

```

        e.Handled = false;
    }
    else
    {
        e.Handled = true;
    }
}

```

In this code, the user can enter any number key, a single decimal point, the backspace key or the <Enter> key. If <Enter> is pressed, the focus is transferred to the **Interest Rate** text box (**txtInterest**).

The code for the **txtInterest\_KeyPress** event is based on the routine just presented:

```

private void txtInterest_KeyPress(object sender, KeyPressEventArgs e)
{
    // only allow numbers, backspace, decimal point, enter
    if (e.KeyChar >= '0' && e.KeyChar <= '9')
    {
        e.Handled = false;
    }
    else if ((int)e.KeyChar == 8)
    {
        e.Handled = false;
    }
    else if (e.KeyChar == '.')
    {
        if (txtInterest.Text.IndexOf(".") == -1)
        {
            e.Handled = false;
        }
        else
    }
}

```

```

    {
        e.Handled = true;
    }
}

else if ((int)e.KeyChar == 13)
{
    if (computePayment)
    {
        txtMonths.Focus();
    }
    else
    {
        txtPayment.Focus();
    }
    e.Handled = false;
}
else
{
    e.Handled = true;
}
}

```

In this code, the user can enter any number key, a single decimal point, the backspace key or the <Enter> key. If <Enter> is pressed, the focus is transferred to the **Number of Months** text box (**txtMonths**), if computing the payment. Focus is transferred to the **Monthly Payment** text box (**txtPayment**), if computing the number of months.

The code for the **txtMonths\_KeyPress** event is:

```

private void txtMonths_KeyPress(object sender, KeyPressEventArgs e)
{
    // only allow numbers, backspace, enter
}

```

```

if (e.KeyChar >= '0' && e.KeyChar <= '9')
{
    e.Handled = false;
}
else if ((int)e.KeyChar == 8)
{
    e.Handled = false;
}
else if ((int)e.KeyChar == 13)
{
    btnCompute.Focus();
    e.Handled = false;
}
else
{
    e.Handled = true;
}
}

```

In this code, the user can enter any number key, the backspace key or the <Enter> key. No decimal point is allowed since the number of payments must be an integer number. If <Enter> is pressed, the focus is transferred to the **btnCompute** button control.

Lastly, the code for the **txtPayment\_KeyPress** event is:

```

private void txtPayment_KeyPress(object sender, KeyPressEventArgs e)
{
    // only allow numbers, backspace, decimal point, enter
    if (e.KeyChar >= '0' && e.KeyChar <= '9')
    {
        e.Handled = false;
    }
    else if ((int)e.KeyChar == 8)

```

```

{
    e.Handled = false;
}
else if (e.KeyChar == '.')
{
    if (txtPayment.Text.IndexOf(".") == -1)
    {
        e.Handled = false;
    }
    else
    {
        e.Handled = true;
    }
}
else if ((int)e.KeyChar == 13)
{
    btnCompute.Focus();
    e.Handled = false;
}
else
{
    e.Handled = true;
}
}

```

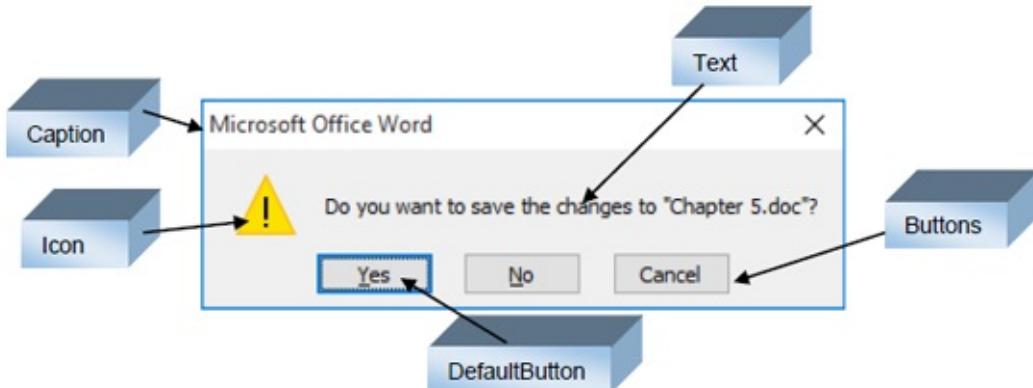
In this code, the user can enter any number key, a single decimal point, the backspace key or the <Enter> key. If <Enter> is pressed, the focus is transferred to the **btnCompute** button control.

Once all the **KeyPress** events have been added to the project, save it and run it. Make sure the key trapping works as it should. Make sure the focus moves to the proper control when <Enter> is pressed.

# Message Box Dialog

If one of the text box controls (that should have an entry) is empty, we need to inform the user of such an error and give him/her a chance to correct it. Similarly, when using the ‘compute number of months’ mode, if the user enters a payment that is too low, we need to ask the user if they’d like to use the minimum payment instead. We can do both of these tasks using the **MessageBox** function, a widely used Visual C# function. This function lets you display messages to your user and, optionally, receive feedback for further information. It can be used to display error messages, describe potential problems or just to show the result of some computation. The **MessageBox** function is versatile, with the ability to display any message, an optional icon, and a selected set of buttons. The user responds by clicking a button in the message box

You've seen message boxes if you've ever used a Windows application. Think of all the examples you've seen. For example, message boxes are used to ask you if you wish to save a file before exiting and to warn you if a disk drive is not ready. For example, while writing these notes in Microsoft Word, if I attempt to exit, I see this message box:



In this message box, the different parts that you control have been labeled. You will see how you can format a message box any way you desire.

To use the **MessageBox** function, you decide what the **Text** of the message should be, what **Caption** you desire, what **Icon** and **Buttons** are appropriate, and which **DefaultButton** you want. To display the message box in code, you use the **MessageBox Show** method.

There are several ways to implement the MessageBox function **Show** method. Some of the more common ways are:

**MessageBox.Show(Text);**  
**MessageBox.Show(Text, Caption);**  
**MessageBox.Show(Text, Caption, Buttons);**  
**MessageBox.Show(Text, Caption, Buttons, Icon);**  
**MessageBox.Show(Text, Caption, Buttons, Icon, DefaultButton);**

In these implementations, if **DefaultButton** is omitted, the first button is default. If **Icon** is omitted, no icon is displayed. If **Buttons** is omitted, an ‘OK’ button is displayed. And, if **Caption** is omitted, no caption is displayed.

You decide what you want for the message box **Text** and **Caption** information (**string** data types). The other arguments are defined by Visual C# predefined constants. The **Buttons** constants are defined by the **MessageBoxButtons** constants:

<b>Member</b>	<b>Description</b>
AbortRetryIgnore	Displays Abort, Retry and Ignore buttons
OK	Displays an OK button
OKCancel	Displays OK and Cancel buttons
RetryCancel	Displays Retry and Cancel buttons
YesNo	Displays Yes and No buttons
YesNoCancel	Displays Yes, No and Cancel buttons

The syntax for specifying a choice of buttons is the usual dot-notation:

### **MessageBoxButtons.Member**

So, to display an **OK** and **Cancel** button, the constant is:

### **MessageBoxButtons.OKCancel**

You don’t have to remember this, however. When typing the code, the Intellisense feature will provide a drop-down list of button choices when you

reach that argument! Again, like magic! This will happen for all the arguments in the **MessageBox** function.

The displayed **Icon** is established by the **MessageBoxIcon** constants:

<b>Member</b>	<b>Description</b>
IconAsterisk	Displays an information icon
IconInformation	Displays an information icon
IconError	Displays an error icon (white X in red circle)
IconHand	Displays an error icon
IconNone	Display no icon
IconStop	Displays an error icon
IconExclamation	Displays an exclamation point icon
IconWarning	Displays an exclamation point icon
IconQuestion	Displays a question mark icon

To specify an icon, the syntax is:

### **MessageBoxIcon.Member**

Note there are nine different members of the **MessageBoxIcon** constants, but only four icons (information, error, exclamation, question) available. This is because the current Windows operating system only offers four icons. Future implementations may offer more.

When a message box is displayed, one of the displayed buttons will have focus or be the default button. If the user presses <Enter>, this button is selected. You specify which button is default using the **MessageBoxDefaultButton** constants:

<b>Member</b>	<b>Description</b>
Button1	First button in message box is default
Button2	Second button in message box is default
Button3	Third button in message box is default

To specify a default button, the syntax is:

## **MessageBoxDefaultButton.Member**

The specified default button is relative to the displayed buttons, left to right. So, if you have **Yes**, **No** and **Cancel** buttons displayed and the second button is selected as default, the **No** button will have focus (be default).

When you invoke the **Show** method of the **MessageBox** function, the function returns a value from the **DialogResult** constants. The available members are:

<b>Member</b>	<b>Description</b>
Abort	The Abort button was selected
Cancel	The Cancel button was selected
Ignore	The Ignore button was selected
No	The No button was selected
OK	The OK button was selected
Retry	The Retry button was selected
Yes	The Yes button was selected

### **Message Box Example:**

This little code snippet (the first line is very long):

```
if (MessageBox.Show("This is an example of a message box", "Message  
Box Example", MessageBoxButtons.OKCancel,  
MessageBoxIcon.Information, MessageBoxDefaultButton.Button1) ==  
DialogResult.OK)  
{  
    'everything is OK  
}  
else  
{  
    'cancel was pressed  
}
```

displays this message box:



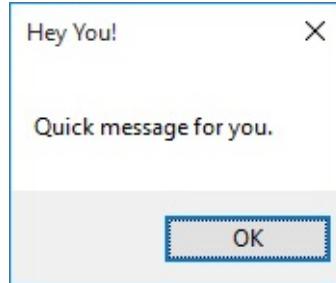
Of course, you would need to add code for the different tasks depending on whether **OK** or **Cancel** is clicked by the user.

### Another Message Box Example:

Many times, you just want to display a quick message to the user with no need for feedback (just an **OK** button). This code does the job:

```
MessageBox.Show("Quick message for you.", "Hey You!");
```

The resulting message box:



Notice there is no icon and the **OK** button (default if no button specified) is shown. Also, notice in the code, there is no need to read the returned value – we know what it is! You will find a lot of uses for this simple form of the message box (with perhaps some kind of icon) as you progress in Visual C#.

# Code Design – Input Validation

The code to check each text box for no entry is similar. The steps are:

- Check to see if **Text** is blank.
- If **Text** is not blank, continue as usual, knowing the text box contains only valid entries (since we are using key trapping).
- If **Text** is blank, display message box telling user so, return focus to text box control, allowing another entry.

All modifications for checking for blank text boxes go in the **btnCompute Click** event. The modified method (changes are shaded, with some unmodified code not shown) that implements each of the above steps for each text box control is:

```
private void btnCompute_Click(object sender, EventArgs e)
{
    double balance, interest, payment;
    int months;
    double monthlyInterest, multiplier;
    double loanBalance, finalPayment;
    if (txtBalance.Text != "")
    {
        balance = Convert.ToDouble(txtBalance.Text);
    }
    else
    {
        MessageBox.Show("Must enter a Loan Balance value.", "Input Error", MessageBoxButtons.OK, MessageBoxIcon.Information);
        txtBalance.Focus();
        return;
    }
    if (txtInterest.Text != "")
```

```
{  
    interest = Convert.ToDouble(txtInterest.Text);  
}  
else  
{  
    MessageBox.Show("Must enter an Interest Rate value.", "Input  
Error", MessageBoxButtons.OK, MessageBoxIcon.Information);  
    txtInterest.Focus();  
    return;  
}  
  
monthlyInterest = interest / 1200;  
if (computePayment)  
{  
    // Compute loan payment  
    if (txtMonths.Text != "")  
    {  
        months = Convert.ToInt32(txtMonths.Text);  
    }  
    else  
    {  
        MessageBox.Show("Must enter a Number of Payments  
value.", "Input Error", MessageBoxButtons.OK,  
MessageBoxIcon.Information);  
        txtMonths.Focus();  
        return;  
    }  
    if (interest == 0)  
    {  
        payment = balance / months;  
    }  
    else  
    {
```

```

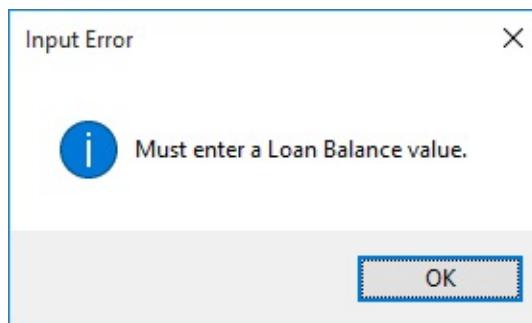
        multiplier = Math.Pow(1 + monthlyInterest, months);
        payment = balance * monthlyInterest * multiplier / (multiplier
- 1);
    }
    txtPayment.Text = String.Format("{0:f2}", payment);
}
else
{
    // Compute number of payments
    if (txtPayment.Text != "")
    {
        payment = Convert.ToDouble(txtPayment.Text);
    }
    else
    {
        MessageBox.Show("Must enter a Monthly Payment value.",
"Input Error", MessageBoxButtons.OK, MessageBoxIcon.Information);
        txtPayment.Focus();
        return;
    }
    if (interest == 0)
    {
        months = (int)(balance / payment);
    }
    else
    {
        months = (int)((Math.Log(payment) - Math.Log(payment -
balance * monthlyInterest)) / Math.Log(1 + monthlyInterest));
    }
    txtMonths.Text = months.ToString();
}
.

```

}

After making these modifications, save and run the project. Make sure each input validation works correctly. Make sure it works in both computation modes. And, make sure your changes have not affected previously correct calculations.

Each validation is similar. If the text box is not blank, things proceed as usual. If blank, a message box like this appears (this one appears when **txtBalance** is left blank, other message boxes are similar):



The user clicks **OK**, the focus is returned to the blank control for another chance at inputting a non-blank value.

We have one last input validation to implement and then the consumer loan assistant project is complete (unless you can think of other improvements). Recall, when computing the number of months, we must enter a minimum payment or the balance will continue to grow. The minimum payment is the loan balance times the monthly interest:

**minimumPayment = balance \* monthlyInterest;**

If this payment is made each month, it is called an “interest only” loan. This means, we just pay the interest owed each month, never decreasing the balance. Since our goal is to decrease the balance, we will suggest to the user a minimum payment at least \$1 greater than the interest only option, or we will use:

**minimumPayment = balance \* monthlyInterest + 1;**

The steps for minimum payment validation are (only needed when

**computePayment** is false):

- If entered **payment** is less than minimum value, display message box informing user of minimum needed and ask if they would like to use that value.
  - o If user responds **Yes**, set **payment**, display in **txtPayment** and continue.
  - o If user responds **No**, set focus on **txtPayment** to allow new entry.
- If entered **payment** is above minimum value, continue as usual.

These modifications go in the **btnCompute Click** event. While implementing improvements to the loan assistant project, we have made many modifications to this event. For reference purposes, here is the final version of the **btnCompute Click** method (with new additions shaded):

```
private void btnCompute_Click(object sender, EventArgs e)
{
    double balance, interest, payment;
    int months;
    double monthlyInterest, multiplier;
    double loanBalance, finalPayment;
    if (txtBalance.Text != "")
    {
        balance = Convert.ToDouble(txtBalance.Text);
    }
    else
    {
        MessageBox.Show("Must enter a Loan Balance value.", "Input
Error", MessageBoxButtons.OK, MessageBoxIcon.Information);
        txtBalance.Focus();
        return;
    }
    if (txtInterest.Text != "")
    {
```

```
interest = Convert.ToDouble(txtInterest.Text);
}
else
{
    MessageBox.Show("Must enter an Interest Rate value.", "Input
Error", MessageBoxButtons.OK, MessageBoxIcon.Information);
    txtInterest.Focus();
    return;
}
monthlyInterest = interest / 1200;
if (computePayment)
{
    // Compute loan payment
    if (txtMonths.Text != "")
    {
        months = Convert.ToInt32(txtMonths.Text);
    }
    else
    {
        MessageBox.Show("Must enter a Number of Payments
value.", "Input Error", MessageBoxButtons.OK,
MessageBoxIcon.Information);
        txtMonths.Focus();
        return;
    }
    if (interest == 0)
    {
        payment = balance / months;
    }
    else
    {
        multiplier = Math.Pow(1 + monthlyInterest, months);
```

```

        payment = balance * monthlyInterest * multiplier / (multiplier
- 1);
    }
    txtPayment.Text = String.Format("{0:f2}", payment);
}
else
{
    // Compute number of payments
    if (txtPayment.Text != "")
    {
        payment = Convert.ToDouble(txtPayment.Text);
        if (payment <= (balance * monthlyInterest + 1.0))
        {
            if (MessageBox.Show("Minimum payment must be $" +
String.Format("{0:f2}", (int)(balance * monthlyInterest + 1.0)) + "\r\n" +
"Do you want to use the minimum payment?", "Input Error",
MessageBoxButtons.YesNo, MessageBoxIcon.Question) ==
DialogResult.Yes)
            {
                txtPayment.Text =
String.Format("{0:f2}", (int)(balance * monthlyInterest + 1.0));
                payment = Convert.ToDouble(txtPayment.Text);
            }
            else
            {
                txtPayment.Focus();
                return;
            }
        }
    }
    else
    {
        MessageBox.Show("Must enter a Monthly Payment value.",
```

```

    "Input Error", MessageBoxButtons.OK, MessageBoxIcon.Information);
    txtPayment.Focus();
    return;
}
if (interest == 0)
{
    months = (int)(balance / payment);
}
else
{
    months = (int)((Math.Log(payment) - Math.Log(payment -
balance * monthlyInterest)) / Math.Log(1 + monthlyInterest));
}
txtMonths.Text = months.ToString();
}

// reset payment prior to analysis to fix at two decimals
payment = Convert.ToDouble(txtPayment.Text);
// show analysis
txtAnalysis.Text = "Loan Balance: $" + String.Format("{0:f2}",
balance);
txtAnalysis.Text += "\r\n" + "Interest Rate: " + String.Format(
{0:f2}", interest) + "%";
// process all but last payment
loanBalance = balance;
for (int paymentNumber = 1; paymentNumber <= months - 1;
paymentNumber++)
{
    loanBalance += loanBalance * monthlyInterest - payment;
}
// find final payment
finalPayment = loanBalance;
if (finalPayment > payment)
{

```

```

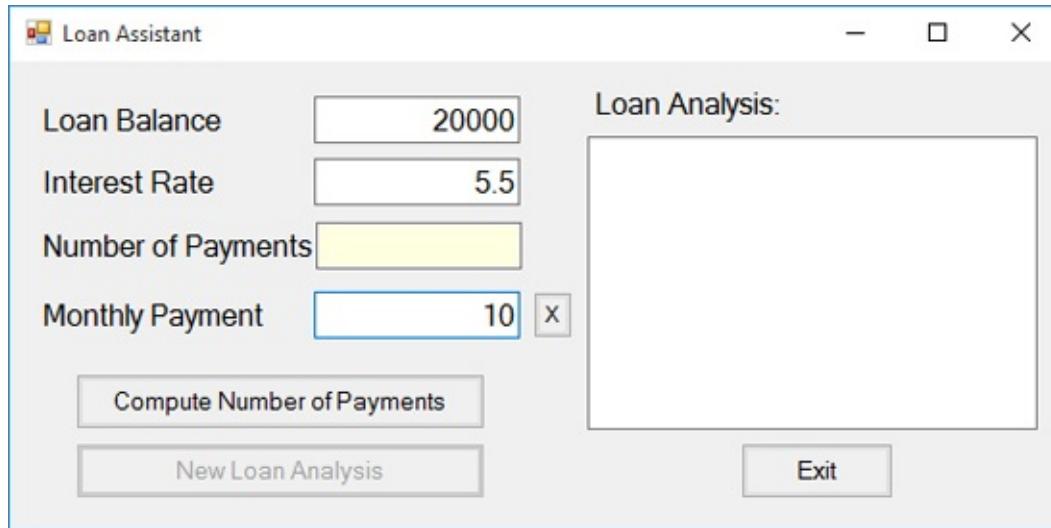
// apply one more payment
loanBalance += loanBalance * monthlyInterest - payment;
finalPayment = loanBalance;
months++;
txtMonths.Text = months.ToString();
}

txtAnalysis.Text += "\r\n\r\n" + Convert.ToString(months - 1) + "
Payments of $" + String.Format("{0:f2}", payment);
txtAnalysis.Text += "\r\n" + "Final Payment of: $" + String.Format(
{0:f2}", finalPayment);
txtAnalysis.Text += "\r\n" + "Total Payments: $" + String.Format(
{0:f2}", (months - 1) * payment + finalPayment);
txtAnalysis.Text += "\r\n" + "Interest Paid $" + String.Format(
{0:f2}", (months - 1) * payment + finalPayment - balance);
btnCompute.Enabled = false;
btnNewLoan.Enabled = true;
btnNewLoan.Focus();
}

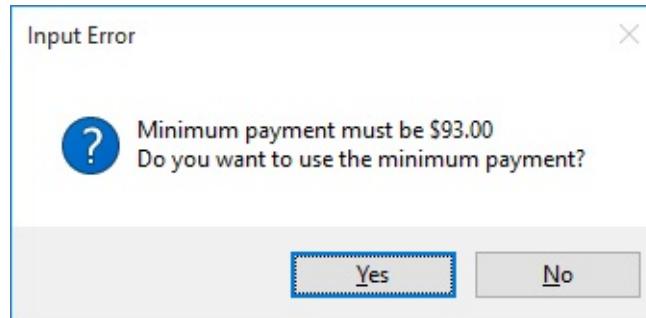
```

Make the noted changes.

Save and run the loan assistant project. Switch to ‘compute number of payments’ mode. Enter a **Loan Balance** and an **Interest Rate**. Enter a “too low” **Monthly Payment** amount. Here’s some numbers I used:

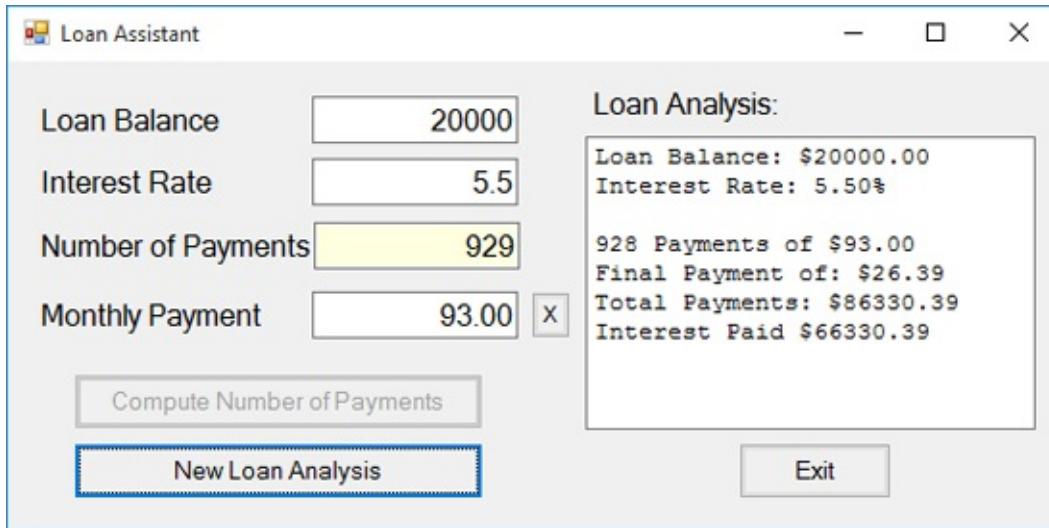


Now, click **Compute Number of Payments**. A message box like this should appear:



At this point, if you click **No**, you will be returned to the **Monthly Payment** text box for another chance.

Click **Yes** and analysis will proceed using the suggested minimum payment (\$92.00 in my example):



With this low minimum payment, it would take over 102 years to pay off the loan!! And, unfortunately, many credit card companies don't let you know how many years it takes to pay off a balance if you just make the minimum payment each month. This new project arms you with the tool you need to make such computations.

# Consumer Loan Assistant Project Review

The **Consumer Loan Assistant** project is now complete. Save and run the project and make sure it works as designed. Check that you can move back and forth between computation modes. Use the project to make informed payment decisions regarding any loans or credit cards you may have.

If there are errors in your implementation, go back over the steps of form and code design. Use the debugger when needed. Go over the developed code – make sure you understand how different parts of the project were coded. As mentioned in the beginning of this chapter, the completed project is saved as **Loan Assistant** in the **HomeVCS\HomeVCS Projects** folder.

While completing this project, new concepts and skills you should have gained include:

- Proper use of the text box control.
- How to use tab order and control focus.
- Different ways to improve a Visual C# project.
- How to do key trapping to eliminate unwanted keystrokes.
- How to use message boxes in conjunction with input validation.

This project also showed that once you have built a working project, there is often still a lot of work to do. Much of the code in the loan assistant project was added to improve the application – making it more user friendly and less susceptible to erroneous entries. As mentioned previously in these notes, it is relatively easy to write a project that works properly when the user does everything correctly. It's difficult and takes time to write a project that can handle all the possible wrong things a user can do and still not bomb out. Added improvements separate the good projects from the adequate projects.

# Consumer Loan Assistant Project Enhancements

Possible enhancements to the consumer loan assistant project include:

- Many times, you know how much you can afford monthly and want to know how much you can borrow. Add a capability to compute balance, given interest, months and payment. Follow similar steps for computing the other parameters.
- Add single payment processing capability so you can see how much the balance decreases each month and how much interest you are paying. Show results in the current text box or add other controls.
- Add the capability to stop after a certain number of payments have been processed.
- Add printing capability to see a complete repayment schedule for any loan you design. Printing is discussed in the final project in these notes – **Home Inventory**.
- Add an output to the loan analysis that tells you what date your loan will be paid off based on the number of monthly payments.

6

## Flash Card Math Quiz Project

# Review and Preview

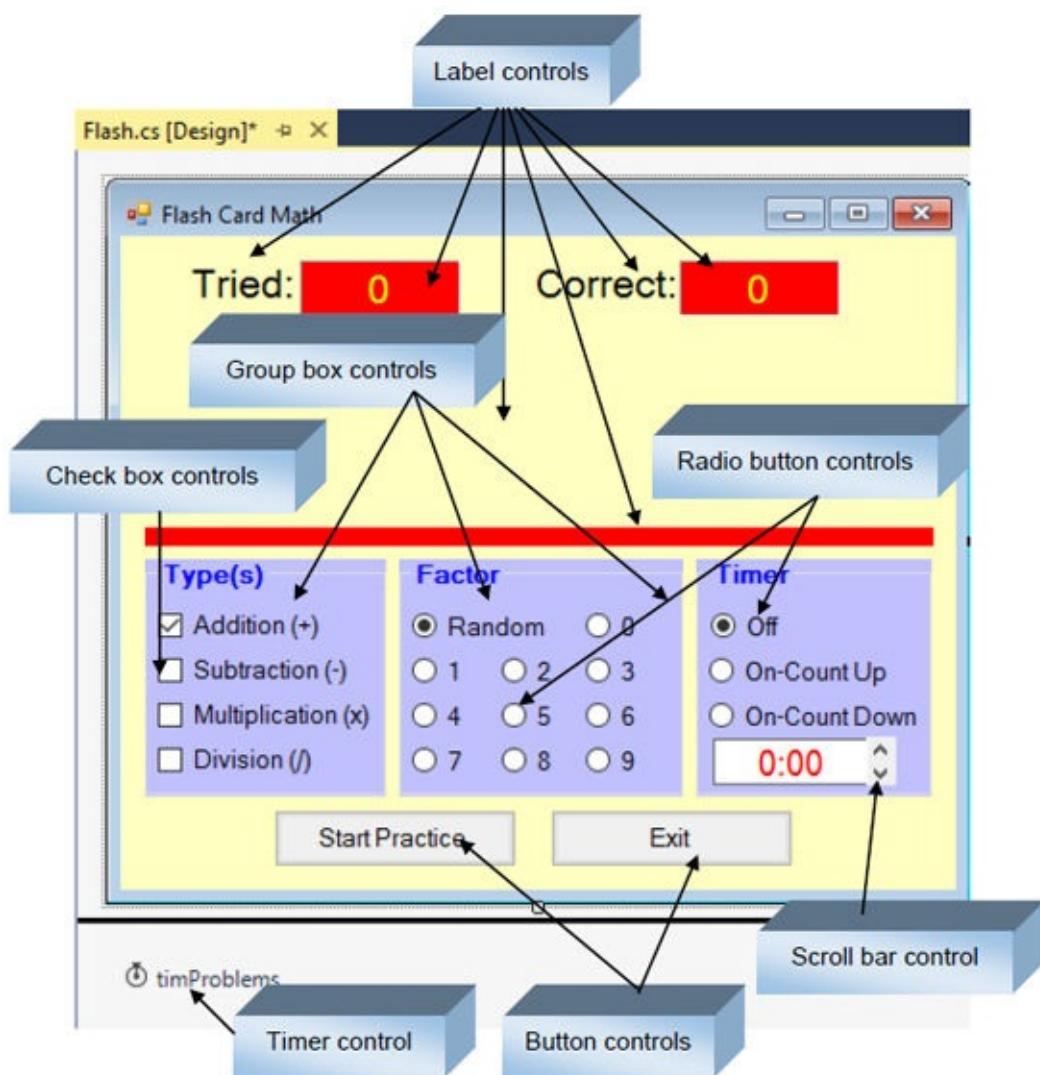
In this chapter, we build a project that lets kids (or adults) practice their basic addition, subtraction, multiplication and division skills. The **Flash Card Math Quiz Project** allows you to select problem type, what numbers you want to use and has three timing options.

Several new controls are introduced: the check box, the radio button, the group box and the scroll bar. We also look at using random numbers and how multiple events can invoke a single event method.

# Flash Card Math Quiz Project Preview

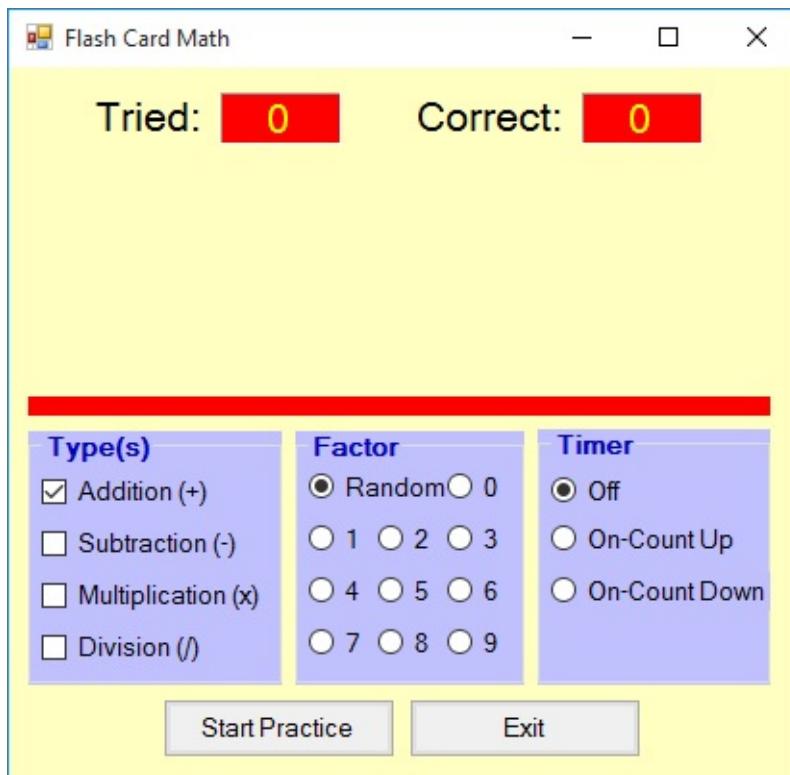
In this chapter, we will build a **flash card math** program. Random math problems (selectable from addition, subtraction, multiplication, and/or division) using the numbers from 0 to 9 are presented. Timing options are available to help build both accuracy and speed.

The finished project is saved as **Flash Card** in the **HomeVCS\HomeVCS Projects** folder. Start Visual C# and open the finished project. Open the form (double-click **Flash.cs** in Solution Explorer) and you will see:

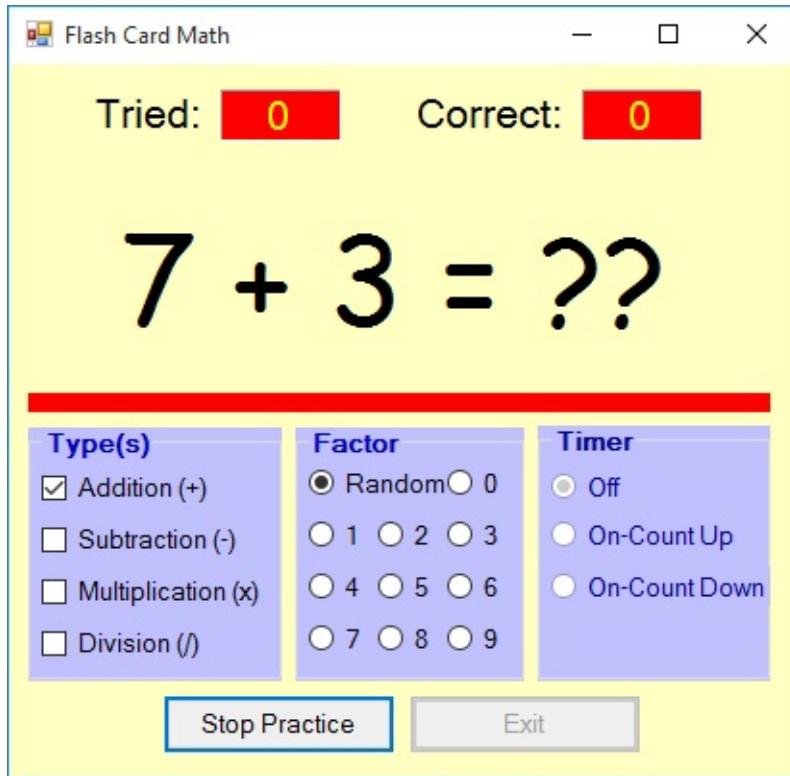


There are lots of controls here (several of which we've yet to see in building projects). Two label controls are used for title information, two for scoring. There's a large label in the middle of the form (there is nothing in it, so you can't see it) used to display the math problem. And, a final label (the skinny red box) is just used for "decoration." Two button controls are used to start and stop the problems and to exit the project. There are also three group box controls. The first holds four check box controls used to select problem type. The second holds eleven radio button controls used to select numbers used in the problems. The third group box holds three radio button controls used to select the timing option. A vertical scroll bar control (next to a label) is used to adjust the amount of time used in the flash card drills. A timer control is used for overall timing.

Run the project (press <F5>). The flash card math program will appear as:



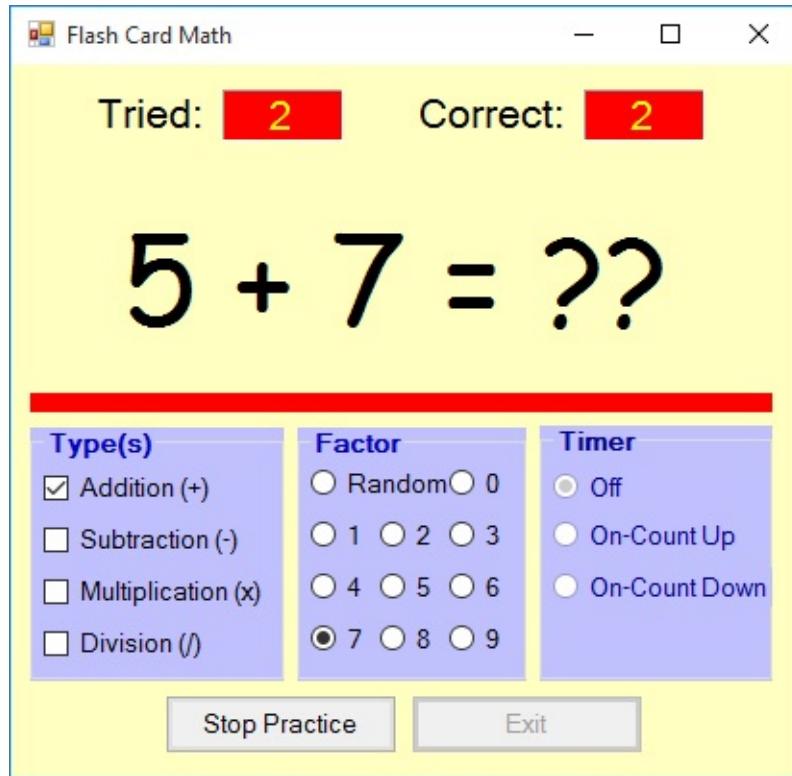
Many options are available. First, choose problem type from the **Type** group box. Choose from **Addition**, **Subtraction**, **Multiplication**, and/or **Division** problems (you may choose more than one problem type). Choose your **Factor**, any number from 0 to 9, or choose **Random** for random factors. These options may be changed at any time. To practice math facts, click on the **Start Practice** button. When I click **Start Practice** (using the default choices), I see:



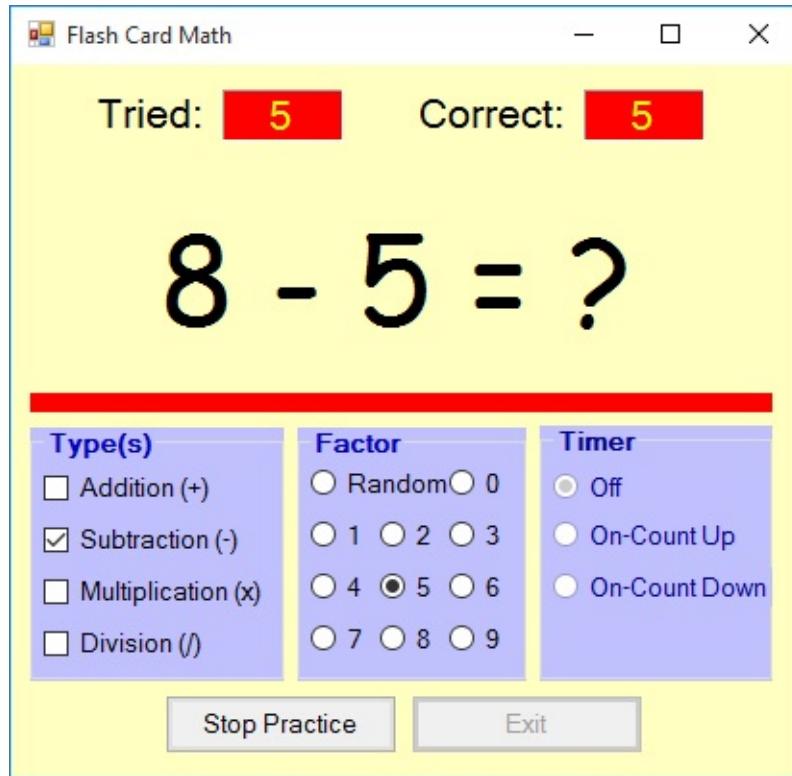
You can now see there is a large label control in the middle where the problem ( $7 + 3 =$ ) is displayed. The program is waiting for an answer to this problem. Type your answer. If it is correct, the number in the label control next to **Correct:** is incremented. Whether correct or not, another problem is presented.

A few notes on entering your answer. The primary goal of the program is to build speed in solving simple problems. As such, you have one chance to enter an answer - there is no erasing. If the answer has more than two digits (the number of digits in the answer is shown using question marks), type your answer from left to right. For example, if the answer is 10, type a 1 then a 0. Try several addition problems to see how answers are entered. You can stop practicing math problems, at any time, by clicking the **Stop Practice** button.

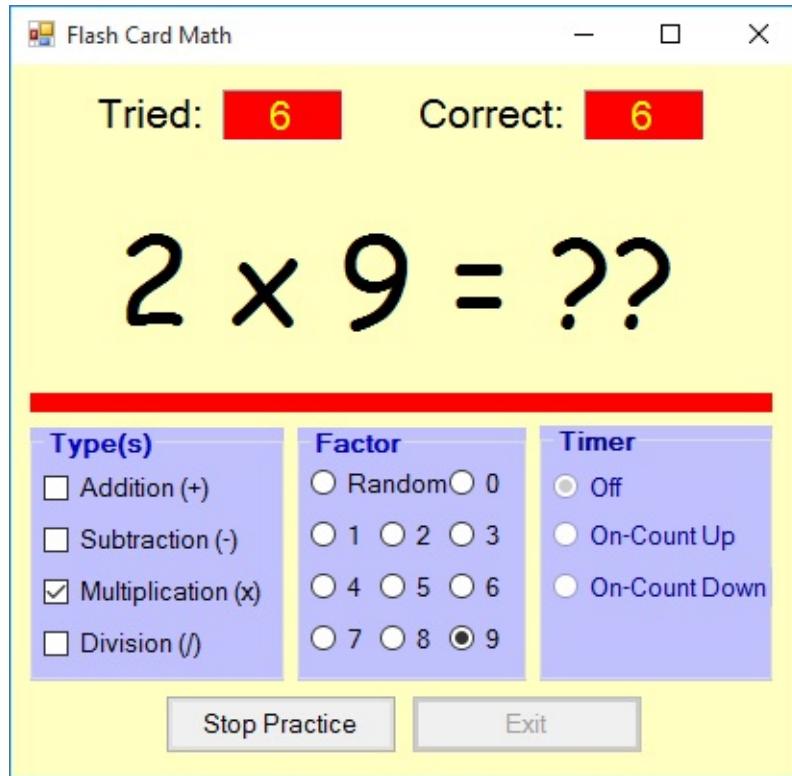
Other problem types can be selected and a new factor chosen at any time. Each problem is generated randomly, based on problem type and factor value. For **Addition**, you are given problems using your factor as the second addend. If you choose 7 as your factor, an example problem would be:



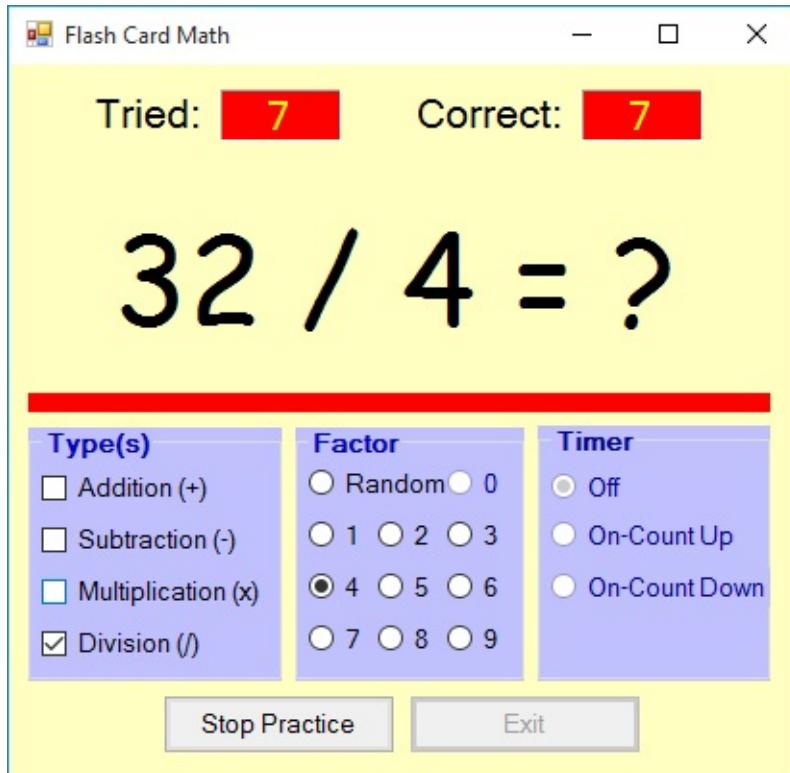
For **Subtraction**, you are given problems using your factor as the subtrahend (the number being subtracted). Selecting a factor of 5, an example subtraction problem is:



For **Multiplication**, you are given problems using your factor as the multiplier (the number you're multiplying by). If a factor of **9** is selected, an example multiplication problem is:



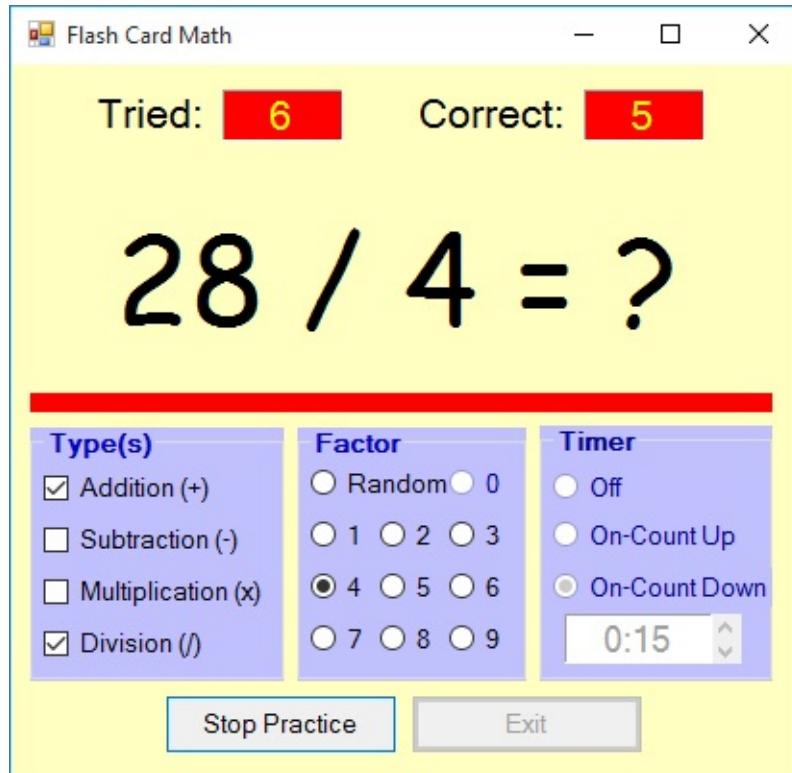
Lastly, for **Division**, you are given problems using your factor as the divisor (the number you are dividing by). If the selected factor is 4, a typical division problem would be:



As mentioned, you do not have to choose a specific factor – **Random** factors can be chosen. Try all kinds of factors with all kinds of problem types.

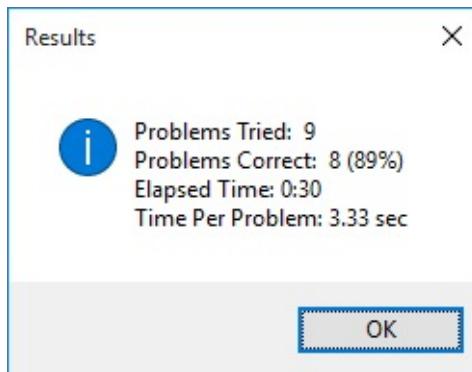
There is another option to consider when using the flash card math project – the corresponding option choices are in the **Timer** group box. These options can only be selected when not solving problems. There are three choices here. If you select **Timer Off**, you solve problems until you click **Stop Practice**. If you select **On-Count Up**, a timer will appear and the computer will keep track of how long you were solving problems (a maximum of 30 minutes is allowed). If you select **On-Count Down**, a timer will appear, along with a scroll bar control. The scroll bar control is used to set how long you want to solve problems (a maximum of 30 minutes is allowed). The timer will then count down, allowing you to solve problems until the allotted time expires.

Try the timer options if you'd like. Here's the beginning of a run I made using the **On-Count Down** option (starting at 1 minute):



Once you are done practicing math problems (either you clicked **Stop Practice** or time ran out with the **On-Count Down** option), a message box appears giving you the results of your little quiz. This box tells you how many problems you solved and how many you got correct (including a percentage score). If the timer was on, you are also told how long you were solving problems and how much time (on average) you spent on each problem.

Here's the message box I saw when I finished the quiz I started above:



Click **OK** and you can try again. Click the **Exit** button in **Flash Card Math** when you are done solving problems.

You will now build this project in several stages. We first address **form design**. We discuss the controls used to build the form and establish initial properties. And, we address **code design** in detail. We cover random generation of problems, selection of the various program options, including have multiple events access a single event method, and how to use timing.

There are several new controls in this project – the check box, the radio button, the group box and the scroll bar. We review these controls and their use before starting the project.

# CheckBox Control

**In Toolbox:**



**On Form (Default Properties):**



Visual C# features many ‘point and click’ controls that let the user make a choice simply by clicking with the mouse. These controls are attractive, familiar and minimize the possibility of errors in your application. We will see many such controls. The first, the **CheckBox** control, is examined here.

The **CheckBox** control provides a way to make choices from a list of potential candidates. Some, all, or none of the choices in a group may be selected. Check boxes are used in all Windows applications (even the Visual C# IDE). Examples of their use would be to turn options on and off in an application or to select from a ‘shopping’ list.

**CheckBox Properties:**

<b>Name</b>	Gets or sets the name of the check box (three letter prefix for check box name is <b>chk</b> ).
<b>BackColor</b>	Get or sets the check box background color.
<b>Checked</b>	Gets or sets a value indicating whether the check box is in the checked state.
<b>Font</b>	Gets or sets font name, style, size.
<b>ForeColor</b>	Gets or sets color of text or graphics.
<b>Text</b>	Gets or sets string displayed next to check box.
<b> TextAlign</b>	Gets or sets the alignment of text of the check box.

**CheckBox Methods:**

<b>Focus</b>	Moves focus to this check box.
--------------	--------------------------------

### CheckBox Events:

**CheckedChanged** Occurs when the value of the **Checked** property changes, whether in code or when a check box is clicked.

**Click** Triggered when a check box is clicked. **Checked** property is automatically changed by Visual C#.

When a check box is clicked, if there is no check mark there (**Checked** = **False**), Visual C# will place a check there and change the **Checked** property to **True**. If clicked and a check mark is there (**Checked** = **True**), then the check mark will disappear and the **Checked** property will be changed to **False**. The check box can also be configured to have three states: checked, unchecked or indeterminate. Consult on-line help if you require such behavior.

Typical use of **CheckBox** control:

- Set the **Name** and **Text** property. Initialize the **Checked** property.
- Monitor **Click** or **CheckChanged** event to determine when button is clicked. At any time, read **Checked** property to determine check box state.
- You may also want to change the **Font**, **Backcolor** and **Forecolor** properties.

# RadioButton Control

**In Toolbox:**



**On Form (Default Properties):**



**RadioButton** controls provide the capability to make a “mutually exclusive” choice among a group of potential candidate choices. This simply means, radio buttons work as a group, only one of which can be selected. Radio buttons are seen in all Windows applications. They are called radio buttons because they work like a tuner on a car radio – you can only listen to one station at a time! Examples for radio button groups would be twelve buttons for selection of a month in a year, a group of buttons to let you select a color or buttons to select the difficulty in a game.

A first point to consider is how do you define a ‘group’ of radio buttons that work together? Any radio buttons placed on the form will act as a group. That is, if any radio button on a form is ‘selected’, all other buttons will be automatically ‘unselected.’ What if you need to make two independent choices; that is, you need two independent groups of radio buttons? To do this requires one of two grouping controls in Visual C#: the **GroupBox** control or the **Panel** control. Radio buttons placed on either of these controls are independent from other radio buttons. We will discuss the group box control next – we’ll cover the panel control in a later chapter. For now, we’ll just be concerned with how to develop and use a single group of radio buttons.

**RadioButton Properties:**

<b>Name</b>	Gets or sets the name of the radio button (three letter prefix for radio button name is <b>rdo</b> ).
<b>BackColor</b>	Get or sets the radio button background color.
<b>Checked</b>	Gets or sets a value indicating whether the radio

	button is checked.
<b>Font</b>	Gets or sets font name, style, size.
<b>ForeColor</b>	Gets or sets color of text or graphics.
<b>TextAlign</b>	Gets or sets the alignment of text of the radio button.

### RadioButton Methods:

<b>Focus</b>	Moves focus to this radio button.
<b>PerformClick</b>	Generates a Click event for the button, simulating a click by a user.

### RadioButton Events:

<b>CheckedChanged</b>	Occurs when the value of the Checked property changes, whether in code or when a radio button is clicked.
<b>Click</b>	Triggered when a button is clicked. <b>Checked</b> property is automatically changed by Visual C#.

When a radio button is clicked, its Checked property is automatically set to True by Visual C#. And, all other radio buttons in that button's group will have a Checked property of False. Only one button in the group can be checked.

Typical use of **RadioButton** control:

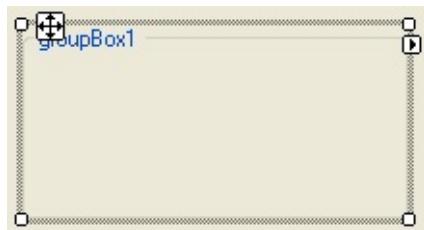
- Establish a group of radio buttons.
- For each button in the group, set the **Name** (give each button a similar name to identify them with the group) and **Text** property. You might also change the **Font**, **BackColor** and **Forecolor** properties.
- Initialize the **Checked** property of one button to **True**.
- Monitor the **Click** or **CheckChanged** event of each radio button in the group to determine when a button is clicked. The 'last clicked' button in the group will always have a **Checked** property of **True**.

# GroupBox Control

**In Toolbox:**



**On Form (Default Properties):**



We've seen that both radio buttons and check boxes usually work as a group. Many times, there are logical groupings of controls. For example, you may have a scroll device setting the value of a displayed number. The **GroupBox** control provides a convenient way of grouping related controls in a Visual C# application. And, in the case of radio buttons, group boxes affect how such buttons operate.

To place controls in a group box, you first draw the GroupBox control on the form. Then, the associated controls must be placed in the group box. This allows you to move the group box and controls together. There are several ways to place controls in a group box:

- Place controls directly in the group box using any of the usual methods.
- Draw controls outside the group box and drag them in.
- Copy and paste controls into the group box (prior to the paste operation, make sure the group box is selected).

To insure controls are properly place in a group box, try moving it and make sure the associated controls move with it. To remove a control from the group box, simple drag it out of the control.

Like the Form object, the group box is a **container** control, since it 'holds' other

controls. Hence, some controls placed in a group box will share **BackColor**, **ForeColor** and **Font** properties. To change this, select the desired control (after it is placed on the group box) and change the desired properties. Another feature of a group box control is that when its **Enabled** property is False, all controls in the group box are disabled. Likewise, when the group box control **Visible** property is False, all controls in the group box are not visible.

Moving the group box control on the form uses a different process than other controls. To move the group box, first select the control. Note a ‘built-in’ handle (looks like two sets of arrows) for moving the control appears at the upper left corner:



Click on this handle and you can move the control.

As mentioned, group boxes affect how radio buttons work. Radio buttons within a **GroupBox** control work as a **group**, independently of radio buttons in other **GroupBox** controls. Radio buttons on the form, and not in group boxes, work as another independent group. That is, the form is itself a group box by default. We'll see this in the next example.

### **GroupBox Properties:**

<b>Name</b>	Gets or sets the name of the group box (three letter prefix for group box name is <b>grp</b> ).
<b>BackColor</b>	Get or sets the group box background color.
<b>Enabled</b>	Gets or sets a value indicating whether the group box is enabled. If False, all controls in the group box are disabled.
<b>Font</b>	Gets or sets font name, style, size.
<b>ForeColor</b>	Gets or sets color of text.
<b>Text</b>	Gets or sets string displayed in title region of group box.

<b>Visible</b>	If False, hides the group box (and all its controls).
----------------	---

The **GroupBox** control has some methods and events, but these are rarely used. We are more concerned with the methods and events associated with the controls in the group box.

Typical use of **GroupBox** control:

- Set **Name** and **Text** property (perhaps changing **Font**, **BackColor** and **ForeColor** properties).
- Place desired controls in group box. Monitor events of controls in group box using usual techniques.

# ScrollBar Controls

## Horizontal ScrollBar In Toolbox:



## Horizontal ScrollBar On Form (Default Properties):



## Vertical ScrollBar In Toolbox:

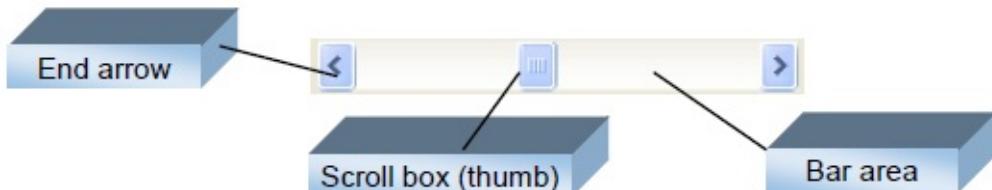


## Vertical ScrollBar On Form (Default Properties):



Scroll bars are widely used in Windows applications. Scroll bars provide an intuitive way to move through a list of information and make great input devices. Here, we use a scroll bar to obtain a whole number (**int** data type). The toolbox has both horizontal (**HScrollBar**) and vertical (**VScrollBar**) scroll bar controls.

Both type of scroll bars are comprised of three areas that can be clicked, or dragged, to change the scroll bar value. Those areas are:



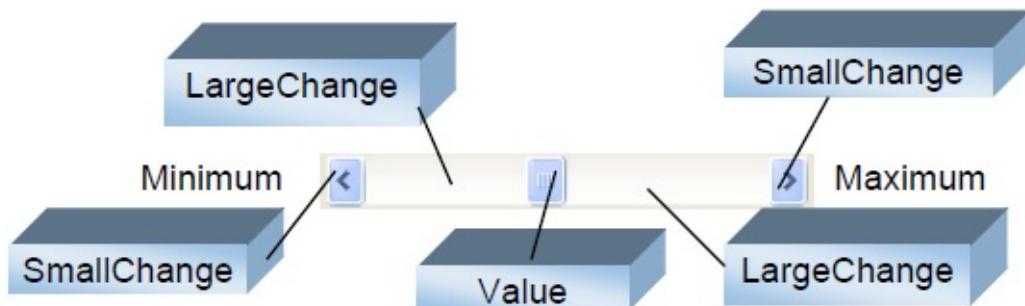
Clicking an **end arrow** increments the **scroll box** a small amount, clicking the **bar area** increments the scroll box a large amount, and dragging the scroll box (thumb) provides continuous motion. Using the properties of scroll bars, we can

completely specify how one works. The scroll box position is the only output information from a scroll bar.

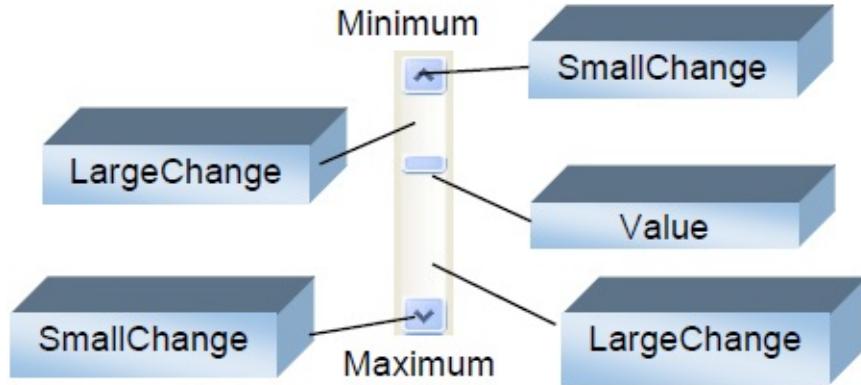
**ScrollBar Properties** (apply to both horizontal and vertical controls):

<b>Name</b>	Gets or sets the name of the scroll bar (three letter prefix for horizontal scroll bar is <b>hsb</b> , for vertical scroll bar <b>vsb</b> ).
<b>LargeChange</b>	Increment added to or subtracted from the scroll bar Value property when the bar area is clicked.
<b>Maximum</b>	The maximum value of the horizontal scroll bar at the far right and the maximum value of the vertical scroll bar at the bottom.
<b>Minimum</b>	The minimum value of the horizontal scroll bar at the left and the vertical scroll bar at the top
<b>SmallChange</b>	The increment added to or subtracted from the scroll bar Value property when either of the scroll arrows is clicked.
<b>Value</b>	The current position of the scroll box (thumb) within the scroll bar. If you set this in code, Visual C# moves the scroll box to the proper position.

Location of properties for **horizontal** scroll bar:



Location of properties for **vertical** scroll bar:



A couple of important notes about scroll bar properties:

1. Notice the vertical scroll bar has its **Minimum** at the top and its **Maximum** at the bottom. This may be counter-intuitive in some applications. That is, users may expect things to ‘go up’ as they increase. You can give this appearance of going up by defining another variable that varies ‘negatively’ with the scroll bar **Value** property.
2. If you ever change the **Value**, **Minimum**, or **Maximum** properties in code, make sure **Value** is at all times between **Minimum** and **Maximum** or the program will stop with an error message.

### ScrollBar Events:

<b>Scroll</b>	Occurs when the scroll box has been moved by either a mouse or keyboard action.
<b>ValueChanged</b>	Occurs whenever the scroll bar <b>Value</b> property changes, either in code or via a mouse action.

Typical use of **HScrollBar** and **VScrollBar** controls:

- Decide whether horizontal or vertical scroll bar fits your needs best.
- Set the **Name**, **Minimum**, **Maximum**, **SmallChange**, **LargeChange** properties. Initialize **Value** property.
- Monitor **Scroll** or **ValueChanged** event for changes in Value.

### A Note on the **Maximum** Property:

If **LargeChange** is not equal to one, the **Maximum** value cannot be achieved by clicking the end arrows, the bar area or moving the scroll box. It can only be achieved by setting the **Value** property in code. The maximum achievable **Value**, via mouse operations, is given by the relation:

$$\text{AchievableMaximum} = \text{Maximum} - \text{LargeChange} + 1$$

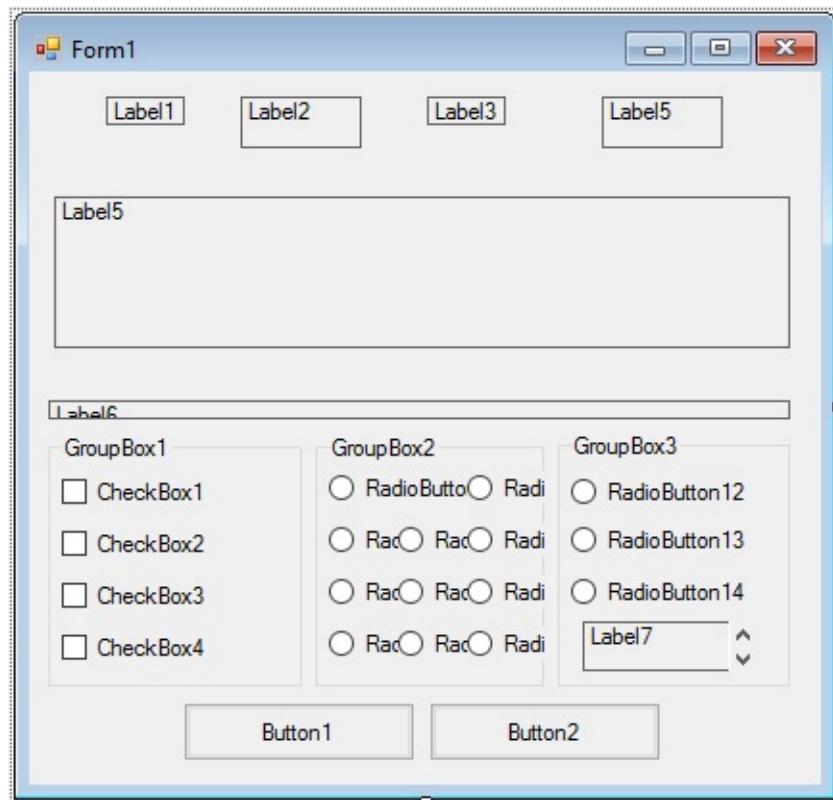
What does this mean? To meet an “achievable maximum,” you need to set the scroll bar Maximum property using this equation:

$$\text{Maximum} = \text{AchievableMaximum} + \text{LargeChange} - 1$$

For example, if you want a scroll bar to be able to reach 100 (with a LargeChange property of 10), you need to set **Maximum** to **109**, not 100! Very strange, I'll admit.

# Flash Card Math Form Design

We can begin building the flash card math project. Let's build the form. Start a new project in Visual C#. Place six label controls (for four of them, set **AutoSize** to **False** to allow resizing), three group box controls, and two button controls on your form. Add two timers to the project. In the first group box, place four check box controls. In the second group box, place eleven radio button controls. In the final group box, place three radio buttons, a label (**AutoSize** to **False**) and a vertical scroll bar. Resize and position controls so the form looks similar to this (I've temporarily set each label control's **BorderStyle** property to **FixedSingle** so you can see their size and location):



The first four label controls are used for scoring information. The large label is used for problem display. The wide, skinny label is a separator. The first group box is used to choose problem type, the second is used to select the problem factor and the third used to choose timer options. One button starts and stops the flash card problem and one exits the project. The timer control is used to time the overall process.

Set the control properties using the properties window:

**Form1** Form:

<b>Property Name</b>	<b>Property Value</b>
Name	frmFlash
BackColor	LightYellow
KeyPreview	True
Text	Flash Card Math
FormBorderStyle	Fixed Single
StartPosition	CenterScreen

**label1** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Tried:
Font Size	16

**label2** Label:

<b>Property Name</b>	<b>Property Value</b>
Name	lblTried
Text	0
TextAlign	MiddleCenter
AutoSize	False
BorderStyle	Fixed3D
BackColor	Red
ForeColor	Yellow
Font Size	16

**label3** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Correct:
Font Size	16

**label4** Label:

<b>Property Name</b>	<b>Property Value</b>
Name	lblCorrect
AutoSize	False
Text	0
.TextAlign	MiddleCenter
BackColor	Red
ForeColor	Yellow
BorderStyle	Fixed3D
Font Size	16

**label5** Label:

<b>Property Name</b>	<b>Property Value</b>
Name	lblProblem
AutoSize	False
Text	[Blank]
.TextAlign	MiddleCenter
ForeColor	Black
Font Name	Comic Sans MS
Font Size	48

**label6** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	[Blank]
BackColor	Red

**groupBox1** Group Box:

<b>Property Name</b>	<b>Property Value</b>
Name	grpType
Text	Type(s)
BackColor	Light Blue
ForeColor	Dark Blue

Font Size	10
Font Style	Bold

#### **checkBox1** Check Box:

<b>Property Name</b>	<b>Property Value</b>
Name	chkAdd
Text	Addition (+)
ForeColor	Black
Checked	True
Font Size	10

#### **checkBox2** Check Box:

<b>Property Name</b>	<b>Property Value</b>
Name	chkSubtract
Text	Subtraction (-)
ForeColor	Black
Font Size	10

#### **checkBox3** Check Box:

<b>Property Name</b>	<b>Property Value</b>
Name	chkMultiply
Text	Multiplication (x)
ForeColor	Black
Font Size	10

#### **checkBox4** Check Box:

<b>Property Name</b>	<b>Property Value</b>
Name	chkDivide
Text	Division (/)
ForeColor	Black
Font Size	10

**groupBox2** Group Box:

<b>Property Name</b>	<b>Property Value</b>
Name	grpFactor
Text	Factor
BackColor	Light Blue
ForeColor	Dark Blue
Font Size	10
Font Style	Bold

**radioButton1** Radio Button:

<b>Property Name</b>	<b>Property Value</b>
Name	rdoFactorRandom
Text	Random
ForeColor	Black
Checked	True
Font Size	10

**radioButton2** Radio Button:

<b>Property Name</b>	<b>Property Value</b>
Name	rdoFactor0
Text	0
ForeColor	Black
Font Size	10

**radioButton3** Radio Button:

<b>Property Name</b>	<b>Property Value</b>
Name	rdoFactor1
Text	1
ForeColor	Black
Font Size	10

**radioButton4** Radio Button:

<b>Property Name</b>	<b>Property Value</b>
Name	rdoFactor2
Text	2
ForeColor	Black
Font Size	10

**radioButton5** Radio Button:

<b>Property Name</b>	<b>Property Value</b>
Name	rdoFactor3
Text	3
ForeColor	Black
Font Size	10

**radioButton6** Radio Button:

<b>Property Name</b>	<b>Property Value</b>
Name	rdoFactor4
Text	4
ForeColor	Black
Font Size	10

**radioButton7** Radio Button:

<b>Property Name</b>	<b>Property Value</b>
Name	rdoFactor5
Text	5
ForeColor	Black
Font Size	10

**radioButton8** Radio Button:

<b>Property Name</b>	<b>Property Value</b>
Name	rdoFactor6
Text	6

ForeColor	Black
Font Size	10

**radioButton9** Radio Button:

<b>Property Name</b>	<b>Property Value</b>
Name	rdoFactor7
Text	7
ForeColor	Black
Font Size	10

**radioButton10** Radio Button:

<b>Property Name</b>	<b>Property Value</b>
Name	rdoFactor8
Text	8
ForeColor	Black
Font Size	10

**radioButton11** Radio Button:

<b>Property Name</b>	<b>Property Value</b>
Name	rdoFactor9
Text	9
ForeColor	Black
Font Size	10

**groupBox3** Group Box:

<b>Property Name</b>	<b>Property Value</b>
Name	grpTimer
Text	Timer
BackColor	Light Blue
ForeColor	Dark Blue
Font Size	10

Font Style              Bold

**radioButton12** Radio Button:

<b>Property Name</b>	<b>Property Value</b>
Name	rdoTimerOff
Text	Off
ForeColor	Black
Checked	True
Font Size	10

**radioButton13** Radio Button:

<b>Property Name</b>	<b>Property Value</b>
Name	rdoTimerOnUp
Text	On-Count Up
ForeColor	Black
Font Size	10

**radioButton14** Radio Button:

<b>Property Name</b>	<b>Property Value</b>
Name	rdoTimerOnDown
Text	On-Count Down
ForeColor	Black
Font Size	10

**label7** Label:

<b>Property Name</b>	<b>Property Value</b>
Name	lblTimer
AutoSize	False
Text	0:00
TextAlign	MiddleCenter
BackColor	White
ForeColor	Red

BorderStyle	Fixed3D
Font Size	14
Visible	False

**vScrollBar1** Vertical Scroll Bar:

Name	vsbTimer
Value	1
Minimum	1
Maximum	60
SmallChange	1
LargeChange	1
Visible	False

**button1** Button:

<b>Property Name</b>	<b>Property Value</b>
Name	btnStart
Text	Start Practice
Font Size	10

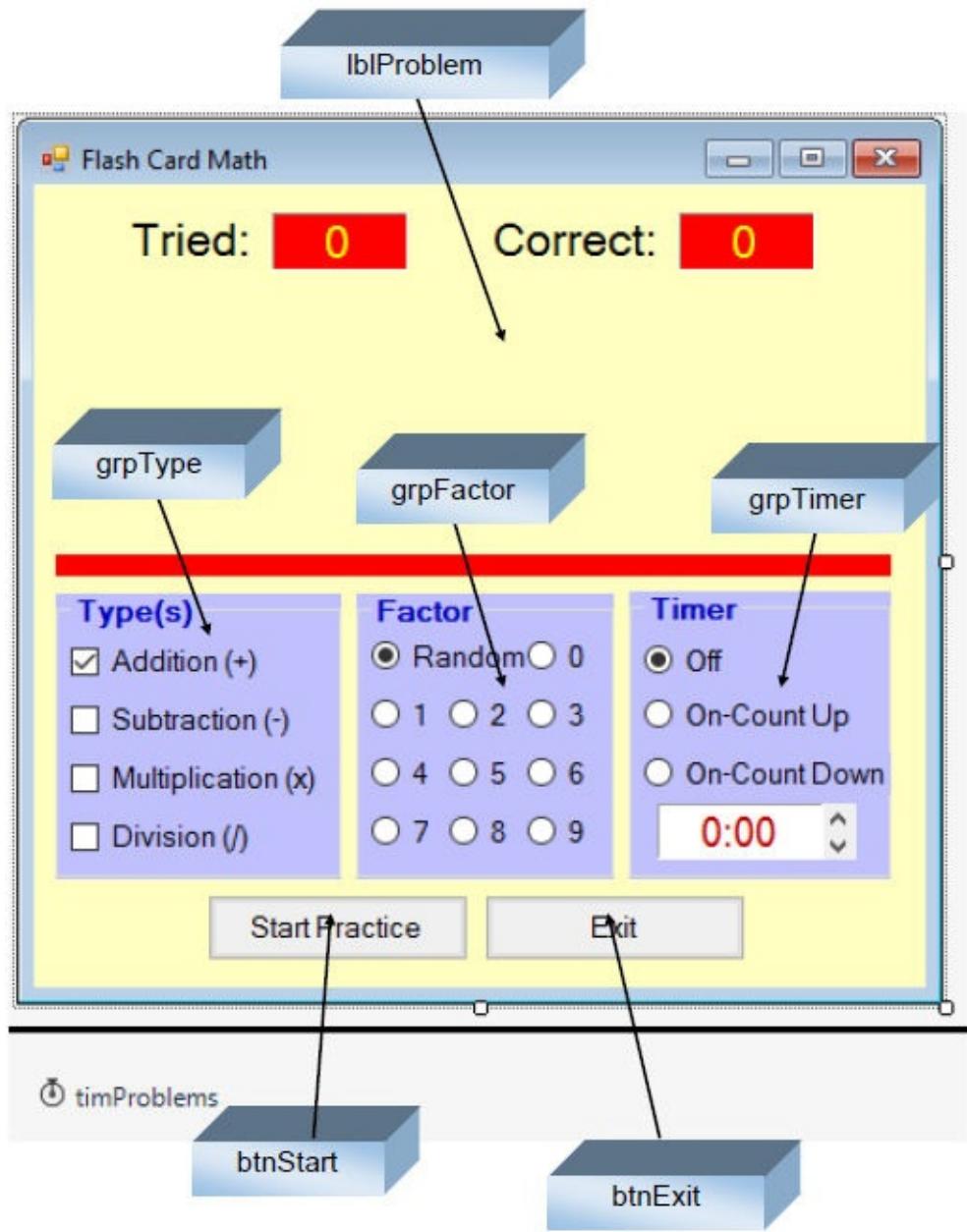
**button2** Button:

<b>Property Name</b>	<b>Property Value</b>
Name	btnExit
Text	Exit
Font Size	10

**timer1** Timer:

<b>Property Name</b>	<b>Property Value</b>
Name	timProblems
Interval	1000

When done setting properties, my form looks like this:



This completes the form design (we have not identified all the controls – they should be obvious).

We will begin writing code for the application. We will write the code in several steps. As a first step, we write the code that generates a random problem (using the **Random** object – review Chapter 2 if you need to) and gets the answer from the user, updating the score.

# Code Design – Start Practice

The idea of the flash card math project is to display a problem, receive an answer from the user and check for correctness. Problems can be of four different types with different factor choices and different timer options. For now, we will ignore the timer options. Once this initial code is working satisfactorily, timing will be considered. Again, this step-by-step approach to building a project is far simpler than trying to build everything at once.

Things begin by clicking the **Start Practice** button (**btnStart**). When this happens, the following steps are taken:

- Change **Text** property of **btnStart** to **Stop Practice**.
- Disable **btnExit**.
- Set number of problems tried and number correct to zero.
- Generate and display a problem in **lblProblem**.
- Obtain answer from user.
- Check answer and update score.

Once each generated problem is answered, subsequent problems are generated and answered.

The user answers problems until he/she clicks **Stop Practice** (or time elapses in timed drills). The steps followed at this point are:

- Change **Text** property of **btnStart** to **Start Practice**
- Enable **btnExit**.
- Clear **lblProblem**.
- Present results.

This code (for the **btnStart Click** event) is fairly straightforward. Let's build the framework. First, create a **Random** object and declare two form level variables to keep track of the number of problems tried and the number correct:

```
Random myRandom = new Random();
```

```
int numberTried, numberCorrect;
```

Now, use this code in the **btnStart Click** event (implements the steps above, except for presenting results):

```
private void btnStart_Click(object sender, EventArgs e)
{
    if (btnStart.Text == "Start Practice")
    {
        btnStart.Text = "Stop Practice";
        btnExit.Enabled = false;
        numberTried = 0;
        numberCorrect = 0;
        lblTried.Text = "0";
        lblCorrect.Text = "0";
        lblProblem.Text = GetProblem();
    }
    else
    {
        btnStart.Text = "Start Practice";
        btnExit.Enabled = true;
        lblProblem.Text = "";
    }
}
```

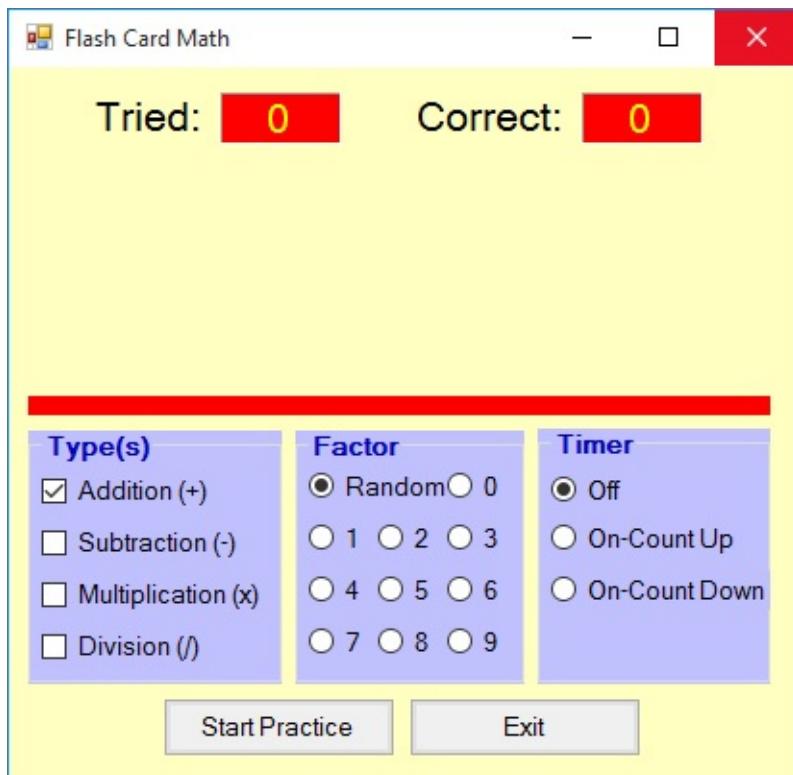
This code uses a general method **GetProblem** to generate the random problem and return it as a **string** type. Add this nearly empty method (we'll fill it in soon).

```
private string GetProblem()
{
    return ("Problem!");
}
```

And, while we're at it, code the **btnExit Click** event.

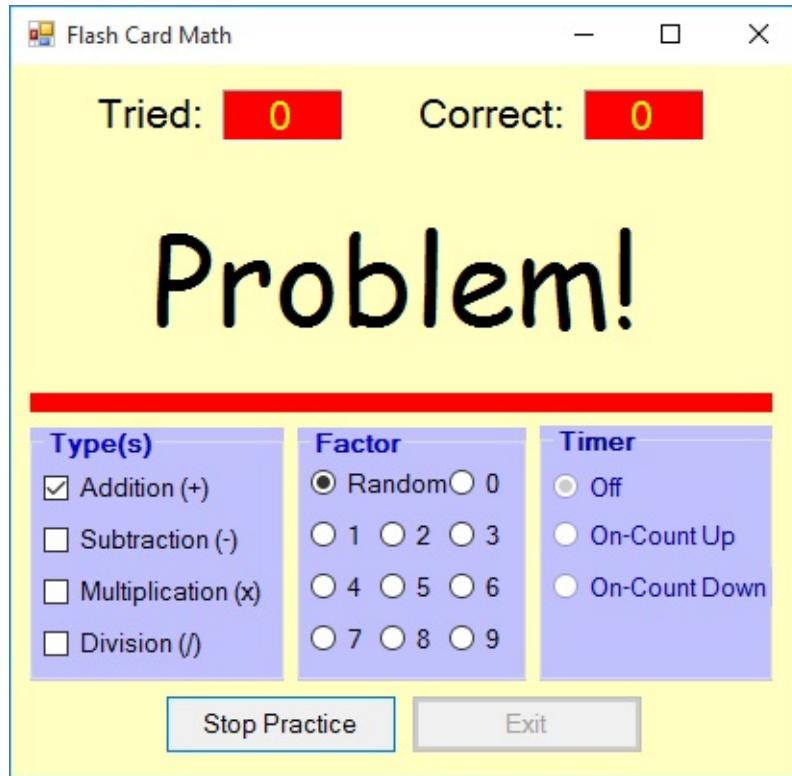
```
private void btnExit_Click(object sender, EventArgs e)
{
    this.Close();
}
```

Save and run the project. The form should appear as:



All controls are in their initial configuration. You can change options, but nothing will happen since there is no code behind any of the check boxes or radio buttons.

Click **Start Practice** to make sure buttons change as planned. You will also see the generated “problem”:



Now, click **Stop Practice**. Make sure **Exit** works.

This framework seems acceptable. We continue code design by discussing **problem generation** and **obtaining an answer** (including **scoring**) from the user. Then, later we discuss **timing** and **presenting the results**.

# Code Design – Problem Generation

To generate a problem, we examine the current options selected by the user and produce a random problem based on these selections. All code will be in the **GetProblem** general method currently in the framework code.

The steps involved in generating a random flash card problem are:

- Select problem type (random selection based on checked choices in **Type** group box)
- Generate factor (based on selection in **Factor** group box)
- Formulate problem and determine correct answer.
- Return problem as **string** type, replacing correct answer with question marks (?) in place of digits. An example of the desired form of the returned value is:

$$8 + 6 = ??$$

where question marks tell the user how many digits are in the correct answer.

Let's look at each step of the problem generation process. The first step is to choose a random problem type from the maximum of four possibilities. We will use a simple approach, first generating a random number from 1 to 4 (1 representing addition, 2 representing subtraction, 3 representing multiplication and 4 representing division). If the check box (in the **Type** group box) corresponding to the random number is checked, that will be the problem type. If the check box corresponding to the random number is not checked, we choose another random number. We continue this process until a problem type is selected. Notice this approach assumes at least one check box is always selected. We will make sure this is the case when developing code for the problem type option. There are more efficient ways to choose problem type which don't involve loops, but, for this simple problem, this works quite well.

A snippet of code that performs the choice of problem type (**p**) is:

```
int pType, p, number;
```

```

p = 0;
do
{
    pType = myRandom.Next(4) + 1;
    if (pType == 1 && chkAdd.Checked)
    {
        // Addition
        p = pType;
    }
    else if (pType == 2 && chkSubtract.Checked)
    {
        // Subtraction
        p = pType;
    }
    else if (pType == 3 && chkMultiply.Checked)
    {
        // Multiplication
        p = pType;
    }
    else if (pType == 4 && chkDivide.Checked)
    {
        // Division
        p = pType;
    }
}
while (p == 0);

```

Once a problem type is selected, we determine the factor used to generate a problem. It can be a selected value from 0 to 9, or a random value from 0 to 9, based on the radio button selected in the **Factor** group box. For now, we assume that value is provided by a general method **GetFactor(p)** that returns an **int** value, based on problem type **p**.

Each problem has four variables associated with it: **factor**, representing the

value returned by **GetFactor**, **number**, the other number used in the math problem, **correctAnswer**, the problem answer, and **problem**, a string representation of the unsolved problem. Once a problem type and factor have been determined, we find values for each of these variables. Each problem type has unique considerations for problem generation. Let's look at each type.

For **Addition** problems, the selected factor is the second **addend** in the problem. The string form of addition problems (**problem**) will be:

number + factor =

where **number** is a random value from 0 to 9, while recall **factor** is the selected factor. A snippet of code to generate an addition problem and determine the **correctAnswer** is:

```
number = myRandom.Next(10);
factor = GetFactor(1);
correctAnswer = number + factor;
problem = number.ToString() + " + " + factor.ToString() + " = ";
```

For **Subtraction** problems, the factor is the **subtrahend** (the number being subtracted). The string form of subtraction problems (**problem**) will be:

number - factor =

We want all the possible answers to be positive numbers between 0 and 9. Because of this, we formulate the problem in a backwards sense, generating a random answer (**correctAnswer**), then computing **number** based on that answer and the known factor (**factor**). The code that does this is:

```
factor = GetFactor(2);
correctAnswer = myRandom.Next(10);
number = correctAnswer + factor;
problem = number.ToString() + " - " + factor.ToString() + " = ";
```

For **Multiplication** problems, the selected factor is the **multiplier** (the number

you're multiplying by) in the problem. The string form of multiplication problems (**problem**) will be:

number x factor =

where **number** is a random value from 0 to 9, and **factor** is the factor. A snippet of code to generate a multiplication problem and determine the **correctAnswer** is:

```
number = myRandom.Next(10);
factor = GetFactor(3);
correctAnswer = number * factor; problem = number.ToString() + " x " +
factor.ToString() + " = ";
```

For **Division** problems, the factor is the **divisor** (the number doing the dividing). The string form of division problems (**problem**) will be:

number / factor =

Like in subtraction, we want all the possible answers to be positive numbers between 0 and 9. So, we again formulate the problem in a backwards sense, generating a random answer (**correctAnswer**), then computing **number** based on that answer and the known factor (**factor**). The code that does this is:

```
factor = GetFactor(4);
correctAnswer = myRandom.Next(10);
number = correctAnswer * factor;
problem = number.ToString() + " / " + factor.ToString() + " = ";
```

Note with division, we must make sure the factor is never zero (can't divide by zero).

The **GetFactor** routine provides the factor based on the radio button selection in the **Factor** group box and problem type **p**. For random factors, it will make sure a zero is not returned if a division problem is being generated. The **GetFactor** general method is thus:

```
private int GetFactor(int p)
{
    if (rdoFactor0.Checked)
        return (0);
    else if (rdoFactor1.Checked)
        return (1);
    else if (rdoFactor2.Checked)
        return (2);
    else if (rdoFactor3.Checked)
        return (3);
    else if (rdoFactor4.Checked)
        return (4);
    else if (rdoFactor5.Checked)
        return (5);
    else if (rdoFactor6.Checked)
        return (6);
    else if (rdoFactor7.Checked)
        return (7);
    else if (rdoFactor8.Checked)
        return (8);
    else if (rdoFactor9.Checked)
        return (9);
    else
    {
        //random
        if (p == 4)
            return (myRandom.Next(9) + 1);
        else
            return (myRandom.Next(10));
    }
}
```

If **Random** (**rdoFactorRandom**) option is not selected, the selected factor is returned (we will have to make sure zero is not a choice when doing division). If the **Random** option is selected, 0 to 9 is returned for addition, subtraction and multiplication problems; 1 to 9 is returned for division problems (**p = 4**).

The **GetProblem** function is nearly complete. We want to return the **problem** variable with appended question marks that represent the number of digits (**numberDigits**, another form level variable) in the correct answer. The code snippet that does this is:

```
if (correctAnswer < 10)
{
    numberDigits = 1;
    return (problem + "?");
}
else
{
    numberDigits = 2;
    return (problem + "??");
}
```

We can now assemble all the little code snippets into a final form for **GetProblem** method. First, add these declarations for form level variables:

```
int correctAnswer, numberDigits;
string problem;
```

To form the **GetProblem** method, start with the snippet that selects problem type. Then, add each problem generation segment (one for each of the four mathematical operations) in its corresponding location. Finally, add the question mark appending code. The finished method is:

```
private string GetProblem()
{
    int pType, p, number, factor;
```

```
p = 0;
do
{
    pType = myRandom.Next(4) + 1;
    if (pType == 1 && chkAdd.Checked)
    {
        // Addition
        p = pType;
        number = myRandom.Next(10);
        factor = GetFactor(1);
        correctAnswer = number + factor;
        problem = number.ToString() + " + " + factor.ToString() + " =
";
    }
    else if (pType == 2 && chkSubtract.Checked)
    {
        // Subtraction
        p = pType;
        factor = GetFactor(2);
        correctAnswer = myRandom.Next(10);
        number = correctAnswer + factor;
        problem = number.ToString() + " - " + factor.ToString() + " =
";
    }
    else if (pType == 3 && chkMultiply.Checked)
    {
        // Multiplication
        p = pType;
        number = myRandom.Next(10);
        factor = GetFactor(3);
        correctAnswer = number * factor;
        problem = number.ToString() + " x " + factor.ToString() + " =
";
```

```

";
}

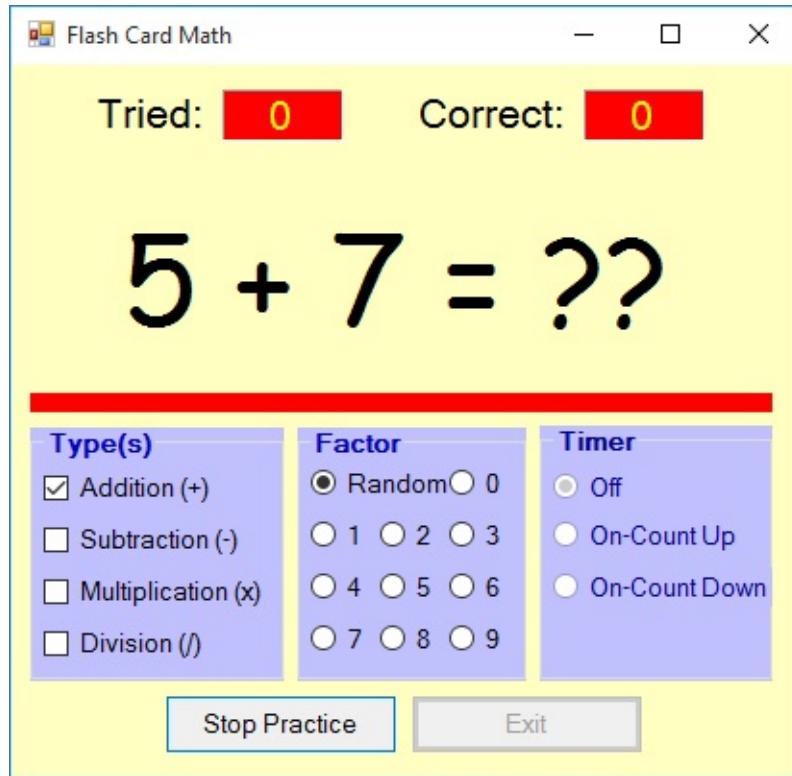
else if (pType == 4 && chkDivide.Checked)
{
    // Division
    p = pType;
    factor = GetFactor(4);
    correctAnswer = myRandom.Next(10);
    number = correctAnswer * factor;
    problem = number.ToString() + " / " + factor.ToString() + " =
";
}

while (p == 0);
if (correctAnswer < 10)
{
    numberDigits = 1;
    return (problem + "?");
}
else
{
    numberDigits = 2;
    return (problem + "??");
}
}

```

Add this to the project along with the code for **GetFactor**.

Save and run the project. Click **Start Practice** and you should see a random addition problem:



The two question marks tell us there are two digits in the correct answer. We'll see how to get that answer next. At this point, all you can do is click **Stop Practice**. You can then click **Start Practice** to see another addition problem if you'd like. View as many addition problems as you want.

Actually, you can also view other types of problems with other factors. But, you need to be careful. You need to make sure there is always at least one problem type selected. And, if **Division** problems are selected, make sure the selected **Factor** is not zero (**0**). Later, in code, we will make sure this doesn't happen.

# Code Design – Obtaining Answer

Once a problem is displayed, the user can enter the digits in the answer. These digits will be entered using the keyboard. The keystrokes will be handled by the form's **KeyPress** event. The form has a **KeyPress** event similar to that used in the previous chapter's project text box controls. When the form has focus, any keys pressed can be interpreted and used. In **Form Design**, we set the form's **KeyPreview** property to **True**. This allows any key strokes to be processed by the form **KeyPress** event, prior to being used by any other control. We will also insure the form has focus when it needs it. That way, no key strokes will be missed.

The steps for obtaining and checking an answer are:

- Make sure keystroke is a number (0 to 9).
- If number, keep keystroke as part of your answer and replace question mark with number.
- If a question mark remains, exit waiting for another keystroke.
- If all question marks are gone, compare your answer with correct answer.
- Increment the number of problems tried.
- If your answer is correct, increment the number of correct problems.
- Update scoring label controls.
- Generate another problem.

Declare form level variables to hold your answer and the current digit number in your answer:

```
string yourAnswer;  
int digitNumber;
```

The **frmFlash\_KeyPress** event that incorporates the steps listed above is then:

```
private void frmFlash_KeyPress(object sender, KeyPressEventArgs e)  
{
```

```
if (btnStart.Text == "Start Practice")
    return;
// only allow number keys
if (e.KeyChar >= '0' && e.KeyChar <= '9')
{
    e.Handled = false;
    yourAnswer += e.KeyChar;
    lblProblem.Text = problem + yourAnswer;
    if (digitNumber != numberDigits)
    {
        digitNumber++;
        lblProblem.Text += "?";
        return;
    }
    else
    {
        numberTried++;
        // check answer
        if (Convert.ToInt32(yourAnswer) == correctAnswer)
        {
            numberCorrect++;
        }
        lblTried.Text = numberTried.ToString();
        lblTried.Refresh();
        lblCorrect.Text = numberCorrect.ToString();
        lblCorrect.Refresh();
        lblProblem.Text = GetProblem();
        lblProblem.Refresh();
    }
}
else
{
```

```
    e.Handled = true;  
}  
}
```

In the first few lines of code, we make sure we are solving problems before allowing any keystrokes. Notice how all digits in your answer (represented by the typed character in **e.KeyChar**) are saved and concatenated into **yourAnswer**. Also, notice how the displayed problem is updated, overwriting a question mark, with each keystroke. As mentioned earlier, the program only gives you one chance to enter an answer - there is no erasing.

You need to add a few lines to the **GetProblem** method to initialize **yourAnswer** and **digitNumber**, with each new problem, and also give the form focus, so answers can be typed. The modified method is (new lines are coded, most unmodified code is not shown):

```
private string GetProblem()  
{  
    int pType, p, number, factor;  
    p = 0;  
    do  
    {  
        .  
        .  
    }  
    while (p == 0);  
    yourAnswer = "";  
    digitNumber = 1;  
    this.Focus();  
    if (correctAnswer < 10)  
    {  
        numberDigits = 1;  
        return (problem + "?");  
    }
```

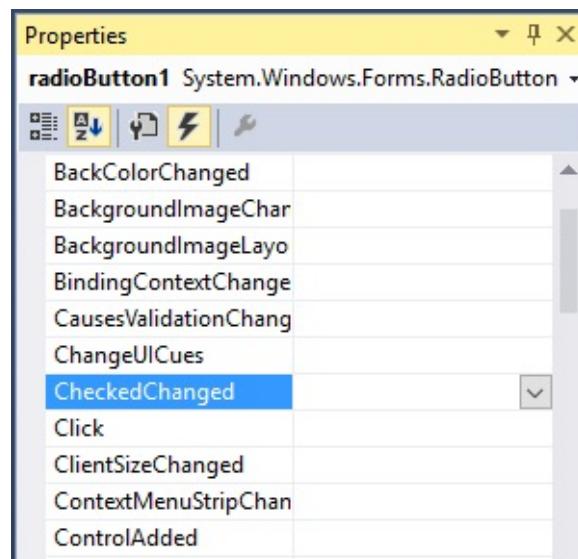
```
else
{
    numberDigits = 2;
    return (problem + "??");
}
}
```

Save and run the project. You should now be able to answer as many random addition problems as you'd like. Try it. Make sure the score is updating properly. You can stop practicing problems, at any time, by clicking the **Stop Practice** button. As mentioned earlier, you can also solve other types of problems with other factors, if you're careful changing the options. Make sure one problem type is always selected and make sure zero is not used with division problems. We'll fix this problem soon. But first, we address how to have multiple events address a single event method (needed for changing options).

# Handling Multiple Events in a Single Event Method

In the projects we've built thus far, each event method handles a single event. Now that we are grouping controls like check boxes and radio buttons, it would be nice if a single method could handle multiple events. For example, if we have 4 radio buttons in a group, when one button is clicked, it would be preferable to have a single method where we decide which button was clicked, as opposed to having to monitor 4 separate event methods. Let's see how to do this.

We use this radio button example to illustrate. Assume we have four radio buttons (**radioButton1**, **radioButton2**, **radioButton3**, **radioButton4**) that we want to share the same **CheckedChanged** event method. To make this method sharing possible, we use the properties window to establish events. First, we select **radioButton1**, go to the properties window and click the **Events** button. The **CheckedChanged** event will be highlighted since it is the default event for the radio button control:



Usually, at this point, we would double-click the event and the Visual C# environment would build an event method named **radioButton1\_CheckedChanged**. The name assigned to this method by Visual C# is arbitrary. We can assign any name we want and it would still handle the

**CheckedChanged** event for **radioButton1**. To use a different name, we type the name next to the event in the properties window. Since the method will now handle multiple events, we would want to use a name to represent a group of radio buttons, not just a single button. A name like **group1\_CheckedChanged** might be more appropriate – we will use that in this example.

Once you type a name for an event in the properties window and press <Enter>, the code window will open in the new method. For this example, we would see:

```
private void group1_CheckedChanged(object sender, EventArgs e)
{
}
```

Right now, this method only handles the **CheckedChanged** event for **radioButton1**.

To have another control share this method, we again use the properties window. Next, we would choose **radioButton2** in the properties window, then display its events and click the drop-down box that appears to the right of the **CheckedChanged** event. One of the choices will be the **group1\_CheckedChanged** event method. By simply choosing this existing method, it will now be the method called for the **CheckedChanged** event for **radioButton2**. We then repeat this process for each control we want to share the same event method. **Hint:** a quick way to do this is to “group select” every control on the form that is to share an event method. Then using event selection in the properties window, choose the shared method.

With this process, we can attach any existing event of any control to any method we want! It is best to append like events of like controls. It is possible to process dissimilar events of dissimilar controls with a single event method, but you need to be careful.

If we have a single method responding to events from multiple controls, how do we determine which particular event from which particular control invoked the method. In our example with a single method handling the **CheckedChanged** event for 4 radio buttons, how do we know which of the 4 buttons is invoking the method? The **sender** argument of the event method provides the answer.

Each event method has a **sender** argument (the first argument of two) which identifies the control whose event caused the method to be invoked. With a little Visual C# coding, we can identify the **Name** property (or any other needed property) of the sending control. For our radio button example, that code is:

```
RadioButton rdoExample;  
string buttonName;  
// Determine which button was clicked  
rdoExample = (RadioButton) sender;  
buttonName = rdoExample.Name;
```

In this code, we define a variable (**rdoExample**) to be the type of the control attached to the method (a **RadioButton** in this case). Then, we assign this variable to **sender** (after casting **sender** to the proper control type). Once this variable is established, we can determine any property we want to identify the button. In the above example, we find the radio button's **Name** property. With this information, we now know which particular button was selected and we can process any code associated with this radio button. Notice a shortcut way to identify the **Name** property (without declaring any variables) is to use:

**((RadioButton) sender).Name**

# Code Design – Choosing Problem Type

The selection of problem type seems simple. Choose the check box or check boxes you want and the correct problem will be generated. But there are a couple of problems we've alluded to. Currently, there is nothing to prevent a user from "unchecked" all the boxes, leaving no problem type to select. We must make sure at least one box is always selected. And, if **Division** problems are selected, we cannot allow zero (0) to be used as a factor. We now write code to address these problems.

We can use the technique of having a single method handle multiple events to process clicking check boxes in the **Type** group box. We create a single method named **chkType\_CheckedChanged** that will handle the **CheckedChanged** events for any of four check boxes: **chkAdd**, **chkSubtract**, **chkMultiply**, and **chkDivide**.

To create this method, first access the **CheckedChanged** event method for **chkAdd**. You should see:

```
private void chkAdd_CheckedChanged(object sender, EventArgs e)
{
    .
    .
}
```

Now, simply change the method name to **chkType\_CheckedChanged** and assign this event method to the other three check boxes – **chkSubtract**, **chkMultiply**, and **chkDivide**. To do this, select the new method name from the events list in the properties window for the particular control.

In this method, these steps are followed:

- Determine which check box was clicked.
- Determine how many boxes are checked.
- If **chkDivide** is checked, make sure zero is not the selected factor; if it is,

change the factor to 1. Also, if **chkDivide** is checked, disable **btnFactor0**.

- If no boxes are checked, “recheck” selected check box.
- Set focus on form.

The **chkType\_CheckedChanged** method that implements these steps is:

```
private void chkType_CheckedChanged(object sender, EventArgs e)
{
    CheckBox clickedBox;
    int numberChecks;
    // determine which box was clicked
    clickedBox = (CheckBox) sender;
    // determine how many boxes are checked
    numberChecks = 0;
    if (chkAdd.Checked)
        numberChecks++;
    if (chkSubtract.Checked)
        numberChecks++;
    if (chkMultiply.Checked)
        numberChecks++;
    if (chkDivide.Checked)
    {
        numberChecks++;
        // make sure zero not selected factor
        if (rdoFactor0.Checked)
            rdoFactor1.PerformClick();
        rdoFactor0.Enabled = false;
    }
    else
    {
        rdoFactor0.Enabled = true;
    }
}
```

```
// if all boxes unchecked, recheck last clicked box
if (numberChecks == 0)
    clickedBox.Checked = true;
this.Focus();
}
```

You should be able to see how the various steps are implemented. Enter this code into your project.

Save and run the project. Make sure all the newly installed code is doing its job. Try to “uncheck” all the problem type boxes – one box will always remain. Check **Division** problems and notice that the 0 option for **Factor** is disabled. Uncheck **Division** problems. Choose 0 as a factor. Now, check **Division** again. Notice the factor is changed to 1 and the 0 option is disabled. You can now solve any problem type with any factor. If you change options while solving problems, the changes will be seen once you finish solving the current problem. Try solving problems, changing problem type and factors.

# Code Design – Timing Options

Having coded problem generation and answer checking, we can now address the use of timing in the flash card math project. Up to now, we've assumed no timer has been used. We have two possibilities for a timer: (1) one where the timer counts up, keeping track of how long you are solving problems, and (2) one where the timer counts down from some preset value. In both cases, a label control (**lblTimer**) displays the time in **minutes:seconds** form. In the second case, a vertical scroll bar (**vsbTimer**) is used to set the value. The timing will be controlled with a timer control (**timProblems**) with an interval of 1 second (1000 milliseconds).

We will use a form level variable (**problemTime**) to store the time value (whether counting up or down) in seconds. Add this variable in the general declarations area:

```
int problemTime;
```

First, we write the code to switch from one timing option to the next. We will use a single method to handle the **CheckedChanged** event for the three radio buttons in the **Timer** group box. The steps followed in this method:

- If Off (**rdoTimerOff**) is selected: disable **lblTimer** and **vsbTimer**.
- If On-Count Up (**rdoTimerOnUp**) is selected, enable **lblTimer** (initialize display) and disable **vsbTimer**. Initialize **problemTime** to 0.
- If On-Count Down (**rdoTimerOnDown**) is selected, enable **lblTimer** (initialize display) and **vsbTimer**. Initialize **problemTime** to 30 times **vsbTimer Value** property (30 seconds for each increment).

The method (**rdoTimer\_CheckedChanged**) that implements the steps is (again, this handles the three radio buttons in the **Timer** group box):

```
private void rdoTimer_CheckedChanged(object sender, EventArgs e)
{
    if (rdoTimerOff.Checked)
```

```

{
    lblTimer.Visible = false;
    vsbTimer.Visible = false;
}
else if (rdoTimerOnUp.Checked)
{
    problemTime = 0;
    lblTimer.Text = GetTime(problemTime);
    lblTimer.Visible = true;
    vsbTimer.Visible = false;
}
else if (rdoTimerOnDown.Checked)
{
    problemTime = 30 * vsbTimer.Value;
    lblTimer.Text = GetTime(problemTime);
    lblTimer.Visible = true;
    vsbTimer.Visible = true;
}
}

```

Add this method to the project.

The **rdoTimer CheckedChanged** event uses a general method **GetTime** that returns the time properly formatted:

```

private string GetTime(int s)
{
    int min, sec;
    string ms, ss;
    min = (int) (s / 60);
    sec = s - 60 * min;
    ms = min.ToString();
    ss = sec.ToString();
}

```

```

if (sec < 10)
    ss = "0" + ss;
return (ms + ":" + ss);
}

```

This method takes the time (**S**) in seconds and breaks it into minutes and seconds. Add this new code to your project.

When **rdoTimerOnDown** is checked (the count down option), the scroll bar (**vsbTimer**) **Value** property is used to initialize the **problemTime** variable. In the code, each increment on the scroll bar adds 30 seconds to the timer. If you look back to where we set control properties, you will see the **Maximum** property for this scroll bar was set to 60. This allows a maximum of 30 minutes (1800 seconds) for a timed flash card math session. We need a **vsbTimer Scroll** event method to change the displayed time whenever the scroll bar arrows are clicked. That method is:

```

private void vsbTimer_Scroll(object sender, ScrollEventArgs e)
{
    lblTimer.Text = GetTime(30 * vsbTimer.Value);
    lblTimer.Refresh();
}

```

We allow changing problem type and factors while solving problems. It wouldn't make sense to be able to change timer options while solving problems – the times would not be correct. We will only allow selection of timer options prior to clicking **Start Practice**. Clicking **Start Practice** will start the timing process (controlled by **timProblems**); the steps are:

- Disable **grpTimer**.
- If **rdoTimerOff** is selected, do nothing else.
- If **rdoTimerOnUp** is selected:
  - o Initialize **ProblemTime** to zero; display **ProblemTime**.
  - o Enable **timProblems**.
- If **rdoTimerOnDown** is selected:
  - o Initialize **ProblemTime** to 30 times **vsbTimer.Value**; display **ProblemTime**.

- o Enable **timProblems**.

Clicking **Stop Practice** will stop the timing process. The corresponding steps:

- Enable **grpTimer**.
- Disable **timProblems**.

Each of these steps is handled in the **btnStart Click** event. The modified method (changes are shaded) is:

```
private void btnStart_Click(object sender, EventArgs e)
{
    if (btnStart.Text == "Start Practice")
    {
        btnStart.Text = "Stop Practice";
        btnExit.Enabled = false;
        numberTried = 0;
        numberCorrect = 0;
        lblTried.Text = "0";
        lblCorrect.Text = "0";
        grpTimer.Enabled = false;
        if (!rdoTimerOff.Checked)
        {
            if (rdoTimerOnUp.Checked)
                problemTime = 0;
            else
                problemTime = 30 * vsbTimer.Value;
            lblTimer.Text = GetTime(problemTime);
            timProblems.Enabled=true;
        }
        lblProblem.Text = GetProblem();
    }
    else
```

```

{
    grpTimer.Enabled = true;
    timProblems.Enabled = false;
    btnStart.Text = "Start Practice";
    btnExit.Enabled = true;
    lblProblem.Text = "";
}
}

```

Make the indicated changes. We're almost ready to try the timing – just one more method to code.

When the timer (**timProblems**) is enabled, the time display (**lblDisplay**) is updated every second (we use an **Interval** property of 1000). The displayed time is incremented if counting up, decremented if counting down. The steps involved for counting up are:

- Increment **ProblemTime** by 1.
- Display **ProblemTime**.
- If **ProblemTime** is 1800 (30 minutes), stop solving problems.

Note we limit the total solving time to 30 minutes.

The steps for counting down are:

- Decrement **ProblemTime** by 1.
- Display **ProblemTime**.
- If **ProblemTime** is 0, stop solving problems.

The code to update the displayed time is placed in the **timProblems Tick** event. The code that implements the above steps are:

```

private void timProblems_Tick(object sender, EventArgs e)
{
    if (rdoTimerOnUp.Checked)

```

```

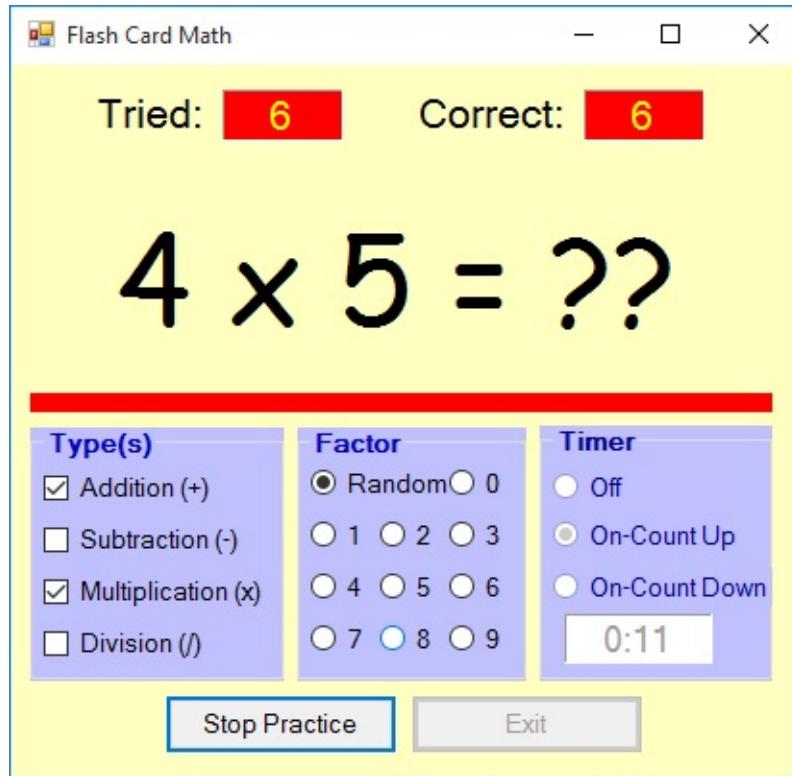
{
    problemTime++;
    lblTimer.Text = GetTime(problemTime);
    if (problemTime >= 1800)
    {
        btnStart.PerformClick();
        return;
    }
}
else
{
    problemTime--;
    lblTimer.Text = GetTime(problemTime);
    if (problemTime == 0)
    {
        btnStart.PerformClick();
        return;
    }
}
lblTimer.Refresh();
}

```

Notice to stop solving problems, we simulate a click on **Stop Practice** (the **btnStart** button). Add this method to the project.

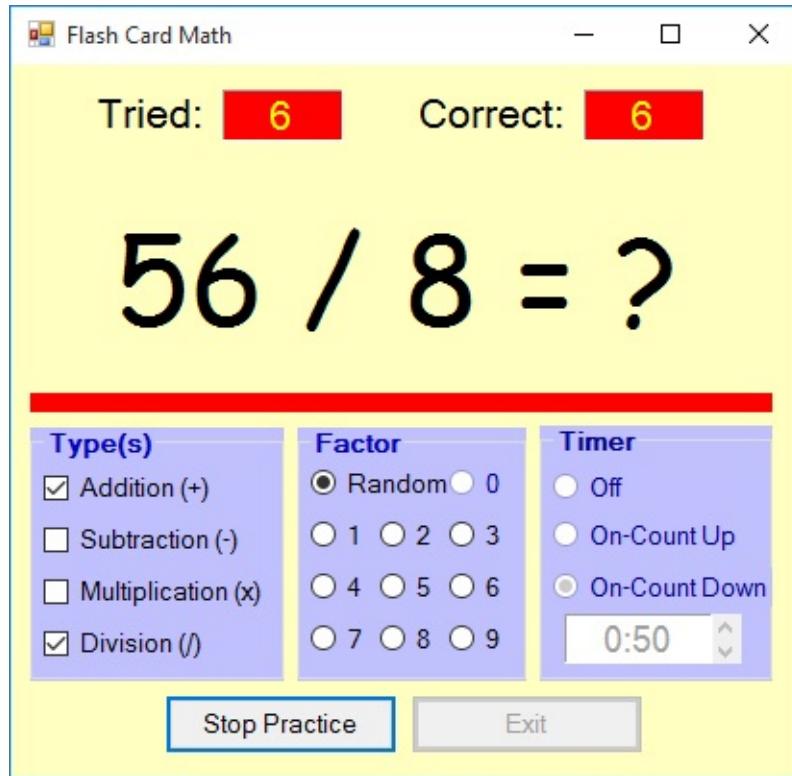
We're done implementing the modifications to add timing in the flash card math project. Save and run the project. You want to make sure all the timer options work correctly. First, check to see that the project still works correctly with no timer.

Once you are convinced the no timer option still works, stop solving problems and choose the **On-Count Up** option. Run the project. Make sure the timer increments properly. Here's a run I just started (note the **Timer** group box is properly disabled):



Click **Stop Practice** at some point. You should also make sure the program automatically stops after 30 minutes (go have lunch while the program runs).

Choose the **On-Count Down** option. Change the amount of allowed time using the vertical scroll bar. Make sure it reaches a maximum of 30:00 (it has a minimum of 0:30). Start the project. Make sure the time decrements correctly. Here's a run I made using a starting time of 1:00:



Make sure the program stops once the time elapses.

# Code Design – Presenting Results

Once a user stops solving problems, we want to let he/she know how well they did in answering problems. The information of use would be:

- The number of problems tried
- The number of correct answers
- The percentage score
- If timing, amount of elapsed time and time spent (on average) on each problem

If timing up, the elapsed time is equal to **problemTime**. If timing down, the elapsed time is equal to the initial amount of time minus **problemTime**.

All of this information is readily available from the current variable set. The results are presented in the **btnStart Click** event method (following clicking of **Stop Practice**). We will use a simple message box to relay the results. The modified **btnStart Click** method (changes are shaded) that displays the results is:

```
private void btnStart_Click(object sender, EventArgs e)
{
    int score;
    string message = "";
    if (btnStart.Text == "Start Practice")
    {
        btnStart.Text = "Stop Practice";
        btnExit.Enabled = false;
        numberTried = 0;
        numberCorrect = 0;
        lblTried.Text = "0";
        lblCorrect.Text = "0";
        grpTimer.Enabled = false;
```

```

if (!rdoTimerOff.Checked)
{
if (rdoTimerOnUp.Checked)
    problemTime = 0;
else
    problemTime = 30 * vsbTimer.Value;
lblTimer.Text = GetTime(problemTime);
timProblems.Enabled=true;
}
lblProblem.Text = GetProblem();

}
else
{
    grpTimer.Enabled = true;
    timProblems.Enabled = false;
    btnStart.Text = "Start Practice";
    btnExit.Enabled = true;
    lblProblem.Text = "";
    if (numberTried > 0)
    {
        score = (int)(100 * (double) (numberCorrect) / numberTried);
        message = "Problems Tried: " + numberTried.ToString() +
"\r\n";
        message += "Problems Correct: " +
numberCorrect.ToString() + " (" + score.ToString() + "%)" + "\r\n";
        if (rdoTimerOff.Checked)
        {
            message += "Timer Off";
        }
        else
        {
            if (rdoTimerOnDown.Checked)

```

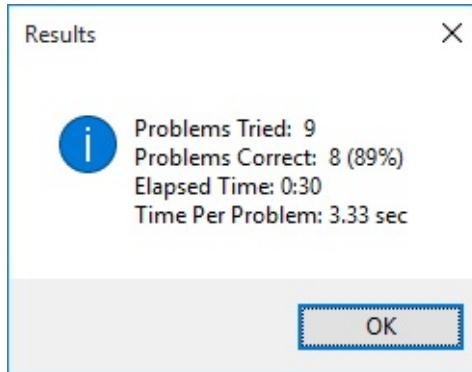
```

    {
        problemTime = 30 * vsbTimer.Value - problemTime;
    }
    message += "Elapsed Time: " + GetTime(problemTime) +
    "\r\n";
    message += "Time Per Problem: " + String.Format(
    "{0:f2}", (double) (problemTime) / numberTried) + " sec";
}
MessageBox.Show(message, "Results",
MessageBoxButtons.OK, MessageBoxIcon.Information);
}
}

```

Add the noted changes.

One last time – save and run the project. Solve some problems and see the results. Make sure the results display correctly whether timing or not. Here is a set of results I received while using the timing down option:



# Flash Card Math Quiz Project Review

The **Flash Card Math Quiz** project is now complete. Save and run the project and make sure it works as designed. Recheck that all options work and interact properly. Let your kids (or anyone else) have fun tuning up their basic math skills.

If there are errors in your implementation, go back over the steps of form and code design. Use the debugger when needed. Go over the developed code – make sure you understand how different parts of the project were coded. As mentioned in the beginning of this chapter, the completed project is saved as **Flash Card** in the **HomeVCS\HomeVCS Projects** folder.

While completing this project, new concepts and skills you should have gained include:

- Capabilities and proper use of the check box, radio button and group box controls.
- How to use horizontal and vertical scroll bars as input devices.
- Simple use of the **Random** object.
- Use of the **KeyPress** event for “form” input.
- Invoking a single event method from multiple events.
- Using a message box to report results.

# Flash Card Math Quiz Project Enhancements

Possible enhancements to the flash card math project include:

- As implemented, the only feedback a user gets about entered answers is an update of the score. Some kind of audible feedback would be a big improvement (a positive sound for correct answer, a negative sound for a wrong answer). We discuss adding sounds to a project in Chapter 12 – you might like to look ahead.
- When a user stops answering problems, it would be nice to have a review mode where the problems missed are presented. You would need some way to save each problem that was answered incorrectly.
- Kids like rewards. As you gain more programming skills, a nice visual display of some sort for good work would be a fun addition.
- Currently, once a problem is answered, the next problem is immediately displayed. Some kind of delay (perhaps make it optional and adjustable) might be desired. You would need another timer control.

7

## Multiple Choice Exam Project

# Review and Preview

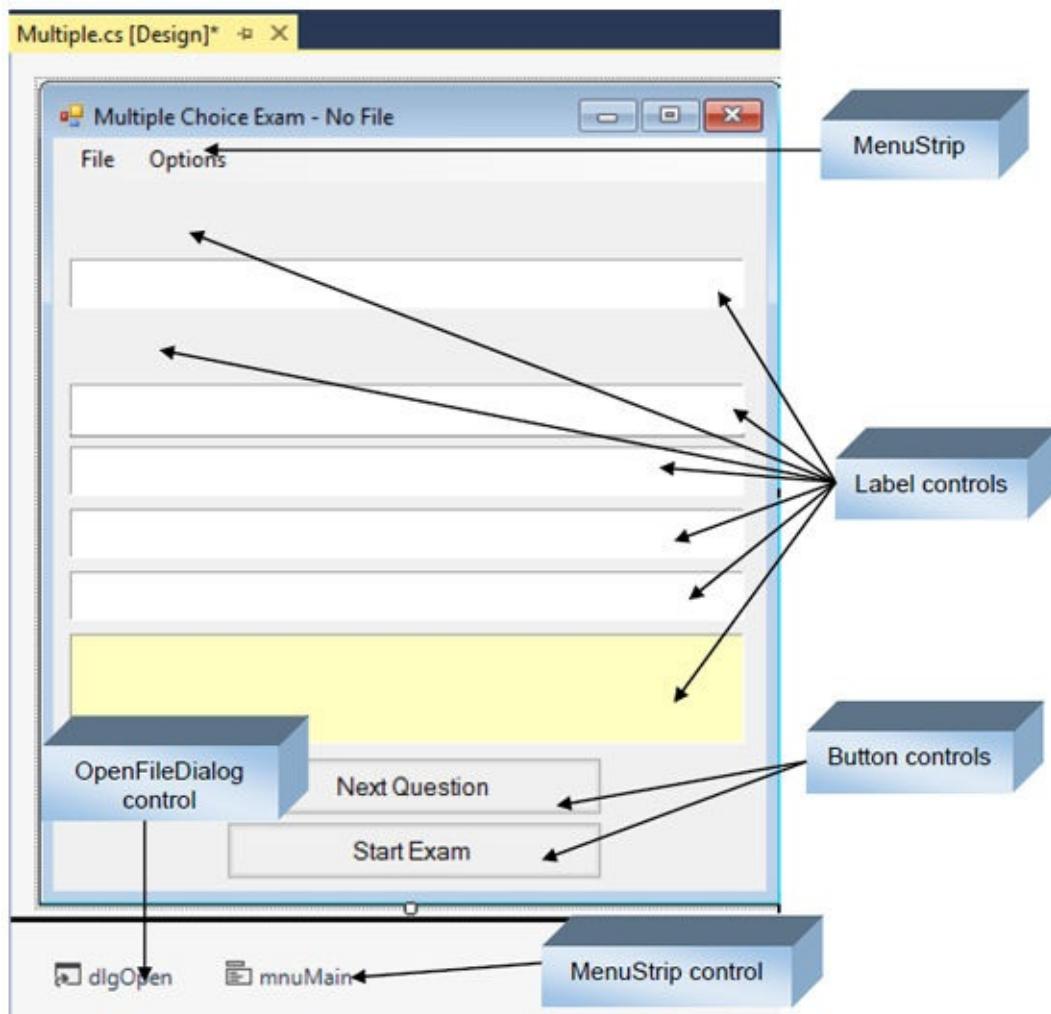
In this chapter, we build a project that quizzes a user on matching pairs of items – for example, states (or countries) and capital cities, words and meanings, books and authors, inventions and inventors.

The **Multiple Choice Exam Project** allows you to select which item is given and which should be provided as the answer and whether the answers should be multiple choice or typed in. The project introduces use of menus in Visual C# projects, as well as reading information (using the open file dialog control) from files.

# Multiple Choice Exam Project Preview

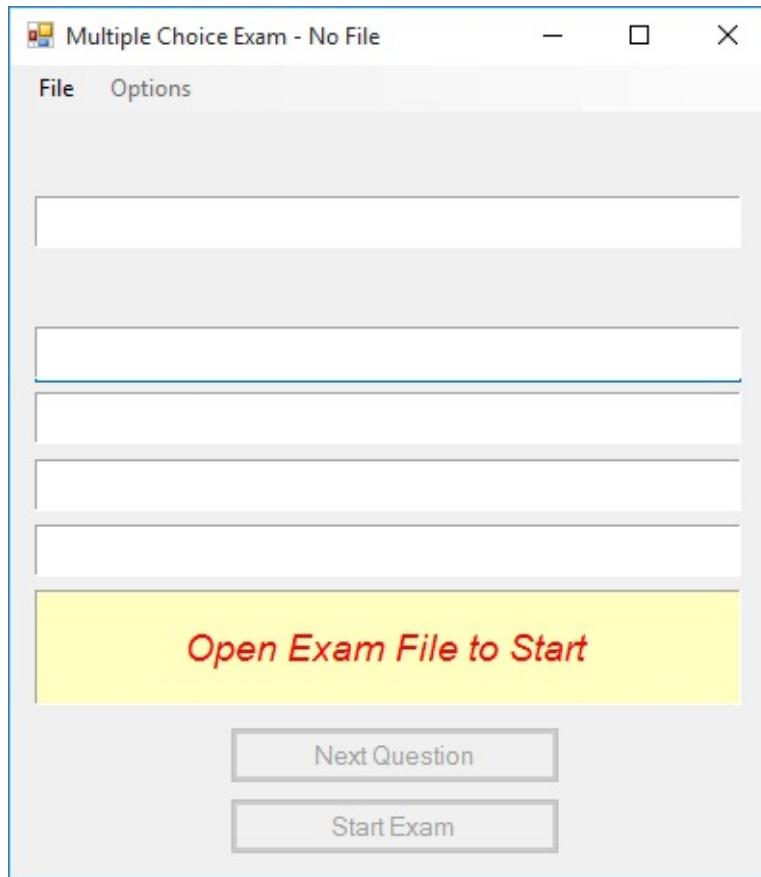
In this chapter, we will build a **multiple choice exam** program. Random items from a provided list are displayed to the user. The user picks the item that matches (or goes with the displayed item). For example, if a country is listed, the user may be asked for the capital city. Answers can be multiple choice or typed in.

The finished project is saved as **Multiple Choice** in the **HomeVCS\HomeVCS Projects** folder. Start Visual C# and open the finished project. Open the form (double-click **Multiple.cs** in Solution Explorer) and you will see:

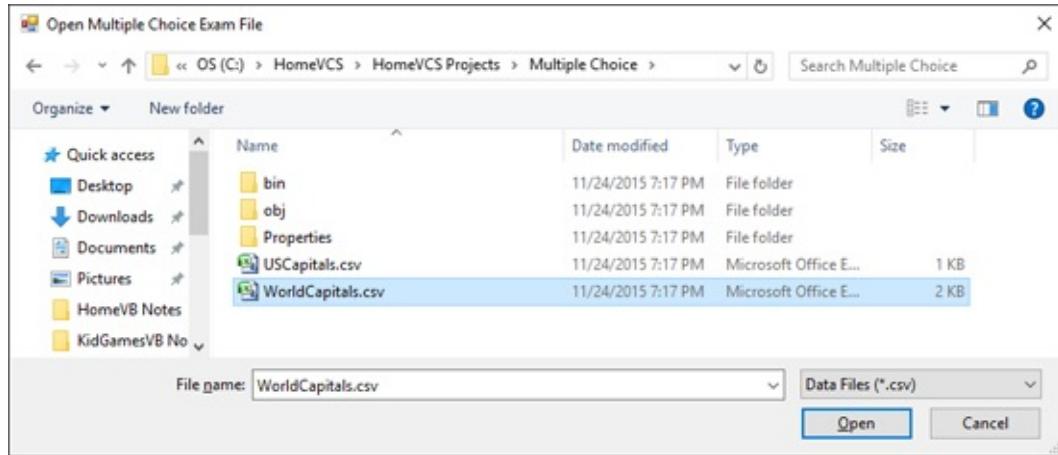


There are lots of controls here (including a couple of new ones). A menu strip is used to control the program. Two label controls (blank, with same background color of form, so they appear invisible) are used for header information. Four white labels are used for multiple choice answers and a large yellow label is used to provide comments to the user. There is also a text box control behind one of the label controls, used for entering typed-in answers. Two button controls are used to move from question to question and to start and stop the exam. At the bottom is an open file dialog control which will be used to open exam files and use them in the project. Also at the bottom is a menu strip control used to define the menu structure. These last two controls are new to these notes – we will discuss them in detail.

Run the project (press <F5>). The multiple choice exam program will appear as:

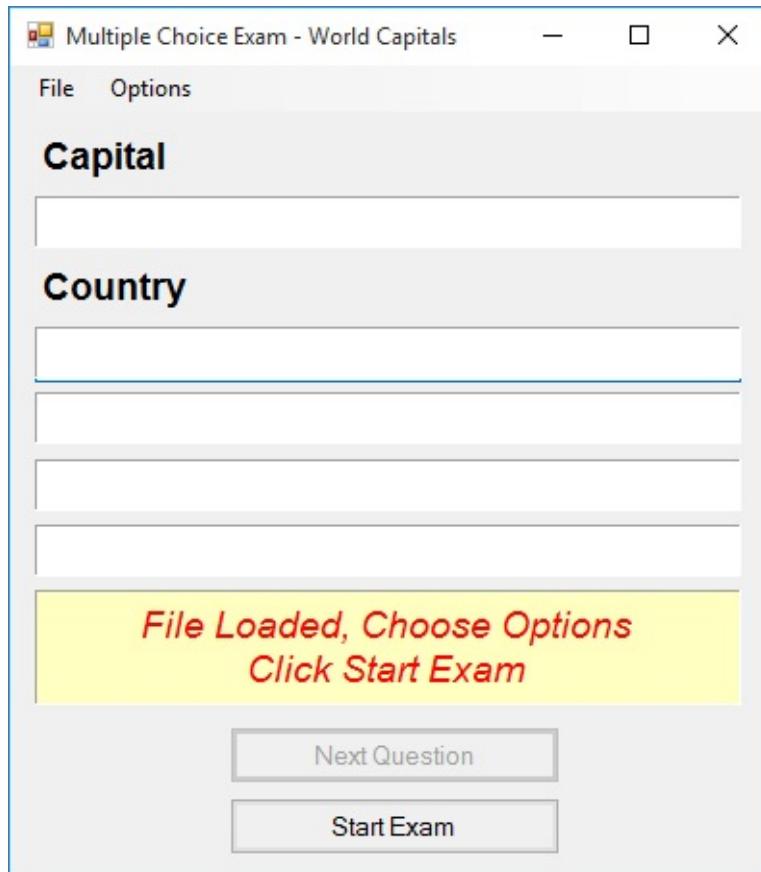


In the message area you see "***Open Exam File to Start***". The information used for a multiple choice exam is stored in files you build (we will discuss how to do this). So, the first step is to open and load such a file. Choose the **File** menu item and click **Open**. An open file dialog box will appear:



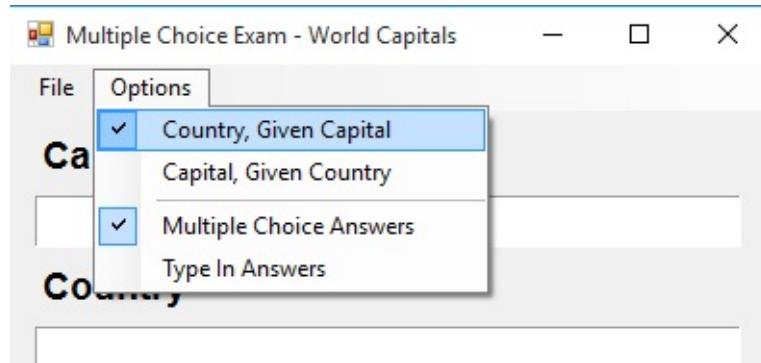
As shown above, navigate to the **HomeVCS\HomeVCS Projects\Multiple Choice** folder. The two files **USCapitals.csv** (listing states and capitals) and **WorldCapitals.csv** (listing countries and capitals) are example exam files included with these notes. Choose **WorldCapitals.csv** and click **Open**.

The file will open and the project form should now appear as:

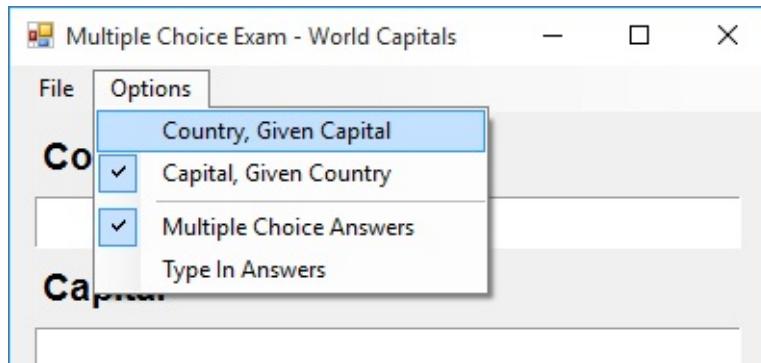


Notice headers (**Capital** and **Country**) are now listed on the form. The form has a caption (**Multiple Choice Exam – World Capitals**) with the exam title. The program is now asking you to select options before starting the exam.

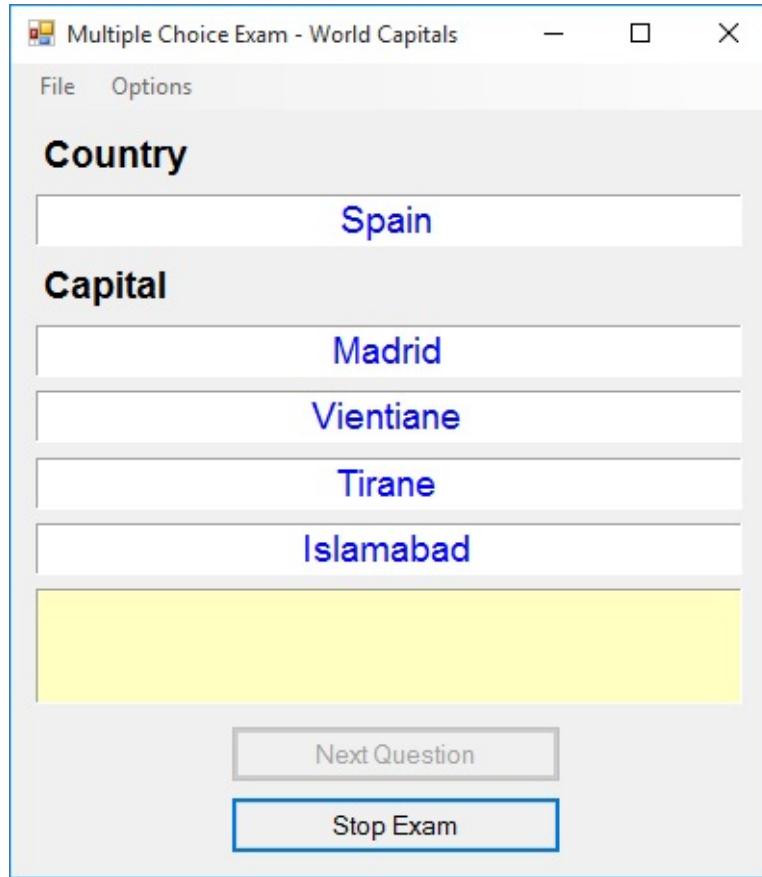
Click the **Options** menu item and you will see four options with the default selections indicated by check marks:



Two choices are to be made. In this example, you are asked whether you want to name the **Country**, given the **Capital**, or vice versa. Let's choose **Capital, Given Country**. The other choice is whether you want to be provided with a list of multiple choice answers or you want to type in your answer. Make sure a check is next to **Multiple Choice Answers**. The choices should now appear as:

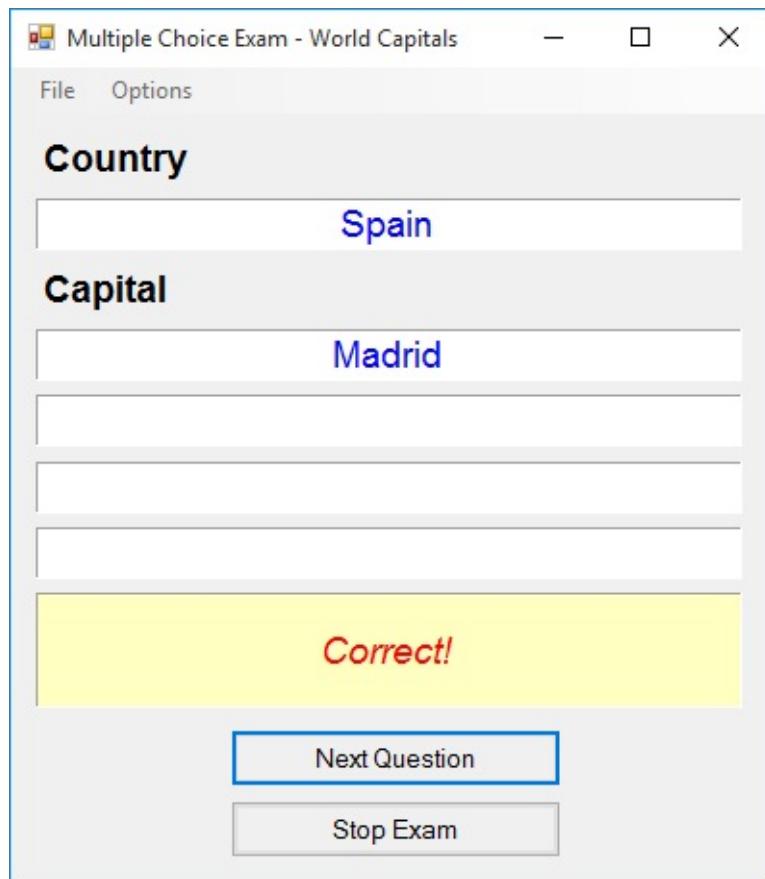


We're ready to start the exam. Click the button marked **Start Exam** and you will see:



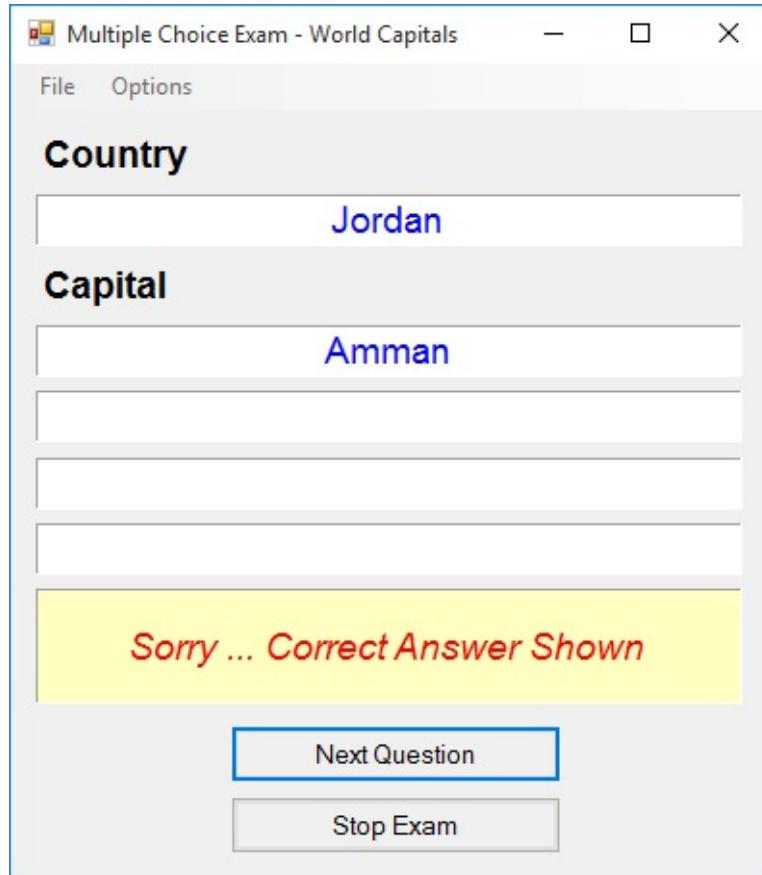
Your entries will be different since the exam questions and possible answers are selected randomly. This question asks for the capital of **Spain** and four possible answers are listed. You click on your choice of capital. You will be told if you are correct or not and given the opportunity to answer another question. You are only given one chance to get the correct answer.

I know the capital of **Spain** is **Madrid**. When I click that selection, I see:

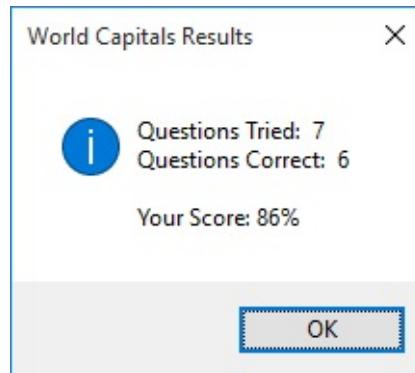


At this point, you have two choices – click **Next Question** to continue or click **Stop Exam** to stop. Try a few more questions.

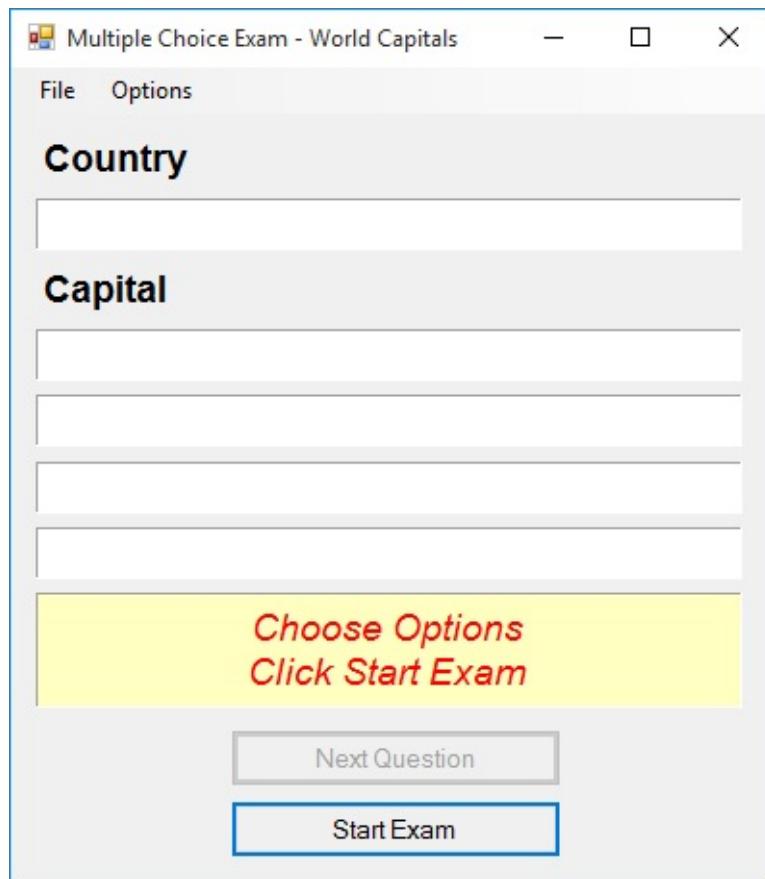
At some point, when you answer incorrectly, you will see a screen similar to this:

A screenshot of a Windows application window titled "Multiple Choice Exam - World Capitals". The window has standard minimize, maximize, and close buttons at the top right. A menu bar with "File" and "Options" is visible. The main area contains two input fields: one labeled "Country" containing "Jordan" and another labeled "Capital" containing "Amman". Below these fields is a yellow message box with the text "Sorry ... Correct Answer Shown". At the bottom are two buttons: "Next Question" (highlighted with a blue border) and "Stop Exam".

So, with an incorrect answer, you are told so and given the correct answer. Keep answering questions as long as you'd like. When you finally click **Stop Exam**, you will be shown a message box with the exam results. Mine for a short exam is:

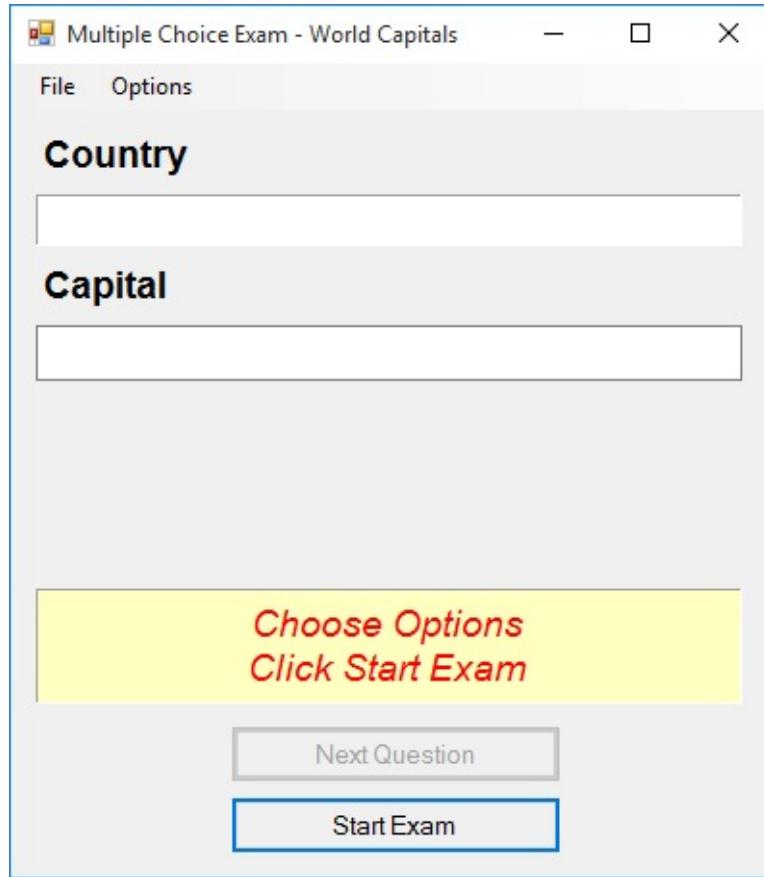


Click **OK** in the message box and your form returns to its initial configuration:



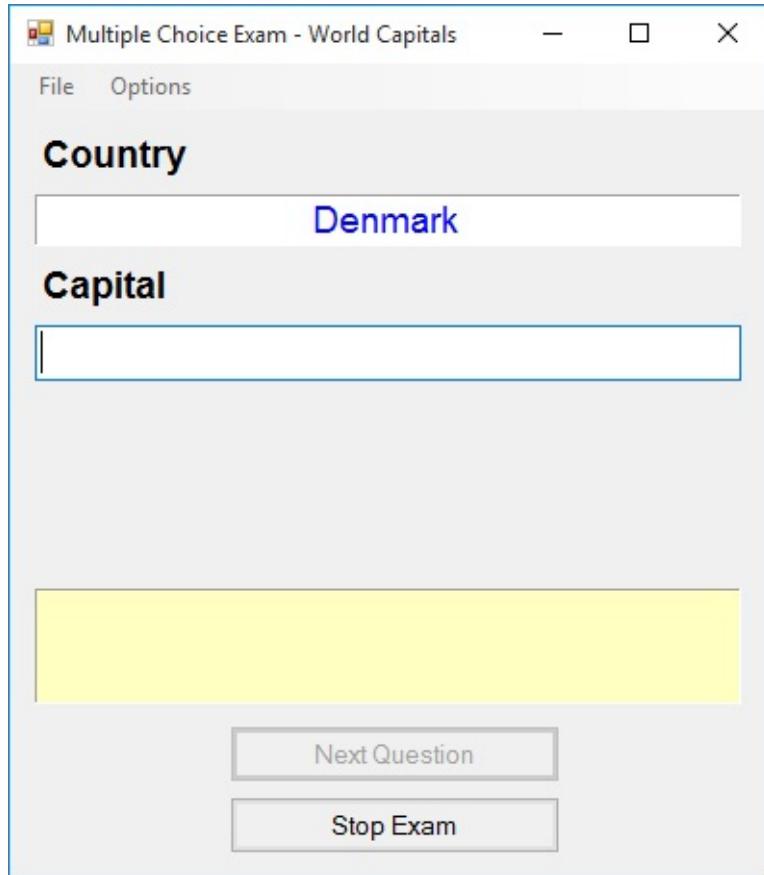
At this point, you can change any option and start a new exam, start a new exam with the same options or load a new exam file. Or, you can choose **Exit** in the **File** menu structure to stop the program.

Click the **Options** menu and choose **Type In Answer**. You will see:

A screenshot of a Windows application window titled "Multiple Choice Exam - World Capitals". The window has standard minimize, maximize, and close buttons at the top right. A menu bar with "File" and "Options" is visible. The main area contains two text input fields: one labeled "Country" and one labeled "Capital", each with a corresponding empty text box below it. Below these fields is a yellow rectangular button containing the text "Choose Options" and "Click Start Exam" in red. At the bottom of the window are two buttons: "Next Question" in a grey box and "Start Exam" in a blue box.

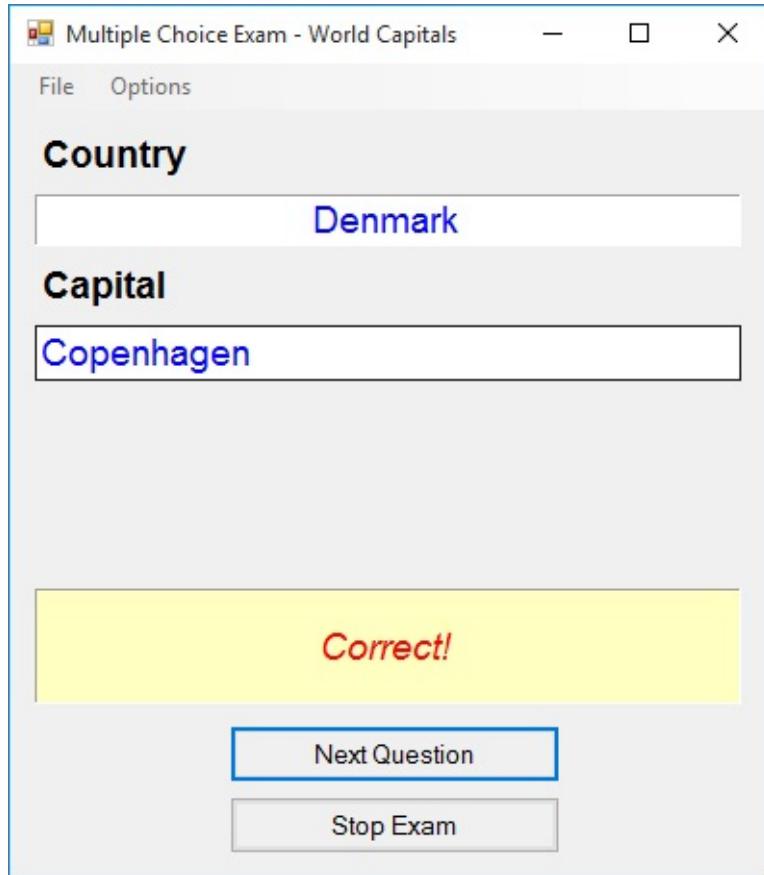
The form has reconfigured – the four multiple choice answer areas (label controls) have been replaced by a single text box control where your answer is typed. Click **Start Exam**.

The first question is displayed (again, yours will be different):



The capital of **Denmark** is **Copenhagen**. If you type **Copenhagen** in the text box area and press <Enter> you will be told this is a correct answer. The program allows your answers to be case-insensitive (we'll show you how to do this in the code design), so even if you type **copenhagen**, you are credited with a correct answer.

When I type **copenhagen** in the text box and press <Enter>, I see:



As mentioned, the answer is accepted and the ‘capitalization’ is corrected.

Now, let’s look at a really neat feature of the program. Many times, when typing answers, you might know the answer but not the correct spelling. This happens a lot with kids – could you spell **Copenhagen** when you were young? Rather than telling a user the answer is wrong, it would be nice to credit a user with the correct answer if the spelling is close. How do you do such magic, you ask? We’ll see in the code design section. For now, let’s just try it.

After getting credit for my **Denmark** question, the next question presented was (again, your question will be different, but try misspelling an answer):

Multiple Choice Exam - World Capitals

File Options

**Country**

**Capital**

I know the capital of **Egypt** is **Cairo**, but what if I mistakenly spell it as **caro**:

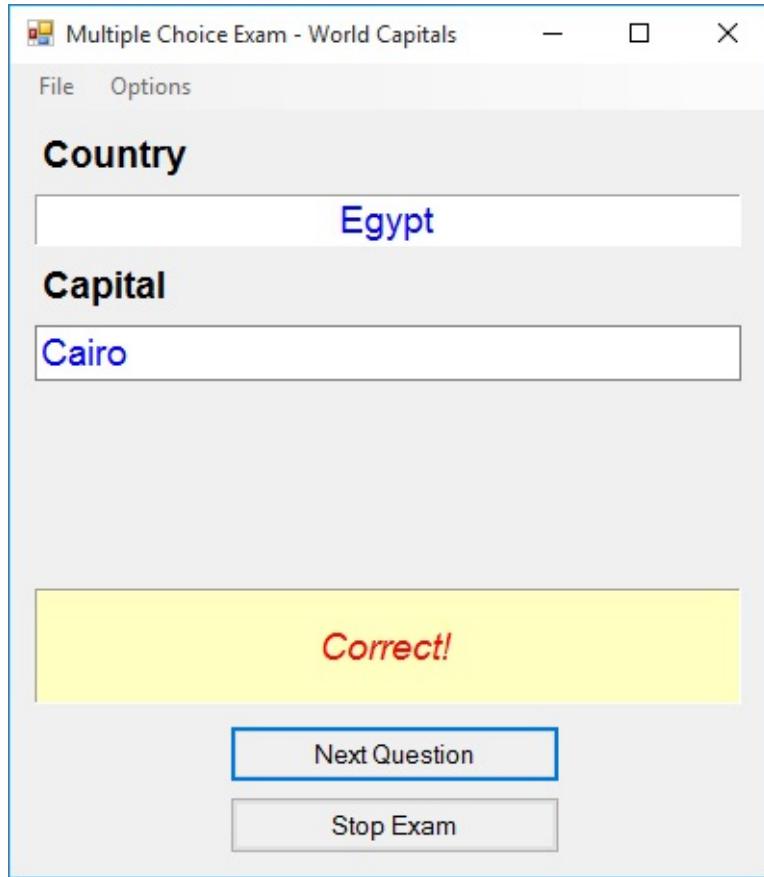
Multiple Choice Exam - World Capitals

File Options

**Country**

**Capital**

When I press <Enter>, I see:



So, even though I misspelled the word, I am given credit for a correct answer and shown the correct spelling. This little feature really helps alleviate user's frustration at not quite knowing how to spell an answer – this is especially useful with kids.

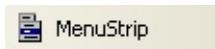
I think you see the idea of the program. Try as many questions, with as many different options, as you like. Maybe load in the **USCapitals.csv** file. When you are finally, finished choose **Exit** under the **File** menu to stop.

You will now build this project in several stages. We address **form design**. We discuss the controls used to build the form and establish initial properties. And, we address **code design** in detail. We cover opening and loading exam files, establishing and switching configurations for different options, validation of both multiple choice and typed in answers and presenting results. And, we show how we did the trick to check for “close spelling”?

There are two new controls in this project – the menu strip and the open file dialog box. We review these controls and their use before starting the project.

# MenuStrip Control

**In Toolbox:**



**Below Form (Default Properties):**



As the applications you build become more and more detailed, with more features for the user, you will need some way to organize those features. A **menu** provides such organization. Menus are a part of most applications. They provide ways to navigate within an application and access desired features. Menus are easily incorporated into Visual C# programs using the **MenuStrip** control.

The **MenuStrip** control has many features. It provides a quick and easy way to add menus, menu items and submenu elements to your Visual C# application. And it also has an editor to make any changes, deletions or additions you need. Modifications to the menu structure are also possible at run-time.

A good way to think about elements of a menu structure is to consider them as a hierarchical list of button-type controls that only appear when pulled down from the menu. Each element in the menu structure is an object of type **ToolStripMenuItem**. When you click on a menu item, some action is taken. Like buttons, menu items are named, have properties and a **Click** event. The best way to learn to use the MenuStrip control is to build an example menu. The menu structure we will build is:

File	Format
New	Bold
Open	Italic
Save	Underline
_____	Size
Exit	Small

Medium

Large

The level of indentation indicates position of a menu item within the hierarchy. For example, **New** is a sub-element of the **File** menu. The line under **Save** in the **File** menu is a separator bar (separates menu items).

With this structure, at run-time, the menu would display:

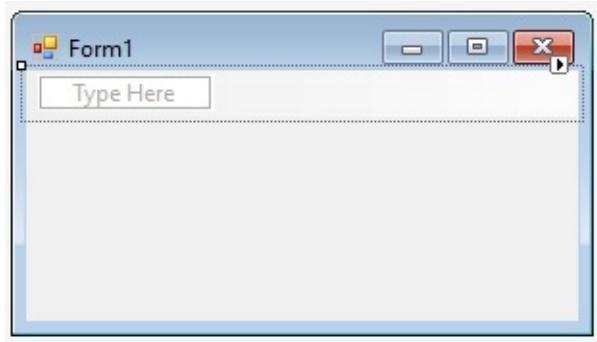
**File**      **Format**

The sub-menus appear when one of these ‘top’ level menu items is selected. Note the **Size** sub-menu under **Format** has another level of hierarchy. It is good practice to not use more than two levels in menus. Each menu element will have a **Click** event associated with it.

When designing your menus, follow formats used by standard Windows applications. For example, if your application works with files, the first heading should be **File** and the last element in the sub-menu under **File** should be **Exit**. If your application uses formatting features, there should be an **Format** heading. By doing this, you insure your user will be comfortable with your application. Of course, there will be times your application has unique features that do not fit a ‘standard’ menu item. In such cases, choose headings that accurately describe such unique features.

We’re ready to use the **MenuStrip** control. For this little example, we’ll start a new project with a blank form. Drag the **MenuStrip** control to the form. The control will be placed in the ‘tray’ area below the form. Set a single property for the menu control – **Name** it **mnuMain**.

Now, select the menu control (make it the active control) and your form will display the first menu element:



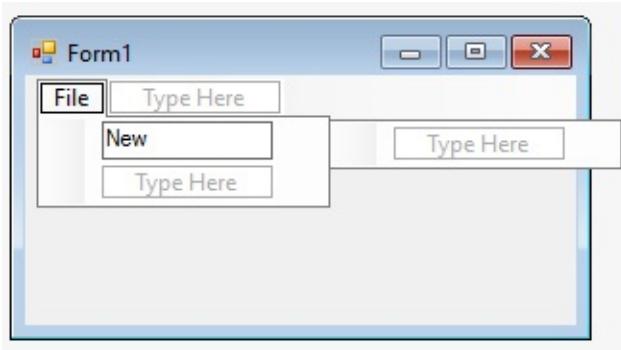
In the displayed box, type the first main heading (**File** in this case). This establishes the first property (**Text**) for the menu element. Once done typing this heading, you can set other properties. What I usually do, though, is type all the **Text** properties, building the menu structure. Then, I return to each item and assign properties. You choose which method you like best: set properties as you go or build structure, then set properties. We'll talk more about properties after building the menu structure.

After typing the first heading, the form will look like this:



At this point, the menu control will let you type either the first sub-menu heading under the **File** heading or move to the next heading. If you press <Enter> after typing an entry, the cursor will stay within the menu structure (go to a sub-heading).

Type the first sub-menu heading of **New**:

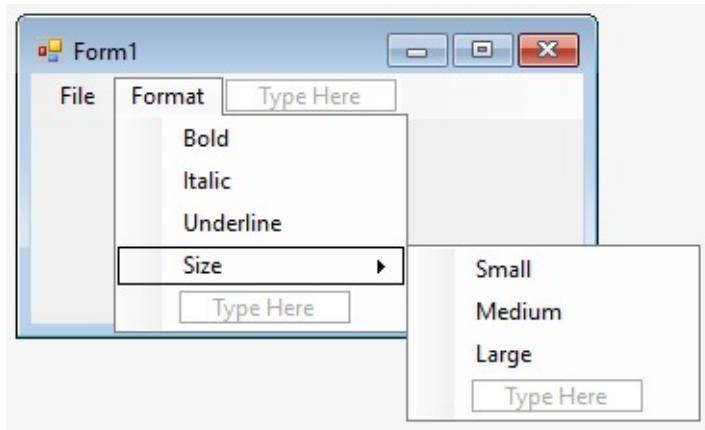


The menu control suggests locations for the next heading.

I think you see how the **MenuStrip** control works. It's really easy. Just type headings (**Text** properties) where they belong in the menu hierarchy. At any time, you can click on an "already entered" value to change it. Once you've built a few menu structures, you will see how easy and intuitive this control is to use. The best hint I can give is just click where you want to type and magical boxes will appear. Click on the form or any other control to stop menu editing.

**Separator bars** are used in menu structures to delineate like groups of menu items. There are two ways to enter a 'separator bar' in a menu. First, you can simply type a **Text** property of a single hyphen (-). Or, right-click on the element below the desired separator bar location and choose **Insert** then **Separator**.

After typing the remaining elements of our example menu (with just the captions), the form will look like this:



Our menu framework is built, but we must still set properties. Each element in the menu structure has several properties that can be established:

<b>Property</b>	<b>Description</b>
Name	Each menu item must have a name, even separator bars! The prefix <b>mnu</b> is used to name menu items. Sub-menu item names usually refer back to main menu headings. For example, if the menu item <b>New</b> is under the main heading <b>File</b> (name <b>mnuFile</b> ) menu, use the name <b>mnuFileNew</b>
Text	Caption appearing on menu item (these captions are what the user sees). Use a hyphen (-) for a separator.
Image	Optional image next to menu item (we will not use any images).
Enabled	If True, the menu item can be selected. If False, the menu item is grayed and cannot be selected.
Checked	If True, a check mark appears next to the menu item.
CheckOnClick	If True, when clicked the Checked status of the button will be toggled (changed).

We've set the **Text** properties already. At this point, you need to go back and select each menu item and set desired properties. To do this, make the appropriate menu item active by clicking on it (or select the menu item in the properties window). Then, set properties in the properties window using the usual process. Do this for each menu item. Yes, I realize there's a lot to do, but then a menu structure does a lot for your application.

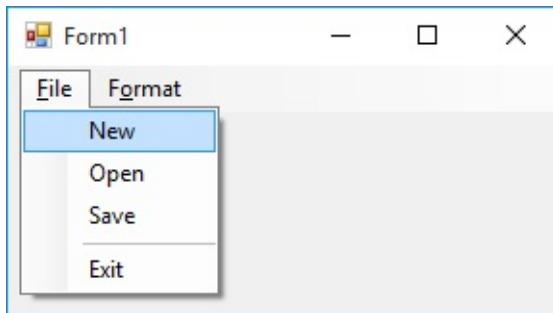
For our example menu structure, I used these properties:

<b>Text</b>	<b>Name</b>
File	mnuFile
New	mnuFileNew
Open	mnuFileOpen
Save	mnuFileSave
-	mnuFileBar (a <b>ToolStripSeparator</b> )
Exit	mnuFileExit

Format	mnuFmt
Bold	mnuFmtBold
Italic	mnuFmtItalic
Underline	mnuFmtUnderline
Size	mnuFmtSize
Small	mnuFmtSizeSmall
Medium	mnuFmtSizeMedium
Large	mnuFmtSizeLarge

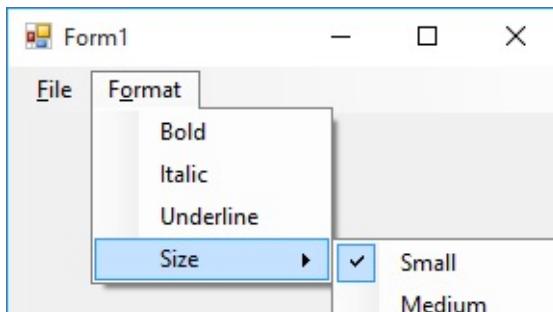
In particular, notice how the naming convention makes it easy to identify the purpose of each menu item. Notice, too, we have to name the separator bar.

When I run this little sample, and click the **File** menu, I see:



Notice the separator bar.

When I click the **Format** menu, then the **Size** sub-menu, I see:



Notice I assigned a **Checked** property of **True** to the **Small** element under the **Size** sub-menu. This indicates the size of the default font in the application using this structure. If you did not select the **CheckOnClick** property, be aware you

need to control when this checkmark appears and disappears. It does not work like the check in the check box control. That is, it may not automatically appear when a menu item is clicked.

You might think we're finally done, but we're not. We still need to write code for each menu item's **Click** event (the **default** event). Yes, even more work is needed. The Click event method for a menu item is found in the same manner any other event method is located. Select the menu item in the properties window, click the **Events** button, and double-click the listed **Click** event. The event framework will appear and code can be entered. Alternately, double-click the desired menu item and the event method will appear.

# OpenFileDialog Control

**In Toolbox:**



**Below Form (Default Properties):**



When we opened the multiple choice exam file in the preview, an '**Open File**' dialog box appeared to select the file. This dialog is similar to the box used in nearly every Windows application for opening files. Visual C# lets us use this same interface in our applications via the **OpenFileDialog** control. This control is one of a suite of dialog controls we can add to our applications. There are also dialog controls to save files, change fonts, change colors, and perform printing operations.

What we learn here is not just limited to opening files for the multiple choice exam. There are many times in application development where we will need a file name from a user. Applications often require data files, initialization files, configuration files, sound files and other graphic files. The **OpenFileDialog** control will also be useful in these cases.

## OpenFileDialog Properties:

<b>Name</b>	Gets or sets the name of the open file dialog (I usually name this control <b>dlgOpen</b> ).
<b>AddExtension</b>	Gets or sets a value indicating whether the dialog box automatically adds an extension to a file name if the user omits the extension.
<b>CheckFileExists</b>	Gets or sets a value indicating whether the dialog box displays a warning if the user specifies a file name that does not exist.
<b>CheckPathExists</b>	Gets or sets a value indicating whether the dialog box displays a warning if the user specifies a path

	that does not exist.
<b>DefaultExt</b>	Gets or sets the default file extension.
<b>FileName</b>	Gets or sets a string containing the file name selected in the file dialog box.
<b>Filter</b>	Gets or sets the current file name filter string, which determines the choices that appear in "Files of type" box.
<b>FilterIndex</b>	Gets or sets the index of the filter currently selected in the file dialog box.
<b>InitialDirectory</b>	Gets or sets the initial directory displayed by the file dialog box.
<b>Title</b>	Gets or sets the file dialog box title.

### OpenFileDialog Methods:

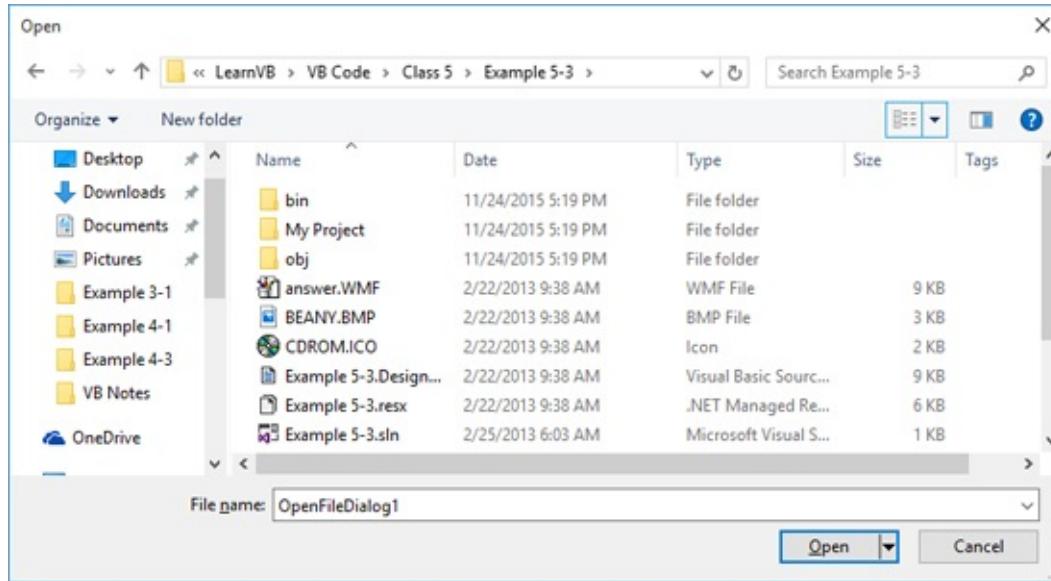
<b>ShowDialog</b>	Displays the dialog box. Returned value indicates which button was clicked by user ( <b>OK</b> or <b>Cancel</b> ).
-------------------	--

To use the **OpenFileDialog** control, we add it to our application the same as any control. Since the OpenFileDialog control has no immediate user interface (you control when it appears), the control does not appear on the form at design time. Such Visual C# controls (the **Timer** control seen in Chapter 4 was a similar control) appear in a ‘tray’ below the form in the IDE Design window. Once added, we set a few properties. Then, we write code to make the dialog box appear when desired. The user then makes selections and closes the dialog box. At this point, we use the provided information for our tasks.

The **ShowDialog** method is used to display the **OpenFileDialog** control. For a control named **dlgOpen**, the appropriate code is:

```
dlgOpen.ShowDialog();
```

And the displayed dialog box is:



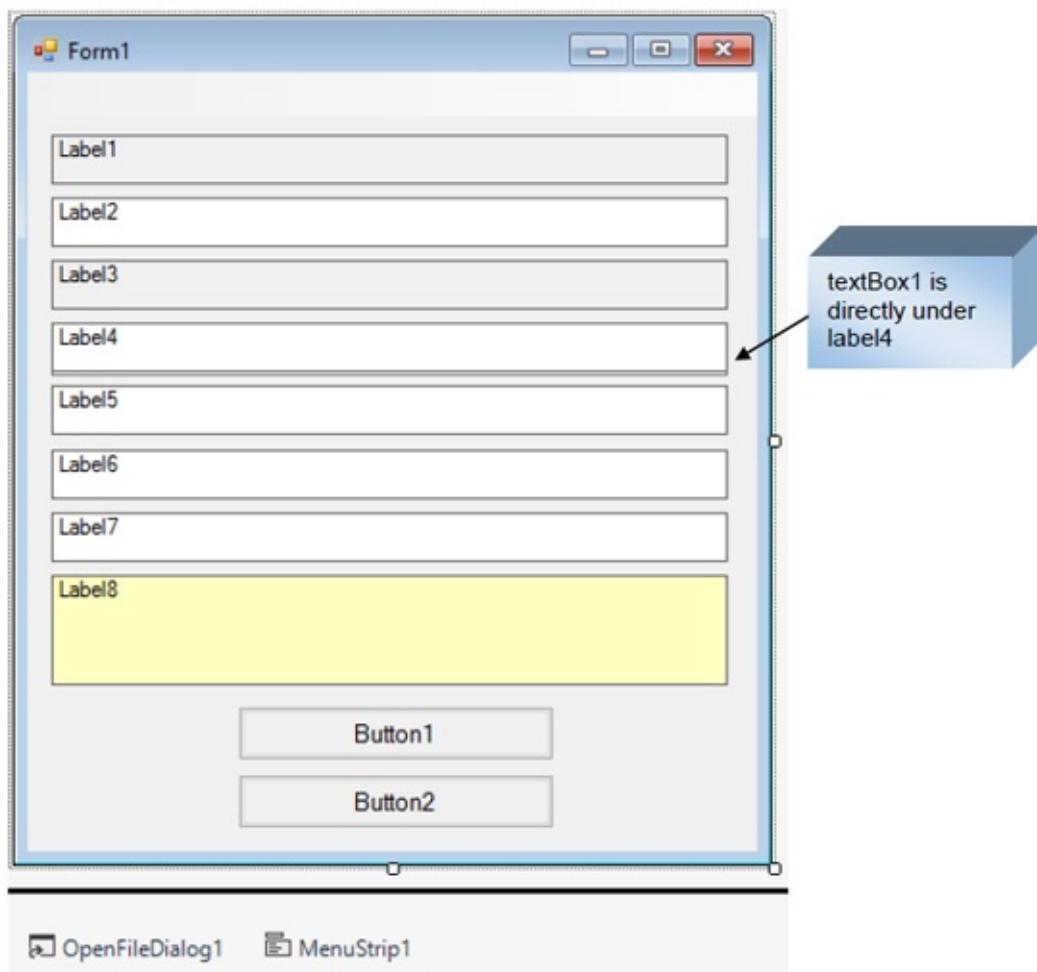
The user selects a file using the dialog control (or types a name in the **File name** box). The file type is selected from the **Files of type** box (values here set with the **Filter** property). Once selected, the **Open** button is clicked. **Cancel** can be clicked to cancel the open operation. The **ShowDialog** method returns the clicked button. It returns **DialogResult.OK** if **Open** is clicked and returns **DialogResult.Cancel** if **Cancel** is clicked. The nice thing about this control is that it can validate the file name before it is returned to the application. The **FileName** property contains the complete path to the selected file.

Typical use of **OpenFileDialog** control:

- Set the **Name**, **Filter**, and **Title** properties.
- Use **ShowDialog** method to display dialog box.
- Read **FileName** property to determine selected file

# Multiple Choice Exam Form Design

We can begin building the multiple choice exam project. Let's build the form. Start a new project in Visual C#. Place eight label controls (set **AutoSize** to **False** to allow resizing), one text box control, and two button controls on your form. Add an open file dialog control and menu strip control to the project. Resize and position controls so the form looks similar to this (I've temporarily set each label control's **BorderStyle** property to **FixedSingle** so you can see their size and location):



The text box control (**textBox1**) is directly under **label4** (it is the same size as the label control).

**label1** and **label3** are used for header information. **label2** is used to list the ‘given’ item. **label4**, **label5**, **label6** and **label7** are used to list the multiple choice answers. The large label is used for comments. The text box control is used to type-in answers. One button starts and stops the exams and one moves you from one question to the next. The open file dialog control is used to load an exam file, while the menu strip is used to open files, exit the program and set options.

Establish the following menu structure and set the properties:

MenuStrip **mnuMain** structure:

File	Options
Open	Header1
_____	Header2
Exit	_____
	Multiple Choice Answers
	Type In Answers

MenuStrip **mnuMain** properties:

Text	Name
File	mnuFile
Open	mnuFileOpen
-	(you choose, a <b>ToolStripSeparator</b> )
Exit	mnuFileExit
Options	mnuOptions
Header1	mnuOptionsHeader1 ( <b>Checked = True</b> )
Header2	mnuOptionsHeader2
-	(you choose, a <b>ToolStripSeparator</b> )
Multiple Choice Answers	mnuOptionsMC ( <b>Checked = True</b> )
Type In Answers	mnuOptionsType

The **Text** for the headers under **Options** will be established once an exam file is opened.

Set the other control properties using the properties window:

**Form1** Form:

<b>Property Name</b>	<b>Property Value</b>
Name	frmMultiple
BorderStyle	FixedSingle
StartPosition	CenterScreen
Text	Multiple Choice Exam – No File

**label1** Label:

<b>Property Name</b>	<b>Property Value</b>
Name	lblHeadGiven
Text	[Blank]
TextAlign	LeftCenter
AutoSize	False
Font	Arial, Bold, Size 14

**label2** Label:

<b>Property Name</b>	<b>Property Value</b>
Name	lblGiven
Text	[Blank]
TextAlign	MiddleCenter
AutoSize	False
BorderStyle	Fixed3D
BackColor	White
ForeColor	Blue
Font	Arial, Size 14

**label3** Label:

<b>Property Name</b>	<b>Property Value</b>
Name	lblHeadAnswer

Text	[Blank]
TextAlign	LeftCenter
AutoSize	False
Font	Arial, Bold, Size 14

**label4** Label:

<b>Property Name</b>	<b>Property Value</b>
Name	lblAnswer1
Text	[Blank]
TextAlign	MiddleCenter
AutoSize	False
BorderStyle	Fixed3D
BackColor	White
ForeColor	Blue
Font	Arial, Size 14

**label5** Label:

<b>Property Name</b>	<b>Property Value</b>
Name	lblAnswer2
Text	[Blank]
TextAlign	MiddleCenter
AutoSize	False
BorderStyle	Fixed3D
BackColor	White
ForeColor	Blue
Font	Arial, Size 14

**label6** Label:

<b>Property Name</b>	<b>Property Value</b>
Name	lblAnswer3
Text	[Blank]
TextAlign	MiddleCenter

AutoSize	False
BorderStyle	Fixed3D
BackColor	White
ForeColor	Blue
Font	Arial, Size 14

**label7 Label:**

<b>Property Name</b>	<b>Property Value</b>
Name	lblAnswer4
Text	[Blank]
TextAlign	MiddleCenter
AutoSize	False
BorderStyle	Fixed3D
BackColor	White
ForeColor	Blue
Font	Arial, Size 14

**label8 Label:**

<b>Property Name</b>	<b>Property Value</b>
Name	lblComment
Text	[Blank]
TextAlign	MiddleCenter
AutoSize	False
BorderStyle	Fixed3D
BackColor	Yellow
ForeColor	Red
Font	Arial, Italic, Size 14

**textBox1 Text Box:**

<b>Property Name</b>	<b>Property Value</b>
Name	txtAnswer
Text	[Blank]

TextAlign	Left
ReadOnly	True
BackColor	White
ForeColor	Blue
Font	Arial, Size 14

This control is hidden behind **lblAnswer1**. To see it, right-click **lblAnswer1** and choose **Send to Back**. Reverse the process to put the text box behind the label control again.

#### **button1** Button:

<b>Property Name</b>	<b>Property Value</b>
Name	btnNext
Text	Next Question
Font Size	10

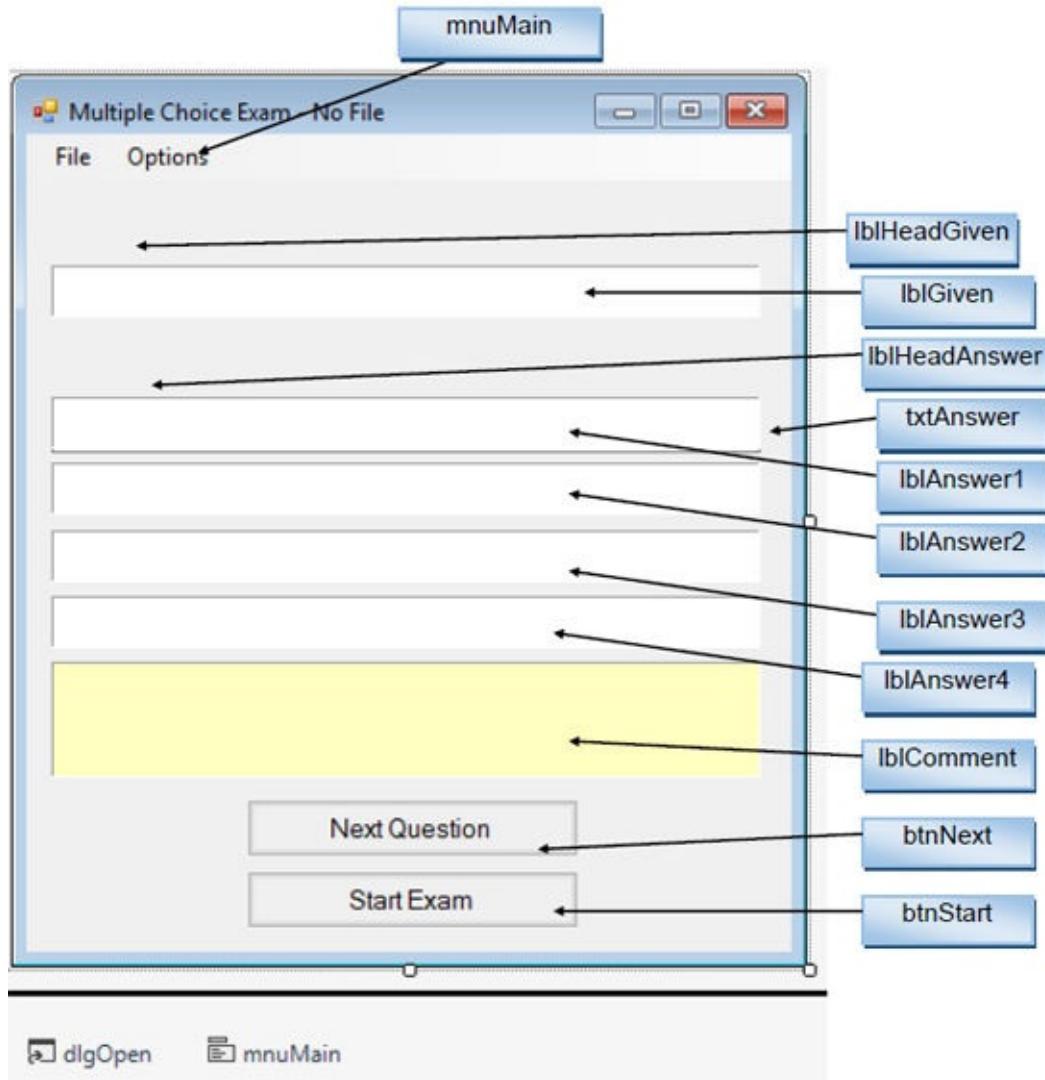
#### **button2** Button:

<b>Property Name</b>	<b>Property Value</b>
Name	btnStart
Text	Start Exam
Font Size	10

#### **openFileDialog1** Open File Dialog:

<b>Property Name</b>	<b>Property Value</b>
Name	dlgOpen
Title	Open Multiple Choice Exam File
Default Extension	csv
Filter	Data Files (*.csv) *.csv

When done setting properties, my form looks like this:



Again, note **txtAnswer** is under **lblAnswer1**. This completes the form design.

We will begin writing code for the application. We will write the code in several steps. As a first step, we write the code that gets the program in initial mode to allow opening an exam file.

# Form Design – Initialization

In the multiple choice exam project, the user opens an exam file, chooses options and proceeds to take a test. Once done, the results are presented. At that point, other options can be selected or other files used. We want to step the user through program use – this minimizes the possibility of errors. When the program starts, the first thing a user must do is open a file. We need to make sure the interface only allows access to the **File** menu option.

When the program first loads, these steps are taken:

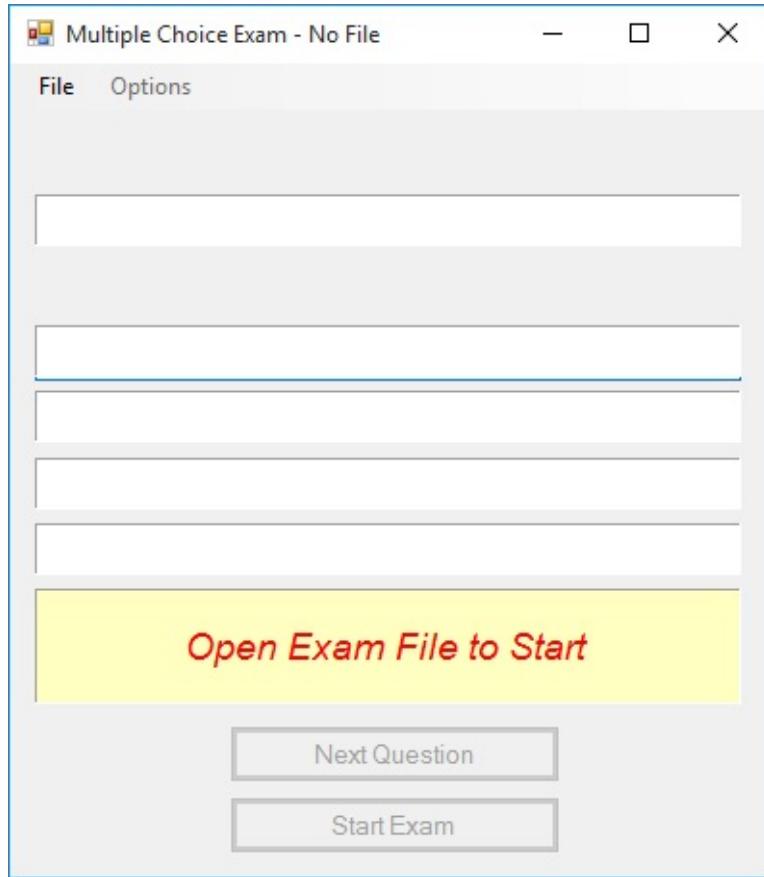
- Disable **btnStart**.
- Disable **btnNext**.
- Disable **mnuOptions**.
- Change **Text** property of **lblComment** to **Open Exam File to Start**.

This initialization code is placed the **frmMultiple Load** event. The code is:

```
private void frmMultiple_Load(object sender, EventArgs e)
{
    // initialize form
    btnStart.Enabled = false;
    btnNext.Enabled = false;
    mnuOptions.Enabled = false;
    lblComment.Text = "Open Exam File to Start";
}
```

Add this code to the project.

Save and run the project. The form will appear as:



Notice at this point, the only thing a user can do is select the **File** menu option, where a file can be opened (**Open**) or the project can be stopped (**Exit**). Let's write code for both options.

The **mnuFileExit Click** event is simple. The code is:

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

Add this method to the project.

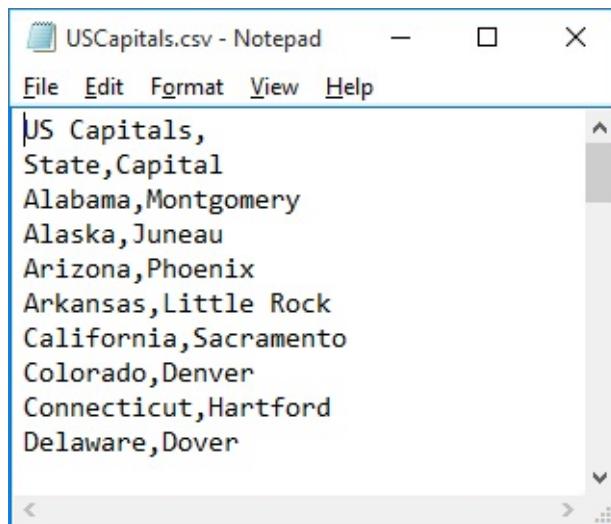
The code for the **mnuFileOpen Click** event is far more involved. We'll spend a lot of time talking about it, building it in stages. We discuss file format, ways to generate exam files, how to open exam files, how to read information from the exam files and how to avoid errors when opening and reading files.

# Code Design – Exam File Format

The files used to store information for multiple choice exams have a specific format – you need to insure any files you generate conform to this standard. The files used are called **sequential files**, indicating they are just line after line of information.

To generate a file, you need to have two lists of matching terms (in our sample files, the lists are states and capitals and countries and capitals). Each term should have an identifying header. And each file (exam) should have a title. Once you have this information, the first line of the file is the exam title, followed by a comma (,). The second line is the two headers describing the listed terms, separated by a comma. Subsequent lines are the pairs of terms, each pair separated by a comma – the program will allow up to 100 matching pairs.

Using Windows Notepad, open the **USCapitals.csv** file in the **HomeVCS\HomeVCS Projects\Multiple Choice** folder. When Notepad opens, choose **Open** under the **File** menu. Then, choose **All Files** under **Files of Type** in the **Open** dialog box (by default, only files with **txt** extensions are shown). Choose the file and click **Open**. Note the format:



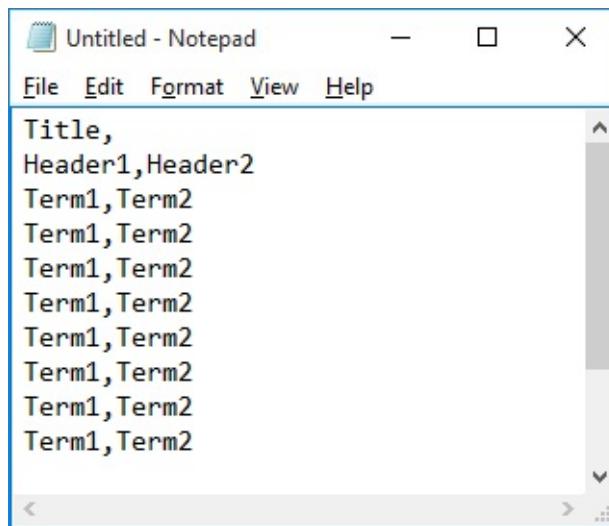
The first line shows the title (**US Capitals**) with an ending comma (don't forget this comma when generating a file). The second line are the headers (**State** and **Capital**), separated by a comma. Following the headers are the 50 pairs of states

and capitals, separated by commas. All files must be in this form. Let's see how you can generate such files.

# Code Design – Generating Exam Files

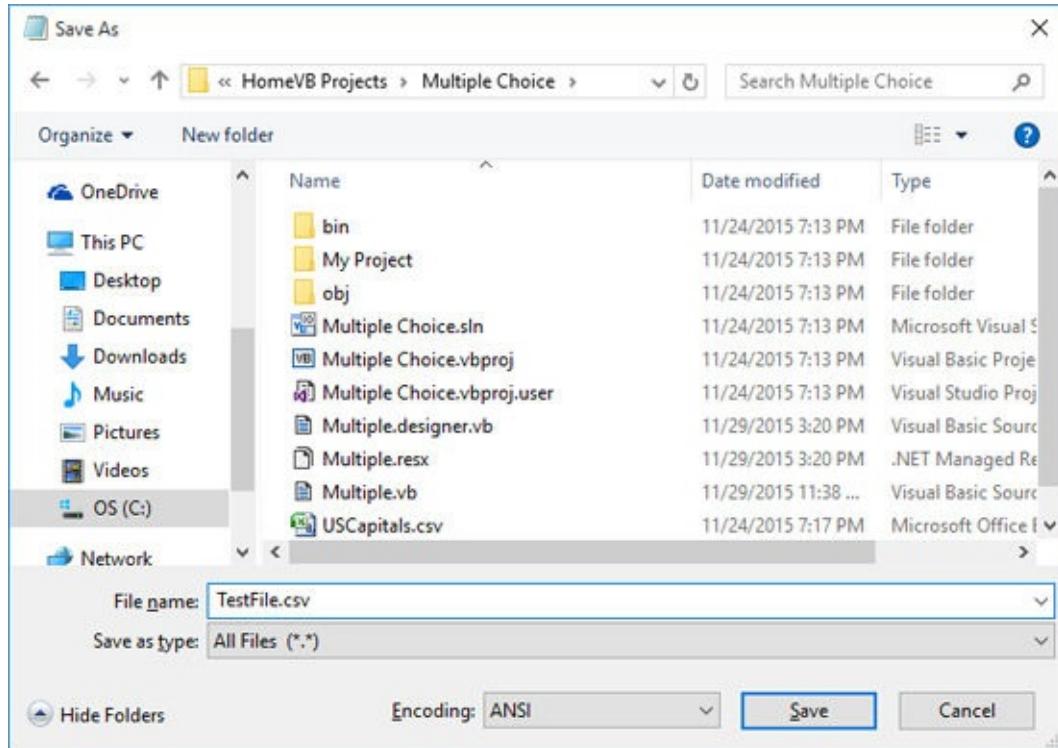
You will eventually want to use exam files other than the two examples included. Hence, you need to know how to generate such files. First, you need to have your list of terms. Choose a title and the two headers. Once you have this information, you need to save it in the proper file format with a **csv** extension. The extension **csv** stands for **comma separated values** – that's why we saw all the commas in the **USCapitals.csv** file.

One way to generate an exam file is use a simple word processor such as the Windows **Notepad**. Start a new file and simply type in the information in the proper format like this:



There are very few restrictions on the information you can use in an exam file. Entries can be letters, numbers, spaces and nearly any “typeable” character. A major restriction is that the entries can have no commas. Since we use a comma as a **delimiter** (the character that separates one term from the other), any other comma in a line would result in an error.

When the file is complete, save it with a **csv** extension. Notepad, by default, will want to save your file with a **txt** extension. To bypass this, when the **Save** dialog opens, choose **All Files** in the **Save as type** drop-down as shown:

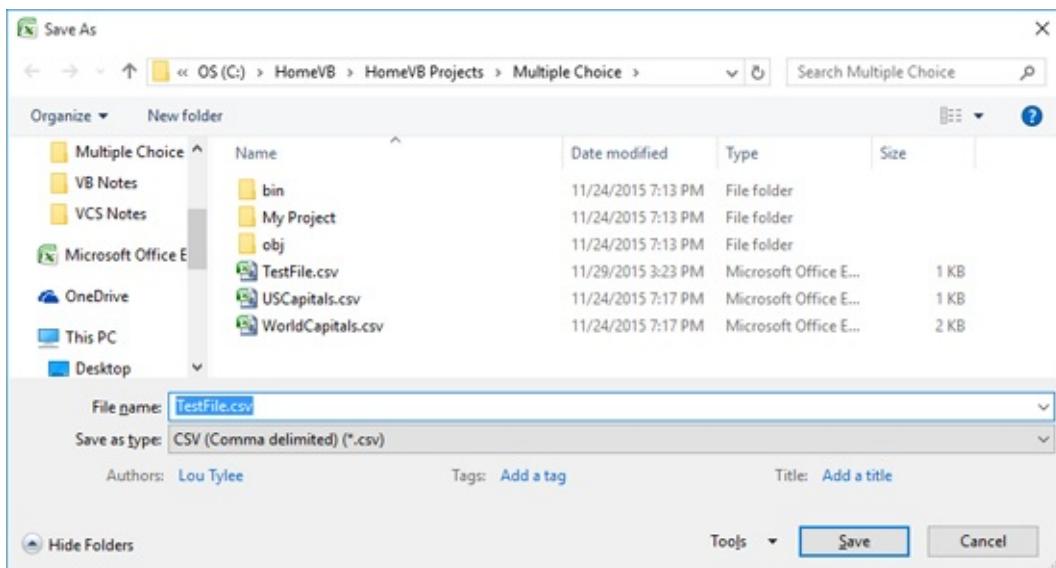


Type your file name with the **csv** extension and click **Save**.

A spreadsheet program such as Microsoft's **Excel** can also be used to generate an exam file. To do this, start Excel. A blank worksheet should appear. Type the information in the spreadsheet cells something like this:

	A	B	C	D	E	F
1	Title					
2	Header1	Header2				
3	Term1	Term2				
4	Term1	Term2				
5	Term1	Term2				
6	Term1	Term2				
7	Term1	Term2				
8	Term1	Term2				
9	Term1	Term2				
10	Term1	Term2				
11	Term1	Term2				
12	Term1	Term2				
13	Term1	Term2				

Once you've entered all your terms, choose **File**, then click **Save As**. When the **Save As** window appears, choose **CSV** under **Save as type** as shown below:



The information in the spreadsheet will be saved as a comma-separated file in the format used by the multiple choice exam project.

You can also open the example exam files in Excel. Choose **Open** under the **File** menu, then choose one of the samples – you have to set **Files of type** to **All Files** when selecting the file. Here's the **WorldCapitals.csv** file in Excel:

	A	B	C	D	E	F	G
1	World Capitals						
2	Country	Capital					
3	Afghanist	Kabul					
4	Albania	Tirane					
5	Australia	Canberra					
6	Austria	Vienna					
7	Banglades	Dacca					
8	Barbados	Bridgetown					
9	Belgium	Brussels					
10	Bulgaria	Sofia					
11	Burma	Rangoon					
12	Cambodia	Phnom Penh					
13	China	Peking					
14	Czechoslo	Prague					
15	Denmark	Copenhagen					
16	Egypt	Cairo					

Now, let's see how to open exam files in our project.

# Code Design – Opening an Exam File

When a user clicks the **Open** entry in the **File** menu, the project should ask the user for an exam file. When that file's name is provided, the program will open the file, read in the information and place that information in the proper program variables. Once this is done, the user can begin to take a quiz. All of this happens in the **mnuFileOpen Click** method. We will begin building that method. As a first step, let's look at the step of obtaining a file name from the user and opening the file for input.

The user needs to tell the program which exam file they want to use. The open file dialog control will be used to provide this name. The steps to do this are:

- Show the open file dialog box.
- If user picks a file name and clicks **Open**, proceed to open file and obtain values.
- If **Cancel** is clicked, do nothing.

We will look at just opening the file for now. Assuming we know the name of the exam file, it is opened using a **StreamReader** object. This class uses the Visual C# **IO** (input/output) namespace, so for any application using file input and output, you will need to add this line to the code:

```
using System.IO;
```

You will see similar **using** statements at the very top of the code window (you may need to expand the **using Directives** listing). These statements load in needed system utilities. Type it there. If you omit the line and try to use any file input/output functions you will receive an error message something like:

**The type or namespace name 'StreamReader' could not be found (are you missing a using directive or an assembly reference?)**

The syntax for opening a sequential file for input is:

```
StreamReader inputFile = new StreamReader(fileName);
```

where **fileName** is a complete path to the file and **inputFile** is the returned file object.

Once opened, we can read information from the file. We will discuss how to do that next. When all values have been read from the sequential file, it is closed using:

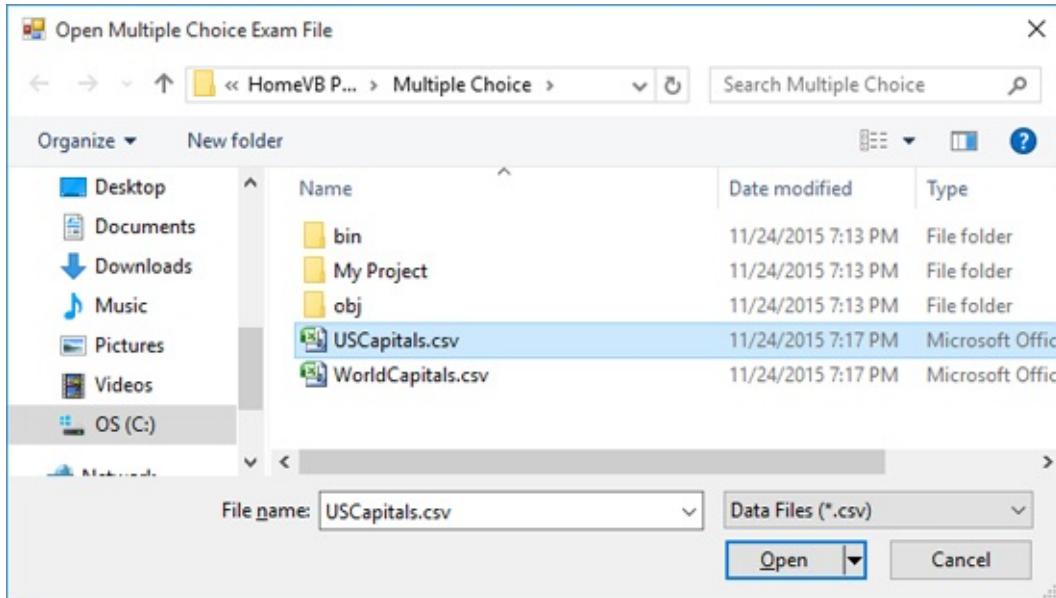
```
inputFile.Close();
```

Let's make sure we can open and close an exam file. The code in the **mnuFileOpen Click** method that accomplishes the above tasks is:

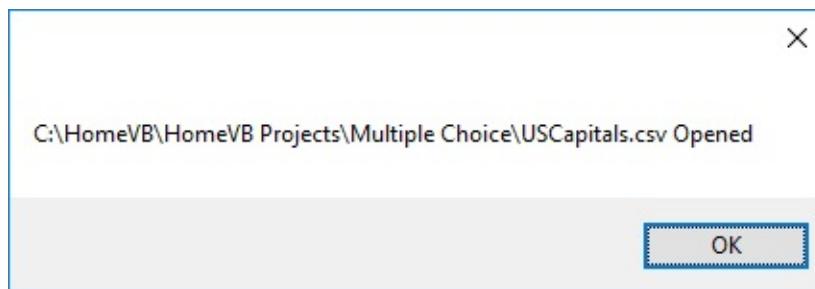
```
private void mnuFileOpen_Click(object sender, EventArgs e)
{
    // Open exam file
    if (dlgOpen.ShowDialog() == DialogResult.OK)
    {
        StreamReader inputFile = new StreamReader(dlgOpen.FileName);
        MessageBox.Show(dlgOpen.FileName + " Opened");
        inputFile.Close();
    }
}
```

We use a temporary message box to tell us if the file opens successfully:

Save and run the project. Choose the **Open** option under the **File** menu. You should see an open file dialog box with the properties set in **Form Design**:



Select an exam file and click **Open** to open the file. When I choose the example **USCapitals.csv** file, the message box I obtain is:



Try opening the other example file. Make sure the **Cancel** option in the open file dialog works properly, meaning nothing changes in the application if **Cancel** is selected. Stop the project and delete the line with the message box – we will no longer need that.

# Code Design – Reading an Exam File

Once an exam file is open, we can read in the information from the file, line-by-line, and obtain needed program variables. Let's first declare those variables. The file will provide us with the exam title (**examTitle**), two headers (**header1**, **header2**) and lists of exam terms (**term1**, **term2**). We will use **string** type variables for all this information (the term lists will be stored in arrays). We will also need an **int** type variable (**numberTerms**) to know how many items are in the lists. Add these variable declarations as form level variables:

```
string examTitle, header1, header2;  
int numberTerms;  
string[] term1 = new string[100];  
string[] term2 = new string[100];
```

We have arbitrarily set the limit on list length to be 100.

The steps to follow after opening an exam file are:

- Read in first line, obtain **examTitle**.
- Read in second line, obtain **header1** and **header2**
- Initialize **numberTerms** to 0.
- Increment **numberTerms**, read in **term1[numberTerms – 1]** and **term2[numberTerms – 1]**
- Continue reading lines until end of file is reached.

Note the term lists are stored in 0-based arrays (meaning the indices start at **0** and end at **numberTerms - 1**).

Let's see how to read the lines and get the needed variables. To read an entire line from a file opened as **inputFile**, use the **ReadLine** method:

```
myLine = inputFile.ReadLine();
```

where **myLine** will be the line represented as a **string** data type. In the exam

file, this line (except for the first line) will have one variable, a comma, then another variable. To obtain the individual variables, we need to ‘parse’ the line. This means we will identify where the comma is in the line then extract one variable to the left of the comma and another variable to the right of the comma. This parsing is done with various string functions.

To determine the location of the comma in **myLine**, we use the **IndexOf** method we have seen before. In the expression:

```
cl = myLine.IndexOf(",");
```

The **int** variable **cl** will tell us which character in **myLine** is a comma. The characters of **myLine** are numbered from **0** to **myLine.Length - 1**, where the **Length** property is the number of characters in **myLine**. As an example, say **myLine** is given by:

```
myLine ="First,Second";
```

Note **myLine.Length** is **12**. If we apply the above **IndexOf** method to this line, we will find **cl** is **5** (remember the first character is index **0**, not **1**).

To extract the two variables from this line, we use the **Substring** method. This function allows you to extract substrings from a string. You need to specify the source string (**myLine**, in this case), the starting position (**start**) and the number of characters (**number**) to extract. The resulting **mySubstring** is obtained using:

```
mySubstring = myLine.Substring(start, number);
```

For multiple choice exam files, to extract the characters to the left of the comma (located at **cl**) in **myLine**, we start at character **0** and extract **cl** characters, or:

```
leftString = myLine.Substring(0, cl);
```

The string to the right of the comma starts at character **cl + 1** and is **myLine.Length - cl - 1** characters in length:

```
rightString = myLine.Substring(cl + 1, myLine.Length - cl - 1);
```

To convince you that this works, let's return to the example with:

```
myLine = "First,Second";
```

where recall **c1** is 5 and **myLine.Length** is 12. Using the **leftString** relation, we see:

```
leftString = myLine.Substring(0, 5);
```

Starting at the first character (character index 0) and extracting five characters, we get:

```
leftString = "First";
```

Success. Now, using the **rightString** relation, we see:

```
rightString = myLine.Substring(5 + 1, 12 - 5 - 1);  
rightString = myLine.Substring(6, 6);
```

Starting at the 7<sup>th</sup> character (character index 6) and extracting six characters, we get:

```
rightString = "Second";
```

It works!!

All we need to know now is how to determine when we've reached the end of the exam file, so we can close the file and continue. After each line is read, we call the **Peek** method of the **StreamReader** object. The **Peek** method reads the next character in the file without changing the place that we are currently reading. If we have reached the end of the file, **Peek** returns -1.

We can now write the code to implement the steps to read and establish the variable values. The modified **mnuFileOpen Click** method (changes are shaded) is:

```
private void mnuFileOpen_Click(object sender, EventArgs e)
```

```

{
    string myLine;
    // Open exam file
    if (dlgOpen.ShowDialog() == DialogResult.OK)
    {
        StreamReader inputFile = new
        StreamReader(dlgOpen.FileName);
        myLine = inputFile.ReadLine();
        examTitle = ParseLeft(myLine);
        myLine = inputFile.ReadLine();
        header1 = ParseLeft(myLine);
        header2 = ParseRight(myLine);
        numberTerms = 0;
        do
        {
            numberTerms++;
            myLine = inputFile.ReadLine();
            term1[numberTerms - 1] = ParseLeft(myLine);
            term2[numberTerms - 1] = ParseRight(myLine);
        }
        while (inputFile.Peek() != -1);
        inputFile.Close();
    }
}

```

This code uses two general methods to parse the left (**ParseLeft**) and right (**ParseRight**) portions of the input line. These methods used the **Substring** method:

```

private string ParseLeft(string s)
{
    int cl;
    // find comma

```

```

cl = s.IndexOf(",");
return (s.Substring(0, cl));
}

private string ParseRight(string s)
{
    int cl;
    // find comma
    cl = s.IndexOf(",");
    return (s.Substring(cl + 1, s.Length - cl - 1));
}

```

You should be able to see all the steps in the code – we read the title, read the headers, then read in each set of variables. Save and run the project. Open and process an exam file. Nothing exciting will happen. The code will just run and the interface won't change. If you want to see how the code works, review the use of the debugger (Chapter 3). Set a breakpoint on the **while (inputFile.Peek() != -1);** line. Then, rerun the project. Check values of all the variables as the code is processed.

Let's add the code that changes the form so it is ready to start an exam. The steps are (assuming an exam file has been read correctly):

- Establish **Text** property for form.
- Set **Text** properties for **mnuOptionHeader1** and **mnuOptionHeader2**.
- Set **Text** properties for **lblHeadGiven** and **lblHeadAnswer** (based on **Checked** properties of menu items under **Option** heading).
- Enable **btnStart**.
- Enable **mnuOptions**.
- Set **Text** property of **lblComment** to indicate the file is loaded.

The code for each of these steps also goes in the **mnuFileOpen Click** method. The changes are shaded:

```
private void mnuFileOpen_Click(object sender, EventArgs e)
```

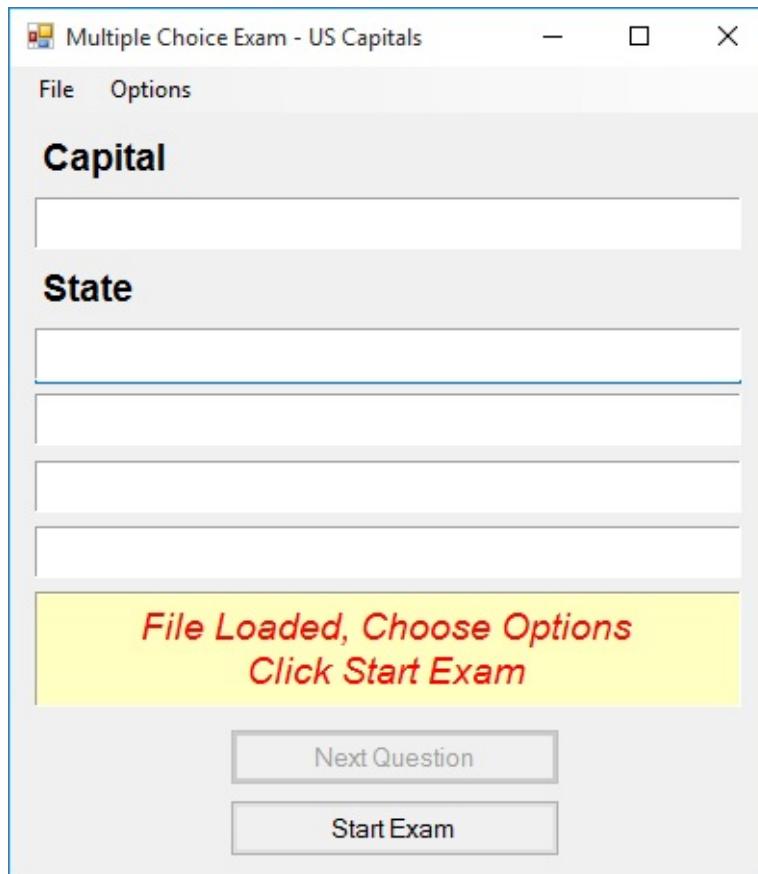
```
{  
    string myLine;  
    // Open exam file  
    if (dlgOpen.ShowDialog() == DialogResult.OK)  
    {  
        StreamReader inputFile = new  
StreamReader(dlgOpen.FileName);  
        myLine = inputFile.ReadLine();  
        examTitle = ParseLeft(myLine);  
        myLine = inputFile.ReadLine();  
        header1 = ParseLeft(myLine);  
        header2 = ParseRight(myLine);  
        numberTerms = 0;  
        do  
        {  
            numberTerms++;  
            myLine = inputFile.ReadLine();  
            term1[numberTerms - 1] = ParseLeft(myLine);  
            term2[numberTerms - 1] = ParseRight(myLine);  
        }  
        while (inputFile.Peek() != -1)  
        inputFile.Close();  
        // establish form title  
        this.Text = "Multiple Choice Exam - " + examTitle;  
        // set up menu items  
        mnuOptionsHeader1.Text = header1 + ", Given " + header2;  
        mnuOptionsHeader2.Text = header2 + ", Given " + header1;  
        if (mnuOptionsHeader1.Checked)  
        {  
            lblHeadGiven.Text = header2;  
            lblHeadAnswer.Text = header1;  
        }  
    }
```

```

else
{
    lblHeadGiven.Text = header1;
    lblHeadAnswer.Text = header2;
}
btnStart.Enabled = true;
mnuOptions.Enabled = true;
lblComment.Text = "File Loaded, Choose Options\r\nClick Start Exam";
}
}

```

Save and run the project. Load in an exam file. When I loaded **USCapitals.csv**, the form looks like this:



The form is ready for a multiple choice exam, where you name the **State**, given

the **Capital** (default options). At this point, the user can change options if desired, then click **Start Exam** to start an exam. We'll look at the code to do that soon, but first let's address the possibilities of errors when trying to open and read an exam file.

# Code Design - Error Trapping and Handling

When working with files in a Visual C#, things can go wrong. For example, in the multiple choice exam project, what if the selected exam file doesn't meet the specified format? Perhaps a comma is left off somewhere or a blank line is encountered. As written, the current project would stop with an error message to the user. Or what if the user selects a **csv** file that looks like an exam file, but isn't? Again, the program will stop when it realizes it can't process the information in the file.

Also in the multiple choice exam project, we must make sure we have a minimum of five entries in the exam files. This insures we can generate multiple choices (when that option is selected). And, recall we have set the maximum number of entries to 100. This limit can be changed by resetting the array limits, but, no matter what the value, we need to insure we don't read in more than the maximum number of values when reading an exam file.

How do we handle the possibility of errors? Checking the limits on the number of allowed terms is relatively simple. If we don't have the minimum number of entries, we can present a message box to the user. If we exceed the maximum number of entries, we can just stop reading values. Detecting that we are reading a file that is not in proper format is a little trickier. We don't know ahead of time just what kind of errors we might encounter in reading a file. To handle these general errors, we can use something called **run-time error trapping**.

With run-time error trapping, Visual C# recognizes an error has occurred and enables you to trap it and do something before your user sees a message they don't recognize or understand. Visual C# uses a structured approach to trapping and handling errors. In its simplest form (all we need here), the structure is referred to as a **try/catch** block. And the annotated syntax for using this block is:

```
try  
{  
    'here is code you try where some kind of error may occur
```

```
    .  
}  
catch  
{  
    ‘if any error occurs, process this code  
    .  
}
```

This code works from the top, down. It ‘tries’ the code between **try** and the **catch** statement. If no error is encountered, the program will continue after the **try/catch** block. If an error occurs, the program will process the code in the **catch** statement. In our multiple choice exam project, we will tell the user (using a message box) that the file is not acceptable and give them another chance to open a file. This is far preferable to the program just stopping, as it would now.

Here is the modified **mnuFileOpen Click** event that incorporates a **try/catch** block to check for file errors and code to check the minimum and maximum number of entries. As always, the changes are shaded:

```
private void mnuFileOpen_Click(object sender, EventArgs e)  
{  
    string myLine;  
    // Open exam file  
    if (dlgOpen.ShowDialog() == DialogResult.OK)  
    {  
        try  
        {  
            StreamReader inputFile = new  
StreamReader(dlgOpen.FileName);  
            myLine = inputFile.ReadLine();  
            examTitle = ParseLeft(myLine);  
            myLine = inputFile.ReadLine();  
            header1 = ParseLeft(myLine);  
            header2 = ParseRight(myLine);  
        }  
    }  
}
```

```
numberTerms = 0;
do
{
    numberTerms++;
    myLine = inputFile.ReadLine();
    term1[numberTerms - 1] = ParseLeft(myLine);
    term2[numberTerms - 1] = ParseRight(myLine);
}
while (inputFile.Peek() != -1 && numberTerms < 100);
if (numberTerms < 5)
{
    MessageBox.Show("Must have at least 5 entries in exam
file.", "Exam File Error", MessageBoxButtons.OK,
MessageBoxIcon.Error);
    return;
}
inputFile.Close();
}
catch
{
    MessageBox.Show("Error reading in input file - make sure file
is correct format.", "Multiple Choice Exam File Error",
MessageBoxButtons.OK, MessageBoxIcon.Error);
    return;
}

// establish form title
this.Text = "Multiple Choice Exam - " + examTitle;
// set up menu items
mnuOptionsHeader1.Text = header1 + ", Given " + header2;
mnuOptionsHeader2.Text = header2 + ", Given " + header1;
if (mnuOptionsHeader1.Checked)
{
```

```

        lblHeadGiven.Text = header2;
        lblHeadAnswer.Text = header1;
    }
    else
    {
        lblHeadGiven.Text = header1;
        lblHeadAnswer.Text = header2;
    }
    btnStart.Enabled = true;
    mnuOptions.Enabled = true;
    lblComment.Text = "File Loaded, Choose Options\r\nClick Start
Exam";
}
}

```

Let's look at the new code in a little detail. We have put all code that opens the file and extracts variables between a **try** and a **catch** statement. If an error occurs in this process, a message box is presented (code in the **catch** block) and the method is exited (after closing the file). If fewer than 5 elements are read in, a message box is presented to the user and method exited. And, notice we have modified the **while** statement to now make sure we have no more than 100 entries.

To make sure this code works, you need some invalid files. I used Notepad to create one file in proper format, but with only a single entry:

```

Title,
Header1,Header2
Term1,Term2

```

I saved this file as **Short.csv**. I also created a file in improper format, leaving off a comma in one line:

```

Title,
Header1,Header2

```

**Term1,Term2**

**Term1,Term2**

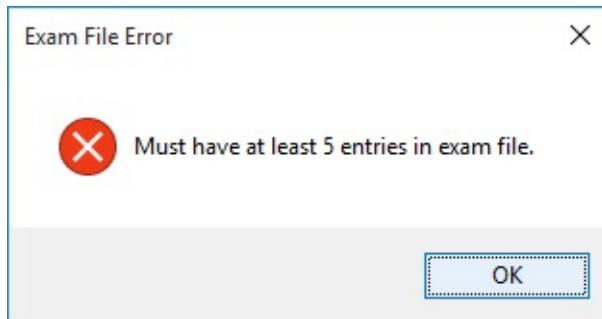
**Term1,Term2**

**Term1,Term2**

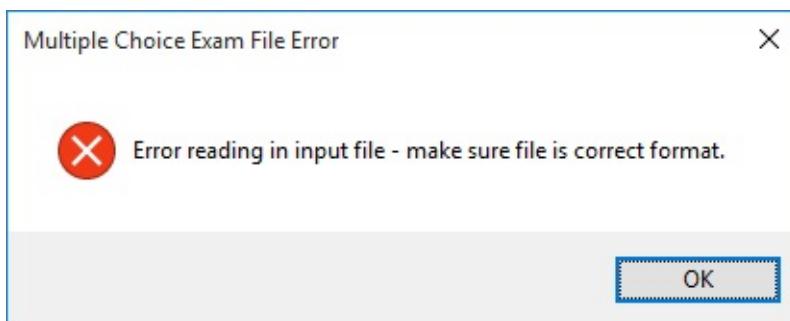
**Term1Term2**

I saved this file as **Bad.csv**. You should do the same – create some test files. Whenever adding error handling to a project, you need to make sure it works!

Save and run the project. The interface should still look the same. Open one of the example exam files to make sure it still opens successfully. Now, try loading a file with too few entries. When I try my **Short.csv** file, I get this message:



And, attempting to open the invalid file (**Bad.csv**), I get:



With either error, the user is returned to the program and allowed another chance at opening a file. This method of handling file errors is far preferable than just having the program stop with the user having no idea of what happened.

The **mnuFileOpen Click** method is now complete. Once an exam file is successfully opened, the user can change options and start an exam. We develop

that code next.

# Form Design – Selecting Options

Once an exam file is opened, the user needs to make two decisions. First, they choose which term in the list they want to have as the ‘given’ value. The other term in the list will then be the answer. As answers, the user can be given multiple answers to choose from or the user can type in the correct answer. This is other option the user must choose. All options are selected under the **Option** heading in the menu structure. The code to switch from one option to the next is in the corresponding menu items’ **Click** events.

Deciding which term will be ‘given’ involves changing the headers in two of the label controls. If the user chooses the menu option that says **header1, Given header2 (mnuOptionsHeader1)**, the steps are:

- Place check next to **mnuOptionsHeader1**
- Remove check next to **mnuOptionsHeader2**
- Set **Text** property of **lblHeadGiven** to **header2**
- Set **Text** property of **lblHeadAnswer** to **header1**

Conversely, if the user chooses the menu option that says **header2, Given header1 (mnuOptionsHeader2)**, the steps are:

- Remove check next to **mnuOptionsHeader1**
- Place check next to **mnuOptionsHeader2**
- Set **Text** property of **lblHeadGiven** to **header1**
- Set **Text** property of **lblHeadAnswer** to **header2**

The **mnuOptionsHeader1** and **mnuOptions2Header2** **Click** events that correspond to these steps are:

```
private void mnuOptionsHeader1_Click(object sender, EventArgs e)
{
    // Set up for naming header1, given header2
    mnuOptionsHeader1.Checked = true;
    mnuOptionsHeader2.Checked = false;
```

```

lblHeadGiven.Text = header2;
lblHeadAnswer.Text = header1;
}

private void mnuOptionsHeader2_Click(object sender, EventArgs e)
{
    // Set up for naming header2, given header1
    mnuOptionsHeader1.Checked = false;
    mnuOptionsHeader2.Checked = true;
    lblHeadGiven.Text = header1;
    lblHeadAnswer.Text = header2;
}

```

Add these event methods.

Choosing between multiple choice and type in answers requires reconfiguration of the form. The multiple choice option requires four label controls to present the possible answers, while the type in option requires a single text box for entry of the answer. The steps involved in choosing the **Multiple Choice Answers** option (**mnuOptionsMC**) are:

- Place check next to **mnuOptionsMC**
- Remove check next to **mnuOptionsType**
- Make four label controls for answer visible.
- Make text box control invisible.

And, conversely, if the **Type In Answer** option (**mnuOptionsType**), the steps are:

- Remove check next to **mnuOptionsMC**
- Place check next to **mnuOptionsType**
- Make four label controls for answer invisible.
- Make text box control visible.

The **mnuOptionsMC** and **mnuOptions2Type Click** events that correspond to

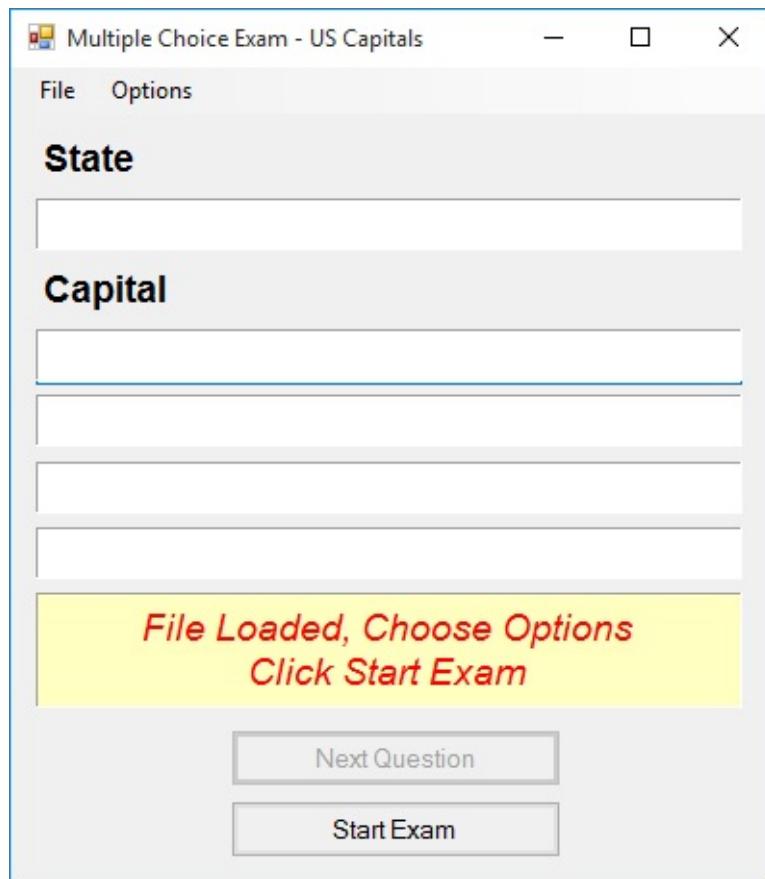
these steps are:

```
private void mnuOptionsMC_Click(object sender, EventArgs e)
{
    // Set up for multiple choice answers
    mnuOptionsMC.Checked = true;
    mnuOptionsType.Checked = false;
    lblAnswer1.Visible = true;
    lblAnswer2.Visible = true;
    lblAnswer3.Visible = true;
    lblAnswer4.Visible = true;
    txtAnswer.Visible = false;
}

private void mnuOptionsType_Click(object sender, EventArgs e)
{
    // Set up for type-in answers
    mnuOptionsMC.Checked = false;
    mnuOptionsType.Checked = true;
    lblAnswer1.Visible = false;
    lblAnswer2.Visible = false;
    lblAnswer3.Visible = false;
    lblAnswer4.Visible = false;
    txtAnswer.Visible = true;
}
```

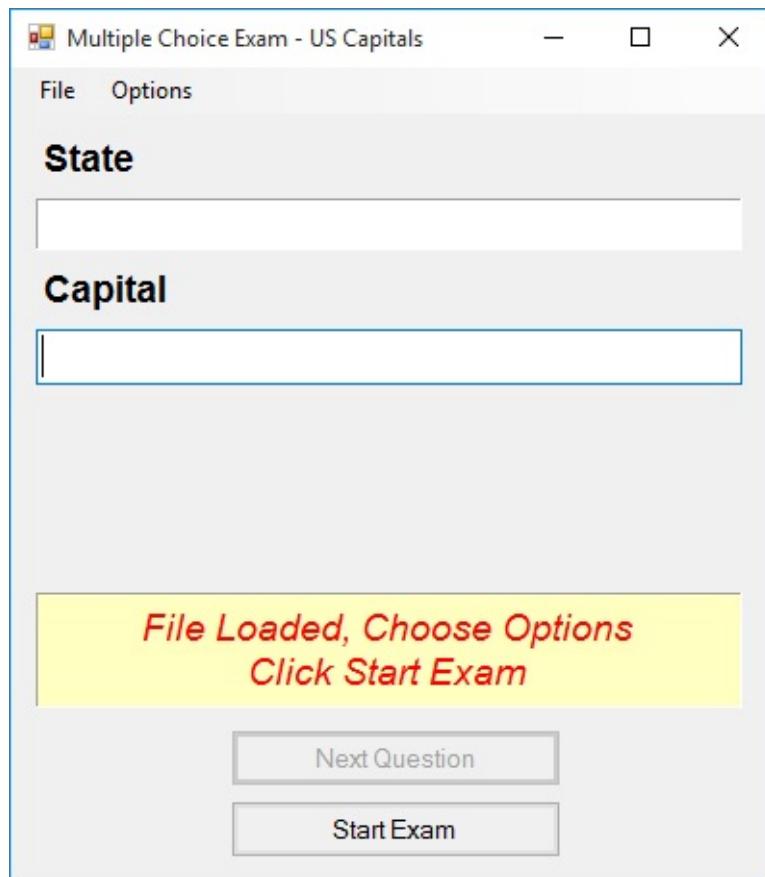
Add these event methods.

Save and run the project. Open an example exam file. Make sure the newly coded options work correctly. When I load the **USCapitals.csv** file and select the **Capital, Given State** option, I see:



Note the proper headers.

By default, the multiple choice answer option is shown. Choosing the **Type In Answers** option changes the form to:



Make sure you can change back to the **Multiple Choice Answers** option.

We've completed the code for opening exam files and configuring the interface (by choosing options). It is now time to address the code to present an exam to the user.

# Code Design – Start Exam

The idea of a multiple choice exam is to display a random question, obtain an answer from the user and check for correctness. Once an exam is complete, scoring results are provided.

An exam is started by clicking **Start Exam (btnStart)**. We want to make sure all the user can do at this point is answer a question. The following steps should occur:

- Change **Text** property of **btnStart** to **Stop Practice**
- Disable **btnNext**.
- Set number of questions tried and number correct to zero.
- Clear **Text** property of **lblComment**.
- Disable **mnuFile**.
- Disable **mnuOptions**.
- Present question.
- Check answer and update score.

Once each question is answered, subsequent questions are presented.

The user answers questions until he/she clicks **Stop Exam** (also **btnStart**). We want the interface to return to where options can be selected or a new file opened. The steps at this point are:

- Change **Text** property of **btnStart** to **Stop Practice**
- Disable **btnNext**.
- Present results.
- Clear **Text** property of all controls used for answers.
- Set **Text** property of **lblComment** to indicate a new exam can be started.
- Enable **mnuFile**.
- Enable **mnuOptions**.

Let's build a framework for the **btnStart Click** event method that implements

most of these steps (we'll look at presenting results later). Declare two form level variables to keep track of the number of questions tried and the number correct:

```
int numberTried, numberCorrect;
```

The **btnStart Click** event method that implements the listed steps (again, except for results) is:

```
private void btnStart_Click(object sender, EventArgs e)
{
    if (btnStart.Text == "Start Exam")
    {
        btnStart.Text = "Stop Exam";
        btnNext.Enabled = false;
        // Reset the score
        numberTried = 0;
        numberCorrect = 0;
        lblComment.Text = "";
        mnuFile.Enabled = false;
        mnuOptions.Enabled = false;
        NextQuestion();
    }
    else
    {
        btnStart.Text = "Start Exam";
        btnNext.Enabled = false;
        lblGiven.Text = "";
        lblAnswer1.Text = "";
        lblAnswer2.Text = "";
        lblAnswer3.Text = "";
        lblAnswer4.Text = "";
        txtAnswer.Text = "";
    }
}
```

```
    lblComment.Text = "Choose Options\r\nClick Start Exam";  
    mnuFile.Enabled = true;  
    mnuOptions.Enabled = true;  
}  
}
```

Add this code to the project.

This code uses a general method **NextQuestion** to generate a random question.  
Add this empty method to the project:

```
private void NextQuestion()  
{  
}
```

Note the use of the keyword **void**, since this method does not return a value, but just generates a question. We'll write code for this method next, once we make sure the changes just made work.

Save and run the project. Open an exam file. Make sure the **Start Exam** and **Stop Exam** buttons function properly, changing the interface as desired.

# Code Design - Question Generation

To generate a question, we examine the options selected by the user and produce a random question based on these selections. The code to generate such a question will be in the **NextQuestion** general method.

The steps involved in generating a random question are:

- Clear **Text** property of **lblComment**.
- Select random item from term list as the “correct answer”.
- Set **Text** property of **lblGiven** to ‘given’ term.
- If **Multiple Choice Answers** is selected:
  - o Generate four possible answers (one of which is the correct answer)
  - o Display answers in label controls.
- If **Type In Answers** is selected:
  - o Set **txtAnswer** **ReadOnly** property to **false**.
  - o Clear **txtAnswer** text box.
  - o Give **txtAnswer** focus.

The code to select a question, set **lblGiven** and to set up for **type in answers** is straightforward. First, add a form level variable to identify the array index of the correct answer and a random object to generate random questions:

```
int correctAnswer;  
Random myRandom = new Random();
```

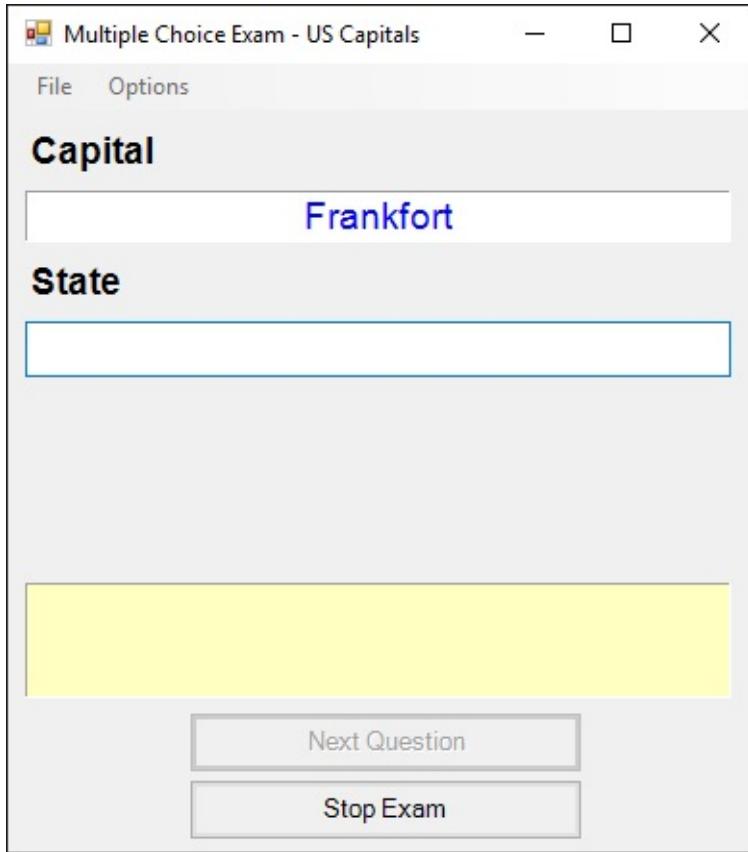
The code for the **NextQuestion** method for these steps is:

```
private void NextQuestion()  
{  
    lblComment.Text = "";  
    // Generate the next question based on selected options  
    correctAnswer = myRandom.Next(numberTerms);
```

```
if (mnuOptionsHeader1.Checked)
{
    lblGiven.Text = term2[correctAnswer];
}
else
{
    lblGiven.Text = term1[correctAnswer];
}
if (mnuOptionsMC.Checked)
{
    // Multiple choice answers
} else
{
    // Type-in answers
    txtAnswer.ReadOnly = false;
    txtAnswer.Text = "";
    txtAnswer.Focus();
}
}
```

Add this code to the project.

Save and run the project if you'd like to see if you can type in answers (make sure you select this option). Here's an example with the **USCapitals.csv** file (you will see a different result because of the **Random** object):



We will see how to check an answer soon.

The code for presenting **multiple choice answers** is more detailed and we need to spend some time looking at it. The tricky part of this code is to select the four multiple choice options, one of which is the correct answer. The approach we follow is to first select four terms at random from the **numberTerms** possibilities, making sure we don't select the correct answer (`index` is **correctAnswer**). Once we have these four possibilities, we replace one at random with the correct answer. Let's look at the steps.

First, we need some way to know if we have already selected a previously chosen answer possibility. We will use a method level **bool** array **termUsed**, dimensioned to **numberTerms** to tell us if a term has been used. Each element in this array is initialized to **false**, indicating all are available. The code snippet that accomplishes this task is:

```
bool[] termUsed = new bool[numberTerms];
for (int i = 0; i < numberTerms; i++)
```

```
{  
    termUsed[i] = false;  
}
```

A **do** loop is used to pick the four random answer possibilities. An **int** array **index**, dimensioned to 4, stores the four selected indices. The code snippet is:

```
int[] index = new int[4];  
int j;  
for (int i = 0; i < 4; i++)  
{  
    do  
    {  
        j = myRandom.Next(numberTerms);  
    }  
    while (termUsed[j] || j == correctAnswer);  
    termUsed[j] = true;  
    index[i] = j;  
}
```

See how this works? For each of the four answers (selected with the **for** loop), a random index **j** is selected making sure the corresponding term has not been selected before and is not the **correctAnswer**.

Once the array **index** is established, one item in the array is replaced with **correctAnswer**. The line of code that accomplishes this replacement is:

```
index[myRandom.Next(4)] = correctAnswer;
```

Now, depending on which is term is given and which is the answer, the **index** array establishes the contents of the label controls used for multiple choice answers.

The modified **NextQuestion** method that incorporates the code for multiple choice answers (changes are shaded) is:

```
private void NextQuestion()
{
    bool[] termUsed = new bool[numberTerms];
    int[] index = new int[4];
    int j;
    lblComment.Text = "";
    // Generate the next question based on selected options
    correctAnswer = myRandom.Next(numberTerms);
    if (mnuOptionsHeader1.Checked)
    {
        lblGiven.Text = term2[correctAnswer];
    }
    else
    {
        lblGiven.Text = term1[correctAnswer];
    }
    if (mnuOptionsMC.Checked)
    {
        // Multiple choice answers
        for (int i = 0; i < numberTerms; i++)
        {
            termUsed[i] = false;
        }
        // Pick four random possibilities
        for (int i = 0; i < 4; i++)
        {
            do
            {
                j = myRandom.Next(numberTerms);
            }
            while (termUsed[j] || j == correctAnswer);
            termUsed[j] = true;
        }
    }
}
```

```

        index[i] = j;
    }

    // Replace one with correct answer
    index[myRandom.Next(4)] = correctAnswer;

    // Display multiple choice answers in label boxes
    if (mnuOptionsHeader1.Checked)
    {
        lblAnswer1.Text = term1[index[0]];
        lblAnswer2.Text = term1[index[1]];
        lblAnswer3.Text = term1[index[2]];
        lblAnswer4.Text = term1[index[3]];
    }
    else
    {
        lblAnswer1.Text = term2[index[0]];
        lblAnswer2.Text = term2[index[1]];
        lblAnswer3.Text = term2[index[2]];
        lblAnswer4.Text = term2[index[3]];
    }
}

else
{
    // Type-in answers
    txtAnswer.ReadOnly = false;
    txtAnswer.Text = "";
    txtAnswer.Focus();
}
}

```

Make the noted modifications.

Save and run the project. Open an example exam file. Using default options and the **USCapitals.csv** file, clicking **Start Exam**, I see (you will see different

results because of the **Random** object):

The screenshot shows a window titled "Multiple Choice Exam - US Capitals". At the top, there's a menu bar with "File" and "Options". The main area has a title "Capital" followed by a question box containing "Indianapolis". Below it is another box labeled "State" with four options: "Minnesota", "Pennsylvania", "New Hampshire", and "Indiana". The "Indiana" option is highlighted with a yellow background. At the bottom, there are two buttons: "Next Question" and "Stop Exam".

The given **Capital** is **Indianapolis**. Note the four possible **State** answers (three random and one the correct answer, **Indiana**). The multiple choice logic seems to be working. All you can do at this point is click **Stop Exam**. You can then click **Start Exam** to see another question (changing options if you wish). View as many questions, with different options, as you wish. Next, we'll see how to get answers to these questions – we consider both multiple choice and type in answers.

# Code Design – Checking Multiple Choice Answers

Once a question is presented using multiple choice answers, the user is asked to click on the correct answer. That answer is then checked - we will only give the user one chance to get the answer right. The steps for checking a multiple choice answer:

- Make sure question hasn't been answered already.
- Increment **numberTried**.
- Determine which label control was clicked.
- Check to see if **Text** property of clicked label control matches the correct answer (correct answer depends on which term is given and which is answer).
- Update the score and provide feedback, presenting the correct answer.

The code corresponding to these steps is placed in a method named **lblAnswer\_Click**. This method will handle the **Click** events on all four label controls used to display answers: **lblAnswer1**, **lblAnswer2**, **lblAnswer3**, **lblAnswer4**. The code for **lblAnswer\_Click** is:

```
private void lblAnswer_Click(object sender, EventArgs e)
{
    bool correct = false;
    Label labelClicked;
    // If exam not in progress or already answered, exit
    if (btnStart.Text == "Start Exam" || btnNext.Enabled)
        return;
    numberTried++;
    // find out which label was clicked
    labelClicked = (Label) sender;
    if (mnuOptionsHeader1.Checked)
    {
```

```

if (labelClicked.Text == term1[correctAnswer])
    correct = true;
}
else
{
    if (labelClicked.Text == term2[correctAnswer])
        correct = true;
}
UpdateScore(correct);
}

```

This code uses a general method **UpdateScore** to update the scoring and prepare the user interface for the next question. The method uses a single **bool** argument that is **true** if the answer was answered correct, **false** is incorrect. The steps involved:

- If answer is correct: increment **numberCorrect** and set **Text** property of **lblComment** to “**Correct!**”
- If answer is incorrect: set **Text** property of **lblComment** to “**Sorry ... Correct Answer Shown**”
- If multiple choice answers are used: put correct answer in **lblAnswer1**, clear all other label controls.
- If type in answers are used: put correct answer in **txtAnswer**.
- Enable **btnStart**.
- Enable **btnNext**.
- Give **btnNext** focus.

The code for **UpdateScore** is:

```

private void UpdateScore(bool correct)
{
    // Check if answer is correct
    if (correct)
    {

```

```

numberCorrect++;
lblComment.Text = "Correct!";
}
else
    lblComment.Text = "Sorry ... Correct Answer Shown";
    // Display correct answer
    if (mnuOptionsMC.Checked)
    {
        if (mnuOptionsHeader1.Checked)
            lblAnswer1.Text = term1[correctAnswer];
        else
            lblAnswer1.Text = term2[correctAnswer];
        lblAnswer2.Text = "";
        lblAnswer3.Text = "";
        lblAnswer4.Text = "";
    }
    else
    {
        if (mnuOptionsHeader1.Checked)
            txtAnswer.Text = term1[correctAnswer];
        else
            txtAnswer.Text = term2[correctAnswer];
    }
    btnStart.Enabled = true;
    btnNext.Enabled = true;
    btnNext.Focus();
}

```

Add the **lblAnswer\_Click** and **UpdateScore** (this routine will also be used when checking typed in answers) code to the project.

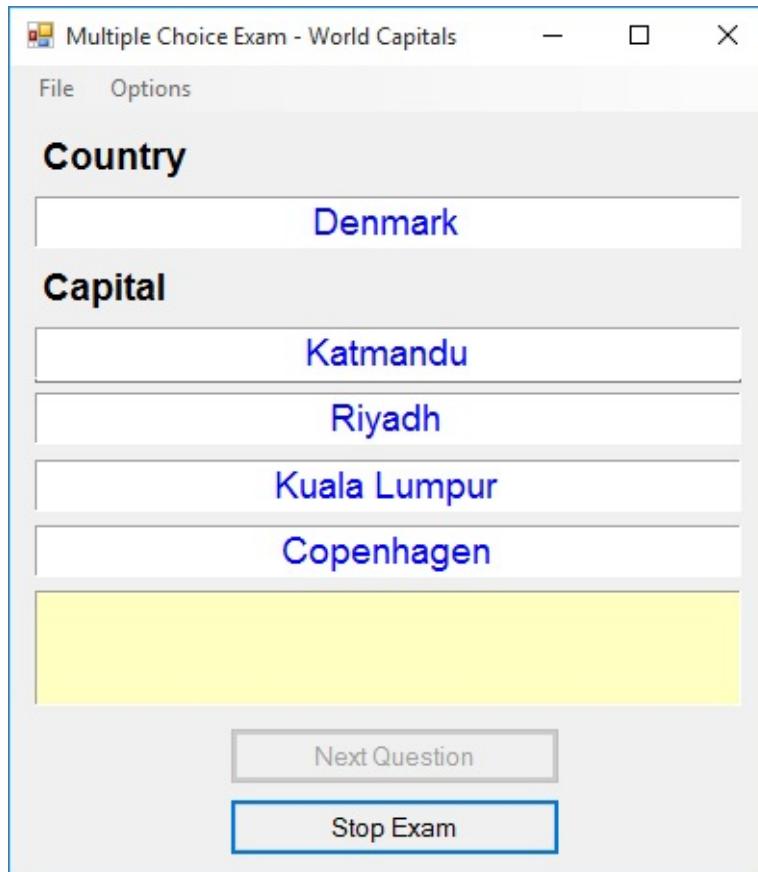
Notice after displaying the correct answer, focus is given to **btnNext**. Clicking this button will present another question to the user. The code for the **btnNext**

**Click** event simply involves disabling the button, once clicked, then invoking the existing **NextQuestion** method:

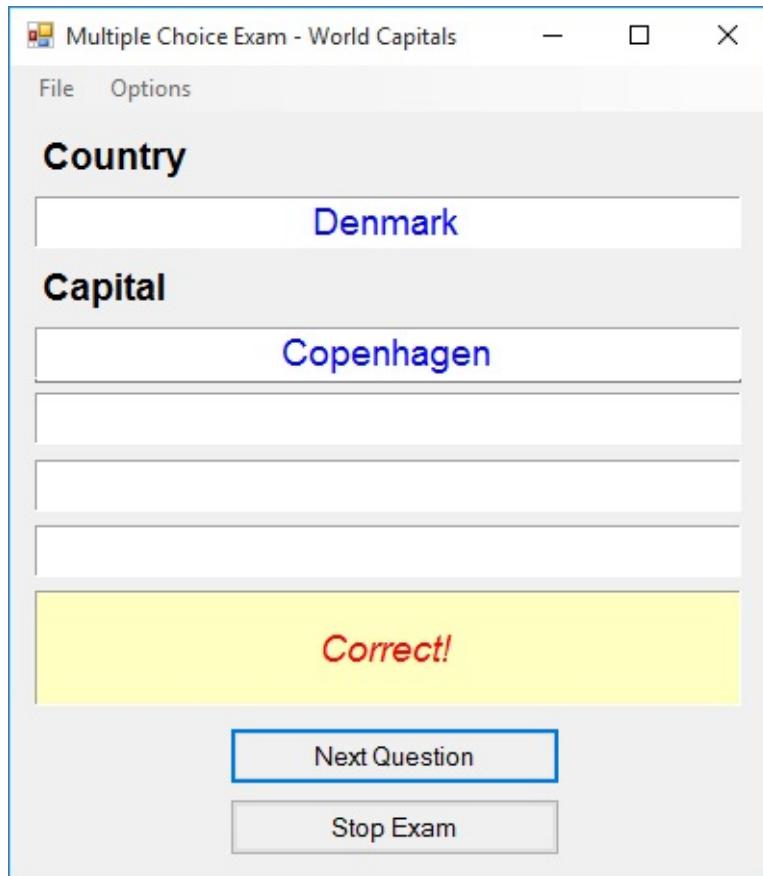
```
private void btnNext_Click(object sender, EventArgs e)
{
    // Generate next question
    btnNext.Enabled = false;
    NextQuestion();
}
```

Enter this code and we are now ready to take exams with multiple choice answers

Save and run the project. Open an exam file. Select options (obviously choose multiple choice answers). For the example here, I use the **WorldCapitals.csv** file, providing capitals, given the country. The first question I see is:



When I click **Copenhagen**, I see:



At this point, I can click **Next Question** for another question, or click **Stop Exam** to stop this test. Answer as many questions as you like.

At some point, answer incorrectly. When I do, I see:



So, with an incorrect answer, you are told so and given the correct answer. The only difference between the results of a correct and incorrect answer is the message displayed to the user (and the score, of course).

# Code Design – Checking Type In Answers

We've talked about problems when using text box controls before and the advantages of using a 'point and click' control instead. Here's a case where problems can arise. We saw it was a clear decision to check whether a multiple choice answer was correct. It's not so clear here.

When a user types an answer, a first consideration that must be made is whether any typed keys are unacceptable. Do we need key trapping? In this project, we will not use key trapping, allowing all keystrokes. A second question is how do we know when a user is done entering an answer? You could have a button to click that says **Check Answer** or have the user press a certain key. In this exam project, we will check the answer once a user presses the <Enter> key. So, we will need a bit of key trapping!

We need to consider case sensitivity when entering alphabetic entries. For example, in the **USCapitals.csv** file, the capital of the state of Washington (our home state) is saved as **Olympia**. If a user types **olympia** (all lower case), do we really want to tell the user the answer is incorrect? Or, what if they type **Olimpia**, a very close spelling? What do we do in this situation? We will solve both of these problems, addressing case-sensitivity first.

Once a user types an answer and presses <Enter>, we take these steps:

- Make sure question hasn't been answered already.
- Set **txtAnswer** **ReadOnly** property to **true**.
- Increment **numberTried**.
- Convert **txtAnswer.Text** (user answer) to all upper case.
- Convert correct answer to all upper case.
- Compare upper case strings to see if they are equal.
- Update the score and provide feedback, presenting the correct answer.

The method **ToUpper** converts a **string** value to all upper case. The function

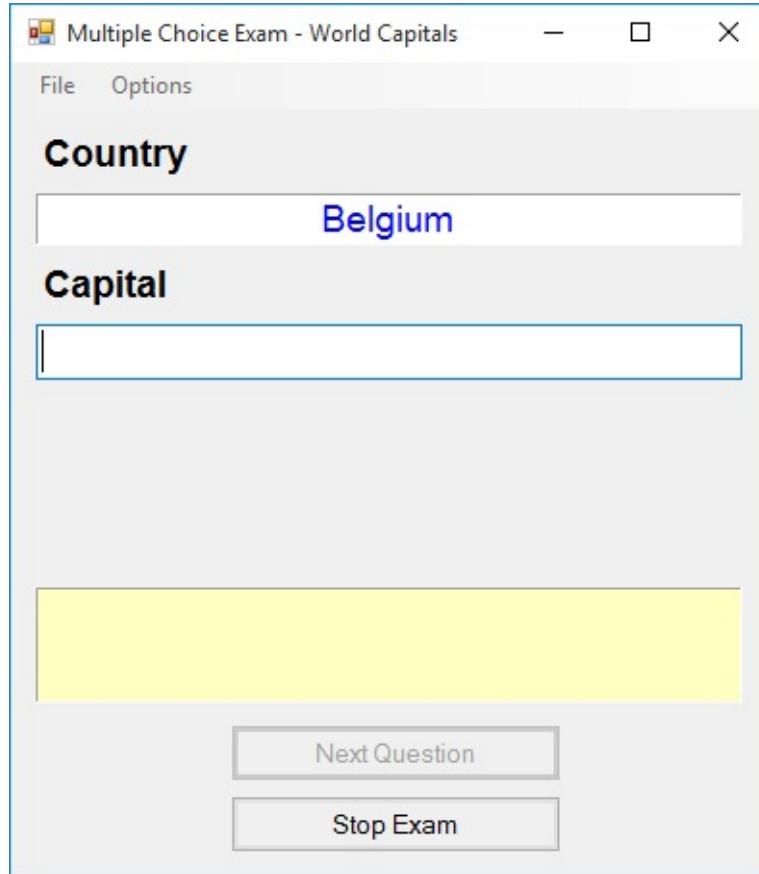
ignores any non-letter characters.

We place the code for these steps in the **txtAnswer KeyPress** event (processing the code when the <Enter> key is pressed). The method is:

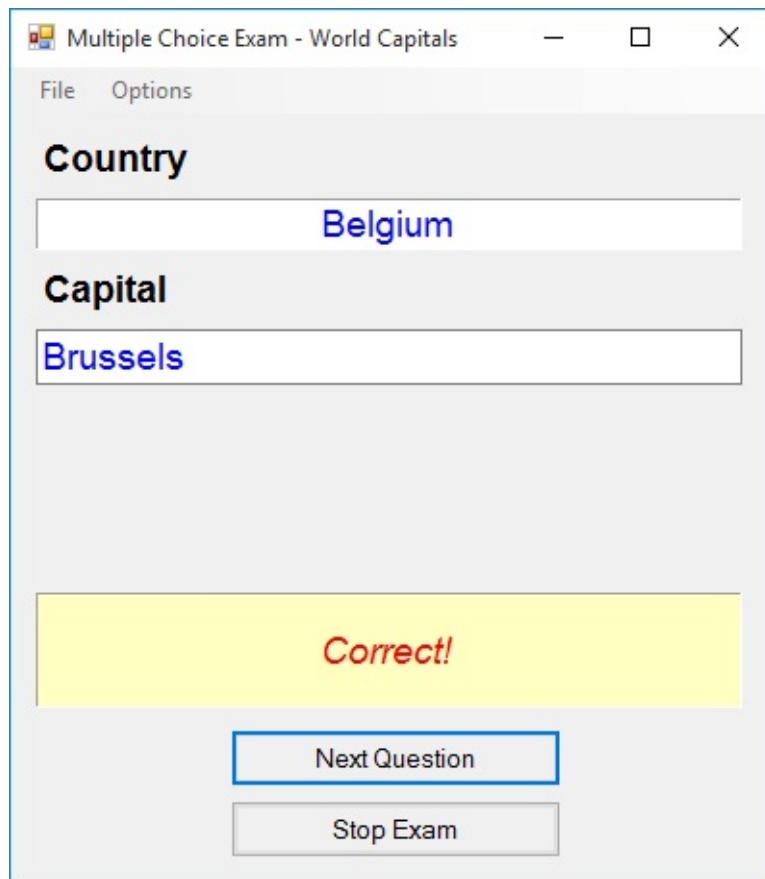
```
private void txtAnswer_KeyPress(object sender, KeyPressEventArgs e)
{
    // Check type in answer
    bool correct;
    string ucTypedAnswer, ucAnswer;
    // If exam not in progress or already answered, exit
    if (btnStart.Text == "Start Exam" || btnNext.Enabled)
        return;
    // wait for <Enter>
    if ((int) e.KeyChar == 13)
    {
        txtAnswer.ReadOnly = true;
        numberTried++;
        ucTypedAnswer = txtAnswer.Text.ToUpper();
        if (mnuOptionsHeader1.Checked)
            ucAnswer = term1[correctAnswer].ToUpper();
        else
            ucAnswer = term2[correctAnswer].ToUpper();
        correct = false;
        if (ucTypedAnswer == ucAnswer)
            correct = true;
        UpdateScore(correct);
    }
}
```

Note the use of the **ToUpper** method. This code also uses the general method **UpdateScore** to update the score and controls after answering. Add the **txtAnswer Keypress** method to the project.

Save and run the project. Select an exam file (I used **WorldCapitals.csv**). Choose the **Type In Answers** option. I also selected **Capitals, Given Country** as an option. Click **Start Exam**. My first question appears as:

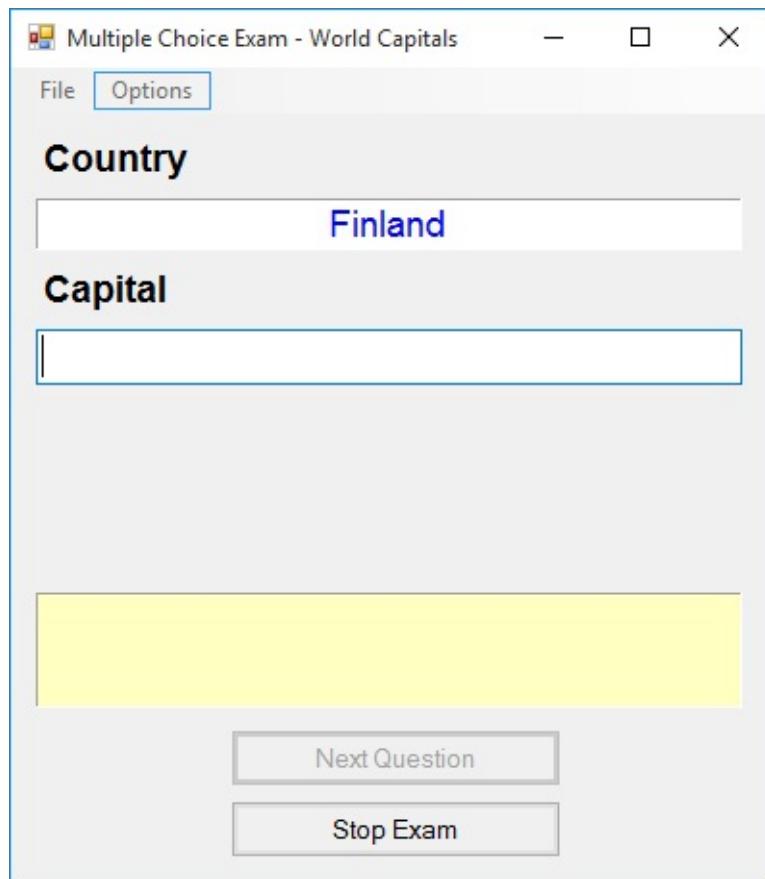


If I type **Brussels**, the correct answer with correct letter case, then press **<Enter>**, I am told the answer is correct:



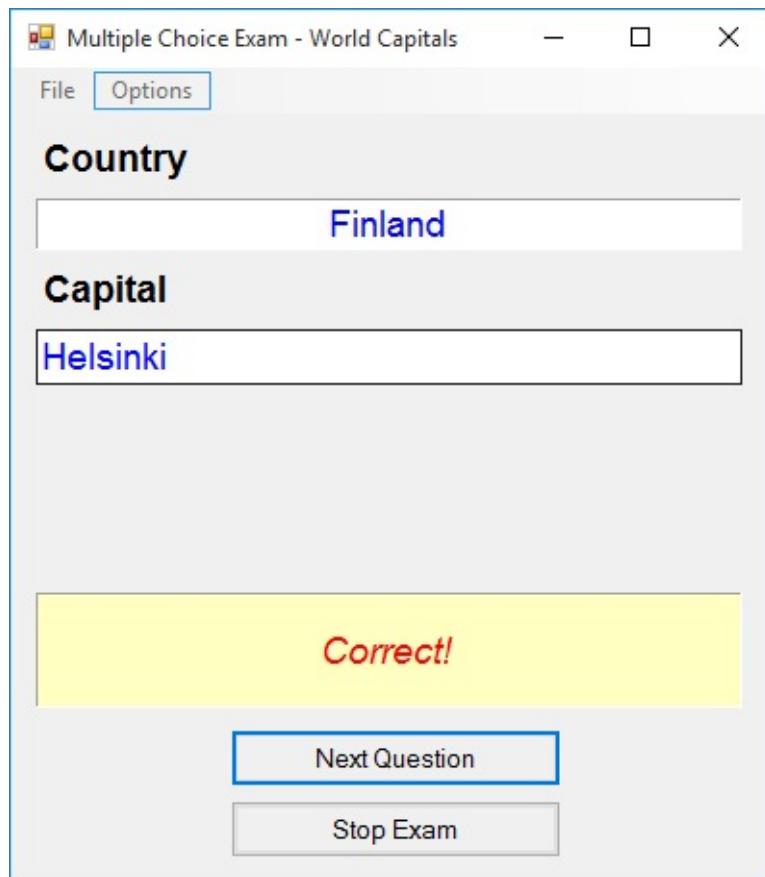
**Click Next Question.**

The next question is:



The capital of **Finland** is **Helsinki**. If you type **Helsinki** in the text box area and press <Enter> you will be told this is a correct answer. Let's make sure the answers are not case-sensitive.

When I type **helsinki** in the text box and click <Enter>, I see:



The answer is accepted and the ‘capitalization’ is corrected.

Continue trying correct and incorrect answers, checking to make sure case-insensitivity is properly incorporated into the project. Try typing an answer with spelling ‘close to’ the correct spelling. You will be told you are incorrect. This can be frustrating to the ‘spelling challenged’ and especially frustrating for children learning how to spell. If your spelling is ‘close’ you should be rewarded and gently corrected, not told you are wrong. Stop the exam and the project and we’ll fix this problem.

# Code Design – Checking Spelling

The techniques behind checking for ‘close spelling’ are called **Soundex** checks. Words, or terms, are assigned something called a **Soundex code**. Any two terms with the same Soundex code will have similar spellings. This is how spell checker programs work. When you misspell a word, you are presented with a list of words with similar Soundex codes from which to choose possible corrections. In our multiple choice exam project, if the Soundex code for a user typed response is equal to the Soundex code for the actual answer, we will credit the user with a correct answer.

The technique we use to determine Soundex codes is based on an article in an issue of **Byte** magazine from the early 1980’s. As a historical footnote, early programmers were always eager to get the latest issue of **Byte**. It would contain programs you could type into your computer and try. These programs were usually written in the BASIC language. The code here is based on one of these programs. It’s fun to go to a local library and look at old issues of **Byte** magazine. You’ll find ads for computers with 8K (yes, I said 8K) of memory for just \$500. And, you’ll see 1/12<sup>th</sup> page ads for a little Bellevue, Washington, company just getting started in the computer business – yes, Microsoft.

To determine the **Soundex** code **c** (a **string** value) for a **string** value **w** (whose first character must be a letter), these steps are followed:

- Convert **w** to all upper case (call the result **ucw**)
- Set the first character of **c** to the first character of **ucw**.
- Cycle through all remaining characters in **ucw**, one at a time.
- Assign letter characters in **ucw** a corresponding numerical value from **0** to **9**, according to provided table. Numerical values are not given to any non-letter characters.
- If numerical value is non-zero and not equal to the previous character’s numerical value, append that number to the end of the **Soundex** code **c**.

The numerical values associated with the 26 letters of the English alphabet are:

$$A = 0 \quad B = 1 \quad C = 2 \quad D = 3 \quad E = 0 \quad F = 1 \quad G = 2$$

H = 0	I = 0	J = 2	K = 2	L = 4	M = 5	N = 5
O = 0	P = 1	Q = 2	R = 6	S = 2	T = 3	U = 0
V = 1	W = 0	X = 2	Y = 0	Z = 2		

Notice the vowels (A, E, I, O, U) and soft consonants (H, W, Y) have zero values.

You should see that a **Soundex** code will be a string starting with a letter, followed by a sequence of numbers (none of which are zero) with no identical consecutive numbers. Let's try it with an example to see how it works, then we'll write the code. We'll use the word 'beautiful'. We'll misspell it as 'buetifull'. First, convert both words to upper case. Initialize the Soundex codes for both to the first letter of the word (both will be **B**). So, obviously a condition for two Soundex codes to match is that the first letter of the two words being compared must be the same. Now, go through all subsequent letters in each capitalized word and assign the corresponding numerical value to the letters. The results are:

<b>BEAUTIFUL</b>	<b>Code: B00030104</b>
<b>BUETIFULL</b>	<b>Code: B00301044</b>

Remove the zeroes and repeated values to get the final codes:

<b>BEAUTIFUL</b>	<b>Code: B314</b>
<b>BUETIFULL</b>	<b>Code: B314</b>

The two codes match, hence have similar spellings. Can you find other words with the same code. Some I came up with are: bad ball (the space is ignored by Soundex), bedful, and bait pail. So, Soundex doesn't always work – call some one 'bait pail' instead of 'beautiful' and you'll see what I mean!

The code to compute a Soundex code will be in a general method **Soundex**. The method will have a single **string** argument, **w**, the word the code is being computed for. The method returns a **string** argument which is the **Soundex code** for **w**:

```
public string SoundEx(string w)
```

```
{
```

```

// Generates Soundex code for w
string[] nvArray = {"0", "1", "2", "3", "0", "1", "2", "0", "0", "2",
"2", "4", "5", "5", "0", "1", "2", "6", "2", "3", "0", "1", "0", "2", "0",
"2"};
string ucw, c, nv;
int l, nvIndex;
ucw = w.ToUpper();
l = w.Length;
c = "";
if (l != 0)
{
    // set first character of code
    c = ucw.Substring(0, 1);
    if (l > 1)
    {
        for (int i = 1; i < l; i++)
        {
            // get numeric value for next character
            nvIndex = (int)
Convert.ToChar(ucw.Substring(i, 1)) - 65;
            if (nvIndex >= 0 && nvIndex <= 25)
                nv = nvArray[nvIndex];
            else
                nv = "0";
            // append if not zero and not previous character
            if (nv.CompareTo("0") != 0 &&
nv.CompareTo(c.Substring(c.Length - 1, 1)) != 0)
                c += nv;
        }
    }
    return (c);
}

```

}

The steps for finding a **Soundex** code are straightforward – the coding may not seem so. Let me explain what's going on here. First, the 26 numeric values are stored in a string array named **nvArray**. **nvArray[0]** represents the numeric value for an **A** up to **nvArray[25]**, which represents the numeric value for a **Z**. The input word (**w**) is converted to all upper case (**ucw**). The returned code (**c**) is initialized to the first character of **ucw** (obtained using the **Substring** method).

The tricky part of the code is getting the numeric values for the subsequent characters in **ucw**. The characters are related to their corresponding index in **nvArray** by their **Unicode** value (**65** for an **A**, up to **90** for a **Z**). So, the process is:

- Find the next **character** in **ucw** using **Substring**.
- Find **Unicode** value for character and subtract 65, this is the array index (**nvIndex**).
- Find character's numeric value (**nv**).
- Append **nv** to **c** if not a zero (0) and not equal to the last character currently in **c**.

This last step uses the string **CompareTo** method. The syntax for this method for two strings **a** and **b** is:

### **a.CompareTo(b)**

The returned value is negative if **a** is “less than” **b**, 0 if **a** “equals” **b** and positive if **a** is “greater than” **b**. You should be able to see all these steps in the code. Add this method to the project.

To use the **Soundex** function, we must modify a single line of code in the **txtAnswer\_KeyPress** method to not only check for exact spelling, but for equal **Soundex** codes. The modified line is shaded (most unmodified code is not shown):

```
private void txtAnswer_KeyPress(object sender, KeyPressEventArgs e)  
{
```

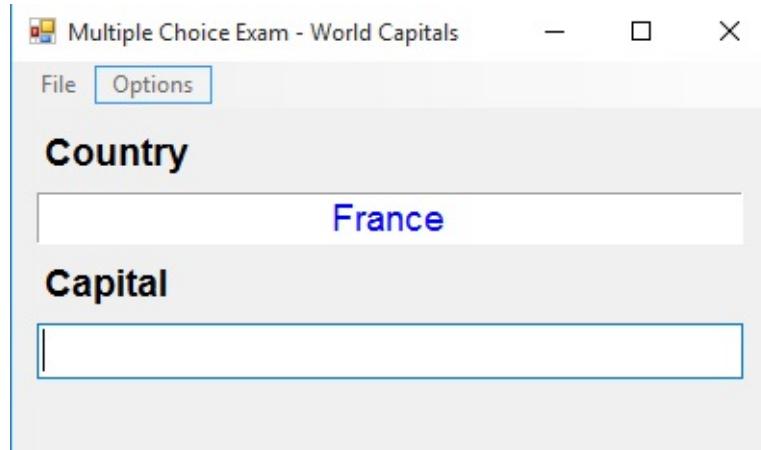
```

        .
        .
        correct = false;
        if (ucTypedAnswer == ucAnswer || SoundEx(ucTypedAnswer) ==
SoundEx(ucAnswer))
            correct = true;
        UpdateScore(correct);
    }
}

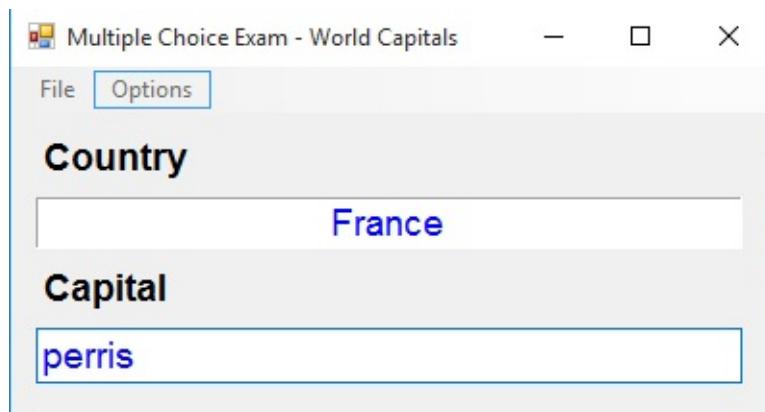
```

Make this change. Now, let's give the **Soundex** code a try!

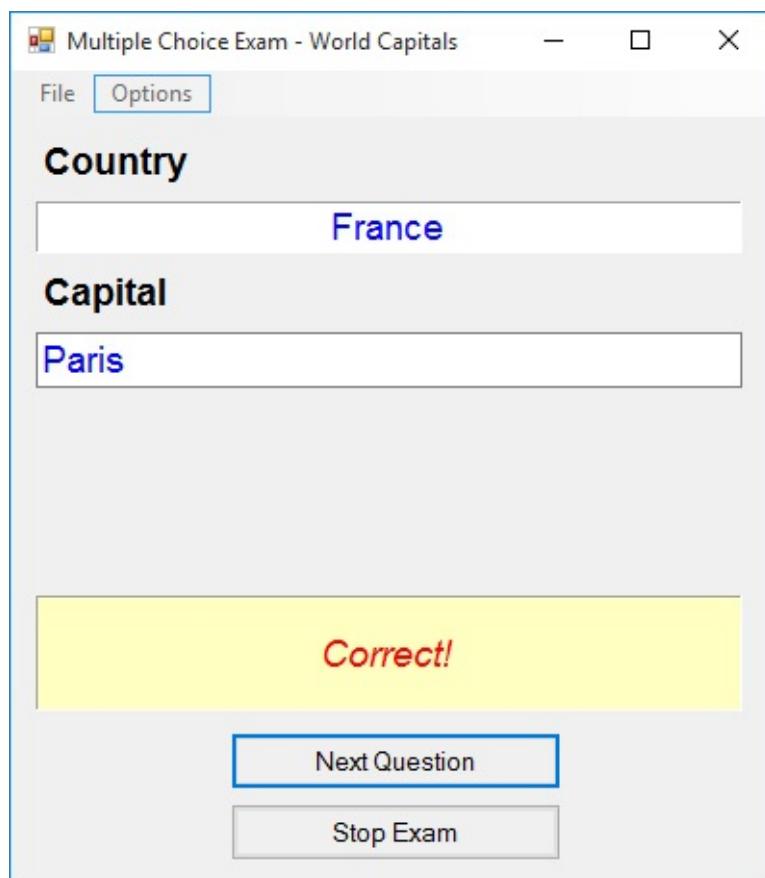
Save and run the project. Select an exam file (I again used **WorldCapitals.csv**). Choose the **Type In Answers** option. I again selected **Capitals, Given Country** as an option. Click **Start Exam**. My first question appears as:



The capital of **France** is **Paris** (Soundex code is **P62**), but what if I mistakenly spell it as **perris**:



When I press <Enter>, I see:



So, even though I misspelled the word, I am given credit for a correct answer and shown the correct spelling. This happens because the Soundex code for 'perris' is P62, the same code as 'Paris'.

The program is nearly complete. Keep trying exams with different options to

make sure everything works correctly. Play with the type in answers to see how well the Soundex codes work. How close does the spelling really need to be? Choose **File**, then **Exit** to stop the program when you want.

# Code Design – Presenting Results

Once a user stops a particular exam, we want to let them know how well they did in answering questions. The information of use would be:

- The number of questions tried.
- The number of correct answers.
- The percentage score.

All of this information is available from the defined variables.

The exam results are presented in the **btnStart Click** event method (following clicking of **Stop Exam**). A message box is used to display the results. The modified **btnStart Click** method (changes are shaded) is:

```
private void btnStart_Click(object sender, EventArgs e)
{
    string message;
    if (btnStart.Text == "Start Exam")
    {
        btnStart.Text = "Stop Exam";
        btnNext.Enabled = false;
        // Reset the score
        numberTried = 0;
        numberCorrect = 0;
        lblComment.Text = "";
        mnuFile.Enabled = false;
        mnuOptions.Enabled = false;
        NextQuestion();
    }
    else
    {
```

```

btnStart.Text = "Start Exam";
btnNext.Enabled = false;
if (numberTried > 0)
{
    message = "Questions Tried: " + numberTried.ToString() +
"\r\n";
    message += "Questions Correct: " +
numberCorrect.ToString() + "\r\n\r\n";
    message += "Your Score: " + String.Format("{0:f0}", 100.0 *
((double) numberCorrect / numberTried)) + "%";
    MessageBox.Show(message, examTitle + " Results",
MessageBoxButtons.OK, MessageBoxIcon.Information);
}

lblGiven.Text = "";
lblAnswer1.Text = "";
lblAnswer2.Text = "";
lblAnswer3.Text = "";
lblAnswer4.Text = "";
txtAnswer.Text = "";
lblComment.Text = "Choose Options\r\nClick Start Exam";
mnuFile.Enabled = true;
mnuOptions.Enabled = true;
}
}

```

Make the noted changes.

And, one last time, save and run the project. Load in an exam file. Take some kind of exam. Answer some questions – miss a few to make sure the scoring works. At some point, click **Stop Exam** and some results should appear. Here's a message box I received after taking an exam:

World Capitals Results

X



Questions Tried: 7  
Questions Correct: 6

Your Score: 86%

OK

# Multiple Choice Exam Project Review

The **Multiple Choice Exam** project is now complete. Save and run the project and make sure it works as designed. Recheck that all options work and interact properly. Create some exam files (or use the two examples) and have fun learning.

If there are errors in your implementation, go back over the steps of form and code design. Use the debugger when needed. Go over the developed code – make sure you understand how different parts of the project were coded. As mentioned in the beginning of this chapter, the completed project is saved as **Multiple Choice** in the **HomeVCS\HomeVCS Projects** folder.

While completing this project, new concepts and skills you should have gained include:

- How to use the menu strip control.
- How to use the open file dialog control to obtain a filename.
- Creating and saving an exam file.
- Opening a sequential file, inputting and parsing data lines.
- Error trapping techniques.
- Checking spelling using Soundex codes.

# Multiple Choice Exam Project Enhancements

Possible enhancements to the multiple choice exam project include:

- The only feedback a user gets about entered answers is a displayed message. Some kind of audible feedback would be nice (a positive sound for correct answer, a negative sound for a wrong answer). We discuss adding sounds to a project in Chapter 12 – you might like to look ahead.
- Modify the program and scoring system to allow multiple tries at the answer. Award higher scores for fewer missed guesses. If using type in answers, you would need some kind of ‘I Give Up’ button or just give a specified number of guesses.
- The user only learns the results after an exam. Add some controls that always display the current results.
- Add an option that allows a user to review the entries in an exam file.
- Build an ‘Exam Builder’ tool that lets a user enter the needed information and save the exam file. You need to know how to save sequential files, a topic discussed in Chapter 10.
- Add printing capabilities where you can print out exams to take on your own time. We discuss printing in Chapter 11.

8

# Blackjack Card Game Project

# Review and Preview

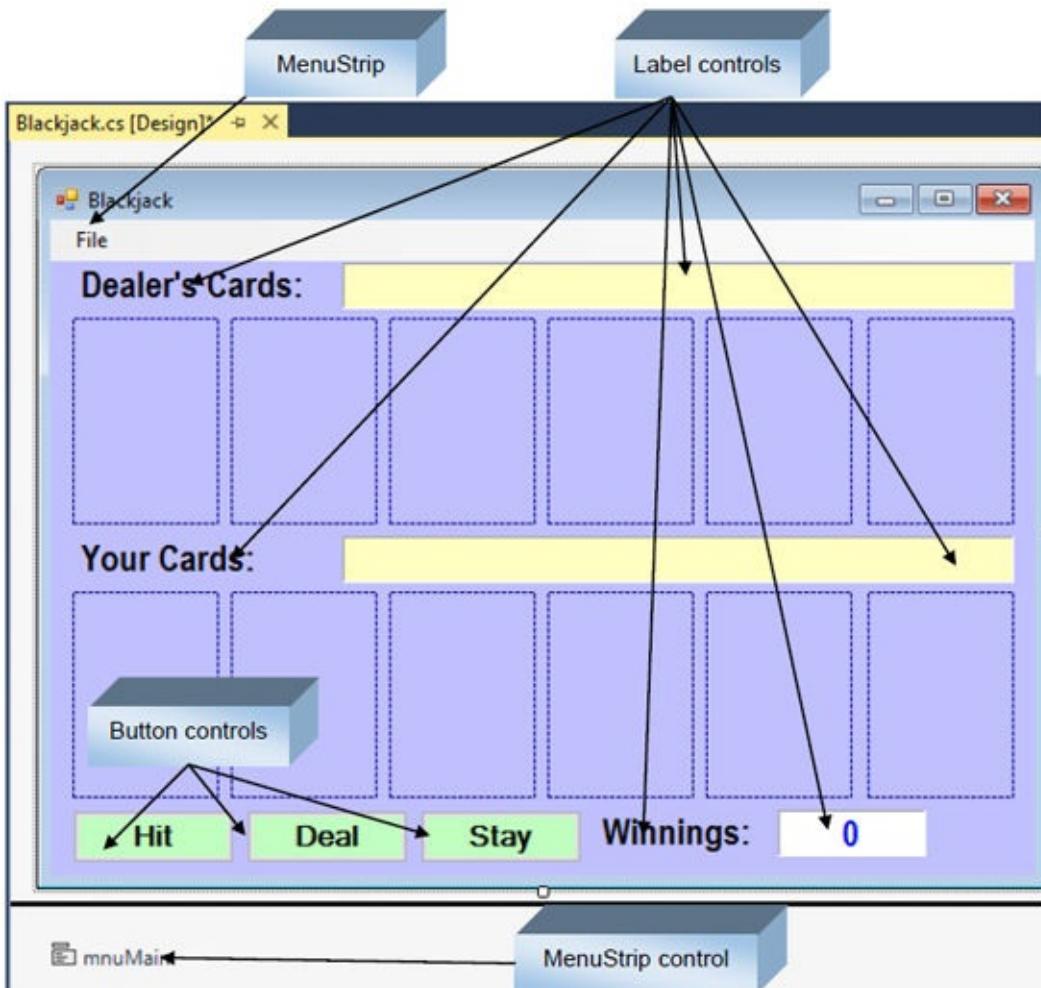
The first popular computer games appeared in the early 1970's with the introduction of timeshare computing. There was a classic set of DEC (Digital Equipment Corporation) programs written in BASIC for timeshare users. The set included gambling games, simulations and the ever-popular Star Trek game. A favorite DEC program was the casino card game Blackjack.

In this chapter, we build a version of that game. The **Blackjack Card Game Project** allows a single player to compete against the computer dealer. The project introduces use of the picture box control and discusses the math and logic involved in shuffling and displaying a deck of cards. You will see that the odds are stacked against you so keep your real money in your wallet!

# Blackjack Card Game Project Preview

In this chapter, we will build a **Blackjack card game** program. This program allows a single player to compete against the computer dealer. The idea of Blackjack is to score higher than the dealer's hand without exceeding twenty-one points. Cards count their value, except face cards (Jacks, Queens, Kings) count for ten, and Aces count for either one or eleven (you pick). If you beat the dealer, you get 10 points. If you get Blackjack (21 with just two cards) and beat the dealer, you get 15 points. If the dealer beats you, you lose 10 points.

The finished project is saved as **Blackjack** in the **HomeVCS\HomeVCS Projects** folder. Start Visual C# and open the finished project. Open the form (double-click **Blackjack.cs** in Solution Explorer) and you will see:



A menu strip is used to control the program Label controls are used for header information and to provide feedback and winnings information to the player. Three button controls are used by the player to ‘talk to’ the dealer. At the bottom is a menu strip control used to define the menu structure. The unidentified controls are picture box controls used to display the cards. The picture box is a new control and we will discuss it before building the project.

Defining the interface for this project is straightforward - the code behind the interface is not trivial. There are lots of rules involved with playing Blackjack and we need to determine some way to display the cards. We will build the code slowly. For now, let’s review the rules used in this version of Blackjack and see how the program works.

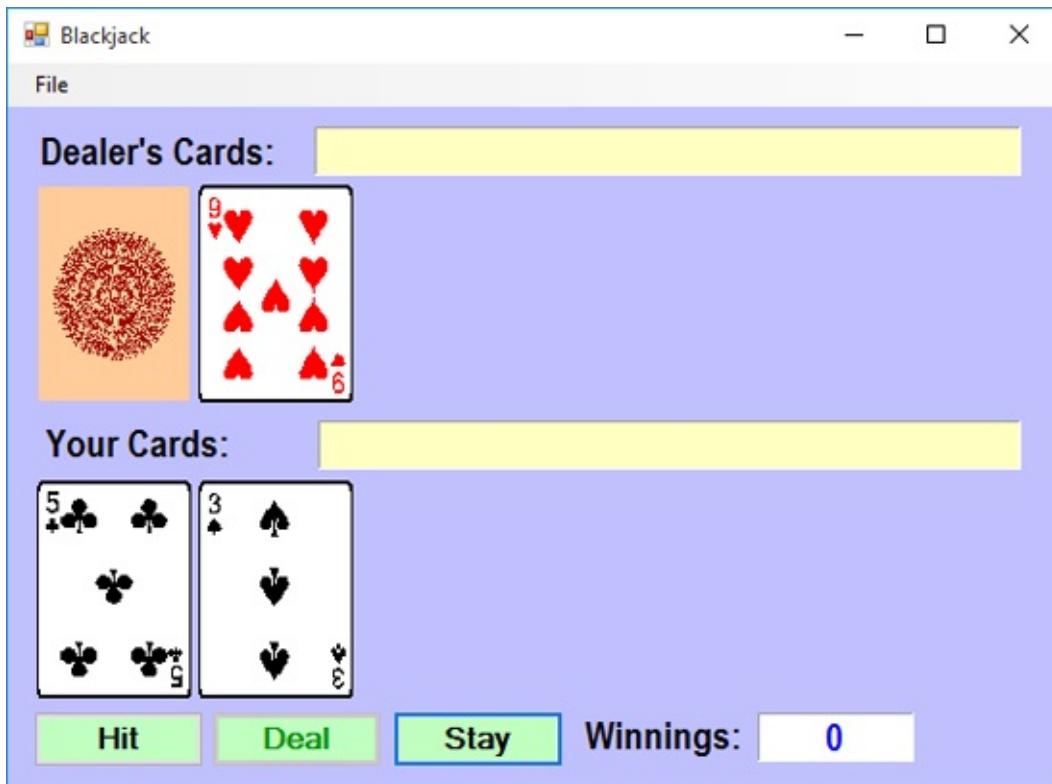
**Blackjack** starts by giving two cards (from a standard 52 card deck – reshuffles are done when only a few cards remain) to the dealer (one face down) and two cards to the player (you). The player decides whether to **Hit** (receive another card) or **Stay** (stop receiving cards). The player can choose as many extra cards as desired. If the player’s score exceeds 21 before staying, it is a loss (-10 points) and we say the player **busted**. If the player does not exceed 21, it becomes the dealer’s turn. The dealer must add cards to his score until 16 is exceeded. When this occurs, if the dealer also exceeds 21 (**busts**) or if his score is less than the player’s, he loses (+10 points for you). If the dealer’s score is greater than the player’s score (and under 21), the dealer wins (-10 points for you). If the dealer and the player have the same score, it is called a **push** (no points added or subtracted). The dealer must always take an Ace to be 11 points, unless it causes him to bust.

If either the player or dealer get ‘Blackjack’ which is defined as 21 points with just two cards (an Ace and a card worth 10 points), they automatically win. If the dealer gets Blackjack, the player loses 10 points. If the player gets Blackjack, he wins 15 points. If both the dealer and the player get Blackjack, it’s a push.

A special rule for this version of Blackjack (not used in casinos) involves the number of cards received. Theoretically, you can have eleven cards given to you and still not bust! We don’t want to display that many cards since it would be a rare occurrence. You can see in the interface, we limit the display to six cards. So, a special rule in this implementation is that, if the player gets six cards and has 21 or fewer points, the player is declared a winner. Similarly, if the player

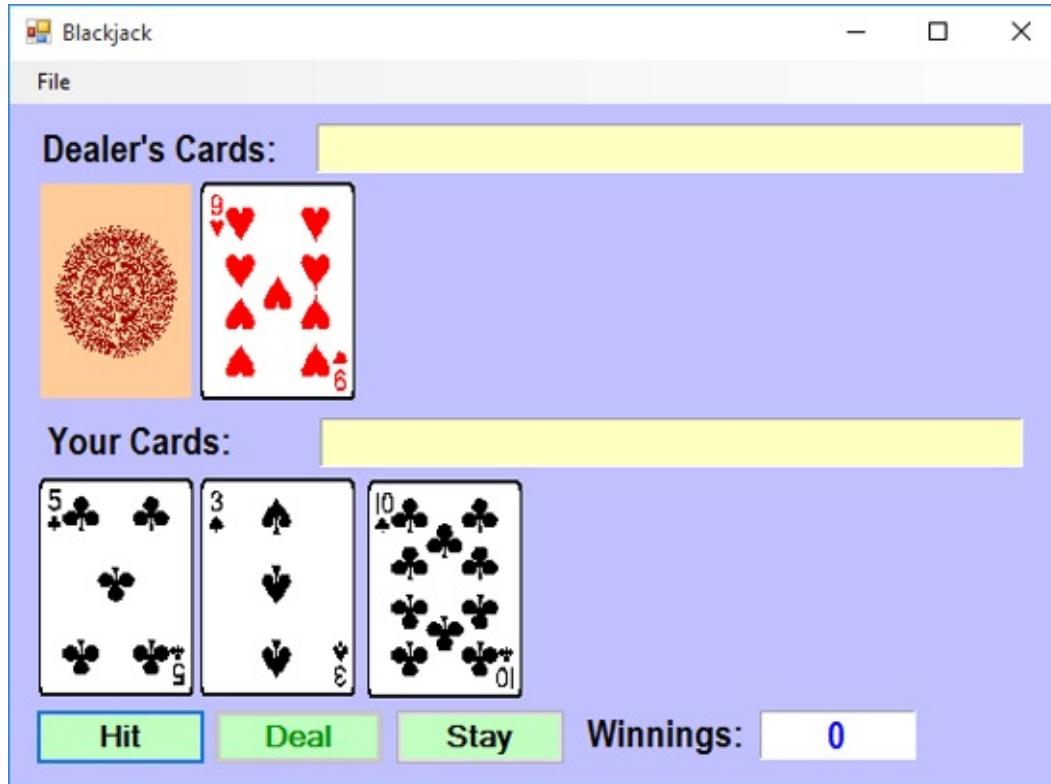
has fewer than six cards and the dealer is able to draw six cards without exceeding 16 points (since the dealer must stop adding cards after 16 points), the dealer wins, regardless of score.

Like we said – there are lots of rules here. Let's see these rules in action. Run the project (press <F5>). The Blackjack program will appear as (you will see different cards – the results are random):



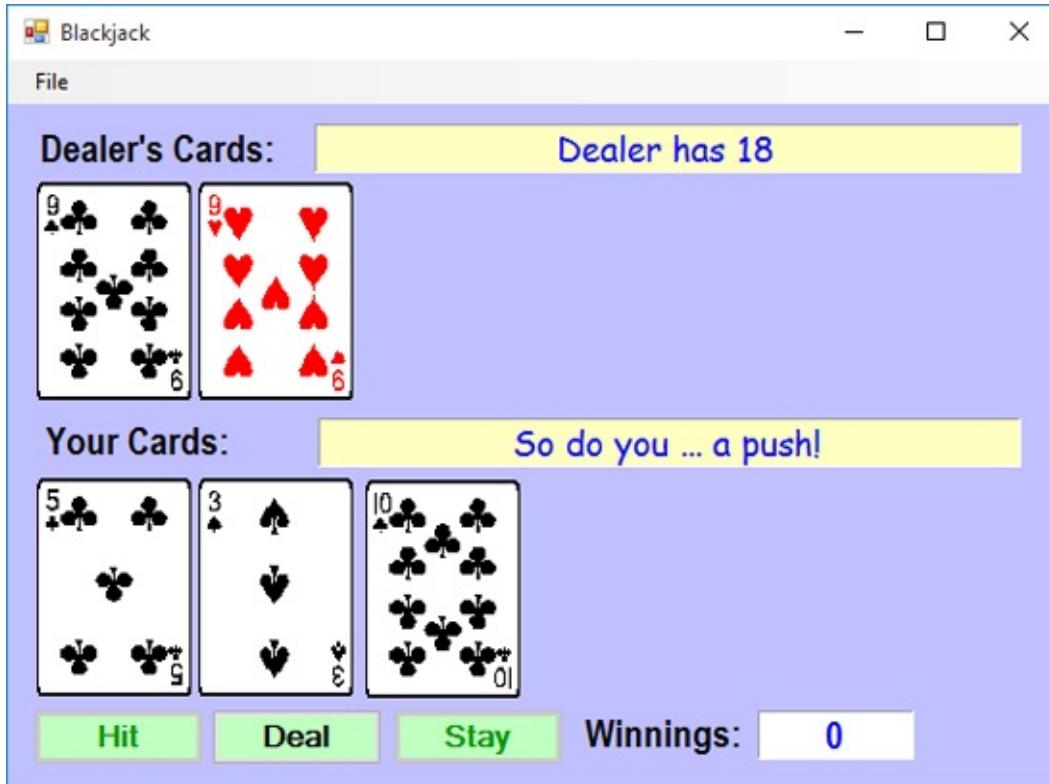
Notice the card graphics displayed in the picture box controls. In card lingo, the displayed cards are referred to **hands**. The dealer plays one hand, while the player plays the other hand. One of the dealer's cards is face down – the other is a 9. I have a 5 and a 3 showing (8 points). I can either get another card (**Hit**) or stop (**Stay**). I think you'd agree that **Hit** is the correct choice since I'm far from 21.

When I click **Hit**, I receive a Jack, worth 10 points.



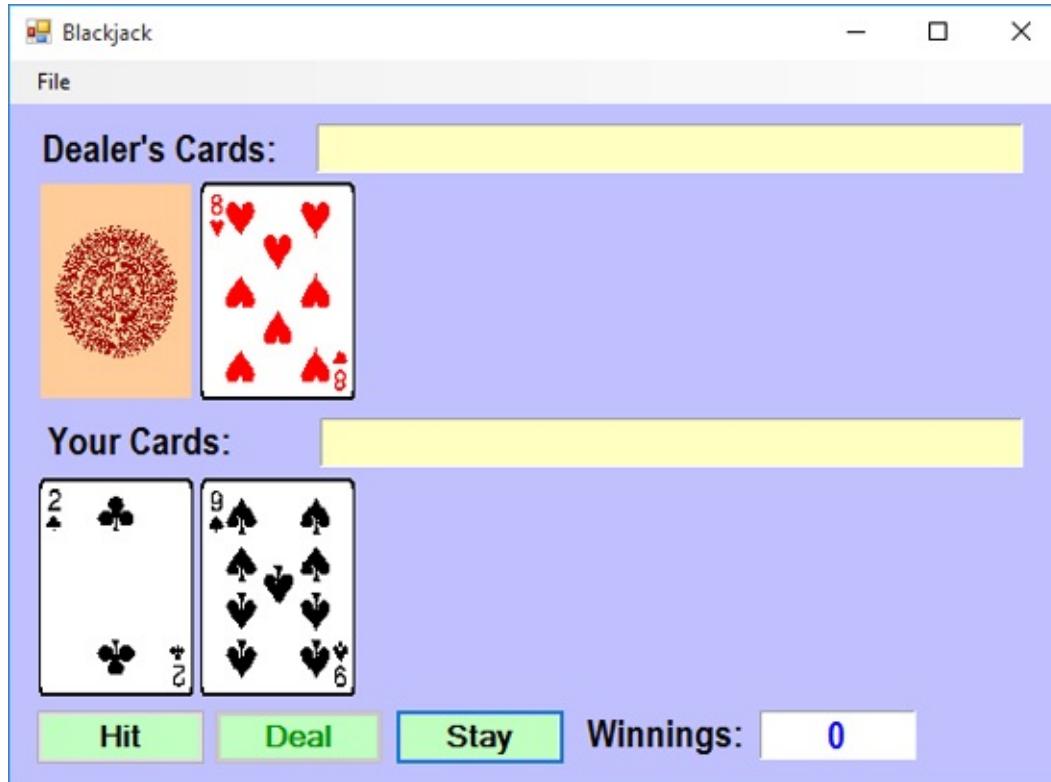
I now have 18 points – a good time to **Stay**.

After clicking **Stay**, the dealer plays out his cards according to the prescribed rules:



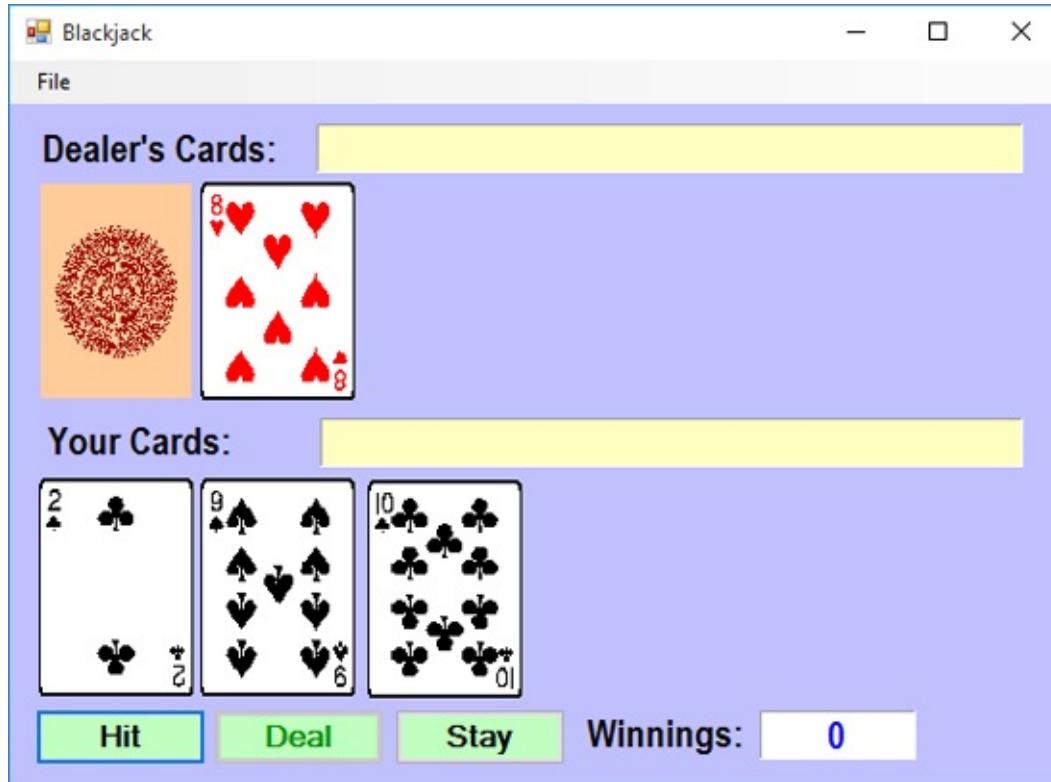
The first card is ‘flipped’ over revealing a 9, giving the dealer 18 points. We tied (called a push). Notice the label controls telling me the results and displaying my winnings (zero in this case). Click **Deal** to play another hand.

The next hand I see is:



I have 11 points, so I'll choose **Hit**.

When I click **Hit**, I receive a Jack, worth 10 points.



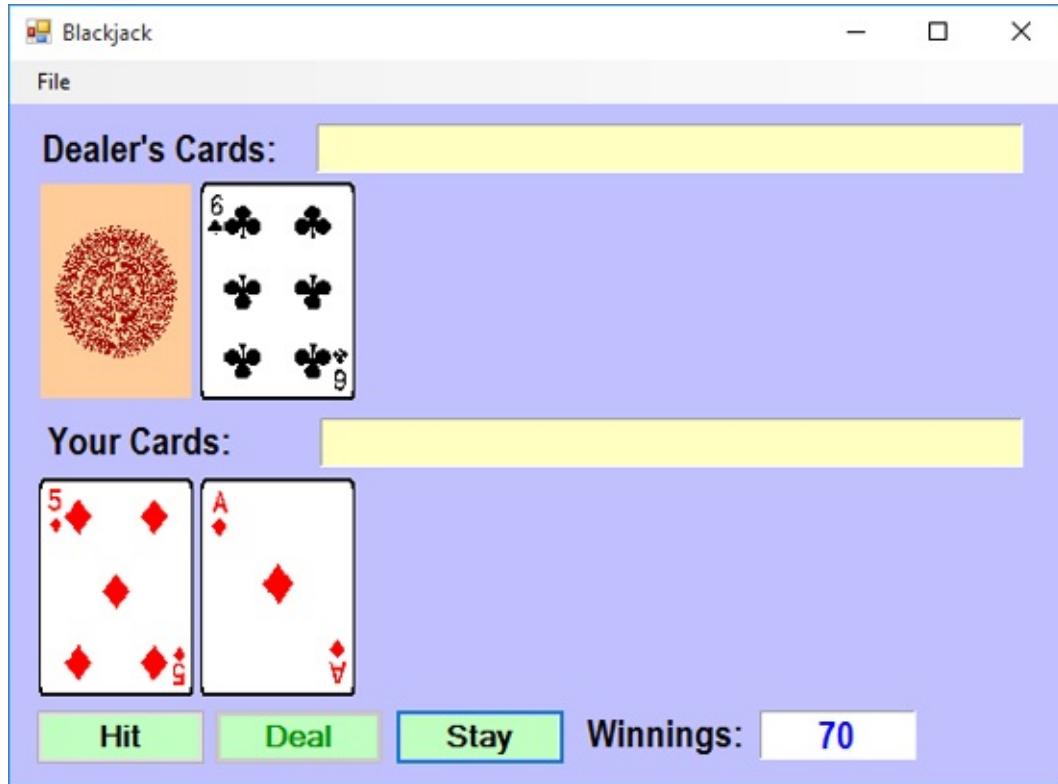
I now have 21 points. I can't do any better – a good time to **Stay**.

I click **Stay** and the dealer plays out:



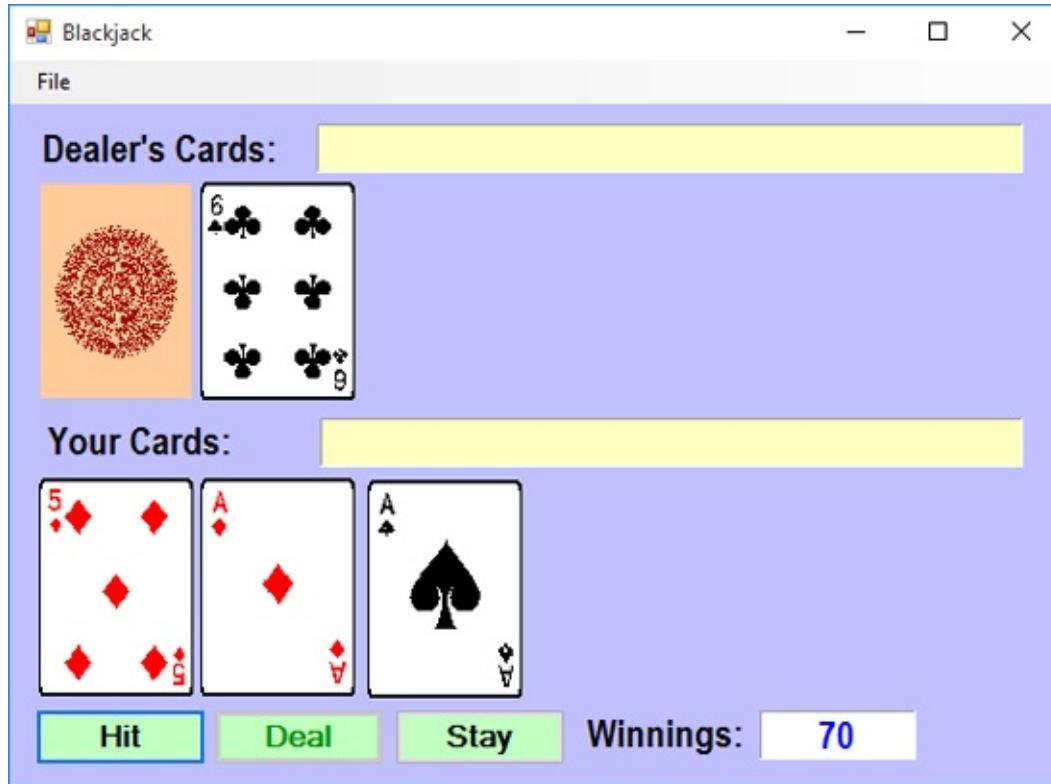
The face-down card is revealed to be an 6, giving him 20 points. I win! My winnings are 10.

After playing a few more hands (doing pretty well, since my **Winnings** are now **70**), I got these cards.:



An Ace can be either 1 or 11 points. Choosing 1 in this case gives me 6 points ( $5 + 1$ ). I choose to **Hit**.

After a **Hit**, I see:



Another Ace. If I now take one Ace to be 11 points and the other 1, I have 17 ( $5 + 11 + 1$ ), a good score. I choose to **Stay**.

After clicking **Stay**, the dealer plays his cards according to the rules:



The dealer reveals his first card (K), giving him 16 points ( $10 + 6$ ). The dealer takes another card (10), to give him 26 points. The dealer loses. I win again - ten more points. Maybe I should quit while I'm ahead.

As I play more hands, winning a few, losing a few, in one game I got Blackjack (21 points with two cards):



I won 15 points with this hand.

Continue playing hands until you understand how the rules of the game, especially those that the dealer uses, are applied. Try to figure out some good strategy for playing Blackjack. At any point, a new game can be started (resetting the winnings) by choosing the **New Game** option under the **File** menu. Selecting **Exit** under the **File** menu will stop the **Blackjack** program.

You will now build this project in several stages. We address **form design**. We discuss the controls used to build the form and establish initial properties. And, we address **code design** in detail. We will discuss how to shuffle a deck of cards and how to display the card graphics. We also cover the logic behind the complicated rules of play and how to determine who wins (or if it is a push). There is one new control in this project – the picture box control, used to display the card graphics. We will review its use before starting the project.

# PictureBox Control

**In Toolbox:**



**On Form (Default Properties):**



Visual C# has powerful features for graphics. The **PictureBox** control is a primary tool for exploiting these features. The picture box control can display graphics files (in a variety of formats), can host many graphics functions and can be used for detailed animations. Here, we concentrate on using the control to display a graphics file. In the Blackjack project, picture box controls are used to display card graphics.

## PictureBox Properties:

<b>Name</b>	Gets or sets the name of the picture box (three letter prefix for picture box name is <b>pic</b> ).
<b>BackColor</b>	Get or sets the picture box background color.
<b>BorderStyle</b>	Indicates the border style for the picture box.
<b>Height</b>	Height of picture box in pixels.
<b>Image</b>	Establishes the graphics file to display in the picture box.
<b>Left</b>	Distance from left edge of form to left edge of picture box, in pixels.
<b>SizeMode</b>	Indicates how the image is displayed.
<b>Top</b>	Distance bottom of form title bar area to top edge of picture box, in pixels.
<b>Width</b>	Width of picture box in pixels.

## PictureBox Events:

<b>Click</b>	Triggered when a picture box is clicked.
--------------	--

The **Image** property specifies the graphics file to display. Five types of graphics files that can be viewed in a picture box are:

<b>File Type</b>	<b>Description</b>
Bitmap	An image represented by pixels and stored as a collection of bits in which each bit corresponds to one pixel. This is the format commonly used by scanners and paintbrush programs. Bitmap filenames have a <b>.bmp</b> extension.
Icon	A special type of bitmap file of maximum 32 x 32 size. Icon filenames have an <b>.ico</b> extension. Icons are used to represent Windows applications.
Metafile	A file that stores an image as a collection of graphical objects (lines, circles, polygons) rather than pixels. Metafiles preserve an image more accurately than bitmaps when resized. Many graphics files available for download from the internet are metafiles. Metafile filenames have a <b>.wmf</b> extension.
JPEG	JPEG (Joint Photographic Experts Group) is a compressed bitmap format which supports 8 and 24 bit color. It is popular on the Internet and is a common format for digital cameras. JPEG filenames have a <b>.jpg</b> extension.
GIF	GIF (Graphic Interchange Format) is a compressed bitmap format originally developed by CompuServe. It supports up to 256 colors and is also popular on the Internet. GIF filenames have a <b>.gif</b> extension.

The **SizeMode** property dictates how a particular image will be displayed. There are five possible values for this property: **Normal**, **CenterImage**, **StretchImage**, **AutoSize**, **Zoom**. The effect of each value is:

<b>SizeMode</b>	<b>Effect</b>
Normal	Image appears in original size. If picture box is larger than image, there will be blank space. If picture box is smaller than image, the image will be cropped.
CenterImage	Image appears in original size, centered in picture box. If picture box is larger than image, there will be blank space. If picture box is smaller than image, image is cropped.
StretchImage	Image will ‘fill’ picture box. If image is smaller than picture box, it will expand. If image is larger than picture box, it will scale down. Bitmap files do not scale nicely. Metafiles, JPEG and GIF files do scale nicely.
AutoSize	Reverse of StretchImage - picture box will change its dimensions to match the original size of the image. Be forewarned – metafiles are usually very large!
Zoom	Similar to StretchImage. The image will adjust to fit within the picture box, however its actual height to width ratio is maintained.

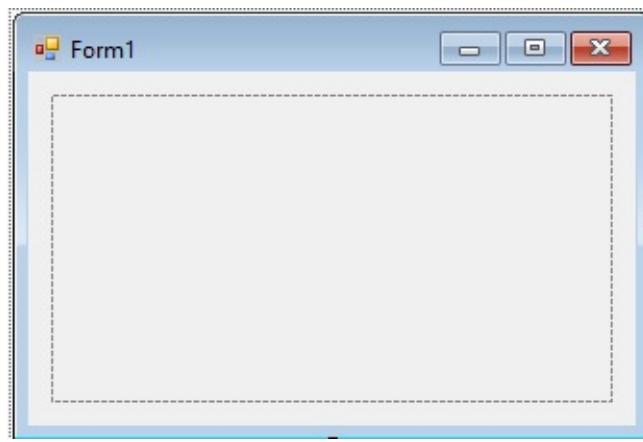
Notice picture box dimensions remain fixed for **Normal**, **CenterImage**, **StretchImage**, and **Zoom** **SizeMode** values. With **AutoSize**, the picture box will grow in size. This may cause problems at run-time if your form is not large enough to ‘contain’ the picture box.

Typical use of **PictureBox** control for displaying images:

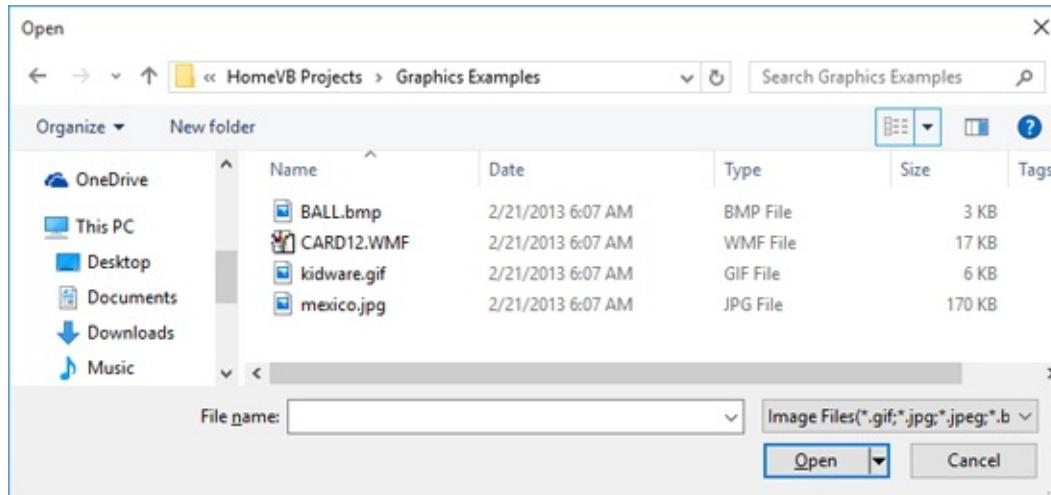
- Set the **Name** and **SizeMode** property (most often, **StretchImage**).
- Set **Image** property, either in design mode or at run-time.

# PictureBox Examples

Let's spend a little time looking at some example files, so you can see how different graphics files display, using different values of the **SizeMode** property. The **Graphics Examples** folder in the **HomeVCS\HomeVCS Projects** folder has several examples to view. Start a new project in Visual C#. Place a single picture box control on the form. Mine looks like this:

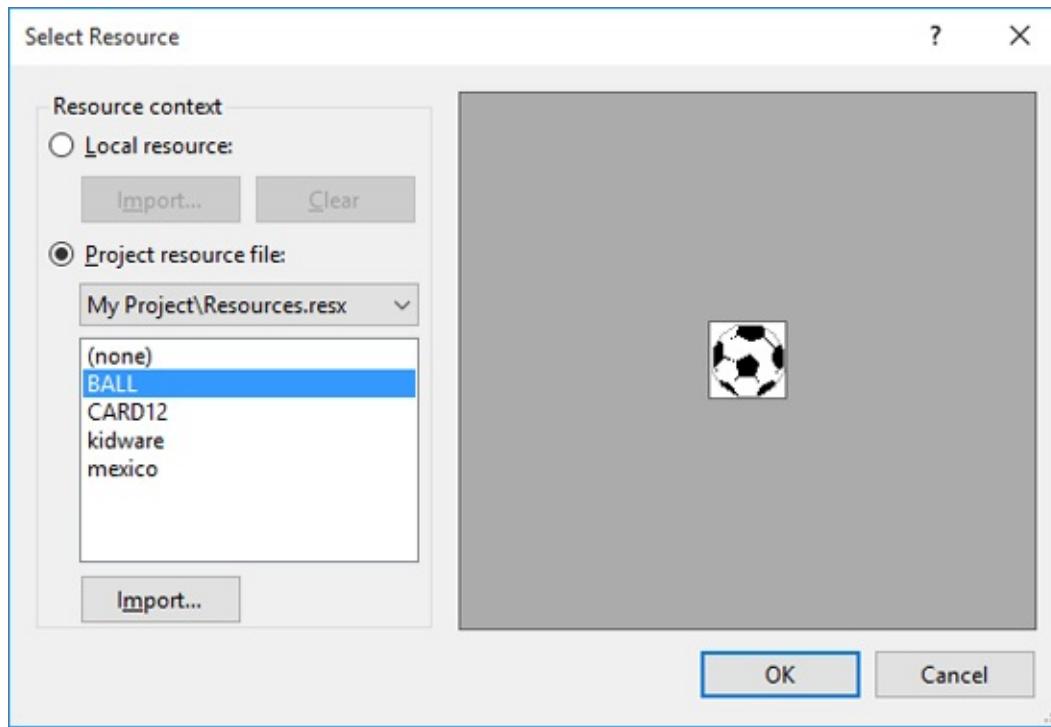


The **Image** property can be set in design mode or at run-time. Graphics files used at design-time are saved as program **Resources**. Once these resources are established, they can be used to set the **Image** property. The process to follow is to display the **Properties** window for the picture box control and select the **Image** property. An ellipsis (...) will appear. Click the ellipsis. A **Select Resource** window will appear. Make sure the **Project resource file** radio button is selected and click the button marked **Import** - a file open window will appear. Move (as shown) to the **\HomeVCS\HomeVCS Projects\Graphics Examples** folder and you will see our sample files listed:



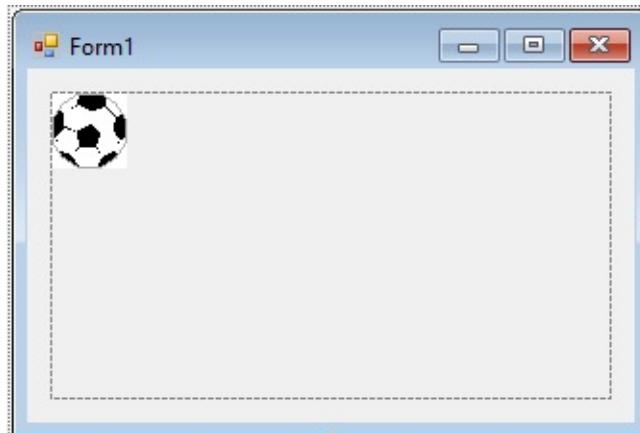
We have included one of each file type (except **ico** which is the same as **bmp** in behavior). There is a bitmap picture of a ball (**ball.bmp**), a metafile of a queen of hearts card used in Blackjack (**card12.wmf**), a copy of the KIDWare logo (**kidware.gif**) and a picture from my Mexico vacation (**mexico.jpg**). Select all four files and click **Open**.

You should now see the **Select Resource** window, with all four files now added to the program resources. Individual images are selected using this window:



Select the ball graphic (a bitmap) and click the **OK** button.

On your form, you should see something like this (depending on the size of your picture box):



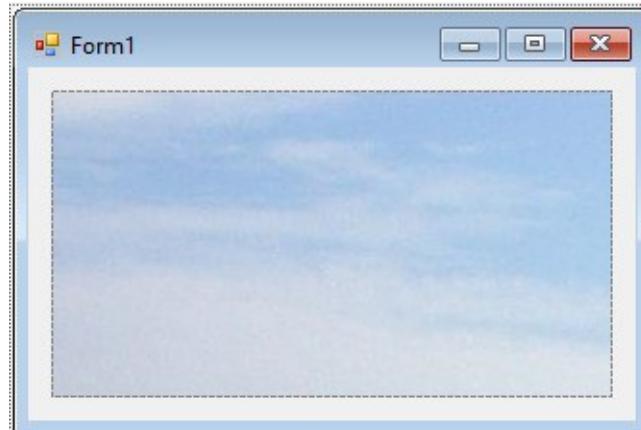
The image is in the upper left hand corner of the picture box. It appears in full-size.

Return to the properties window and choose the **kidware** logo gif file for the picture box Image property:



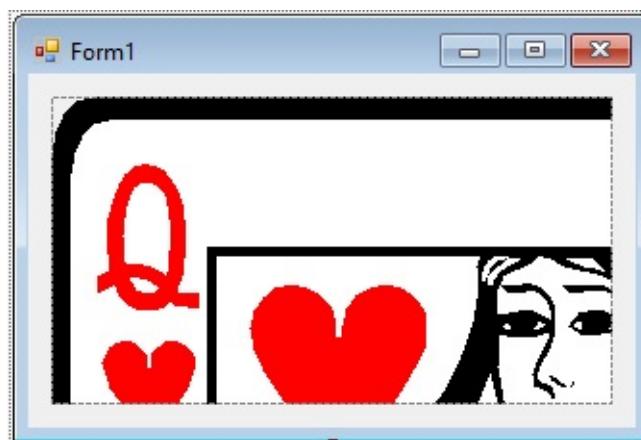
In this example, the picture box is shorter than the graphic, so the picture is vertically “cropped.” It is located in the upper left hand corner and appears in full-size. If the picture is cropped in your example too, you can resize the picture box control to see the entire graphic.

Load and view the **mexico** JPEG file:



There's not much to see here. The picture box is smaller than the photo, so only the sky is seen. The picture appears in full-size and is seriously cropped.

Lastly, load the card graphic (**card11.wmf**) to see:



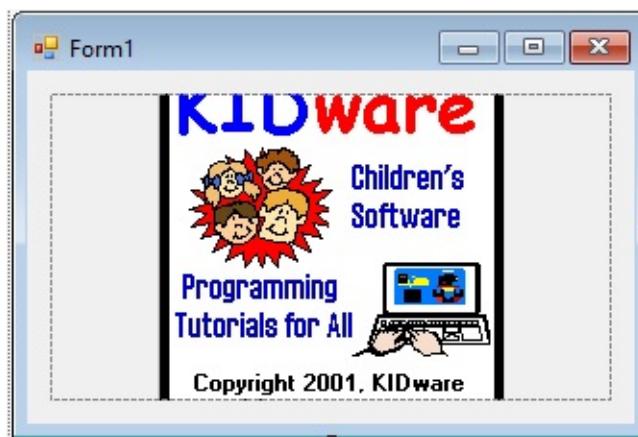
Again, lots of cropping going on here. We obviously can't use this display mode in our Blackjack game.

There are times you may want to delete the graphic displayed in a picture box. To do this, click **Image** in the properties window. In the right side of the window will be the current file (with a very tiny copy of the graphic). Select this information (double-click to highlight it) and press the keyboard **Del** key. The displayed picture will vanish and the property will read (**None**).

The examples shown have used the default **AutoSize** property of **Normal**. We see that the bitmap file seems to display satisfactorily. The GIF, JPEG and metafile files had cropping problems though. Other **SizeMode** values give us

some control on how we want a graphic to display. This will help us solve some of the problems we've seen. The most useful (in my opinion) of the **SizeMode** property choices is **StretchImage**. With this property, the image always fills the space you give it.

Continue the previous project. Change the **SizeMode** property of the picture box control to **CenterImage**. Reload each of the sample graphics files. With the **CenterImage** **SizeMode** property, note the difference in how the files are displayed. In particular, the GIF, JPEG and metafile are still cropped, but now they're centered in the control. For example, here is the **kidware** logo graphic with the image centered:



Change the **SizeMode** to **StretchImage**. Reload each of the sample graphics files. Notice how the graphic takes up the entire picture box. Here's the **mexico** graphic:



Try resizing the picture box control. How do the different graphics types resize?

After resizing the picture box control, does the picture still look recognizable? You should find that bitmaps (in most cases) scale poorly, while GIF, JPEG and metafile graphics scale very nicely. Change the **SizeMode** to **Zoom**. Notice the graphic displays are very similar to those seen with **StretchImage**. The mexico graphic appears clearer, since the height to width ratio are correct:



The Blackjack program will use metafiles to represent the card graphics. Metafiles are usually very large graphics files. As such, they should only be displayed using either the **StretchImage** or **Zoom SizeMode** properties. Change the picture box **SizeMode** to **StretchImage** and load the card metafile. Resize the picture so it has the shape of a card. You will have to resize the form also. Mine looks like this:



We see the entire card.

There are 52 cards in a standard deck. This means we will need and use 52 different metafiles to display cards in the **Blackjack** program. You may have a couple of questions at this point. The first is probably where do we get these files? The second is how are we going to set the **Image** property of the picture box controls used to display the cards.

We'll answer the first question first. The files provided with this program came from a CD collection with a title something like **100,000 Graphics for Your Computer**. Such collections are sold in the software section of your favorite computer superstore. Or, you can probably find similar graphics available for download from various Internet sites.

To answer the second question, we obviously can't set the **Image** properties of the picture box controls at design time. How can we set the Image if we don't know what card to display? In the **Blackjack** program, the Image properties will be set in code, at run-time, using the **FromFile** method associated with the **Image** object. As an example, to load the file **c:\HomeVCS\HomeVCS Projects\Graphics Examples\card11.wmf** into an Image object (either the Image property of a picture box control or a variable established as an Image object), the code statement is:

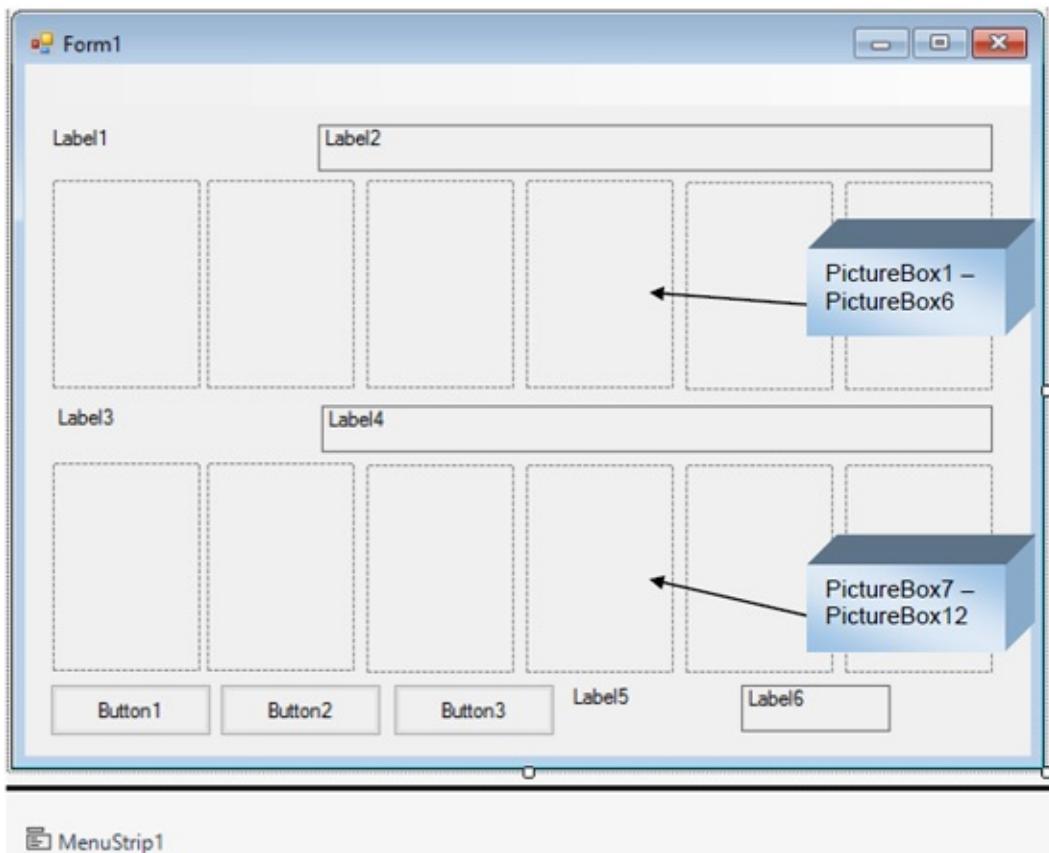
```
Image.FromFile("c:\HomeVCS\HomeVCS Projects\Graphics Examples\card11.wmf")
```

The argument in the **Image.FromFile** method must be a legal, complete path and file name, or your program will stop with an error message. When we develop the Blackjack code, we discuss where to place the card image files so they can be successfully loaded and displayed. We will also discuss how to determine which file to display.

To clear an image from a picture box control at run-time, simply set the corresponding Image property to **null** (a C# keyword). This disassociates the Image property from the last loaded image.

# Blackjack Form Design

We can begin building the Blackjack project. Let's build the form. Start a new project in Visual C#. Place six label controls (for three of them, set **AutoSize** to **False** to allow resizing), twelve picture box controls, and three button controls on your form. Add a menu strip control to the project. Resize and position controls so the form looks similar to this (for the resized labels, I've temporarily set the **BorderStyle** property to **FixedSingle** so you can see their size and location):



**label1**, **label3** and **label5** are used for titling information. **label2** and **label4** are used to say who won and who lost. And, **label6** displays your winnings. The first six picture box controls are used to display the dealer cards, while the other six display the player (your) cards. The three button controls are used to indicate if you wish to take a hit, stay or deal a new hand. The menu strip is used to start a new game and exit the program.

Establish the following menu structure and set the properties:

MenuStrip **mnuMain** structure:

```
File
  New Game
  -----
  Exit
```

MenuStrip **mnuMain** properties:

<b>Text</b>	<b>Name</b>
File	mnuFile
New Game	mnuFileNew
-	mnuFileBar (a <b>ToolStripSeparator</b> )
Exit	mnuFileExit

Set the other control properties using the properties window:

**Form1** Form:

<b>Property Name</b>	<b>Property Value</b>
Name	frmBlackjack
BackColor	LightBlue
BorderStyle	FixedSingle
StartPosition	CenterScreen
Text	Blackjack

**label1** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Dealer's Cards:
Font	Arial Narrow, Bold, Size 16

**label2** Label:

<b>Property Name</b>	<b>Property Value</b>
Name	lblDealerComment
Text	[Blank]
TextAlign	MiddleCenter
AutoSize	False
BorderStyle	Fixed3D
BackColor	Light Yellow
ForeColor	Blue
Font	Comic Sans MS, Size 14

**label3** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Your Cards:
Font	Arial Narrow, Bold, Size 16

**label4** Label:

<b>Property Name</b>	<b>Property Value</b>
Name	lblPlayerComment
Text	[Blank]
TextAlign	MiddleCenter
AutoSize	False
BorderStyle	Fixed3D
BackColor	Light Yellow
ForeColor	Blue
Font	Comic Sans MS, Size 14

**label5** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Winnings:
Font	Arial Narrow, Bold, Size 16

**label6** Label:

<b>Property Name</b>	<b>Property Value</b>
Name	lblWinnings
Text	0
TextAlign	MiddleCenter
AutoSize	False
BorderStyle	Fixed3D
BackColor	White
ForeColor	Blue
Font	Arial Narrow, Bold, Size 16

**button1** Button:

<b>Property Name</b>	<b>Property Value</b>
Name	btnHit
Text	Hit
BackColor	Light Green
Font Size	12
Font Style	Bold

**button2** Button:

<b>Property Name</b>	<b>Property Value</b>
Name	btnDeal
Text	Deal
BackColor	Light Green
Font Size	12
Font Style	Bold

**button3** Button:

<b>Property Name</b>	<b>Property Value</b>
Name	btnStay
Text	Stay
BackColor	Light Green
Font Size	12

Font Style      Bold

**pictureBox1** Picture Box:

<b>Property Name</b>	<b>Property Value</b>
Name	picDealer1
SizeMode	StretchImage

**pictureBox2** Picture Box:

<b>Property Name</b>	<b>Property Value</b>
Name	picDealer2
SizeMode	StretchImage

**pictureBox3** Picture Box:

<b>Property Name</b>	<b>Property Value</b>
Name	picDealer3
SizeMode	StretchImage

**pictureBox4** Picture Box:

<b>Property Name</b>	<b>Property Value</b>
Name	picDealer4
SizeMode	StretchImage

**pictureBox5** Picture Box:

<b>Property Name</b>	<b>Property Value</b>
Name	picDealer5
SizeMode	StretchImage

**pictureBox6** Picture Box:

<b>Property Name</b>	<b>Property Value</b>
Name	picDealer6
SizeMode	StretchImage

**pictureBox7** Picture Box:

<b>Property Name</b>	<b>Property Value</b>
Name	picPlayer1
SizeMode	StretchImage

**pictureBox8** Picture Box:

<b>Property Name</b>	<b>Property Value</b>
Name	picPlayer2
SizeMode	StretchImage

**pictureBox9** Picture Box:

<b>Property Name</b>	<b>Property Value</b>
Name	picPlayer3
SizeMode	StretchImage

**pictureBox10** Picture Box:

<b>Property Name</b>	<b>Property Value</b>
Name	picPlayer4
SizeMode	StretchImage

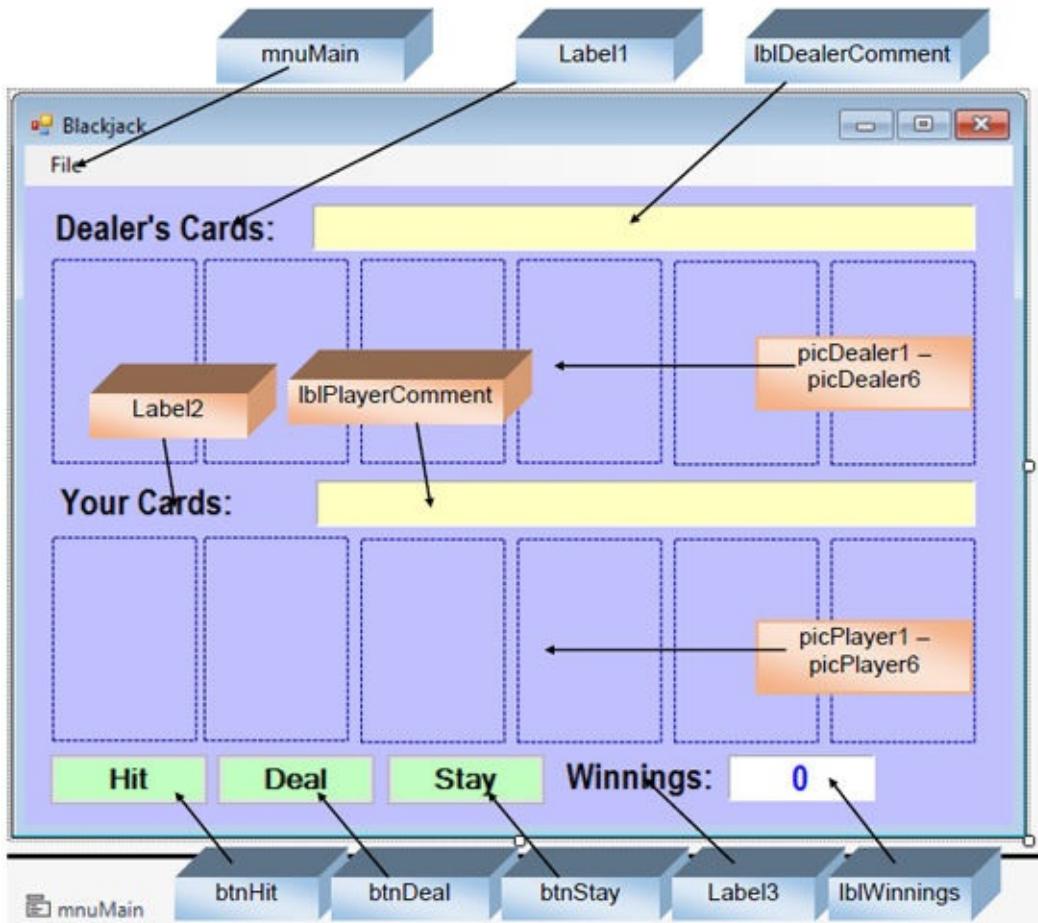
**pictureBox11** Picture Box:

<b>Property Name</b>	<b>Property Value</b>
Name	picPlayer5
SizeMode	StretchImage

**pictureBox12** Picture Box:

<b>Property Name</b>	<b>Property Value</b>
Name	picPlayer6
SizeMode	StretchImage

When done setting properties, my form looks like this:



We will begin writing code for the application. Many tasks are repeated in the Blackjack card game. We need to shuffle a deck of cards, deal a new hand, display cards for the dealer and player as play continues, and end a hand when a winner is declared. The approach we take is to build the code in modules that perform these repeated tasks. As we build the modules (general methods), we use them to write code for the event methods. One drawback to this modular approach is that we will have to write lots of code before anything can be tested. As a first step, we write the code that defines a deck of cards.

# Code Design – Card Definition

Defining a card consists of answering two questions: what is the card suit and what is the card value? The four suits are Hearts, Diamonds, Clubs, and Spades. The thirteen card values are: Ace (A), 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack (J), Queen (Q), King (K). Since there are 52 cards in a standard deck of playing cards, we will use integers from 0 to 51 (array indices) to represent the cards. How do we translate that card number to a card suit and value? (Notice the distinction between card **number** and card **value** - card number ranges from 0 to 51, card value can only range from Ace to King.) We need to develop some type of translation rule. This is done all the time in programming. If the number you compute with or work with does not directly translate to information you need, you need to make up rules to do the translation. For example, the numbers 1 to 12 are used to represent the months of the year. But, these numbers tell us nothing about the names of the month - we need a rule to translate each number to a month name.

We know we need 13 of each card suit. Hence, an easy rule to decide suit is: cards numbered 0 - 12 are Hearts, cards numbered 13 - 25 are Diamonds, cards numbered 26 - 38 are Clubs, and cards numbered 39 - 51 are Spades. For card values, lower numbers should represent lower cards. A rule that does this for each number in each card suit is:

**Card Numbers**

<b>Hearts</b>	<b>Diamonds</b>	<b>Clubs</b>	<b>Spades</b>	<b>Card Value</b>
0	13	26	39	A
1	14	27	40	2
2	15	28	41	3
3	16	29	42	4
4	17	30	43	5
5	18	31	44	6
6	19	32	45	7
7	20	33	46	8
8	21	34	47	9

9	22	35	48	10
10	23	36	49	J
11	24	37	50	Q
12	25	38	51	K

As examples, notice card number 11 is a Queen of Hearts. Card number 30 is a 5 of Clubs. These card numbers will be used to establish the graphics file associated with the card.

As an aside, if you have used KIDWare's **Beginning Visual C#** or **Visual C# for Kids** products, you will notice the rules for displaying cards here are slightly different than those used in the Card Wars project. In Card Wars, Ace was a high card, where here it is considered a low card.

As mentioned, a card number is used to establish the graphics file that represents the corresponding card. In the **HomeVCS\HomeVCS Projects\Card Graphics** folder are 52 graphics files (metafiles) that represent the 52 playing cards. These files are named **card00.wmf** to **card51.wmf**. And, yes, the file numbers (the last two digits in the name) correspond to the card numbers we've assigned. So **card11.wmf** is a Queen of Hearts and **card30.wmf** is a 5 of Clubs. So, once we know a card number, we know which file is used to display that card. Just how are these files used in the Blackjack program?

Two approaches can be taken to display cards in the Blackjack program. The first is that whenever a card must be displayed in a picture box control, we could load the appropriate file into a picture box control using the **Image.FromFile** function. In this approach, every time a card is needed, the program would have to find the file and load it from disk. This approach would require multiple accesses to disk files, slowing down the program. The second approach (and the one we use) is to preload all graphics files (still using the **Image.FromFile** function) into an array of **Image** objects. Then, when a card must be displayed, we simply set the **Image** property of the picture box control displaying a card to the **Image** object representing the card. This is a much faster approach and only requires opening the graphics files one time. The preloading of images is done in the form's **Load** event.

Before coding this **Load** event, we address where the graphics files should be

located in the project file structure. The accepted standard for using files (graphics files, sound files, data files, configuration files) associated with a project is to store those files in the same folder as the executable application (the so-called **exe** file). The folder holding the exe file is the **Bin\Debug** folder within your project folder (this folder is created by Visual C# when you start a new project). We will keep all the graphics files in this folder. Copy the 52 card graphics (plus the file **cardback.wmf**, which holds the graphics to represent the back of a card) into your project's **Bin\Debug** folder. If you want, open and view the **Bin\Debug** folder in the **HomeVCS\HomeVCS Projects\Blackjack** project to see these files in the included project.

Visual C# maintains a parameter that has the location of a project's executable file. That path is a string data type defined by:

### **Application.StartupPath**

We will use this parameter to load graphic files into **Image** objects. Let's show how.

We will define 53 **Image** objects for card display – one for the card back (**cardBack**) and an array of 52 images (**cardImage**) for the individual cards. These will be form level variables in our program. Add this single declaration to your project:

```
Image cardBack;
Image[] cardImage = new Image[52];
```

Now, the **frmBlackjack Load** event that establishes each **Image** object is:

```
private void frmBlackjack_Load(object sender, EventArgs e)
{
    string cn;
    // load card images and determine points for each
    cardBack = Image.FromFile(Application.StartupPath +
    "\cardback.wmf");
    for (int cardNumber = 0; cardNumber < 52; cardNumber++)
    {
```

```

        cn = cardNumber.ToString();
        if (cardNumber < 10)
            cn = "0" + cn;
        cardImage[cardNumber] =
Image.FromFile(Application.StartupPath + "\\card" + cn + ".wmf");
    }
}

```

In this code, we first set **cardBack**. Then, for all 52 cards, we form the appropriate file name using string functions and load the 52 **cardImage** values from file. Note use of the **Application.StartupPath** parameter and the need for an additional backslashes (\\\) in the path name. Two backslashes are used in C# to represent a single backslash to discriminate from control sequences (like \r\n).

Code the **frmBlackjack Load** event, then save and run the project. If the program runs without errors (the form appears), this tells you that all needed files are properly located in the **Bin\Debug** folder. If there are errors, you need to correct them.

At this point, we have an array (**cardImage**) of the graphics used to represent each of the 52 cards. In Blackjack, each card also has a point value. An Ace (initially, at least) is worth 1 point, the cards 2 through 10 have point values equal to their card value. And, the face cards (Jack, Queen, King) are each worth 10 points. A form level array (**cardPoints**) is used to hold the point value for each card. Add this declaration:

```
int[] cardPoints = new int[52];
```

Modify the **frmBlackjack Load** event to establish the elements of this array (changes are shaded):

```

private void frmBlackjack_Load(object sender, EventArgs e)
{
    string cn;
    // load card images and determine points for each
    cardBack = Image.FromFile(Application.StartupPath +

```

```

"\\cardback.wmf");
for (int cardNumber = 0; cardNumber < 52; cardNumber++)
{
    cn = cardNumber.ToString();
    if (cardNumber < 10)
        cn = "0" + cn;
    cardImage[cardNumber] =
Image.FromFile(Application.StartupPath + "\\card" + cn + ".wmf");
    int i = cardNumber % 13 + 1; // get a number from 1 (A) to 13 (K)
    if (i == 11 || i == 12 || i == 13) // Jack, Queen, King
        cardPoints[cardNumber] = 10;
    else // A through 10
        cardPoints[cardNumber] = i;
}
}

```

This new code uses the modulus (remainder) operator (%) to assign a point value to a card (**cardNumber**) from 0 to 51. The expression using the modulus operator converts any card number to a number (**i**) from 1 (Ace) to 13 (King), regardless of suit. The result is then used to assign the point value. Try a few values to convince yourself this works. Make the noted modifications. Save and run the project, if you'd like.

We now have all the information we need to define a card. The array **cardImage** has images for specific cards, while the array **cardPoints** has the corresponding point values. The index on the array, called the **card number**, ranges from 0 to 51 (Ace of Hearts to King of Spades). Let's learn how to “shuffle” these cards.

# Code Design – Card Shuffle

With 52 cards, we need to randomly sort the integers from 0 to 51 to “simulate” the shuffling process. How do we do this?

Usually when we need a computer version of something we can do without a computer, it is fairly easy to write down the steps taken and duplicate them in code. When we shuffle a deck of cards, we separate the deck in two parts, then interleaf the cards as we fan each part, making that familiar shuffling noise. I don’t know how you could write code to do this. We’ll take another approach which is hard or tedious to do off the computer, but is easy to do on a computer.

We perform what is called a “one card shuffle.” In a one card shuffle, you pull a single card (at random) out of the deck and lay it aside on a pile. Repeat this 52 times and the cards are shuffled. Try it! I think you see this idea is simple, but doing a one card shuffle with a real deck of cards would be awfully time-consuming. We’ll use the idea of a one card shuffle here, with a slight twist. Rather than lay the selected card on a pile, we will swap it with the bottom card in the stack of cards remaining to be shuffled. This takes the selected card out of the deck and replaces it with the remaining bottom card. The result is the same as if we lay it aside.

Here’s how the shuffle works with n numbers:

- Start with a list of n consecutive integers.
- Randomly pick one item from the list. Swap that item with the last item. You now have one fewer items in the list to be sorted (called the remaining list), or n is now n - 1.
- Randomly pick one item from the remaining list. Swap it with the item on the bottom of the remaining list. Again, your remaining list now has one fewer items.
- Repeatedly remove one item from the remaining list and swap it with the item on the bottom of the remaining list until you have run out of items. When done, the list will have been replaced with the original list in random order.

The code to do a one card shuffle, or sort n integers, is placed in a general method named **SortIntegers**. The single argument is **n** the number of integers to sort. The method returns an array containing the randomly sorted integers. The returned array is zero-based, returning random integers from 0 to n - 1, not 1 to n. If you need integers from 1 to n, just simply add 1 to each value in the returned array! The code is:

```
private int[] SortIntegers(int n)
{
    /*
     * Returns n randomly sorted integers 0 -> n - 1
     */
    int[] sortedArray = new int[n];
    int temp, s;
    Random sortRandom = new Random();
    // initialize array from 0 to n - 1
    for (int i = 0; i < n; i++)
    {
        sortedArray[i] = i;
    }
    // i is number of items remaining in list
    for (int i = n; i >= 1; i--)
    {
        s = sortRandom.Next(i);
        temp = sortedArray[s];
        sortedArray[s] = sortedArray[i - 1];
        sortedArray[i - 1] = temp;
    }
    return(sortedArray);
}
```

You should be able to see each step of the shuffle method. This method is general (sorting n integers) and can be used in other projects requiring random lists of integers.

Add the **SortIntegers** method to your Blackjack project. It will be used every time we need to shuffle the 52 cards. In the project, we will use a form level array **card** (dimensioned to 52) to hold the randomly sorted integers (the shuffled cards). A form level variable **currentCard** will be used to indicate the current index of the **card** array being used. Add these declarations to your project:

```
int[] card = new int[52];  
int currentCard;
```

The snippet of code that does a shuffle is:

```
card = SortIntegers(52);  
currentCard = 0;
```

In this code, we obtain the shuffled cards in **card** and set **currentCard** to zero so we are ‘pointing’ to the first card (array index zero) in the deck.

We can now use the shuffling process and card descriptions to begin building modules to play the Blackjack game.

# Code Design – Start New Game

To start a new **Blackjack** game, a user chooses **New Game** from the **File** menu. The steps in this method are:

- Set winnings to zero and reset winnings display.
- Shuffle cards.
- Start a new hand.

A form level variable **Winnings** is used to track the player's winnings. Add this declaration to the project:

```
int winnings;
```

The code for the **mnuFileNew Click** event method is:

```
private void mnuFileNew_Click(object sender, EventArgs e)
{
    // start new game - clear winnings and start over
    winnings = 0;
    lblWinnings.Text = "0";
    card = SortIntegers(52);
    currentCard = 0;
    NewHand();
}
```

Add this method to the project – the steps are obvious. This method uses a general method **NewHand** to start a new hand of Blackjack. We will code that next, but let's take care of a couple of other tasks first.

Add the **mnuFileExit Click** method:

```
private void mnuFileExit_Click(object sender, EventArgs e)
{
```

```
    this.Close();
}
```

When the Blackjack program first begins, we also want to start a new game. Add the shaded line of code to the form's **Load** method. This line will cause the **mnuFileNew Click** event method to be executed:

```
private void frmBlackjack_Load(object sender, EventArgs e)
{
    string cn;
    // load card images and determine points for each
    cardBack = Image.FromFile(Application.StartupPath +
    "\\cardback.wmf");
    for (int cardNumber = 0; cardNumber < 52; cardNumber++)
    {
        cn = cardNumber.ToString();
        if (cardNumber < 10)
            cn = "0" + cn;
        cardImage[cardNumber] =
        Image.FromFile(Application.StartupPath + "\\card" + cn + ".wmf");
        int i = cardNumber % 13 + 1; // get a number from 1 (A) to 13 (K)
        if (i == 11 || i == 12 || i == 13) // Jack, Queen, King
            cardPoints[cardNumber] = 10;
        else // A through 10
            cardPoints[cardNumber] = i;
    }
    mnuFileNew.PerformClick();
}
```

Now, we'll code the **NewHand** method.

# Code Design – Start New Hand

Each “round” of Blackjack begins with a new hand. In a new hand, two dealer cards (one face down) and two player cards are displayed and the interface is set so the player can begin playing his hand. Many steps are required to start a new hand:

- Clear all cards.
- Clear dealer and player comments.
- Enable **Hit** button.
- Enable **Stay** button.
- Disable **Deal** button.
- Reshuffle if necessary (if more than 35 cards have been used).
- Add two cards to dealer hand.
- Add two cards to player hand.
- Check if either hand is a Blackjack. If so, end the hand.

Six form level variables are used to know the status of the dealer and player hands. Add these declarations:

```
int numberCardsDealer, acesDealer, scoreDealer;  
int numberCardsPlayer, acesPlayer, scorePlayer;
```

**numberCardsDealer** tells us how many cards are currently in the dealer’s hand, **acesDealer** tells us how many of those cards are Aces, and **scoreDealer** tells us the dealer point total. We track Aces separately since their score can be either a 1 or 11. **numberCardsPlayer** tells us how many cards are currently in the player’s hand, **acesPlayer** tells us how many of those cards are Aces, and **scorePlayer** tells us the player point total. In the **NewHand** method, all of these will be initialized at zero, prior to adding cards to the hands.

The **NewHand** general method that implements the listed steps is:

```
private void NewHand()
```

```
{  
    // Deal a new hand  
    // Clear table of cards  
    picDealer1.Image = null;  
    picDealer2.Image = null;  
    picDealer3.Image = null;  
    picDealer4.Image = null;  
    picDealer5.Image = null;  
    picDealer6.Image = null;  
    picPlayer1.Image = null;  
    picPlayer2.Image = null;  
    picPlayer3.Image = null;  
    picPlayer4.Image = null;  
    picPlayer5.Image = null;  
    picPlayer6.Image = null;  
    lblDealerComment.Text = "";  
    lblPlayerComment.Text = "";  
    btnHit.Enabled = true;  
    btnStay.Enabled = true;  
    btnDeal.Enabled = false;  
    // reshuffle occasionally  
    if (currentCard > 34)  
    {  
        card = SortIntegers(52);  
        currentCard = 0;  
    }  
    // Get two dealer cards  
    scoreDealer = 0;  
    acesDealer = 0;  
    numberCardsDealer = 0;  
    AddDealerCard();  
    AddDealerCard();
```

```

// Get two player cards
scorePlayer = 0;
acesPlayer = 0;
numberCardsPlayer = 0;
AddPlayerCard();
AddPlayerCard();
// Check for blackjacks
if (scoreDealer == 11 && acesDealer == 1)
    scoreDealer = 21;
if (scorePlayer == 11 && acesPlayer == 1)
    scorePlayer = 21;
if (scoreDealer == 21 && scorePlayer == 21)
    EndHand("Dealer has Blackjack!", "And, you have Blackjack .. a
push!", 0);
else if (scoreDealer == 21)
    EndHand("Dealer has Blackjack!", "You lose ...", -10);
else if (scorePlayer == 21)
    EndHand("Dealer loses ...", "You have Blackjack!", 15);
}

```

Let's look at the **NewHand** method in a little detail. The cards are cleared by setting the picture box control **Image** properties to **null**. A reshuffle is done when **currentCard** is greater than **34**. The dealer hand status variables are set to zero and two cards are added to the dealer hand using a general method **AddDealerCard**. Similarly for the player's hand, two cards are added using **AddPlayerCard**. We will write these methods soon (they update the three status variables for the dealer and player).

The last part of the method checks each hand for Blackjack (having an Ace and a card worth 10 points, a 10, a Jack, a Queen, or a King). If either has a Blackjack, the general method **EndHand** is called. In this method, appropriate messages are displayed and the player's winnings are updated. The messages and winnings change are passed as arguments to the method.

Add the **NewHand** method to your project. We still can't test the project. We

still need three more methods which we'll code next – **EndHand**, **AddDealerCard**, **AddPlayerCard**.

# Code Design – End Hand

When a hand has ended, we want to tell the player whether he/she won and update their winnings. A new hand can then be dealt. The steps involved in ending a hand are:

- Display the dealer's face down card (just to make sure it is showing)
- Display dealer and player comments.
- Update **winnings** and display new value.
- Disable **Hit** button.
- Disable **Stay** button.
- Enable **Deal** button.

Since the first dealer card is shown face down (unless there is a Blackjack or until the player stays or busts), we need a variable to hold that card's image (**DealerFaceDown**). This variable will be established in the **AddDealerCard** method. Add this declaration to your project:

```
Image dealerFaceDown;
```

The **EndHand** general method that accomplishes the above tasks is:

```
private void EndHand(string dealerComment, string playerComment, int change)
{
    // make sure dealer cards are seen
    picDealer1.Image = dealerFaceDown;
    lblDealerComment.Text = dealerComment;
    lblPlayerComment.Text = playerComment;
    // Hand has ended - update winnings
    winnings += change;
    lblWinnings.Text = winnings.ToString();
    btnHit.Enabled = false;
```

```
btnStay.Enabled = false;
btnDeal.Enabled = true;
}
```

Note (as seen in **NewHand**) the dealer and player comments along with the amount to update the player's winnings are passed as arguments to the method. Add this method to the project.

Just two more methods and we can see if all this works! We need to add cards to the dealer and player hands.

# Code Design – Display Dealer Card

Here, we build a general method to add a card to the dealer's hand and display that card. The **currentCard** variable, used with the **card** array, identifies the card added to the dealer's hand. Recall three form level variables (**numberCardsDealer**, **acesDealer**, **scoreDealer**) are used to provide specifics about the dealer's hand. Also, recall **dealerFaceDown** saves the dealer's face down card.

Knowing **currentCard**, the steps involved in adding a card to the dealer's hand are:

- Determine **cardNumber** from the **card** array.
- Increment **numberCardsDealer**.
- If displaying first card:
  - o Set **dealerFaceDown** to **cardImage[cardNumber]**.
  - o Set **picDealer1 Image** to **cardBack**.
- If display second through sixth card:
  - o Set appropriate dealer card picture box control **Image** to **cardImage[cardNumber]**.
- Increment dealer's score by **cardPoint[cardNumber]**
- Increment **acesDealer**, if card is an Ace.
- Increment **currentCard**.

In these steps, if we are adding the first card, we save the image and display the card back. For other cards, the appropriate image is displayed. We then update the score, noting if an Ace has been added. As a last step, the current card index is incremented by one. At all times, we know the status of the dealer's hand (number of cards, number of aces and score).

The steps of the process to add a card to the dealer's hand are coded in a general method named **AddDealerCard**:

```
private void AddDealerCard()
```

```
{  
    int cardNumber;  
    cardNumber = card[currentCard];  
    // Adds a card to dealer hand  
    numberCardsDealer++;  
    switch (numberCardsDealer)  
    {  
        case 1:  
            dealerFaceDown = cardImage[cardNumber];  
            picDealer1.Image = cardBack;  
            break;  
        case 2:  
            picDealer2.Image = cardImage[cardNumber];  
            break;  
        case 3:  
            picDealer3.Image = cardImage[cardNumber];  
            break;  
        case 4:  
            picDealer4.Image = cardImage[cardNumber];  
            break;  
        case 5:  
            picDealer5.Image = cardImage[cardNumber];  
            break;  
        case 6:  
            picDealer6.Image = cardImage[cardNumber];  
            break;  
    }  
    scoreDealer += cardPoints[cardNumber];  
    if (cardPoints[cardNumber] == 1)  
        acesDealer++;  
    currentCard++;  
}
```

Add this method to your project. Notice the score (**scoreDealer**) always considers Aces as a single point. This may change when final hands are considered.

# Code Design – Display Player Card

The method to add a card to the player's hand is similar to the code just developed. The only difference is that there is never a 'face-down' card in the player's hand. The **currentCard** variable, used with the **card** array, identifies the card added to the player's hand. Three form level variables (**numberCardsPlayer**, **acesPlayer**, **scorePlayer**) are used to provide specifics about the player's hand.

Knowing **currentCard**, the steps involved in adding a card to the player's hand are:

- Determine **cardNumber** from the **card** array.
- Increment **numberCardsPlayer**.
- Set appropriate player card picture box control **Image** to **cardImage[cardNumber]**.
- Increment player's score by **cardPoints[cardNumber]**
- Increment **acesPlayer**, if card is an Ace.
- Increment **currentCard**.

In these steps, the appropriate image is displayed. We then update the score, noting if an Ace has been added. As a last step, the current card index is incremented by one. At all times, we know the status of the player's hand (number of cards, number of aces and score).

The steps of the process to add a card to the player's hand are coded in a general method named **AddPlayerCard**:

```
private void AddPlayerCard()
{
    int cardNumber;
    cardNumber = card[currentCard];
    // Adds a card to player hand
    numberCardsPlayer++;
}
```

```

switch (numberCardsPlayer)
{
    case 1:
        picPlayer1.Image = cardImage[cardNumber];
        break;
    case 2:
        picPlayer2.Image = cardImage[cardNumber];
        break;
    case 3:
        picPlayer3.Image = cardImage[cardNumber];
        break;
    case 4:
        picPlayer4.Image = cardImage[cardNumber];
        break;
    case 5:
        picPlayer5.Image = cardImage[cardNumber];
        break;
    case 6:
        picPlayer6.Image = cardImage[cardNumber];
        break;
}
scorePlayer += cardPoints[cardNumber];
if (cardPoints[cardNumber] == 1)
    acesPlayer++;
currentCard++;
}

```

Add this method to your project. Again, notice the score (**scorePlayer**) always considers Aces as a single point. This may change when final hands are considered.

After all the code we have added, we are finally at a point to try running the project. Save and run the project to make sure there are no syntax errors in the

code. If there are no errors, the form with the first hand should appear. Here's what I see:



You should see something similar, unless there is a Blackjack. If one of the first hands is a Blackjack, you will see messages to say so and the form will be set so a new hand can be dealt (the **Deal** button will be enabled).

If you encounter syntax errors in trying to run the project, you need to go back over all the code and see what went wrong. Hopefully, by taking things slow and step-by-step, fixing problems should be straightforward. In the current mode, the user can click **Hit** or **Stay**. If there is a Blackjack, the user can click **Deal**. The last remaining programming tasks are to code the **Click** events for these three buttons. The general methods we have written will help in this additional coding. The **Deal** button has the simplest coding, so we'll do it first.

# Code Design – Deal New Hand

When a hand has ended, the user can either start a new game, exit the program or deal a new hand. We have already coded event methods for starting a new game and exiting the program. Here we write code for dealing a new hand, once the user clicks the **Deal** button.

The code for the **btnDeal Click** method is made simple because of all the code we have already developed. It is a single line of code that calls the existing **NewHand** method:

```
private void btnDeal_Click(object sender, EventArgs e)
{
    NewHand();
}
```

Add this method to the project.

# Code Design – Player ‘Hit’

When a player chooses the **Hit** button, a new card is added to his/her hand and the results evaluated. The steps are:

- Add a player card.
- If player’s score exceeds 21, end hand announcing player has busted.
- If player has 6 cards, end hand announcing player has won.

As mentioned earlier in this chapter, this last step is a special rule used in our version of Blackjack.

The code is placed in the **btnHit Click** event method:

```
private void btnHit_Click(object sender, EventArgs e)
{
    // Add a card if player requests
    AddPlayerCard();
    if (scorePlayer > 21)
        EndHand("Dealer wins", "You busted!", -10);
    else if (numberCardsPlayer == 6)
        EndHand("No dealer play", "You win - 6 cards and not over 21!",
10);
}
```

Add this method to the project.

Save and run the project. Try the **Hit** button. Keep adding cards until you bust (exceed 21) or get 6 cards. You can’t choose to **Stay** – we need to write some code behind that method. Once you bust or get 6 cards, you can click **Deal** to try again. You won’t be able to test the **Hit** button if there is an initial Blackjack. In this case, click the **Deal** button until hands without a Blackjack appear. Then try the **Hit** button.

# Code Design – Player ‘Stay’

We save the most detailed event— clicking the **Stay** button - for last. Lots of things need to happen in this code. We need to determine player’s final score, then allow the dealer to play out his hand according to the fixed set of rules. There are lots of decisions to be made. The method steps are:

- Disable **Hit** button.
- Disable **Stay** button.
- Determine player’s highest possible score without exceeding 21 (accounting for any aces).
- Display the dealer’s face down card.
- Play dealer’s hand (repeat all steps until hand is ended):
  - o Determine dealer’s highest possible score without exceeding 21 (accounting for any aces).
  - o If dealer’s score is above 16, determine winner and end hand.
  - o If dealer has six cards and still under 16, end hand and declare dealer the winner.
  - o Add card to dealer’s hand.
  - o If above 21, end hand and declare player the winner.

As you can see most of the logic is in playing the dealer’s hand. Also, notice the special “six card” rule we use.

Determining either the player’s or dealer’s score (considering the possibility of Aces) is a little tricky. Let’s look at a snippet of code that does the task for the player:

```
if (acesPlayer != 0 && scorePlayer <= 11)
{
    scorePlayer += 10;
    acesPlayer--;
}
```

Recall the running score (**scorePlayer**) always considers Aces as one point. If the player has no Aces, there is no score adjustment. Otherwise 10 points is added to the score, if that adjusted score would not exceed 21. If a player has multiple Aces, only one can count for 11 points (would exceed 21, otherwise). Similar code is used for the dealer score.

The **btnStay Click** event method is:

```
private void btnStay_Click(object sender, EventArgs e)
{
    bool dealerDone = false;
    int scoreTemp, acesTemp;
    btnHit.Enabled = false;
    btnStay.Enabled = false;
    // Check for aces in player hand and adjust score
    // to highest possible
    if (acesPlayer != 0 && scorePlayer <= 11)
    {
        scorePlayer += 10;
        acesPlayer--;
    }
    // Uncover dealer face down card and play dealer hand
    picDealer1.Image = dealerFaceDown;
    do
    {
        scoreTemp = scoreDealer;
        acesTemp = acesDealer;
        // Check for aces and adjust score
        if (acesTemp != 0 && scoreDealer <= 11)
        {
            scoreTemp += 10;
            acesTemp--;
        }
    }
```

```

// add card unless score above 16 or dealer has 6 cards
if (scoreTemp > 16)
{
    if (scoreTemp > scorePlayer)
        EndHand("Dealer wins with " + scoreTemp.ToString(),
"You lose with " + scorePlayer.ToString(), -10);
    else if (scoreTemp == scorePlayer)
        EndHand("Dealer has " + scoreTemp.ToString(), "So do
you ... a push!", 0);
    else
        EndHand("Dealer loses with " + scoreTemp.ToString(),
"You win with " + scorePlayer.ToString(), 10);
    dealerDone = true;
    continue;
}
else if (numberCardsDealer == 6)
{
    EndHand("Dealer wins ... 6 cards and not over 16!", "You lose
...", -10);
    dealerDone = true;
    continue;
}
else
{
    AddDealerCard();
    // dealer loses if busted
    if (scoreDealer > 21)
    {
        EndHand("Dealer busts!", "You win!!", 10);
        dealerDone = true;
        continue;
    }
}

```

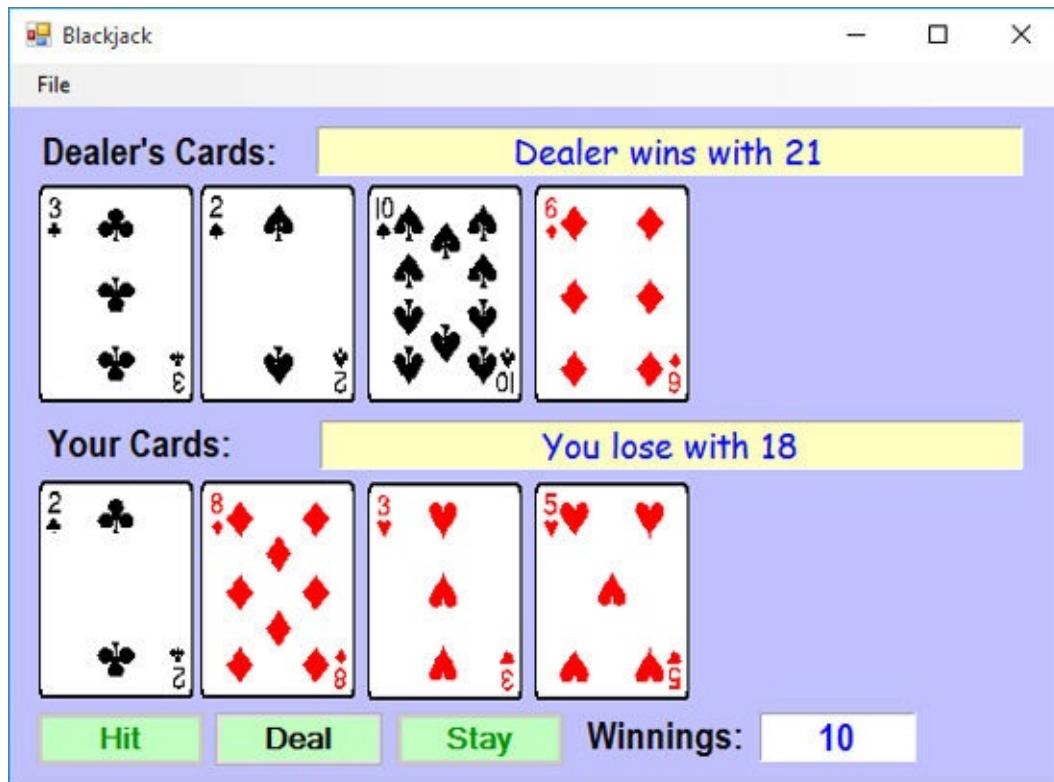
```

    }
    while (!dealerDone);
}

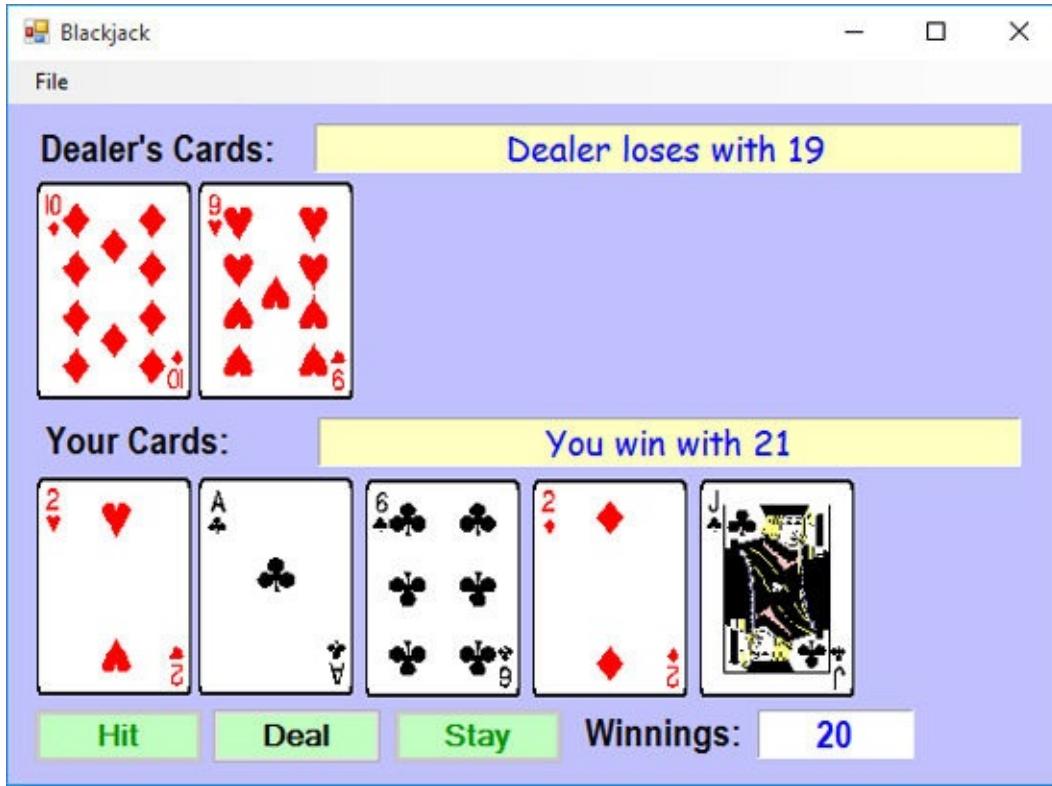
```

We use a Boolean variable **dealerDone** to let us know when the dealer is done playing his cards. Notice the code to determine player and dealer scores. For the dealer, we use temporary variables (**scoreTemp** and **acesTemp**) to represent the dealer score. We don't want to destroy the values of **scoreDealer** and **acesDealer** in case more cards may be added. A **do/while** structure is implemented to allow the dealer to continue to add cards to his hand until the hand ends. Notice whenever a call to **EndHand** is encountered, it is followed by setting **dealerDone** to **true** and a **continue** statement to move to the **while** statement so the dealer no longer adds cards. Add this final method to the project.

Save and run the project. You should now have a complete, running version of the Blackjack game. Have fun playing it! See if you can come up with some kind of winning strategy. Here's one of the first games I played. I lost after taking two hits:



In the next game, I won with 21 points:



In a later game, I won after taking three hits. Notice the Ace in my hand counts as 1 point:



And, in one game, I got Blackjack!!



# Blackjack Card Game Project Review

The **Blackjack Card Game Project** is now complete. Save and run the project and make sure it works as designed. Play lots of games to make sure winners are always declared correctly and that the dealer logic is implemented correctly. You may have to play lots of hands before both the dealer and player have Blackjack. And, you may have to play many, many hands to see if the special “six card” rule works correctly.

If there are errors in your implementation, go back over the steps of form and code design. Use the debugger when needed. Go over the developed code – make sure you understand how different parts of the project were coded. As mentioned in the beginning of this chapter, the completed project is saved as **Blackjack** in the **HomeVCS\HomeVCS Projects** folder.

While completing this project, new concepts and skills you should have gained include:

- Using the picture box control to display graphics.
- How the **SizeMode** property affects the display of different file types.
- How to define a deck of cards using card number indices.
- How to use the **Image** object to store a graphics file.
- How to “shuffle” a deck of cards using the **SortIntegers** method.

# Blackjack Card Game Project Enhancements

Possible enhancements to the Blackjack card game project include:

- As you probably know, Blackjack is a gambling game. The idea is for you (the player) to win as much money as possible from the dealer. Our version of Blackjack is a simplification of the casino version. The casino version allows betting – our version doesn't (you either win or lose 10 points with each hand; well, you win 15 if you get a Blackjack). Unfortunately, Casinos tend to make a lot of profit from uneducated players. Our purpose here is to educate you on how the odds are stacked against you. We recommend you enjoy this game at home and keep your hard earned money in your wallet!
- Casinos also allow you, in certain cases, to double your bet after a hand has been dealt. And, if your two initial cards are the same, you can split them and play two hands. Not being a gambler, I don't know all the specifics behind "double-down" and "splitting." Ask some who does or consult a gambling guide. If you want, implement these modifications to the program.
- Some casinos have different rules for dealer play. In our version, an Ace must always take on its highest value (without exceeding 21 of course). In other versions, the dealer has discretion. Perhaps, you would like to give the dealer in your program this discretion.
- To make play more difficult, some casinos play Blackjack with more than one deck of cards. Maybe have the number of card decks being used be an option in your program. In such a case, or even in the current configuration, it might be nice to announce to the player when a reshuffle of the cards is done.
- Now that you know the high risks involved with gambling, let's move on to a more practical application.

9

## Weight Monitor Project

# Review and Preview

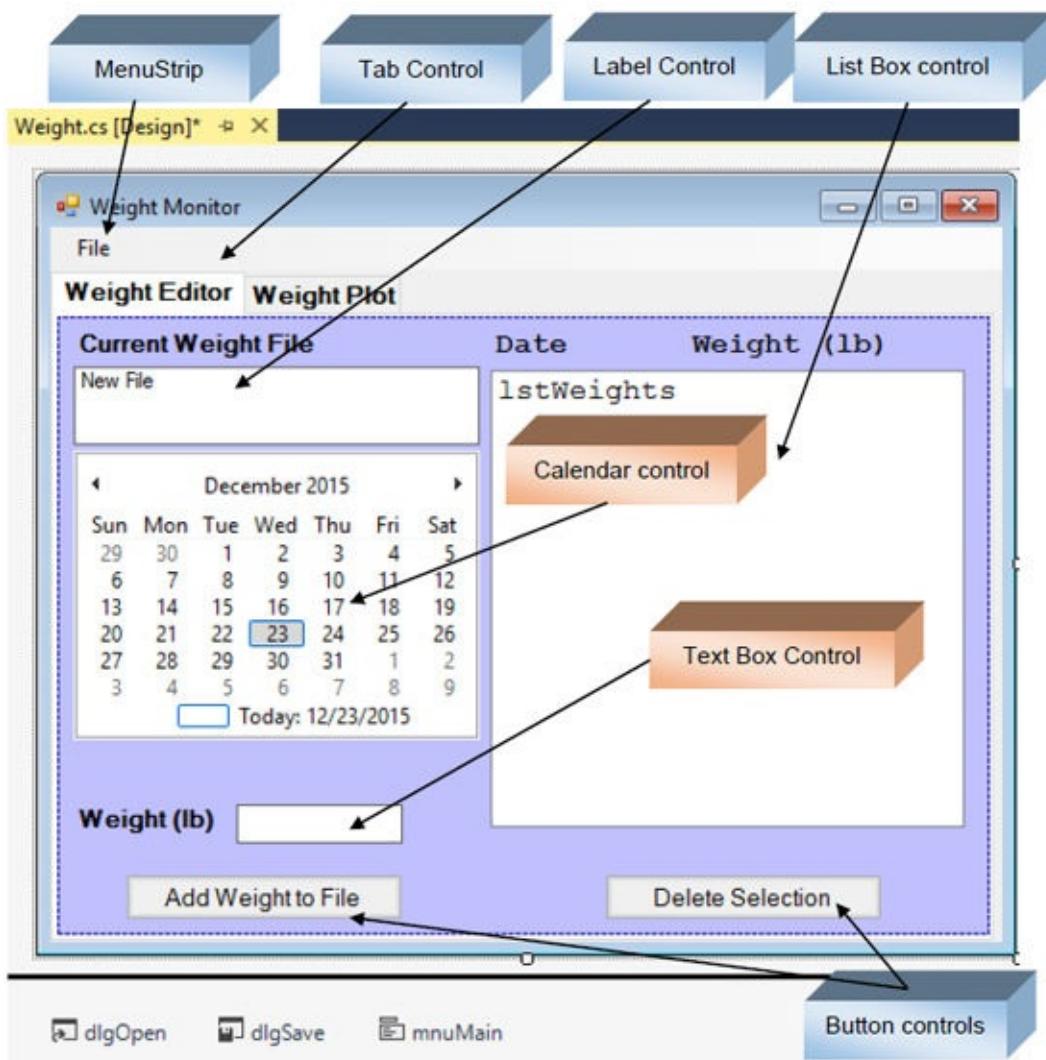
Everyone these days seems to be watching their weight. In this project, we build a program that tracks your weight each day and helps you monitor progress toward goals. The **Weight Monitor Project** lets you choose a date from a calendar and enter your weight on that day.

Plots of your daily weight are provided along with a computation of the trend in your weight. New controls (tab control, list box, save file dialog) are introduced as is sequential file input and output.

# Weight Monitor Project Preview

In this chapter, we will build a **weight monitor** program. This program allows you to enter your weight each day, then examine a plot to observe trends.

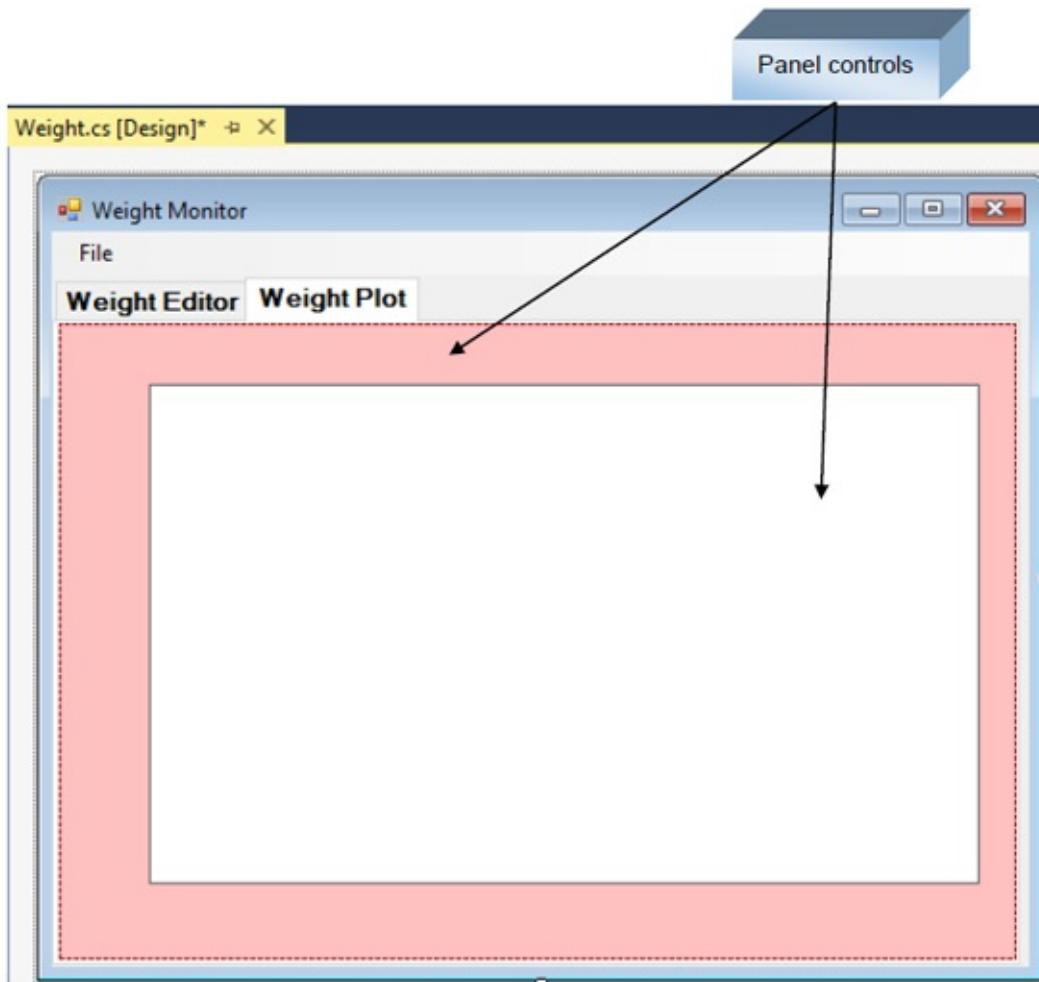
The finished project is saved as **Weight** in the **HomeVCS\HomeVCS Projects** folder. Start Visual C# and open the finished project. Open the form (double-click **Weight.vb** in Solution Explorer) and you will see:



This project is built using a tab control which allows multiple pages (tabs) of information on a single form. There are two tabs: **Weight Editor**, **Weight Plot**. Each tab has a single panel control upon which other controls are placed. We

initially see the controls on the **Weight Editor** tab panel. Most of the labels (used for titling) are unidentified. A label control displays the most recent weight file. A calendar control is used to select the date and a text box used to enter a weight value. Button controls are used to add and delete entries from the weight file, which are displayed in the list box control. At the bottom are a menu strip control used to specify the simple menu structure and two dialog controls used to open and save weight files. The tab, list box and save file dialog controls are all new – we will discuss these in some detail.

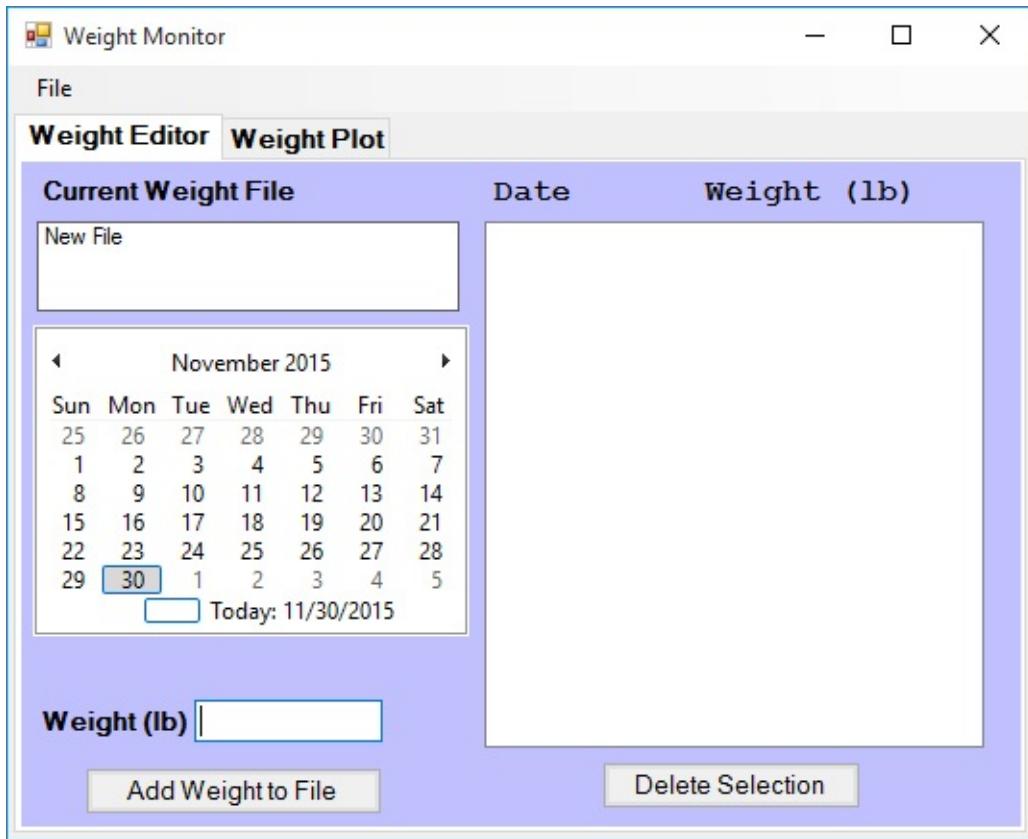
Click the **Weight Plot** tab and you will see:



On this tab is a single panel with another smaller panel control (will be used to show the weight plot).

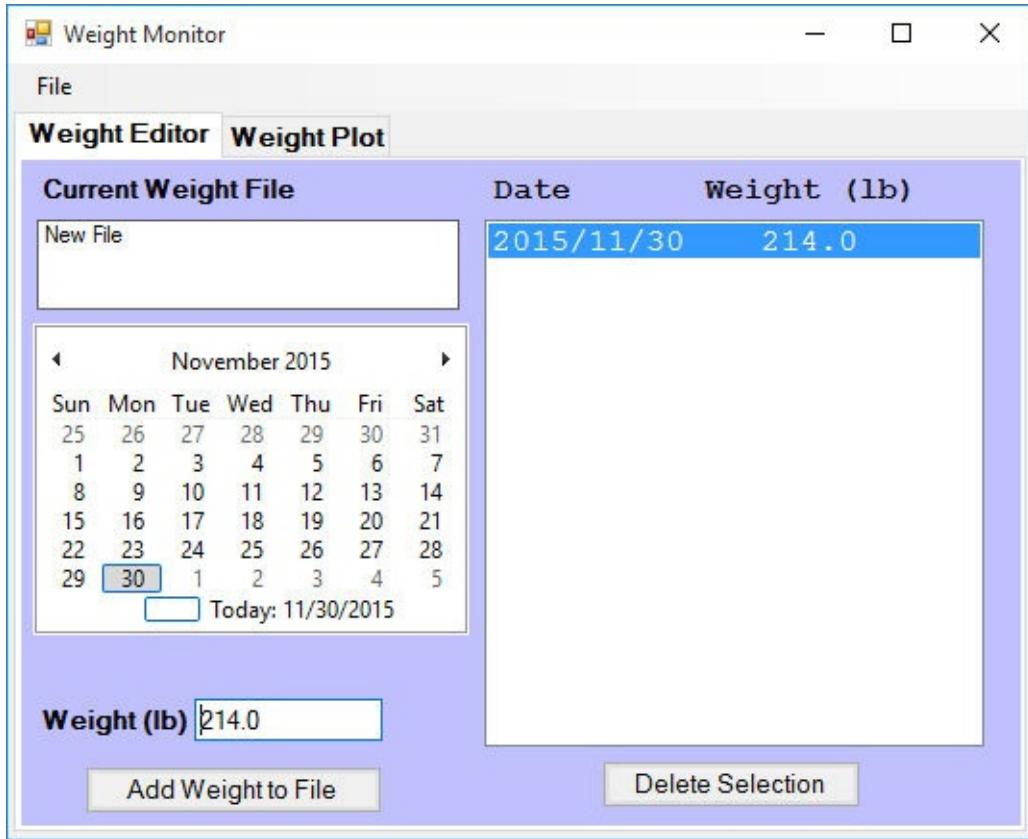
The normal way to use the weight monitor is to run the program, open an

existing weight file, modify it with new entries, view the trends, resave the file and exit. A nice feature of the weight monitor program is that, when it begins, it will automatically open the last opened/saved file (saving you that step for daily recording). Run the project (press <F5>). The weight monitor program will appear in an initial condition (since no file has been saved yet) with the **Weight Monitor** tab displayed:



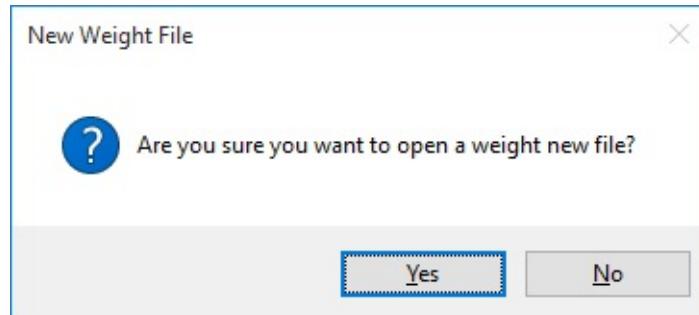
The program indicates we are working with a new file. Today's date is displayed on the calendar. At this point, you enter a weight in the text box control and press <Enter> or click **Add Weight to File**. Give it a try.

When I enter my weight, I see:



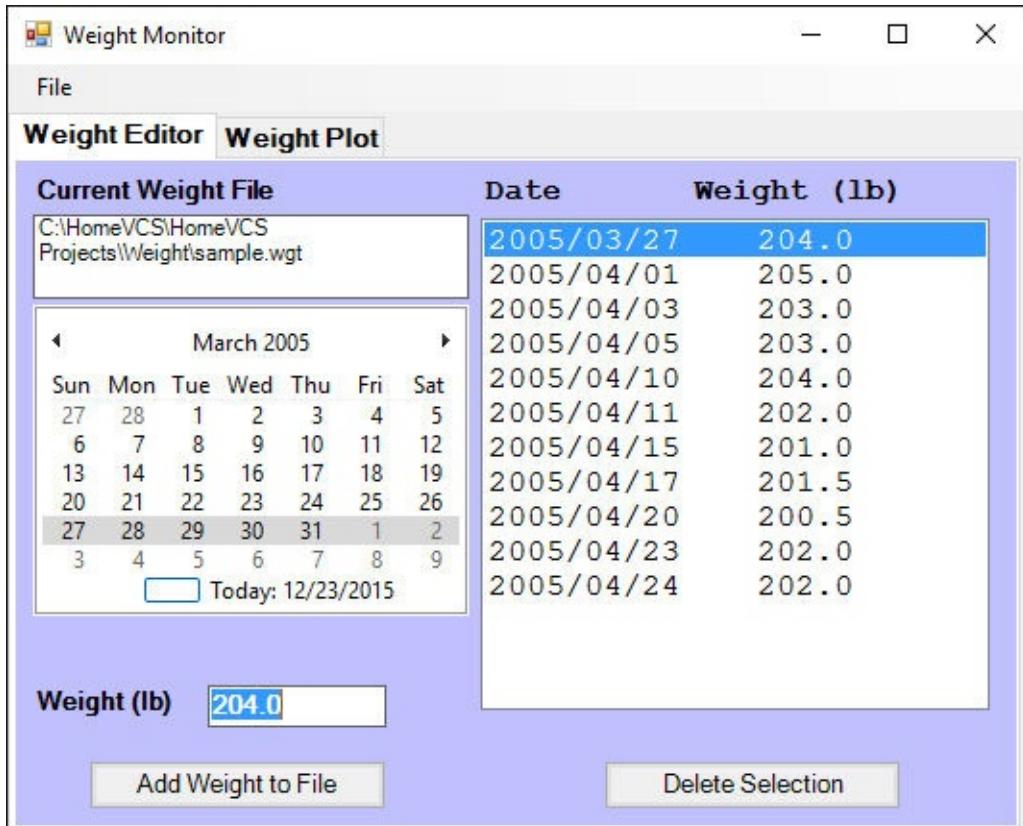
Notice the date and weight have been added to the list box control. And, that's what the program does – it records your weight each day. By running the program periodically, you will have a log of your weight that you can plot and view any trends. Try adding more values (by selecting other dates on the calendar) if you'd like. A file is saved using the **Save Weight File** option under the **File** menu. The **File** menu can also be used to start a new with file or open previously saved weight files.

In the **HomeVCS\HomeVCS Projects\Weight** folder is a sample weight file named **sample.wgt**. Use the **Open Weight File** option under the **File** menu to open that file. You will see a message box:



Answer **Yes**. The program always asks before you want to change the displayed information. Such protective mechanisms can save you (and your users) from losing important data. An open file dialog control will appear. Navigate to the **sample.wgt** file and click **Open**.

Here's my **Weight Editor** tab after opening the sample file:



The weight monitor program allows you to **add** (we've seen how to do that), **delete** or **modify** entries. To delete an entry, select that entry in the list box control and click **Delete Selection**. To edit an entry, select that entry in the list box; the corresponding date will appear on the calendar control and the weight in the text box. Make any changes and click **Add Weight to File**. The calendar control and list box entries are always coordinated (assuming there is a list box entry matching the selected date). See that coordination in the example above (the March 10 entry is shown). Try adding, deleting and modifying weight entries, if you want. At some point, click the **Weight Plot** tab.

Here's what I see on the **Weight Plot** tab (I didn't modify any of the entries):



The program has provided a nice line plot of the recorded weights over the specified time period. At the top of the plot, I am shown an indication of the trend in my weight (going down at 0.79 pounds each week – a good trend).

That's what you do with the weight monitor project. Periodically enter your weight in a saved file using the **Weight Editor** tab. View your weight trends using the **Weight Plot** tab. To stop the program, select **Exit** under the **File** menu. The program will automatically save the last file opened and/or saved. After stopping, if you restart the program (assuming you opened the sample weight file), the weight monitor will automatically display the **sample.wgt** file (the last file opened/saved). Play with the program some more if you want. Start a new file, select some dates, enter some weights, view your plot. Save your file.

You will now build this project in several stages. We first address **form design**. We discuss the controls used to build each tab page on the form and establish initial properties. And, we address **code design** in detail. We discuss how to open/save/edit the weight files. We discuss the graphics methods behind generating a line plot like the one used to display the weight trend. Three new

controls are introduced in this project – the tab control, the list box control and the save file dialog control. We will review their use before starting the project.

# TabControl Control

## In Toolbox:



## On Form (Default Properties):



The **TabControl** control provides an easy way to present several dialogs or screens of information on a single form. This is the same interface seen in many commercial Windows applications. The tab control provides a group of tabs, each of which acts as a container (works just like a group box or panel) for other controls. In particular, groups of radio buttons within a tab ‘page’ operate as an independent group. Only one tab can be active at a time. Using this control is easy. Just build each tab container as a separate group: add controls, set properties, and write code like you do for any application. Navigation from one tab to the next is simple: just click on the corresponding tab.

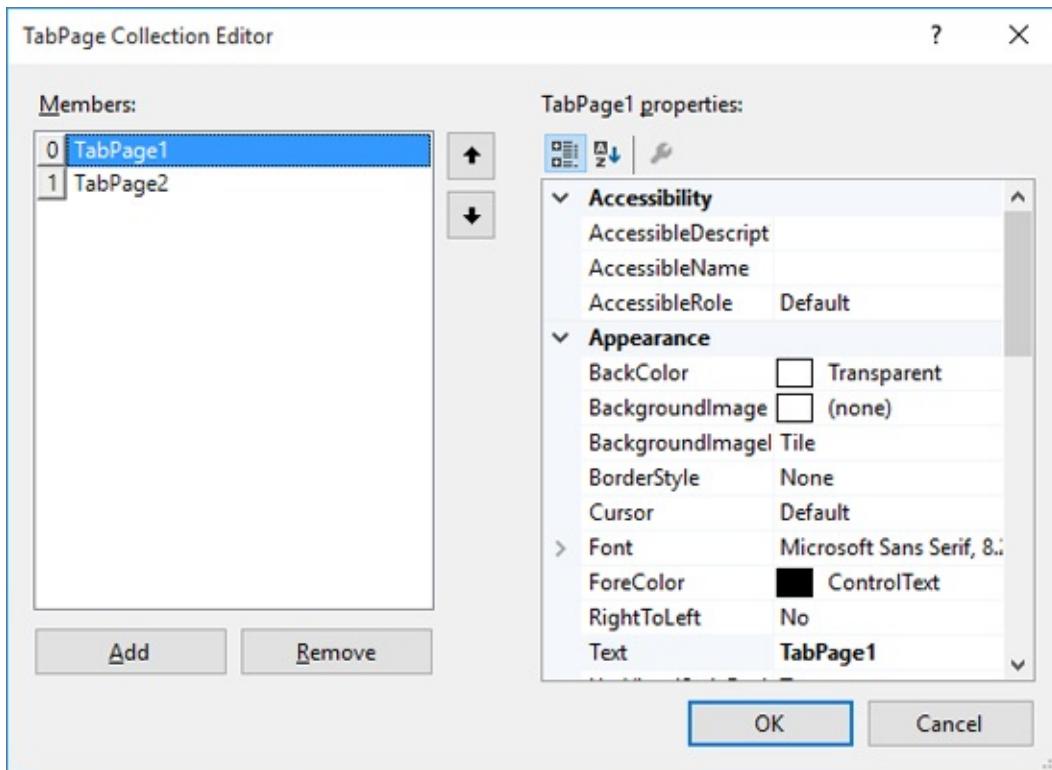
## TabControl Properties:

<b>Name</b>	Gets or sets the name of the tab control (three letter prefix for control name is <b>tab</b> ).
<b>BackColor</b>	Get or sets the tab control background color.
<b>BorderStyle</b>	Gets or sets the border style for the tab control.
<b>Font</b>	Gets or sets font name, style, size.
<b>ForeColor</b>	Gets or sets color of text or graphics.
<b>ItemSize</b>	Size structure determining tab size.
<b>SelectedIndex</b>	Gets or sets the currently displayed tab index.
<b>SizeMode</b>	Determines how tabs are sized.
<b>TabPages</b>	Collection describing each tab page.

## TabControl Events:

<b>SelectedIndexChanged</b>	Occurs when the <b>SelectedIndex</b> property changes.
-----------------------------	--

The most important property for the tab control is **TabPage**. It is used to design each tab (known as a **TabPage**). Choosing the **TabPage** property in the Properties window and clicking the ellipsis that appears will display the **TabPage Collection Editor**. With this editor, you can add, delete, insert and move tab pages. To add a tab page, click the **Add** button. A name and index will be assigned to a tab. After adding one page, the editor will look like this:



Add as many tab pages as you like. The tab page ‘array’ is zero-based; hence, if you have N tabs, the first is index 0, the last index N – 1.

You can change any tab page property you desire in the **Properties** area:

## TabPage Properties:

<b>Name</b>	Gets or sets the name of the tab page (three letter
-------------	---

prefix for control name is **tab**).

<b>BackColor</b>	Get or sets the tab page background color.
<b>BorderStyle</b>	Gets or sets the border style for the tab page.
<b>Font</b>	Gets or sets font name, style, size.
<b>ForeColor</b>	Gets or sets color of text or graphics.
<b>Text</b>	Titling information appearing on tab.

When done, click **OK** to leave the TabPage Collection Editor.

The next step is to add controls to each ‘page’ of the tab control. This is straightforward. Simply display the desired tab page by clicking on the tab. Then place controls on the tab page, treating the page like a group box or panel control. I usually ‘cover’ the tab page with a panel control before adding other controls. Make sure your controls become ‘attached’ to the tab page. You can still place controls on the form that are not associated with any tab. As the programmer, you need to know which tab is active (**SelectedIndex** property). And, you need to keep track of which controls are available with each tab page.

Typical use of **TabControl** control:

- Set the **Name** property and size appropriately.
- Establish each tab page using the **TabPage Collection Editor**.
- Add controls to tabs and form.
- Write code for the various events associated with controls on the tab control and form.

# ListBox Control

## In Toolbox:



## On Form (Default Properties):



A **ListBox** control displays a list of items (with as many items as you like) from which the user can select one or more items. If the number of items exceeds the number that can be displayed, a scroll bar is automatically added. Both single item and multiple item selections are supported. In this project, a list box control displays the user-entered dates and weights.

## ListBox Properties:

<b>Name</b>	Gets or sets the name of the list box (three letter prefix for list box name is <b>Ist</b> ).
<b>BackColor</b>	Get or sets the list box background color.
<b>Font</b>	Gets or sets font name, style, size.
<b>ForeColor</b>	Gets or sets color of text.
<b>Items</b>	Gets the Items object of the list box.
<b>SelectedIndex</b>	Gets or sets the zero-based index of the currently selected item in a list box.
<b>SelectedIndices</b>	Zero-based array of indices of all currently selected items in the list box.
<b>SelectedItem</b>	Gets or sets the currently selected item in the list box.
<b>SelectedItems</b>	SelectedItems object of the list box.
<b>SelectionMode</b>	Gets or sets the method in which items are

	selected in list box (allows single or multiple selections).
<b>Sorted</b>	Gets or sets a value indicating whether the items in list box are sorted alphabetically.
<b>Text</b>	Text of currently selected item in list box.
<b>TopIndex</b>	Gets or sets the index of the first visible item in list box.

### ListBox Methods:

<b>ClearSelected</b>	Unselects all items in the list box.
<b>FindString</b>	Finds the first item in the list box that starts with the specified string.
<b>GetSelected</b>	Returns a value indicating whether the specified item is selected.
<b>SetSelected</b>	Selects or clears the selection for the specified item in a list box.

### ListBox Events:

<b>SelectedIndexChanged</b>	Occurs when the SelectedIndex property has changed.
-----------------------------	---

Some further discussion is need to use the list box **Items** object, **SelectedItems** object and **SelectionMode** property. The **Items** object has its own properties to specify the items in the list box. It also has its own methods for adding and deleting items in the list box. The **Items** object is a zero-based array of the items in the list and **Count** (a property of **Items**) is the number of items in the list. Hence, the first item in a list box named **IstExample** is:

**IstExample.Items[0]**

The last item in the list is:

**IstExample.Items[IstExample.Items.Count – 1]**

The minus one is needed because of the zero-based array.

To add an item to a list box, use the **Add** method, to delete an item, use the **Remove** or **RemoveAt** method and to clear a list box use the **Clear** method. For our example list box, the respective commands are:

Add Item:      **lstExample.Items.Add(ItemToAdd)**  
Delete Item:    **lstExample.Items.Remove(ItemToRemove)**  
                  **lstExample.Items.RemoveAt(IndexofItemToRemove)**  
Clear list box: **lstExample.Items.Clear**

List boxes normally list string data types, though other types are possible. Note, when removing items, that indices for subsequent items in the list change following a removal.

In a similar fashion, the **SelectedItems** object has its own properties to specify the currently selected items in the list box. Of particular use is **Count** which tells you how many items are selected. This value, in conjunction with the **SelectedIndices** array, identifies the set of selected items.

The **SelectionMode** property specifies whether you want single item selection or multiple selections. When the property is **SelectionMode.One**, you can select only one item (works like a group of option buttons). When the **SelectionMode** property is set to **SelectionMode.MultiExtended**, pressing <Shift> and clicking the mouse or pressing <Shift> and one of the arrow keys extends the selection from the previously selected item to the current item. Pressing <Ctrl> and clicking the mouse selects or deselects an item in the list. When the property is set to **SelectionMode.MultiSimple**, a mouse click or pressing the spacebar selects or deselects an item in the list.

Typical use of **ListBox** control:

- Set **Name** property, **SelectionMode** property and populate **Items** object (usually in **Form\_Load** method).
- Monitor **SelectedIndexChanged** event for individual selections.
- Use **SelectedIndex** and **SelectIndices** properties to determine selected items.

# SaveFileDialog Control

In Toolbox:



Below Form (Default Properties):



We will need a tool to let us open previously saved weight files. The **SaveFileDialog** control will provide the capability to obtain filenames prior to writing a file. It also provides “safety factors.” The control insures that any path selected for saving a file exists and that if an existing file is selected, the user has agreed to overwriting that file. Note this control is the complement to the open file dialog control studied back in Chapter 7.

SaveFileDialog **Properties**:

<b>Name</b>	Gets or sets the name of the save file dialog (I usually name this control <b>dlgSave</b> ).
<b>AddExtension</b>	Indicates whether the dialog box automatically adds an extension to a file name if the user omits the extension.
<b>CheckFileExists</b>	Indicates whether the whether the dialog box displays a warning if the user specifies a file name that does not exist. Useful if you want the user to save to an existing file.
<b>CheckPathExists</b>	Indicates whether the dialog box displays a warning if the user specifies a path that does not exist.
<b>CreatePrompt</b>	Indicates whether the dialog box prompts the user for permission to create a file if the user specifies a file that does not exist.
<b>DefaultExt</b>	Gets or sets the default file extension.
<b>FileName</b>	Gets or sets a string containing the file name

	selected in the file dialog box.
<b>Filter</b>	Gets or sets the current file name filter string, which determines the choices that appear in "Files of type" box.
<b>FilterIndex</b>	Gets or sets the index of the filter currently selected in the file dialog box.
<b>InitialDirectory</b>	Gets or sets the initial directory displayed by the file dialog box.
<b>OverwritePrompt</b>	Indicates whether the dialog box displays a warning if the user specifies a file name that already exists. Default value is True.
<b>Title</b>	Gets or sets the file dialog box title.

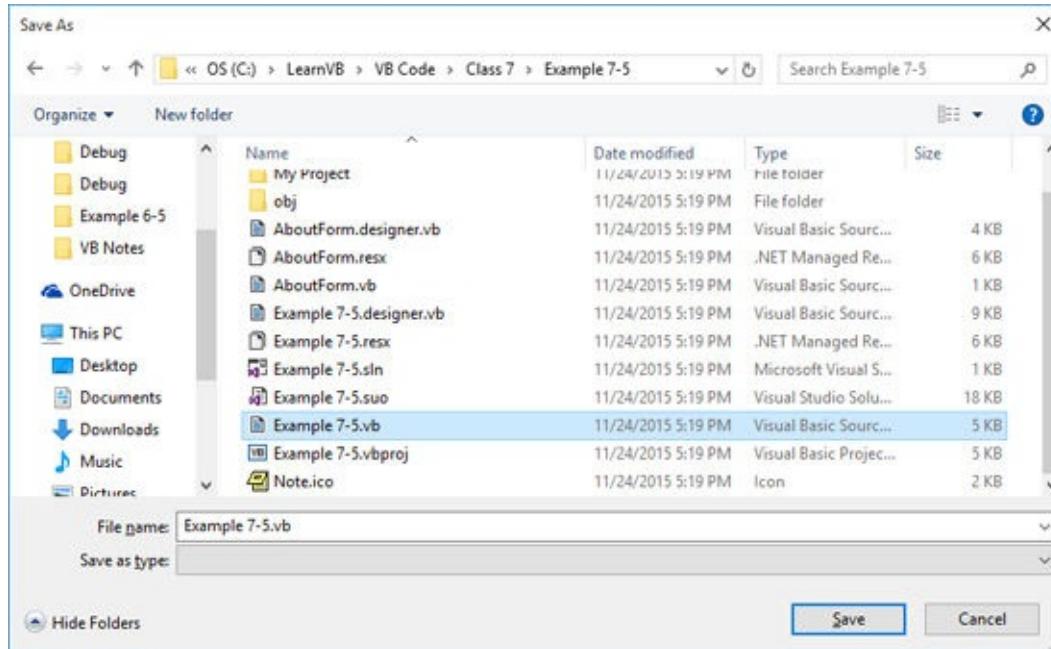
#### SaveFileDialog Methods:

<b>ShowDialog</b>	Displays the dialog box. Returned value indicates which button was clicked by user ( <b>OK</b> or <b>Cancel</b> ).
-------------------	--

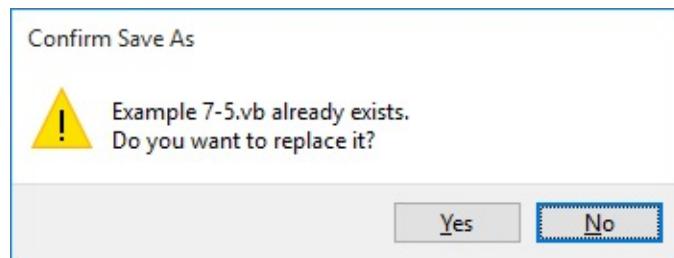
The **SaveFileDialog** control will appear in the tray area of the design window. The **ShowDialog** method is used to display the **SaveFileDialog** control. For a control named **dlgSave**, the appropriate code is:

**dlgSave.ShowDialog()**

And the displayed dialog box is:



The user types a name in the **File name** box (or selects a file using the dialog control). The file type is selected from the **Files of type** box (values here set with the **Filter** property). Once selected, the **Save** button is clicked. **Cancel** can be clicked to cancel the save operation. If the user selects an existing file and clicks **Save**, the following dialog will appear:



This is the aforementioned protection against inadvertently overwriting an existing file.

The **ShowDialog** method returns the clicked button. It returns **DialogResult.OK** if **Save** is clicked and returns **DialogResult.Cancel** if **Cancel** is clicked. The **FileName** property contains the complete path to the selected file.

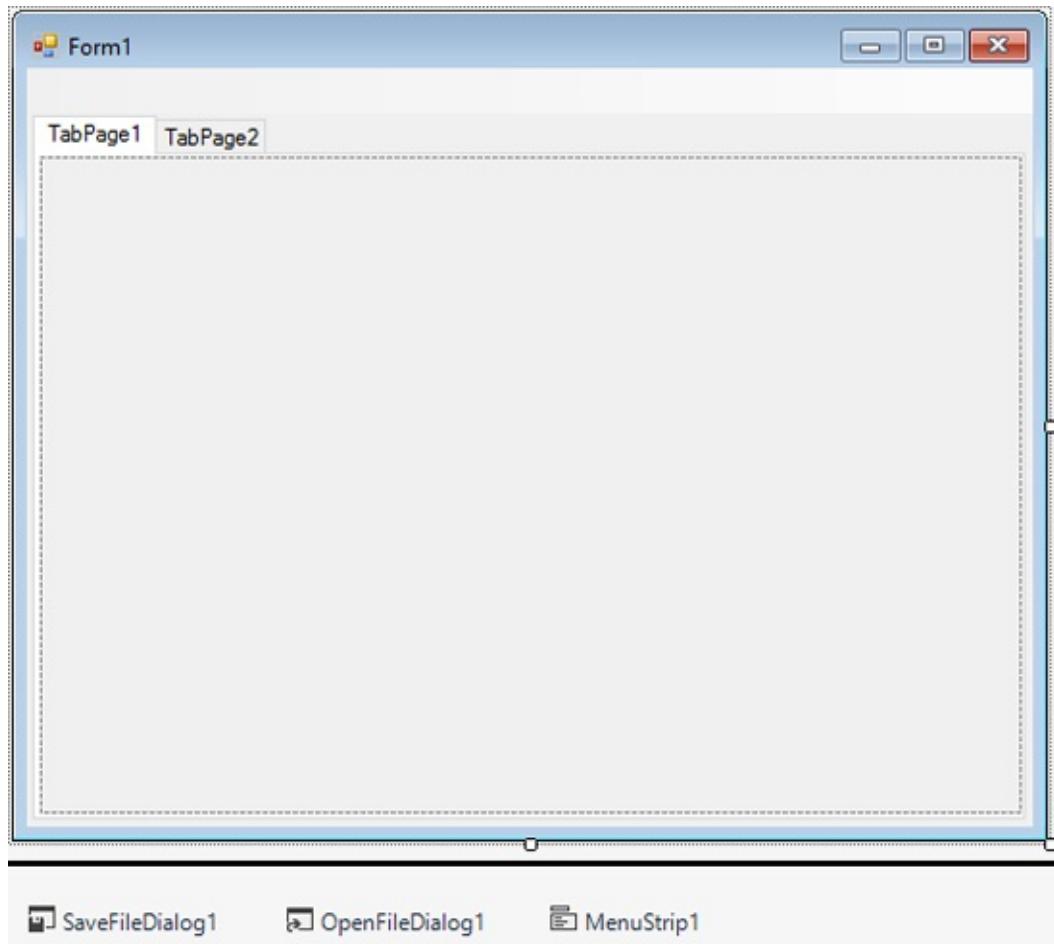
Typical use of **SaveFileDialog** control:

- Set the **Name**, **DefaultExt**, **Filter**, and **Title** properties.

- Use **ShowDialog** method to display dialog box.
- Read **FileName** property to determine selected file

# Weight Monitor Form Design

We can begin building the weight monitor project. Let's build the form – creating each tab page separately. Start a new project in Visual C#. Place a tab control on the form. Add a menu strip, an open file dialog and a save file dialog control to the project. Resize the form and tab control until the form looks similar to this:



The tab control will host the two tab pages. The dialog controls are used to open and save weight files. The menu will allow starting a new file, opening and saving files and exiting the program. Make sure the menu appears on the form and not on the tab control.

Establish the following menu structure and set the properties:

MenuStrip **mnuMain** structure:

```
File
  New Weight File
  Open Weight File
  Save Weight File
  -----
  Exit
```

MenuStrip **mnuMain** properties:

<b>Text</b>	<b>Name</b>
File	mnuFile
New Weight File	mnuFileNew
Open Weight File	mnuFileOpen
Save Weight File	mnuFileSave
-	(you choose, a <b>ToolStripSeparator</b> )
Exit	mnuFileExit

Set the other control properties using the properties window:

**Form1** Form:

<b>Property Name</b>	<b>Property Value</b>
Name	frmWeight
BorderStyle	FixedSingle
StartPosition	CenterScreen
Text	Weight Monitor

**tabControl1** Tab Control:

<b>Property Name</b>	<b>Property Value</b>
Name	frmWeight
BorderStyle	FixedSingle
Font Size	10

Font Style              Bold

**saveFileDialog1** Save File Dialog:

<b>Property Name</b>	<b>Property Value</b>
Name	dlgSave
DefaultExt	wgt
Filter	Weight Files (*.wgt) *.wgt
Title	Save Weight File

**openFileDialog1** Open File Dialog:

<b>Property Name</b>	<b>Property Value</b>
Name	dlgOpen
DefaultExt	wgt
Filter	Weight Files (*.wgt) *.wgt
Title	Open Weight File

Using the **Tab Pages** property of the tab control, establish two tab pages with these properties:

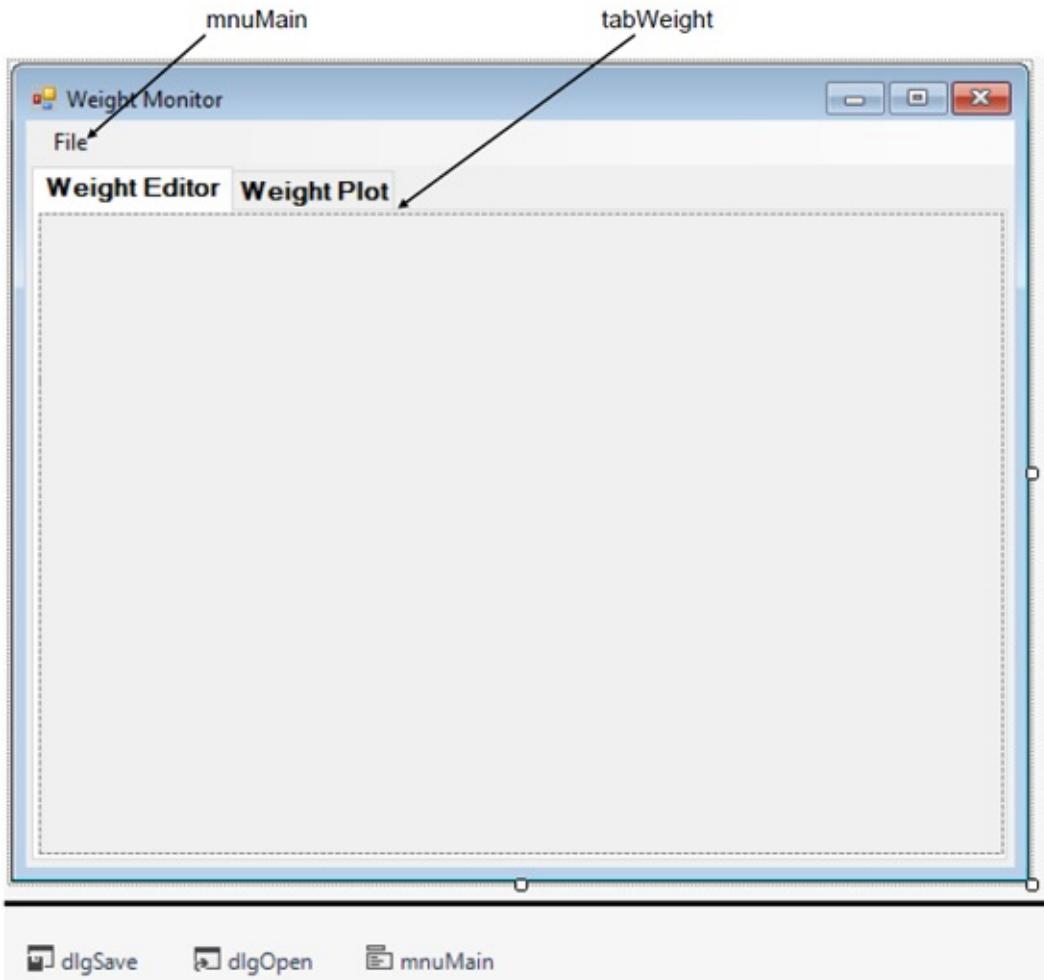
**tabPage1** Tab Page:

<b>Property Name</b>	<b>Property Value</b>
Name	tabEditor
Text	Weight Editor

**tabPage2** Tab Page:

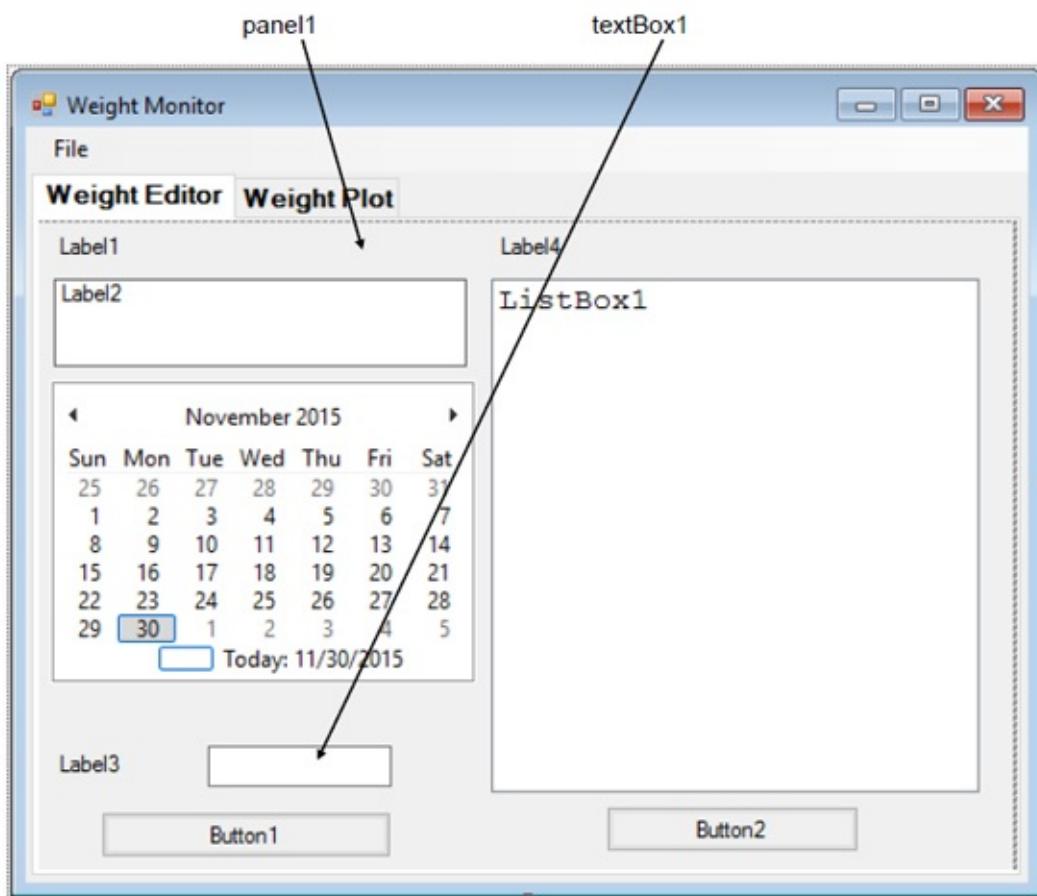
<b>Property Name</b>	<b>Property Value</b>
Name	tabPlot
Text	Weight Plot

When done setting properties, my form looks like this (the first tab page is shown):



# Form Design – Weight Editor Tab

Let's build this first tab page. Place a panel control on the tab page and make it the same size as the tab page. That is, have the panel cover the entire tab page. Add four label controls (one with **AutoSize** set to **False**), a month calendar control, a text box control, a list box control and two button controls to the panel. Resize and position controls to arrange the panel as (on **label2**, the resizable label, I've already set the **BorderStyle** property to **FixedSingle** so you can see its size):



**label1**, **label3** and **label4** are used for header information. **label2** holds the current weight file name. The calendar control is used for date selection and the text box control is used to enter the weight for that day. The list box holds the date and weight values (they will be sorted in ascending order of date). One button is used to enter a weight and one is used to delete a selected entry.

Set properties using the properties window:

**panel1** Panel:

<b>Property Name</b>	<b>Property Value</b>
Name	pnlEditor
BackColor	Light Blue

**label1** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Current Weight File
Font Size	10
Font Style	Bold

**label2** Label:

<b>Property Name</b>	<b>Property Value</b>
Name	lblFile
Text	New File
AutoSize	False
BackColor	White
BorderStyle	FixedSingle

**label3** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Weight (lb)
Font Size	10
Font Style	Bold

**label4** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Date Weight (lb) [there are 7 spaces between Date and Weight]

**Font** Courier New, Bold, Size 12

**monthCalendar1** MonthCalendar1:

<b>Property Name</b>	<b>Property Value</b>
Name	calWeights
Font Size	10

**textBox1** Text Box:

<b>Property Name</b>	<b>Property Value</b>
Name	txtWeight
Text	[Blank]
BorderStyle	FixedSingle
Font Size	10
Font Style	Bold

**listBox1** List Box:

<b>Property Name</b>	<b>Property Value</b>
Name	lstWeights
Sorted	True
Font	Courier New, Size 12

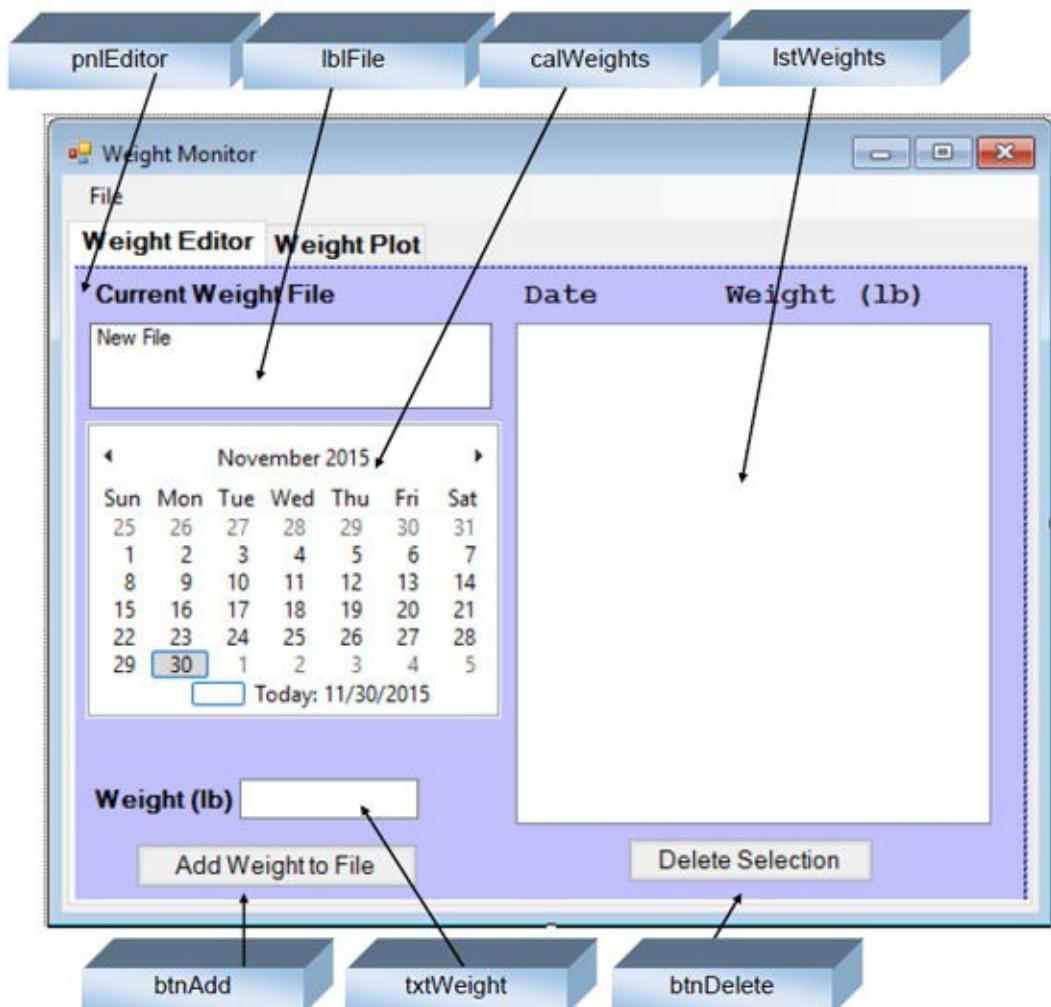
**button1** Button:

<b>Property Name</b>	<b>Property Value</b>
Name	btnAdd
Text	Add Weight to File
Font Size	10

**button2** Button:

<b>Property Name</b>	<b>Property Value</b>
Name	btnDelete
Text	Delete Selection
Font Size	10

When done setting properties, my tab page looks like this:



We will begin writing code for the tab page. As always, the code will be written in steps. We'll begin by writing code that starts a new weight file and initializes the application in its starting mode.

# Code Design – New Weight File

When the project first begins, we want a new file to be established for weight values. (Later, we will make modifications so the program automatically opens the last opened/saved file.) The steps to initialize the program for a new file are:

- Make **Weight Editor** tab page active.
- Set calendar date to current day.
- Clear list box control.
- Set **Text** property of **lblFile** to **New File**.
- Blank out **txtWeight**.
- Give focus to **txtWeight**.

With these steps, the program is ready to accept the first entry for the current date.

We put these initialization steps in a general method named **Initialize**. The code is:

```
private void Initialize()
{
    tabWeight.SelectedIndex = 0;
    calWeights.SelectionStart = DateTime.Today;
    lstWeights.Items.Clear();
    lblFile.Text = "New File";
    txtWeight.Text = "";
    txtWeight.Focus();
}
```

This general method should be called when the form first loads. The **frmWeight Load** event method is:

```
private void frmWeight_Load(object sender, EventArgs e)
```

```
{  
    this.Show();  
    Initialize();  
}
```

There is a need to **Show** the form prior to **Initialize**. The text box control (**txtWeight**) cannot be given focus unless the form is displayed.

This method should also be called when the user selects **New Weight File** in the **File** menu. Before calling it, though, the user should be asked if he/she really wants to start a new file. The code for the corresponding **mnuFileNew Click** event is:

```
private void mnuFileNew_Click(object sender, EventArgs e)  
{  
    if (MessageBox.Show("Are you sure you want to start a new weight  
file?", "New File", MessageBoxButtons.YesNo,  
MessageBoxIcon.Question) == DialogResult.Yes)  
    {  
        Initialize();  
    }  
}
```

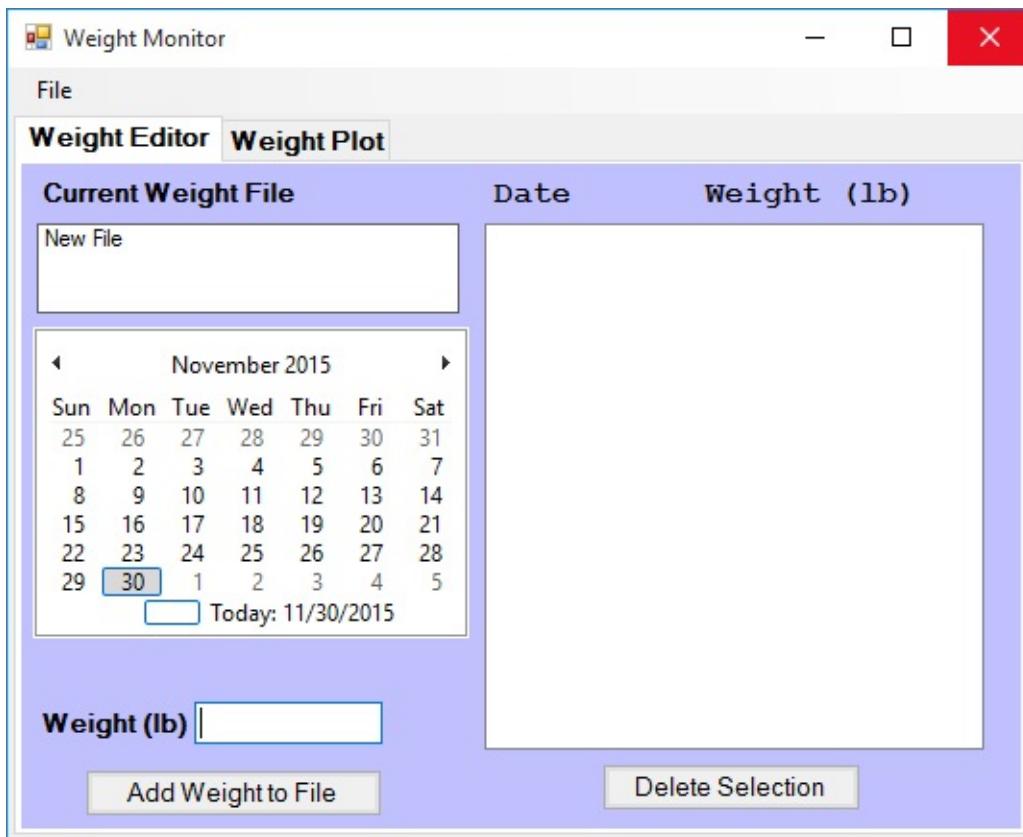
And, let's take care of the **mnuFileExit Click** method while we're at it. It is the usual one line method:

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

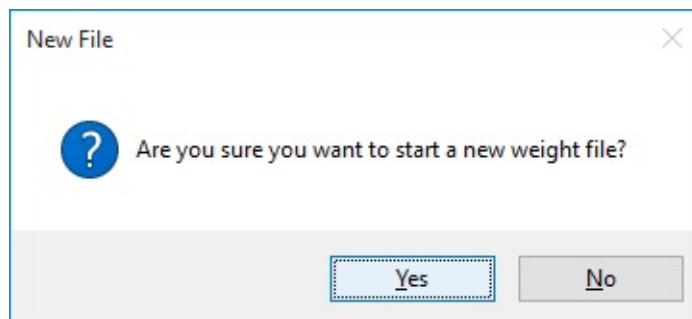
You should make sure you have saved any edits before clicking **Exit**. You could put a message box here asking the user if they really mean to exit. I've chosen not to.

Add these four methods (**Initialize**, **frmWeight\_Load**, **mnuFileNew\_Click**, **mnuFileExit\_Click**) to the project.

Save and run the application to make sure things initialize correctly. The form should appear as:



On your form, the current date will appear. Try the **New Weight File** option under the **File** menu. This message box should appear asking you if you're sure:



Make sure the **Exit** option works.

# Code Design – Entering Weights

When the program opens, the user selects a date from the month calendar (if not the current date) and enters a corresponding weight in the text box control. The user then clicks **Add Weight to File** to have that entry placed in the list box control. Here, we write the code that accomplishes this task.

Once a user selects a date and enters a weight, we need to process the following steps to add the entry to the list box control:

- Make sure there is a weight entry.
- See if entry already exists in list box for selected date; if so, delete the entry to avoid a repeat.
- Add new date and new weight to list box.
- Highlight (select) new entry in list box.

The code to implement these steps will be in the **btnAdd Click** method. Before writing code to process these steps, let's look at how the information will be formatted in the list box control.

We want the date and weight information to be neatly represented in a single line of information in the list box control. The format we choose is to have each line be 19 characters long (I picked this number because it fit nicely in the space provided). The first 10 columns of the line will be the date in a **yyyy/MM/dd** format (where **yyyy** is the year number, **MM** is the month number, and **dd** the day number). Using this format insures the lines are properly sorted in ascending date order (recall we set the list box's **Sorted** property to **True**). The weight (with a single decimal place) will be right justified in the remaining 9 columns. A fixed width font (**Courier New**) is used. As an example, if the weight is 202 on April 24, 2005, the line in the list box control will be:

**2005/04/24 202.0**

There are four spaces between the date and the weight, making the line the required 19 characters long.

We use a general method to form the above specified data line. The function is named **FormLine**. It requires a date (**DateTime** type) and weight (**string** type) as input arguments. It returns the data line as a **string** type:

```
private string FormLine(DateTime d, string w)
{
    string s;
    s=String.Format("{0:yyyy/MM/dd}", d);
    s += String.Format("{0,9:f1}", Convert.ToDouble(w));
    return (s);
}
```

You should recognize the steps in forming the line. Note the special formatting of the weight (w) value to have it be nine characters in width with a single decimal place (9:f1)..

With the **FormLine** function, we can add a date and weight to the list box control. We also need the capability of ‘parsing’ out date and weight values from a list box line. The two methods that do this are **GetDate** and **GetWeight**. Each method has the list box data line (19 characters long) as the input argument. **GetDate** returns the date (**DateTime** type) and **GetWeight** returns the weight (**string** type). Those methods are:

```
private DateTime GetDate(string s)
{
    return(Convert.ToDateTime(s.Substring(0, 10)));
}

private string GetWeight(string s)
{
    s = s.Substring(10, 9);
    return (s.Trim());
}
```

One more method is needed. The last step in adding a line to the list box is to

highlight the newly added line. This involves searching through the list box and finding the line with the matching date (using the **GetDate** method). The method **FindDate** takes a date (**DateTime** type) as input and returns the index (**int** type) of the line in the list box with that date. A negative one (-1) is returned if no matching line is found. The code that does the search is:

```
private int FindDate(DateTime d)
{
    if (lstWeights.Items.Count != 0)
    {
        for (int i = 0; i < lstWeights.Items.Count; i++)
        {
            if (GetDate(lstWeights.Items[i].ToString()) == d)
                return (i);
        }
    }
    return (-1);
}
```

With these new methods, we can now write the **btnAdd Click** method that implements the previously outlined steps:

```
private void btnAdd_Click(object sender, EventArgs e)
{
    int i;
    if (txtWeight.Text.Trim() == "")
    {
        txtWeight.Focus();
        return;
    }
    // add to list (check to see if date already there)
    i = FindDate(calWeights.SelectionStart);
    if (i != -1)
        lstWeights.Items.RemoveAt(i);
```

```

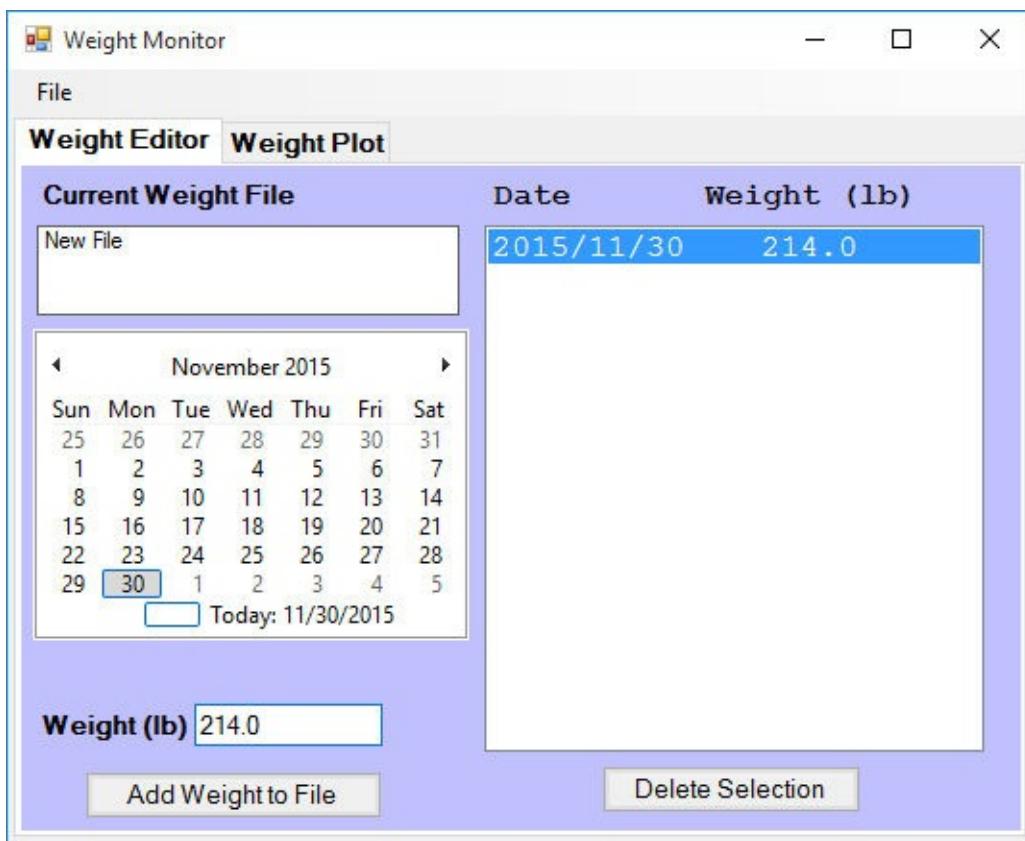
lstWeights.Items.Add(FormLine(calWeights.SelectionStart,
txtWeight.Text));

// find location and highlight
i = FindDate(calWeights.SelectionStart);
lstWeights.SelectedIndex = i;
}

```

Note how the **FindDate** method is used twice, once to see if there is a matching date and then to highlight the added line. Add **btnAdd\_Click**, **FormLine**, **GetDate**, **GetWeight**, **FindDate** to your project.

Save and run the project. Type a weight in the text box control and click the **Add Weight to File**. When I type my weight in for today, I see:



Note the proper formatting in the list box control. Pick some other dates on the calendar and enter other weights if you'd like. Stop the project when you're done.

You may or may not have noted that when entering a weight in the text box, there are no restrictions on what you can type. As usual, we need to write a **KeyPress** event method that limits the keystrokes. The event should allow numbers, backspace and a single decimal point. And a useful feature would be when the user presses <Enter>, the entry is added to the list box (**btnAdd** is ‘clicked’). The **txtWeight KeyPress** method that accomplishes these tasks is:

```
private void txtWeight_KeyPress(object sender, KeyPressEventArgs e)
{
    // only allow numbers, backspace, decimal point, enter
    if ((e.KeyChar >= '0' && e.KeyChar <= '9') || (int)e.KeyChar == 8)
        e.Handled = false;
    else if (e.KeyChar == '.')
    {
        if (txtWeight.Text.IndexOf(".") == -1)
            e.Handled = false;
        else
            e.Handled = true;
    }
    else if ((int)e.KeyChar == 13)
    {
        // enter key â€“ press Add butt
        btnAdd.PerformClick();
        e.Handled = false;
    }
    else
        e.Handled = true;
}
```

Add this method to the project. Rerun the project to make sure the key trapping works as desired. Make sure weight entries are added to the list box when the <Enter> key is pressed.

# Code Design – Editing Weights

We have the capability to enter weights for selected dates. We should also have the capability to edit existing entries in list box control, in case of incorrect entries.

The most drastic editing feature is to delete an entry in the list box control. To do this, the user selects the entry to delete, then clicks the button marked **Delete Selection**. The code for this process goes in the **btnDelete Click** event:

```
private void btnDelete_Click(object sender, EventArgs e)
{
    // remove selected item
    lstWeights.Items.Remove(lstWeights.SelectedItem);
}
```

Note, we have not given the user an option to change his/her mind about deleting. You might like to do this using a message box.

Now, let's look at a less drastic editing step – changing a previously entered weight value. When a user clicks an entry in the list box control, he/she should be given the ability to edit the entry. Conversely, when the user clicks a date on the calendar, if there is an corresponding entry in the list box, editing should be available. Note both editing tasks require some coordination between the calendar date and the selected list box entry.

When a user selects a list box line to edit, the following should occur:

- Parse the date from the selected line and establish that date on the calendar control.
- Parse the weight from the selected line and place it in **txtWeight**.
- Give focus to **txtWeight**.

After this, the user can change the weight and click **Add Weight to File** (or press <Enter>) to register the change.

The code for the above steps will go in the **lstWeights SelectedIndexChanged** event. The corresponding method is:

```
private void lstWeights_SelectedIndexChanged(object sender, EventArgs e)
{
    // display corresponding date
    if (lstWeights.SelectedIndex >= 0)
    {
        calWeights.SelectionStart =
            GetDate(lstWeights.SelectedItem.ToString());
        txtWeight.Text = GetWeight(lstWeights.SelectedItem.ToString());
        txtWeight.Focus();
    }
}
```

Add this method to the project.

When a user selects a date from the calendar control, the following should occur:

- Check to see if there is a corresponding date entry in list box control.
- If corresponding entry is found:
  - o Select (highlight) entry in list box.
  - o Parse the weight from the selected line and place it in **txtWeight**.
  - o Give focus to **txtWeight**.
- If corresponding entry is not found:
  - o Unselect all items in list box.
  - o Blank out **txtWeight**.
  - o Give focus to **txtWeight**.

In either case, the user can enter a weight for the selected date and click **Add Weight to File** (or press <Enter>) to register the change.

The code for responding to a date selection is placed in the **calWeights DateChanged** event method. That code is:

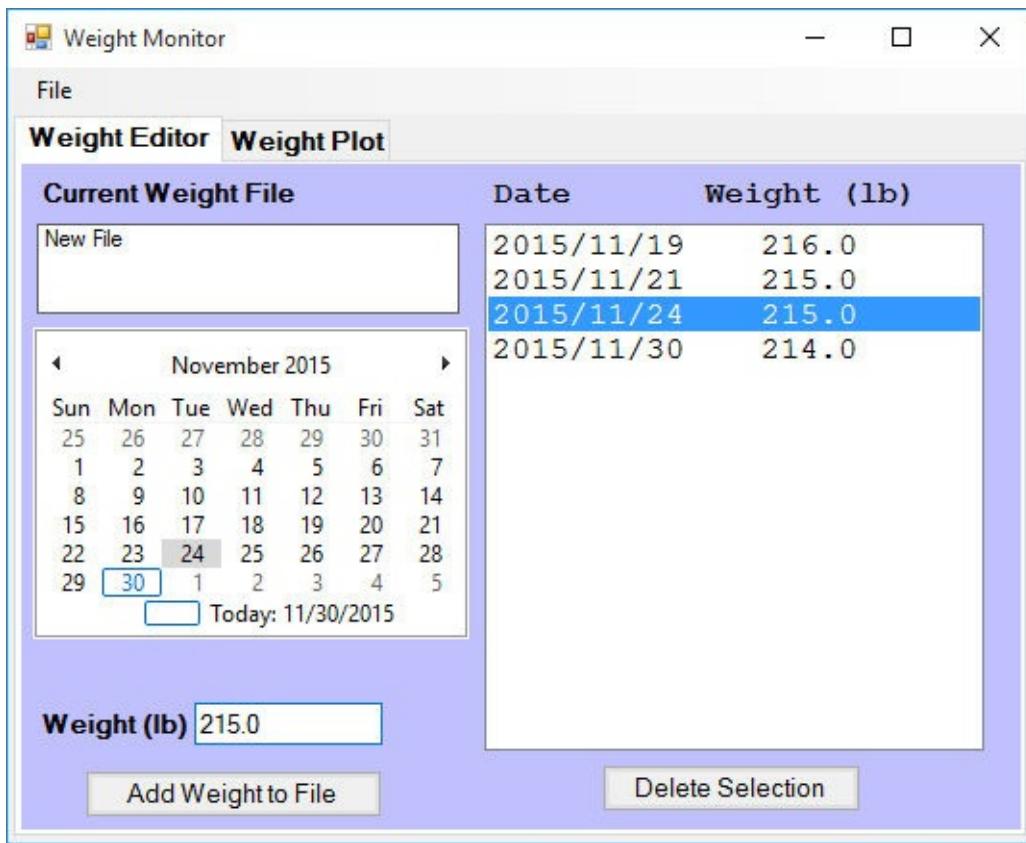
```

private void calWeights_DateChanged(object sender,
DateRangeEventArgs e)
{
    // show corresponding list box element (if there is one)
    int i;
    i = FindDate(calWeights.SelectionStart);
    if (i != -1)
    {
        lstWeights.SelectedIndex = i;
        txtWeight.Text = GetWeight(lstWeights.Items[i].ToString());
    }
    else
    {
        lstWeights.ClearSelected();
        txtWeight.Text = "";
    }
    txtWeight.Focus();
}

```

Place this method in the project. Notice use of the **FindDate** method to highlight the corresponding line (if it is there).

Save and run the project. Click a few dates on the calendar control and add some weights. Then, try the editing features. Click a date on the calendar – see if there is a matching line in the list box control. Click a line in the list box control – the date should be highlighted in the calendar control and the weight available for edit in the text box. Here's such a case in a run I made:



We now have full editing capability for the weight monitor project. We still need the capability to save any entries we might make. And we need the ability to read any saved values. We will use sequential files to save the date and weight values. In Chapter 7 (the multiple choice exam project), we looked at opening sequential files and reading lines of information from such files. In the last chapter (the biorhythm tracker project), we opened a file and read a single variable from the file. We have not looked at saving data to files. Prior to writing the code for saving and opening weight files, we will do a general review of sequential file access. We start with writing variables to a file, since we need to write a file before we can read it.

# Sequential File Output (Variables)

We will first look at **writing** values of **variables** to sequential files. The initial step in accessing any sequential file (either for input or output) is to open the file, knowing the name of the file. Visual C# uses the **StreamWriter** class to open a file for output. This class uses the Visual C# **IO** (input/output) namespace, so for any application using file input and output, you will need to add this line to the code:

```
using System.IO;
```

at the top of your code window. Place it with the other **Using directives**.

The constructor for opening a sequential file (**myFile**) for output is:

```
StreamWriter outputFile = new StreamWriter(myFile);
```

where **myFile** is the name (a **string**) of the file to open and **outputFile** is the returned **StreamWriter** object used to write variables to disk. The filename must be a complete path to the file. As you type this line, the Intellisense feature of the IDE will help you fill in the arguments.

A word of warning - when you open a file using the **StreamWriter** class, if the file already exists, it will be erased immediately! So, make sure you really want to overwrite the file. We will use the **SaveFileDialog** control to prevent accidental overwriting.

When done writing to a sequential file, it must be closed using the **Close** method. For our example, the syntax is:

```
outputFile.Close();
```

Once a file is closed, it is saved on the disk under the path (if used) and filename used to open the file.

Information (variables or text) is written to a sequential file in an appended

fashion. Separate Visual Express statements are required for each appending. There are different ways to write variables to a sequential file. For our work, we will look at just one – the **WriteLine** statement.

For a file opened as **outputFile**, the syntax is to write a variable named **myVariable** is:

```
outputFile.WriteLine(myVariable);
```

This statement will write the specified variable on a single line. In the weight monitor project, for each entry in the list box, we will write the date on one line and the corresponding weight on the subsequent line.

**Example** using **WriteLine** method:

```
int a;  
string b;  
double c, e;  
bool d;  
StreamWriter outputFile = new StreamWriter("c:\\junk\\testout.txt");  
outputFile.WriteLine(a);  
outputFile.WriteLine(b);  
outputFile.WriteLine(c);  
outputFile.WriteLine(d);  
outputFile.WriteLine(e);  
outputFile.Close();
```

After this code runs, the file **c:\junk\testout.txt** will have five lines, each of the variables (a, b, c, d, e) on a separate line. This, of course, assumes proper values have been assigned to each of the variables.

# Sequential File Input (Variables)

To **read variables** from a sequential file, we essentially reverse the write procedure. First, open the file using a **StreamReader**:

```
StreamReader inputFile = new StreamReader(myFile);
```

where **inputFile** is the returned file object and **myFile** is a valid path to the file. If the file you are trying to open does not exist, an error will occur. We will use the **OpenFileDialog** control to insure the file exists before trying to open it.

When all values have been read from the sequential file, it is closed using:

```
inputFile.Close();
```

Variables are read from a sequential file in the same order they were written. Hence, to read in variables from a sequential file, you need to know:

- How many variables are in the file
- The order the variables were written to the file
- The type of each variable in the file

If you developed the structure of the sequential file (say for a configuration file), you obviously know all of this information. And, if it is a file you generated from another source (spreadsheet, database), the information should be known. If the file is from an unknown source, you may have to do a little detective work. Open the file in a text editor and look at the data. See if you can figure out what is in the file.

Many times, you may know the order and type of variables in a sequential file, but the number of variables may vary. For example, you may export monthly sales data from a spreadsheet. One month may have 30 variables, the next 31 variables, and February would have 28 or 29. In such a case, you can read from the file until you reach an end-of-file condition. To determine whether we have reached the end of the file, we call the **Peek** method of the **StreamReader** object. The **Peek** method reads the next character in the file without changing

the place that we are currently reading. If we have reached the end of the file, **Peek** returns **-1**.

Variables are read from a sequential file using the **ReadLine** method. The syntax for our example file is:

```
myVariableString = inputFile.ReadLine();
```

where **myVariableString** is the **string** representation of the variable being read. To retrieve the variable value from this string, we need to convert the string to the proper type. Conversions for **int**, **double**, **bool**, and **DateTime** variables are:

```
myintVariable = Convert.ToInt32(myVariableString);
mydoubleVariable = Convert.ToDouble(myVariableString);
myboolVariable = Convert.ToBoolean(myVariableString);
myDateTimeVariable = Convert.ToDateTime(myVariableString);
```

**Example** using **ReadLine** method:

```
int a;
string b;
double c, e;
bool d;
StreamReader inputFile = new StreamReader("c:\\junk\\testout.txt");
a = Convert.ToInt32(inputFile.ReadLine());
b = inputFile.ReadLine();
c = Convert.ToDouble(inputFile.ReadLine());
d = Convert.ToBoolean(inputFile.ReadLine());
e = Convert.ToDouble(inputFile.ReadLine());
inputFile.Close();
```

This code opens the file **testout.txt** (in the **c:\junk\** folder) and sequentially reads five variables (one on each line). Notice how the **ReadLine** method is used directly in the conversions. Also notice, the string variable **b** (obviously) requires no conversion.

# Code Design – Saving Weight Files

We can now write the code that saves date and weight information to a sequential file. Each date and corresponding weight value ('parsed' from the list box control) will be saved on separate lines in the file. The date will be saved in **DateTime** format and the weight will be saved as a **string** (we use a **string** rather than numeric format to allow direct entry into the text box for editing).

When the user selects the **Save Weight File** option under the **File** menu, the following steps should be taken:

- Make sure there is at least one entry in the list box.
- Display save file dialog to obtain file name.
- If user clicks **Save**, then:
  - o Open file for output.
  - o For each line in list box control, obtain date and weight.
  - o Write date and weight to file.
  - o Close file.
- If user clicks **Cancel**, do nothing.

The code to implement these steps goes in the **mnuFileSave\_Click** event method:

```
private void mnuFileSave_Click(object sender, EventArgs e)
{
    if (lstWeights.Items.Count == 0)
    {
        MessageBox.Show("You need to enter at least one weight value.",
        "File Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
        return;
    }
    if (dlgSave.ShowDialog() == DialogResult.OK)
    {
```

```

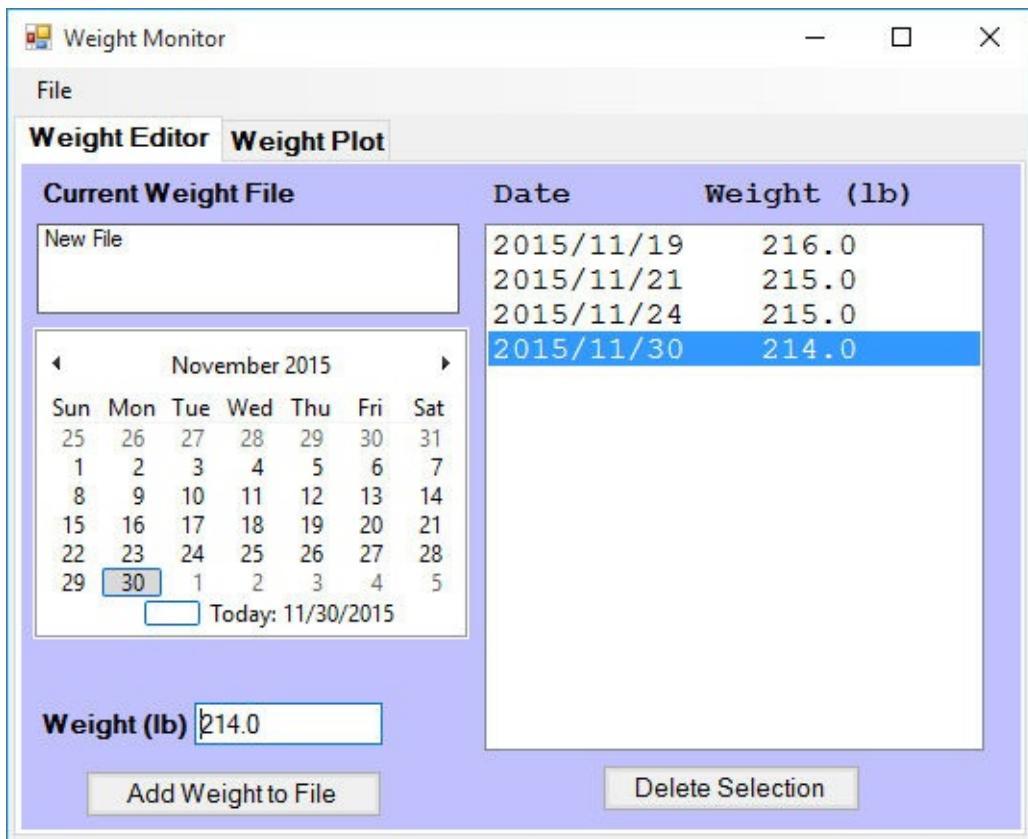
try
{
    StreamWriter outputFile = new
    StreamWriter(dlgSave.FileName);
    lblFile.Text = dlgSave.FileName;
    for (int i = 0; i < lstWeights.Items.Count; i++)
    {
        outputFile.WriteLine(GetDate(lstWeights.Items[i].ToString( )));

        outputFile.WriteLine(GetWeight(lstWeights.Items[i].ToString()));
    }
    outputFile.Close();
}
catch
{
    MessageBox.Show("An error occurred saving the weight file.",
    "File Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
}
}
}

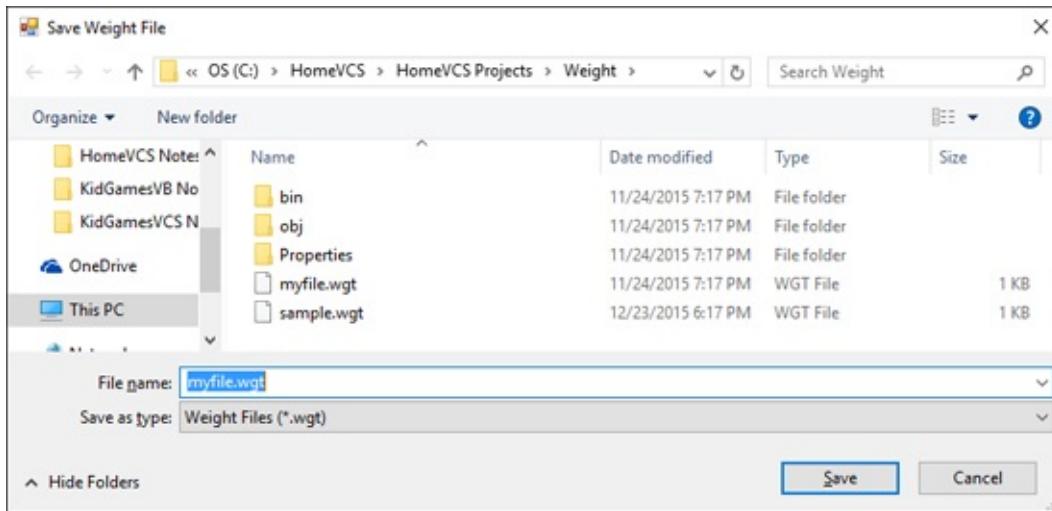
```

We have placed the save procedure in a **try/catch** block to catch any errors that might occur. Notice the use of the **GetDate** and **GetWeight** methods in the **WriteLine** statements. With this code, two lines are written to the file for each entry in the list box – one line for the date and one for the weight. Add this method to the project.

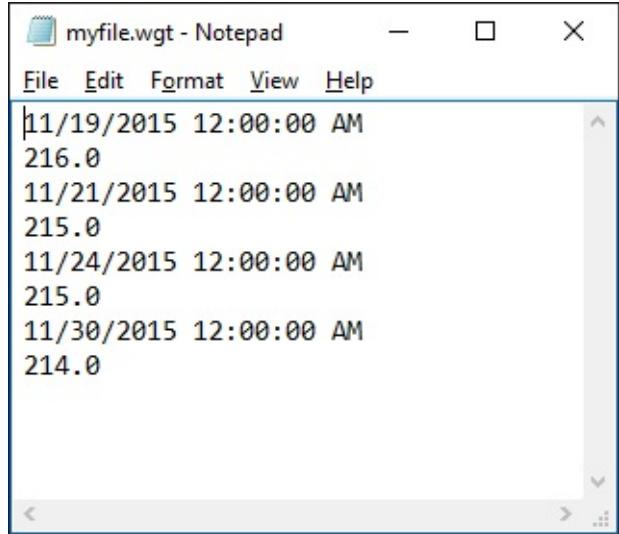
Save and run the project. Click some dates and enter some weights. Here are some values I entered:



Choose **Save Weight File** from the **File** menu. You will see:



Give a name to your file (I chose **myfile**) and note the folder it is saved in. Click **Save**. Start a text editor (like **Notepad**) and open the file just created. Since we use a **wgt** extension for a weight file, you will have to make sure you display all file types, not just **txt** files. The file I created looks like this:



Note formatting is just what we expected – one entry on each line (a date, followed by a weight).

# Code Design – Opening Weight Files

Now, with a capability to save weight files, we need code to open and read information from those files. When the user selects **Open Weight File** from the **File** menu, the following occurs:

- Make sure the user wants to open a new file; if not, do nothing.
- Display open file dialog to obtain file name.
- If user clicks **Open**, then:
  - o Open file for input.
  - o Set **Text** property of **lblFile**.
  - o Read each date and weight pair.
  - o Add entry to list box control using**FormLine** function.
  - o Close file.
- If user clicks **Cancel**, do nothing.

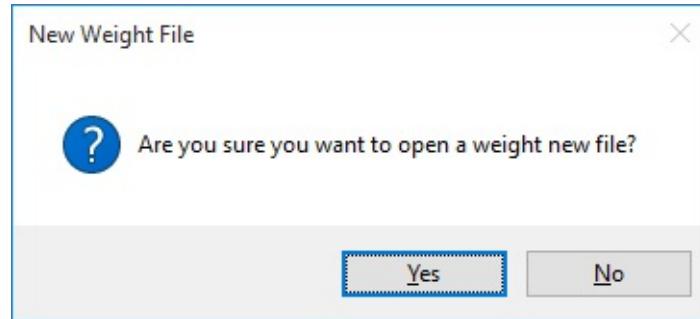
The code corresponding to these steps goes in the **mnuFileOpen Click** event method:

```
private void mnuFileOpen_Click(object sender, EventArgs e)
{
    DateTime d;
    string w;
    if (MessageBox.Show("Are you sure you want to open a weight new
file?", "New Weight File", MessageBoxButtons.YesNo,
MessageBoxIcon.Question) == DialogResult.Yes)
    {
        if (dlgOpen.ShowDialog() == DialogResult.OK)
        {
            try
            {
                Initialize();
                StreamReader inputFile = new
```

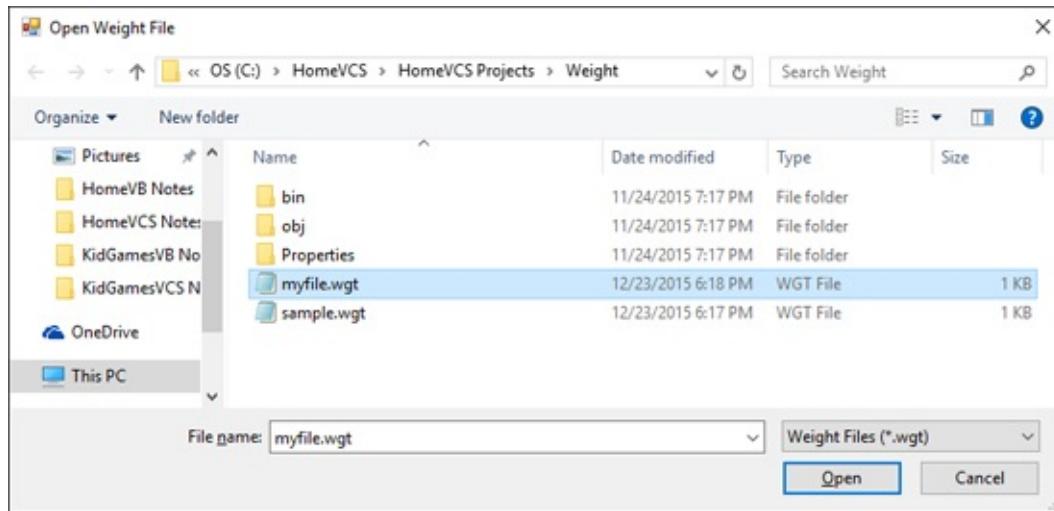
```
StreamReader(dlgOpen.FileName);
    lblFile.Text = dlgOpen.FileName;
    do
    {
        d =
Convert.ToDateTime(inputFile.ReadLine());
        w = inputFile.ReadLine();
        lstWeights.Items.Add(FormLine(d, w));
    }
    while (inputFile.Peek() != -1);
    inputFile.Close();
}
catch
{
    MessageBox.Show("An error occurred opening the weight
file.", "File Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
}
}
```

Like the save procedure, we have placed the open procedure in a **try/catch** block to catch any errors that might occur. Notice how the **FormLine** function is used to add lines to the list box control. And, note how the **Peek** method is used to read until the end-of-file is reached. Add this method to your project.

Save and run the project. Choose **Open Weight File** under the **File** menu. You will see the message box:

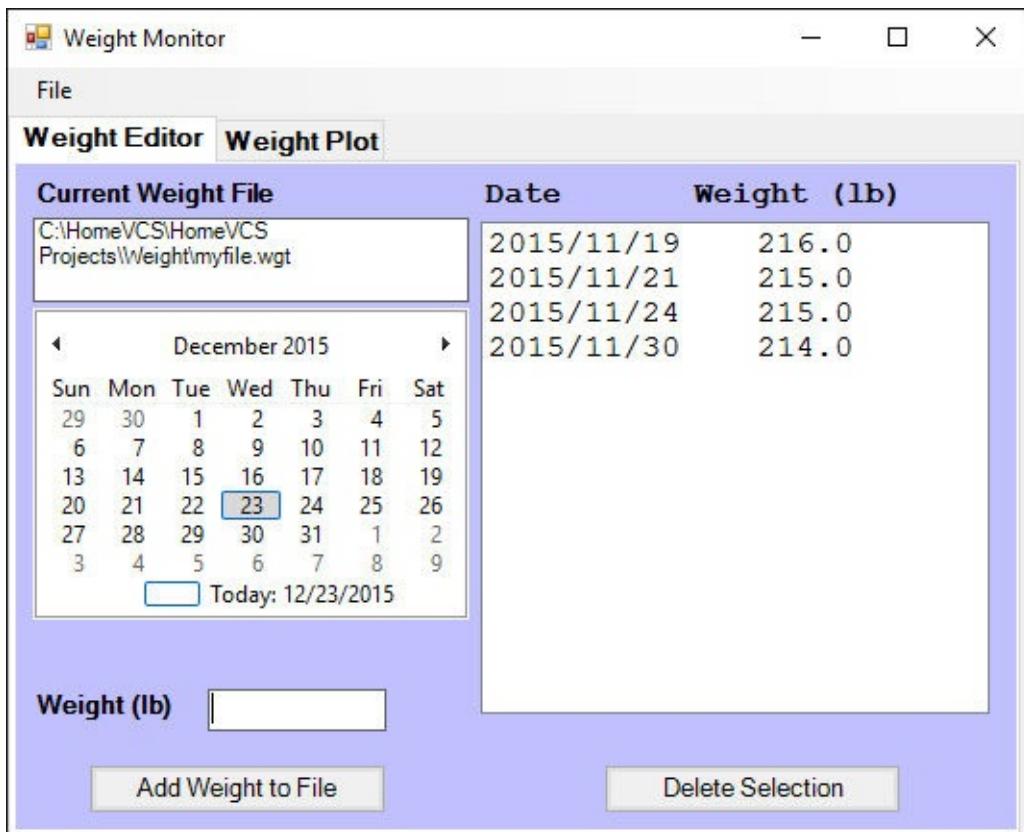


Click **Yes** and the open file dialog box will appear:



Navigate to the file you just created and click **Open**. The dates and weights you entered will appear.

Here is what I see when I open the file I created (**myfile.wgt**):



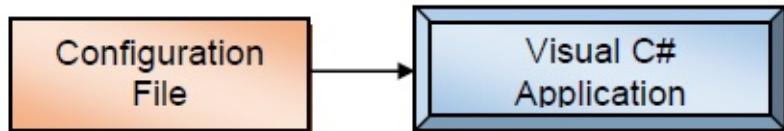
The file name appears under **Current Weight File** and the date and weight values are displayed. They can now be edited.

The **Weight Editor** tab is nearly complete. The most common use for the weight monitor program is to enter your daily weight. This involves starting the program, opening your weight file, making any new entries, saving the file, then exiting. You might also view a plot of your weight (using the **Weight Plot** tab we develop next). After a while, you get tired of opening the same file each time you run the program. It would be nice if your file opened automatically and the date and weight values were displayed. And, you get tired of remembering to save your file before exiting. It would be nice if the program would just automatically save the last file you were working with.

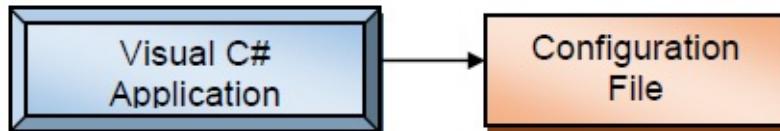
We can solve both of these problems. To do this, we need to discuss configuration files. Such files are used to save information needed to initialize a program. In our case, we want to use a configuration file to save the last file opened so that file is automatically opened/saved the next time we run the weight monitor program.

# Configuration Files

Another use for sequential files is to provide initialization information for a project. Such a file is called a **configuration** or **initialization file** and almost all applications use such files. Here is the idea for a Visual C# Project:



In this diagram, the configuration file (a sequential file) contains information that can be used to initialize different parameters (control properties, variable values) within the Visual C# project. The file is opened when the application begins, the file values are read and the various parameters established. Similarly, when we exit an project, we could have it write out current parameter values to an output configuration file:



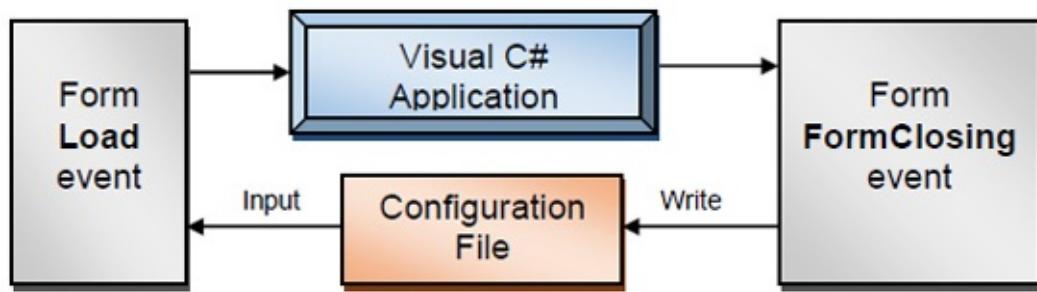
This output file could then become an input file the next time the project is executed.

How do you decide what your configuration file will contain and how it will be formatted? That is completely up to you, the project designer. Typical information stored in a **configuration** file includes: file names, current dates and times, check box settings, radio button settings, selected colors, font name, font style, font size, and selected menu options. You decide what is important in your project. You develop variables to save information and read and write these variables from and to the sequential configuration file. There is usually one variable (numeric, string, date, Boolean) for each option being saved. In our weight monitor project, the configuration file will have the name of the last opened/saved file.

Once you've decided on values to save and the format of your file, how do you

proceed? A first step is to create an initial file using a text editor. If the number of variables being saved is relatively short, I suggest putting one variable on each line of the file. Save your configuration file in the same folder your project's executable will be written to (the **Bin\Debug** folder of your project in Visual C#). Configuration files will always be kept in the project path. And, the usual three letter file extension for a configuration file is **ini** (for initialization).

Once you have developed the configuration file, you need to write code to fit this framework:



When your project begins (Form **Load** event), open (**StreamReader** object) and read (**.ReadLine** method) the configuration file and use the variables to establish the respective options. Establishing options involves things like setting Font objects, establishing colors, simulating click events on check boxes and radio buttons, and setting properties. The code to open the configuration file should be in a **try/catch** block. Many times, users may delete files (including configuration files) not knowing what they're doing. We want to make sure if this (or some other error) happens, the program will still run. If an error occurs when opening the file, we establish values for the missing configuration file variables and continue.

When your project ends (Form **FormClosing** event), examine all options to be saved, establish respective variables to represent these options, and open (**StreamWriter** method) and write (**WriteLine** method) the configuration file. We write the configuration file in the **FormClosing** event for two reasons. Usually a project will have an **Exit** button or an **Exit** option in the menu structure. The code to exit a project is usually:

```
this.Close();
```

This statement will activate the **FormClosing** event. Also, most Windows projects have a little box with an X in the upper right hand corner of the form that can be used to stop an project. When this ‘X box’ is clicked, any exit routine you have coded is **ignored** and the program immediately transfers to the **FormClosing** event.

# Code Design – Configuration File

We want to use a configuration file (named **weight.ini**) to save the name of the last weight file opened and/or saved in the weight monitor project. We need code in the form's **Load** method to open the configuration file and then open the specified weight file. And, we need code in the **FormClosing** method to save the configuration file and specified weight file. Let's address the **Load** method first.

Establish a form level variable (**string** type) to hold the last file opened and/or saved:

```
string lastFile;
```

Before writing the code to open the configuration file and, subsequently, open the corresponding weight file, we need to make a couple of code modifications. I'll explain why. Once we have the name of the file to automatically open, we need to access code currently in the **mnuFileOpen Click** event method. We don't want to invoke that method directly because the user would need to respond to a message box, then pick the file from an open dialog box. This completely defeats the purpose of the configuration file. We need to be able to bypass the message box and the open file dialog box and access the code that opens and reads the file. We will put that code in a separate method.

The existing **mnuFileOpen Click** method is:

```
private void mnuFileOpen_Click(object sender, EventArgs e)
{
    DateTime d;
    string w;
    if (MessageBox.Show("Are you sure you want to open a weight new
file?", "New Weight File", MessageBoxButtons.YesNo,
MessageBoxIcon.Question) == DialogResult.Yes)
    {
        if (dlgOpen.ShowDialog() == DialogResult.OK)
        {
```

```

try
{
    Initialize();
    StreamReader inputFile = new
    StreamReader(dlgOpen.FileName);
    lblFile.Text = dlgOpen.FileName;
    do
    {
        d =
        Convert.ToDateTime(inputFile.ReadLine());
        w = inputFile.ReadLine();
        lstWeights.Items.Add(FormLine(d, w));
    }
    while (inputFile.Peek() != -1);
    inputFile.Close();
}
catch
{
    MessageBox.Show("An error occurred opening the weight
file.", "File Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
}
}
}
}

```

We have shaded the lines of code involved with opening the weight file.

To directly access the code to open the file, we create a general method **OpenWeightFile**. This method uses a single **string** argument (**fn**) holding the name of the file to open. The shaded code above forms the basis for this method. A couple of changes need to be made. First, **fn** is used to represent the file name. Second, we need lined to establish a value for **lastFile**. These changes are shaded below:

```

private void OpenWeightFile(string fn)
{
    DateTime d;
    string w;
    try
    {
        Initialize();
        StreamReader inputFile = new StreamReader(fn);
        lblFile.Text = fn;
        do
        {
            d = Convert.ToDateTime(inputFile.ReadLine());
            w = inputFile.ReadLine();
            lstWeights.Items.Add(FormLine(d, w));
        }
        while (inputFile.Peek() != -1);
        inputFile.Close();
        lastFile = fn;
    }
    catch
    {
        MessageBox.Show("An error occurred opening the weight file.",
        "File Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
        lastFile = " ";
    }
}

```

And, the **mnuFileOpen Click** method is modified to be (new line is shaded):

```

private void mnuFileOpen_Click(object sender, EventArgs e)
{
    if (MessageBox.Show("Are you sure you want to open a weight new

```

```

file?", "New Weight File", MessageBoxButtons.YesNo,
MessageBoxIcon.Question) == DialogResult.Yes)
{
    if (dlgOpen.ShowDialog() == DialogResult.OK)
    {
        OpenWeightFile(dlgOpen.FileName);
    }
}

```

Add **OpenWeightFile** to the project along with the modified **mnuFileOpen Click** method.

The form **Load** method opens the configuration file and uses the new **OpenWeightFile** method to automatically open **LastFile**. The new **frmWeight Load** method is (changes are shaded):

```

private void frmWeight_Load(object sender, EventArgs e)
{
    // open .ini file
    try
    {
        StreamReader inputFile = new
        StreamReader(Application.StartupPath + "\\weight.ini");
        lastFile = inputFile.ReadLine();
        inputFile.Close();
    }
    catch
    {
        // initialization file not found
        lastFile = "";
    }
    this.Show();
}

```

```

if (lastFile != "")
    OpenWeightFile(lastFile);
else
    Initialize();
}

```

Notice use of the **Application.StartupPath** parameter to make sure the file is in the project **Bin\Debug** folder. If there are any errors in opening the configuration file, **lastFile** is set to an empty string and the form is initialized to a new condition. Make the noted modifications to the **Load** method. With these changes, when the program begins, the configuration file is opened and the last saved weight file opened and those values displayed.

We now make similar modifications to the project to save the configuration file and save the last weight file. First, we build a general method **SaveWeightFile** to let us bypass the save file dialog box when automatically saving the weight file. The existing **mnuFileSave Click** method is:

```

private void mnuFileSave_Click(object sender, EventArgs e)
{
    if (lstWeights.Items.Count == 0)
    {
        MessageBox.Show("You need to enter at least one weight value.",
        "File Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
        return;
    }
    if (dlgSave.ShowDialog() == DialogResult.OK)
    {
        try
        {
            StreamWriter outputFile = new
            StreamWriter(dlgSave.FileName);
            lblFile.Text = dlgSave.FileName;
            for (int i = 0; i < lstWeights.Items.Count; i++)

```

```

    {

outputFile.WriteLine(GetDate(lstWeights.Items[i].ToString( )));

outputFile.WriteLine(GetWeight(lstWeights.Items[i].ToString()));
}

outputFile.Close();
}

catch
{
    MessageBox.Show("An error occurred saving the weight
file.", "File Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
}
}

}

```

We have shaded the lines of code involved with saving the weight file. These lines form the basis for the **SaveWeightFile** method.

The **SaveWeightFile** general method is (we use the variable **fn** to represent the file name – see shaded lines):

```

private void SaveWeightFile(string fn)
{
    try
    {
        StreamWriter outputFile = new StreamWriter(fn);
        lblFile.Text = fn;

        for (int i = 0; i < lstWeights.Items.Count; i++)
        {

outputFile.WriteLine(GetDate(lstWeights.Items[i].ToString( )));

outputFile.WriteLine(GetWeight(lstWeights.Items[i].ToString()));

```

```

        }
        outputFile.Close();
        lastFile = fn;
    }
    catch
    {
        MessageBox.Show("An error occurred saving the weight file.",
        "File Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
        lastFile = "";
    }
}

```

And, the **mnuFileSave Click** method is modified to be (new line is shaded):

```

private void mnuFileSave_Click(object sender, EventArgs e)
{
    if (lstWeights.Items.Count == 0)
    {
        MessageBox.Show("You need to enter at least one weight value.",
        "File Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
        return;
    }
    if (dlgSave.ShowDialog() == DialogResult.OK)
    {
        SaveWeightFile(dlgSave.FileName);
    }
}

```

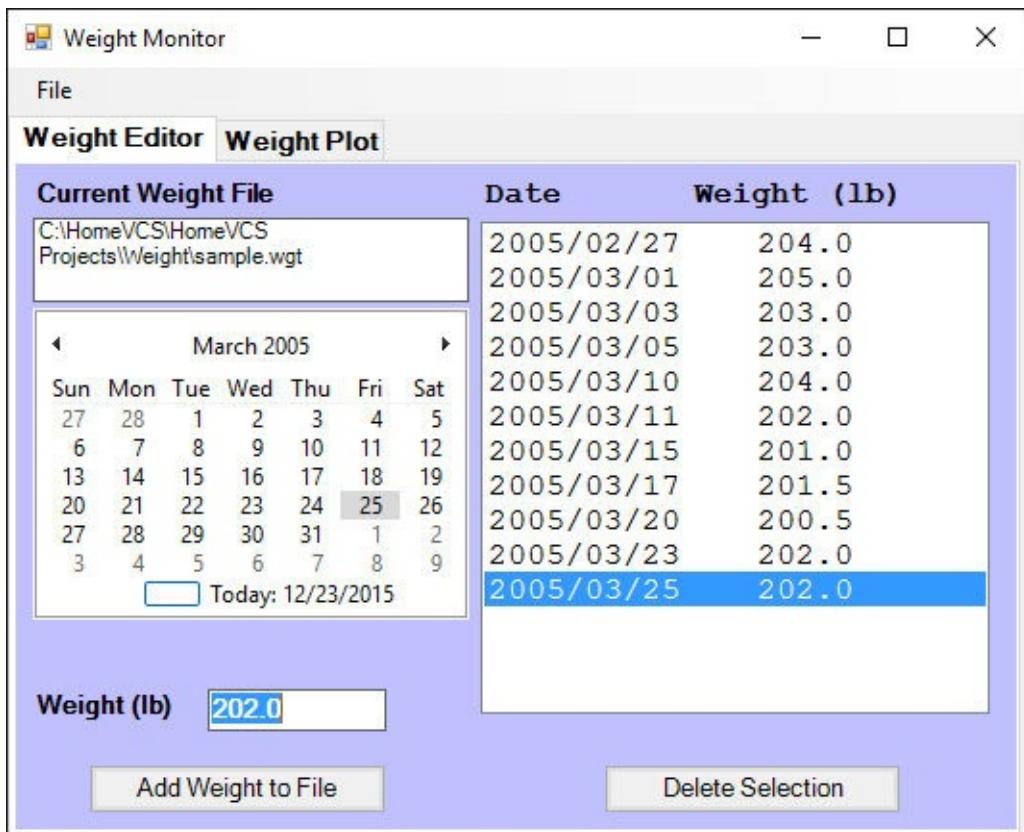
Add **SaveWeightFile** to the project along with the modified **mnuFileSave Click** method.

The form **FormClosing** method saves the configuration file and uses **SaveWeightFile** to automatically save **lastFile**. The **frmWeight FormClosing** method is:

```
private void frmWeight_FormClosing(object sender,
FormClosingEventArgs e)
{
    // Write out initialization file
    StreamWriter outputFile = new
    StreamWriter(Application.StartupPath + "\\weight.ini");
    outputFile.WriteLine(lastFile);
    outputFile.Close();
    // save last file
    if (lastFile != "")
        SaveWeightFile(lastFile);
}
```

With these changes, when the program ends, the configuration file is saved, as is the last opened weight file. Make the noted changes. The **Weight Editor** tab page is complete. Let's try it.

Save and run the project. The form will appear in its initial configuration since no **lastFile** value has been saved yet (there is no **weight.ini** file). Click **Open Weight File** under the **File** menu. In the **HomeVCS\HomeVCS Projects\Weight** folder is a sample weight file named **sample.wgt**. Open that file. You should see:

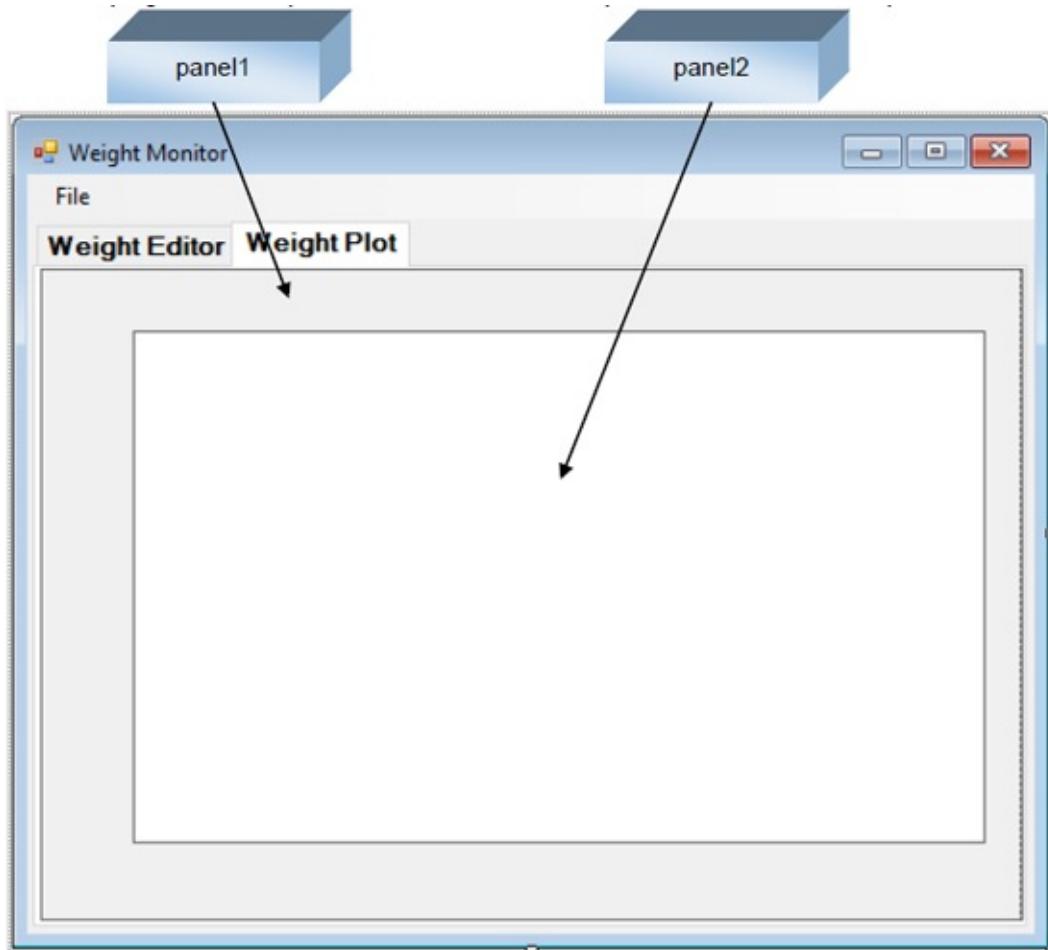


Edit some values if you want or add some values. Now, stop the project – the weight file will be automatically saved. And, at this point, the **weight.ini** file is written to your project's **Bin\Debug** folder. Look in that folder to make sure it's there. Now, run the project again. The data in the sample file (with any changes you made) should appear:

Now, whenever you run the weight monitor project, it will automatically begin using the last set of weight values you were editing. When you stop, any changes you made are automatically saved. This is usually what you want to do. You can always override this automatic feature if you want. If the values displayed upon opening are not correct, simply open the correct file using **Open** option in the **File** menu. Similarly, a file can be saved at any time using the **Save** option. Now, let's take a look at plotting these weight values.

# Form Design – Weight Plot

We now build the tab page that holds a plot of weight values. In design mode, click the **Weight Plot** tab to make it active. Have one panel control cover the entire tab page. Then, place a second smaller panel control on that panel:



**panel1** will be used for labeling information. **panel2** will hold the weight plot.

Set properties using the properties window:

**panel1** Panel:

Property Name	Property Value
Name	pnlWeight

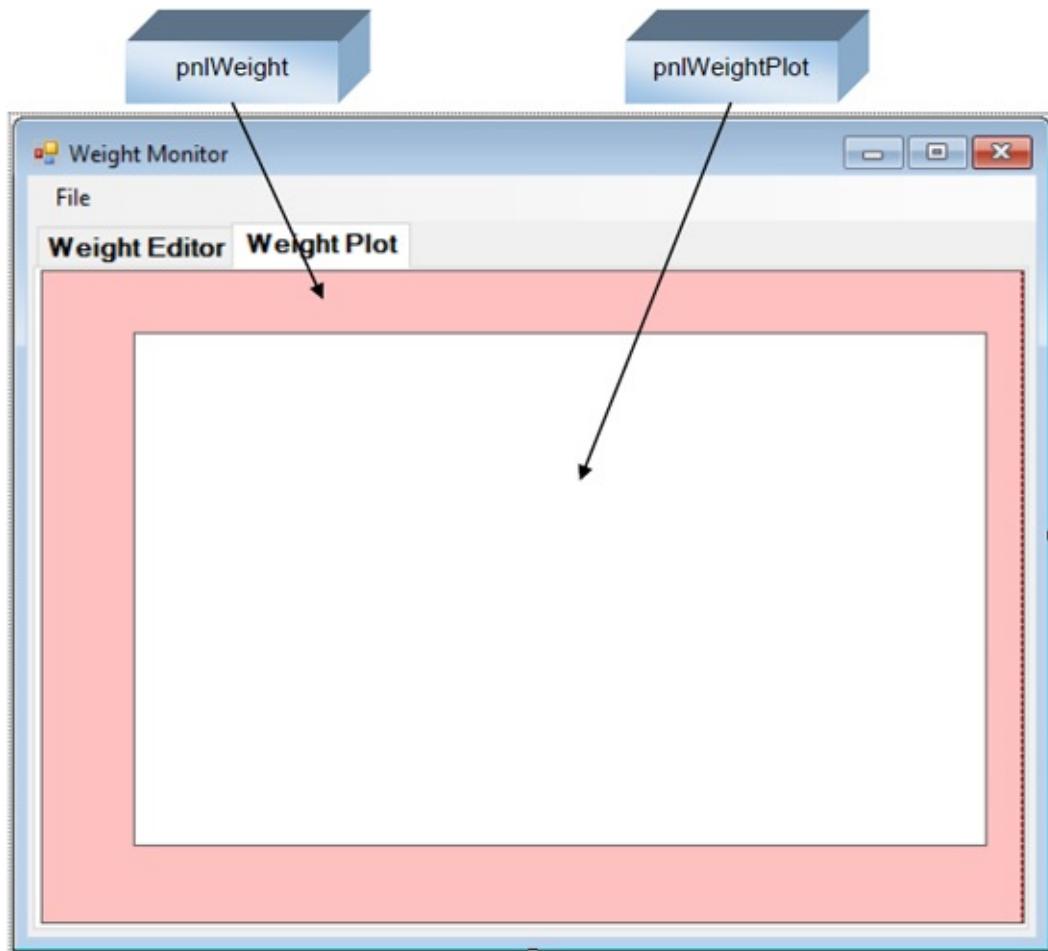
BackColor

Light Red

**panel2 Panel:**

<b>Property Name</b>	<b>Property Value</b>
Name	pnlWeightPlot
BackColor	White
BorderStyle	FixedSingle

When done setting properties, my tab page looks like this:



As usual, we will write the code for this tab page in several steps. There are two primary tasks in drawing the weight plot. The first is to “connect the points” entered in the list box control to draw the plot in a panel control. The second is to put useful labeling information around the plot. Let’s look at the plot drawing task first.

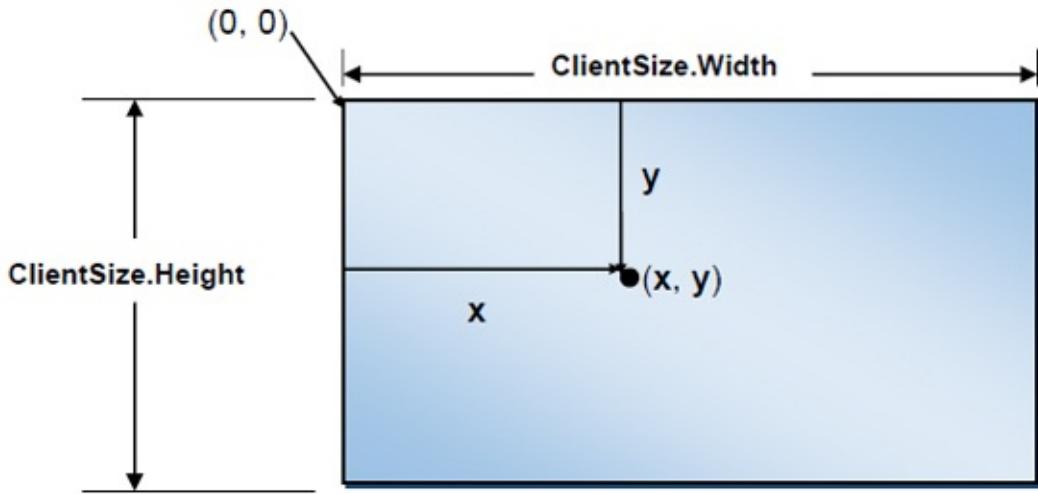
# Code Design – Weight Plot

When a user clicks the **Weight Plot** tab, we want to display a plot of the input weights. To draw a plot, we use the graphics methods discussed in Chapter 9 (plus some new ones). Review that material if necessary. We generate what is known as a **line plot**, which connects Cartesian pairs of points. We discussed such data pairs in Chapter 9. The horizontal axis will be the number of days that have elapsed since the first weight entry. The vertical axis will be the corresponding weight value. Such a plot will give us some idea of any trends noted over time. The steps to generate such a plot are fairly simple:

- Cycle through all values in the list box control, extracting the date and weight values. Store the number of elapsed days (difference between ‘current’ date and first date) in an array **d**. Store the corresponding weight values in an array **w**.
- Loop through all array elements, connecting consecutive points (with **d** as the horizontal point and **w** as the vertical point) using the **DrawLine** method.

In our work, both **d** and **w** will be **zero-based arrays** (to match up with the items of the list box control). Hence, each array will have **LstWeights.Items.Count** elements, numbered from **0** to **LstWeights.Items.Count – 1**.

The plot will be drawn on a graphics object hosted in a panel control (**pnlWeightPlot**). Recall the coordinate system used by such an object is:



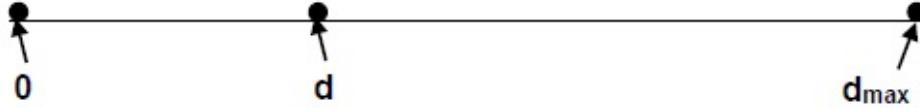
In this graphics object, the horizontal (**x**) coordinate increases from left to right, starting at **0** and extending to **ClientSize.Width - 1**. The vertical (**y**) coordinate increases from top to bottom, starting at **0** and ending at **ClientSize.Height - 1**. All measurements are **int** type and in units of **pixels**.

In the weight plot, the horizontal value (a date difference) will range from **0** (the first day in the weight file) to **d[IstWeights.Items.Count - 1]** (we'll call this **d<sub>max</sub>**, the difference between the last date in the list box control and the first date). This value increases from left to right. The vertical value will range from the minimum weight value (**w<sub>min</sub>**) to the maximum weight value (**w<sub>max</sub>**) - we will need to find these extremes. This value increases from bottom to top. Hence, to plot our data, we need to compute where each (**d**, **w**) pair in our weight plot fits within the dimensions of the graphics object specified by the **ClientSize.Width** and **ClientSize.Height** properties. This is a straightforward coordinate conversion computation. Recall we did such a conversion for the biorhythm plots.

Let's look at the horizontal axis first. As seen in the previous diagram, the horizontal (**x** axis) in the graphics object is **ClientSize.Width** pixels wide. The far left pixel is at **x = 0** and the far right is at **x = ClientSize.Width - 1**. **x** increases from left to right:



The horizontal weight plot value (**d**) runs from a minimum, **0**, at the left to a maximum, **d<sub>max</sub>**, at the right. Thus, the first pixel on the horizontal axis of our weight plot will be **0** and the last will be **d<sub>max</sub>**:



With these two depictions, we can compute the **x** value corresponding to a given **d** value using simple **proportions**, taking the distance from some point on each axis to the minimum and dividing by the total distance. The process is also called **linear interpolation**. These proportions show:

$$\frac{d - 0}{d_{\max} - 0} = \frac{x - 0}{\text{ClientSize.Width} - 1 - 0}$$

Solving this for **x** yields the desired conversion from a days value on the horizontal axis (**d**) to a graphics object value for plotting:

$$x = d(\text{ClientSize.Width} - 1)/d_{\max}$$

You can see this is correct at each extreme value. When **d = 0**, **x = 0**. When **d = d<sub>max</sub>**, **x = ClientSize.Width - 1**.

Now, we find the corresponding conversion for the vertical (**y**) axis. We'll place the two axes side-by-side for easy comparison (graphics object on left, weight axis on right):



The vertical ( $y$  axis) in the graphics object is **ClientSize.Height** pixels high. The topmost pixel is at  $y = 0$  and the bottom is at  $y = \text{ClientSize.Height} - 1$ .  $y$  increases from top to bottom. The vertical data (weight axis,  $w$ ) in our weight plot, runs from a minimum,  $w_{\min}$ , at the bottom, to a maximum,  $w_{\max}$ , at the top. Thus, the top pixel on the vertical axis will be  $w_{\max}$  and the bottom will be  $w_{\min}$  (note the weight axis increases up, rather than down).

With these two depictions, we can compute the  $y$  value corresponding to a given  $w$  value using linear interpolation. The computations show:

$$\frac{w - w_{\min}}{w_{\max} - w_{\min}} = \frac{y - (\text{ClientSize.Height} - 1)}{0 - (\text{ClientSize.Height} - 1)}$$

Solving this for  $y$  yields the desired conversion from a weight value on the vertical axis ( $w$ ) to a graphics object value for plotting (this requires a bit algebra, but it's straightforward):

$$y = (w_{\max} - w)(\text{ClientSize.Height} - 1)/(w_{\max} - w_{\min})$$

Again, check the extremes. When  $w = w_{\min}$ ,  $y = \text{ClientSize.Height} - 1$ . When  $w = w_{\max}$ ,  $y = 0$ . It looks good.

We will use two general methods to do these coordinate conversions. First, for the horizontal axis, we use **DToX**. This method has two input arguments: the **d** value and the maximum **d** value, **dmax**. Both values are of **double** data type.

The method returns the graphics object coordinate (an **int** type):

```
private int DToX(double d, double dmax)
{
    return ((int)(d * (pnIWeightPlot.ClientSize.Width - 1) / dmax));
}
```

Note this is used with the panel control where the plot will be drawn (**pnIWeightPlot**).

For the vertical axis, we use **WToY**. This method has three input arguments: the **w** value, the minimum **w** value, **wmin**, and the maximum value, **wmax**. All values are of **double** data type. The method returns the graphics object coordinate (an **int** type):

```
private int WToY(double w, double wmin, double wmax)
{
    return ((int)((wmax - w) * (pnIWeightPlot.ClientSize.Height - 1) /
(wmax - wmin)));
}
```

Add both these methods to your project.

With the ability to transform coordinates, we can now rewrite the steps to generate a weight plot:

- Cycle through all values in the list box control, extracting the date and weight values. Store the number of elapsed days (difference between ‘current’ date and first date) in an array **d**. Store the corresponding weight values in an array **w**. These are both zero-based arrays. While extracting values, determine the minimum and maximum weight values (**w<sub>min</sub>** and **w<sub>max</sub>**).
- Loop through all array elements. For each point, convert the **d** and **w** values to graphics object coordinates, then connect the current point with the previous point using the **DrawLine** function.

To insure persistent graphics in the weight plot, all code to draw the plot goes in

the **pnlWeightPlot** Paint event. The code to implement the above steps is:

```
private void pnlWeightPlot_Paint(object sender, PaintEventArgs e)
{
    double[] d = new double[lstWeights.Items.Count];
    double[] w = new double[lstWeights.Items.Count];
    double wmin, wmax;
    Graphics weightPlot;
    Pen myPen;
    if (lstWeights.Items.Count < 2)
        return;
    weightPlot = pnlWeightPlot.CreateGraphics();
    myPen = new Pen(Color.Blue, 2);
    wmin = 1000.0;
    wmax = 0.0;
    for (int i = 0; i < lstWeights.Items.Count; i++)
    {
        TimeSpan diff = GetDate(lstWeights.Items[i].ToString()) -
GetDate(lstWeights.Items[0].ToString());
        d[i] = diff.Days;
        w[i] =
Convert.ToDouble(GetWeight(lstWeights.Items[i].ToString()));
        wmin = Math.Min(w[i], wmin);
        wmax = Math.Max(w[i], wmax);
    }
    weightPlot.Clear(pnlWeightPlot.BackColor);
    for (int i = 1; i < lstWeights.Items.Count; i++)
    {
        // connect current point to previous point
        weightPlot.DrawLine(myPen, DToX(d[i - 1],
d[lstWeights.Items.Count - 1]), WToY(w[i - 1], wmin, wmax), DToX(d[i],
d[lstWeights.Items.Count - 1]), WToY(w[i], wmin, wmax));
    }
}
```

```

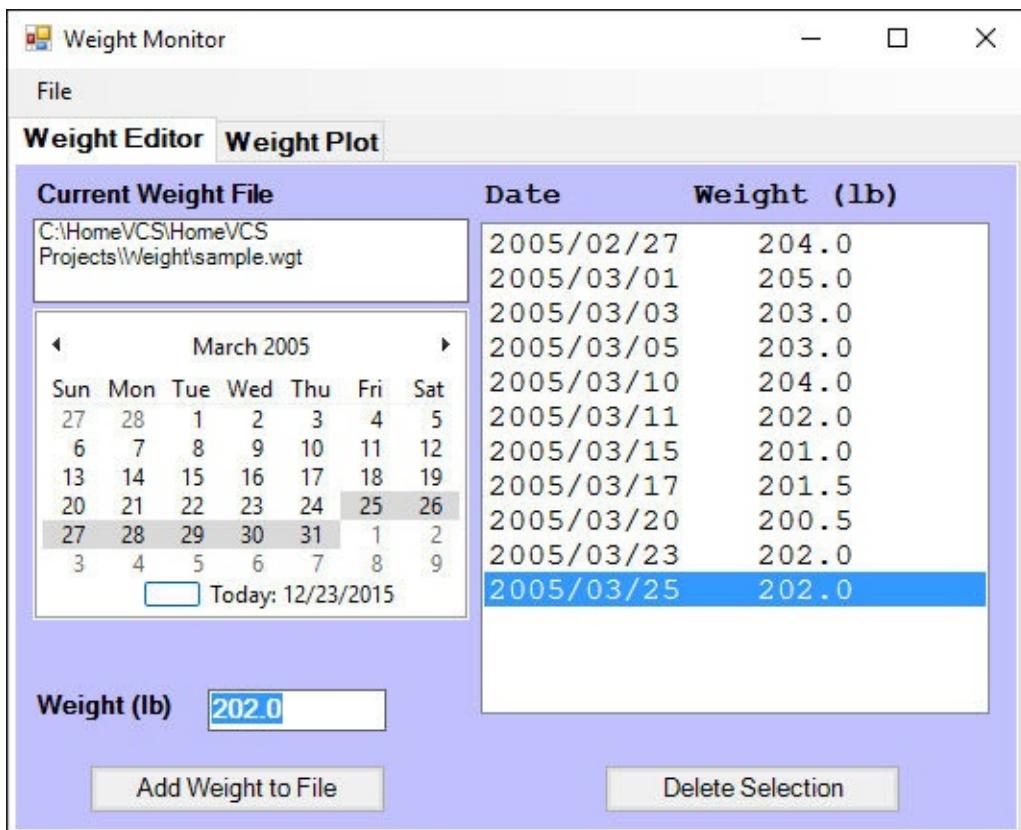
    weightPlot.Dispose();
    myPen.Dispose();
}

```

Add this method to your project.

Let's look at this code in detail. The graphics object and pen object for plotting are created. If there are fewer than 2 points, we return the user to the **Weight Editor** tab and exit the method, since you need two points to draw a line. Next, the **d** and **w** array values are obtained from the list box items and the minimum and maximum weight values are found. To draw the plot, we cycle through all points in the array, connecting the current point with the previous point. Make sure you understand these steps.

Save and run the project. If you opened the sample weight file before, the weight values for that file will be displayed. If they are not displayed, open the **sample.wgt** file found in the **HomeVCS\HomeVCS Projects\Weight** folder. You should see:



Now, click the **Weight Plot** tab and you will see this data in a line plot:



Success!

As drawn, the weight plot (though informative) is pretty boring. It lacks grid lines indicating weight values. And, it lacks labeling information to tell us what we're looking at. We need labels on the vertical axis telling us the weight range. We need labels on the horizontal axis telling us the represented date range. To add labels, we need to look at a couple of new graphics ideas – the **Brush** object and the **DrawString** method. We will do that. But first, let's address grid lines.

# Code Design – Grid Lines

With horizontal grid lines, we would be better able to determine plotted weight values. How many grid lines should there be and how far apart should they be spaced? We will use grid line spacing that results in a “nice” plot.

If you look back at the weight values in the **Weight Editor** tab, you will see that the weights range from a minimum of 200.5 to 205.0. So, as drawn, the bottom of the vertical axis in the plot is 200.5 and the top is 205.0. Notice the line plot hits these extremes in a few points. With this example, we could choose a grid line spacing of 0.5 pounds. That would result in 10 weight value labels (200.5, 201.0, 201.5, 202.0, 202.5, 203.0, 203.5, 204.0, 204.5, 205.0) and 8 grid lines (we don’t need grid lines at the bottom or top of the plot). Such a plot would be pretty cluttered, not a “nice” plot. Let’s develop some rules for nicer grid line spacing. We’ll use whole numbers for spacing and whole number for labels. And, we’ll try to make sure the weight plot never touches either vertical extreme.

Here’s the rules I use (you may come up with some others):

- Round maximum weight up to next integer value.
- Round minimum weight down to next integer value.
- If difference between maximum and minimum is less than 5 pounds, set grid line spacing to 1 pound.
- If difference between maximum and minimum is less than 10 pounds, set grid line spacing to 2 pounds.
- If difference between maximum and minimum is less than 25 pounds, set grid line spacing to 5 pounds.
- If difference between maximum and minimum is less than 50 pounds, set grid line spacing to 10 pounds.
- For larger differences, use a grid line spacing of 20 pounds.
- Adjust maximum value to next highest integer multiple of the grid line spacing (if necessary).
- Adjust minimum value to next lowest integer multiple of the grid line spacing (if necessary).

Once the grid line spacing is determined, the grid lines can be drawn using the **DrawLine** method. As stated, grid lines are drawn at each vertical position, except the bottom and top.

The modified **pnlWeightPlot** Paint event that computes grid line spacing and draws the grid lines is (changes are shaded):

```
private void pnlWeightPlot_Paint(object sender, PaintEventArgs e)
{
    double[] d = new double[lstWeights.Items.Count];
    double[] w = new double[lstWeights.Items.Count];
    double wmin, wmax;
    int intervals;
    double gridSpacing, wLegend;
    Graphics weightPlot;
    Pen myPen;
    if (lstWeights.Items.Count < 2)
        return;
    weightPlot = pnlWeightPlot.CreateGraphics();
    myPen = new Pen(Color.Blue, 2);
    wmin = 1000.0;
    wmax = 0.0;
    for (int i = 0; i < lstWeights.Items.Count; i++)
    {
        TimeSpan diff = GetDate(lstWeights.Items[i].ToString()) -
GetDate(lstWeights.Items[0].ToString());
        d[i] = diff.Days;
        w[i] =
Convert.ToDouble(GetWeight(lstWeights.Items[i].ToString()));
        wmin = Math.Min(w[i], wmin);
        wmax = Math.Max(w[i], wmax);
    }
    // adjust Wmin/Wmax for 'nice' intervals
```

```

if (wmin == wmax)
    wmin = wmax - 1;
    wmax = (double) ((int)(wmax + 0.5)); // round up
    wmin = (double) ((int)(wmin - 0.5)); // round down
    if (wmax - wmin <= 5.0)
        gridSpacing = 1.0;
    else if (wmax - wmin <= 10.0)
        gridSpacing = 2.0;
    else if (wmax - wmin <= 25.0)
        gridSpacing = 5.0;
    else if (wmax - wmin <= 50.0)
        gridSpacing = 10.0;
    else
        gridSpacing = 20.0;
    if (wmax % (int)gridSpacing != 0)
        wmax = gridSpacing * (int)(wmax / gridSpacing) + gridSpacing;
    if (wmin % (int)gridSpacing != 0)
        wmin = gridSpacing * (int)(wmin / gridSpacing);
    intervals = (int)((wmax - wmin) / gridSpacing);

    weightPlot.Clear(pnlWeightPlot.BackColor);
    for (int i = 1; i < lstWeights.Items.Count; i++)
    {
        // connect current point to previous point
        weightPlot.DrawLine(myPen, DToX(d[i - 1],
            d[lstWeights.Items.Count - 1]), WToY(w[i - 1], wmin, wmax), DToX(d[i],
            d[lstWeights.Items.Count - 1]), WToY(w[i], wmin, wmax));
    }

    wLegend = wmin;
    for (int i = 0; i <= intervals; i++)
    {
        if (i > 0 && i < intervals)
        {

```

```
// draw grid line (except at top and bottom)
    weightPlot.DrawLine(Pens.Black, 0, WToY(wLegend, wmin,
wmax), pnlWeightPlot.ClientSize.Width, WToY(wLegend, wmin,
wmax));
}
wLegend += gridSpacing;
}

weightPlot.Dispose();
myPen.Dispose();
}
```

In this code, **gridSpacing** is the spacing between grid lines and **intervals** is the number of grid intervals between **wmin** and **wmax**. **wLegend** is the weight value at the current grid line (it starts at **wmin** and increases by **gridSpacing** after drawing a grid line). The grid lines are drawn with a black pen with 1 pixel width. You should see all the grid spacing calculation steps. Make the indicated changes in your project.

Save and run the project. Make sure the **sample.wgt** file is opened. Click the **Weight Plot** tab. Here's the modified plot:



Notice the grid lines (they're spaced apart by 1 pound, by the way). This is a nicer plot. It's still not "nice enough." There's no indication of what the grid line spacing is. We don't know what the weight range is. We don't know what the date range is. All that information is provided with plot labeling. Before adding the labeling though, we need to discuss **Brush** objects and the **DrawString** graphics method (both used to "draw" the labels).

# Brush Object

We will soon learn how to add text to a graphic object using the **DrawString** method. The text in that method is drawn with a **Brush** object. Like the Pen object, a brush is just like a brush you use to paint – just pick a color. You can use brushes built-in to Visual C# or create your own brush. In this chapter, we only look at solid color brushes.

In most cases, the brush objects built into Visual C# are sufficient. The **Brushes** class provides brush objects that paint using one of the 141 built-in color names we've seen before. The syntax to refer to such a brush is:

**Brushes.ColorName**

To create your own **Brush** object (**myBrush**), you first declare the brush using:

**Brush myBrush;**

The solid color brush is then created using the **SolidBrush** constructor:

**myBrush = new SolidBrush(Color);**

where **Color** is the color your new brush will paint with. This color argument can be one of the built-in colors or one generated with the **FromArgb** function.

Once created, you can change the color of a brush any time using the **Color** property of the brush object. The syntax is:

**myBrush.Color = newColor;**

where **newColor** is a newly specified color.

When done painting with a brush object, it should be disposed using the **Dispose** method:

**myBrush.Dispose();**

# DrawString Method

The **DrawString** method will let us add text (like plot labels) to any graphic object. The **DrawString** method is easy to use. You need to know what text you want to draw, what font you want to use, what brush object to use and where you want to locate the text. The method operates on a previously created graphics object (use **CreateGraphics** method with a control). The syntax is:

```
myGraphics.DrawString(myString, myFont, myBrush, x, y);
```

where **myGraphics** is the graphics object, **myString** is the text to display (a **string** type), **myFont** is the font to use (**Font** object), **myBrush** is the selected brush (**Brush** object) and (**x**, **y**) is the Cartesian coordinate (**int** type) specifying location of the upper left corner of the text string

Let's look at an example. Can you see what this line of code will produce?

```
myGraphics.DrawString("Hello World!", new Font("Arial", 24),  
Brushes.Red, 0, 0);
```

This says print the string “**Hello World!**” using an **Arial**, **Size 24**, font and a **red solid brush** in the upper left hand corner (**x = 0**, **y = 0**) of the graphics object **myGraphics**.

A key decision in using **DrawString** is placement. That is, what **x** and **y** values should you use? To help in this decision, it is helpful to know what size a particular text string is. If we know how wide and how tall (in pixels) a string is, we can perform precise placements, including left, center and right justifications.

If you are only interested in obtaining the height of a string for correct vertical placement, you can use the Visual C# **GetHeight** method. This method operates on the font object. If you have a font object named **myFont**, the height is given by:

```
myHeight = myFont.GetHeight();
```

The returned **myHeight** value (**float** data type) is in pixels.

If you need both width and height, the Visual C# method **MeasureString** gives us this information. The size (**mySize**) of a string **myString** written using **myFont** is computed using:

```
mySize = myGraphics.MeasureString(myString, myFont);
```

The returned value **mySize** is a **SizeF** structure (another Visual C# concept) with two properties, **Width** and **Height**. To determine string size, we need a **SizeF** structure.

There are two steps involved in creating a **SizeF** structure. We first declare the structure using the standard statement:

```
SizeF mySize;
```

Placement of this statement depends on scope. Place it in a method for method level scope. Place it with other form level declarations for form level scope. Once declared, the structure is created using the **SizeF** constructor:

```
mySize = new SizeF();
```

No arguments are needed. You can assign width and height arguments, but we don't need to. We are just using **mySize** to determine such properties.

If we use this **SizeF** structure with our previous example:

```
mySize = myGraphics.MeasureString("Hello World!", new Font("Arial", 24));
```

We would see (you could use the debugger to get these values):

**mySize.Width = 190.151**

**mySize.Height = 39.75**

These measurements are floating point representations (type **double**) of pixels.

The ‘F’ in **SizeF** implies a floating point, rather than integer, number. There is also a **Size** structure in Visual C# that provides integer information. This structure, though, can’t be used with **MeasureString**.

The height of a string lets us know how much to increment the desired vertical position after displaying each line in multiple lines of text. Or, it can be used to ‘vertically justify’ a string within a graphics object. For example, assume we have found the height of a string (**mySize.Height**). To vertically justify this string in the host control (**myObject**) for a graphics object, the y coordinate (converted to an **int** type needed by **DrawString**) would be:

```
y = (int)(0.5 * (myObject.ClientSize.Height - mySize.Height));
```

This assumes the string is ‘shorter’ than the graphics object.

Similarly, the width of a string lets us define margins and left, right or center justify a string within a graphics object. For left justification, establish a left margin and set the **x** coordinate to this value in the **DrawString** method. If we know the width of a string (**mySize.Width**), it is centered justified in a graphics object’s host control **myObject** using an **x** value of (again converted to **int**):

```
x = (int)(0.5 * (myObject.ClientSize.Width - mySize.Width));
```

To right justify the same string, use:

```
x = (int)(myObject.ClientSize.Width - mySize.Width);
```

Both of the above equations, of course, assume the string is ‘narrower’ than the graphics object.

We can now use **Brush** objects and the **DrawString** method to add labels to our weight plot.

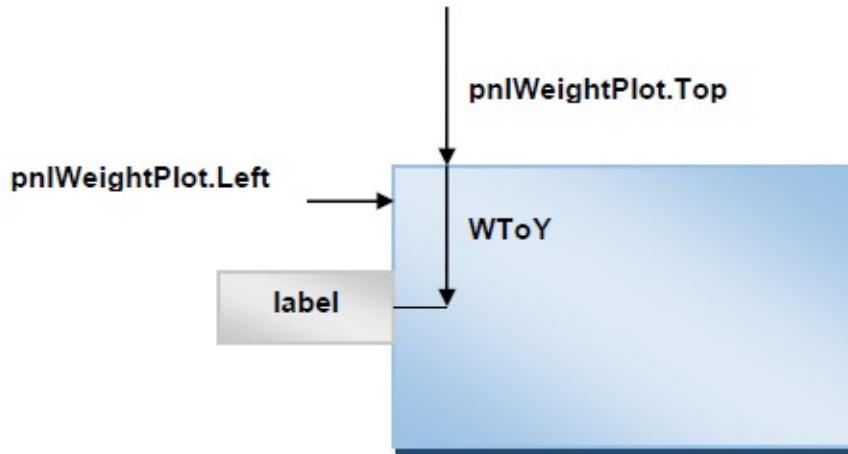
# Code Design – Plot Labels

We will add labels (not label controls, just text information for labeling) for the weight axis first. When drawing the grid lines, we wrote code that specified the weight values (**wLegend**) for the labels. We just need to add code that converts these values to strings and places them in the appropriate location on the plot tab page. For each **wLegend** value:

- Convert **wLegend** to **string** type (formatted with no decimal places)
- Determine width and height of string.
- Position string in proper vertical location and right justified to left side of plot.

The **MeasureString** method will be used to find the width and height of the string. The **WToY** method helps position the string label vertically.

Each label will be “drawn” outside the plot area on **pnlWeight** (not **pnlWeightPlot**), hence another graphics object will be needed. Here’s a sketch that shows you how one label (for a weight value **W**) is positioned:



To horizontally position **label**, you use:

**x = pnlWeightPlot.Left – label.Width;**

To vertically position **label**, use:

```
y = pnlWeightWeightPlot.Top + WtoY(w, wmin, wmax) - (int)(0.5 *  
label.Height);
```

The pair (x, y) are used to position the string label using **DrawString** on **pnlWeight**. The width and height of the label are found using the **MeasureString** method.

The code to draw the weight axis labels is processed in the **pnlWeight Paint** event. We could write a separate **pnlWeight Paint** event method. However, each time **pnlWeight** needs to painted, **pnlWeightPlot** will also need to be painted. Hence, we will put code that draws to **pnlWeight** in the **pnlWeightPlot Paint** event, making sure the **pnlWeight Paint** event accesses this method. This centralizes all the plot drawing code. The modified **pnlWeightPlot Paint** event method that adds weight axis labeling is (modifications are shaded; much unmodified code is not shown):

```
private void pnlWeightPlot_Paint(object sender, PaintEventArgs e)  
{  
    double[] d = new double[lstWeights.Items.Count];  
    double[] w = new double[lstWeights.Items.Count];  
    double wmin, wmax;  
    int intervals;  
    double gridSpacing, wLegend;  
    Graphics weightPlot;  
    Graphics legendArea;  
    string s;  
    SizeF sSize;  
    Font legendFont;  
    Pen myPen;  
    .  
    .  
    weightPlot.Clear(pnlWeightPlot.BackColor);  
    for (int i = 1; i < lstWeights.Items.Count; i++)  
    {
```

```

    // connect current point to previous point
    weightPlot.DrawLine(myPen, DToX(d[i - 1],
d[lstWeights.Items.Count - 1]), WToY(w[i - 1], wmin, wmax), DToX(d[i],
d[lstWeights.Items.Count - 1]), WToY(w[i], wmin, wmax));
}

legendArea = pnlWeight.CreateGraphics();
legendFont = new Font("Arial", 10, FontStyle.Bold);
wLegend = wmin;
for (int i = 0; i <= intervals; i++)
{
    s = wLegend.ToString();
    sSize = legendArea.MeasureString(s, legendFont);
    legendArea.DrawString(s, legendFont, Brushes.Black,
pnlWeightPlot.Left - sSize.Width, pnlWeightPlot.Top + WToY(wLegend,
wmin, wmax) - (int)(0.5 * sSize.Height));

    if (i > 0 && i < intervals)
    {
        // draw grid line (except at top and bottom)
        weightPlot.DrawLine(Pens.Black, 0, WToY(wLegend, wmin,
wmax), pnlWeightPlot.ClientSize.Width, WToY(wLegend, wmin,
wmax));
    }
    wLegend += gridSpacing;
}
weightPlot.Dispose();
legendArea.Dispose();
myPen.Dispose();
}

```

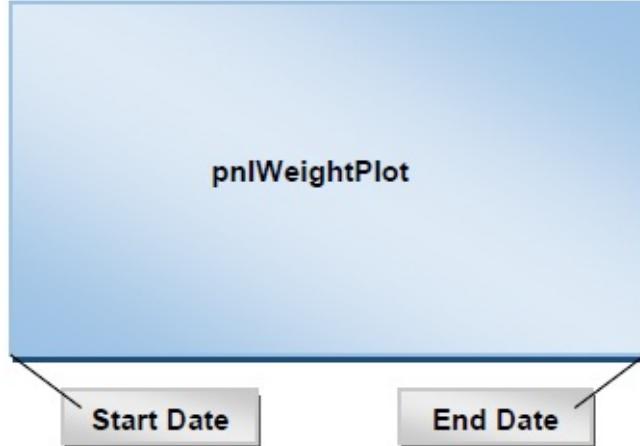
We use a black brush to draw the text. Make the noted changes to the method.

Save and run the project. The values for **sample.wgt** should appear. Click the **Weight Plot** tab:



Our plot now has very nice labels.

Now, we add labels to the horizontal weight plot axis. Recall this axis tells us how many days have elapsed since we started the weight file. We could label the axis with such day values, choosing an appropriate horizontal spacing. Instead of doing this, we will simply label the axis with the starting date and the ending date. We feel this is more meaningful information. The form of the labeling will be:



That is, we will display the two dates and draw lines indicating where these dates fall on the plot. Recall the **start** date is given by **GetDate(lstWeights.Items[0].ToString())** and the **end** date is given by **GetDate(lstWeights.Items[lstWeights.Items.Count - 1].ToString())**.

The code to generate these labels also goes in the **pnlWeightPlot Paint** event method. The new method is (once again, changes are shaded with much unmodified code not shown):

```
private void pnlWeightPlot_Paint(object sender, PaintEventArgs e)
{
    .
    .
    .
    legendArea = pnlWeight.CreateGraphics();
    legendFont = new Font("Arial", 10, FontStyle.Bold);
    s = "Start: " + String.Format("{0:MMMM d, yyyy}",
GetDate(lstWeights.Items[0].ToString()));
    legendArea.DrawString(s, legendFont, Brushes.Black,
pnlWeightPlot.Left + 10, pnlWeightPlot.Top + pnlWeightPlot.Height +
10);
    legendArea.DrawLine(Pens.Black, pnlWeightPlot.Left + 10,
pnlWeightPlot.Top + pnlWeightPlot.Height + 10, pnlWeightPlot.Left,
pnlWeightPlot.Top + pnlWeightPlot.Height);
    s = "End: " + String.Format("{0:MMMM d, yyyy}",
GetDate(lstWeights.Items[lstWeights.Items.Count - 1].ToString()));
```

```
sSize = legendArea.MeasureString(s, legendFont);
legendArea.DrawString(s, legendFont, Brushes.Black,
pnlWeightPlot.Left + pnlWeightPlot.Width - sSize.Width - 10,
pnlWeightPlot.Top + pnlWeightPlot.Height + 10);
legendArea.DrawLine(Pens.Black, pnlWeightPlot.Left +
pnlWeightPlot.Width - 10, pnlWeightPlot.Top + pnlWeightPlot.Height +
10, pnlWeightPlot.Left + DToX(d[lstWeights.Items.Count - 1],
d[lstWeights.Items.Count - 1]), pnlWeightPlot.Top +
pnlWeightPlot.Height);

wLegend = wmin;
.
.
}

}
```

You should be able to see how the labels are formed and positioned. Make the noted changes.

Save and run the project. Click **Weight Plot** (using the **sample.wgt** values) and you should see:



Don't you agree the plot looks much nicer with labels?

# Code Design – Weight Plot Trend

We're almost done with our weight monitor project. Just one more change. When you track your weight with a plot, you want to know how you're doing with your weight management plan. Are you gaining weight? Losing weight? Maintaining weight? You want to know if there are any trends in the plotted values.

We want to add a ‘**trend line**’ to the weight plot. Such a straight line can give us some idea of what direction our weight is going in. A very simple trend line would be to connect the first point in the plot to the last point. This approach, however, ignores all other points in the plot. The approach we take will consider every point in the plot, but, be forewarned, some mathematics are needed.

The trend line we use will represent a “best fit” to all the points in the weight plot. Mathematically speaking, we do a **linear regression** on the data. This regression involves calculus and solving linear equations, so we won’t bore you with the details (unless you want to see them). We’ll just give you the needed equations so they can be added to the project code.

Our trend line ‘models’ the weight values using the straight line equation:

$$w_m = td + w_0$$

where:

$w_m$  – modeled weight

$d$  – horizontal axis value (number of days since first weight entry)

$t$  – trend value (pounds/day), called the slope of the line

$w_0$  – modeled weight when  $d = 0$

With the above model, the trend line connects two Cartesian end points:  $(0, w_0)$  and  $(d_{max}, td_{max} + w_0)$ . So, to draw the trend line, we need to know values for  $t$  and  $w_0$  (we know  $d_{max}$ ). Values for these two terms are found using the  $d$  and  $w$  arrays currently used to create the weight plots. The equations for  $t$  and  $w_0$ , using these arrays are (these equations come from the linear regression we mentioned):

$$t = \frac{N \sum_{k=0}^{N-1} d[k]w[k] - \sum_{k=0}^{N-1} d[k] \sum_{k=0}^{N-1} w[k]}{N \sum_{k=0}^{N-1} d^2[k] - (\sum_{k=0}^{N-1} d[k])^2}$$

$$w_0 = \frac{\sum_{k=0}^{N-1} d^2[k] \sum_{k=0}^{N-1} w[k] - \sum_{k=0}^{N-1} d[k] \sum_{k=0}^{N-1} d[k]w[k]}{N \sum_{k=0}^{N-1} d^2[k] - (\sum_{k=0}^{N-1} d[k])^2}$$

where recall the Greek sigma in the above equations indicates you add up all the corresponding elements next to the sigma. Also, **N** is the number of elements in each array (**IstWeights.Items.Count**).

I know the above equations are messy, but they yield a very nice trend line and are straightforward to program. You simply declare a variable for each of the summation terms and form the sums as you establish the two arrays **d** and **w**. Then a little math gives you values for **t** and **w<sub>0</sub>**.

For those interested in the mathematics involved in deriving these relations, I'll outline them for you. For those not interested, leave this paragraph now. The idea behind linear regression is to minimize the squared error between the modeled weight points and the actual weight points. That error (**e**) is given by:

$$e = \sum_{k=0}^{N-1} \{w[k] - (td[k] + w_0)\}^2$$

We want **e** to be as small as possible, seeking the so-called least square error solution. For **e** to be minimum the partial derivative of **e** with respect to **t** and the partial derivative with respect to **w<sub>0</sub>** must be zero. Those derivatives are (here's where the calculus shows up):

$$2 \sum_{k=0}^{N-1} \{w[k] - (td[k] + w_0)\}d[k] = 0 \text{ (partial derivative with respect to } t\text{)}$$

$$2 \sum_{k=0}^{N-1} \{w[k] - (td[k] + w_0)\} = 0 \text{ (partial derivative with respect to } w_0\text{)}$$

If we rearrange these equations a bit, we get:

$$t \sum_{k=0}^{N-1} d^2[k] + w_0 \sum_{k=0}^{N-1} d[k] = \sum_{k=0}^{N-1} d[k]w[k]$$

$$t \sum_{k=0}^{N-1} d[k] + w_0 N = \sum_{k=0}^{N-1} w[k]$$

We have two linear equations with two unknowns ( $t$  and  $w_0$ ). We can use Cramer's rule to solve these equations to yield the previously seen relations for  $t$  and  $w_0$ .

The code to compute and draw the trend line goes in the **pnlWeight Plot** event method. In addition to drawing the trend line, we add a label at the top of the plot to indicate a “weekly” trend value ( $7 * t$ ), showing how much weight you are losing or gaining each week. The modified method is (changes are shaded):

```
private void pnlWeightPlot_Paint(object sender, PaintEventArgs e)
{
    double[] d = new double[lstWeights.Items.Count];
    double[] w = new double[lstWeights.Items.Count];
    double wmin, wmax;
    int intervals;
    double gridSpacing, wLegend;
    double sumD, sumD2, sumW, sumDW;
    double t, w0;

    Graphics weightPlot;
    Graphics legendArea;
    string s;
```

```

SizeF sSize;
Font legendFont;
Pen myPen;
if (lstWeights.Items.Count < 2)
    return;
weightPlot = pnlWeightPlot.CreateGraphics();
myPen = new Pen(Color.Blue, 2);
wmin = 1000.0;
wmax = 0.0;
sumD = 0.0;
sumD2 = 0.0;
sumW = 0.0;
sumDW = 0.0;
for (int i = 0; i < lstWeights.Items.Count; i++)
{
    TimeSpan diff = GetDate(lstWeights.Items[i].ToString()) -
GetDate(lstWeights.Items[0].ToString());
    d[i] = diff.Days;
    w[i] =
Convert.ToDouble(GetWeight(lstWeights.Items[i].ToString()));
    wmin = Math.Min(w[i], wmin);
    wmax = Math.Max(w[i], wmax);
    // values for trend line
    sumD += d[i];
    sumD2 += d[i] * d[i];
    sumW += w[i];
    sumDW += d[i] * w[i];
}
// adjust Wmin/Wmax for 'nice' intervals
if (wmin == wmax)
    wmin = wmax - 1;
wmax = (double) ((int)(wmax + 0.5)); // round up

```

```

wmin = (double) ((int)(wmin - 0.5)); // round down
if (wmax - wmin <= 5.0)
    gridSpacing = 1.0;
else if (wmax - wmin <= 10.0)
    gridSpacing = 2.0;
else if (wmax - wmin <= 25.0)
    gridSpacing = 5.0;
else if (wmax - wmin <= 50.0)
    gridSpacing = 10.0;
else
    gridSpacing = 20.0;
if (wmax % (int)gridSpacing != 0)
    wmax = gridSpacing * (int)(wmax / gridSpacing) + gridSpacing;
if (wmin % (int)gridSpacing != 0)
    wmin = gridSpacing * (int)(wmin / gridSpacing);
intervals = (int)((wmax - wmin) / gridSpacing);
weightPlot.Clear(pnlWeightPlot.BackColor);
for (int i = 1; i < lstWeights.Items.Count; i++)
{
    // connect current point to previous point
    weightPlot.DrawLine(myPen, DToX(d[i - 1],
d[lstWeights.Items.Count - 1]), WToY(w[i - 1], wmin, wmax), DToX(d[i],
d[lstWeights.Items.Count - 1]), WToY(w[i], wmin, wmax));
}
legendArea = pnlWeight.CreateGraphics();
legendFont = new Font("Arial", 10, FontStyle.Bold);
s = "Start: " + String.Format("{0:MMMM d, yyyy}",
GetDate(lstWeights.Items[0].ToString()));
legendArea.DrawString(s, legendFont, Brushes.Black,
pnlWeightPlot.Left + 10, pnlWeightPlot.Top + pnlWeightPlot.Height +
10);
legendArea.DrawLine(Pens.Black, pnlWeightPlot.Left + 10,
pnlWeightPlot.Top + pnlWeightPlot.Height + 10, pnlWeightPlot.Left,

```

```

    pnlWeightPlot.Top + pnlWeightPlot.Height);
    s = "End: " + String.Format("{0:MMMM d, yyyy}",
GetDate(lstWeights.Items[lstWeights.Items.Count - 1].ToString())));
    sSize = legendArea.MeasureString(s, legendFont);
    legendArea.DrawString(s, legendFont, Brushes.Black,
pnlWeightPlot.Left + pnlWeightPlot.Width - sSize.Width - 10,
pnlWeightPlot.Top + pnlWeightPlot.Height + 10);
    legendArea.DrawLine(Pens.Black, pnlWeightPlot.Left +
pnlWeightPlot.Width - 10, pnlWeightPlot.Top + pnlWeightPlot.Height +
10, pnlWeightPlot.Left + DToX(d[lstWeights.Items.Count - 1],
d[lstWeights.Items.Count - 1]), pnlWeightPlot.Top +
pnlWeightPlot.Height);

    wLegend = wmin;
    for (int i = 0; i <= intervals; i++)
    {
        s = wLegend.ToString();
        sSize = legendArea.MeasureString(s, legendFont);
        legendArea.DrawString(s, legendFont, Brushes.Black,
pnlWeightPlot.Left - sSize.Width, pnlWeightPlot.Top + WToY(wLegend,
wmin, wmax) - (int)(0.5 * sSize.Height));
        if (i > 0 && i < intervals)
        {
            // draw grid line (except at top and bottom)
            weightPlot.DrawLine(Pens.Black, 0, WToY(wLegend, wmin,
wmax), pnlWeightPlot.ClientSize.Width, WToY(wLegend, wmin,
wmax));
        }
        wLegend += gridSpacing;
    }

    // trend computations
    t = (lstWeights.Items.Count * sumDW - sumD * sumW) /
(lstWeights.Items.Count * sumD2 - sumD * sumD);
    w0 = (sumD2 * sumW - sumD * sumDW) / (lstWeights.Items.Count *
sumD2 - sumD * sumD);

```

```

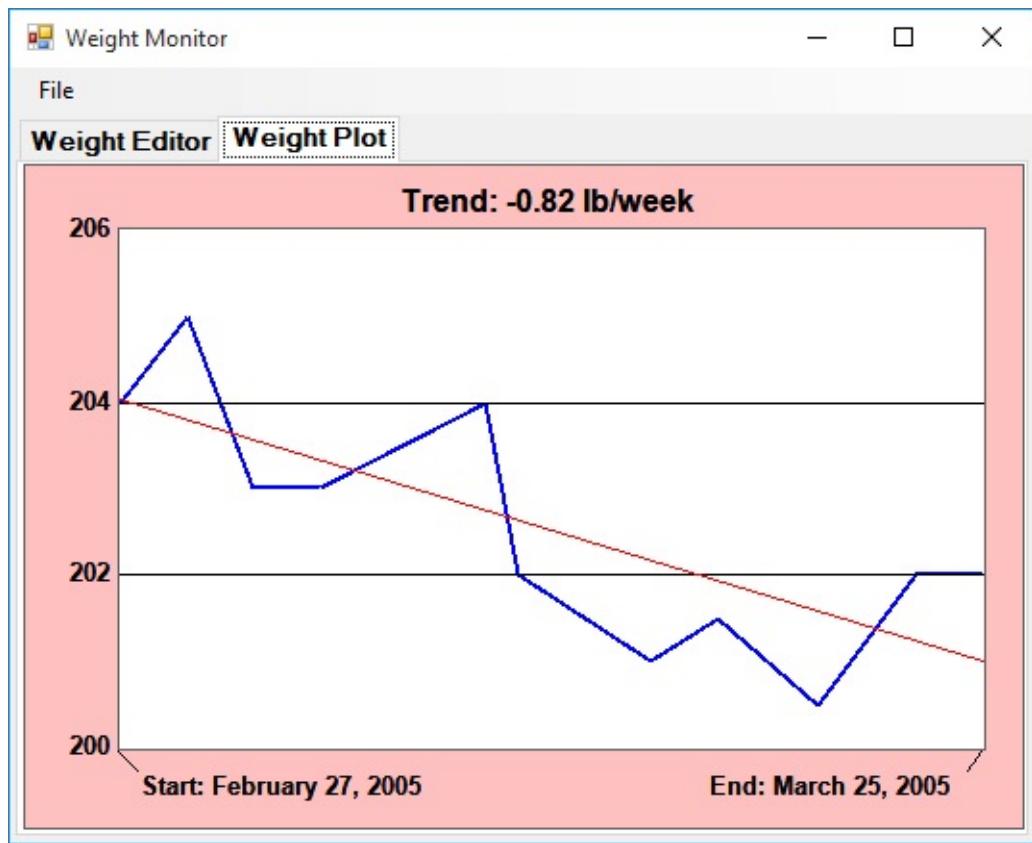
// draw line
    weightPlot.DrawLine(Pens.Red, 0, WToY(w0, wmin, wmax),
DToX(d[lstWeights.Items.Count - 1], d[lstWeights.Items.Count - 1]),
WToY(t * d[lstWeights.Items.Count - 1] + w0, wmin, wmax));
    legendFont = new Font("Arial", 12, FontStyle.Bold);
    s = "Trend: ";
    if (t > 0)
        s += "+";
    s += String.Format("{0:f2}", 7 * t) + " lb/week";
    sSize = legendArea.MeasureString(s, legendFont);
    legendArea.DrawString(s, legendFont, Brushes.Black,
pnlWeightPlot.Left + (int)(0.5 * pnlWeightPlot.Width - 0.5 *
sSize.Width), pnlWeightPlot.Top - sSize.Height - 3);

    weightPlot.Dispose();
    legendArea.Dispose();
    myPen.Dispose();
}

```

This is the final version of the **pnlWeightPlot Paint** event. Make the noted changes. You should see how the trend line is computed and drawn. Also notice how the trend value is printed at the top of the plot.

Save and run the project. Click **Weight Plot** one last time using **sample.wgt** and you will see a nice red trend line and corresponding label indicating I'm losing nearly a pound a week:



# Weight Monitor Project Review

The **Weight Monitor Project** is now complete. Save and run the project and make sure it works as designed. Use the program to track your weight each day (or let your family try it). Hopefully the program can become an integral part of an overall health program.

If there are errors in your implementation, go back over the steps of form and code design. Use the debugger when needed. Go over the developed code – make sure you understand how different parts of the project were coded. As mentioned in the beginning of this chapter, the completed project is saved as **Weight** in the **HomeVCS\HomeVCS Projects** folder.

While completing this project, new concepts and skills you should have gained include:

- Use of the tab, list box and save file dialog controls.
- Input/output of variables with sequential files.
- Using configuration files in projects.
- Creating brush objects.
- Adding text to a graphics object.
- Doing unit conversions need for plotting.
- Making “nice” intervals for plots.

# Weight Monitor Project Enhancements

Possible enhancements to the weight monitor project include:

- We discuss printing in the next chapter. Once you understand how to print, you might like to add such capabilities to the weight monitor project. Print out your date/weight values. Print out the weight plot, including trend line.
- Many times, you are trying to achieve a certain weight goal. Modify the program to allow a user to enter a desired goal and a desired goal date. Provide computations that show how well the user is doing in trying to reach this goal. Draw a “goal” line on the weight plot.
- As implemented, weights need to be in pounds. Most of the world uses kilograms for weight. Add the capability to choose either unit for weight. You’ll have to decide if you want to change any current values to the new units or just keep file in one particular set of units.

**10**

## **Home Inventory Manager Project**

# Review and Preview

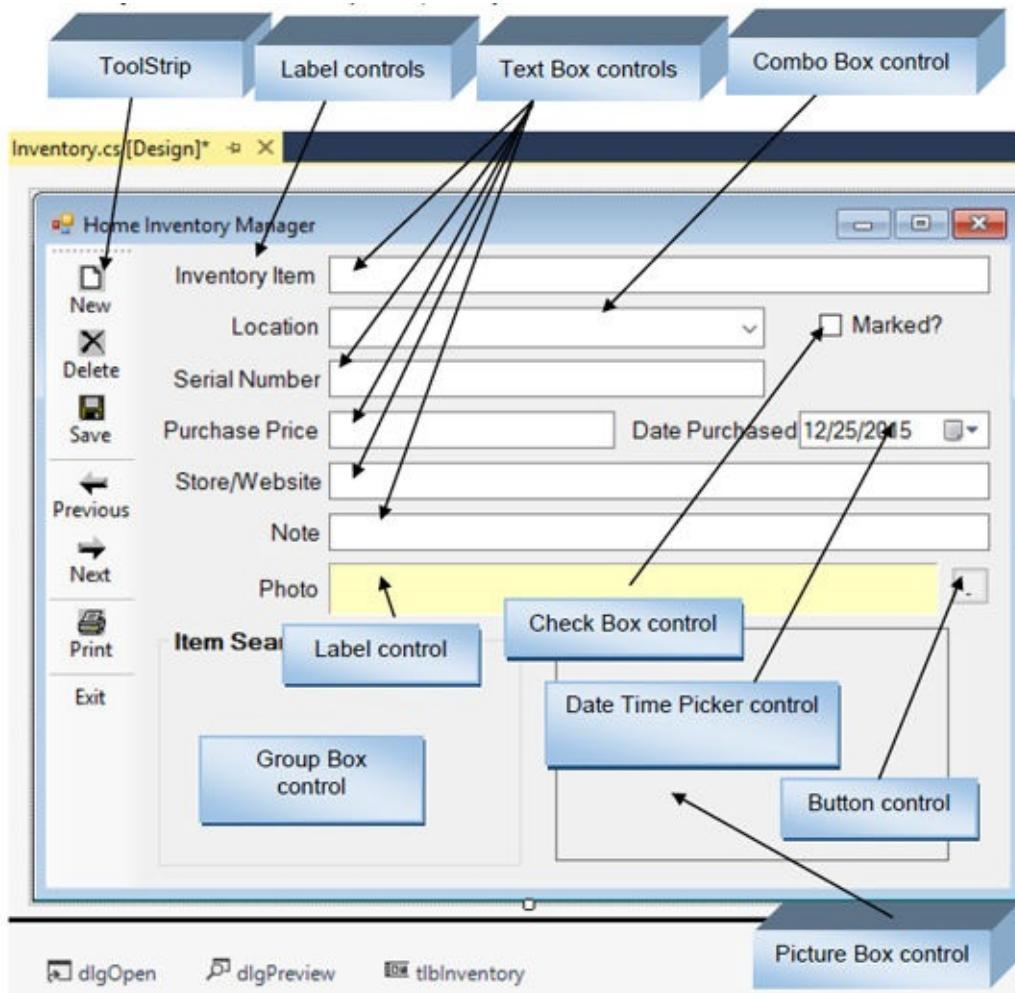
The **Home Inventory Manager Project** helps you keep track of your valuable belongings. For every item in your inventory, the program stores a description, location, serial number, purchase information, and even a photo! A printed inventory is available - very useful for insurance purposes.

Four new controls are introduced – the toolbar, the combo box and the print preview dialog. We also introduce some object-oriented programming concepts and how to print from a project.

# Home Inventory Manager Project Preview

In this chapter, we will build a **home inventory manager** program. This program lets you keep a record of your belongings.

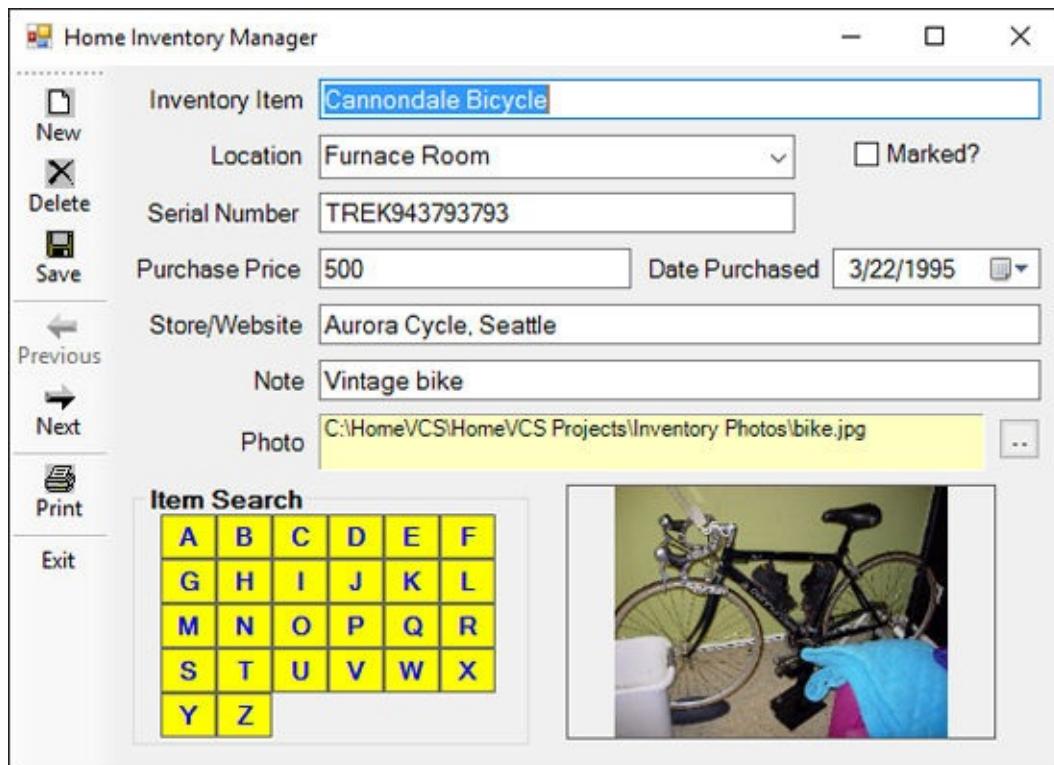
The finished project is saved as **Inventory** in the **HomeVCS\HomeVCS Projects** folder. Start Visual C# and open the finished project. Open the form (double-click **Inventory.cs** in Solution Explorer) and you will see:



A toolStrip control is used to add, delete and save items from the inventory. It is also used to navigate from one item to the next. The primary way to enter

information about an inventory item is with several text box controls. A combo box is used to specify location, while a date time picker is used to select purchase date. A check box control indicates if an item is marked with identifying information. A button control (with an ellipsis) selects a photo to display in the picture box control. A group box control will hold “clickable” labels for searching (these labels will be added at run-time). At the bottom are a toolbar control used to specify the toolbar, an open file dialog is used to open photo files and a print preview control lets you see your inventory in printed form. The toolbar, combo box and print preview dialog controls are all new – we will discuss these in some detail.

Run the project (press <F5>). The program has a built-in sample inventory file – the first item in that file will display (items are listed alphabetically by **Inventory Item**):



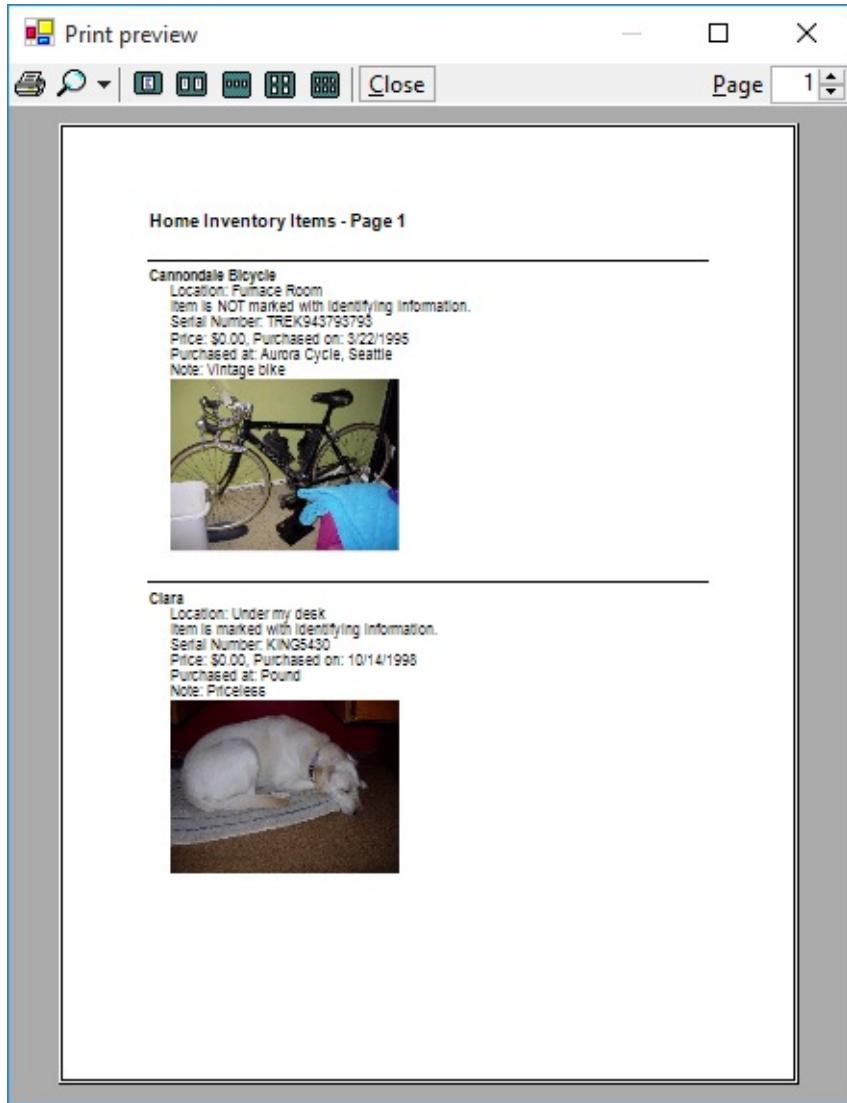
You will, of course, be able to replace the built-in file with your own belongings, but for now, let's see how the program works.

The idea of the program is to enter and/or view descriptive information about each item in your inventory. You can enter:

<b>Inventory Item</b>	A description of the item (required)
<b>Location</b>	Description of where item is located
<b>Marked</b>	Indicates if item is marked with some kind of identifying information (social security number, driver's license number, phone number)
<b>Serial Number</b>	Item serial number
<b>Purchase Price</b>	How much you paid for the item.
<b>Date Purchased</b>	When you purchased the item.
<b>Store/Website</b>	Where you purchased the item.
<b>Note</b>	Any additional information about the item.
<b>Photo</b>	View a stored JPEG photo of the item.

On the toolbar are two buttons marked **Previous** and **Next**. Use these to move from one item to the next. The sample file has 10 items to view. In the **Item Search** group box are 26 labels, each with a letter of the alphabet. These are used to search through the inventory for items beginning with the clicked letter. Try searching the sample inventory, if you'd like.

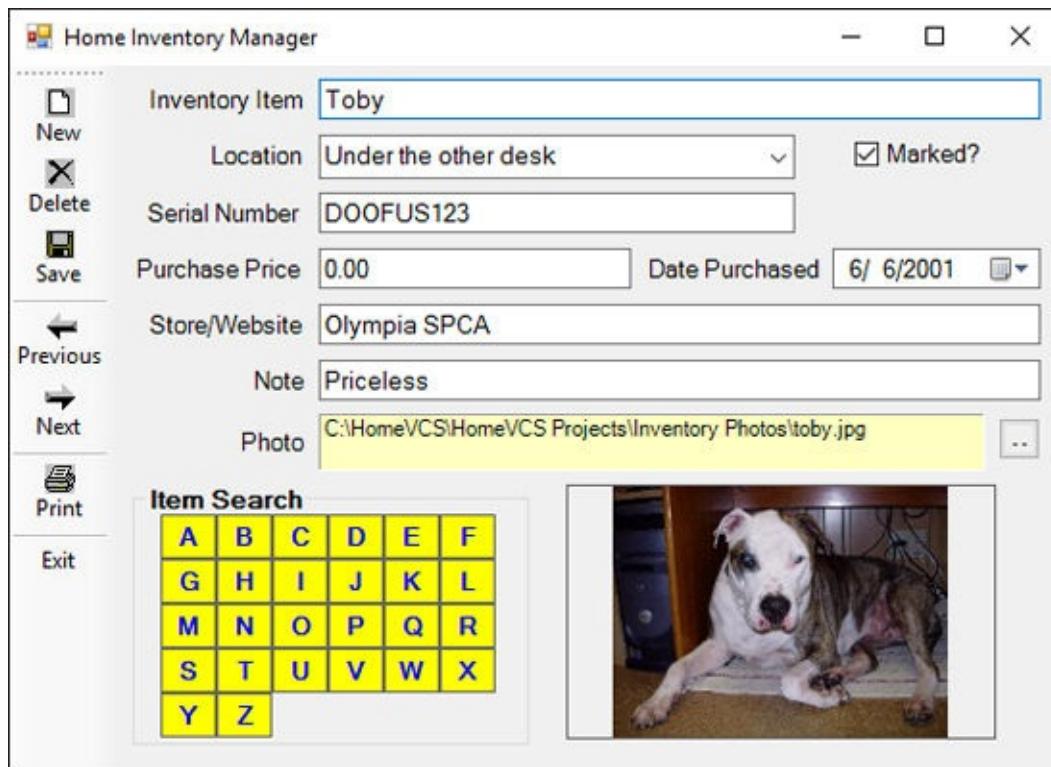
Another nice feature of the project is the ability to get a printed record of your inventory. Click the toolbar button marked **Print** (don't worry, nothing will print). You will see:



This window provides you with a preview of your printed inventory (achieved using the **print preview** dialog control). You can scroll through all pages and obtain a hard copy (click the **Printer** button) if you want. Close the **Print preview** window.

A primary task of the home inventory manager is to add, edit, save and delete inventory items. To add an item, you click the **Add** button in the toolbar. You then enter the necessary information and click the **Save** button. To edit an existing item, you first display the item to edit. Make the desired changes and click **Save**. To delete an item, you display the item, then click the **Delete** toolbar button. Let's try the editing features.

Navigate to one of the existing items in the sample file (use the **Previous** or **Next** buttons or try a search). I moved to **Toby**, my ever faithful dog:



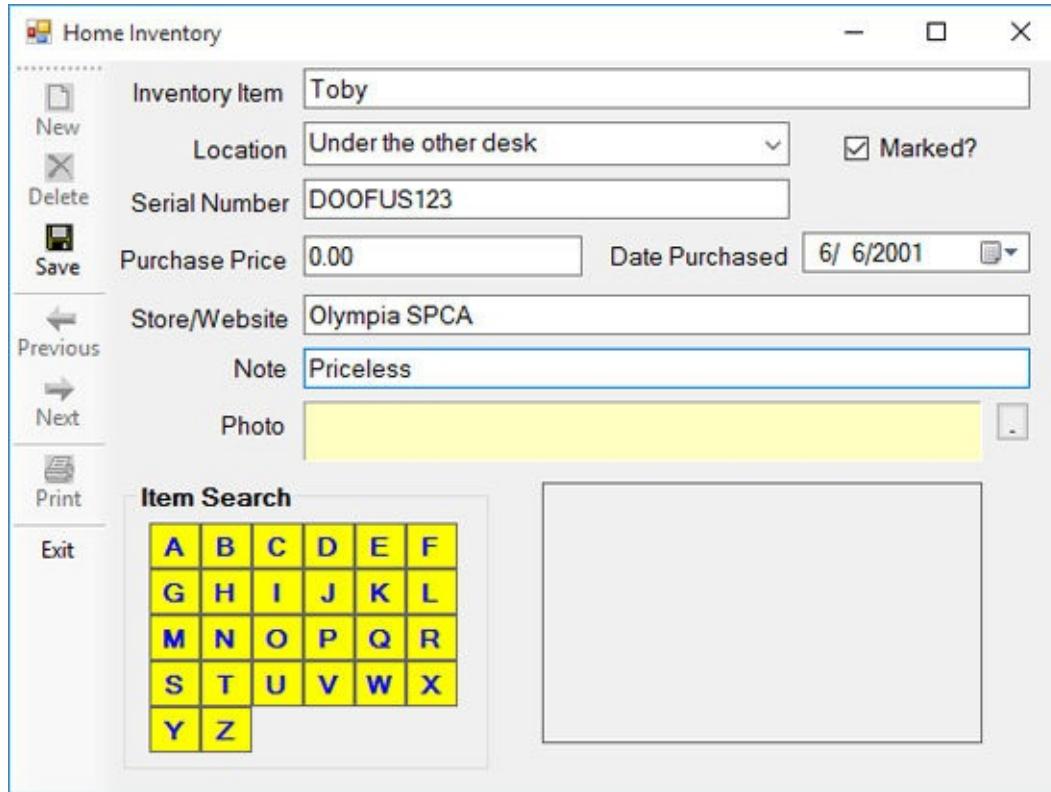
We'll delete this item, then rebuild it to demonstrate how to enter information. Click the **Delete** button – choose **Yes** when asked if you really want Toby to go away. The display will show the next item in the inventory. Click the **New** button to start a new item.

The blank inventory screen appears as:

At this point, you simply work your way down the form entering the desired information at the desired locations. When done, you click **Save** and the item is added to your inventory. We'll add Toby back to the file.

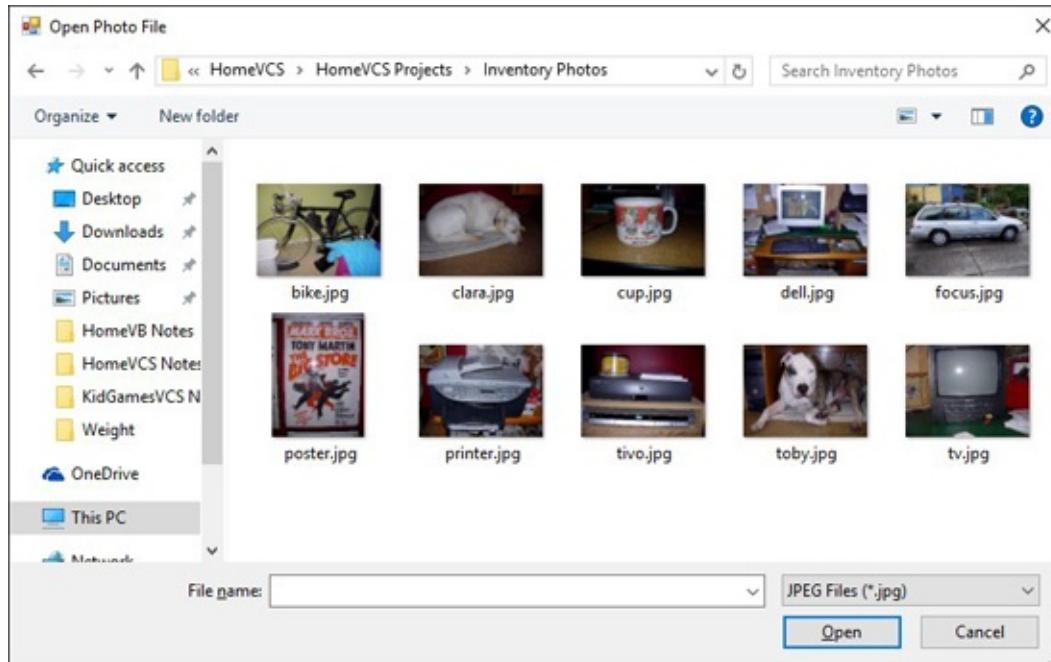
Under **Inventory Item**, type **Toby** and press <Enter>. This is the only required piece of information – all other entries are optional. For **Location**, click the drop-down arrow in the combo box. A list of choices is presented. Choose one of these items or type your own. If you type an entry that's not in the combo box, it will be added and saved for future items. Choose **Under the other desk** for Toby (he's always there). Put a check mark next to **Marked?**. Make up a **Serial Number** for Toby – I used **DOOFUS123**. We got Toby for free, so his **Purchase Price** is **0.00**. We got Toby on **June 6, 2001**. Under **Date Purchased**, click the drop-down arrow. On the calendar that appears, navigate to this date and click it. Under **Store/Website**, type **Olympia SPCA** (he's a pound puppy) and under **Note**, type **Priceless**.

At this point, the form should look like this:



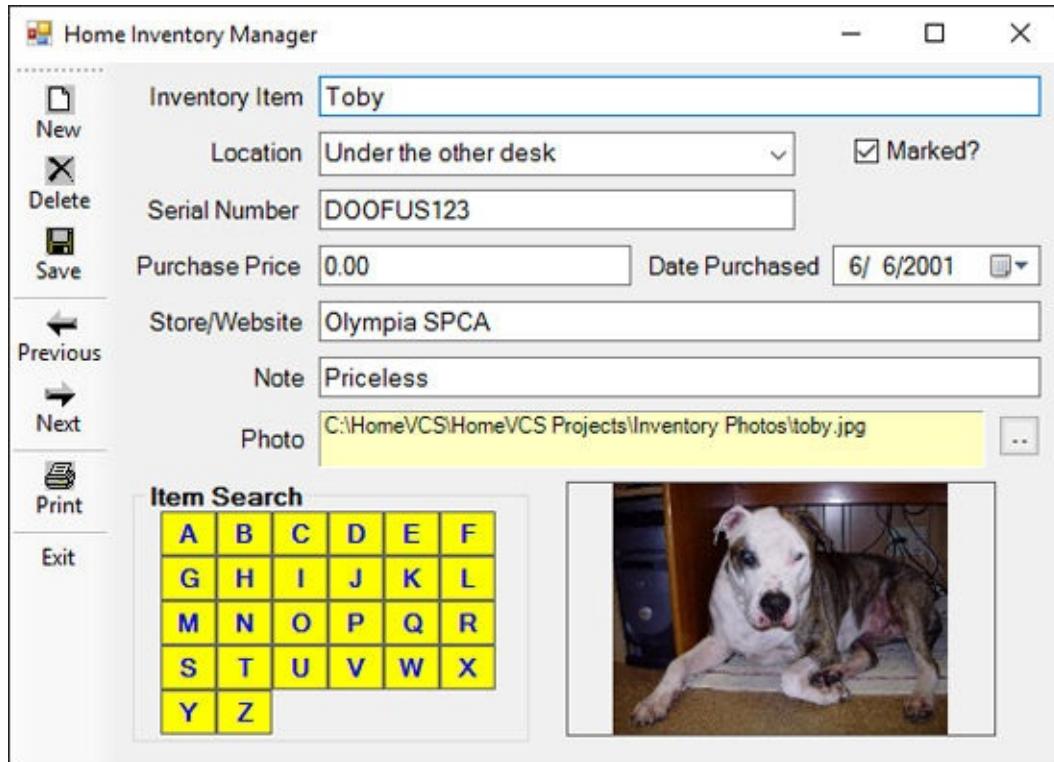
The last step is adding a photo.

Click the button with the ellipsis (...) next to the **Photo** label area. An open file dialog box will appear:



The photo can be any JPEG file (what a digital camera uses). You simply navigate to a photo location and click **Open**. The samples for these notes are in the **HomeVCS\HomeVCS Projects\Inventory Photos** folder. Move to that folder and select **toby.jpg** as shown. Click **Open** and the photo will appear.

The final **Toby** inventory item page looks like this:



Notice the photo and the file name listed under **Photo**. At this point, click **Save** and Toby is back in the list (properly sorted alphabetically).

That's the idea of the program. Fill in an entry page for each item in your inventory and click **Save**. Click **Exit** on the toolbar when done. Upon exiting the program, all your inventory items are saved to a file (the built-in file currently holding the sample entries). This same file is automatically opened when you rerun the program, so your items are always available for additions, changes and deletions.

We will now build this program in several stages. We discuss **form design**. We discuss the controls used to build the form and establish initial properties. We see how to add a toolbar to the project. And, we address **code design** in detail.

We introduce object-oriented programming (OOP) concepts to store the inventory data. We discuss how to read and write the inventory file, how to perform the various editing features, how to load a photo file, how to create the labels used in the search function, and how to print out the inventory. Four new controls are used in this project – the toolbar control (used to create the toolbar), the combo box control, and the print preview dialog control. We review the use of all but the print preview control before starting the project. The print preview control is discussed when we address printing from a Visual C# project.

# ToolStrip Control

**In Toolbox:**



**On Form (Default Properties):**



**Below Form (Default Properties):**



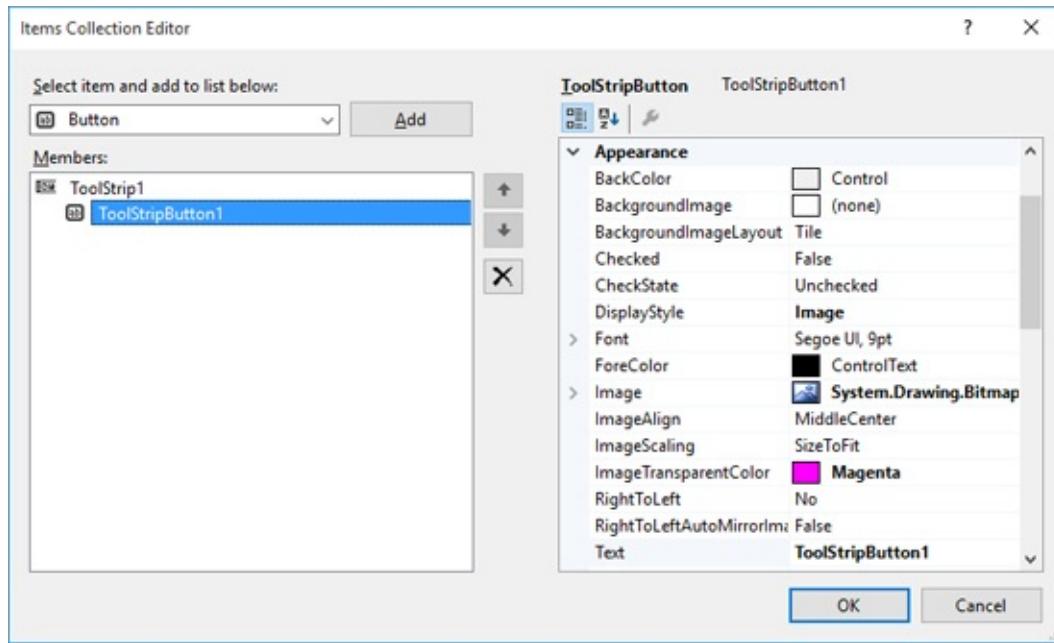
Almost all Windows applications these days use toolbars. A toolbar provides quick access to the most frequently used menu commands in an application. The **ToolStrip** control (also referred to as the **Toolbar** control) is a mini-application in itself. It provides everything you need to design and implement a toolbar into your application. Possible uses for this control include: provide a consistent interface between applications with matching toolbars, place commonly used functions in an easily-accessed space and provide an intuitive, graphical interface for your application.

## ToolStrip Properties:

<b>Name</b>	Gets or sets the name of the toolbar (toolbar) control (three letter prefix for label name is <b>tlb</b> ).
<b>BackColor</b>	Background color of toolbar.
<b>Items</b>	Gets the collection of controls assigned to the toolbar control.
<b>LayoutStyle</b>	Establishes whether toolbar is vertical or horizontal.
<b>Dock</b>	Establishes location of toolbar on form.

The primary property of concern is the **Items** collection. This establishes each item in the toolbar. Choosing the **Items** property in the Properties window and

clicking the ellipsis that appears will display the **Items Collection Editor**. With this editor, you can add, delete, insert and move items. We will look at adding just two types of items: **ToolStripButton** and **ToolStripSeparator** (used to separate tool bar buttons). To add a button, make sure **ToolStripButton** appears in the drop-down box and click the **Add** button. A name will be assigned to a button. After adding one button, the editor will look like this:



Add as many buttons as you like. You can change any property you desire in the **Properties** area.

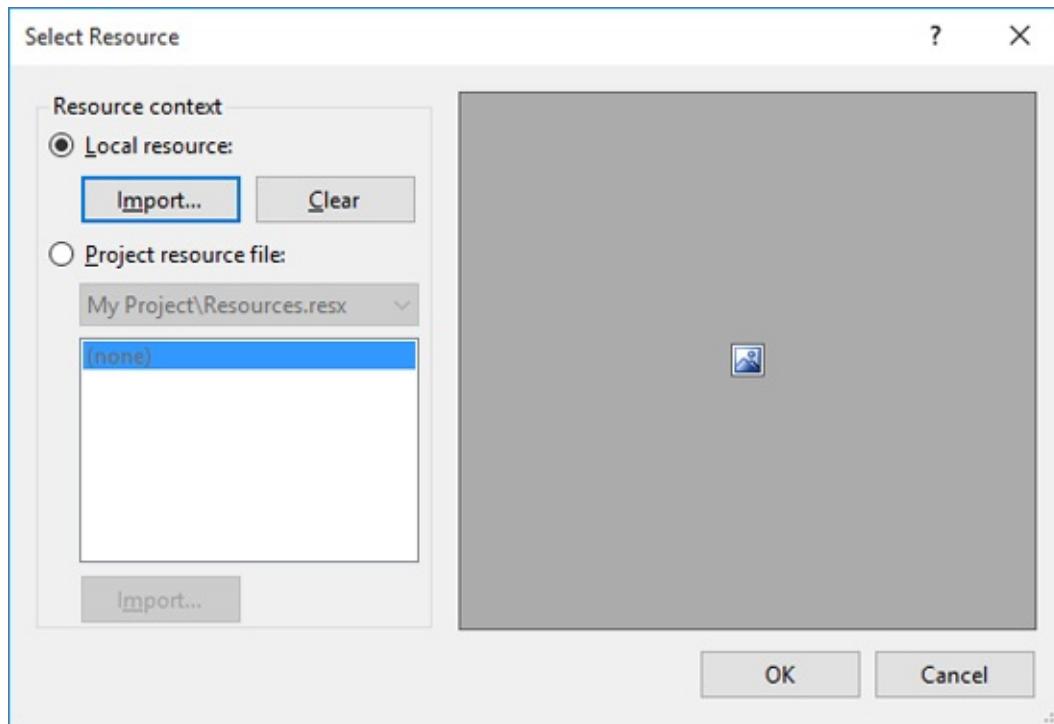
### ToolStripButton Properties:

<b>Name</b>	Gets or sets the name of the button (three letter prefix for control name is <b>tlb</b> ).
<b>DisplayStyle</b>	Sets whether image, text or both are displayed on button.
<b>Image</b>	Image to display on button.
<b>Text</b>	Caption information on the button, often blank.
<b>TextImageRelation</b>	Where text appears relative to image.
<b>ToolTipText</b>	Text to display in button tool tip.

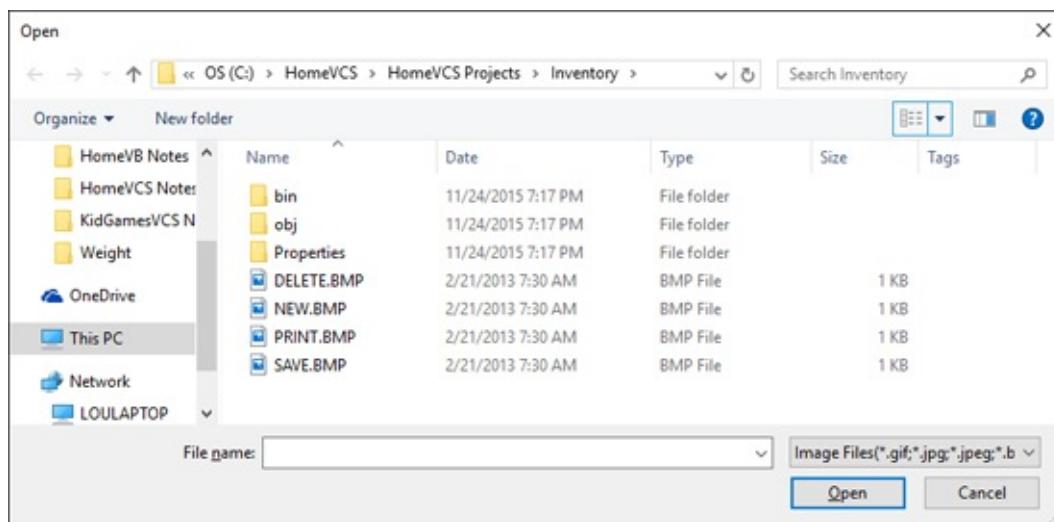
To add a separator, make sure **ToolStripSeparator** appears in drop-down box

and click **Add**. When done editing buttons, click **OK** to leave the Items Collection Editor.

Setting the **Image** property requires a few steps. First, click the ellipsis next to the **Image** property in the property window. This **Select Resource** window will appear:



Click **Import** to see a **Open** window display graphics files (if you want to see an **ico** file, you must change **Files of type** to **All Files**):



Navigate to the desired file and click **Open**. Then click **OK** in the **Select Resource** window.

After setting up the toolbar, you need to write code for the **Click** event for each toolbar button. This event is the same **Click** event we encounter for button controls.

Typical use of **ToolStrip** control:

- Set the **Name** property and desired location.
- Decide on image, text, and tooltip text for each button.
- Establish each button/separator using the **Items Collection Editor**.
- Write code for the each toolbar button's **Click** event.

# ComboBox Control

## In Toolbox:



## On Form (Default Properties):



In the previous chapter, we used the **ListBox** for displaying and selecting items in the weight control project. A related control is the **ComboBox** control. The **ComboBox** allows the selection of a single item from a list (usually viewed using a drop-down button). In this project, we use the control to choose inventory item location. We will allow the user to type in an alternate response, if a listed choice is not appropriate.

## ComboBox Properties:

ComboBox properties are nearly identical to those of the **ListBox**, with the deletion of the **SelectionMode** property and the addition of a **DropDownStyle** property.

<b>Name</b>	Gets or sets the name of the combo box (three letter prefix for combo box name is <b>cbo</b> ).
<b>BackColor</b>	Get or sets the combo box background color.
<b>DropDownStyle</b>	Specifies one of three combo box styles.
<b>Font</b>	Gets or sets font name, style, size.
<b>ForeColor</b>	Gets or sets color of text.
<b>Items</b>	Gets the Items object of the combo box.
<b>MaxDropDownItems</b>	Maximum number of items to show in dropdown portion.
<b>SelectedIndex</b>	Gets or sets the zero-based index of the currently selected item in list box portion.
<b>SelectedItem</b>	Gets or sets the currently selected item in the list

	box portion.
<b>SelectedText</b>	Gets or sets the text that is selected in the editable portion of combo box.
<b>Sorted</b>	Gets or sets a value indicating whether the items in list box portion are sorted alphabetically.
<b>Text</b>	String value displayed in combo box.

### ComboBox Events:

<b>KeyPress</b>	Occurs when a key is pressed while the combo box has focus.
<b>SelectedIndexChanged</b>	Occurs when the SelectedIndex property has changed.

The **Items** object for the list portion of the ComboBox control is identical to that of the ListBox control. **Item** is a zero-based array of the items in the list and **Count** is the number of items in the list. Hence, the first item in a combo box named **cboExample** is:

**cboExample.Items[0]**

The last item in the list is:

**cboExample.Items[cboExample.Items.Count – 1]**

The minus one is needed because of the zero-based array.

To add an item to a combo box, use the **Add** method, to delete an item, use the **Remove** or **RemoveAt** method and to clear a list box use the **Clear** method. For our example combo box, the respective commands are:

- Add Item:     **cboExample.Items.Add(ItemToAdd)**
- Delete Item:   **cboExample.Items.Remove(ItemToRemove)**  
**cboExample.Items.RemoveAt(IndexofItemToRemove)**
- Clear items:   **cboExample.Items.Clear**

Combo boxes normally list string data types, though other types are possible.

Note, when removing items, that indices for subsequent items in the list change following a removal.

The **DropDownStyle** property has three different values. The values and their description are:

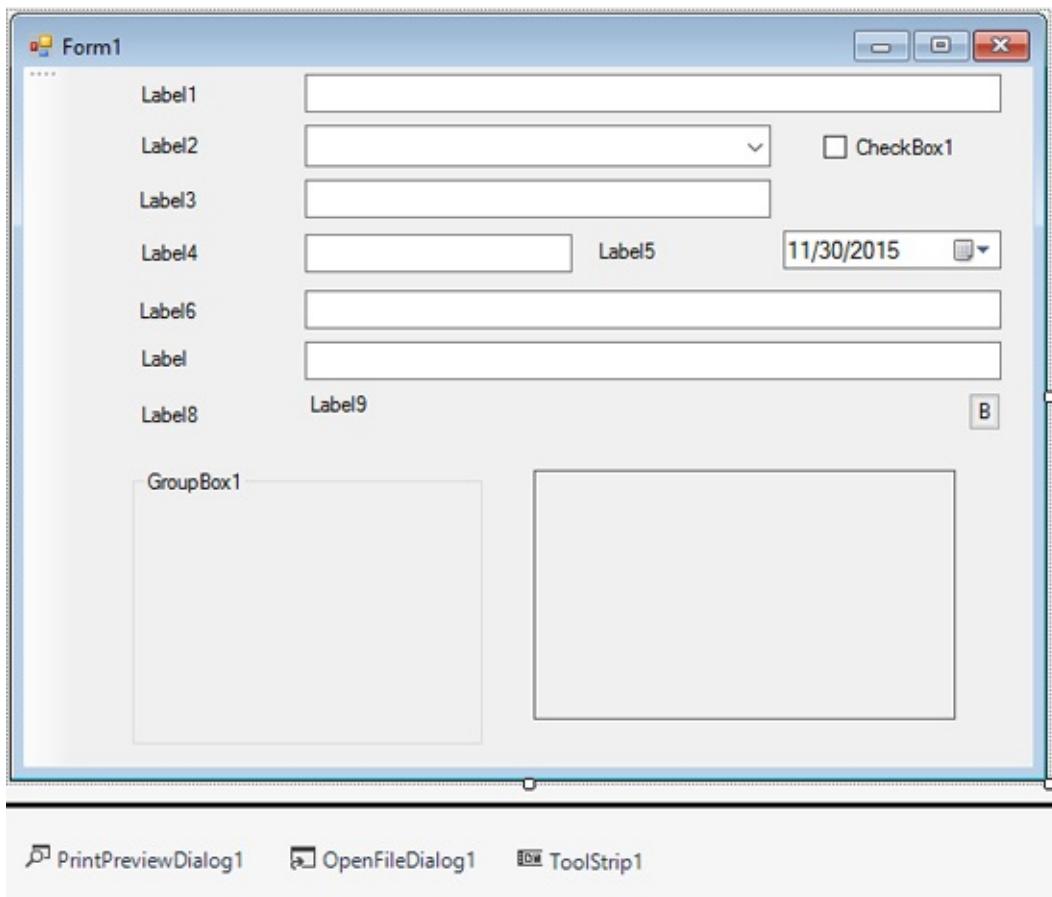
<b>Value</b>	<b>Description</b>
DropDown	Text portion is editable; drop-down list portion.
DropDownList	Text portion is not editable; drop-down list portion.
Simple	The text portion is editable. The list portion is always visible. With this value, you'll want to resize the control to set the list box portion height.

Typical use of **ComboBox** control:

- Set **Name** property, **DropDownStyle** property and populate **Items** object (usually in form **Load** method).
- Monitor **SelectedIndexChanged** event for individual selections.
- Read **Text** property to identify choice.

# Home Inventory Manager Form Design

We can begin building the home inventory manager project. Let's build the form. Start a new project in Visual C#. There are lots of controls. You need 9 label controls, 5 text box controls, a combo box control, a check box control, a date time picker, a small button, a group box and a picture box control. Add a tool strip (set **LayoutStyle** to **VerticalStackWithOverflow**, **Dock** to **Left**), an open file dialog and a print preview dialog control to the project. Resize and position controls until the form looks similar to this:



**label1** through **label8** are used for titling. **label9** holds the plot file name. The 'empty' controls are text boxes used to input item information. The combo box control is used to select location information. The check box indicates if an item is marked. The date time picker selects purchase date. The button (very small) is used to select the photo file. The photo is displayed in the picture box control. The group box control will hold labels used for searching. The tool strip is used

to edit items and navigate from one item to the next.

Set the control properties using the properties window:

**Form1** Form:

<b>Property Name</b>	<b>Property Value</b>
Name	frmInventory
BorderStyle	FixedSingle
StartPosition	CenterScreen
Text	Home Inventory Manager

**label1** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Inventory Item
Font Size	10

**textBox1** Text Box:

<b>Property Name</b>	<b>Property Value</b>
Name	txtItem
Font Size	10

**label2** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Inventory Item
Font Size	10

**comboBox1** Combo Box:

<b>Property Name</b>	<b>Property Value</b>
Name	cboLocation
Font Size	10
Sorted	True

**checkBox1** Check Box:

<b>Property Name</b>	<b>Property Value</b>
Name	chkMarked
Font Size	10

**label3** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Serial Number
Font Size	10

**textBox2** Text Box:

<b>Property Name</b>	<b>Property Value</b>
Name	txtSerialNumber
Font Size	10

**label4** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Purchase Price
Font Size	10

**textBox3** Text Box:

<b>Property Name</b>	<b>Property Value</b>
Name	txtPrice
Font Size	10

**label5** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Date Purchased
Font Size	10

**dateTimePicker1** Date Time Picker:

<b>Property Name</b>	<b>Property Value</b>
Name	dtpPurchase
Font Size	10
Format	Short

**label6** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Store/Website
Font Size	10

**textBox4** Text Box:

<b>Property Name</b>	<b>Property Value</b>
Name	txtPurchase
Font Size	10

**label7** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Note
Font Size	10

**textBox5** Text Box:

<b>Property Name</b>	<b>Property Value</b>
Name	txtNote
Font Size	10

**label8** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Photo
Font Size	10

**label9** Label:

<b>Property Name</b>	<b>Property Value</b>
Name	lblPhoto
AutoSize	False
Text	[Blank]
BackColor	LightYellow
BorderStyle	Fixed3D

**button1** Button:

<b>Property Name</b>	<b>Property Value</b>
Name	btnPhoto
Text	... (an ellipsis)
Font Size	10
Font Style	Bold

**groupBox1** Group Box:

<b>Property Name</b>	<b>Property Value</b>
Name	grpSearch
Text	Item Search
Font Size	10
Font Style	Bold

**pictureBox1** Picture Box:

<b>Property Name</b>	<b>Property Value</b>
Name	picPhoto
BorderStyle	FixedSingle
SizeMode	Zoom

**toolStrip1** Toolstrip:

<b>Property Name</b>	<b>Property Value</b>
Name	tlbInventory
LayoutStyle	VerticalStackWithOverflow
Dock	Left

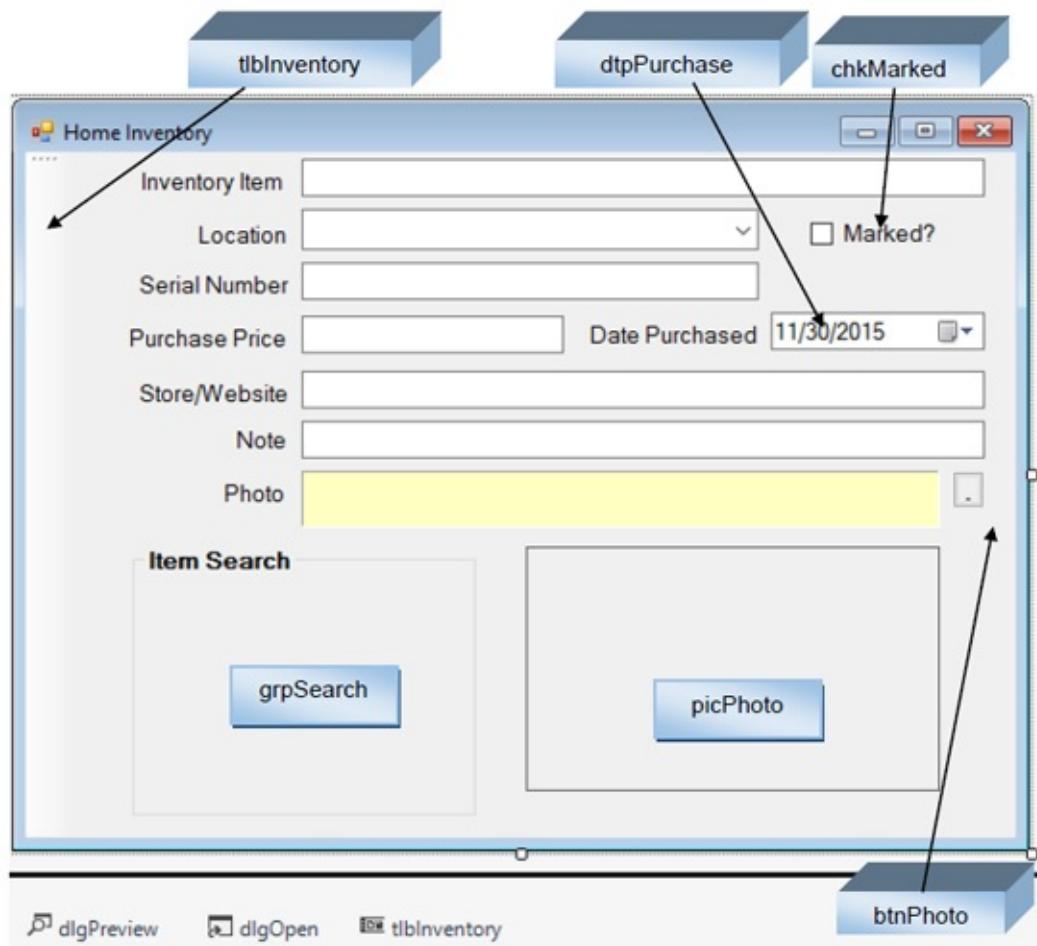
**openFileDialog1** Open File Dialog:

Property Name	Property Value
Name	dlgOpen
DefaultExt	jpg
Filter	JPEG Files (*.jpg) *.jpg
Title	Open Photo File

**printPreviewDialog1** Print Preview Dialog:

Property Name	Property Value
Name	dlgPreview

When done setting properties, my form looks like this:



We only label a few of the controls (we don't label the text box and label controls) here. Most should be obvious.

Before beginning the code for the project, let's look at designing the toolbar.

# Form Design – Toolbar

The toolbar (toolstrip) is used to edit the inventory items and navigate through them. It is also used to print the items and exit the program. The toolbar will have seven buttons: one to create a **new** item, one to **delete** an item, one to **save** an item, one to view the **previous** item, one to view the **next** item, one to **print** the inventory and one to **exit** the program. All but the last button will have an image.

Click on the **Items** property of the toolstrip control. The **Items Collection Editor** will appear. We use this editor to sequentially add six buttons and three separators to the toolbar. One separator will be between the editing features (new, delete, save) and the navigation features (previous, next). One will be between navigation features and the print button and one is separating the print function from the exit button. Sequentially add these nine elements:

## **toolStripButton1** Toolbar Button:

Name	tlbNew
DisplayStyle	ImageAndText
Image	NEW.BMP (in <b>HomeVCS\HomeVCSPublic\Inventory</b> folder)
Text	New
ToolTipText	Add New Item
TextImageRelation	ImageAboveText

## **toolStripButton2** Toolbar Button:

Name	tlbDelete
DisplayStyle	ImageAndText
Image	DELETE.BMP (in <b>HomeVCS\HomeVCSPublic\Inventory</b> folder)
Text	Delete
ToolTipText	Delete Current Item
TextImageRelation	ImageAboveText

**toolStripButton3** Toolbar Button:

Name	tlbSave
DisplayStyle	ImageAndText
Image	SAVE.BMP (in <b>HomeVCS\HomeVCSPublic\Inventory</b> folder)
Text	Save
ToolTipText	Save Current Item
TextImageRelation	ImageAboveText

**toolStripSeparator1** Toolbar Separator

**toolStripButton4** Toolbar Button:

Name	tlbPrevious
DisplayStyle	ImageAndText
Image	ARWLT.ICO (in <b>HomeVCS\HomeVCSPublic\Inventory</b> folder)
Text	Previous
ToolTipText	Display Previous Item
TextImageRelation	ImageAboveText

**toolStripButton5** Toolbar Button:

Name	tlbNext
DisplayStyle	ImageAndText
Image	ARWRT.BMP (in <b>HomeVCS\HomeVCSPublic\Inventory</b> folder)
Text	Next
ToolTipText	Display Next Item
TextImageRelation	ImageAboveText

**toolStripSeparator2** Toolbar Separator

**toolStripButton6** Toolbar Button:

Name	tlbPrint
DisplayStyle	ImageAndText

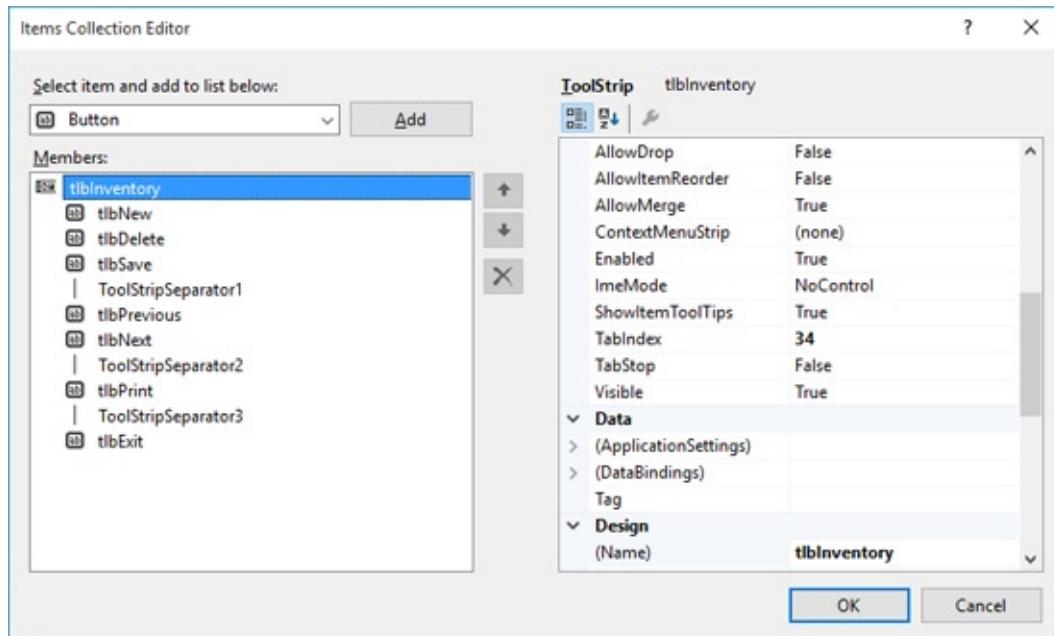
Image	PRINT.BMP (in <b>HomeVCS\HomeVCSProjects\Inventory</b> folder)
Text	Print
ToolTipText	Print Inventory List
TextImageRelation	ImageAboveText

**toolStripSeparator3** Toolbar Separator

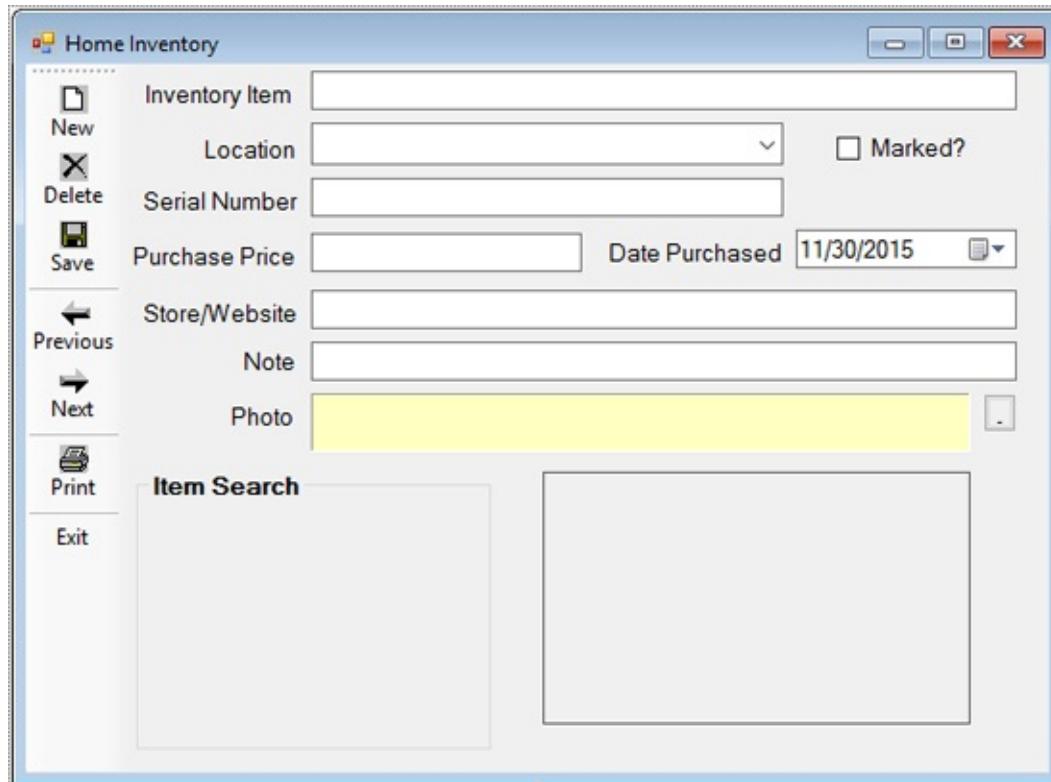
**toolStripButton7** Toolbar Button:

Name	tlbExit
DisplayStyle	Text
Text	Exit
ToolTipText	Exit Program

When done, the **Items Collection Editor** should look like this:



The finished form (including toolbar):



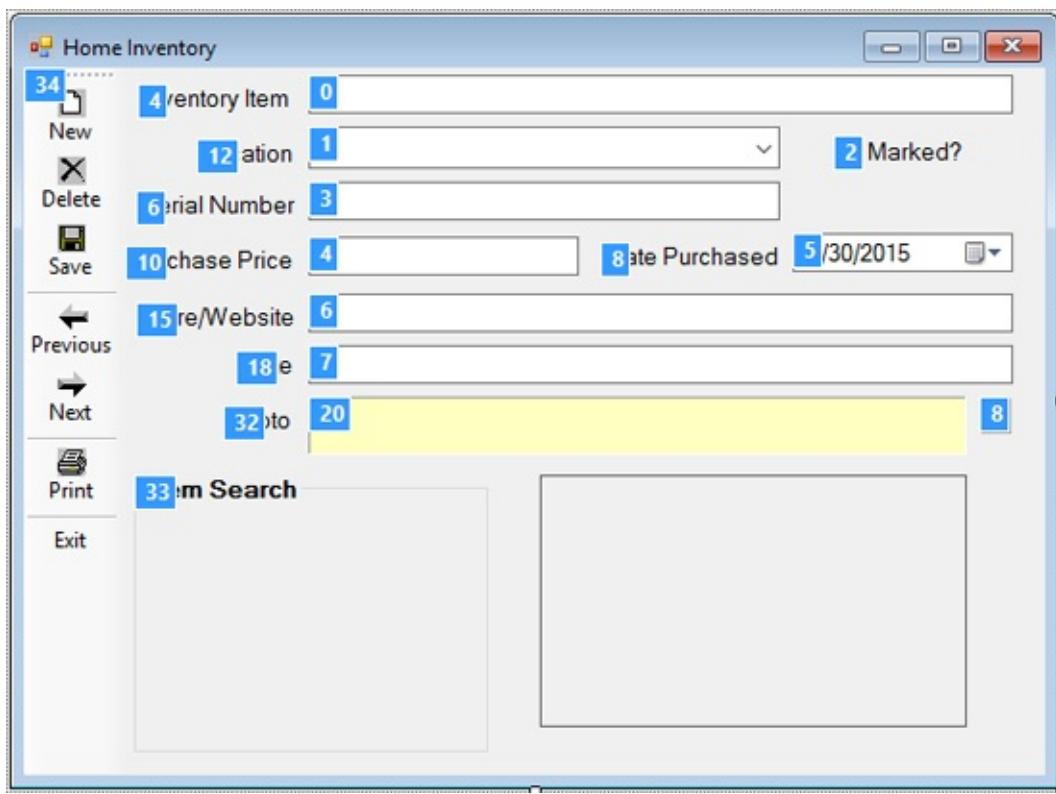
Notice the pictures and separators in the toolbar.

We now start writing the code. There are many steps. First, let's address proper ordering of the controls for input.

# Form Design – Tab Order and Focus

This project has many controls for user input. We want to make sure it's clear to the user just what information is needed and when. We want the input to 'flow' from the top of the form to the bottom.

First, let's set the tabs. With the form in view, select **View** from the menu and click **Tab Order**. Set the tabs as shown:



The tab sequence starts at the description text box (**txtItem**) and sequentially works down through all the input controls (text boxes, combo box, check box, date picker) to the note text box (**txtNote**), then the photo button (**btnPhoto**).

Save and run the project. Check that the tab order is as desired.

Another feature for the input is that whenever the user presses <Enter> after entering a value (in the combo box or text boxes) or clicks a date in the date selector, the control focus should move to the next control. This feature is

implemented in the **KeyPress** events for the combo box control and the five text boxes and the **CloseUp** event for the date control. Let's start at the top of the form and work our way down, in proper tab order. First, the **txtItem** **KeyPress** event method:

```
private void txtItem_KeyPress(object sender, KeyPressEventArgs e)
{
    if ((int)e.KeyChar == 13)
        cboLocation.Focus();
}
```

This method moves focus to **cboLocation**, which has this **KeyPress** event method:

```
private void cboLocation_KeyPress(object sender, KeyPressEventArgs e)
{
    if ((int)e.KeyChar == 13)
        chkMarked.Focus();
}
```

This method moves focus to the check box control. There is no equivalent to a **KeyPress** event for the check box, so focus at this point has to manually shift to the serial number text box (**txtSerialNumber**), which has a **KeyPress** event method:

```
private void txtSerialNumber_KeyPress(object sender,
KeyEventArgs e)
{
    if ((int)e.KeyChar == 13)
        txtPrice.Focus();
}
```

This method moves focus to **txtPrice**, which has this **KeyPress** event method:

```
private void txtPrice_KeyPress(object sender, KeyPressEventArgs e)
```

```
{  
    if ((int)e.KeyChar == 13)  
        dtpPurchase.Focus();  
}
```

This method moves focus to **dtpPurchase**, which has this **CloseUp** event method (called when the user selects a date from the drop-down calendar):

```
private void dtpPurchase_CloseUp(object sender, EventArgs e)  
{  
    txtPurchase.Focus();  
}
```

This method moves focus to **txtPurchase**, which has this **KeyPress** event method:

```
private void txtPurchase_KeyPress(object sender, KeyPressEventArgs e)  
{  
    if ((int)e.KeyChar == 13)  
        txtNote.Focus();  
}
```

This method moves focus to **txtNote**, which has this **KeyPress** event method:

```
private void txtNote_KeyPress(object sender, KeyPressEventArgs e)  
{  
    if ((int)e.KeyChar == 13)  
        btnPhoto.Focus();  
}
```

Lastly, this method moves focus to **btnPhoto**, which is clicked to load a photo onto the form.

Save and run the project. Press <Enter> in each control (except the check box) to make sure the focus transfers properly. You will see this bit of work will be

well worth it when you start entering information onto the form. We'll start writing code to process entries soon. First, let's look at how we'll structure all the information used to describe an inventory item.

# Introduction to Object-Oriented Programming

Each inventory item requires nine individual pieces of information. Each item and the data type represented are:

- Description (**string** type)
- Location (**string** type)
- Marked indicator (**bool** type)
- Serial number (**string** type)
- Purchase Price (**string** type)
- Purchase Date (**DateTime** type)
- Purchase Location (**string** type)
- Note (**string** type)
- Photo file (**string** type)

One way to store all this information is to use nine different arrays, one for each quantity, each element of the array representing a single item in the inventory. Using arrays would be “doable,” but messy. It would be especially messy to write code for swapping inventory items (needed to make sure items remain in alphabetical order). Each swap would require swapping nine different array elements. And, what if we later want to add more information to an inventory item? We would need to remember everywhere these arrays were referenced in code to make the needed changes. There must be a better way to structure all this information. And there is.

We say Visual C# is an **object-oriented** language. At this point in this course, we have used many of the built-in objects included with Visual C#. We have used button objects, text box objects, label objects and many other controls. We have used graphics objects, font objects, pen objects, and brush objects. Having used these objects, we are familiar with such concepts as **declaring** an object, **constructing** an object and using an object’s **properties** and **methods**.

We have seen that objects are just things that have attributes (properties) with

possible actions (methods). We'll use the idea here to create our own “inventory item” objects. Our objects will only have properties (specified above) and no methods. You will see how creating such objects saves us lots of work.

Before getting started, you may be asking the question “If Visual C# is an object-oriented language, why have we waited so long to start talking about using our own objects?” And, that’s a good question. Many books on Visual C# dive right into building objects. We feel it’s best to see objects and use objects before trying to create your own. Visual C# is a great language for doing this. The wealth of existing, built-in objects helps you learn about OOP before needing to build your own.

Now, let’s review some of the vocabulary of object-oriented programming. These are terms you’ve seen before in working with the built-in objects of Visual C#. A **class** provides a general description of an **object**. All objects are created from this class description. The first step in creating an object is adding a class to a Visual C# project. Every application we have built in this course is a class itself. Note the top line of every application has the keyword **class**.

The **class** provides a framework for describing three primary components:

- **Properties** – attributes describing the objects
- **Constructors** – methods that initialize the object
- **Methods** – procedures describing things an object can do

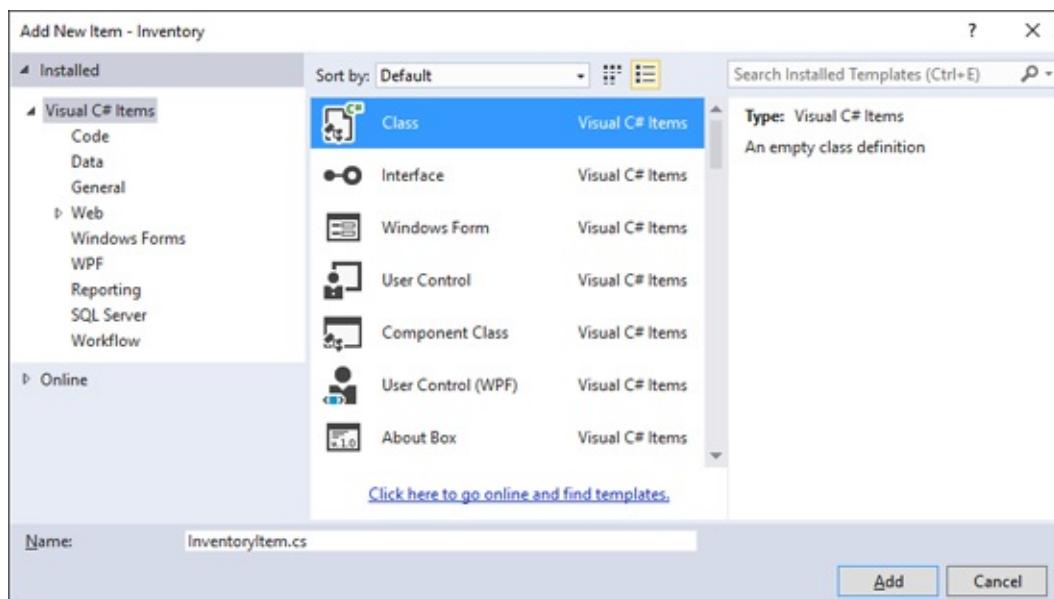
Once a class is defined, an object can be created or **instantiated** from the class. This simply means we use the class description to create a copy of the object we can work with. Once the instance is created, we **construct** the finished object for our use.

One last important term to define, related to OOP, is **inheritance**. This is a capability that allows one object to ‘borrow’ properties and methods from another object. This prevents the classic ‘reinventing the wheel’ situation. Inheritance is one of the most powerful features of OOP. In this chapter, we will only look at using a simple object, with just some properties. In the next chapter, we will look at adding methods and using inheritance in a project.

# Code Design – InventoryItem Class

The first step in creating our own object is to define the class from which the object will be created. This step (and all following steps) is best illustrated by example. And our example will be our inventory item object. We will be creating **InventoryItem** objects that have nine properties, one for each previously-listed piece of information input on the form.

We need to add a class to our project to allow the definition of our **InventoryItem** objects. We could add the class in the existing form file. However, doing so would defeat a primary advantage of objects, that being reuse. Hence, we will create a separate file to hold our class. To do this, go to **Solution Explorer** for your project, right-click the project name (**Inventory**), choose **Add**, then **Class**. The following window will appear:



Type **InventoryItem** in the **Name** box and make sure the **Class** template is selected. Click **Open** to open the newly created class file.

A file named **InventoryItem.cs** will now be seen in the Solution Explorer window and that file will open. Inside the **namespace Inventory** brackets will be these lines in this file:

```
public class InventoryItem
{
    public InventoryItem()
    {
        }
    }
}
```

All our code (to define properties, constructors and methods) will be written in the **public class InventoryItem** code block, following the default constructor (**public InventoryItem**) already in place.

Add nine property declarations so the file looks like this:

```
public class InventoryItem
{
    public InventoryItem()
    {
        }

    public string Description;
    public string Location;
    public bool Marked;
    public string SerialNumber;
    public string PurchasePrice;
    public DateTime PurchaseDate;
    public string PurchaseLocation;
    public string Note;
    public string PhotoFile;
}
```

You should see how each property relates to the information on the form. The keyword **public** indicates the variable is available to any form.

To declare an **InventoryItem** object named **myItem** (in our inventory application) use this line of code:

```
InventoryItem myItem;
```

As soon as begin typing, the Intellisense feature pops up suggesting possible object types. **InventoryItem** is one of the selections! This is due to the existence of the **InventoryItem** class in the project.

To construct this object, use this line of code:

```
myItem = new InventoryItem();
```

This line just says “give me a new inventory item.” Our **InventoryItem** object is now complete, ready for use. This uses the **default** constructor automatically included with every class. The default constructor simply creates an object with no defined properties.

Once we have created an object, we can refer to properties using the usual notation:

```
objectName.PropertyName
```

So, to set the **Description** property of our example inventory item, we would use:

```
myItem.Description = "This is my inventory item";
```

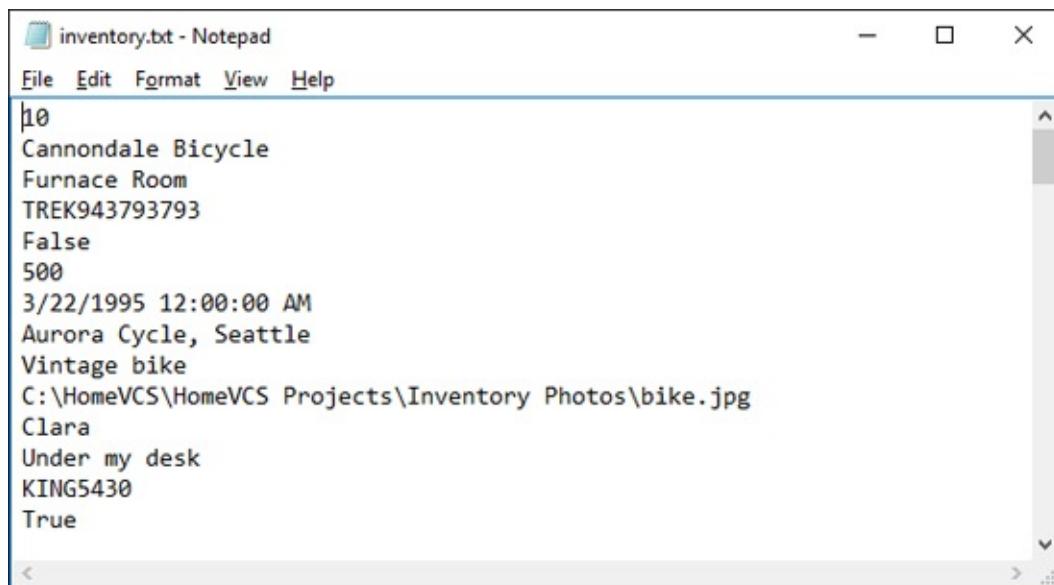
You will see that when typing the properties, the Intellisense feature again recognizes the existence of the **InventoryItem** class, providing a drop-down list of available properties.

Let’s see how to use our **InventoryItem** class in the home inventory manager project. In this project, we will use an array of inventory items to keep track of things.

# Code Design – Inventory File Input

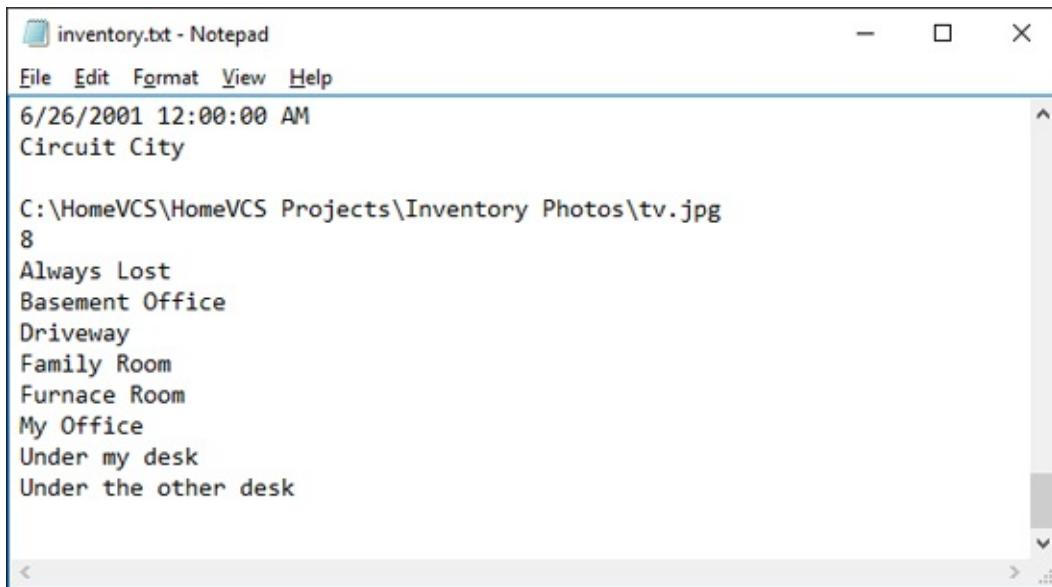
We can now use the **InventoryItem** class to define how we will save inventory information in our project. First, we look at how to input that information from a file. We have chosen to store the inventory information in a built-in, “hard-wired” file. The program looks for a file named **inventory.txt** in the project’s **Bin\Debug** folder. If the file can’t be found, the program begins with an empty inventory and, once items are added, these items are written to a new copy of **inventory.txt**.

The file (**inventory.txt**) keeps track of each item in the inventory. The file also stores the items in the combo box used to specify item location. In later code, we will look at how to modify these items, so new ones are saved. The **inventory.txt** file is sequential and stores one piece of information on each line. The first line is the number of inventory items in the file. After this line are 9 lines for each item in the inventory (each line saving one property for the **InventoryItem** object). After each item is described in the file is a line containing the number of items in the combo box control ( **cboLocation**). Following this line are the corresponding combo box items. In the **HomeVCS\HomeVCS Projects\Inventory** folder is the sample provided with these notes (seen in the project preview). Open this file in a text editor and you will see:



You see this file has 10 entries and there will be 9 lines per entry. You can see the first entry (**Cannondale Bicycle**) and the beginning of the second (**Clara**).

Scroll down to the bottom to see:



The screenshot shows a Windows Notepad window titled "inventory.txt - Notepad". The menu bar includes File, Edit, Format, View, and Help. The content of the file is as follows:

```
6/26/2001 12:00:00 AM
Circuit City

C:\HomeVCS\HomeVCS Projects\Inventory Photos\tv.jpg
8
Always Lost
Basement Office
Driveway
Family Room
Furnace Room
My Office
Under my desk
Under the other desk
```

After the last entry are the eight (8) items used in the combo box control. Let's write the code to read this file.

When the home inventory manager project begins, the program should read in the **inventory.txt** file to store the inventory item information. Here's where we'll use the **InventoryItem** object. The steps are:

- Open **inventory.txt** for input.
- Read in the number of entries.
- For each entry in the file:
  - Create a new **InventoryItem** object.
  - Read in the nine object properties.
- When done reading inventory items, read in the number of items in combo box control.
- Read in combo box items.
- Close file.

Make sure your are now working with the **Inventory.cs** file. We will use the

**StreamReader** object to read the file, so add this using statement at the top of the code window:

```
using System.IO;
```

The code associated with the above steps goes in the **frmInventory Load** method so it is executed when the program first begins. Define three form level variables:

```
const int maximumEntries = 300;  
int numberEntries;  
InventoryItem[] myInventory = new InventoryItem[maximumEntries];
```

**maximumEntries** is the maximum number of inventory items allowed, **numberEntries** is the number of entries in our inventory and **myInventory** is a 0-based array of **InventoryItem** objects used to store the information. With these variables, the code to open and read the **inventory.txt** file is:

```
private void frmInventory_Load(object sender, EventArgs e)  
{  
    int n;  
    // open file for entries  
    try  
    {  
        StreamReader inputFile = new  
        StreamReader(Application.StartupPath + "\\inventory.txt");  
        numberEntries =  
        Convert.ToInt32(inputFile.ReadLine());  
        if (numberEntries != 0)  
        {  
            for (int i = 0; i < numberEntries; i++)  
            {  
                myInventory[i] = new InventoryItem();  
                myInventory[i].Description = inputFile.ReadLine();  
                myInventory[i].Location = inputFile.ReadLine();  
            }  
        }  
    }  
}
```

```

        myInventory[i].SerialNumber = inputFile.ReadLine();
        myInventory[i].Marked =
Convert.ToBoolean(inputFile.ReadLine());
        myInventory[i].PurchasePrice = inputFile.ReadLine();
        myInventory[i].PurchaseDate =
Convert.ToDateTime(inputFile.ReadLine());
        myInventory[i].PurchaseLocation = inputFile.ReadLine();
        myInventory[i].Note = inputFile.ReadLine();
        myInventory[i].PhotoFile = inputFile.ReadLine();
    }
}

// read in combo box elements
n = Convert.ToInt32(inputFile.ReadLine());
if (n != 0)
{
    for (int i = 0; i < n; i++)
    {
        cboLocation.Items.Add(inputFile.ReadLine());
    }
}
inputFile.Close();
}

catch
{
    numberEntries = 0;
}
if (numberEntries == 0)
{
    tlbNew.Enabled = false;
    tlbDelete.Enabled = false;
    tlbNext.Enabled = false;
    tlbPrevious.Enabled = false;
}

```

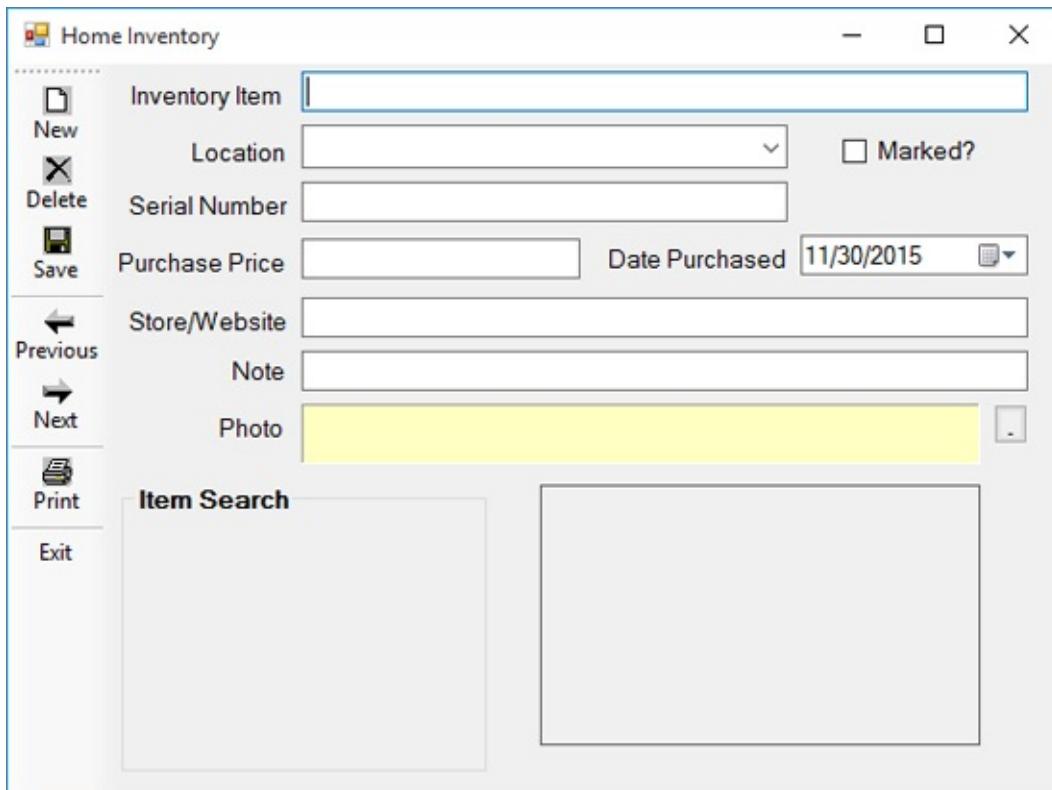
```

    tlbPrint.Enabled = false;
}
}

```

Let's take a look at this code. All the code is in a **try/catch** structure in case the input file cannot be opened. If the file is successfully opened, we read **numberEntries**, then read each subsequent entry, creating a new **InventoryItem** object for each entry. Once the inventory items have been input, the combo box items are read in. If **numberEntries** is zero when done (meaning the file couldn't be opened or truly had zero elements), we set the toolbar buttons so only a new item can be entered. Add this method to your project. Notice when typing the input lines, the properties associated with the **InventoryItem** objects appear as selections in the Intellisense window.

Let's try this code. Copy the sample **inventory.txt** file into your project's **Bin\Debug** folder. Save and run the project. If the file opens and reads successfully, you should see a blank form with all toolbar buttons enabled (try clicking the drop-down in the combo box to see the items added):



If only the **Save** button is enabled, there was an error in reading the file. If this occurs, make sure the file is in the correct folder and double-check the code.

# Code Design – Viewing Inventory Item

We can now read in the input file, but it's not very satisfying not being able to see the results. We remedy that now by writing code to display the properties of an inventory item (including the photo). The code simply sets the correct form control with the corresponding property.

We will use a general method **ShowEntry** to display the properties of a single inventory item. The method will have an **int** argument, specifying the entry number (from **1** to **numberEntries**) to display. The method is:

```
private void ShowEntry(int j)
{
    // display entry j (1 to numberEntries)
    txtItem.Text = myInventory[j - 1].Description;
    cboLocation.Text = myInventory[j - 1].Location;
    chkMarked.Checked = myInventory[j - 1].Marked;
    txtSerialNumber.Text = myInventory[j - 1].SerialNumber;
    txtPrice.Text = myInventory[j - 1].PurchasePrice;
    dtpPurchase.Value = myInventory[j - 1].PurchaseDate;
    txtPurchase.Text = myInventory[j - 1].PurchaseLocation;
    txtNote.Text = myInventory[j - 1].Note;
    ShowPhoto(myInventory[j - 1].PhotoFile);
    txtItem.Focus();
}
```

This code simply transfers the properties of the **myInventory InventoryItem** object into the appropriate controls.

To show a photo, we use another general method **ShowPhoto** which uses error trapping to insure the program doesn't stop if the file can't be opened (the **string** argument is the file name):

```
private void ShowPhoto(string photoFile)
```

```

{
    if (photoFile != "")
    {
        try
        {
            lblPhoto.Text = photoFile;
            picPhoto.Image = Image.FromFile(photoFile);
        }
        catch
        {
            lblPhoto.Text = "";
            picPhoto.Image = null;
        }
    }
    else
    {
        lblPhoto.Text = "";
        picPhoto.Image = null;
    }
}

```

Place both of these methods (**ShowEntry** and **ShowPhoto**) in your project.

We need to modify the **frmInventory Load** method to call the display routine. First, add another form level variable declaration:

```
int currentEntry;
```

**currentEntry** will always point to the current entry in the inventory file (going from **1** to **numberEntries**). The modified **Load** method that uses this variable to display the entry is (changes are shaded, much unmodified code is not shown):

```
private void frmInventory_Load(object sender, EventArgs e)
{

```

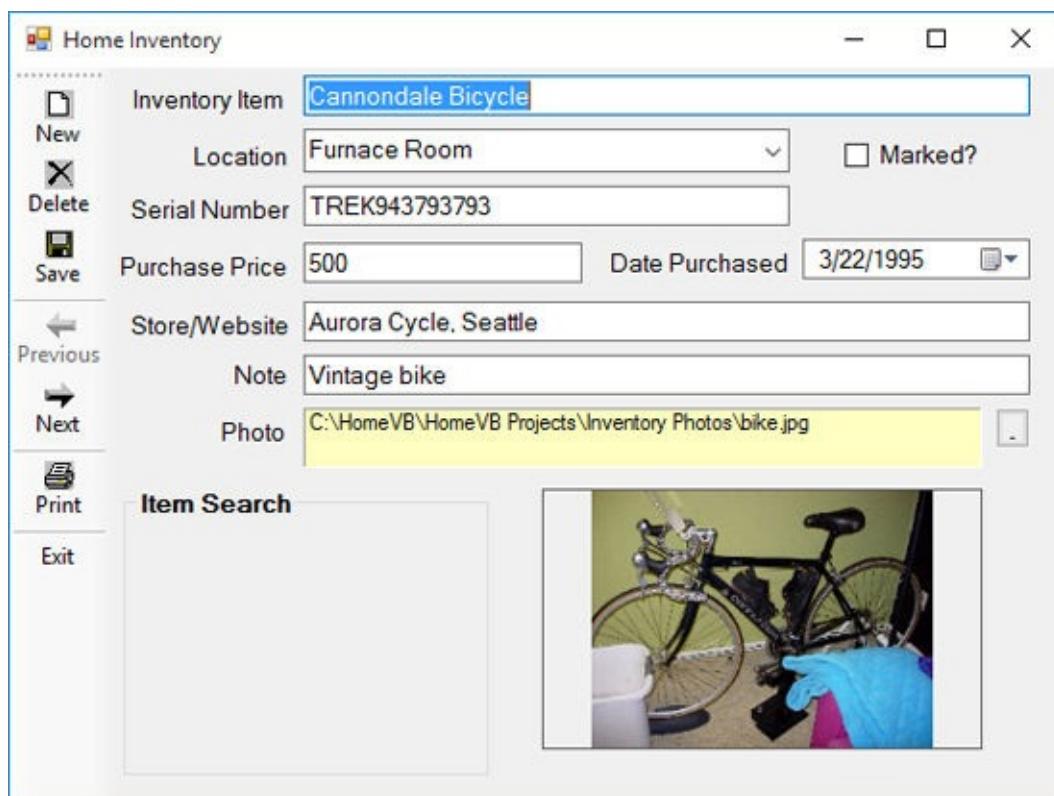
```
int n;  
// open file for entries  
try  
{  
    StreamReader inputFile = new  
StreamReader(Application.StartupPath + "\\inventory.txt");  
    .  
    .  
    inputFile.Close();  
    currentEntry = 1;  
    ShowEntry(currentEntry);  
}  
catch  
{  
    numberEntries = 0;  
    currentEntry = 0;  
}  
if (numberEntries == 0)  
{  
    tlbNew.Enabled = false;  
    tlbDelete.Enabled = false;  
    tlbNext.Enabled = false;  
    tlbPrevious.Enabled = false;  
    tlbPrint.Enabled = false;  
}  
}
```

Make the noted changes.

Before trying the project, a word about photo location. The example file assumes the inventory photos are located in a folder named **c:\HomeVCS\HomeVCS Projects\Inventory Photos**. Your copy of the photos may or may not be in such a folder, depending on where you installed your version of these notes and

projects. If they are in such a folder, great! If not, you have a few choices: (1) create such a folder and copy the photos to that folder, (2) hand edit the **inventory.txt** file to change the file names to your particular folder, (3) ignore the location and let the program run, knowing the photos won't display. You choose. Once the program is fully functional, you can load each picture individually from folders on your computer. Then, when the data file is saved, those locations are saved correctly for your machine.

Save and run the project. You should now see the first item in the inventory displayed (including photo, assuming you solved any "photo location" problem you may have had):



At this point, we would like to be able to move to the next item, or move backward. Let's write the code to do that.

First, we modify the **ShowEntry** method to establish proper **Enabled** properties for the two toolbar buttons (**tlbPrevious** and **tlbNext**) used to move among the inventory items. We set these properties based on whether we're at the beginning, at the end or in the middle of the item list (changes are shaded):

```

private void ShowEntry(int j)
{
    // display entry j (1 to numberEntries)
    txtItem.Text = myInventory[j - 1].Description;
    cboLocation.Text = myInventory[j - 1].Location;
    chkMarked.Checked = myInventory[j - 1].Marked;
    txtSerialNumber.Text = myInventory[j - 1].SerialNumber;
    txtPrice.Text = myInventory[j - 1].PurchasePrice;
    dtpPurchase.Value = myInventory[j - 1].PurchaseDate;
    txtPurchase.Text = myInventory[j - 1].PurchaseLocation;
    txtNote.Text = myInventory[j - 1].Note;
    ShowPhoto(myInventory[j - 1].PhotoFile);

    tlbNext.Enabled = true;
    tlbPrevious.Enabled = true;
    if (j == 1)
        tlbPrevious.Enabled = false;
    if (j == numberEntries)
        tlbNext.Enabled = false;

    txtItem.Focus();
}

```

Make these changes.

Now, we need code for the **Click** methods on the **tlbPrevious** and **tlbNext** buttons. In each case, we adjust **currentEntry** in the proper direction and display the item. The methods are:

```

private void tlbPrevious_Click(object sender, EventArgs e)
{
    currentEntry--;
    ShowEntry(currentEntry);
}

```

```
private void tlbNext_Click(object sender, EventArgs e)
{
    currentEntry++;
    ShowEntry(currentEntry);
}
```

Add these methods to your project.

And, while we're at it, add the **tlbExit Click** event method to your project:

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

Again, save and run the project. You should now be able to view all 10 items in the sample file by using the **Previous** and **Next** buttons in the toolbar. Give it a try. Try the **Exit** button.

# Code Design – Inventory File Output

If inventory entries are edited or new items added (we'll see how to do this next), we want to save all entries back to the **inventory.txt** file. We do this output in the **FormClosing** event method. That method is essentially the same as the code in the **Load** method with the **ReadLine** lines replaced by **WriteLine** statements:

```
private void frmInventory_FormClosing(object sender,  
FormClosingEventArgs e)  
{  
    // write entries back to file  
    StreamWriter outputFile = new  
    StreamWriter(Application.StartupPath + "\\inventory.txt");  
    outputFile.WriteLine(numberEntries);  
    if (numberEntries != 0)  
    {  
        for (int i = 0; i < numberEntries; i++)  
        {  
  
            outputFile.WriteLine(myInventory[i].Description);  
            outputFile.WriteLine(myInventory[i].Location);  
  
            outputFile.WriteLine(myInventory[i].SerialNumber);  
            outputFile.WriteLine(myInventory[i].Marked);  
  
            outputFile.WriteLine(myInventory[i].PurchasePrice);  
            outputFile.WriteLine(myInventory[i].PurchaseDate);  
  
            outputFile.WriteLine(myInventory[i].PurchaseLocation);  
            outputFile.WriteLine(myInventory[i].Note);  
  
            outputFile.WriteLine(myInventory[i].PhotoFile);  
        }  
    }  
}
```

```
        }

    }

// write combo box entries
outputFile.WriteLine(cboLocation.Items.Count);
if (cboLocation.Items.Count != 0)
{
    for (int i = 0; i < cboLocation.Items.Count; i++)
        outputFile.WriteLine(cboLocation.Items[i]);
}
outputFile.Close();
}
```

Add this method to your project. Note the code to write the combo box items back to file also.

Save and run the project. The input file will be read and the items displayed. Stop the project. The file will be written back to disk. Currently, the same file will be written back. This is because we have no editing capability.

# Code Design – Input Validation

Before adding editing capability, we need to address a few input validation issues. The first regards the **PurchasePrice** property. This must be a numeric input, with at most one decimal point. The usual **KeyPress** event is used to insure validity. The **txtPrice KeyPress** event is (replaces existing method):

```
private void txtPrice_KeyPress(object sender, KeyPressEventArgs e)
{
    // only allow numbers, a single decimal point, backspace or enter
    if ((e.KeyChar >= '0' && e.KeyChar <= '9') || (int) e.KeyChar == 8)
        e.Handled = false;
    else if ((int) e.KeyChar == 13)
    {
        dtpPurchase.Focus();
        e.Handled = false;
    }
    else if (e.KeyChar == '.')
    {
        if (txtPrice.Text.IndexOf(".") == -1)
            e.Handled = false;
        else
            e.Handled = true;
    }
    else
        e.Handled = true;
}
```

Replace the current **txtPrice KeyPress** method with this new method.

Next, we address the **Location** property. The value for this property is obtained from the **cboLocation** control. Two selection possibilities exist: (1) choose from

an existing location, (2) type in a new location. If a new location is typed in, it is added to the list box portion of the combo box, so it can be saved for future edits (in the **inventory.txt** file). So, we need code to check a typed entry versus the existing list to see if the new entry needs to be added to the list box. The code to do this goes in the **cboLocation Leave** event method (invoked when a user leaves the combo box after making a selection or typing an entry):

```
private void cboLocation_Leave(object sender, EventArgs e)
{
    if (cboLocation.Text.Trim() == "")
        return;
    if (cboLocation.Items.Count != 0)
    {
        for (int i = 0; i < cboLocation.Items.Count; i++)
        {
            if (cboLocation.Text == cboLocation.Items[i].ToString())
                return;
        }
    }
    // If not found, add to list box
    cboLocation.Items.Add(cboLocation.Text);
}
```

You should be able to follow the logic of what's going on here. Add this method to the project.

Lastly, we address how to load a photo into the picture box control on the form. When the user clicks the button with an ellipsis (**btnPhoto**) next to the **Photo** label control, the open file dialog control (**dlgOpen**) should appear allowing the user to select a file. When that file is selected, the picture box control should display the photo. We have already written code to do most of these steps in the **ShowPhoto** method we use to display an already stored photo file. So, the **btnPhoto Click** event method is:

```
private void btnPhoto_Click(object sender, EventArgs e)
```

```
{  
    if (dlgOpen.ShowDialog() == DialogResult.OK)  
        ShowPhoto(dlgOpen.FileName);  
}
```

Add this method to the project.

Save and run the project to make sure the code compiles. You can try the new validations with the existing inventory items. Try opening a photo file. Any changes to an item won't be saved (changes to the combo box will be saved). We now add editing capability to allow saving changes to existing and new inventory items.

# Code Design – New Inventory Item

When the project first begins (with an empty input file) or when the user clicks the **New** button in the toolbar, we want the form to be in a state to accept a new set of inventory information. The steps are:

- Disable all toolbar buttons, except **tlbSave**.
- Blank out all text box controls and combo box.
- Uncheck check box control (**chkMarked**).
- Set calendar to today's date.
- Blank out **picPhoto** picture box and **lblPhoto** label.
- Give **txtItem** focus.

The code for these steps is placed in a general method **BlankValues**. We use a method because it is needed here, to start a new item, and later in the delete method (in case we delete the last item in the inventory). The method to implement the above steps is:

```
private void BlankValues()
{
    // blank input screen
    tlbNew.Enabled = false;
    tlbDelete.Enabled = false;
    tlbSave.Enabled = true;
    tlbPrevious.Enabled = false;
    tlbNext.Enabled = false;
    tlbPrint.Enabled = false;
    txtItem.Text = "";
    cboLocation.Text = "";
    chkMarked.Checked = false;
    txtSerialNumber.Text = "";
    txtPrice.Text = "";
    dtpPurchase.Value = DateTime.Today;
```

```
txtPurchase.Text = "";
txtNote.Text = "";
lblPhoto.Text = "";
picPhoto.Image = null;
txtItem.Focus();
}
```

Add this method to your project.

With this general method, the **tlbNew Click** event is coded simply as:

```
private void tlbNew_Click(object sender, EventArgs e)
{
    BlankValues();
}
```

Add this method to your project.

Now we need code to save entries for a new inventory item. When the user clicks the **Save** button on the toolbar, the following steps occur:

- Make sure there is an entry in **txtItem** (the only required input). Capitalize the first character (to insure proper ordering).
- Increment **numberEntries**.
- Determine entry location in **myInventory** array (alphabetically, using **txtItem Text** property)
- Once location is determined, move all items “below” location down one position in **myInventory** array.
- Establish properties for new array entry.
- Display new entry.
- Disable **tlbNew**, if we’ve reached the maximum number of entries.
- Enable **tlbDelete** and **tlbPrint**.

The tricky part of the code associated with these steps involves moving elements in the **myInventory** array. Let me explain. With normal Visual C# variables **a**, **b**,

**c** if you write:

```
a = b;  
b = c;
```

a will be replaced by the value in b. When b is replaced by c, a is unchanged, retaining the original value for b.

What if **a**, **b**, **c** are objects (such as elements of the **myInventory** array) and we write the same code:

```
a = b;  
b = c;
```

With objects, a will be assigned the same memory location as b, not a copy of its value. When b is then assigned to c, a will also change to c since it shares the same memory location. To avoid this, we need one additional step following the assignment of b to a. The modified code is:

```
a = b;  
b = new Object();  
b = c;
```

In this code, once b is assigned to a, we create a new object for b, giving it a new memory location prior to assigning it to c. This “breaks” the connection between memory locations for a and b. This modified code gives us the desired result of a having the original value for b and b having the new value for c.

The code for saving an entry is placed in the **tlbSave Click** event method:

```
private void tlbSave_Click(object sender, EventArgs e)  
{  
    // check for description  
    txtItem.Text = txtItem.Text.Trim();  
    if (txtItem.Text == "")  
    {
```

```
    MessageBox.Show("Must have item description.", "Error",
MessageBoxButtons.OK, MessageBoxIcon.Error);
    txtItem.Focus();
    return;
}
// capitalize first letter
txtItem.Text = txtItem.Text.Substring(0,1).ToUpper() +
txtItem.Text.Substring(1, txtItem.Text.Length - 1);
txtItem.Refresh();
numberEntries++;
// determine new current entry location based on description
currentEntry = 1;
if (numberEntries != 1)
{
    do
    {
        if (txtItem.Text.CompareTo(myInventory[currentEntry -
1].Description) < 0)
            break;
        currentEntry++;
    }
    while (currentEntry < numberEntries);
}
// move all entries below new value down one position unless at end
if (currentEntry != numberEntries)
{
    for (int i = numberEntries; i >= currentEntry + 1; i--)
    {
        myInventory[i - 1] = myInventory[i - 2];
        myInventory[i - 2] = new InventoryItem();
    }
}
```

```

myInventory[currentEntry - 1] = new InventoryItem();
myInventory[currentEntry - 1].Description = txtItem.Text;
myInventory[currentEntry - 1].Location = cboLocation.Text;
myInventory[currentEntry - 1].Marked = chkMarked.Checked;
myInventory[currentEntry - 1].SerialNumber = txtSerialNumber.Text;
myInventory[currentEntry - 1].PurchasePrice = txtPrice.Text;
myInventory[currentEntry - 1].PurchaseDate = dtpPurchase.Value;
myInventory[currentEntry - 1].PurchaseLocation = txtPurchase.Text;
myInventory[currentEntry - 1].PhotoFile = lblPhoto.Text;
myInventory[currentEntry - 1].Note = txtNote.Text;
ShowEntry(currentEntry);
if (numberEntries < maximumEntries)
    tlbNew.Enabled = true;
else
    tlbNew.Enabled = false;
    tlbDelete.Enabled = true;
    tlbPrint.Enabled = true;
}

```

Study this code. If there is no entry in **txtItem**, a message box is displayed. If there is an entry, capitalize the first character, to obtain proper ordering. We then determine the location of the new entry in the list of current entries. Once that position (**currentEntry**) is found, all other array elements are moved down one position (paying attention to how we “equate” objects). A new **InventoryItem** is created at **currentEntry - 1** (recall we’re using a 0-based array) and the control values placed in the appropriate object properties. Based on the number of entries, toolbar button status is modified. Add this method to your project.

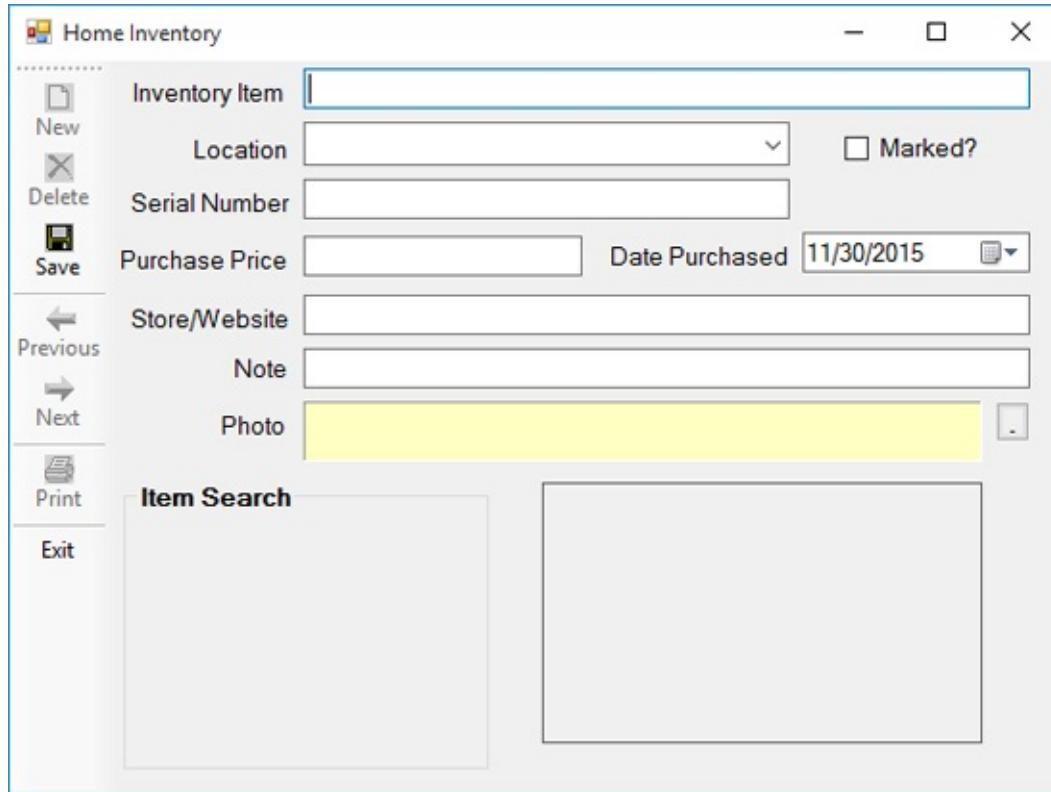
One option the user has while adding a new inventory item is to click on the **Exit** button in the toolbar, essentially stopping the program. We want to add a message box to the program to make sure if this happens, the user really means to exit. This message box could be placed in the **tlbExit Click** event (which closes the form, invoking the **FormClosing** event). Recall, however, a user can also exit a program by clicking the X in the upper right corner of the form, also invoking the **FormClosing** event. So, to intercept all exit requests, we place this

new message box in the **FormClosing** event method. The modified method is (changes are shaded, unmodified code writing properties back to file is not shown):

```
private void frmInventory_FormClosing(object sender,
FormClosingEventArgs e)
{
    if (MessageBox.Show("Are you sure you want to exit?\r\nAny
unsaved changes will be lost.", "Exit Program",
MessageBoxButtons.YesNo, MessageBoxIcon.Question,
MessageBoxDefaultButton.Button2) == DialogResult.No)
        e.Cancel = true;
    // write entries back to file
    StreamWriter outputFile = new
    StreamWriter(Application.StartupPath + "\\inventory.txt");
    .
    .
}
}
```

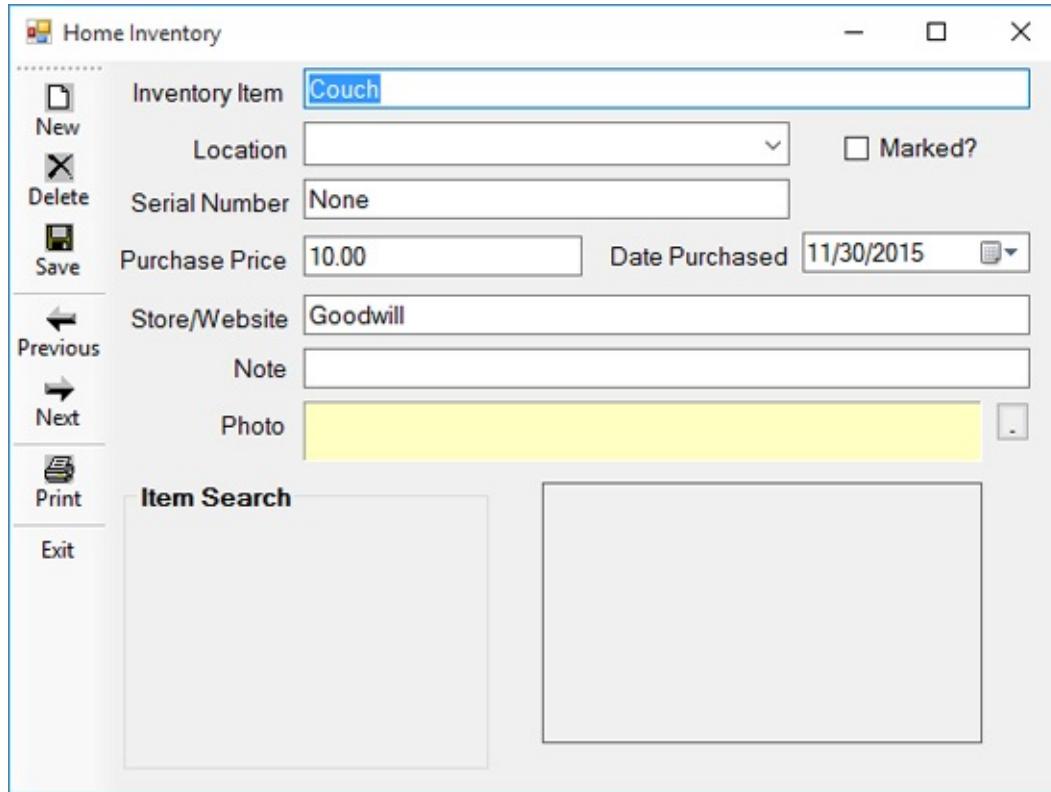
Notice in this code, if the user decides to ‘cancel’ the exit request, we do not simply exit the method. We need to cancel the **FormClosing** event request by setting the property, **e.Cancel**, to **true**. Add these code modifications to your project.

Save and run the project. You should now have the capability to add and save a new item to the inventory. Let’s try it. Click the **New** toolbar button to see a blank form ready for input (only the **Save** button and **Exit** buttons are enabled):

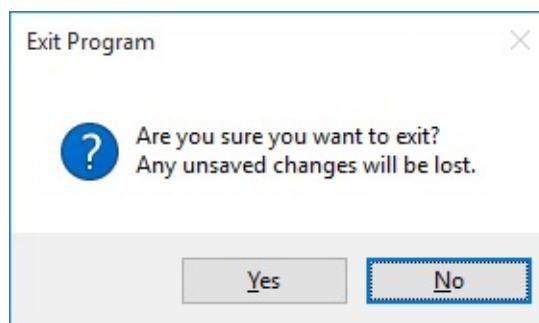


Type in some entries. Try the combo box selections (type a new one if you want). Add a photo if you have one. When done, click **Save** to make sure the item is properly sorted.

I added a **Couch** to my inventory (no photo). After clicking **Save**, I see:



Notice all toolbar buttons are now active. I can add another item or move to another item. By clicking **Previous** and **Next**, I can see that the item is properly located in the list (right after my **Coffee Cup**). Stop the project when you want. When you click **Exit**, you should see the message box we added to prevent inadvertent stopping of the program:



Make sure both the **Yes** and **No** options work correctly. We have made the **No** button 'default' to force the user to click **Yes** to exit.

# Code Design – Deleting Inventory Items

After entering a new item (or when viewing an existing item), the **Delete** toolbar button is enabled, but there is no code “behind” the button. Let’s write that code.

When a user clicks the **Delete** button while displaying an entry, the following should happen:

- Ask the user if he/she really wants to delete the entry.
- Move all items “below” displayed entry up one position in **myInventory** array. This removes the entry from the array.
- Decrement **numberEntries**.
- If entry deleted is last item, set form up for new entry.
- If more entries remain after deletion, display entry preceding deleted entry.

The code for the above steps is placed in the **tlbDelete\_Click** event method:

```
private void tlbDelete_Click(object sender, EventArgs e)
{
    if (MessageBox.Show("Are you sure you want to delete this item?",
"Delete Inventory Item", MessageBoxButtons.YesNo,
MessageBoxIcon.Question) == DialogResult.No)
        return;
    DeleteEntry(currentEntry);
    if (numberEntries == 0)
    {
        currentEntry = 0;
        BlankValues();
    }
    else
    {
        currentEntry--;
    }
}
```

```

if (currentEntry == 0)
    currentEntry = 1;
    ShowEntry(currentEntry);
}
}

```

Notice if we delete the last item in the inventory, the form is ‘blanked.’ Otherwise, the entry preceding the deleted entry (if there is one) is displayed. Add this method to your project.

The above code uses a general method **DeleteEntry** to remove an entry from the **myInventory** array. The code for this method is (the **int** argument indicates which item to remove):

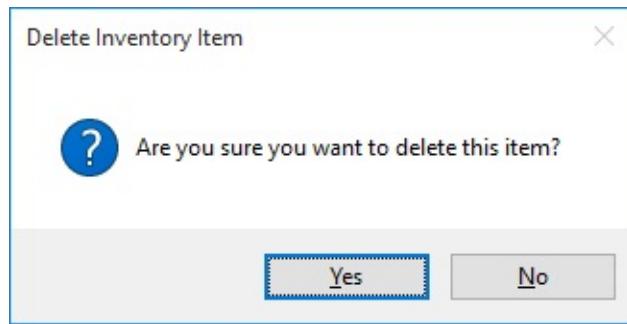
```

private void DeleteEntry(int j)
{
    // delete entry j
    if (j != numberEntries)
    {
        // move all entries under j up one level
        for (int i = j; i < numberEntries; i++)
        {
            myInventory[i - 1] = new InventoryItem();
            myInventory[i - 1] = myInventory[i];
        }
    }
    numberEntries--;
}

```

Again, notice the special way to “equate” objects. Add this method to your project.

Save and run the project with these changes. Make sure you can delete any entries you added earlier. You will see this message box when you try to delete an entry:



I was able to successfully delete my **Couch** from the inventory.

# Code Design – Editing Inventory Items

There is one problem you may notice. If you edit a current entry in the inventory and click **Save**, a new item is added to the inventory, rather than a simple update of the existing item. We need to modify the save method to be able to handle editing an existing item.

The approach we take is that if we are editing an existing item and click **Save**, we first delete it, then treat the modified item as if it is a new item. This allows us to use the existing save code and also properly sorts the edited item (if the **Description** property changed). The modified **tlbSave Click** event method is (changes are shaded, much unmodified code not shown):

```
private void tlbSave_Click(object sender, EventArgs e)
{
    // check for description
    txtItem.Text = txtItem.Text.Trim();
    if (txtItem.Text == "")
    {
        MessageBox.Show("Must have item description.", "Error",
MessageBoxButtons.OK, MessageBoxIcon.Error);
        txtItem.Focus();
        return;
    }
    if (tlbNew.Enabled)
    {
        // delete edit entry then resave
        DeleteEntry(currentEntry);
    }
    .
    .
    .
}
```

In this code, we use the **Enabled** status of **tlbNew** to determine if we are saving an existing item (**Enabled** is **true**) or a new item (**Enabled** is **false**). Add this method to your project.

Save and run the project. You now have full editing capability in the home inventory manager project. You can view inventory items, add new items to your inventory, delete items, or modify existing items. All information can now be properly saved.

We still need to add search and print capabilities to our project. But, first we need to address one “small” annoyance. If you edit an existing item and then click **New**, **Previous**, or **Next** without clicking **Save**, your changes are lost. (You can also click **Exit**, but we have already added code for that event to give the user a chance to reconsider). It would be nice if the program would “save us from ourselves” and ask us if we’d like to save the changes before moving on. This is a straightforward modification. We essentially need to know if anything was changed for a particular item. If changes were made and we attempt to move away from that item (click **New**, **Previous** or **Next**) without clicking **Save**, we can display a message box asking if the changes should be saved.

Declare a form level variable (**bool** type named **edited**):

```
bool edited;
```

This variable will be **true** when a current inventory item has been edited.

Modify the **tlbNew**, **tlbPrevious** and **tlbNext Click** event methods to ask the user if they want to save their changes (if **edited** is **true**). If the user responds **Yes**, do a save. The modified methods are (changes are shaded):

```
private void tlbNew_Click(object sender, EventArgs e)
{
    CheckSave();
    BlankValues();
}
```

```
private void tlbPrevious_Click(object sender, EventArgs e)
```

```

{
    CheckSave();
    currentEntry--;
    ShowEntry(currentEntry);
}

private void tlbNext_Click(object sender, EventArgs e)
{
    CheckSave();
    currentEntry++;
    ShowEntry(currentEntry);
}

```

The general method **CheckSave** used by each of these methods is:

```

private void CheckSave()
{
    if (edited)
    {
        if (MessageBox.Show("You have edited this item. Do you want to
save the changes?", "Save Item", MessageBoxButtons.YesNo,
MessageBoxIcon.Question) == DialogResult.Yes)
            tlbSave.PerformClick();
    }
}

```

Make the noted additions to the event methods and add **CheckSave** to your project.

We now need to modify the code to set **edited** to the proper value at the proper locations. **edited** should be **false** whenever a new inventory item is displayed. Add this single line:

```
edited = false;
```

as the first line in the **ShowEntry** general method.

How do we know if a user has edited a property associated with an inventory item? To know for sure if an edit occurred would take a bit of code. We could save the property values when the item is first displayed. Then, when the **New**, **Previous** or **Next** button is clicked, we could recheck to see if any of the values were changed. Such code would have to involve questions of just what constitutes a “different” value. We take a simpler approach. If the user accesses any of the controls used to set an inventory item property, we set **edited** to **true**. This is a safe approach. Even though nothing may have changed, we are assuming something did.

So we need this statement:

```
edited = true;
```

as the first line in the **txtItem\_KeyPress** event method (called if user changes the **Description**) and as the only line in the **txtItem\_Leave** event method (called if user moves to another control). The modified **KeyPress** event and the new **Leave** event methods are:

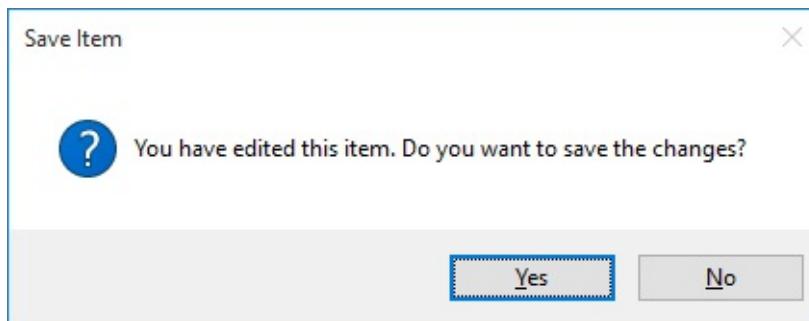
```
private void txtItem_KeyPress(object sender, KeyPressEventArgs e)
{
    edited = true;
    if ((int)e.KeyChar == 13)
        cboLocation.Focus();
}

private void txtItem_Leave(object sender, EventArgs e)
{
    edited = true;
}
```

Make these changes to your project.

Save and run the project. Add a new inventory item, make some entries

describing the item and click **Save**. Now modify something about your new item and click **New**, **Previous** or **Next**. You should see this message box:



You decide whether to save the changes or not. If you do, notice you don't have to click the **Save** button; it is done for you in code.

# Form Design – Inventory Item Search

As an inventory list grows, you would like to have some capability to search for particular items. In this project, we will use 26 small label controls in the group box marked **Item Search**. Each of these labels will have a letter of the alphabet. When a letter is clicked, the first item in the inventory beginning with that letter (if there is such an item) is displayed on the form.

Notice, up to this point in the form design, we avoided the task of adding these controls. It would be tedious to determine the size of each label and how to position them in the group box. Then, we need somehow to connect the 26 labels to a single **Click** event method. With just a bit of code, we can automate this task of **adding controls** to a form at **run-time**. We can also remove controls if desired. We will describe the method in general, then apply it to our search task.

To add a control at run-time, we need to follow five steps: (1) declare the control, (2) create the control, (3) set control properties, (4) add control to form and (5) connect event handlers. We look at each step separately for a generic control named **myControl** of type **ControlType**.

The first step is to declare a control using the usual statement:

```
ControlType myControl;
```

Quite often we declare an array of controls as we will for this project. Then, the control is created using the respective constructor:

```
myControl = new ControlType();
```

At this point, **myControl** is established with a default set of properties. You can overwrite any properties you choose. In particular, you must set values for the **Left** and **Top** properties. If you don't, all your new controls will be stacked in the upper left corner of the form (**Left** = 0, **Top** = 0). You probably also want to change the **Width** and **Height** properties. Once the properties are set, the control is added to the form using the **Add** method of the **Controls** object:

```
this.Controls.Add(myControl);
```

If you are adding the control to a group box or panel control, you would replace **this** (referring to the form) in the above statement with the container control's name.

So, now the control is created and on the form, but recognizes no events. Decide what events you want your control to respond to. If you want event **myEvent** to be handled by a method **myMethod**, an event handler is created using:

```
myControl.myEvent += new System.EventHandler(this.myMethod);
```

Before using this statement, the method **myMethod** should exist in the code window. It could be a method corresponding to an existing control or a new method you create. If you create it, the format is:

```
private void myMethod(object sender, EventArgs e)
{
    .
    .
}
```

You would write code in this method, assuming event handlers are added at runtime.

As an aside, you can add event methods in code for existing controls also. This sometimes saves a little typing at design time. For example, say you have a method that handles clicking on 20 button controls. Rather than assign the event method 20 times in the properties window, you could add it in code.

You can also remove controls from your application. To remove a control (named **myControl**) from the form, use:

```
this.Controls.Remove(myControl);
```

If you are removing a control from a group box or panel control, replace the keyword **this** with the container control's name. When the control is removed,

all event handlers for this control are modified to no longer include the removed control. This also happens when a control is deleted at design time.

Let's write code to add the 26 search labels to our inventory project. The code will go in the form **Load** event (we only want to execute this code once). The steps to add these labels are:

- Declare an array of 26 label controls (**searchLabel**).
- Determine desired label height and width.
- For each label:
  - o Construct label object.
  - o Set label properties (**Text**, **Top**, **Left**, **AutoSize**, ...)
  - o Add label to group box control.
  - o Create event handler for label **Click** event.

The trickiest part is determining the height and width of each button and positioning within the group box. A lot of trial and error is used. Try some values and see if the labels fit. Play with properties until the final result is what you want. In this project, we use four rows of 6 labels and a fifth row with 2 labels. For search label width, we divide the group box width by 7 (giving us a half-button width on each side for margin).

The modified form **Load** method that implements these steps is (changes are shaded, much unmodified code is not shown):

```
private void frmInventory_Load(object sender, EventArgs e)
{
    int n;
    int w, l, t;
    Label[] searchLabel = new Label[26];
    // search labels
    // determine label width
    w = (int)(grpSearch.Width / 7);
    l = (int)(0.5 * w);
    t = 16; // found by trial and error
```

```
// create and position 26 labels
for (int i = 0; i < 26; i++)
{
    // create new label
    searchLabel[i] = new Label();
    searchLabel[i].AutoSize = false;
    searchLabel[i].TextAlign = ContentAlignment.MiddleCenter;
    // set text property
    searchLabel[i].Text = ((char) (65 + i)).ToString();
    // position (label height set to 25 pixels)
    searchLabel[i].Width = w;
    searchLabel[i].Height = 24;
    searchLabel[i].Left = l;
    searchLabel[i].Top = t;
    // give cool colors
    searchLabel[i].ForeColor = Color.Blue;
    searchLabel[i].BackColor = Color.Yellow;
    searchLabel[i].BorderStyle = BorderStyle.FixedSingle;
    searchLabel[i].Font = new Font("Microsoft Sans Serif", 10,
FontStyle.Bold);
    // add button to group box
    grpSearch.Controls.Add(searchLabel[i]);
    // add event handler
    searchLabel[i].Click += new
System.EventHandler(this.SearchLabelClick);
    // next left
    l += w;
    // six buttons per row
    if ((i + 1) % 6 == 0)
    {
        l = (int)(0.5 * w);
        t += searchLabel[i].Height;
```

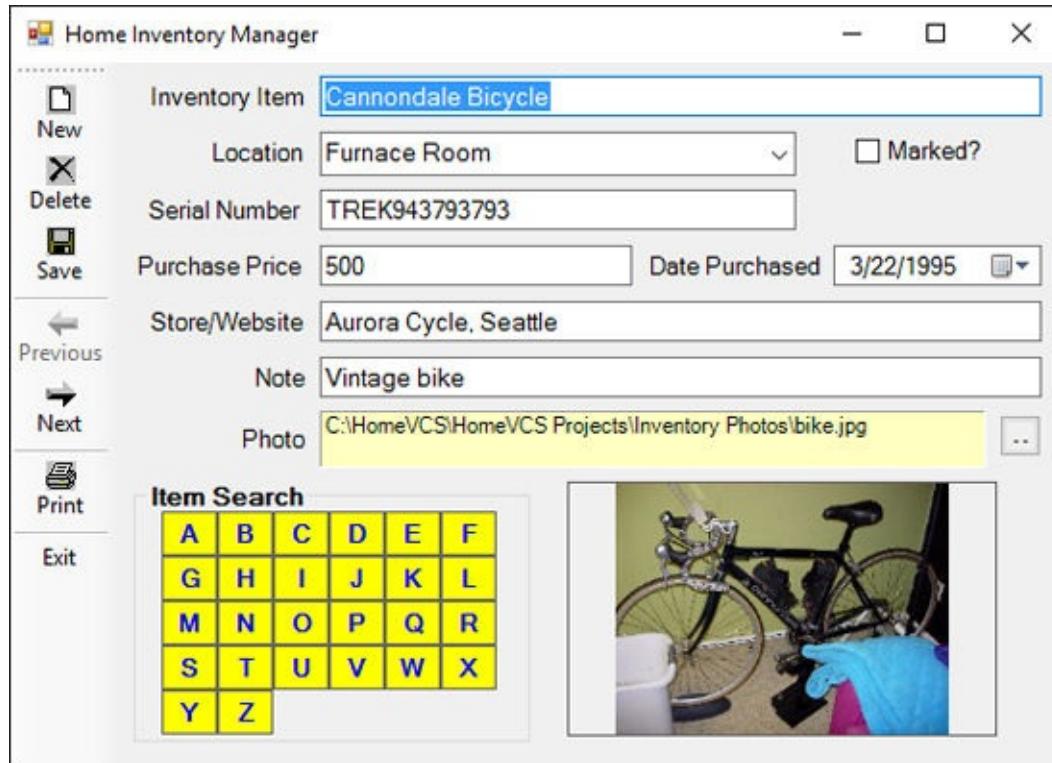
```
        }  
    }  
// open file for entries  
.  
.  
.  
}
```

Note how each property is assigned before placing the label in the group box. I did a lot of playing around with properties before settling on this final set. Note how we place 6 labels in each row. And, notice how the event handler (method is **SearchLabelClick**) is added to each label. Make these modifications in your project.

You also need to add the method that will handle the **Click** event for each of the 26 search labels. Add this empty framework for now:

```
private void SearchLabelClick(object sender, EventArgs e)  
{  
}
```

Save and run the project. The search labels should now appear on the form in the group box control:



Now, we need code to implement the search.

# Code Design – Inventory Item Search

When a user clicks one of the search label controls, the following happens:

- Determine which search label was clicked.
- Find first item in inventory list that begins with ‘clicked’ letter – display that item.
- If no matching item found, display a message box.

The code to implement these steps is straightforward and is placed in the **SearchLabelClick** method:

```
private void SearchLabelClick(object sender, EventArgs e)
{
    Label labelClicked;
    string s;
    int i;
    if (numberEntries == 0)
        return;
    // search for item letter
    labelClicked = (Label) sender;
    i = 0;
    do
    {
        s = myInventory[i].Description.Substring(0, 1);
        if (s == labelClicked.Text)
        {
            currentEntry = i + 1;
            ShowEntry(currentEntry);
            return;
        }
        i++;
    }
```

```

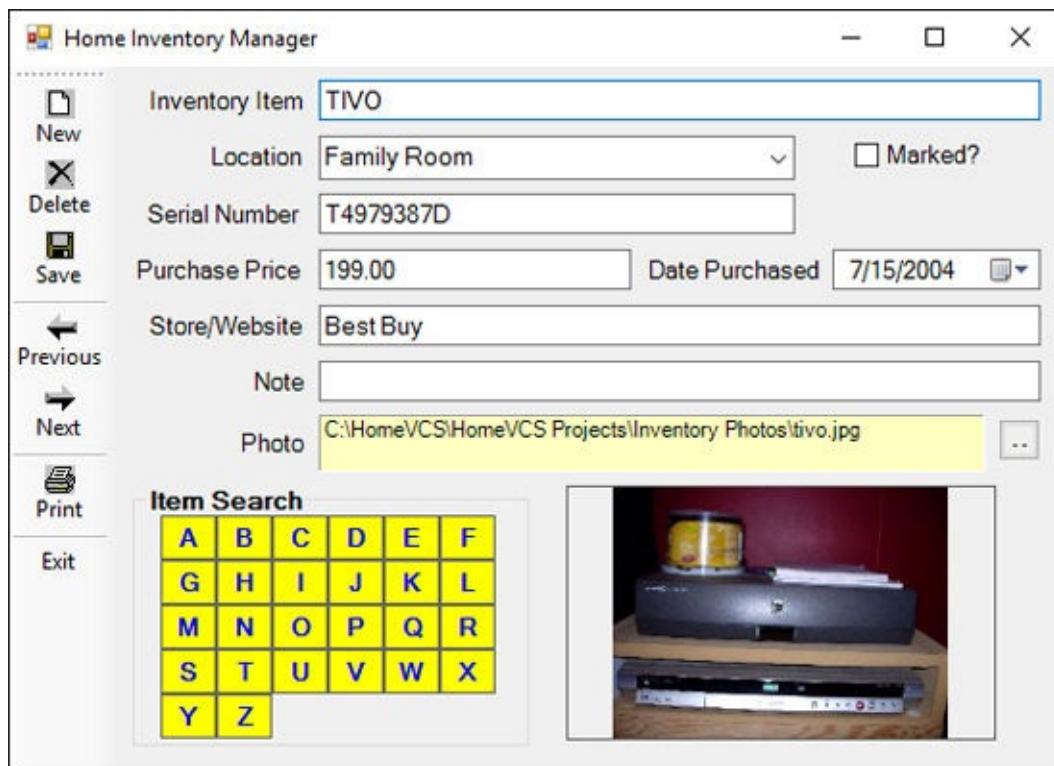
        }

        while (i < numberEntries);
        MessageBox.Show("No " + labelClicked.Text + " inventory items.",
        "None Found", MessageBoxButtons.OK, MessageBoxIcon.Information);
    }
}

```

Add this method to your project.

Save and run the project. Notice how the search labels are properly created and positioned. When I click the ‘T’ search label using the sample inventory, I see:



Notice we see the first T entry (**TIVO**). To see entries ‘around’ this choice, use the **Previous** and **Next** buttons. Click on ‘R’ and you’ll see:

None Found

X



No R inventory items.

OK

# Printing with Visual C#

One last capability we will add to our home inventory project is printing. A printed copy of our items would be helpful for archival purposes and for any potential insurance claims. Printing is one of the more tedious programming tasks within Visual C#. But, fortunately, it is straightforward and there are dialog controls that help with the tasks. We will introduce lots of new topics here. All steps will be reviewed. Objects used in printing are found in the **Drawing.Printing** namespace. For any application with printing, add this **using** declaration at the top of the code window:

```
using System.Drawing.Printing;
```

To perform printing in Visual C#, we use the **PrintDocument** object. This object controls the printing process and has four important properties:

Property	Description
DefaultPageSettings	Indicates default page settings for the document.
DocumentName	Indicates the name displayed while the document is printing.
PrintController	Indicates the print controller that guides the printing process.
PrinterSettings	Indicates the printer that prints the document.

The steps to print a document (which may include text and graphics) using the **PrintDocument** object are:

- Declare a **PrintDocument** object
- Create a **PrintDocument** object
- Set any properties desired.
- Print the document using the **Print** method of the **PrintDocument** object.

The first three steps are straightforward. To declare and create a **PrintDocument** object named **MyDocument**, use:

```
PrintDocument myDocument;  
.  
.  
myDocument = new PrintDocument();
```

Any properties needed usually come from print dialog boxes we'll examine in a bit.

The last step poses the question: how does the **PrintDocument** object print with the **Print** method? Printing is done in a general Visual C# method associated with the **PrintDocument.PrintPage** event. This is a method you must create and write. The method tells the **PrintDocument** object what goes on each page of your document. Once the method is written, you need to add the event handler in code (we just learned how to do that) so the **PrintDocument** object knows where to go when it's ready to print a page. It may sound confusing now, but once you've done a little printing, it's very straightforward.

The general Visual C# method for printing your pages (**PrintPage** in this case) must be of the form:

```
private void PrintPage(object sender, PrintPageEventArgs e)  
{  
.  
.  
}
```

In this method, you 'construct' each page that the **PrintDocument** object is to print. And, you'll see the code in this method is familiar.

In the **PrintPage** method, the argument **e** (of type **PrintPageEventArgs**) has many properties with information about the printing process. The most important property is the **graphics object**:

### e.Graphics

Something familiar! **PrintDocument** provides us with a **graphics** object to

‘draw’ each page we want to print. This is the same graphics object we used in Chapters 9 and 10 to draw lines and text. And, all the methods we learned there apply here! We’ll look at how to do this in detail next. But, first, let’s review how to establish and use the **PrintDocument** object.

A diagram summarizes the printing process:



The **PrintDocument** object provides a **Graphics Object** to ‘draw’ each page. Each page is created in the **PrintPage** event handler (called by the **PrintDocument Print** method).

Here is an annotated code segment that establishes a **PrintDocument** object (**MyDocument**) and connects it to a method named **PrintPage** that provides the pages to print via the graphics object:

```
// Declare the document
PrintDocument myDocument;
.

.

// Create the document and name it
myDocument = new PrintDocument();
myDocument.DocumentName = "My Document";
.

.

// You could set other properties here
.

.

// Add code handler
myDocument.PrintPage += new PrintPageEventHandler(this.PrintPage);

// Print document
myDocument.Print();
```

```
// Dispose of document when done printing  
myDocument.Dispose();
```

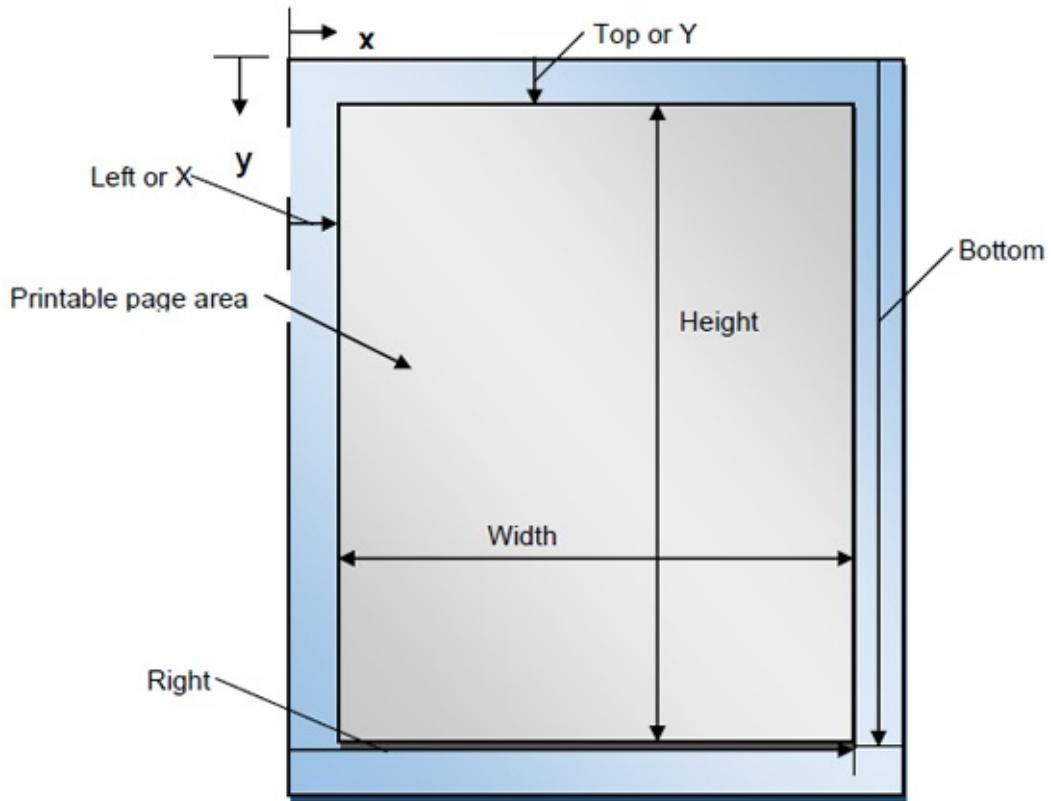
This code assumes the method **PrintPage** is available. Let's see how to build such a method.

# Printing Document Pages

The **PrintDocument** object provides (in its **PrintPage** event) a graphics object (**e.Graphics**) for ‘drawing’ our pages. And, that’s just what we do using familiar graphics methods. For each page in our printed document, we draw the desired text information (**DrawString** method), any lines (**DrawLine** method), or images (**DrawImage** method, a new method we will review).

Once a page is completely drawn to the graphics object, we ‘tell’ the **PrintDocument** object to print it. We repeat this process for each page we want to print. This does require a little bit of work on your part. You must know how many pages your document has and what goes on each page. I usually define a page number variable to help keep track of the current page being drawn.

Let’s look at the graphics object for a single page. The boundaries of the printed page are defined by the **e.MarginBounds** properties (these are established by the **PrinterSettings** property):



This becomes our palette for positioning items on a page. Horizontal position is governed by **x** (increases from 0 to the right) and vertical position is governed by **y** (increases from 0 to the bottom).

The process for each page is to decide “what goes where” and then position the desired information using the appropriate graphics method. Any of the graphics methods we have learned can be used to put information on the graphic object.

To place text on the graphics object (**e.Graphics**), use the **DrawString** method introduced in Chapter 10. To place the string **myString** at position **(x, y)**, using the font object **myFont** and brush object **myBrush**, the syntax is:

```
e.Graphics.DrawString(myString, myFont, myBrush, x, y);
```

With this statement, you can place any text, anywhere you like, with any font, any color and any brush style. You just need to make the desired specifications. Each line of text on a printed page will require a **DrawString** statement.

Also in Chapter 10, we saw two methods for determining the size of strings. This is helpful for both vertical and horizontal placement of text on a page. To determine the height (in pixels) of a particular font, use:

```
myFont.GetHeight();
```

If you need width and height of a string use:

```
e.Graphics.MeasureString(myString, myFont);
```

This method returns a **SizeF** structure with two properties: **Width** and **Height** (both in pixels). These two properties are useful for justifying (left, right, center, vertical) text strings.

Many times, you use lines in a document to delineate various sections. To draw a line on the graphics object, use the **DrawLine** method (from Chapter 9):

```
e.Graphics.DrawLine(myPen, x1, y1, x2, y2);
```

This statement will draw a line from **(x1, y1)** to **(x2, y2)** using the pen object

**myPen.**

Finally, the **DrawImage** method is used to position an image (**myImage**) object on a page. This is a new method, but easy to use. The syntax is:

```
e.Graphics.DrawImage(myImage, x, y, width, height);
```

The upper left corner of **myImage** will be at **(x, y)** with the specified **width** and **height**. Any image will be scaled to fit the specified region.

We've seen all of these graphics methods before, so their use should be familiar. You should note that each item on a printed page requires at least one line of code. That results in lots of coding for printing. So, if you're writing lots of code in your print routines, you're probably doing it right.

As a last step, we need to inform the **PrintDocument** object when a page is complete and can be printed. Once a page is complete, there are two possibilities: there are more pages to print or there are no more pages to print. The **e.HasMorePages** property (Boolean) is used specify which possibility exists. If a page is complete and there are still more pages to print, use:

```
e.hasMorePages = true;
```

In this case, the **PrintDocument** object will return to the **PrintPages** event for the next page. If the page is complete and printing is complete (no more pages), use:

```
e.hasMorePages = false;
```

This tells the **PrintDocument** object its job is done. At this point, you should dispose of the **PrintDocument** object.

The best way to learn how to print in Visual C# is to do lots of it. You'll develop your own approaches and techniques as you gain familiarity. Many print jobs just involve the user clicking a button marked '**Print**' and the results appear on printed page with no further interaction. If more interaction is desired, there are dialog controls that help specify desired printing job properties. Using these controls adds more code to your application. You must take any user inputs and

implement these values in your program. We'll look at the **PrintDialog** and **PrintPreviewDialog** controls here. In our inventory project, we'll use the **PrintPreviewDialog** control, which is especially cool!!

# PrintDialog Control

## In Toolbox:



## Below Form (Default Properties):



The **PrintDialog** control allows the user to select which printer to use, choose page orientation, printed page range and number of copies. This is the same dialog box that appears in many Windows applications.

### PrintDialog Properties:

<b>Name</b>	Gets or sets the name of the print dialog (I usually name this control <b>dlgPrint</b> ).
<b>AllowPrintToFile</b>	Gets or sets a value indicating whether the Print to file check box is enabled.
<b>AllowSelection</b>	Gets or sets a value indicating whether the From...To... Page option button is enabled.
<b>AllowSomePages</b>	Gets or sets a value indicating whether the Pages option button is enabled.
<b>Document</b>	Gets or sets a value indicating the PrintDocument used to obtain PrinterSettings.
<b>PrinterSettings</b>	Gets or sets the PrinterSettings the dialog box is to modify.
<b>PrintToFile</b>	Gets or sets a value indicating whether the Print to file check box is checked

### PrintDialog Methods:

<b>ShowDialog</b>	Displays the dialog box. Returned value indicates which button was clicked by user ( <b>OK</b> or
-------------------	---

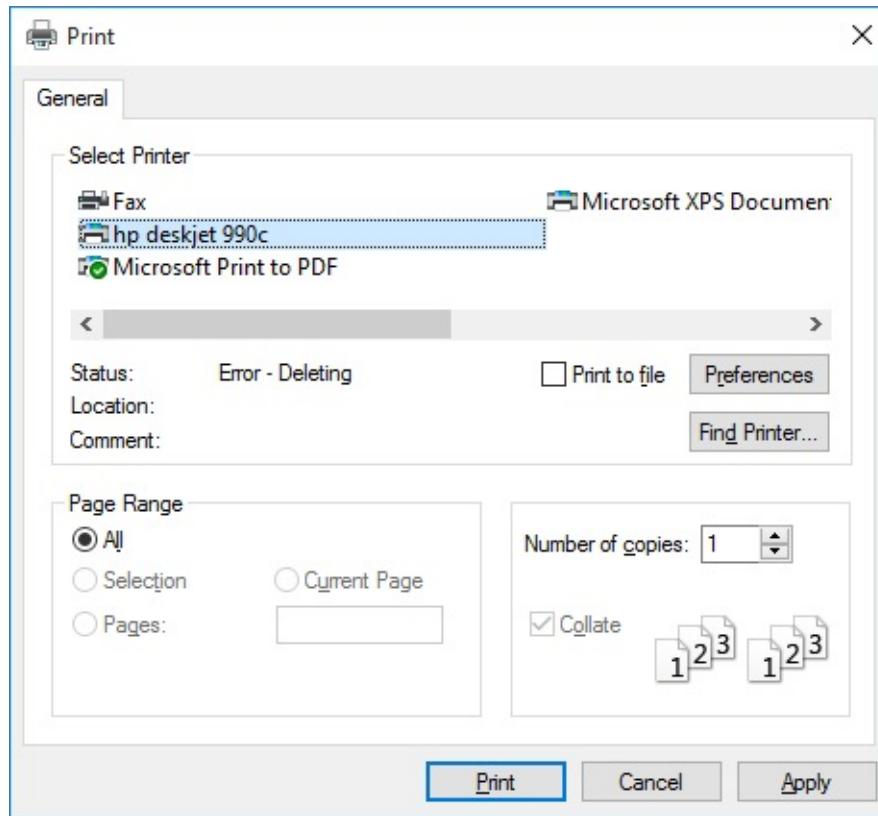
**Cancel).**

To use the **PrintDialog** control, we add it to our application the same as any control. It will appear in the tray below the form. Once added, we set a few properties. Then, we write code to make the dialog box appear when desired. The user then makes selections and closes the dialog box. At this point, we use the provided information for our tasks.

The **ShowDialog** method is used to display the **PrintDialog** control. For a control named **dlgPrint**, the appropriate code is:

```
dlgPrint.ShowDialog();
```

And the displayed dialog box is:



The user makes any desired choices. Once complete, the **OK** button is clicked. At this point, various properties are available for use (namely **PrinterSettings**). **Cancel** can be clicked to cancel the changes. The **ShowDialog** method returns the clicked button. It returns **DialogResult.OK** if **OK** is clicked and returns

**DialogResult.Cancel** if **Cancel** is clicked.

Typical use of **PrintDialog** control:

- Set the **Name** property. Decide what options should be available.
- Use **ShowDialog** method to display dialog box, prior to printing with the **PrintDocument** object.
- Use **PrinterSettings** properties to change printed output.

# PrintPreviewDialog Control

## In Toolbox:



## Below Form (Default Properties):



The **PrintPreviewDialog** control is a great addition to Visual C#. It lets the user see printed output in preview mode. They can view all pages, format page views and zoom in on or out of any. The previewed document can also be printed from this control. This is also a useful “temporary” control for a programmer to use while developing printing routines. By viewing printed pages in a preview mode, rather than on a printed page, many trees are saved as you find tune your printing code.

## PrintPreviewDialog Properties:

<b>Name</b>	Gets or sets the name of the print preview dialog (I usually name this control <b>dlgPreview</b> )
<b>AcceptButton</b>	Gets or sets the button on the form that is clicked when the user presses the <Enter> key.
<b>Document</b>	Gets or sets the document to preview.
<b>Text</b>	Gets or sets the text associated with this control.

## PrintPreviewDialog Methods:

<b>ShowDialog</b>	Displays the dialog box. Returned value indicates which button was clicked by user ( <b>OK</b> or <b>Cancel</b> ).
-------------------	--

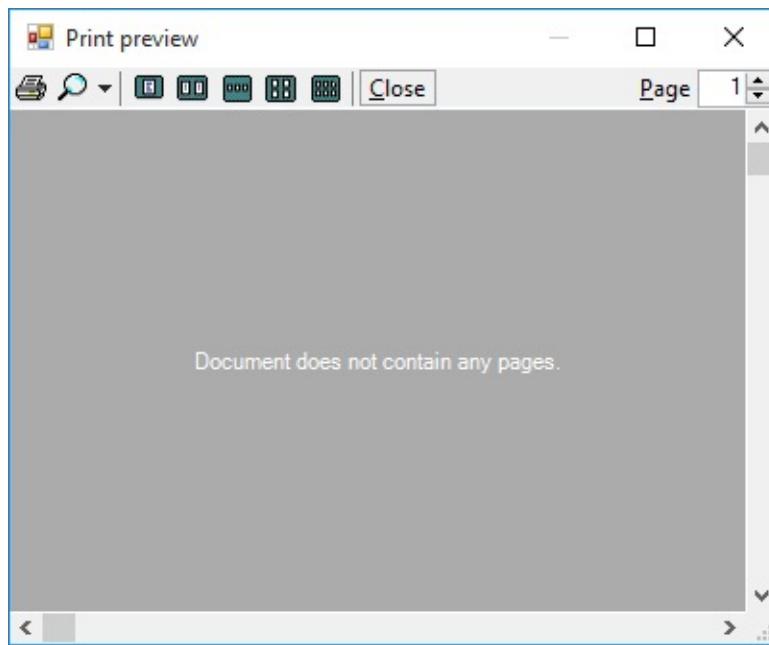
To use the **PrintDialog** control, we add it to our application the same as any control. It will appear in the tray below the form. Once added, we set a few properties, primarily **Document**. Make sure the **PrintPage** event Is properly

coded for the selected **Document**. Add code to make the dialog box appear when desired. The document pages will be generated and the user can see it in the preview window.

The **ShowDialog** method is used to display the **PrintPreviewDialog** control. For a control named **dlgPreview**, the appropriate code is:

```
dlgPreview.ShowDialog();
```

And the displayed dialog box (with no document) is:



The user can use the various layout, zoom and print options (in the control's toolbar when document is available) in previewing the displayed document. When done, the user closes the dialog control.

Typical use of **PrintPreviewDialog** control:

- Set the **Name** property. Set the **Document** property.
- Use **ShowDialog** method to display dialog box and see the previewed document.

# Code Design – Printing the Inventory

The format used for the printed inventory is straightforward – modify it as you see fit. Each page will have a simple header (giving the page number) and will hold two items from the inventory. Each property for each item (including the picture) will be printed. Items will be separated by a single straight line.

The code to establish the print document and display the printed inventory in a preview dialog goes in the **tlbPrint Click** event method. But, first, we need three form level variable declarations:

```
int pageNumber;  
PrintDocument myDocument;  
const int entriesPerPage = 2;
```

The **tlbPrint Click** method is then:

```
private void tlbPrint_Click(object sender, EventArgs e)  
{  
    // do printing  
    pageNumber = 1;  
    // create document  
    myDocument = new PrintDocument();  
    myDocument.DocumentName = "Home Inventory Items";  
    myDocument.PrintPage += new  
    PrintPageEventHandler(this.PrintPage);  
    dlgPreview.Document = myDocument;  
    dlgPreview.ShowDialog();  
    myDocument.Dispose();  
}
```

We initialize the page number and create the **PrintDocument** object. The **PrintPage** handler is attached and the document is viewed in the preview dialog control (allowing it to be printed if desired). Add this method and the variable

declarations to your project.

The **PrintPage** method to accomplish the printing is:

```
private void PrintPage(object sender, PrintPageEventArgs e)
{
    Font printFont;
    int y, i, iEnd;
    Image myImage;
    float ratio;
    // here you decide what goes on each page and draw it
    printFont = new Font("Arial", 14, FontStyle.Bold);
    e.Graphics.DrawString("Home Inventory Items - Page " +
pageNumber.ToString(), printFont, Brushes.Black, e.MarginBounds.Left,
e.MarginBounds.Top);
    // starting y position
    printFont = new Font("Arial", 12, FontStyle.Regular);
    y = (int)(e.MarginBounds.Top + 3 * printFont.GetHeight());
    iEnd = entriesPerPage * pageNumber;
    if (iEnd > numberEntries)
        iEnd = numberEntries;
    for (i = 1 + entriesPerPage * (pageNumber - 1); i <= iEnd; i++)
    {
        // dividing line
        e.Graphics.DrawLine(Pens.Black, e.MarginBounds.Left, y,
e.MarginBounds.Right, y);
        printFont = new Font("Arial", 12, FontStyle.Bold);
        e.Graphics.DrawString(myInventory[i - 1].Description, printFont,
Brushes.Black, e.MarginBounds.X, y);
        y += (int)(printFont.GetHeight());
        printFont = new Font("Arial", 12, FontStyle.Regular);
        e.Graphics.DrawString("Location: " + myInventory[i -
1].Location, printFont, Brushes.Black, e.MarginBounds.X + 25, y);
    }
}
```

```

y += (int)(printFont.GetHeight());
if (myInventory[i - 1].Marked)
    e.Graphics.DrawString("Item is marked with identifying
information.", printFont, Brushes.Black, e.MarginBounds.X + 25, y);
else
    e.Graphics.DrawString("Item is NOT marked with identifying
information.", printFont, Brushes.Black, e.MarginBounds.X + 25, y);
y += (int)(printFont.GetHeight());
e.Graphics.DrawString("Serial Number: " + myInventory[i -
1].SerialNumber, printFont, Brushes.Black, e.MarginBounds.X + 25, y);
y += (int)(printFont.GetHeight());
e.Graphics.DrawString("Price: $" + String.Format("{0:f2}",
myInventory[i - 1].PurchasePrice) + ", Purchased on: " + myInventory[i -
1].PurchaseDate.ToShortDateString(), printFont, Brushes.Black,
e.MarginBounds.X + 25, y);
y += (int)(printFont.GetHeight());
e.Graphics.DrawString("Purchased at: " + myInventory[i -
1].PurchaseLocation, printFont, Brushes.Black, e.MarginBounds.X + 25,
y);
y += (int)(printFont.GetHeight());
e.Graphics.DrawString("Note: " + myInventory[i - 1].Note,
printFont, Brushes.Black, e.MarginBounds.X + 25, y);
y += (int)(printFont.GetHeight());
try
{
    // maintain original width/height ratio
    myImage = Image.FromFile(myInventory[i - 1].PhotoFile);
    ratio = (float) (myImage.Width) / myImage.Height;
    e.Graphics.DrawImage(myImage, e.MarginBounds.X + 25, y,
200 * ratio, 200);
}
catch
{
    // have place to go in case image file doesn't open
}

```

```

        }
        y += 2 * (int)(printFont.GetHeight()) + 200;
    }
    pageNumber++;
    if (iEnd != numberEntries)
        e.HasMorePages = true;
    else
    {
        e.HasMorePages = false;
        pageNumber = 1;
    }
}

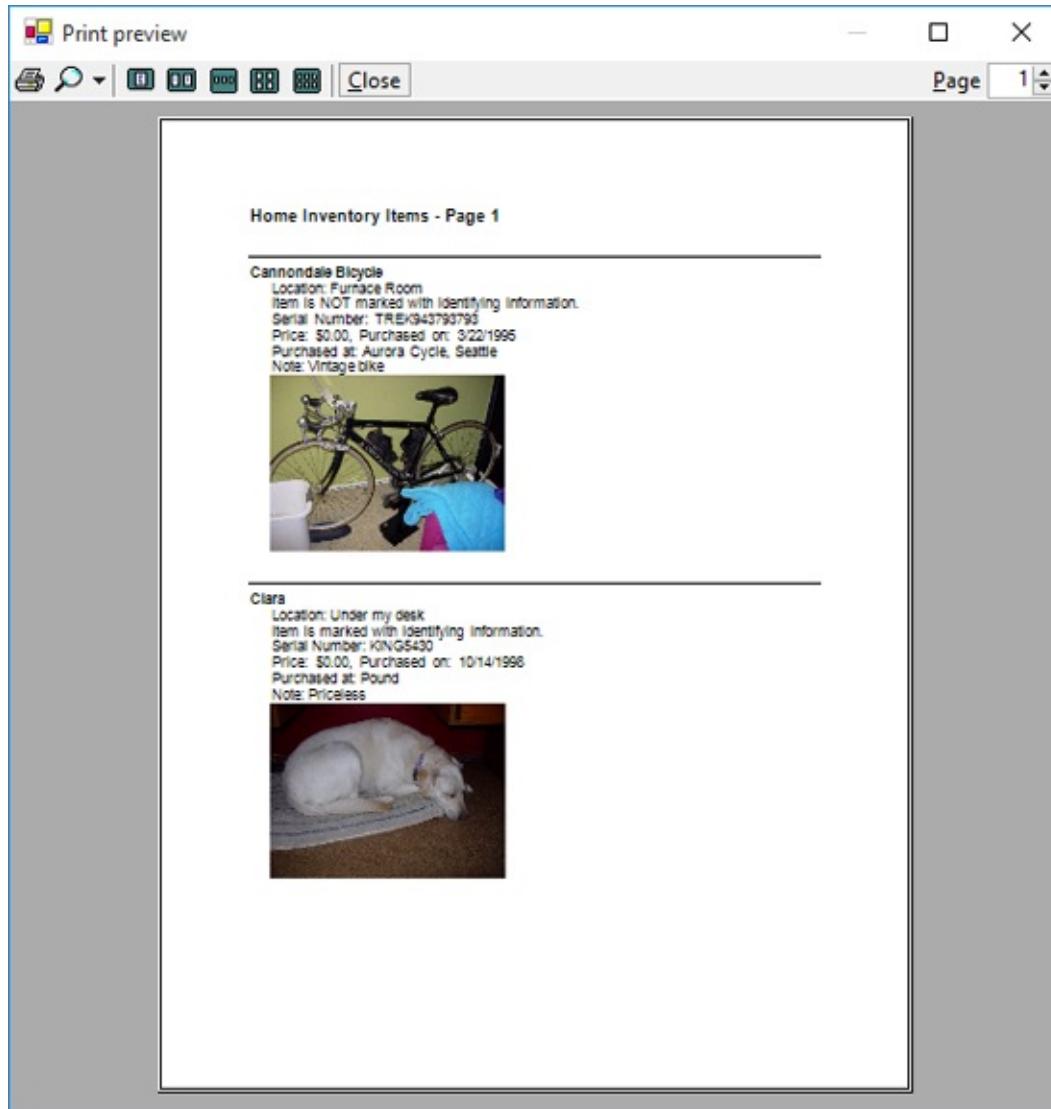
```

Yes, there's lots of code here, but the steps are straightforward:

- Print the header.
- For next two items in inventory:
  - o Draw dividing line.
  - o Print on separate lines: **Description, Location, Marked Statement, Serial Number, Purchase Price, Purchase Date, Purchase Location**, and any **Note**.
  - o Print picture (maintaining original height-to-width ratio).
- Check to see if more pages are to be printed.

You should see all of these steps in the above code. Note specifically how the vertical print location is updated.

Save and run the project. Click the **Print** button on the toolbar. For the example inventory, the print preview dialog should appear, displaying the first 'printed' page:



You can view other pages. Or you can obtain a hard copy by clicking the button with the **Printer** icon.

# Home Inventory Manager Project Review

The **Home Inventory Manager Project** is now complete. Save and run the project and make sure it works as designed. Use the program to keep track of your belongings. You'll want to delete the **inventory.txt** file currently in the project **Bin\Debug** folder and start over adding your own items and establishing your own combo box elements.

If there are errors in your implementation, go back over the steps of form and code design. Use the debugger when needed. Go over the developed code – make sure you understand how different parts of the project were coded. As mentioned in the beginning of this chapter, the completed project is saved as **Inventory** in the **HomeVCS\HomeVCS Projects** folder.

While completing this project, new concepts and skills you should have gained include:

- Use of image list and toolstrip controls for toolbars.
- Use of combo box control.
- Basic object-oriented programming concepts and how to define your own classes and objects.
- How to add controls at run-time.
- How to add printing to a project, including use of the print preview dialog control.

# Home Inventory Manager Project Enhancements

Possible enhancements to the home inventory manager project include:

- After clicking the **New** toolbar button, you must add an item to the inventory and click **Save**. There is no **Cancel** option – add such an option.
- The implemented search is rather basic. Add a search capability that looks through all the information in the inventory for certain terms or parts of terms. Use the **Soundex** function from the **Multiple Choice Exam Project** to do “sound-alike” searches.
- Modify the project to allow opening and saving of separate inventory files. That is, replace the built-in file (inventory.txt) with one you open/save using the dialog controls.

**11**

## **Snowball Toss Game Project**

# Review and Preview

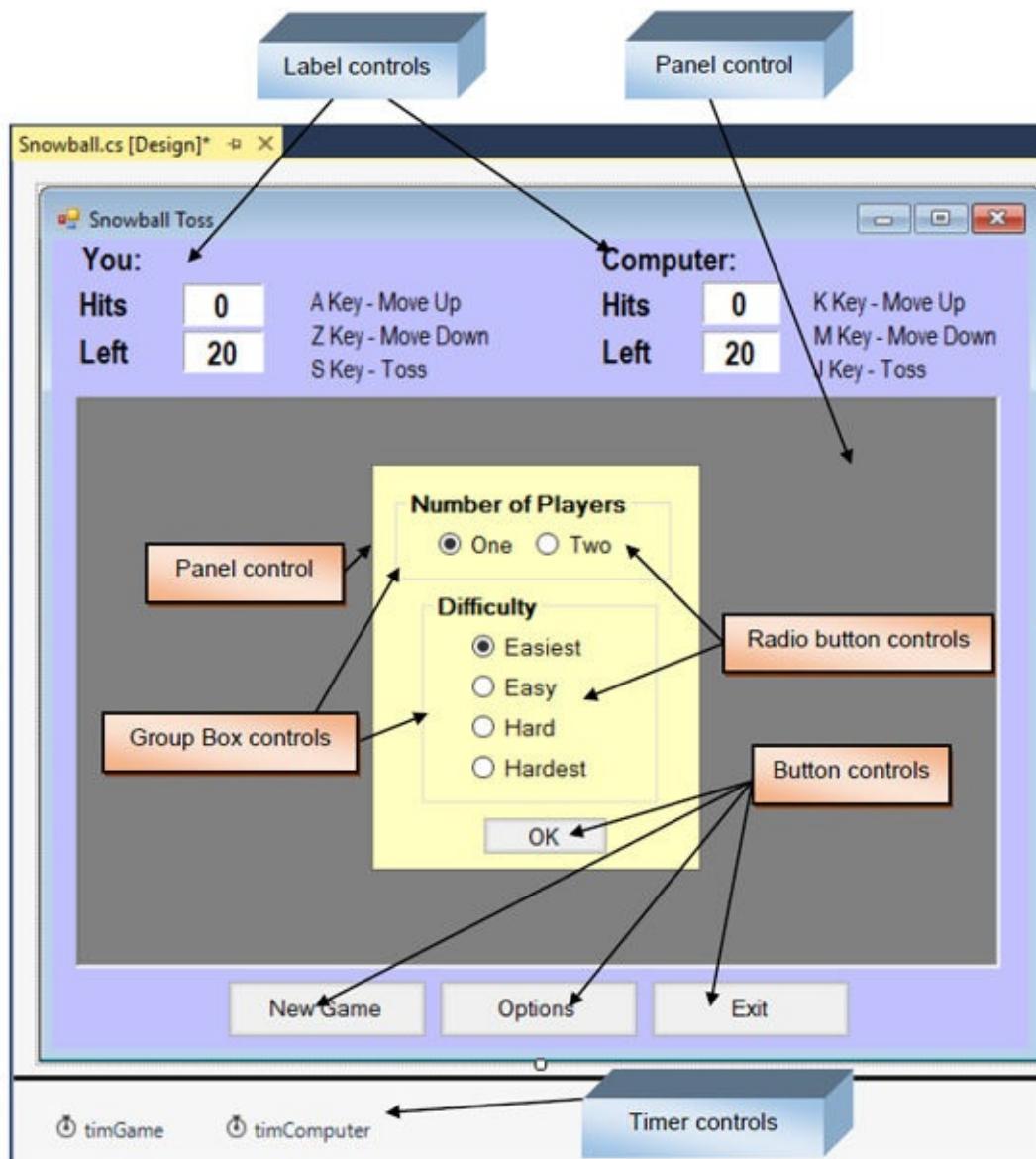
In the final project, we'll have some fun. In the **Snowball Toss Game Project**, two players toss snowballs at each other or a single player plays against the computer - the most hits wins! We introduce concepts needed for game programming – animation, collision detection, keyboard control, and sounds.

We also look at adding methods to custom objects and how to use inheritance. And, we'll look at how to give our computer a semblance of intelligence.

# Snowball Toss Game Project Preview

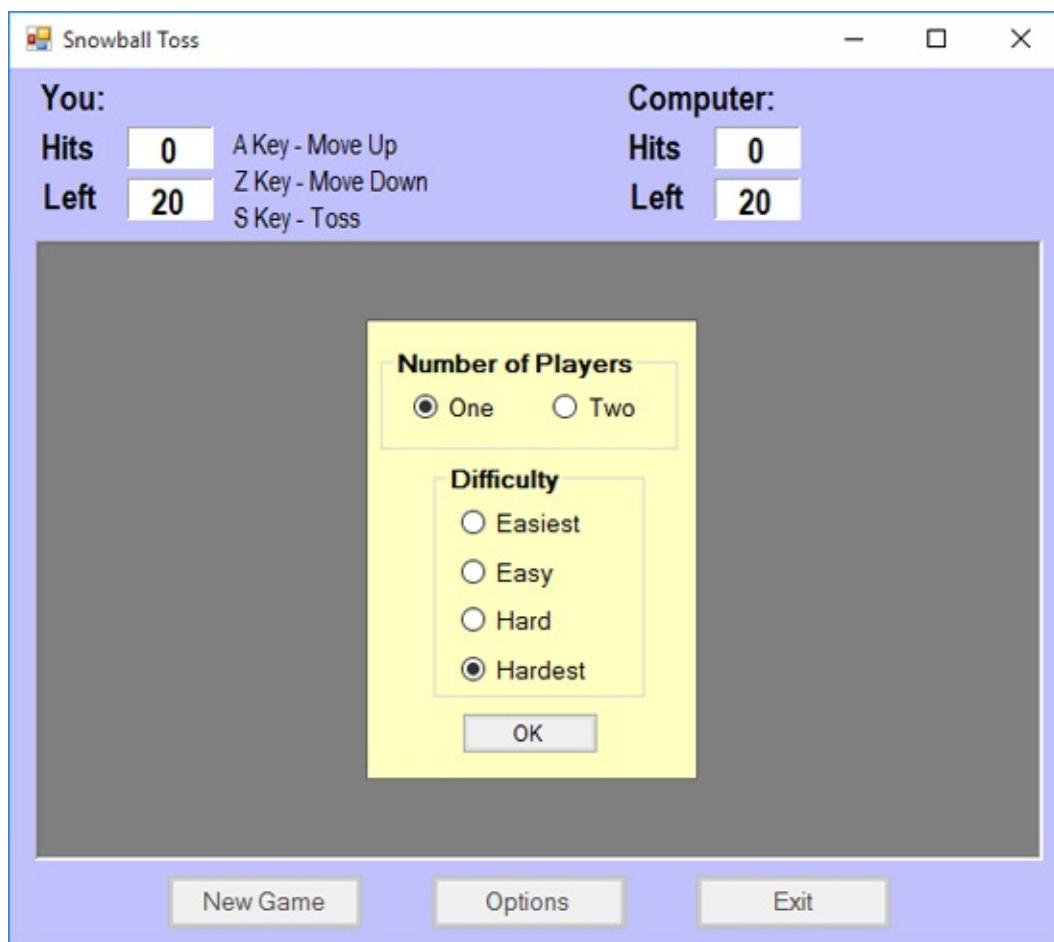
In this chapter, we will build a **Snowball Toss Game** program. This program lets two players compete in throwing snowballs at each other. Or, optionally, a single player can play against a computer with adjustable ‘smarts’.

The finished project is saved as **Snowball** in the **HomeVCS\HomeVCS Projects** folder. Start Visual C# and open the finished project. Open the form (double-click **Snowball.cs** in Solution Explorer) and you will see:



The label controls at the top of the form provide game status (score, snowballs left to toss and keys to use to move players). There are two panel controls. The large gray panel is the game field. The smaller panel control (only appears when not playing) is used to select game options. Radio buttons within group boxes select the number of players and the game difficulty (if playing against the computer). Three button controls are used to start/stop the game, set options and stop the program. Two timer controls are used – one to update the game graphics and one to simulate the computer moves (when playing with one player). There are no new controls being used here.

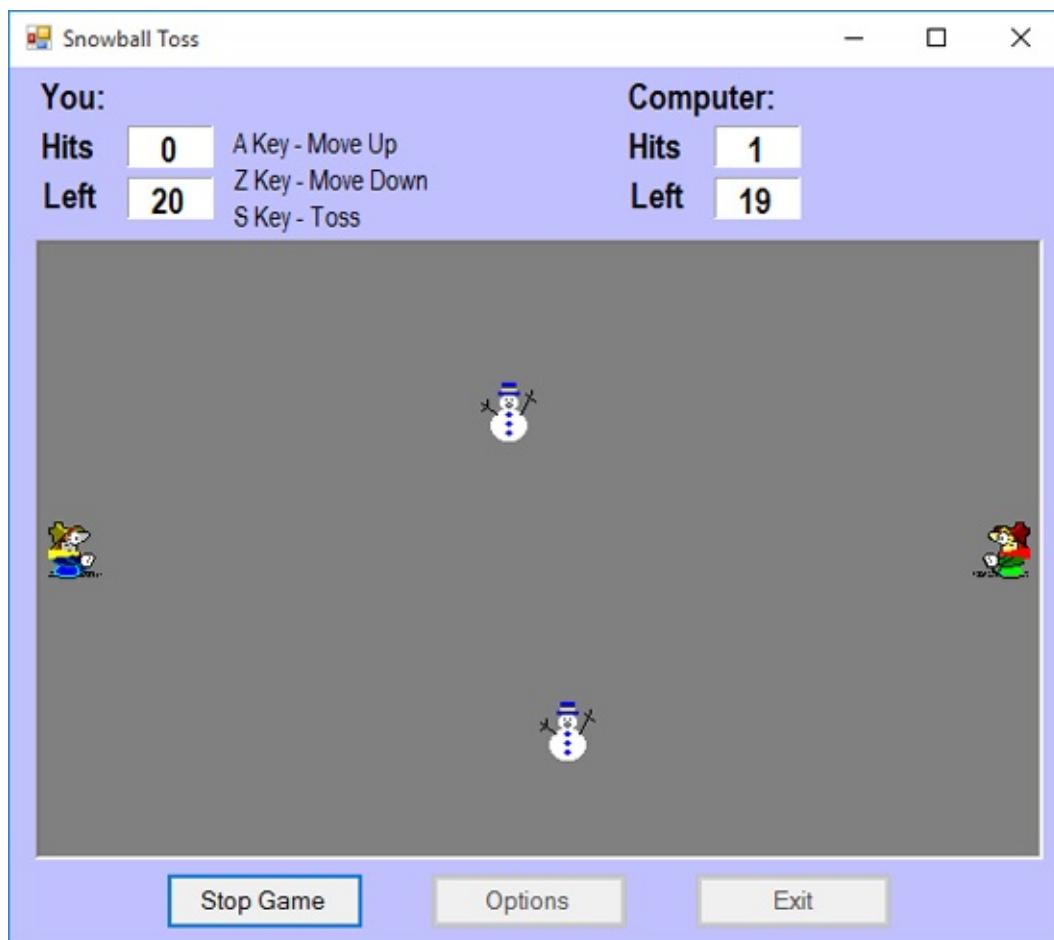
Run the project (press <F5>). When the form appears, click **Options**. You should see:



In the snowball toss game, you can have two players competing against each other or one player against the computer. For now, choose **One Player**. With a single player, you can also choose **Difficulty** (setting the intelligence level of the

computer) – select **Easiest**. Click **OK** to make the **Options** window disappear. Then, click **New Game**.

The game screen shows the two snowball tossing characters, one on each side of the screen (the players are identified by the label controls at the top of the form). Also shown are the player scores (**Hits**) and displays showing how many snowballs are left. The idea of the game is to move your ‘tosser’ up and down the screen, trying to hit the other player with a snowball. Zombie snowmen move in strange ways through the middle of the screen to act as cover and deflect some tosses:

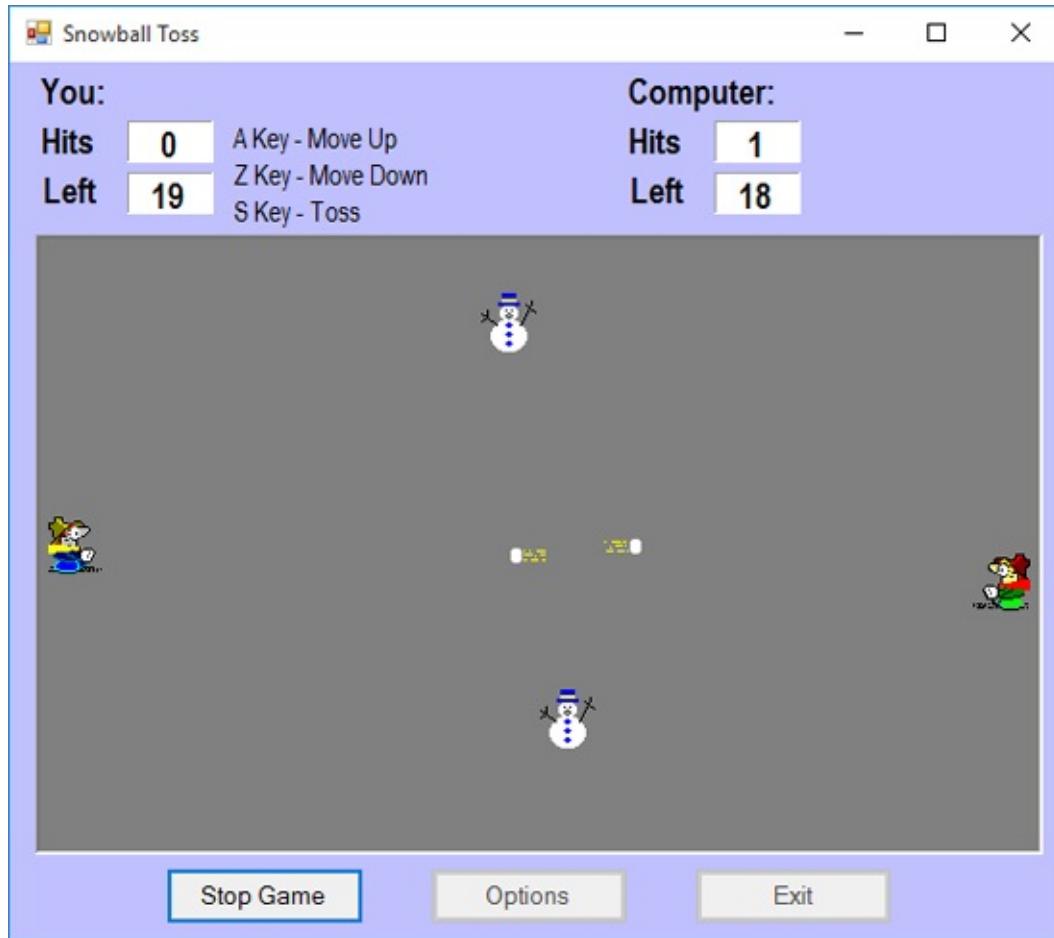


Watch out – the computer may make a toss at you.

Control of the players is via the keyboard. Player 1 (and the player when playing against the computer) uses the **A** key to move up, the **Z** key to move down, and the **S** key to toss a snowball. Player 2 uses the **K** key to move up, the **M** key to

move down, and the **J** key to toss. These instructions are shown on the form. The game ends when all the snowballs have been thrown or when the **Stop** button is clicked.

Try moving your player up and down using the **A** and **Z** keys. When you want, take a toss at the other player by pressing the **S** key. You should hear a ‘throwing’ sound. Here’s both players making a toss:



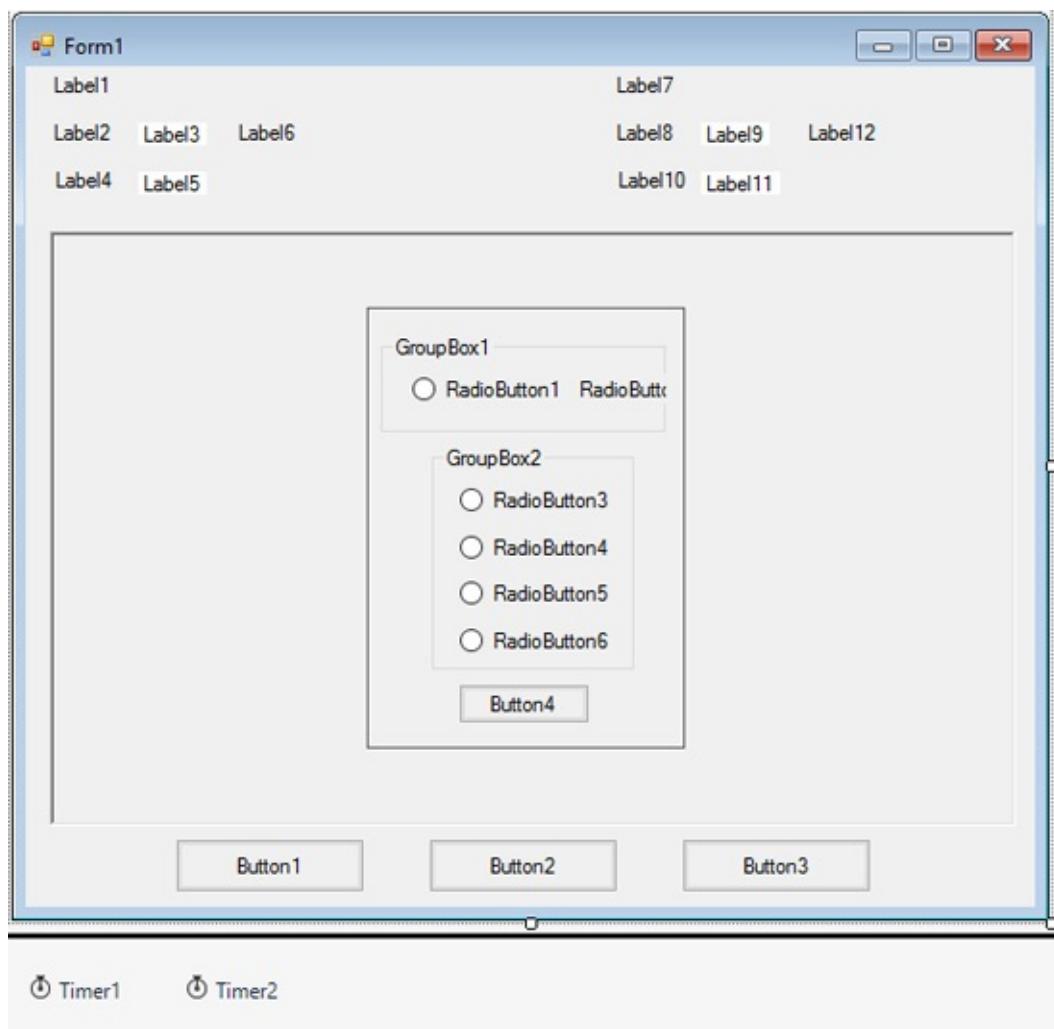
The snowball will move across the screen until it hits something or flies off the side of the form. If you hit the other player (resulting in an ouch sound), you earn one point. The player with the most points when the game stops is the winner.

That’s all there is to this game. Conceptually, it is very simple. Just throw snowballs at each other until you’re out of snowballs. Though simple, there are many topics we need to discuss to build the project.

The project will be built in several stages. We discuss **form design**. We discuss the controls used to build the form and establish initial properties. We show how to configure the form based on selected options. And, we address **code design**. We discuss several areas of game programming: animation, keyboard events, collision detection and sounds. Lastly, we look at how to give the computer intelligence in making decisions needed to play the game.

# Snowball Toss Game Form Design

We can begin building the snowball toss game project. Let's build the form. Start a new project in Visual C#. Place 12 labels, a large panel control and three button controls on the form. In the panel control, place a smaller panel with two group box controls and a button control. In the first group box, place 2 radio buttons; in the second, place 4 radio buttons. Add two timer controls to the project. Resize and position controls until the form looks similar to this:



All label controls are used for titling and providing scoring and game play information. The large panel (**panel1**) is the game field. The three button controls are used to start/stop a game, set options and exit the program. One timer control is used to govern the game animation and one is used to represent

the computer's decision process. The second panel control (**panel2**) is used to select game options. One group box holds radio buttons used to select the number of players. One group box holds radio buttons to select the game difficulty level, when playing against the computer. A small button is used to close the options panel. Default properties are set for a one player game with easiest difficulty.

Set the control properties using the properties window:

**Form1** Form:

<b>Property Name</b>	<b>Property Value</b>
Name	frmSnowball
BackColor	LightBlue
BorderStyle	FixedSingle
KeyPreview	True
StartPosition	CenterScreen
Text	Snowball Toss

**label1** Label:

<b>Property Name</b>	<b>Property Value</b>
Name	lblPlayer1
Text	You:
Font	Arial Narrow, Bold, Size 14

**label2** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Hits
Font	Arial Narrow, Bold, Size 14

**label3** Label:

<b>Property Name</b>	<b>Property Value</b>
Name	lblPlayer1Hits
AutoSize	False

BackColor	White
BorderStyle	Fixed3D
Text	0
TextAlign	MiddleCenter
Font	Arial Narrow, Bold, Size 14

**label4** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Left
Font	Arial Narrow, Bold, Size 14

**label5** Label:

<b>Property Name</b>	<b>Property Value</b>
Name	lblPlayer1Left
AutoSize	False
BackColor	White
BorderStyle	Fixed3D
Text	20
TextAlign	MiddleCenter
Font	Arial Narrow, Bold, Size 14

**label6** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	A Key – Move Up Z Key – Move Down S Key - Toss
Font	Arial Narrow, Size 11

**label7** Label:

<b>Property Name</b>	<b>Property Value</b>
Name	lblPlayer2
Text	Computer:
Font	Arial Narrow, Bold, Size 14

**label8** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Hits
Font	Arial Narrow, Bold, Size 14

**label9** Label:

<b>Property Name</b>	<b>Property Value</b>
Name	lblPlayer2Hits
AutoSize	False
BackColor	White
BorderStyle	Fixed3D
Text	0
TextAlign	MiddleCenter
Font	Arial Narrow, Bold, Size 14

**label10** Label:

<b>Property Name</b>	<b>Property Value</b>
Text	Left
Font	Arial Narrow, Bold, Size 14

**label11** Label:

<b>Property Name</b>	<b>Property Value</b>
Name	lblPlayer2Left
AutoSize	False
BackColor	White
BorderStyle	Fixed3D
Text	20
TextAlign	MiddleCenter
Font	Arial Narrow, Bold, Size 14

**label12** Label:

<b>Property Name</b>	<b>Property Value</b>

Name	lblPlayer2Instructions
Text	K Key – Move Up M Key – Move Down J Key - Toss
Font	Arial Narrow, Size 11
Visible	False

**panel1** Panel:

<b>Property Name</b>	<b>Property Value</b>
Name	pnlSnow
BackColor	Gray
BorderStyle	Fixed3D

**button1** Button:

<b>Property Name</b>	<b>Property Value</b>
Name	btnGame
Text	New Game
Font Size	10
TabStop	False

**button2** Button:

<b>Property Name</b>	<b>Property Value</b>
Name	btnOptions
Text	Options
Font Size	10
TabStop	False

**button3** Button:

<b>Property Name</b>	<b>Property Value</b>
Name	btnExit
Text	Exit
Font Size	10
TabStop	False

**timer1** Timer:

<b>Property Name</b>	<b>Property Value</b>
Name	timComputer

**timer2** Timer:

<b>Property Name</b>	<b>Property Value</b>
Name	timGame
Interval	50

**panel2** Panel:

<b>Property Name</b>	<b>Property Value</b>
Name	pnlOptions
BackColor	Light Yellow
BorderStyle	FixedSingle
Visible	False

**groupBox1** Group Box:

<b>Property Name</b>	<b>Property Value</b>
Name	grpPlayers
Text	Number of Players
Font Size	10
Font Style	Bold

**radioButton1** Radio Button:

<b>Property Name</b>	<b>Property Value</b>
Name	rdoOnePlayer
Text	One
Checked	True
Font Size	10

**radioButton2** Radio Button:

<b>Property Name</b>	<b>Property Value</b>

Name	rdoTwoPlayers
Text	Two
Font Size	10

**groupBox2** Group Box:

<b>Property Name</b>	<b>Property Value</b>
Name	grpDifficulty
Text	Difficulty
Font Size	10
Font Style	Bold

**radioButton3** Radio Button:

<b>Property Name</b>	<b>Property Value</b>
Name	rdoEasiest
Text	Easiest
Checked	True
Font Size	10

**radioButton4** Radio Button:

<b>Property Name</b>	<b>Property Value</b>
Name	rdoEasy
Text	Easy
Font Size	10

**radioButton5** Radio Button:

<b>Property Name</b>	<b>Property Value</b>
Name	rdoHard
Text	Hard
Font Size	10

**radioButton6** Radio Button:

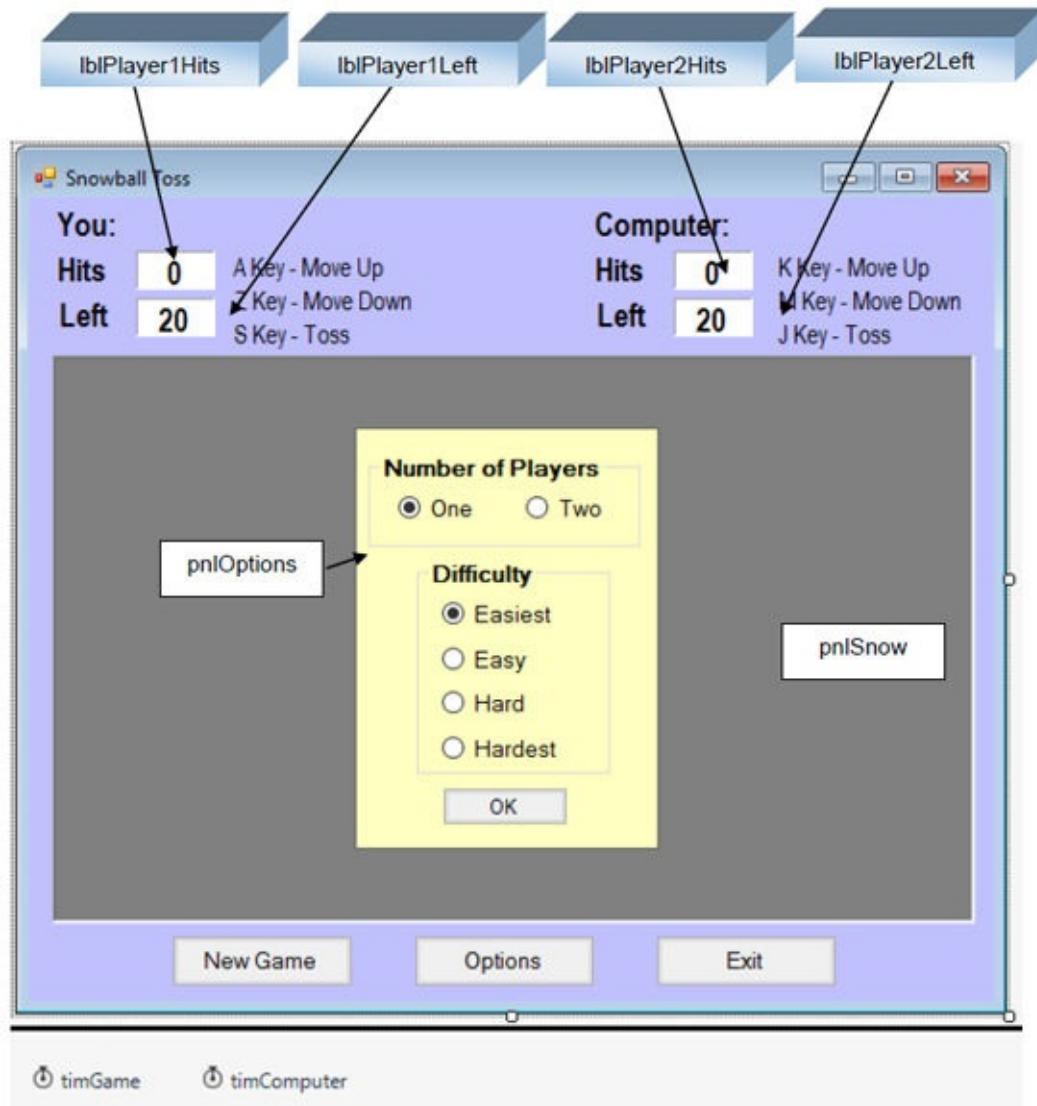
<b>Property Name</b>	<b>Property Value</b>
----------------------	-----------------------

Name	rdoHardest
Text	Hardest
Font Size	10

**button4** Button:

Property Name	Property Value
Name	btnOK
Text	OK
Font Size	10

When done setting properties, my form looks like this:



We don't label all of the controls. Most should be obvious.

We now begin writing project code. We first write code that establishes form status based on selected game options.

# Form Design – Choosing Options

When the game begins, a user usually chooses options (number of players and difficulty level). Based on these choices, the form will display different information. When the user clicks the **Options** button (**btnOptions**), the following occurs:

- Disable **btnGame**.
- Disable **btnOptions**.
- Disable **btnExit**.
- Make **pnlOptions** visible.

The code for these steps goes in the **btnOptions Click** event:

```
private void btnOptions_Click(object sender, EventArgs e)
{
    btnGame.Enabled = false;
    btnOptions.Enabled = false;
    btnExit.Enabled = false;
    pnlOptions.Visible = true;
}
```

Add this method.

Once the options panel is displayed, the user can choose one or two players and game difficulty (if playing against the computer). We will use the **int** variable **numberPlayers** to store the number of players and the **int** variable **difficulty** to store the selected difficulty. Add these form level variable declarations to the project:

```
int numberPlayers, difficulty;
```

Add the **frmSnowball Load** event to the project to initialize these variables to default values:

```

private void frmSnowball_Load(object sender, EventArgs e)
{
    numberPlayers = 1;
    difficulty = 1;
}

```

If the one player (**rdoOnePlayer** button) is selected, you play against the computer and the following happens:

- Set **numberPlayers** to 1.
- Set **lblPlayer1** Text property to **You**:
- Set **lblPlayer2** Text property to **Computer**:
- Make **lblPlayer2Instructions** label control invisible.
- Enable **grpDifficulty**.

If the two player (**rdoTwoPlayers** button) is selected, you play against another person and the following happens:

- Set **numberPlayers** to 2.
- Set **lblPlayer1** Text property to **Player 1**:
- Set **lblPlayer2** Text property to **Player 2**:
- Make **lblPlayer2Instructions** label control visible.
- Disable **grpDifficulty**.

Notice game difficulty is only selected when playing against the computer (one player).

The code to implement these steps is placed in a method named **rdoPlayers\_Click** (which handles the **Click** event for both **rdoOnePlayer** and **rdoTwoPlayers**):

```

private void rdoPlayers_Click(object sender, EventArgs e)
{
    RadioButton buttonClicked;
    buttonClicked = (RadioButton) sender;
}

```

```

if (buttonClicked.Text == "One")
{
    numberPlayers = 1;
    lblPlayer1.Text = "You:";
    lblPlayer2.Text = "Computer:";
    lblPlayer2Instructions.Visible = false;
    grpDifficulty.Enabled = true;
}
else
{
    numberPlayers = 2;
    lblPlayer1.Text = "Player 1:";
    lblPlayer2.Text = "Player 2:";
    lblPlayer2Instructions.Visible = true;
    grpDifficulty.Enabled = false;
}
}

```

Add this method to your project.

Similarly, a method named **rdoDifficulty\_Click** handles the **Click** event for the four radio buttons in **grpDifficulty**. Based on which button is clicked, the **Difficulty** variable is set:

```

private void rdoDifficulty_Click(object sender, EventArgs e)
{
    RadioButton buttonClicked;
    buttonClicked = (RadioButton) sender;
    switch (buttonClicked.Text)
    {
        case "Easiest":
            difficulty = 1;
            break;
    }
}

```

```
        case "Easy":  
            difficulty = 2;  
            break;  
        case "Hard":  
            difficulty = 3;  
            break;  
        case "Hardest":  
            difficulty = 4;  
            break;  
    }  
}
```

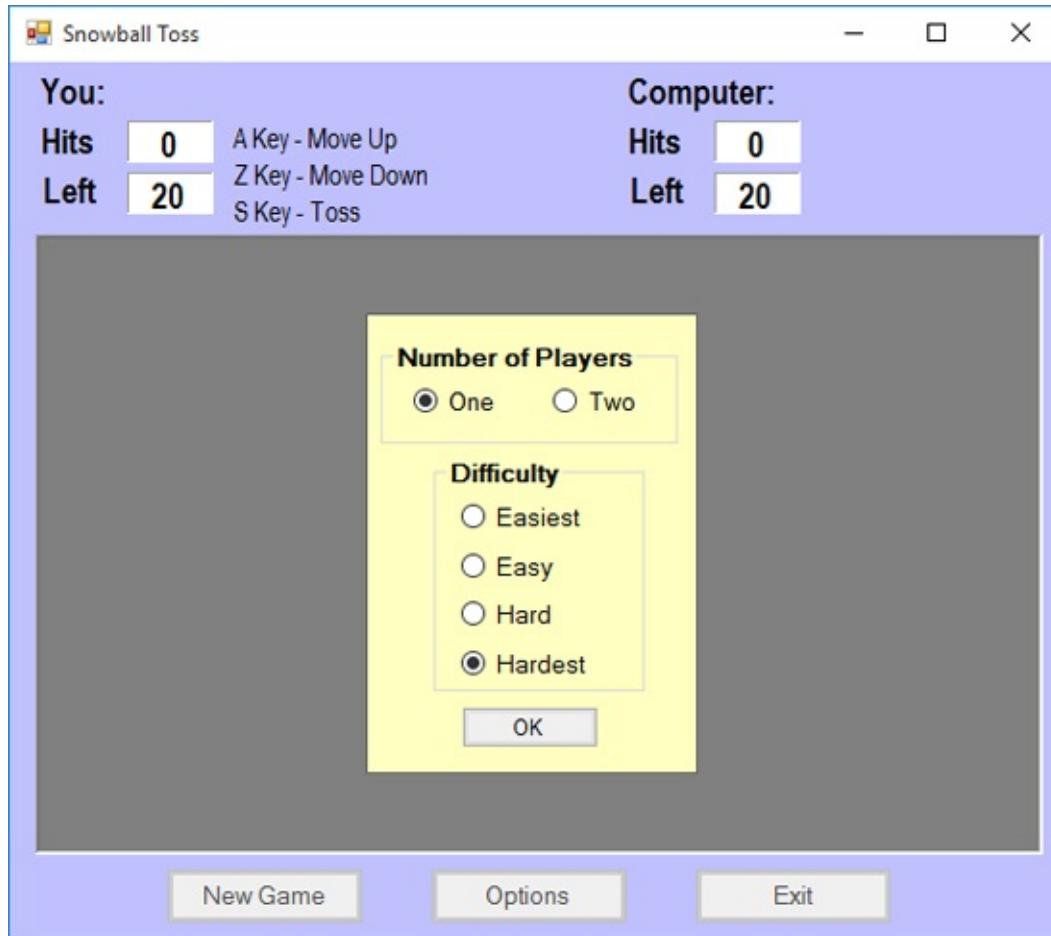
Add this method to your project.

Once the user has selected options, he clicks the **OK** button (**btnOK**) to close out the options panel. The code in the **btnOK Click** event is just the reverse of the code in the **btnOptions Click** event (reversing the Boolean properties):

```
private void btnOK_Click(object sender, EventArgs e)  
{  
    btnGame.Enabled = true;  
    btnOptions.Enabled = true;  
    btnExit.Enabled = true;  
    pnlOptions.Visible = false;  
}
```

Add this final options method to your project. At this point, game play can begin.

Save and run the project. The game screen will appear with default values (one player, easiest difficulty). Click **Options** to make the options panel appear:



Changing the difficulty will not change the form appearance. Changing the number of players will. Choose two players – the header information should change to reflect two players and the **Difficulty** group box should be disabled. Make sure everything works as planned.

Each time you play the game, you would like the options you used the last time you played the game to be “pre-selected.” A configuration file can handle this task.

# Code Design – Configuration File

Since we will be writing/reading files, we need to add this using statement at the top of the code window:

```
using System.IO;
```

The configuration file will hold two pieces of information: the number of players (**numberPlayers**) and the difficulty (**difficulty**). First, let's develop the code to write the configuration file to disk when the program ends. This code goes in the **FormClosing** event:

```
private void frmSnowball_FormClosing(object sender,  
FormClosingEventArgs e)  
{  
    StreamWriter outputFile = new  
    StreamWriter(Application.StartupPath + "\\snowball.ini");  
    outputFile.WriteLine(numberPlayers);  
    outputFile.WriteLine(difficulty);  
    outputFile.Close();  
}
```

As always, the configuration file is in the project's **Bin\Debug** folder. Add this method to your project.

Add this code to the **btnExit Click** event method, so we can stop the project and write the file:

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

The configuration file is opened and read in the form **Load** method. Based on

the values read, we then simulate clicks on the appropriate radio buttons to choose options. The one tricky part in this method is that we can only simulate these clicks if the buttons are visible and enabled – we do this in code. The modified form **Load** method that opens the configuration file and chooses the appropriate radio buttons is (changes are shaded):

```
private void frmSnowball_Load(object sender, EventArgs e)
{
    try
    {
        StreamReader inputFile = new
        StreamReader(Application.StartupPath + "\\snowball.ini");
        numberPlayers =
        Convert.ToInt32(inputFile.ReadLine());
        difficulty =
        Convert.ToInt32(inputFile.ReadLine());
        inputFile.Close();
    }
    catch
    {
        numberPlayers = 1;
        difficulty = 1;
    }
    // make form visible to 'click' options
    this.Show();
    pnlOptions.Visible = true;
    if (difficulty == 1)
        rdoEasiest.PerformClick();
    else if (difficulty == 2)
        rdoEasy.PerformClick();
    else if (difficulty == 3)
        rdoHard.PerformClick();
    else
}
```

```
    rdoHardest.PerformClick();
    if (numberPlayers == 1)
        rdoOnePlayer.PerformClick();
    else
        rdoTwoPlayers.PerformClick();
    pnlOptions.Visible = false;
}
```

Notice we use default values if the configuration file cannot be opened. Notice, too, how we make the form (**this**) and panel (**pnlOptions**) appear before choosing the options. We choose the difficulty option first, since this option is not enabled if the two player optioned is selected. Once options are selected, the panel's **Visible** property is set back to **false**.

Save and run the project. Choose some options. Make sure the form is properly configured after choosing the options. Stop the project – click **Exit** or click the **X** in the upper right corner of the form. Run the project again to make sure your last set of selected options is still selected and the form looks correct. Before stopping the program for the last time, make sure the **Two Player** option is selected. We will use this for most of our design work.

We're now ready to start programming the graphics features of the snowball game – moving the tossers up and down the screen, throwing snowballs, and moving the zombie snowmen across the screen.

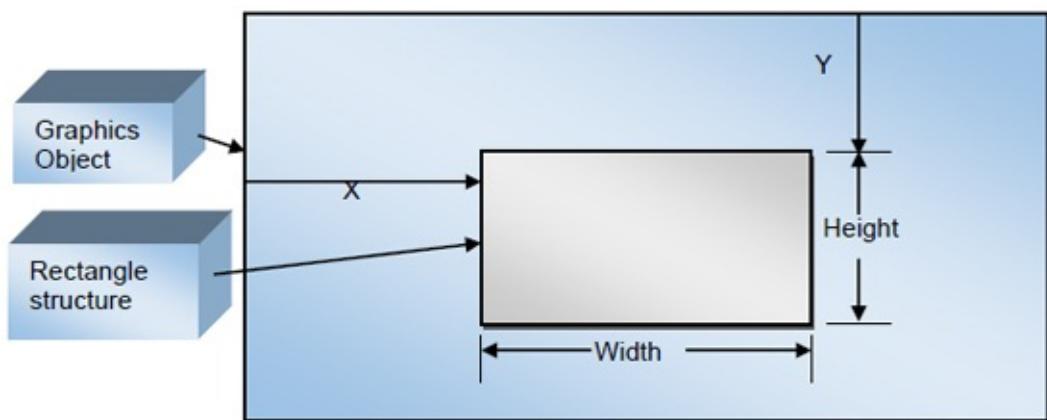
# Animation with Visual C#

Programming animated games in Visual C# requires a specific set of skills. We need to know how to develop a graphic image, how to move (animate) that image and how to see if one image collides with another image. We also want to add sounds to our games. As we build the snowball toss game, we will discuss these new skills. We start with **animation**.

Animating an image in a graphic object involves three steps: (1) erase the image in its current location, (2) determine the new location, (3) move the image to that location. The image region is rectangular. We use the Visual C# **Rectangle** structure to specify such regions. The properties for this structure we will use are:

<b>X</b>	Gets or sets the x-coordinate of the upper-left corner of the rectangle
<b>Y</b>	Gets or sets the y-coordinate of the upper-left corner of the rectangle
<b>Height</b>	Gets or sets the width of the rectangle
<b>Width</b>	Gets or sets the height of the rectangle

The **X** and **Y** values are relative to the graphics object. A diagram shows everything:



**X**, **Y**, **Width** and **Height** can be changed at run-time.

There are two steps involved in creating a **rectangle structure**. We first declare the structure using the standard statement:

```
Rectangle myRectangle;
```

Placement of this statement depends on scope. Place it in a method for method level scope. Place it with other form level declarations for form level scope. Once declared, the structure is created using the **Rectangle** constructor:

```
myRectangle = new Rectangle(X, Y, Width, Height);
```

where **X**, **Y**, **Width** and **Height** are the desired integer measurements (in pixels).

You can move and resize the rectangle in code, by changing any of four properties:

```
myRectangle.X = newX;  
myRectangle.Y = newY;  
myRectangle.Width = newWidth;  
myRectangle.Height = newHeight;
```

where **newX**, **newY**, **newWidth**, and **newHeight** represent new values for the respective properties.

An image is drawn to a graphics object using the **DrawImage** graphics method. Before using **DrawImage**, you need two things: a **Graphics** object to draw to and an **Image** object to draw. The graphics and image objects are declared in the usual manner:

```
Graphics myGraphics;  
Image myImage;
```

We create the graphics object (assume **myObject** is the host object):

```
myGraphics = myObject.CreateGraphics();
```

The **Image** object is usually created from a graphics file:

```
myImage = Image.FromFile(FileName);
```

where **FileName** is a complete path to the graphics file describing the image to draw. At this point, we can draw **myImage** in **myGraphics**.

The **DrawImage** method is:

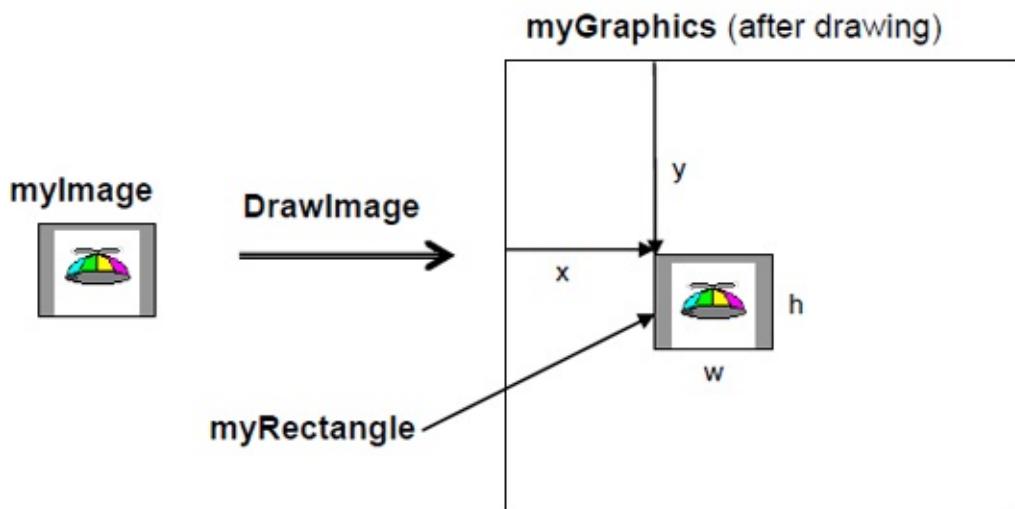
```
myGraphics.DrawImage(myImage, myRectangle);
```

where **myRectangle** is a rectangle structure that positions **myImage** within **myGraphics**. **myRectangle** is specified by **x** the horizontal position, **y** the vertical position, the width **w** and height **h**:

```
myRectangle = new Rectangle(x, y, w, h);
```

The width and height can be the original image size or scaled up or down. It's your choice.

A picture illustrates what's going on with **DrawImage**:



Note how the transfer of the rectangular region occurs. Successive image transfers gives the impression of motion, or animation. Prior to moving an image from one location to the next, it must be erased. One way to do this would be to clear the entire graphics object. This would work, but would require repositioning every image, even ones that haven't moved. This would be a slow,

tedious and unnecessary process.

We will take a more precise approach to erasure. Instead of erasing the entire panel before moving an image, we will only erase the rectangular region previously occupied by the image. To do this, we will use the **FillRectangle** graphics method, a new concept. If applied to a graphics object named **myGraphics**, the syntax is:

```
myGraphics.FillRectangle(myBrush, x, y, w, h);
```

This line of code will “paint” a rectangular region located at **(x, y)**, **w** wide, and **h** high with a brush object (**myBrush**). To erase an image, the brush needs to be a solid brush, the same background color as the object hosting the graphics object.

With our newly-gained knowledge of the **Rectangle** structure, the **DrawImage** graphics method and the **FillRectangle** graphics method, we can summarize the steps needed to move (or animate) an image (**myImage**) in a graphics object named **myGraphics** hosted in a control named **myObject**. Assume the image is currently at location **(x, y)** and is **w** by **h** in size. Assume we want to move the image to a new location **(x + dx, y + dy)**. The steps are:

- Create **Rectangle** structure to describe image region:

```
myRectangle = new Rectangle(x, y, w, h);
```

- Erase image at current location using:

```
myGraphics.FillRectangle(myBrush, myRectangle);
```

where **myBrush** is a solid brush with color **myObject.BackColor**.

- Update **Rectangle** structure to new location:

```
myRectangle.X += dx;  
myRectangle.Y += dy;
```

- Draw image at new location:

```
myGraphics.DrawImage(myImage, myRectangle);
```

Successive application of each of these steps for each image in our graphics region results in a nice smooth animated motion.

We're ready to start writing code to animate our snowball toss game, but first we need to answer one question that might be lingering. In describing the **DrawImage** method, we said the images we use are loaded from files. Your question might be – where do these files come from? The answer is you either need to find them from some source (the Internet is a good place to look) or create them yourself.

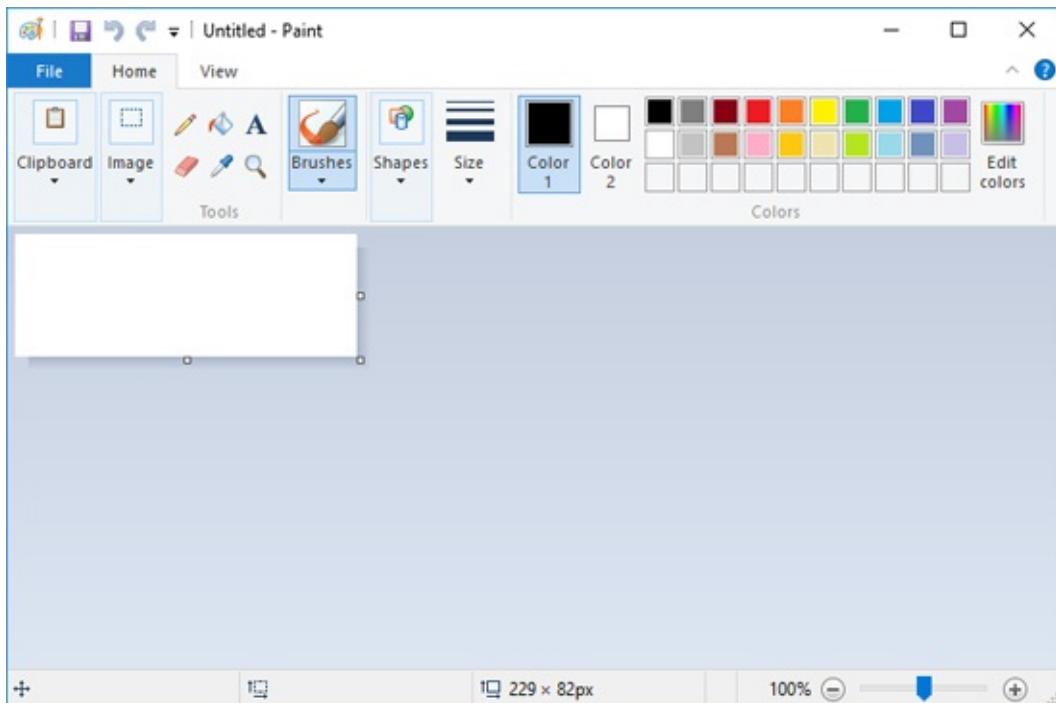
To create animation images, we use the Microsoft **Paint** program that ships with Windows. Draw your picture and save it as a icon file. Or you could use one of many available commercial paintbrush programs.

# Drawing Images with Paint

A small icon appears in the upper left corner of a form in a Visual Basic project. Icons are used in several places in Visual Basic applications: to represent files in Windows Explorer, to represent programs in the Programs menu, to represent programs on the desktop and to identify an application removal tool. Icons are used throughout projects. We can create our own icons and add them to a form. In fact, we do that in the Appendix to these notes, where we discuss distributing a Visual Basic project.

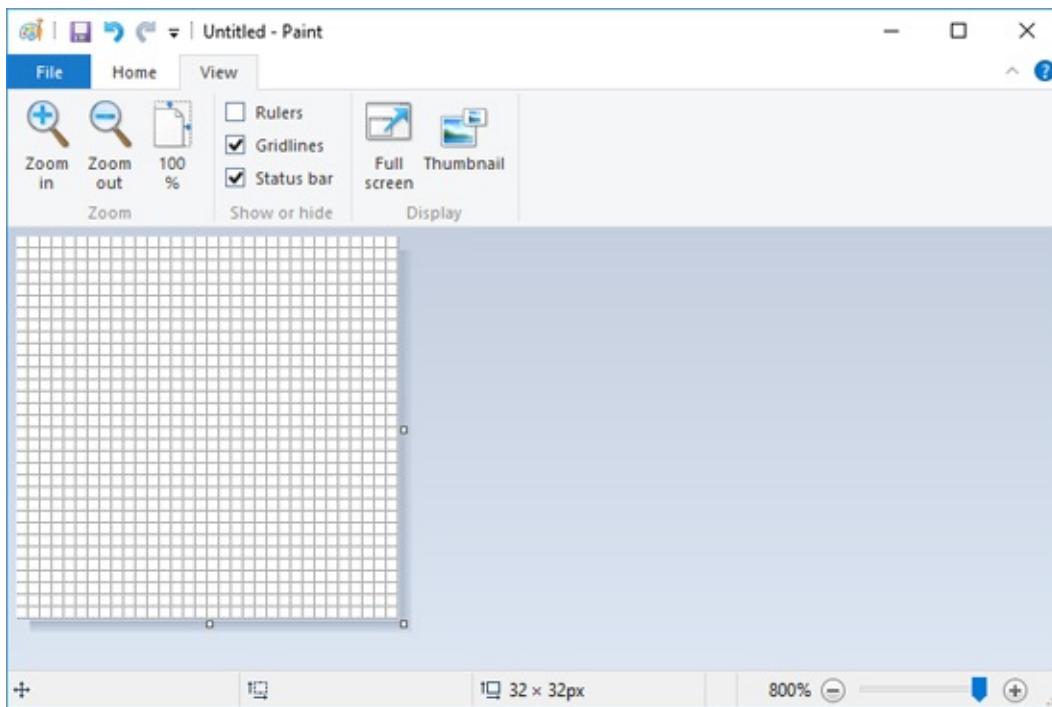
For now, we just want to look at a tool (Microsoft **Paint**) used to create such icons (a file with an **ico** extension) and use it to create the images we need for our snowball toss program. Icons are simply special cases of bitmap files that are 32 bits by 32 bits in size. Their size makes them very useful in games such as this.

To start **Paint**, click the **Start** menu, then **All apps**, then **Paint**. You should see:



First, resize the image to 32 x 32 pixels. Then, use the magnifying tool to make the image as large as possible. Finally, add a grid to the graphic. When done, I

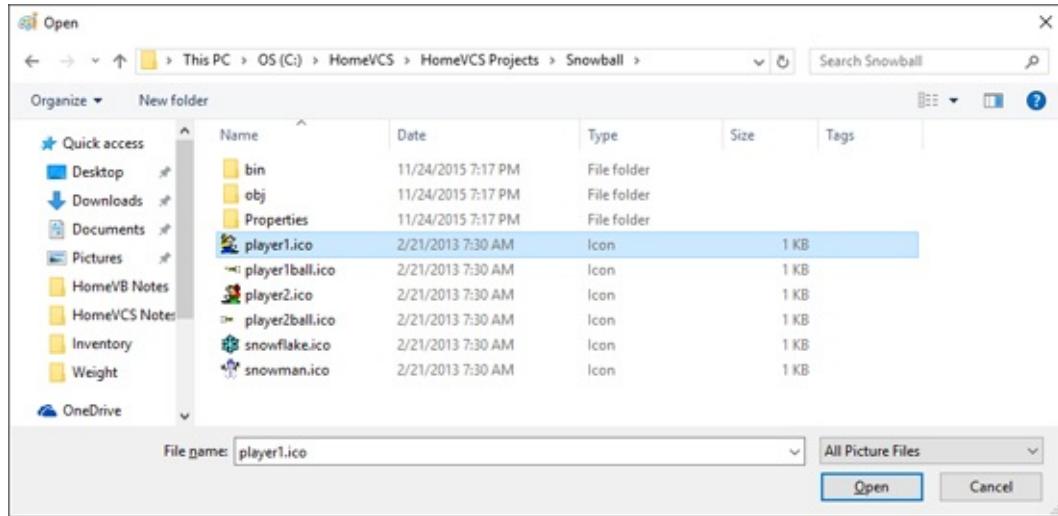
see:



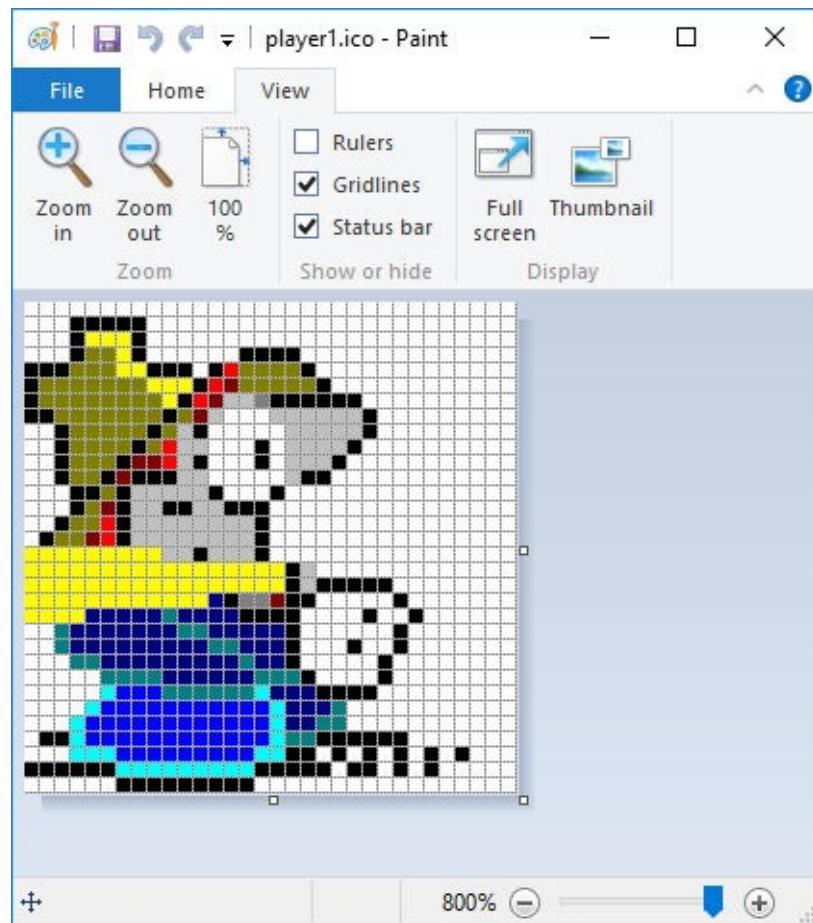
The basic idea of **Paint** is to draw an icon in the large 32 x 32 grid displayed. You can draw single points, lines, open rectangles and ovals, and filled rectangles and ovals. Various colors are available with simple mouse clicks. Once completed, the icon file can be saved for attaching to a form or use within an application.

We won't go into a lot of detail on using the **Paint** program here - I just want you to know it exists and can be used to create and save icon files. Its use is fairly intuitive. Consult the help (click **Help** in the menu) that comes with the program for details.

All graphics used in the snowball toss game were created using **Paint**. These files are included in the **HomeVCS\HomeVCS Projects\Snowball** folder. Let's look at the files used to represent one of the two players. Start **Paint**. Choose **Open** under the **File** menu. An **Open Icon** dialog will appear. Navigate to the above folder. There are two files used to represent the players – **player1.ico** and **player2.ico**. Open **player1.ico** as shown:



This cute little guy will appear (after magnifying and adding grid lines):



Notice you can get quite a lot of detail into a 32 x 32 space. I, personally, have no artistic talent. Someone drew all the graphics in this program for me.

Open the second player file if you like or look ahead at the graphics to represent snowmen and snowballs. Right now, let's look at how to display these little guys in the panel control on our form.

# Code Design – Sprite Class

We will have several images moving around in this project (players, snowballs, snowmen). This is a good place to try our object-oriented programming skills introduced in Chapter 11. We will build a class named **Sprite** to represent the two player objects.

Review the steps to add a class to your Visual C# project in Chapter 11. Name the added class **Sprite.cs**. These lines will be in the new file (between the **namespace** declaration) showing the default constructor:

```
public class Sprite
{
    public Sprite()
    {
    }
}
```

We will be doing drawing with our class, so add this using directive with the other directives at the top of the code window:

```
using System.Drawing;
```

We will use three properties to describe a **Sprite** object – **Graphic** (**Image** displayed), **Location** (**Rectangle** describing size and location), **IsVisible** (**bool** type saying whether object is visible). Add these declarations to the file, initializing **IsVisible** to **false**:

```
public class Sprite
{
    public Sprite()
    {
    }

    public Image Graphic;
```

```
public Rectangle Location;  
public bool IsVisible = false;  
}
```

We want to extend our class description by adding some **methods** to help with the animation task. Class methods allow objects to perform certain tasks. We will write three methods for our **Sprite** object, one that places it on a graphics object, one that moves it to a new location and one that erases it. These methods are added to the class description exactly like general methods are added to a form's code file. To add a method to a class description, you select a name and a type of information the method will return (if there is any returned value). Also determine any needed arguments for the method.

The method **Place** positions a **Sprite** object into a graphics object named **g** (passed as an argument):

```
public void Place(Graphics g)  
{  
    g.DrawImage(this.Graphic, this.Location);  
    this.IsVisible = true;  
}
```

Note the use of the keyword **this** to refer to the current object. Once the object is placed on **g**, we set the **IsVisible** property to **true**.

The method **Move** moves a **Sprite** object to a new location. The required arguments are **g**, the graphics object, **bc**, the brush used to erase the previous location, **dx**, the change in **x** location and **dy**, the change in **y** location:

```
public void Move(Graphics g, Brush bc, int dx, int dy)  
{  
    g.FillRectangle(bc, this.Location);  
    this.Location.X += dx;  
    this.Location.Y += dy;  
    g.DrawImage(this.Graphic, this.Location);  
}
```

The object is first erased. The location is then updated and the object is redrawn.

The method **Remove** erases a **Sprite** object from the graphics object **g** (using brush **bc**):

```
public void Remove(Graphics g, Brush bc)
{
    g.FillRectangle(bc, this.Location);
    this.Visible = false;
}
```

Add these three methods (**Place**, **Move**, **Remove**) to the **Sprite** class file to make the final version look like this (note we also show the **using** directives):

```
#region Using directives
using System;
using System.Collections.Generic;
using System.Text;
using System.Drawing;
#endregion
namespace Snowball
{
    public class Sprite
    {
        public Sprite()
        {

        }

        public Image Graphic;
        public Rectangle Location;
        public bool IsVisible = false;

        public void Place(Graphics g)
```

```

{
    g.DrawImage(this.Graphic, this.Location);
    this.Visible = true;
}

public void Move(Graphics g, Brush bc, int dx, int dy)
{
    g.FillRectangle(bc, this.Location);
    this.Location.X += dx;
    this.Location.Y += dy;
    g.DrawImage(this.Graphic, this.Location);
}

public void Remove(Graphics g, Brush bc)
{
    g.FillRectangle(bc, this.Location);
    this.Visible = false;
}
}

```

With this class, a **Sprite** object is created with:

```
Sprite mySprite;
```

It is constructed with:

```
mySprite = new Sprite();
```

Once constructed, the **Graphic** property can be set using the **Image.FromFile** method and the **Location** property equated to a **Rectangle** structure.

The **Sprite** object is placed in a graphics object (**myGraphics**) using:

```
mySprite.Place(myGraphics);
```

It is moved using:

```
mySprite.Move(myGraphics, myBrush, dx, dy);
```

where **myBrush** is a solid brush the same color as the background of the graphics object, **dx** is how much you want to move the object horizontally and **dy** how much you want to move the object vertically. And, the object is erased using:

```
mySprite.Erase(myGraphics, myBrush);
```

At long last, we have all the background needed to draw something on the panel control (**pnlSnow**) where we play the snowball toss game.

# Code Design – Start/Stop Game

We'll now write the code to start and stop the game. We first define some form level variables to play and control the game. Return to the **Snowball.cs** file. Add these declarations to the project:

```
Graphics myGraphics;  
Brush blankBrush;  
Sprite player1, player2;  
int player1Hits, player2Hits, player1Left, player2Left;  
const int maximumBalls = 20;
```

**myGraphics** is the graphics region used to play the game and **blankBrush** will be used to erase images. Two **Sprite** objects (**player1** and **player2**) represent the two ‘tossers’. **player1Hits** and **player2Hits** will keep track of how many successful snowball tosses each player has. **player1Left** and **player2Left** keep track of how many snowballs each player still has. A constant **maximumBalls** (you can change this if you want) sets the number of snowballs a player starts with.

We add some code to the **frmSnowball Load** event to initialize each of these variables. The modified method (changes are shaded, much unmodified code is not shown) is:

```
private void frmSnowball_Load(object sender, EventArgs e)  
{  
    .  
    .  
    .  
    pnlOptions.Visible = false;  
    player1Left = maximumBalls;  
    player2Left = maximumBalls;  
    lblPlayer1Left.Text = player1Left.ToString();  
    lblPlayer2Left.Text = player2Left.ToString();  
    // create sprites
```

```

myGraphics = pnlSnow.CreateGraphics();
blankBrush = new SolidBrush(pnlSnow.BackColor);
player1 = new Sprite();
player2 = new Sprite();
player1.Graphic =
Image.FromFile(Application.StartupPath + "\\player1.ico");
player2.Graphic =
Image.FromFile(Application.StartupPath + "\\player2.ico");
btnGame.Focus();
}

```

Note the code to create the two **Sprite** objects and initialize the **Graphic** properties. Copy the **player1.ico** and **player2.ico** graphics files into your project's **Bin\Debug** folder or this code will not work.

At this point, the user can click **New Game (btnGame)** to start a game. The following preliminary steps should happen (more steps will be added later):

- Clear graphics object.
- Change **btnGame Text** property to **Stop Game**.
- Disable **btnOptions**.
- Disable **btnExit**.
- Reset **player1Hits** and **player2Hits** to **0**.
- Reset **player1Left** and **player2Left** to **maximumBalls**.
- Place **player1** and **player2** objects in initial positions (use the **Place** method of the **Sprite** class).

The code behind these steps is straightforward. The **player1** object will be centered vertically in **pnlSnow** (the graphics object) near the left edge. The **player2** object will be centered vertically near the right edge.

This same button (**btnGame**) is used to stop a game. When a user clicks the button (when **Stop Game** is displayed), the following should happen:

- Remove **player1** and **player2** objects (use **Sprite** object **Remove**

method).

- Change **btnGame** Text property to **Start Game**.
- Enable **btnOptions**.
- Enable **btnExit**.
- Write **Game Over** message.

The code for all these steps is placed in the **btnGame Click** event method. That code is:

```
private void btnGame_Click(object sender, EventArgs e)
{
    if (btnGame.Text == "New Game")
    {
        myGraphics.Clear(pnlSnow.BackColor);
        btnGame.Text = "Stop Game";
        btnOptions.Enabled = false;
        btnExit.Enabled = false;
        player1Hits = 0;
        player2Hits = 0;
        lblPlayer1Hits.Text = "0";
        lblPlayer2Hits.Text = "0";
        player1Left = maximumBalls;
        player2Left = maximumBalls;
        lblPlayer1Left.Text = player1Left.ToString();
        lblPlayer2Left.Text = player2Left.ToString();
        player1.Location = new Rectangle(5, (int)(0.5 *
pnlSnow.ClientSize.Height - 0.5 * player1.Graphic.Height),
player1.Graphic.Width, player1.Graphic.Height);

        player2.Location = new Rectangle(pnlSnow.ClientSize.Width -
player2.Graphic.Width - 5, (int)(0.5 * pnlSnow.ClientSize.Height - 0.5 *
player2.Graphic.Height), player2.Graphic.Width,
player2.Graphic.Height);

        player1.Place(myGraphics);
```

```
    player2.Place(myGraphics);
}
else
{
    player1.Remove(myGraphics, blankBrush);
    player2.Remove(myGraphics, blankBrush);
    btnGame.Text = "New Game";
    btnOptions.Enabled = true;
    btnExit.Enabled = true;
    myGraphics.DrawString("Game Over", new Font("Arial", 24),
Brushes.Yellow, 180, 140);
}
}
```

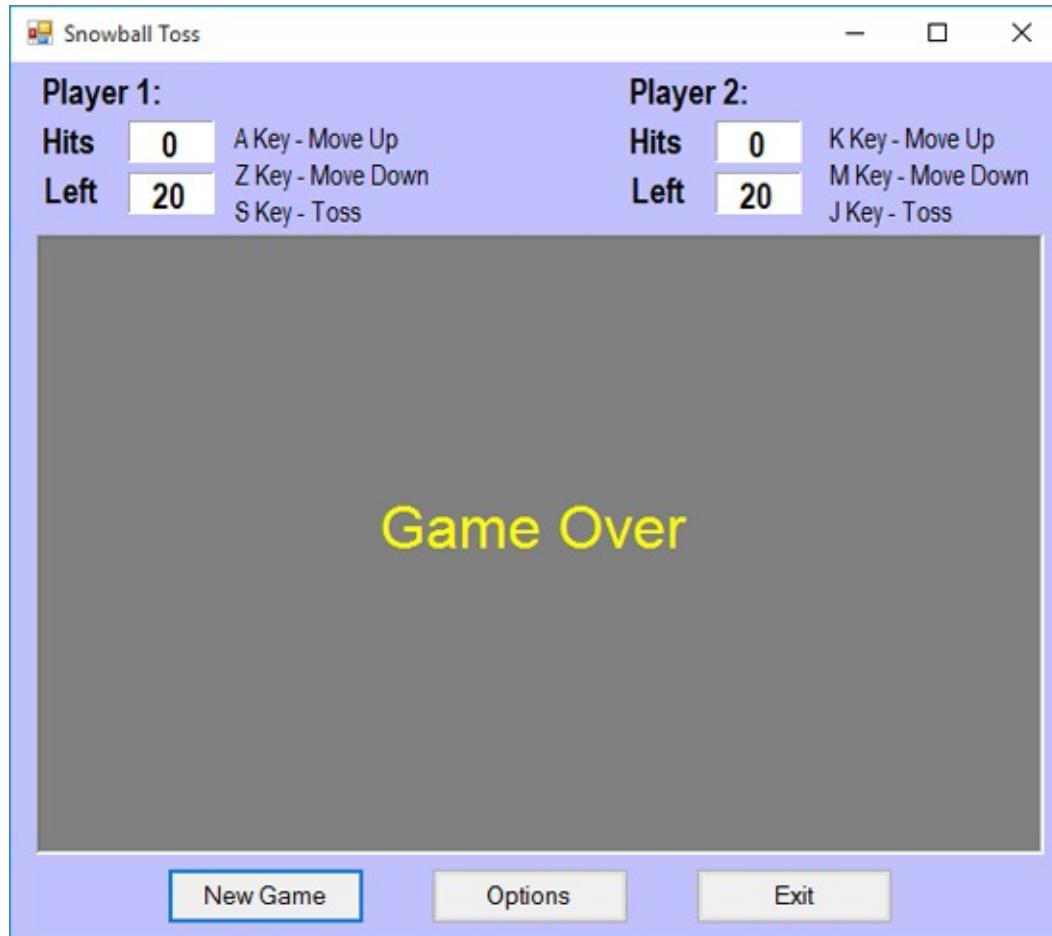
Add this method to your project. The positioning of the **Game Over** message was obtained with trial and error.

Save and run the project. Click **New Game** and the two guys should appear:



The game is ready to play – once we add the capability of moving the guys and throwing snowballs. It took a lot of work to get this far. We had a lot of new information to learn about animation. Things should progress a little faster from now on.

Before leaving, click **Stop Game** to make sure the stop method works and **Game Over** appears:



At this point, you have the option to start a new game, change options or exit.

# Code Design – Moving the Tossers

We need the capability to move our snowball tossers up and down in the graphics region. We choose to use the keyboard for control, using the **KeyDown** event. This event is similar to the **KeyPress** event we've used before, but is more flexible. It allows detecting typed keys, along with the status of keys like the Alt, Ctrl, Shift, cursor control and other control keys. The form of the **KeyDown** event method for a form (**Form**) is:

```
private void Form_KeyDown(object sender, KeyEventArgs e)
{
    .
}
```

The arguments are:

<b>sender</b>	Control active when key pressed to invoke method
<b>e</b>	Event handler revealing which key was pressed and status of certain control keys when key was pressed

We are interested in several properties of the event handler **e**:

<b>e.Alt</b>	Boolean property that indicates if Alt key is pressed
<b>e.Control</b>	Boolean property that indicates if Ctrl key is pressed
<b>e.KeyCode</b>	Gives the key code for the key pressed
<b>e.Shift</b>	Boolean property that indicates if either Shift key is pressed

The **KeyCode** property is a member of the **Keys** class. There are values for every key that can be pressed (see on-line help). The Intellisense feature of the Visual C# IDE will provide **KeyCode** values when needed. The **KeyDown** event

usually has a **switch** structure with a **case** clause for each possible key (or key combination) expected.

In the snowball toss game, we choose the following keys to control player motion and tossing of snowballs (selected based on location on the keyboard):

<b>Player 1</b>	A – Move Up, Z – Move Down, S – Toss
<b>Player 2</b>	K – Move Up, M – Move Down, J – Toss

You can change these if you'd like.

Each time a movement key is pressed, we will move (using the **Sprite** class **Move** method) the player an amount **playerIncrement** (a value you can adjust if needed) if going down and an amount **-playerIncrement** if going up. Add a form level constant declaration for this value:

```
const int playerIncrement = 5;
```

You might wonder how I came up with this value. I tried several values finding one that resulted in smooth motion that wasn't too small or too large. Any time you program a game, you will have several adjustable parameters. There is no real science to setting values – just some guessing, trying and refining. Feel free to change any of the “built-in” values in the snowball toss game.

The code to move the tossers (we'll add throwing logic later) is placed in the form's **KeyDown** event method. The panel control does not support a **KeyDown** method. That code is:

```
private void frmSnowball_KeyDown(object sender, KeyEventArgs e)
{
    if (btnGame.Text == "New Game")
        return;
    switch (e.KeyCode)
    {
        case Keys.A:
            player1.Move(myGraphics, blankBrush, 0, -playerIncrement);
```

```

if (player1.Location.Y < 0)
    player1.Location.Y = 0;
break;
case Keys.Z:
    player1.Move(myGraphics, blankBrush, 0, playerIncrement);
    if (player1.Location.Y > pnlSnow.ClientSize.Height -
player1.Location.Height)
        player1.Location.Y = pnlSnow.ClientSize.Height -
player1.Location.Height;
    break;
case Keys.K:
    player2.Move(myGraphics, blankBrush, 0, -playerIncrement);
    if (player2.Location.Y < 0)
        player2.Location.Y = 0;
    break;
case Keys.M:
    player2.Move(myGraphics, blankBrush, 0, playerIncrement);
    if (player2.Location.Y > pnlSnow.ClientSize.Height -
player2.Location.Height)
        player2.Location.Y = pnlSnow.ClientSize.Height -
player2.Location.Height;
    break;
}
}

```

Notice we don't allow any key down events if we haven't started a game (that is, if **btnGame** is displaying **New Game**). Also notice that most of code is involved with insuring a player never leaves the playing field. Add this method to your project.

We want to make sure the form always has focus to intercept the movement keystrokes. To insure this, add this line:

```
this.Focus();
```

at the end of the **btnGame Click** method code used to start a new game.

Save and run the project. Select the **Two Players** option so you can see if both players can move. Click **New Game**. Press the **A** and **Z** keys to move **Player 1** (the guy on the left) and press the **K** and **M** keys to move **Player 2** (the guy on the right). Make sure they can't move off the top or bottom of the panel control.

Before stopping, reduce the form to an icon, then restore the form. The little guys will be gone. We do not have persistent graphics in this project. We could add them (using a **Paint** event) if we wanted. But, we don't really need them. The little guys are redrawn every time they are moved and later we will see the same is true for any snowballs we throw or any of the zombie snowmen moving around. So, let's start throwing some snowballs.

# Code Design – MovingSprite Class

We want to give our player's the capability of throwing a snowball at each other. The snowballs will be represented by **Image** objects similar to our players, so you might be thinking they fit within the **Sprite** class we've already developed. The one difference here is that once a snowball is thrown, it moves without user interaction at some predetermined speed, its position being updated by a timer control. And, we need to constantly check if a snowball collides with another object. Our **Sprite** class has no property for speed, nor any method for collision checking. Hence, we need a new class to define our snowballs.

We will describe our snowballs using a **MovingSprite** class. This class will have all the properties and methods of the **Sprite** class (to allow movement), plus additional speed properties and a method to check for collisions. To build this class, we could start from scratch – with all new properties and all new methods. Or, we could take advantage of a very powerful concept in object-oriented programming, **inheritance**. Inheritance is the idea that you can base one class on an existing class, adding properties and/or methods as needed. This saves lots of work.

Let's see how inheritance works with our snowballs, considering the speed properties for now. We'll add the collision checking method later. Add another class to the project, naming it **MovingSprite**. Use this code for the class:

```
public class MovingSprite : Sprite
{
    public MovingSprite()
    {
    }
    public int XSpeed;
    public int YSpeed;
}
```

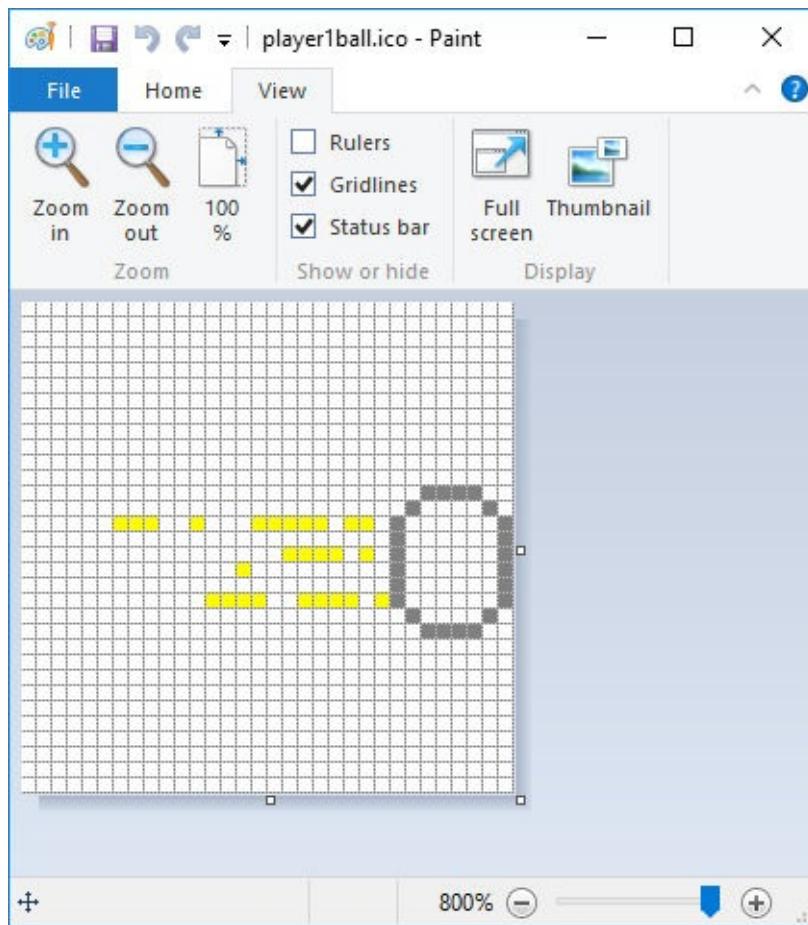
The key line here is:

```
public class MovingSprite : Sprite
```

The shaded addition makes all the properties and methods of the **Sprite** class available to our new class (**MovingSprite**). The remaining lines add the speed properties (**XSpeed**, speed in horizontal direction; **YSpeed**, speed in vertical direction). These speeds represent how much the **MovingSprite** will move in the corresponding direction with each update of position.

# Code Design – Throwing Snowballs

We can use the **MovingSprite** class to add the capability of throwing snowballs to our project. We include two icon files to represent the snowballs (included in the **HomeVCS\HomeVCS Projects\Snowball** folder) named **player1ball.ico** and **player2ball.ico**. Move these files to your project **Bin\Debug** folder. If you open **player1ball.ico** in the **Paint** program, you can see the detail:



Snowballs are represented by two **MovingSprite** objects (**snowball1** is player 1's snowball, while **snowball2** is player 2's snowball). The horizontal speed is set by the constant **snowballSpeed**. The vertical speed is zero. Add these form level declarations to your project (**Snowball.cs**):

```
MovingSprite snowball1, snowball2;  
const int snowballSpeed = 20;
```

The speed is another parameter set by playing around with the program (again, a value you might like to change).

Snowballs are constructed in the form **Load** method. Add these four lines to that method (after the lines establishing the player **Sprite** objects):

```
    snowball1 = new MovingSprite();
    snowball2 = new MovingSprite();
    snowball1.Graphic = Image.FromFile(Application.StartupPath +
        "\\player1ball.ico");
    snowball2.Graphic = Image.FromFile(Application.StartupPath +
        "\\player2ball.ico");
```

Player 1 throws a snowball by pressing the **S** key. Player 2 throws a snowball by pressing the **J** key. We establish a couple of rules for throwing a snowball. First, we will only allow one snowball from each player to be on the screen at any one time (no multiple firings!). Second, the player can't throw a snowball if he is out of snowballs (obviously). Assuming these conditions are met, when a player makes a throw, the following steps occur:

- Decrement the number of snowballs left.
- Update display of snowballs left.
- Position snowball next to throwing player (just to right of Player 1, just to left of Player 2).
- Place snowball on graphics object using **Place** method.

This ‘throwing’ code goes in the existing form **KeyDown** event method. The modified code is (changes are shaded):

```
private void frmSnowball_KeyDown(object sender, KeyEventArgs e)
{
    if (btnGame.Text == "New Game")
        return;
    switch (e.KeyCode)
    {
        case Keys.A:
```

```
player1.Move(myGraphics, blankBrush, 0, -playerIncrement);
if (player1.Location.Y < 0)
    player1.Location.Y = 0;
break;
case Keys.Z:
    player1.Move(myGraphics, blankBrush, 0, playerIncrement);
    if (player1.Location.Y > pnlSnow.ClientSize.Height -
player1.Location.Height)
        player1.Location.Y = pnlSnow.ClientSize.Height -
player1.Location.Height;
    break;
case Keys.S:
    if (!snowball1.IsVisible && player1Left > 0)
    {
        player1Left--;
        lblPlayer1Left.Text = player1Left.ToString();
        snowball1.Location = new Rectangle(player1.Location.X +
player1.Location.Width, player1.Location.Y, snowball1.Graphic.Width,
snowball1.Graphic.Height);
        snowball1.Place(myGraphics);
    }
    break;
case Keys.K:
    player2.Move(myGraphics, blankBrush, 0, -playerIncrement);
    if (player2.Location.Y < 0)
        player2.Location.Y = 0;
    break;
case Keys.M:
    player2.Move(myGraphics, blankBrush, 0, playerIncrement);
    if (player2.Location.Y > pnlSnow.ClientSize.Height -
player2.Location.Height)
        player2.Location.Y = pnlSnow.ClientSize.Height -
player2.Location.Height;
```

```

        break;

    case Keys.J:
        if (!snowball2.Visible && player2Left > 0)
        {
            player2Left--;
            lblPlayer2Left.Text = player2Left.ToString();
            snowball2.Location = new Rectangle(player2.Location.X -
player2.Location.Width, player2.Location.Y, snowball1.Graphic.Width,
snowball1.Graphic.Height);
            snowball2.Place(myGraphics);
        }
        break;
    }
}

```

Make the noted changes. This code gets a snowball started. Let's look at the code to get a snowball moving.

Once thrown, motion of the snowball(s) is updated by a timer control (**timGame**). We need code in the **btnGame Click** event to start that timer (when **New Game** is clicked) and to stop the timer (when **Stop Game** is clicked). We also remove the snowballs when **Stop Game** is clicked. The modified **btnGame Click** method (changes are shaded, most unmodified code is not shown) is:

```

private void btnGame_Click(object sender, EventArgs e)
{
    if (btnGame.Text == "New Game")
    {
        .
        .
        .
        timGame.Enabled = true;
        this.Focus();
    }
    else

```

```

{
    timGame.Enabled = false;
    player1.Remove(myGraphics, blankBrush);
    player2.Remove(myGraphics, blankBrush);
    snowball1.Remove(myGraphics, blankBrush);
    snowball2.Remove(myGraphics, blankBrush);

    .
    .
}

}

```

Add the two new lines.

In the **timGame Tick** event method, we update the position of thrown snowballs using the horizontal speed value and the **Move** method. We also check to see if a snowball goes off the edge of the panel control. If it does, we remove it from the panel to allow another throw. We also check for the end of the game (both players are out of snowballs and none are visible). If the game has ended, we ‘click’ **btnGame**. The code that does all this is:

```

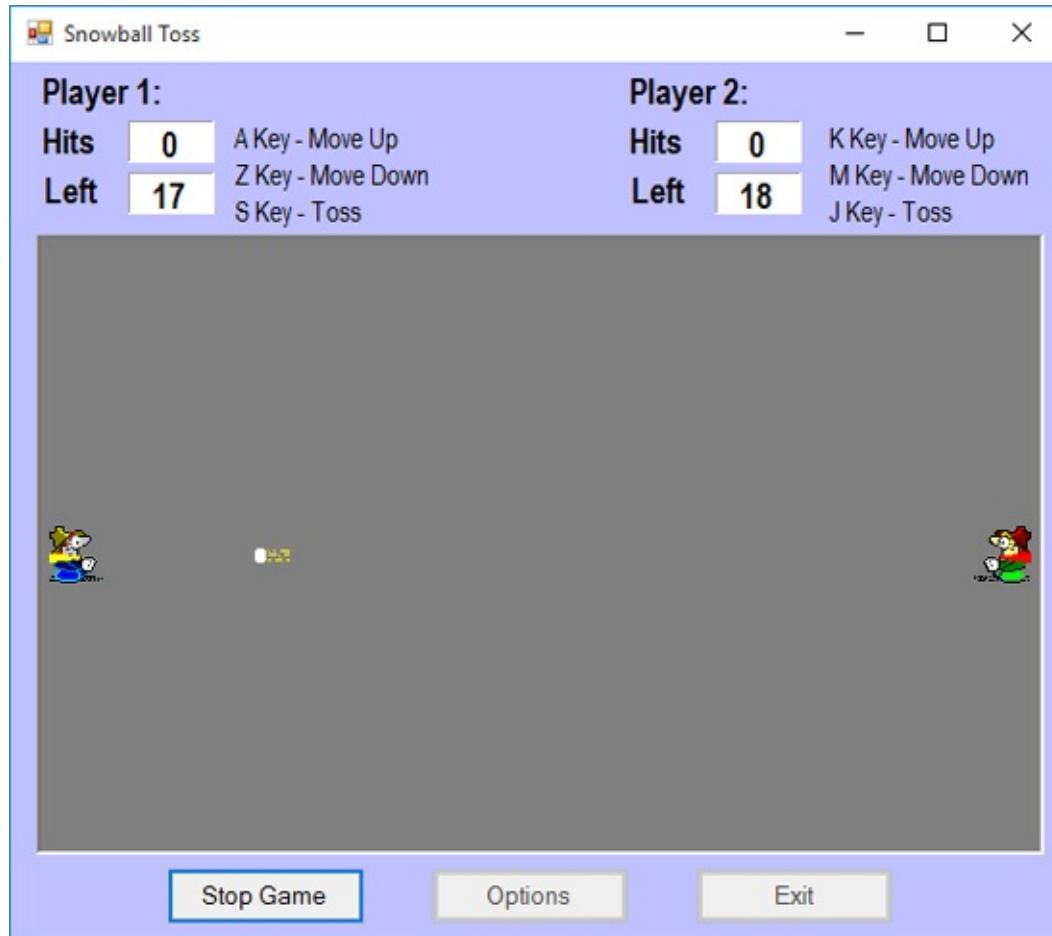
private void timGame_Tick(object sender, EventArgs e)
{
    // status of player 1 snowball
    if (snowball1.Visible)
    {
        snowball1.Move(myGraphics, blankBrush, snowballSpeed, 0);
        if (snowball1.Location.X > pnlSnow.ClientSize.Width)
            snowball1.Remove(myGraphics, blankBrush); // off screen
    }
    // status of player 2 snowball
    if (snowball2.Visible)
    {
        snowball2.Move(myGraphics, blankBrush, -snowballSpeed, 0);
    }
}

```

```
if (snowball2.Location.X < 0)
    snowball2.Remove(myGraphics, blankBrush); // off screen
}
// check status of game
if (!snowball1.IsVisible && player1Left == 0 &&
!snowball2.IsVisible && player2Left == 0)
    btnGame.PerformClick();
}
```

Add this method to your project. By the way, we have selected an **Interval** property of **50** milliseconds for **timGame**. This is another parameter you may want to change to speed up or slow down the game to your liking.

Save and run the project. Make sure you are still using the two player option. Click **New Game**. Click **S** and **J** to throw snowballs. Make sure the labels reflect the proper number of remaining snowballs. If a snowball hits a player, the player graphic may disappear. To make it reappear, press one of the keys that move it up or down. Here is a run I made with a snowball flying:



Stop the game when you want or throw snowballs until neither player has any remaining. Notice we don't have any scoring (counting hits). To do this, we need collision logic, which is discussed next.

# Code Design - Collision Detection

If a thrown snowball hits a player, the other player gets a point (a **Hit**). We need some way to check for such a ‘collision.’ Since rectangular regions describe the moving objects here, we want to know if two rectangles intersect. In Visual C#, this test can be accomplished using the **IntersectsWith** method of the **Rectangle** structure.

To see if two rectangles (call them **rectangle1** and **rectangle2**) intersect, we check the **bool** value of:

```
rectangle1.IntersectsWith(rectangle2)
```

If a **true** is returned, there is an intersection or collision. If **false** is returned, there is no collision.

To use this in our project, we will add a **Collided** method to our **MovingSprite** class. Open the **MovingSprite.cs** file and make the shaded changes:

```
public class MovingSprite : Sprite
{
    public MovingSprite()
    {
    }

    public int XSpeed;
    public int YSpeed;

    public bool Collided(Rectangle r)
    {
        return (this.Location.IntersectsWith(r));
    }
}
```

The method is passed the rectangle structure (**r**) to check for collision with the

**MovingSprite** object.

You also add this using directive to the file (needed to use the **Rectangle** structure):

```
using System.Drawing;
```

You may wonder why we didn't need such a directive in our **Snowball.cs** file – we also use **Rectangle** structures and other drawing methods there. The **System.Drawing** directive is automatically included with a “Windows” application file, but not with a class file.

As an example of using this new method, say we want to check if **snowball1** has hit **player2**. The **bool** value:

```
snowball1.Collided(player2.Location)
```

will be **true** if such a collision has occurred. If a collision occurs, we remove the snowball and update the successful tosser's score.

We check for collisions between snowballs and players in the **timGame Tick** event method. The modified code (shaded) checks for collisions and updates the score accordingly:

```
private void timGame_Tick(object sender, EventArgs e)
{
    // status of player 1 snowball
    if (snowball1.IsVisible)
    {
        snowball1.Move(myGraphics, blankBrush, snowballSpeed, 0);
        if (snowball1.Location.X > pnlSnow.ClientSize.Width)
            snowball1.Remove(myGraphics, blankBrush); // off screen
        else if (snowball1.Collided(player2.Location))
        {
            player1Hits++;
        }
    }
}
```

```

        lblPlayer1Hits.Text = player1Hits.ToString();
        snowball1.Remove(myGraphics, blankBrush);
    }

}

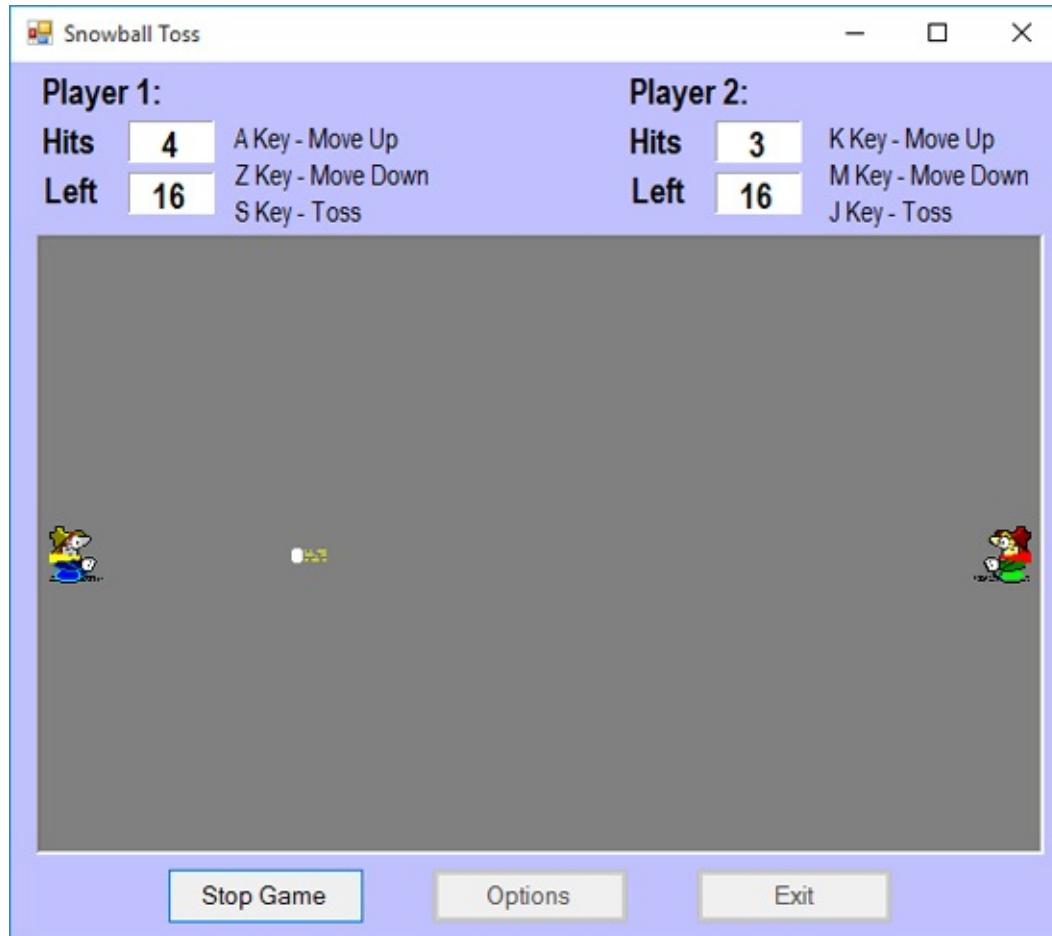
// status of player 2 snowball
if (snowball2.Visible)
{
    snowball2.Move(myGraphics, blankBrush, -snowballSpeed, 0);
    if (snowball2.Location.X < 0)
        snowball2.Remove(myGraphics, blankBrush); // off screen
    else if (snowball2.Collided(player1.Location))
    {
        player2Hits++;
        lblPlayer2Hits.Text = player2Hits.ToString();
        snowball2.Remove(myGraphics, blankBrush);
    }
}

// check status of game
if (!snowball1.Visible && player1Left == 0 && !snowball2.Visible
&& player2Left == 0)
    btnGame.PerformClick();
}

```

Make the noted modifications.

Save and run the project. Now if you throw a snowball and hit the other player, the snowball should disappear, but not the player. Give it a try. Here's the middle of a game I played:

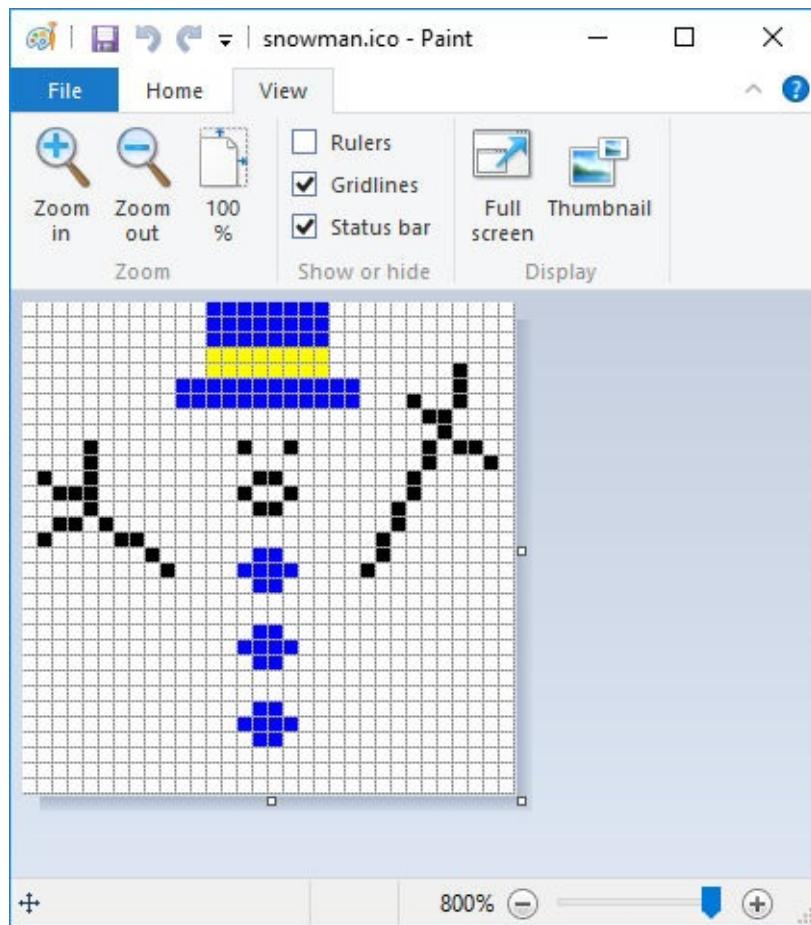


Make sure the score updates properly after each successful toss.

# Code Design – Zombie Snowmen

As designed, the players have no protection from thrown snowballs other than their ability to move up and down. We'll change that now. We'll invent a tribe of "zombie" snowmen that roam up and down in the middle of the playing field. These snowmen will deflect (stop) any snowball that has been thrown.

The icon file **snowman.ico** (included in the **HomeVCS\HomeVCS Projects\Snowball** folder) depicts a snowman. Move this file to your project **Bin\Debug** folder. If you open **snowman.ico** in the **Paint** program, you can see the detail:



Where's the snowman? Pixels outside the snowman are defined as transparent (consult **Paint** to know how to do this). I know – he looks pretty happy for a zombie!

We will have two snowmen (you can choose more if you want). Snowmen are represented by **MovingSprite** objects (**snowman1** and **snowman2**). Their motion will be random, with some restrictions. Add these form level declarations to your project to declare the snowman objects and the random number object.

```
MovingSprite snowman1, snowman2;  
Random myRandom = new Random();
```

Our snowmen will move according to some predetermined rules. **snowman1** will move vertically just to the left of the center of playing field, while **snowman2** will move vertically just to the right of center. The horizontal speed for both will be zero. The vertical speed will be a random value between 1 and 4 (a value you might want to change). The snowmen can move either up or down. If moving up, they start at the bottom of the field. If moving down, they start at the top. How did I come up with all these rules for my zombies? I made them up. That's the nice thing about being a game programmer. You can make your characters do whatever you want them to. Come up with rules for your own set of zombie snowmen if you want.

The snowmen are constructed in the form **Load** method. Add these four lines to that method (after the lines establishing the snowball **MovingSprite** objects):

```
snowman1 = new MovingSprite();  
snowman2 = new MovingSprite();  
snowman1.Graphic = Image.FromFile(Application.StartupPath +  
"\\snowman.ico");  
snowman2.Graphic = Image.FromFile(Application.StartupPath +  
"\\snowman.ico");
```

The snowmen are initially placed in the field when **New Game** is clicked. When this occurs, they are randomly placed within the vertical constraints of the graphics object. And, they are assigned a random speed. The snowmen are removed when **Stop Game** is clicked. The modified **btnGame Click** method that accomplishes these tasks (changes are shaded, with much unmodified code not shown) is:

```
private void btnGame_Click(object sender, EventArgs e)
```

```
{  
    if (btnGame.Text == "New Game")  
    {  
        .  
        .  
        .  
        snowman1.Location = new Rectangle((int)(0.5 *  
pn1Snow.ClientSize.Width - snowman1.Graphic.Width),  
myRandom.Next(pn1Snow.ClientSize.Height),  
snowman1.Graphic.Width, snowman1.Graphic.Height);  
        snowman2.Location = new Rectangle((int)(0.5 *  
pn1Snow.ClientSize.Width),  
myRandom.Next(pn1Snow.ClientSize.Height),  
snowman2.Graphic.Width, snowman2.Graphic.Height);  
        snowman1.YSpeed = SnowmanSpeed();  
        snowman2.YSpeed = SnowmanSpeed();  
        snowman1.Place(myGraphics);  
        snowman2.Place(myGraphics);  
        timGame.Enabled = true;  
        this.Focus();  
    }  
    else  
    {  
        timGame.Enabled = false;  
        player1.Remove(myGraphics, blankBrush);  
        player2.Remove(myGraphics, blankBrush);  
        snowball1.Remove(myGraphics, blankBrush);  
        snowball2.Remove(myGraphics, blankBrush);  
        snowman1.Remove(myGraphics, blankBrush);  
        snowman2.Remove(myGraphics, blankBrush);  
        .  
        .  
    }  
}
```

**snowman1** is just to the left of the middle of the panel control, while **snowman2** is just to the right. Notice how the snowmen are randomly positioned vertically.

The snowman speed is assigned using a general method **SnowmanSpeed**. As mentioned, we choose this value to be random, between 1 and 4. The speed can be positive (for downward motion) or negative (for upward motion). This choice of sign is also random. The code that incorporates this speed assignment is:

```
private int SnowmanSpeed()
{
    const int speedMin = 1;
    const int speedMax = 4;
    int speed;
    speed = myRandom.Next(speedMax - speedMin + 1) + speedMin;
    if (myRandom.Next(2) == 0)
        speed = -speed;
    return(speed);
}
```

Computing the speed value is straightforward. To choose the sign, we do a computerized “coin flip”. This flip is done by looking at the value of:

**myRandom.Next(2)**

This can return one of two values, 0 (“heads”) or 1 (“tails”).

Snowman motion is updated in the **timGame Tick** event. At each update, for each snowman, we need to perform the following steps:

- Move the snowman using the current **YSpeed** property.
- Check to see if the snowman has moved off the playing field.
- If off field, do this:
  - Compute a new speed.
  - If speed is positive, position snowman off top of playing field so it can start moving down.

- o If speed is negative, position snowman off bottom of playing field so it can start moving up.
- After moving or repositioning snowman, check to see if a thrown snowball has collided with it. If there is a collision, remove the snowball from the field.

The modified **timGame Tick** event method that implements these steps is (changes are shaded):

```
private void timGame_Tick(object sender, EventArgs e)
{
    // move snowmen
    snowman1.Move(myGraphics, blankBrush, 0, snowman1.YSpeed);
    if (snowman1.Location.Y < -snowman1.Graphic.Height ||
    snowman1.Location.Y > pnlSnow.ClientSize.Height)
    {
        // recompute speed
        snowman1.YSpeed = SnowmanSpeed();
        if (snowman1.YSpeed > 0)
            snowman1.Location.Y = -snowman1.Graphic.Height;
        else
            snowman1.Location.Y = pnlSnow.ClientSize.Height;
    }
    snowman2.Move(myGraphics, blankBrush, 0, snowman2.YSpeed);
    if (snowman2.Location.Y < -snowman2.Graphic.Height ||
    snowman2.Location.Y > pnlSnow.ClientSize.Height)
    {
        // recompute speed
        snowman2.YSpeed = SnowmanSpeed();
        if (snowman2.YSpeed > 0)
            snowman2.Location.Y = -snowman2.Graphic.Height;
        else
            snowman2.Location.Y = pnlSnow.ClientSize.Height;
    }
}
```

```
}

// status of player 1 snowball
if (snowball1.Visible)
{
    snowball1.Move(myGraphics, blankBrush, snowballSpeed, 0);
    if (snowball1.Location.X > pnlSnow.ClientSize.Width)
        snowball1.Remove(myGraphics, blankBrush); // off screen
    else if (snowball1.Collided(player2.Location))
    {
        player1Hits++;
        lblPlayer1Hits.Text = player1Hits.ToString();
        snowball1.Remove(myGraphics, blankBrush);
    }
    else if (snowball1.Collided(snowman1.Location) ||
    snowball1.Collided(snowman2.Location))
        snowball1.Remove(myGraphics, blankBrush);
}

// status of player 2 snowball
if (snowball2.Visible)
{
    snowball2.Move(myGraphics, blankBrush, -snowballSpeed, 0);
    if (snowball2.Location.X < 0)
        snowball2.Remove(myGraphics, blankBrush); // off screen
    else if (snowball2.Collided(player1.Location))
    {
        player2Hits++;
        lblPlayer2Hits.Text = player2Hits.ToString();
        snowball2.Remove(myGraphics, blankBrush);
    }
    else if (snowball2.Collided(snowman1.Location) ||
    snowball2.Collided(snowman2.Location))
        snowball2.Remove(myGraphics, blankBrush);
}
```

```

        }

    // check status of game
    if (!snowball1.IsVisible && player1Left == 0 && !snowball2.IsVisible
&& player2Left == 0)
        btnGame.PerformClick();
}

```

Make the noted changes. You should understand how all the zombie rules have been applied.

Save and run the project. The snowmen should be moving through the middle of the field deflecting any snowballs they might block. They should move both up and down at varying speeds. Watch them for a while. Here's a run I made:



The two player version of the snowball toss game is essentially complete. All the

animation steps are implemented – we can move the players, we can throw snowballs, the zombie snowmen can block snowballs and the score is properly kept. Next, we'll program the one player version, making the computer control Player 2.

Before doing the one player version, however, let's address one sorely lacking feature – sounds! Any good game has sound and we should have some in this game. Let's add a throwing sound, a splat sound when a snowman is hit and an “Ouch” sound when a player is hit. And, let's add a little tune when the game is over.

# Code Design – Playing Sounds

Most games feature sounds that take advantage of stereo sound cards. By using the Visual C# **SoundPlayer** class, we can add such sounds to our snowball toss game. This class uses the Visual C# **Media** namespace, so for any application using sounds, you will need to add this line to the code:

```
using System.Media;
```

at the top of the code window with the other directives.

The **SoundPlayer** class is used to play one particular type of sound, those represented by **wav** files (files with wav extensions). Most sounds you hear played in Windows applications are saved as **wav** files. These are the files formed when you record using one of the many sound recorder programs available. In the **HomeVCS\HomeVCS Projects\Snowball** folder are four **wav** files for use in this program:

<b>throw.wav</b>	sound to play when a snowball is thrown
<b>splat.wav</b>	sound to play when a snowball hits a snowman
<b>ouch.wav</b>	sound to play when a snowball hits a player
<b>gameover.wav</b>	sound to play when game is over (both players are out of snowballs)

You can play each of these sounds in your computer's media player if you want.

Prior to playing a sound, a **SoundPlayer** object (**mySound**) must be declared using the usual statement.

```
SoundPlayer mySound;
```

The object is then constructed using:

```
mySound = new SoundPlayer(SoundFile);
```

where **SoundFile** is a complete path to the **wav** file to be played.

A sound is loaded into memory using the **Load** method of the **SoundPlayer** class:

```
mySound.Load();
```

This is an optional step that should be done for sounds that will be played often – it allows them to be played quickly.

There are two methods available to play a sound, the **Play** method and the **PlaySync** method:

```
mySound.Play();  
mySound.PlaySync();
```

With the **Play** method, a sound is played **asynchronously**. This means that execution of code continues as the sound is played. With **PlaySync**, the sound is played **synchronously**. In this case, the sound is played to completion, and then code execution continues. With either method, if the sound is not loaded into memory, it will be loaded first. This will slow down your program depending on how big the file is. This is why we preload sounds (using the **Load** method) that are played often.

It is normal practice to include any sound files an application uses in the same directory as the application executable file (the **Bin\Debug** folder). This makes them easily accessible (use the **Application.StartupPath** parameter). Copy the four included sound files into your project's **Bin\Debug** folder.

Let's modify the snowball toss game code to include the sounds. Add this directive to the top of the code window:

```
using System.Media;
```

Add a form level declaration for the variables used to represent the sounds:

```
SoundPlayer throwSound, splatSound, ouchSound, gameOverSound;
```

The code to construct the **SoundPlayer** objects (using the **wav** files) goes in the form **Load** event method. We also add code to preload (using the **Load** method) the throw, splat and ouch sounds, since they will be played many times in a game. Place the shaded lines near the end of that method (right before the line giving **btnGame** focus):

```
private void frmSnowball_Load(object sender, EventArgs e)
{
    try
    {
        StreamReader inputFile = new
        StreamReader(Application.StartupPath + "\\snowball.ini");
        numberPlayers =
Convert.ToInt32(inputFile.ReadLine());
        difficulty =
Convert.ToInt32(inputFile.ReadLine());
        inputFile.Close();
    }
    catch
    {
        numberPlayers = 1;
        difficulty = 1;
    }
    // make form visible to 'click' options
    this.Show();
    pnlOptions.Visible = true;
    if (difficulty == 1)
        rdoEasiest.PerformClick();
    else if (difficulty == 2)
        rdoEasy.PerformClick();
    else if (difficulty == 3)
        rdoHard.PerformClick();
    else
```

```
rdoHardest.PerformClick();
if (numberPlayers == 1)
    rdoOnePlayer.PerformClick();
else
    rdoTwoPlayers.PerformClick();
pnlOptions.Visible = false;
player1Left = maximumBalls;
player2Left = maximumBalls;
lblPlayer1Left.Text = player1Left.ToString();
lblPlayer2Left.Text = player2Left.ToString();
// create sprites
myGraphics = pnlSnow.CreateGraphics();
blankBrush = new SolidBrush(pnlSnow.BackColor);
player1 = new Sprite();
player2 = new Sprite();
player1.Graphic =
Image.FromFile(Application.StartupPath + "\\player1.ico");
player2.Graphic =
Image.FromFile(Application.StartupPath + "\\player2.ico");
snowball1 = new MovingSprite();
snowball2 = new MovingSprite();
snowball1.Graphic =
Image.FromFile(Application.StartupPath + "\\player1ball.ico");
snowball2.Graphic =
Image.FromFile(Application.StartupPath + "\\player2ball.ico");
snowman1 = new MovingSprite();
snowman2 = new MovingSprite();
snowman1.Graphic =
Image.FromFile(Application.StartupPath + "\\snowman.ico");
snowman2.Graphic =
Image.FromFile(Application.StartupPath + "\\snowman.ico");
throwSound = new SoundPlayer(Application.StartupPath +
```

```

    "\\throw.wav");
    splatSound = new SoundPlayer(Application.StartupPath +
    "\\splat.wav");
    ouchSound = new SoundPlayer(Application.StartupPath +
    "\\ouch.wav");
    gameOverSound = new SoundPlayer(Application.StartupPath +
    "\\gameover.wav");
    throwSound.Load();
    splatSound.Load();
    ouchSound.Load();
    btnGame.Focus();
}

```

We have listed the entire **frmSnowball Load** method since we are done changing this particular method.

**throwSound** will play when a snowball is thrown. This action occurs in the **frmSnowball KeyDown** method. The modified code (changes are shaded, most unmodified code is not shown) is:

```

private void frmSnowball_KeyDown(object sender, KeyEventArgs e)
{
    if (btnGame.Text == "New Game")
        return;
    switch (e.KeyCode)
    {
        .
        .
        case Keys.S:
            if (!snowball1.IsVisible && player1Left > 0)
            {
                throwSound.Play();
                player1Left--;
                lblPlayer1Left.Text = player1Left.ToString();
            }
    }
}

```

```

        snowball1.Location = new Rectangle(player1.Location.X +
player1.Location.Width, player1.Location.Y, snowball1.Graphic.Width,
snowball1.Graphic.Height);
        snowball1.Place(myGraphics);
    }

    .
    .

case Keys.J:
    if (!snowball2.IsVisible && player2Left > 0)
    {
        throwSound.Play();
        player2Left--;
        lblPlayer2Left.Text = player2Left.ToString();
        snowball2.Location = new Rectangle(player2.Location.X -
player2.Location.Width, player2.Location.Y, snowball1.Graphic.Width,
snowball1.Graphic.Height);
        snowball2.Place(myGraphics);
    }
}
}

```

Make the two changes.

**splatSound** will play when a snowman is hit by a snowball, **ouchSound** will play when a player is hit by a snowball and **gameOverSound** will play when the players run out of snowballs. All of these actions occur in the **timGame Tick** event. The modified code (changes are shaded) is:

```

private void timGame_Tick(object sender, EventArgs e)
{
    // move snowmen
    snowman1.Move(myGraphics, blankBrush, 0, snowman1.YSpeed);
    if (snowman1.Location.Y < -snowman1.Graphic.Height ||
snowman1.Location.Y > pnlSnow.ClientSize.Height)

```

```

{
    // recompute speed
    snowman1.YSpeed = SnowmanSpeed();
    if (snowman1.YSpeed > 0)
        snowman1.Location.Y = -snowman1.Graphic.Height;
    else
        snowman1.Location.Y = pnlSnow.ClientSize.Height;
}
snowman2.Move(myGraphics, blankBrush, 0, snowman2.YSpeed);
if (snowman2.Location.Y < -snowman2.Graphic.Height ||
snowman2.Location.Y > pnlSnow.ClientSize.Height)
{
    // recompute speed
    snowman2.YSpeed = SnowmanSpeed();
    if (snowman2.YSpeed > 0)
        snowman2.Location.Y = -snowman2.Graphic.Height;
    else
        snowman2.Location.Y = pnlSnow.ClientSize.Height;
}
// status of player 1 snowball
if (snowball1.Visible)
{
    snowball1.Move(myGraphics, blankBrush, snowballSpeed, 0);
    if (snowball1.Location.X > pnlSnow.ClientSize.Width)
        snowball1.Remove(myGraphics, blankBrush); // off screen
    else if (snowball1.Collided(player2.Location))
    {
        ouchSound.Play();
        player1Hits++;
        lblPlayer1Hits.Text = player1Hits.ToString();
        snowball1.Remove(myGraphics, blankBrush);
    }
}

```

```
        else if (snowball1.Collided(snowman1.Location) ||
snowball1.Collided(snowman2.Location))
    {
        splatSound.Play();
        snowball1.Remove(myGraphics, blankBrush);
    }

// status of player 2 snowball
if (snowball2.Visible)
{
    snowball2.Move(myGraphics, blankBrush, -snowballSpeed, 0);
    if (snowball2.Location.X < 0)
        snowball2.Remove(myGraphics, blankBrush); // off screen
    else if (snowball2.Collided(player1.Location))
    {
        ouchSound.Play();
        player2Hits++;
        lblPlayer2Hits.Text = player2Hits.ToString();
        snowball2.Remove(myGraphics, blankBrush);
    }

    else if (snowball2.Collided(snowman1.Location) ||
snowball2.Collided(snowman2.Location))
    {
        splatSound.Play();
        snowball2.Remove(myGraphics, blankBrush);
    }

}

// check status of game
if (!snowball1.Visible && player1Left == 0 && !snowball2.Visible
&& player2Left == 0)
{
    gameOverSound.PlaySync();
```

```
    btnGame.PerformClick();  
}  
}
```

Make the noted changes. The collision sounds are played asynchronously (**Play** method) sound game play can continue. The **gameOverSound** is played synchronously – it must finish playing before user events can occur. This is the final version of this method.

Save and run the project. Play the two player game. Listen for the throw, splat and ouch sounds. And, play until all the snowballs are thrown to hear the cute little “game over” tune. Make these changes. I think you’ll agree that the sounds make the game far more fun to play.

# Code Design – One Player Game

You can't always find someone to play a game with. So why not let the computer be your opponent? In a one player snowball toss game, we will let the computer control Player 2.

For such computer control, we need to develop some rules for the computer to use. In the **Blackjack** card game built earlier in these notes, we played against the computer. The rules used there by the computer were predetermined by those used in most casinos. Here, in the snowball toss game, we have no such rules. We need to develop them ourselves. This is a fun part of programming – giving the computer some semblance of intelligence. The logic presented here are ideas that I just made up as I went along. They seem to work. Feel free to make changes you think are needed.

There are two approaches we could take in writing code for a computer competitor. We could use very simple logic, making it easy for a human to win. Or, we could write more detailed logic, emulating steps you, as a human, might take in playing the game. With more detailed logic, it would be harder for a human to win. In the snowball toss game, we take both approaches. We first develop a simple, random game playing logic, then a more detailed logic. Then, we use the level of difficulty selected with the **Options** button to determine how often we use the random logic versus how often we use the detailed logic. The values I chose to use are:

Difficulty Level	Simple Logic (%)	Detailed Logic (%)
Easiest	100	0
Easy	75	25
Hard	50	50
Hardest	25	75

So, when the **Easiest** level is selected, we use the simple logic 100 percent of the time. When the **Hardest** level is selected, we use the simple logic 25 percent of the time (we don't want our computer to be too smart) and the detailed logic 75 percent of the time

Another value selected by the level of difficulty will be how often the computer makes a move. The computer moves will be controlled by a separate timer control (**timComputer**). For easier games, we want a larger value for the **Interval** property for this control. This slows down the computer's thought process. The **Interval** values I chose are:

Difficulty Level	Interval
Easiest	1000
Easy	750
Hard	500
Hardest	250

So with the **Hardest** difficulty, the computer makes moves 4 times as often as when the **Easiest** difficulty is selected.

We will use an **int** variable (**computerRandom**) to represent the percentage of time simple logic is used and an **int** variable (**computerTime**) to represent the timer control **Interval** property. Add these form level declarations to the project:

```
int computerRandom, computerTime;
```

Values for these two new variables are set in the **rdoDifficulty Click** event method. The modified method is (changes are shaded):

```
private void rdoDifficulty_Click(object sender, EventArgs e)
{
    RadioButton buttonClicked;
    buttonClicked = (RadioButton) sender;
    switch (buttonClicked.Text)
    {
        case "Easiest":
            difficulty = 1;
            computerRandom = 100;
            computerTime = 1000;
            break;
    }
}
```

```

case "Easy":
    difficulty = 2;
    computerRandom = 75;
    computerTime = 750;
    break;
case "Hard":
    difficulty = 3;
    computerRandom = 50;
    computerTime = 500;
    break;
case "Hardest":
    difficulty = 4;
    computerRandom = 25;
    computerTime = 250;
    break;
}
timComputer.Interval = computerTime;
}

```

Make the noted modifications.

We need to start the computer's timer control (**timComputer**) when playing the one player game. And, we need to stop it when done playing. This is done in the **btnGame Click** event (new lines are shaded):

```

private void btnGame_Click(object sender, EventArgs e)
{
    if (btnGame.Text == "New Game")
    {
        myGraphics.Clear(pnlSnow.BackColor);
        btnGame.Text = "Stop Game";
        btnOptions.Enabled = false;
        btnExit.Enabled = false;
    }
}

```

```
player1Hits = 0;
player2Hits = 0;
lblPlayer1Hits.Text = "0";
lblPlayer2Hits.Text = "0";
player1Left = maximumBalls;
player2Left = maximumBalls;
lblPlayer1Left.Text = player1Left.ToString();
lblPlayer2Left.Text = player2Left.ToString();
player1.Location = new Rectangle(5, (int)(0.5 *
pnlSnow.ClientSize.Height - 0.5 * player1.Graphic.Height),
player1.Graphic.Width, player1.Graphic.Height);
player2.Location = new Rectangle(pnlSnow.ClientSize.Width -
player2.Graphic.Width - 5, (int)(0.5 * pnlSnow.ClientSize.Height - 0.5 *
player2.Graphic.Height), player2.Graphic.Width,
player2.Graphic.Height);
player1.Place(myGraphics);
player2.Place(myGraphics);
snowman1.Location = new Rectangle((int)(0.5 *
pnlSnow.ClientSize.Width - snowman1.Graphic.Width),
myRandom.Next(pnlSnow.ClientSize.Height), snowman1.Graphic.Width,
snowman1.Graphic.Height);
snowman2.Location = new Rectangle((int)(0.5 *
pnlSnow.ClientSize.Width),
myRandom.Next(pnlSnow.ClientSize.Height), snowman2.Graphic.Width,
snowman2.Graphic.Height);
snowman1.YSpeed = SnowmanSpeed();
snowman2.YSpeed = SnowmanSpeed();
snowman1.Place(myGraphics);
snowman2.Place(myGraphics);
timGame.Enabled = true;
if (numberPlayers == 1)
    timComputer.Enabled = true;
this.Focus();
}
```

```

else
{
    timGame.Enabled = false;
    timComputer.Enabled = false;
    player1.Remove(myGraphics, blankBrush);
    player2.Remove(myGraphics, blankBrush);
    snowball1.Remove(myGraphics, blankBrush);
    snowball2.Remove(myGraphics, blankBrush);
    snowman1.Remove(myGraphics, blankBrush);
    snowman2.Remove(myGraphics, blankBrush);
    btnGame.Text = "New Game";
    btnOptions.Enabled = true;
    btnExit.Enabled = true;
    myGraphics.DrawString("Game Over", new Font("Arial", 24),
Brushes.Yellow, 180, 140);
}
}

```

Add the shaded lines. We have listed the entire **btnGame Click** method – it is now complete.

Now, let's write the computer playing rules. We'll start with the simple, random rules. In these rules, the computer will just make random moves up and down the field, occasionally tossing a snowball. The only non-random element we add is that we only allow the computer to throw a snowball if it has at least as many snowballs left as the human player. This prevents the computer from tossing all its snowballs and becoming an easy target for the human. The rules I use are:

- Generate a random number from 0 to 4.
- If number is 0, toss snowball if computer has at least as many snowballs as player.
- If number is 1 or 2, move up.
- If number is 3 or 4, move down.

With these rules, a snowball is thrown 1 out of 5 times the computer makes a move, the computer's player moves up 2 of 5 times and moves down 2 of 5 times. There's no real intelligence involved – just random moves. Let's write some more intelligent rules.

In writing more detailed (smarter) playing rules, just think about how you would play the game. Smarter rules would be if the other player is “in range”, take a toss. Otherwise, move away from the other player if he's tossed a snowball (defensive move) or move toward the other player to keep him range (offensive move). In our rules, we define “in range” to mean the difference between the two players' vertical position no more than 80 percent of a player's height. The rules I used are:

- If “in range” and computer has at least as many snowballs as player, take toss.
- If human player has tossed snowball or computer has no snowballs remaining, make defensive move:
  - o If human player is above computer player, move down.
  - o If human player is below computer player, move up.
- Else, make offensive move:
  - o If human player is above computer player, move up.
  - o If human player is below computer player, move down.

Notice we still only toss a snowball when the computer has at least as many snowballs remaining as the player. We don't want the computer to run out of snowballs before the human player.

The code for both the simple and detailed computer logic is placed in the **timComputer Tick** event method. Before writing this code, however, we need to address how we can make the computer player take a toss, move up or move down. With a human second player, pressing **J** would make Player 2 toss, pressing **K** would move Player 2 up and pressing **M** would move Player 2 down. It would be nice if there was a way we could make the computer player press these same keys for the desired action. And, we can. Visual C# has a **SendKeys** method that can be used to simulate a key press on the keyboard. To press the **J** key, the code is:

```
SendKeys.Send("K");
```

For the computer player, we will use this **SendKeys** method to implement computer actions.

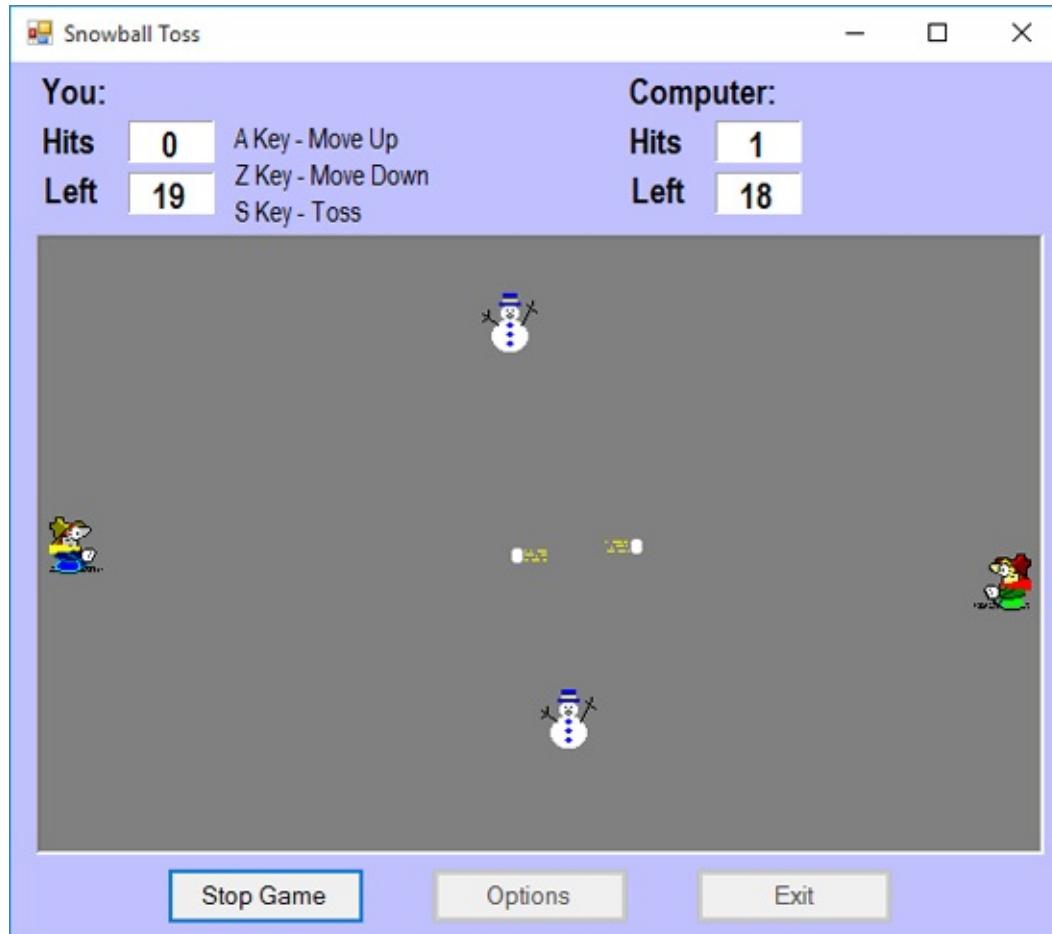
The **timComputer Tick** event method that implements the computer player logic is:

```
private void timComputer_Tick(object sender, EventArgs e)
{
    int i;
    if (myRandom.Next(100) < computerRandom)
    {
        i = myRandom.Next(5); // random move
        if (i == 0)
        {
            if (player2Left >= player1Left)
                SendKeys.Send("J"); // take toss
        }
        else if (i <= 2)
            SendKeys.Send("K"); // move up
        else
            SendKeys.Send("M"); // move down
    }
    else
    {
        if (Math.Abs(player1.Location.Y - player2.Location.Y) < (int)(0.8
* player1.Graphic.Height) && player2Left >= player1Left)
            SendKeys.Send("J"); // take toss
        if (snowball1.Visible || player2Left == 0)
        {
            if (player1.Location.Y - player2.Location.Y < 0)
                SendKeys.Send("M"); // move down
        }
    }
}
```

```
        else
            SendKeys.Send("K"); // move up
        }
    else
    {
        if (player1.Location.Y - player2.Location.Y < 0)
            SendKeys.Send("K"); // move up
        else
            SendKeys.Send("M"); // move down
    }
}
```

Add this method to your project. You should be able to identify each step in the different computer player logics. Make sure you understand how the **computerRandom** value is used to determine whether simple or detailed logic is used.

Save and run the project. Click **Options**, select a **One Player** game. Select a difficulty level. Click **OK**, then **New Game**. Then, watch out, the computer opponent will start tossing snowballs at you!



This completes the snowball toss game. As you play the game, against another player or against the computer, you'll find modifications you want to make. This is a fun part of game programming – tailoring the game play to your desires and needs.

# Snowball Toss Game Project Review

The **Snowball Toss Game Project** is now complete. Save and run the project and make sure it works as designed. Have fun playing the game against friends and family or against the computer. In the Appendix, we'll show you how you can share this game (or any other project) with other users.

If there are errors in your implementation, go back over the steps of form and code design. Use the debugger when needed. Go over the developed code – make sure you understand how different parts of the project were coded. As mentioned in the beginning of this chapter, the completed project is saved as **Snowball** in the **HomeVCS\HomeVCS Projects** folder.

While completing this project, new concepts and skills you should have gained include:

- Use of the **Rectangle** structure in graphics.
- How the **DrawImage** and **FillRectangle** graphics methods are used.
- Using a tool like **IconEdit** to develop graphics files.
- How to add methods to classes.
- Using inheritance with classes.
- Using the keyboard for control of animated characters.
- Detecting collisions between **Rectangle** structures.
- How to play sounds.
- How to develop game playing rules for the computer.

This is the last project in these notes. By now, you should be a fairly competent Visual C# programmer. There's always more to learn though. Consult the Internet and bookstores for more books about skills you might want to gain.

# Snowball Toss Game Project Enhancements

Possible enhancements to the snowball toss game project include:

- Add another option to allow the user to select the number of snowballs to use. Look at the **NumericUpDown** control to set such a value. Modify the configuration file so this value is saved.
- Players like to see their name “in lights.” Add an option to have player’s name placed on the form instead of the generic titling information used now. You might want to save the names in the configuration file.
- Add some horizontal motion to the zombie snowmen.
- In the computer playing logic, no consideration is given to position of the zombie snowmen. Modify the logic so a toss is taken only when a snowman is not blocking the toss.
- If you play the game against a ‘smart’ computer, you will find it is possible to trap the computer player at the top or bottom of the playing field and fire away. Modify the code to have the computer player move away from such trapped situations.
- In the one player game, it is still possible to control the computer player manually using the J, K and M keys. Can you write code so this is not possible? It’s not an easy task because of the **SendKeys** method.

# **Appendix**

## **Distributing a Visual C# Project**

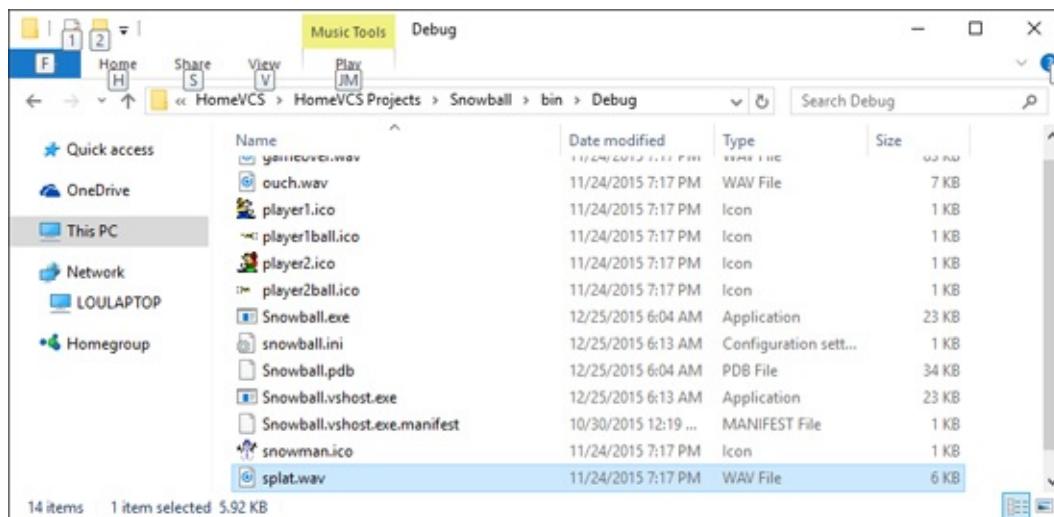
# Preview

I bet you're ready to show your friends, family and colleagues some of the applications you have built using Visual C#. Just give them a copy of all your project files, ask them to download and install Visual C# and learn how to open and run a project. Then, have them open your project and run the application.

I think you'll agree this is asking a lot of your friends. We need to know how to run an application **without** Visual C#. In this Appendix, we look at a simpler way to share your work. We'll use the **Snowball Toss Game** just built as an example.

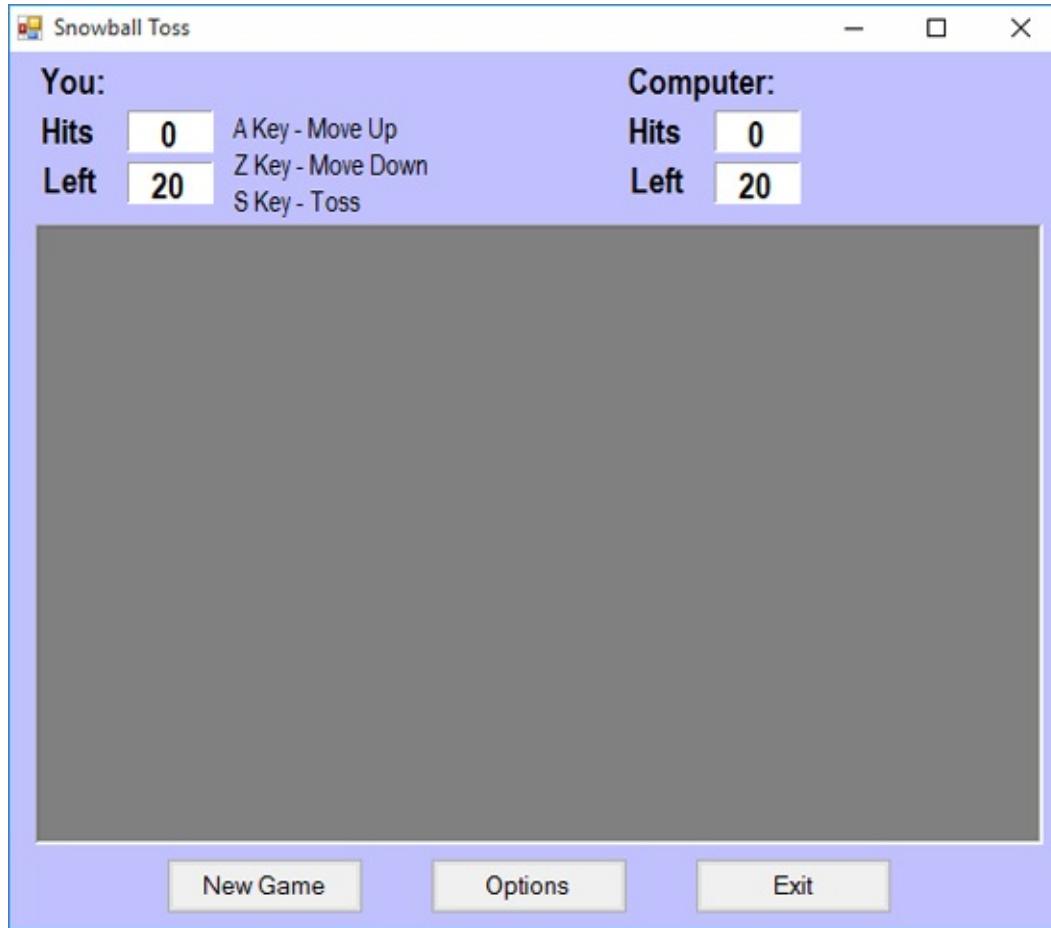
# Distribution of Files

To run an application without Visual C#, you need a **executable** version of the application (an **exe** file). Each time you run your project in the Visual C# environment, such an executable version is created and stored in your project's **Bin\Debug** folder. Open the **Bin\Debug** folder for any project you have built and you'll see a file with your project name of type **Application**. For example, using **My Computer** to open the **Bin\Debug** folder for the **Snowball Toss Game Project** shows:



In this folder are files created by Visual C# and files needed by the application to successfully run. These latter files include the configuration file (**snowball.ini**), graphics files (**player1.ico**, **player2.ico**, **snowball1.ico**, **snowball2.ico**, **snowman.ico**) and sound files (**throw.wav**, **splat.wav**, **ouch.wav**, **gameover.wav**).

The file named **Snowball.exe** (33 KB in size) is the executable version of the application. (Notice the file has a “plain vanilla” form icon next to its listing – we will change that soon.) If I make sure Visual C# is not running and double-click this file, the following appears:



Voila! The **Snowball Toss Game** project is running outside of the Visual C# IDE!

So distributing a Visual C# application is as simple as giving your user a copy of the executable file (and other needed files), having them place it in a folder on their computer and double-clicking the file to run it? Maybe. This worked on my computer (and will work on yours) because I have a very important set of files known as the **.NET Framework** installed (they are installed when Visual C# is installed). Every Visual C# application needs the .NET Framework to be installed on the hosting computer. The .NET Framework is central to Microsoft's .NET initiative. It is an attempt to solve the problem of first determining what language (and version) an application was developed in and making sure the proper runtime files were installed. The .NET Framework supports all Visual Studio languages, so it is the only runtime software need by Visual Studio applications.

The next question is: how do you know if your user has the .NET Framework installed on his or her computer? And, if they don't, how can you get it installed? These are difficult questions. For now, it is best to assume your user does not have the .NET Framework on their computer. It is new technology that will take a while to get disseminated. Once the .NET Framework is included with new Windows operating systems, most users will have the .NET Framework and it can probably be omitted from any distribution package.

So, in addition to our application's executable file, we also need to give a potential user the Microsoft .NET Framework files and inform them how to install and register these files on their computer. Things are getting complicated. Further complications for application distribution are inclusion and installation of ancillary data files, graphics files and configuration files. Fortunately, Visual C# offers help in distributing, or **deploying**, applications.

Visual C# uses **Setup Wizard** for deploying applications. **Setup Wizard** will identify all files needed by your application and bundle these files into a **Setup** program. You distribute this program to your user base (usually on a CD-ROM). Your user then runs the resulting **Setup** program. This program will:

- Install the application (and all needed files) on the user's computer.
- Add an entry to the user's **Start/Programs** menu to allow execution of your application.
- Add an icon to the user's desktop to allow execution of your application.

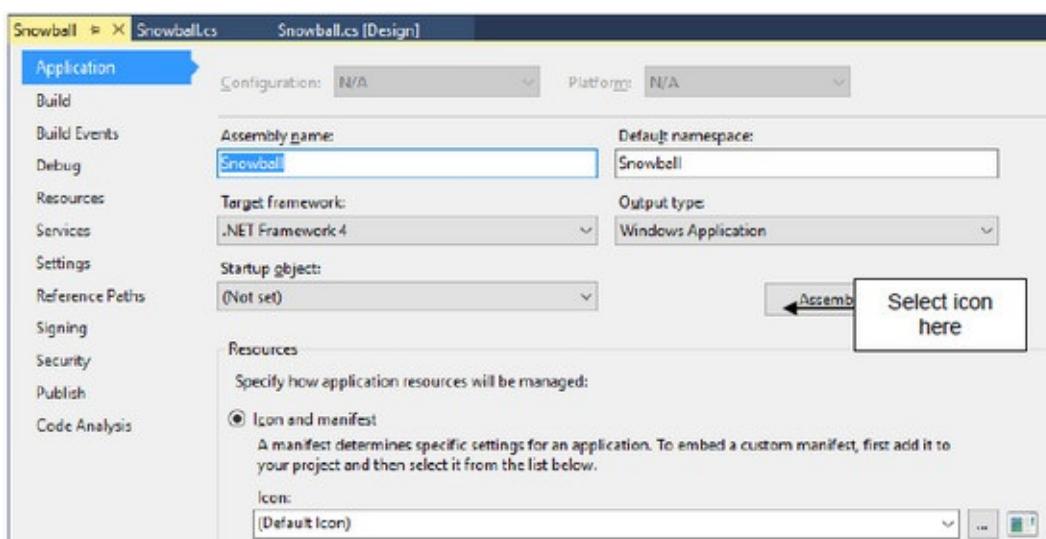
We'll soon look at use of **Setup Wizard** to build a deployment package for a Visual C# application. First, let's quickly look at the topic of icons.

# Application Icons

Recall there is an icon that looks like a little, blank Windows form associated with the project executable. And, notice that whenever you design a form in the Visual C# IDE (and run it), a small icon appears in the upper left hand corner of the form. Icons are used in several places in Visual C# applications: to represent files in **My Computer**, to represent programs in the **Programs** menu, and to represent programs on the desktop. Icons are used throughout applications. The default icons are ugly! We need the capability to change them.

Changing the icon connected to a form is simple. The idea is to assign a unique icon to indicate the form's function. To assign an icon, click on the form's **Icon** property in the properties window. Click on the ellipsis (...) and a window that allows selection of icon files will appear. The icon file you load must have the **ico** filename extension and format.

A different icon can be assigned to the application. This will be the icon that appears next to the executable file's name in Windows Explorer, in the Programs menu and on the desktop. Let's do this for the snowball toss game project. Open the project in Visual C#. Go to the **Solution Explorer** window and highlight the project name (**Snowball**). Choose the **Project** menu item and select **Snowball Properties**. This properties window will appear:



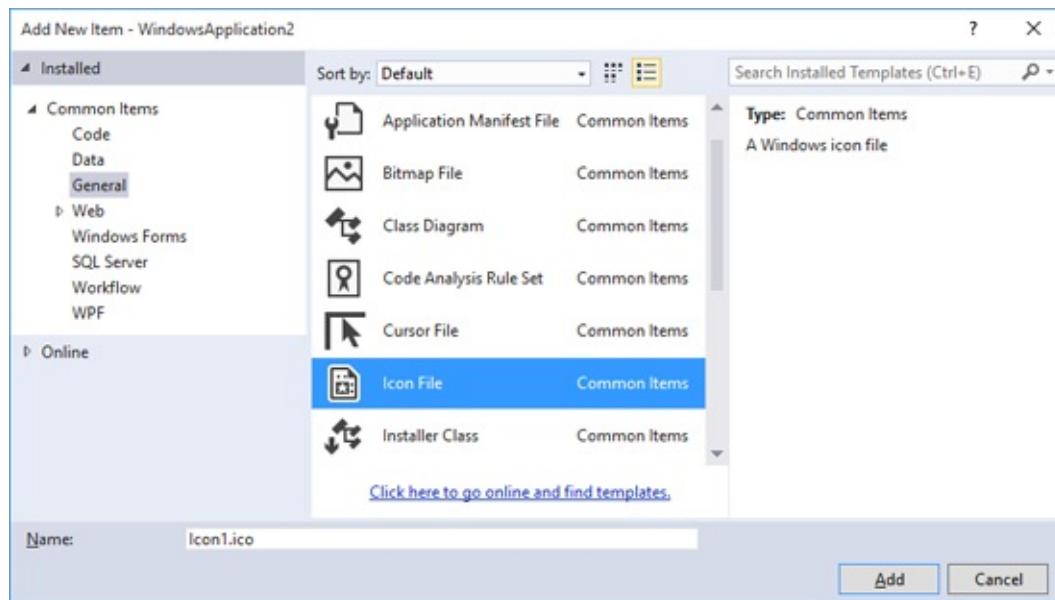
The icon is selected in the **Icon:** drop-down box. You can either choose an icon

already listed or click <Browse> which allows you to select an icon using a dialog box. Once you choose an icon, two things will happen. The icon will appear on the property pages and the icon file will be added to your project's folder. This will be seen in the **Solution Explorer** window.

The Internet and other sources offer a wealth of icon files from which you can choose an icon to assign to your form(s) and applications. But, it's also fun to design your own icon to add that personal touch.

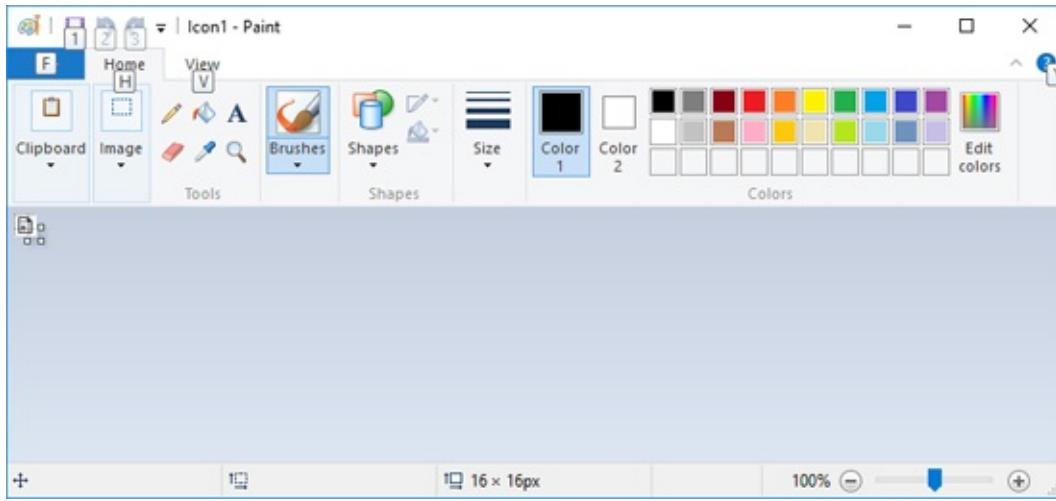
It is possible to create your own icon using Visual Studio. To do this, you need to understand use of the **Microsoft Paint** tool. We will show you how to create a template for an icon and open and save it in Paint. To do this, we assume you have some basic knowledge of using Paint. For more details on how to use Paint, go to the internet and search for tutorials.

To create an icon for a particular project, in **Solution Explorer**, right-click the project name, choose **Add**, then **New Item**. This window will appear:

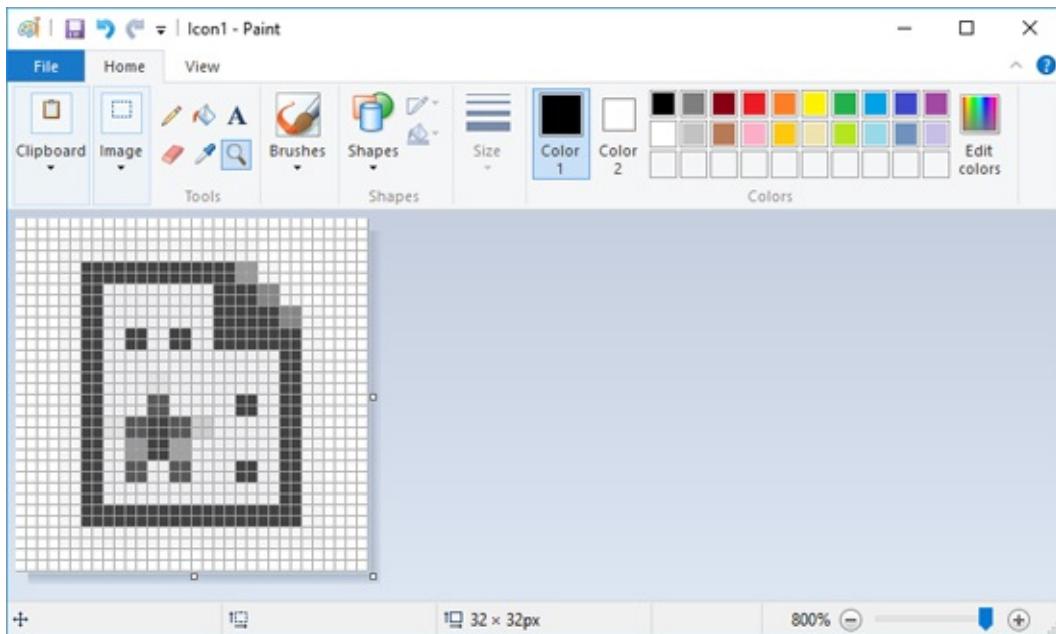


As shown, expand **Common Items** and choose **General**. Then, pick **Icon File**. Name your icon and click **Add**.

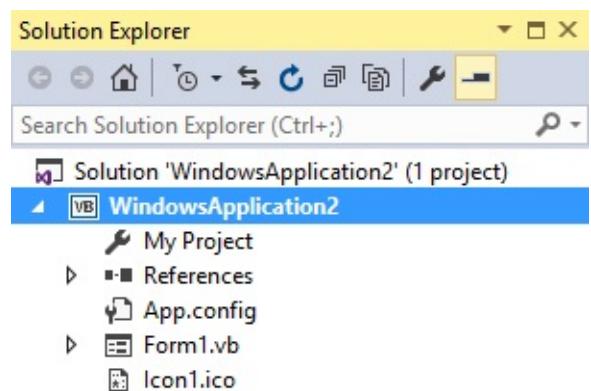
A generic icon will open in the **Microsoft Paint** tool:



The icon is very small. Let's make a few changes to make it visible and editable. First, resize the image to 32 x 32 pixels. Then, use the magnifying tool to make the image as large as possible. Finally, add a grid to the graphic. When done, I see:



At this point, we can add any detail we need by setting pixels to particular colors. Once done, the icon is saved by using the **File** menu. The icon will be saved in your project file and can be used by your project. The icon file (**Icon1.ico** in this case) is also listed in **Solution Explorer**:



# Setup Wizard

As mentioned earlier, to allow someone else to install and run your Visual C# database application requires more than just a simple transfer of the executable file. Visual C# provides **Setup Wizard** that simplifies this task of application **deployment**.

**Note:** **Setup Wizard** must be a part of your Visual Studio installation. To download and install **Setup Wizard**, use the following link:

<https://visualstudiogallery.msdn.microsoft.com/f1cc3f3e-c300-40a7-8797-c509fb8933b9>

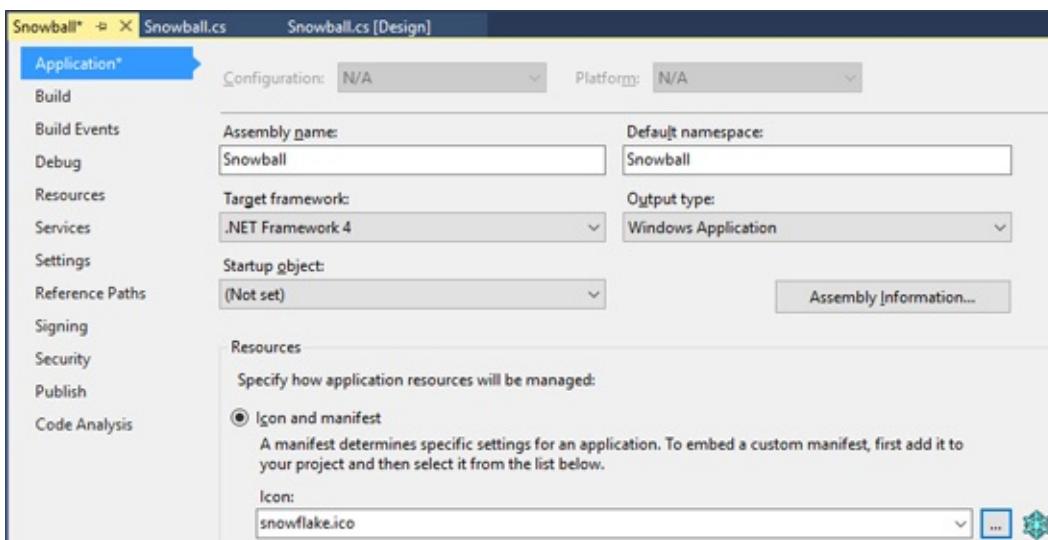
**Setup Wizard** will build a **Setup** program that lets the user install the application (and other needed files) on their computer. At the same time, **Program** menu entries, desktop icons and application removal programs are placed on the user's computer.

The best way to illustrate use of **Setup Wizard** is through an example. In these notes, we will build a Setup program for our **Snowball Toss** example. Follow the example closely to see all steps involved. All results of this example will be found in the **Snowball** and **Snowball Toss** folders of the **HomeVCS\HomeVCS Projects** folder. Let's start.

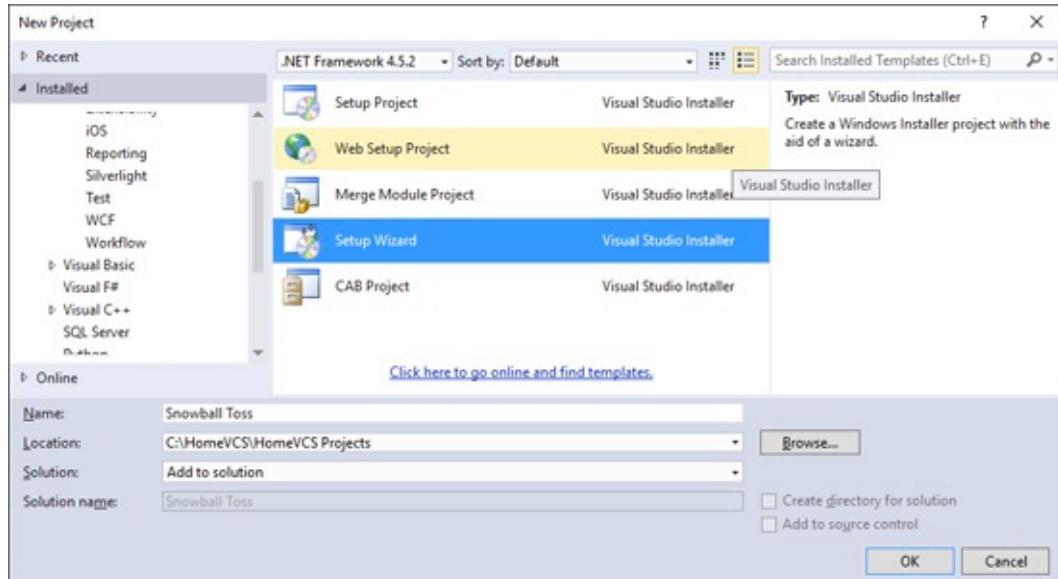
Open the **Snowball** project. Attach an icon to the form (one possible icon, **snowflake.ico**, is included in the project folder). The top of the form should look like this with its new icon:



Assign this same icon to the application using the steps mentioned earlier:

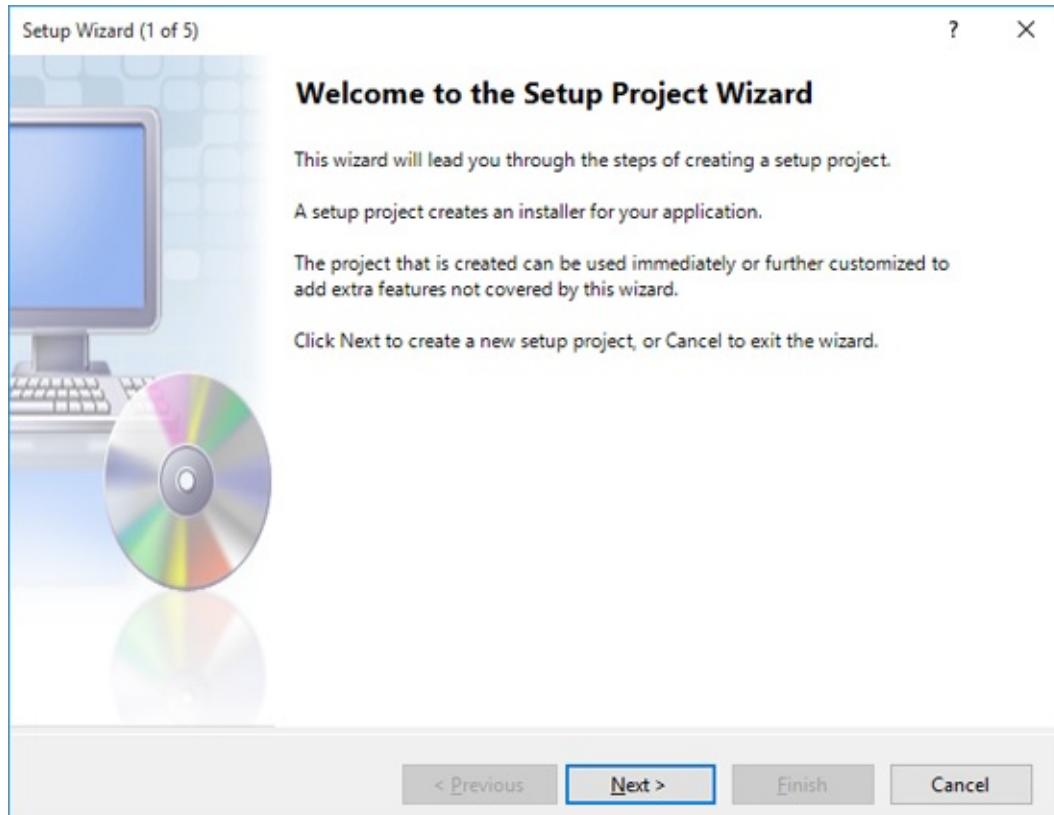


**Setup Wizard** is a separate project you add to your application solution. Choose the **File** menu option, then **New** then **Project**. In the window that appears, expand the **Other Project Types**, select **Visual Studio Installer** and this window appears:



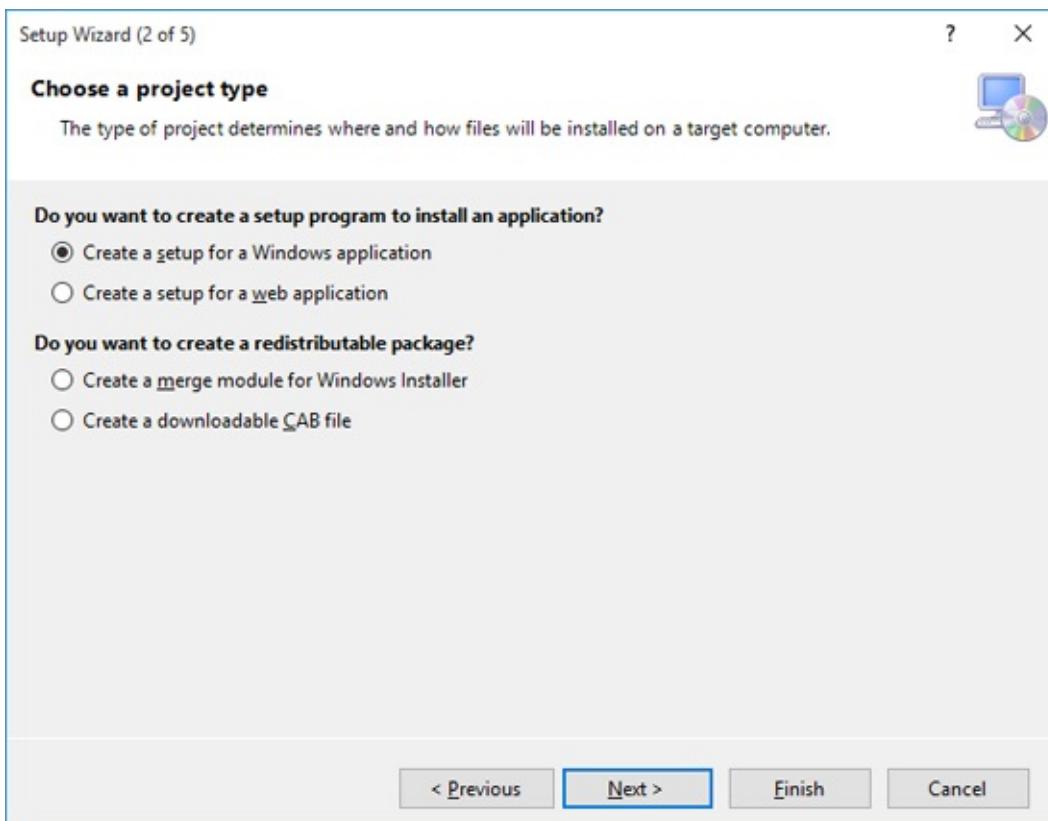
As shown, choose **Setup Wizard** on the right. Under **Solution**, choose **Add to Solution**. Name the project **Snowball Toss** and click **OK**. Notice I have put the project folder in the **HomeVCS\HomeVCS Projects** folder.

The **Setup Wizard** will begin with Step 1 of 5.



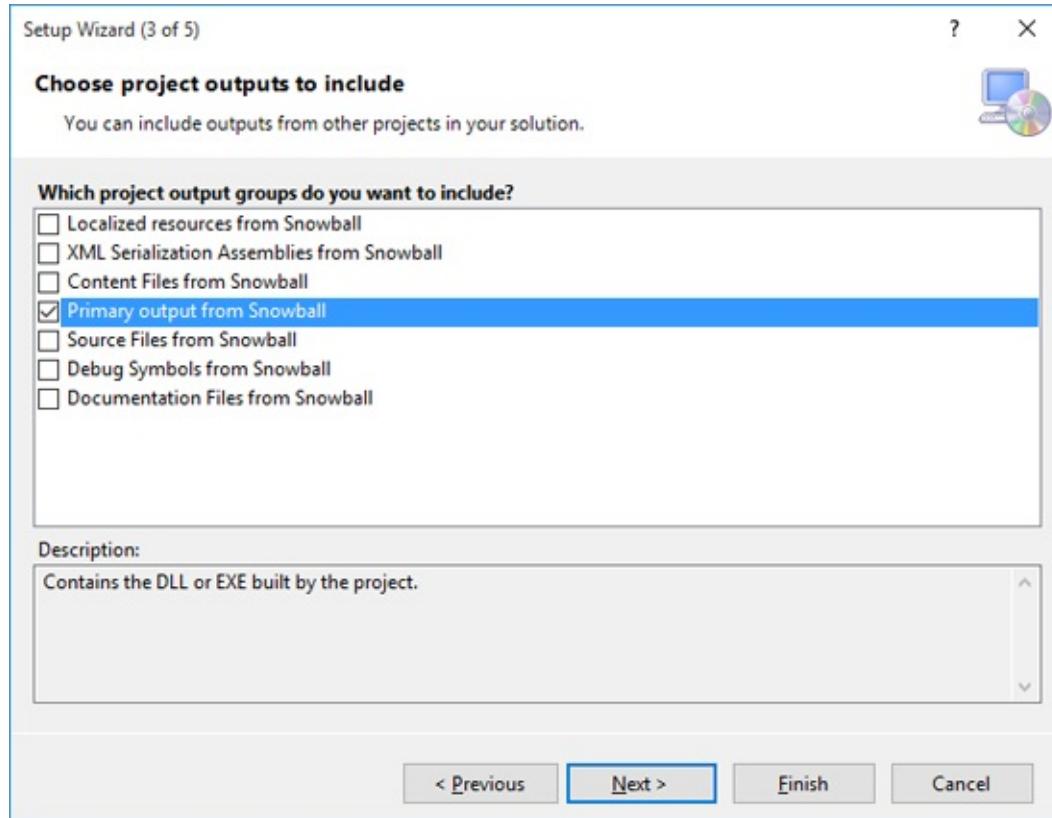
Continue from step to step, providing the requested information. Here, just click **Next**.

## Step 2.



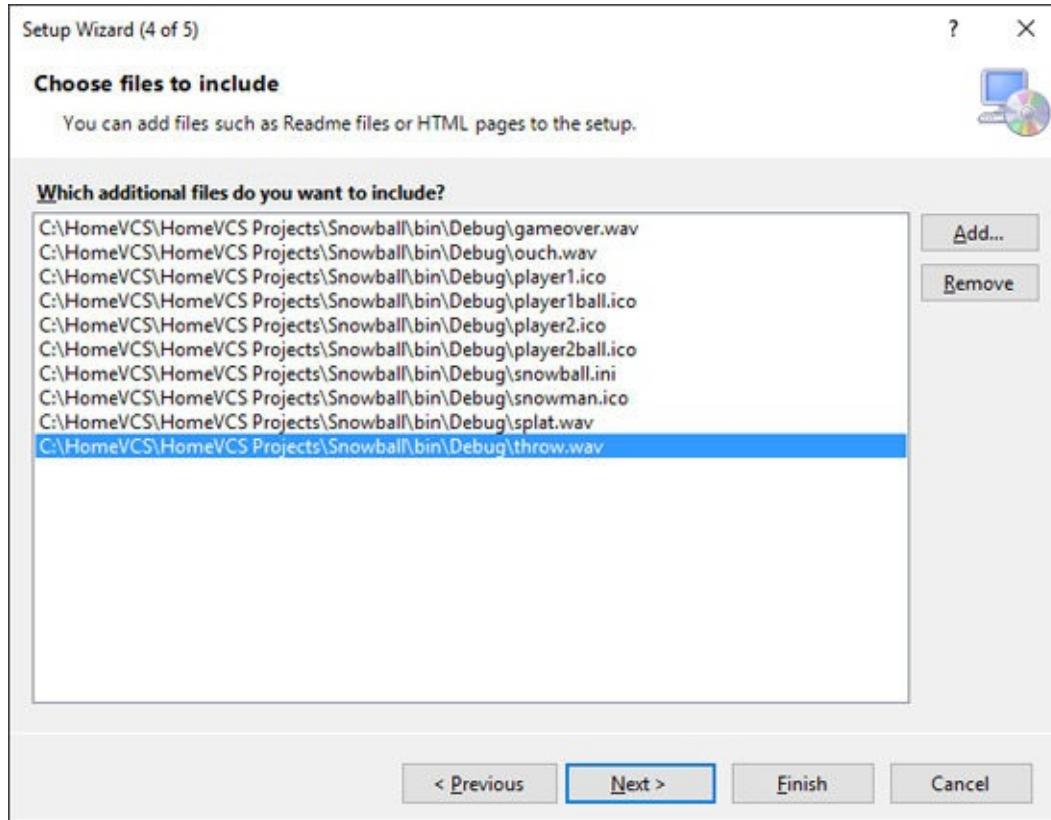
Choose **Create a setup for a Windows application**. Click **Next**.

## Step 3.



Here you choose the files to install. The main one is the executable file (known here as the **primary output file**). Place a check next to **Primary ouput from Snowball** and click **Next**.

#### Step 4.



Here you add additional files with your deployment package. We need these files:

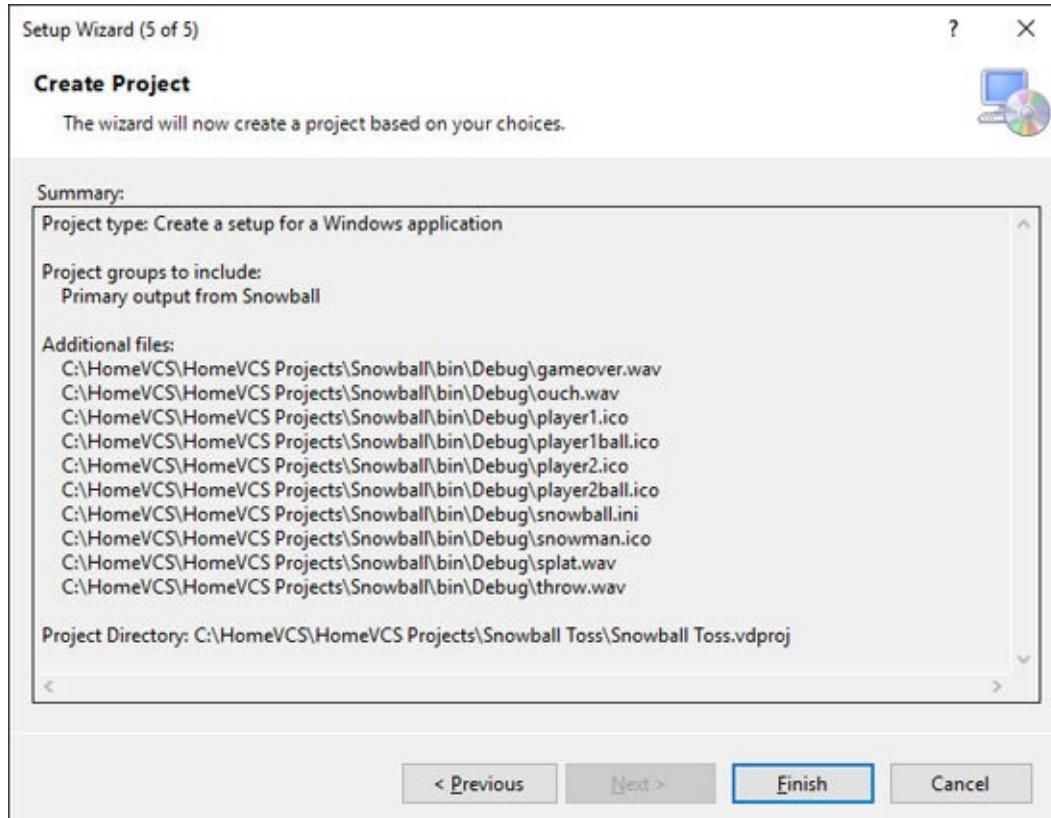
**snowball.ini**

**player1.ico, player2.ico, snowball1.ico, snowball2.ico, snowman.ico  
throw.wav, splat.wav, ouch.wav, gameover.wav**

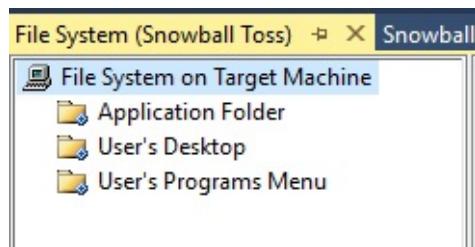
Click **Add** and select each of these files (in the **HomeVCS\HomeVCS Projects\Snowball\bin\Debug** folder).

Move to the next step (click **Next**).

**Step 5.**



Click **Finish** to see the resulting **File System**:



We also want shortcuts to start the program both on the **Desktop** and the **Programs Menu**. First, we do the **Desktop** shortcut. To do this, open the **Application Folder** to see:

File System on Target Machine	
Application Folder	
User's Desktop	
User's Programs Menu	
gameover.wav	File
ouch.wav	File
player1.ico	File
player1ball.ico	File
player2.ico	File
player2ball.ico	File
Primary output fro...	Output
snowball.ini	File
snowman.ico	File
splat.wav	File
throw.wav	File

Right-click **Primary output from ...** and choose **Create Shortcut to Primary output** .... Cut the resulting shortcut from the **Application Folder** and paste it into the **User's Desktop** folder:

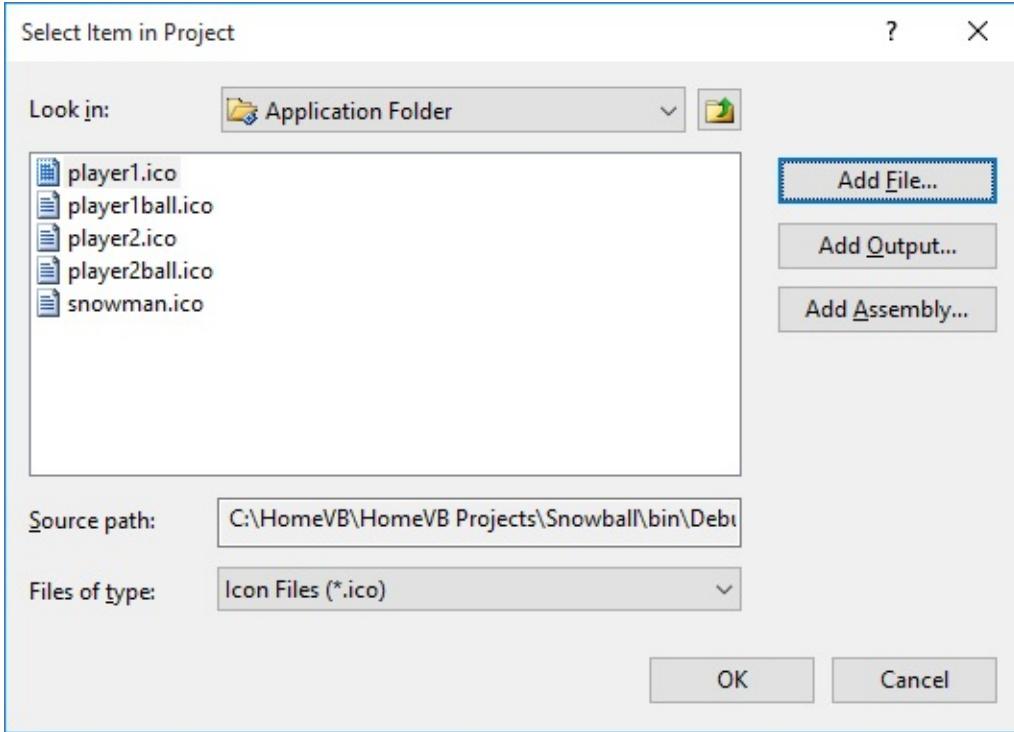
File System on Target Machine	
Application Folder	
User's Desktop	
User's Programs Menu	
Shortcut to Primary...	Shortcut

Rename the shortcut **Snowball Toss** to yield

File System on Target Machine	
Name	Type
Application Folder	
User's Desktop	
User's Programs Menu	
Snowball Toss	Shortcut

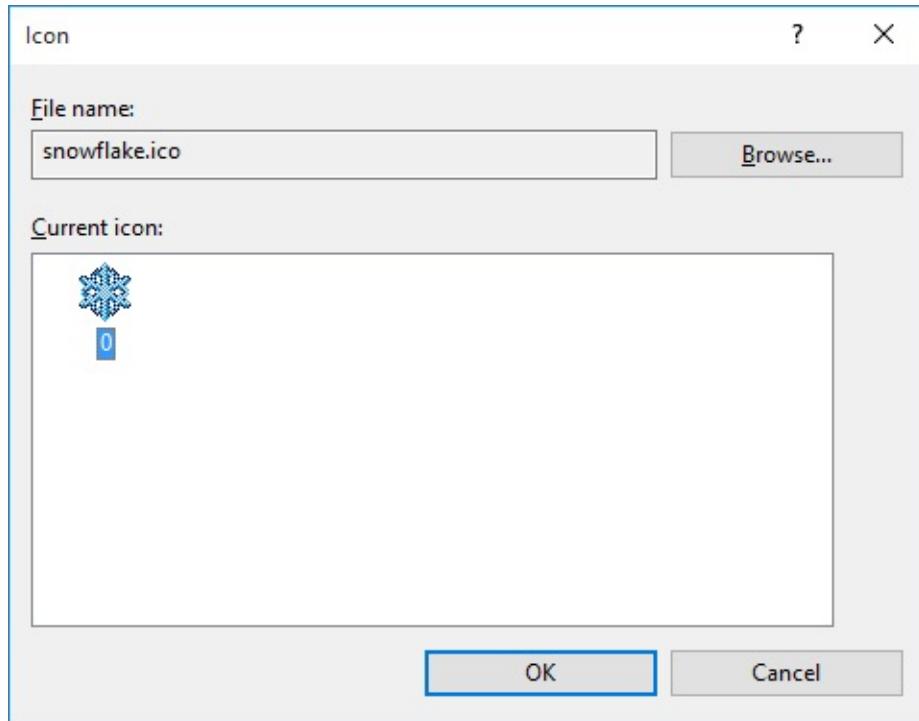
Lastly, we want to change the icon associated with the shortcut. This is a little tricky. The steps are:

- Highlight the shortcut and choose the **Icon** property in the Properties window
- Choose **Browse**.
- When the **Icon** dialog box appears, click **Browse**. You will see



As shown, look in the **Application Folder** and click **Add File**.

- Locate and select the icon file (there is one in the **HomeVCS\HomeVCS Projects\Snowball** folder; it is **snowflake.ico**), then click **OK**. The **Icon** window will appear:

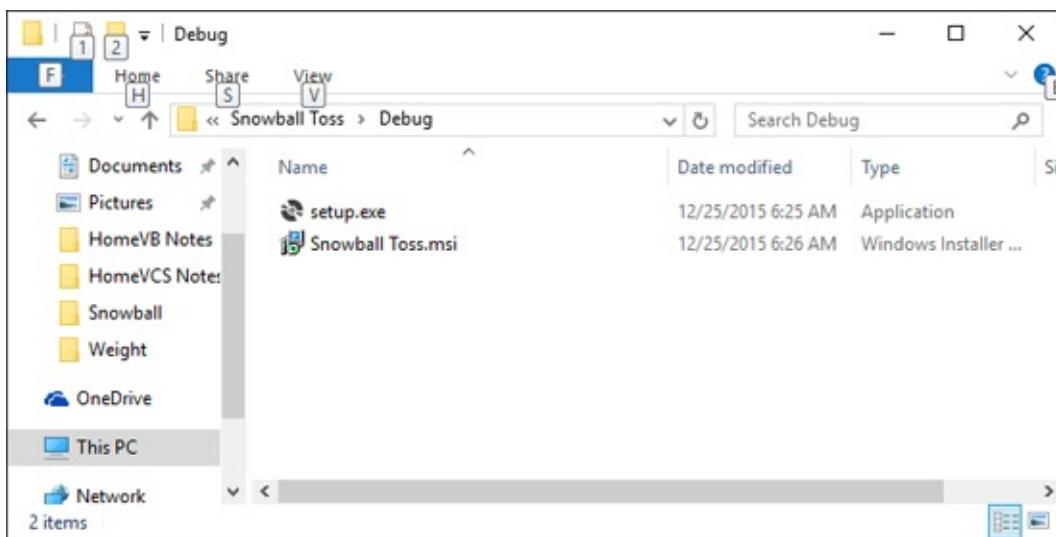


Select the desired icon and click **OK**.

Next, follow nearly identical steps to put a shortcut in the **User's Programs Menu** folder. The **Setup Wizard** has completed its job.

# Building the Setup Program

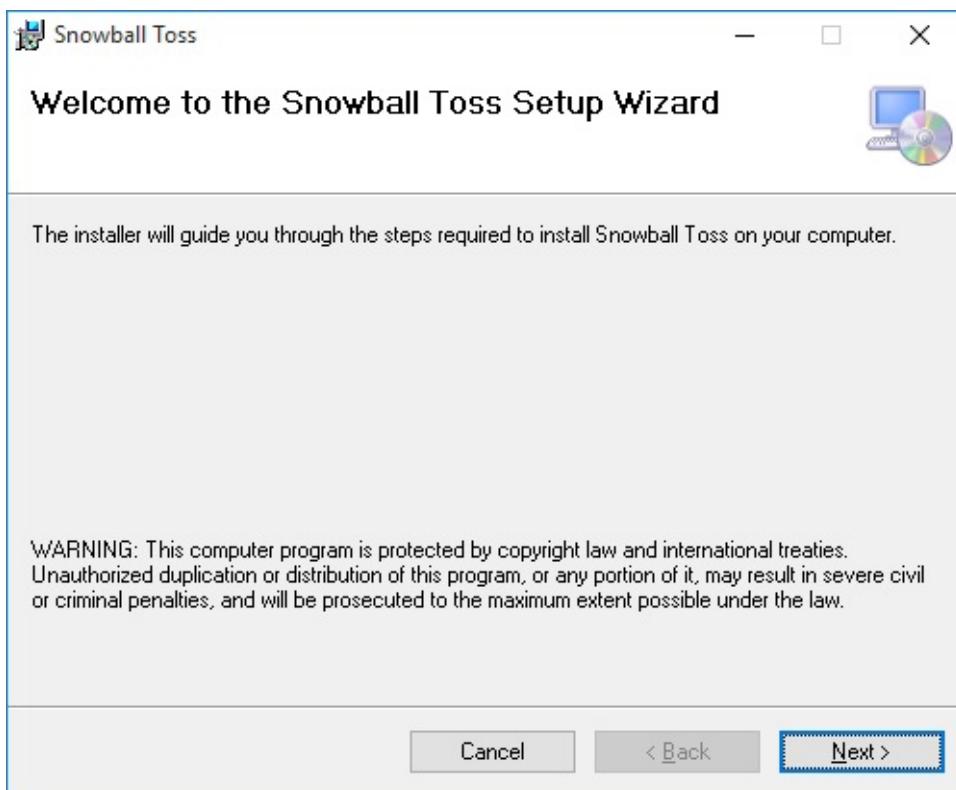
Now, let's build the **Setup** program. In the Solution Explorer window, right-click the **Snowball Toss** project and choose **Build** from the menu. After a short time, the **Setup** program and an msi (Microsoft Installer) file will be written. They will be located in the executable folder of the **Snowball Toss** project folder (**HomeVCS\HomeVCS Projects\Snowball Toss\Debug**). The **Setup** program is small. A look at the resulting directory shows:



Use some media (zip disk, CD-ROM or downloaded files) to distribute these files your user base. Provide the user with the simple instruction to run the **Setup.exe** program and installation will occur.

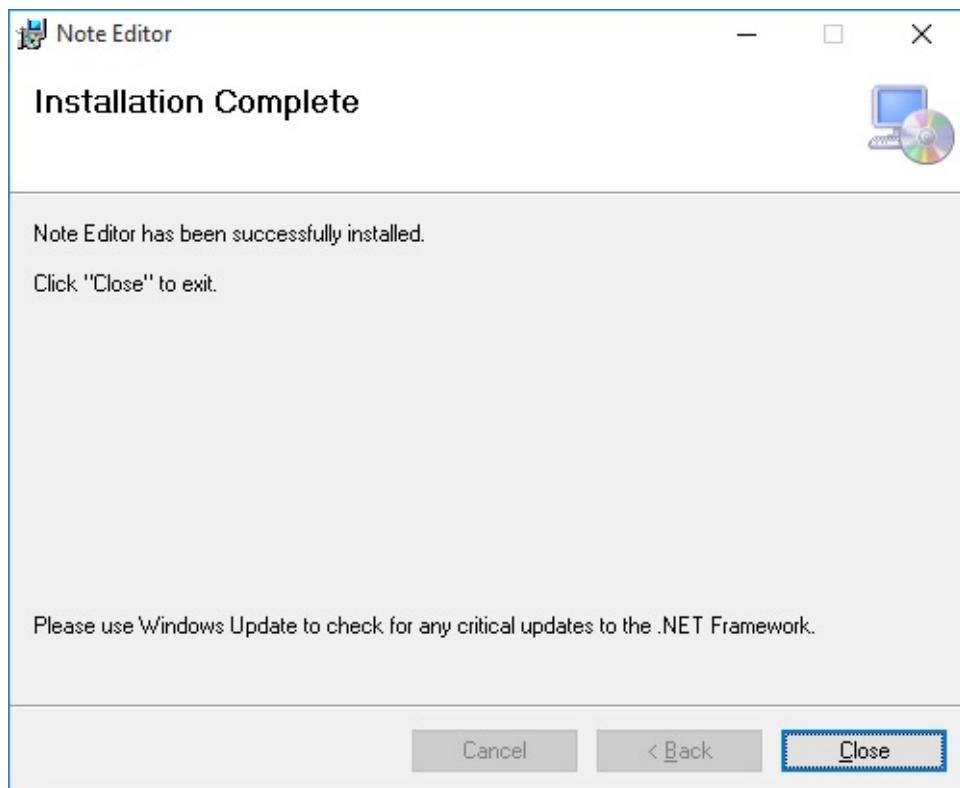
# Installing a Visual C# Application

To install the program, simply run the **Setup.exe** program. These are the same brief instructions you need to provide a user. Users have become very familiar with installing software and running Setup type programs. Let's try the example just created and see what a nice installation interface is provided. Double-click the **Setup.exe** program and this introduction window should appear:



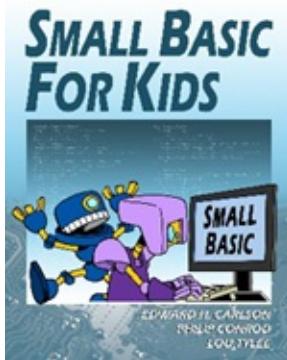
Click **Next** and you will be asked where you want the application installed.

After a few clicks, installation is complete and you will see:

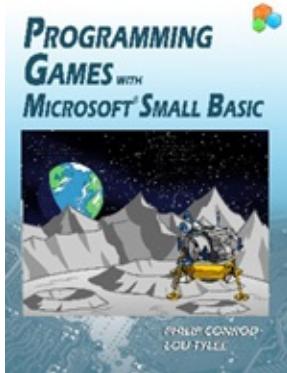


After installing, look on your desktop. There should be an icon named **Snowball Toss**. Double-click that icon and the program will run. Similarly, the program can be accessed by clicking **Start** on your taskbar, then choosing **All Apps**. Click the **Snowball Toss** entry and the program runs. I think you'll agree the installer does a nice job.

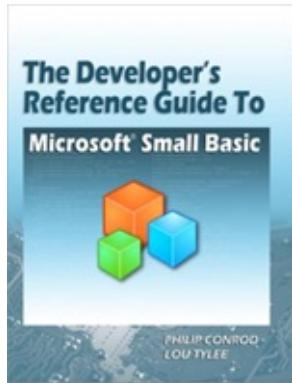
## More Self-Study or Instructor-Led Computer Programming Tutorials by Kidware Software



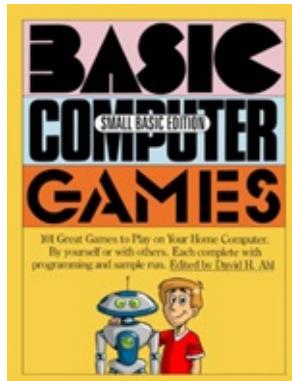
**Small Basic For Kids** is an illustrated introduction to computer programming that provides an interactive, self-paced tutorial to the new Small Basic programming environment. The book consists of 30 short lessons that explain how to create and run a Small Basic program. Elementary students learn about program design and many elements of the Small Basic language. Numerous examples are used to demonstrate every step in the building process. The tutorial also includes two complete games (Hangman and Pizza Zapper) for students to build and try. Designed for kids ages 8+.



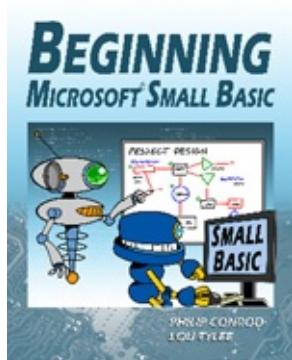
**Programming Games with Microsoft Small Basic** is a self-paced second semester "intermediate" level programming tutorial consisting of 10 chapters explaining (in simple, easy-to-follow terms) how to write video games in Microsoft Small Basic. The games built are non-violent, family-friendly, and teach logical thinking skills. Students will learn how to program the following Small Basic video games: Safecracker, Tic Tac Toe, Match Game, Pizza Delivery, Moon Landing, and Leap Frog. This intermediate level self-paced tutorial can be used at home or school.



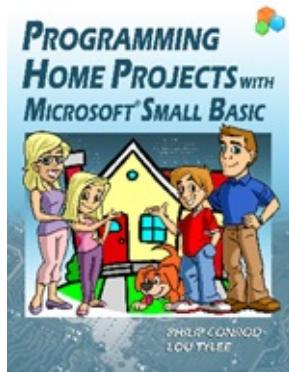
**The Developer's Reference Guide to Microsoft Small Basic** While developing all the different Microsoft Small Basic tutorials we found it necessary to write The Developer's Reference Guide to Microsoft Small Basic. The Developer's Reference Guide to Microsoft Small Basic is over 500 pages long and includes over 100 Small Basic programming examples for you to learn from and include in your own Microsoft Small Basic programs. It is a detailed reference guide for new developers.



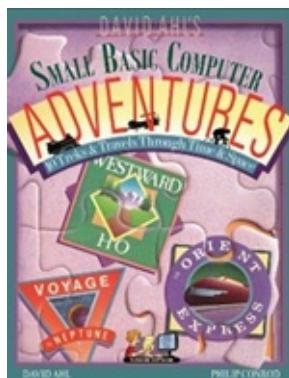
**Basic Computer Games - Small Basic Edition** is a re-make of the classic BASIC COMPUTER GAMES book originally edited by David H. Ahl. It contains 100 of the original text based BASIC games that inspired a whole generation of programmers. Now these classic BASIC games have been re-written in Microsoft Small Basic for a new generation to enjoy! The new Small Basic games look and act like the original text based games. The book includes all the original spaghetti code and GOTO commands!



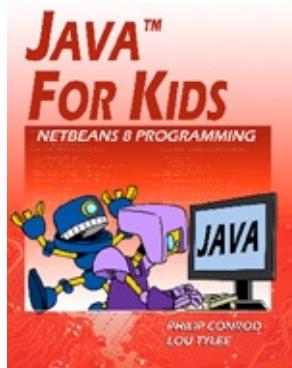
**The Beginning Microsoft Small Basic Programming Tutorial** is a self-study first semester "beginner" programming tutorial consisting of 11 chapters explaining (in simple, easy-to-follow terms) how to write Microsoft Small Basic programs. Numerous examples are used to demonstrate every step in the building process. The last chapter of this tutorial shows you how four different Small Basic games could port to Visual Basic, Visual C# and Java. This beginning level self-paced tutorial can be used at home or at school. The tutorial is simple enough for kids ages 10+ yet engaging enough for adults.



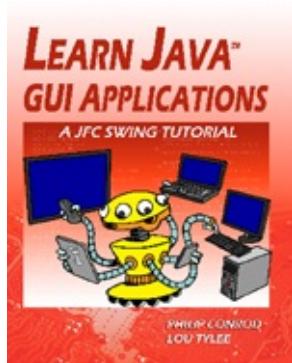
**Programming Home Projects with Microsoft Small Basic** is a self-paced programming tutorial explains (in simple, easy-to-follow terms) how to build Small Basic Windows applications. Students learn about program design, Small Basic objects, many elements of the Small Basic language, and how to debug and distribute finished programs. Sequential file input and output is also introduced. The projects built include a Dual-Mode Stopwatch, Flash Card Math Quiz, Multiple Choice Exam, Blackjack Card Game, Weight Monitor, Home Inventory Manager and a Snowball Toss Game.



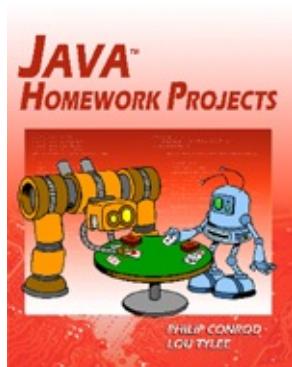
**David Ahl's Small Basic Computer Adventures** is a Microsoft Small Basic re-make of the classic *Basic Computer Games* programming book originally written by David H. Ahl. This new book includes the following classic adventure simulations; Marco Polo, Westward Ho!, The Longest Automobile Race, The Orient Express, Amelia Earhart: Around the World Flight, Tour de France, Subway Scavenger, Hong Kong Hustle, and Voyage to Neptune. Learn how to program these classic computer simulations in Microsoft Small Basic.



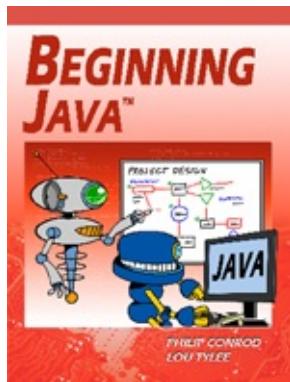
**Java™ For Kids** is a beginning programming tutorial consisting of 10 chapters explaining (in simple, easy-to-follow terms) how to build a Java application. Students learn about project design, object-oriented programming, console applications, graphics applications and many elements of the Java language. Numerous examples are used to demonstrate every step in the building process. The projects include a number guessing game, a card game, an allowance calculator, a state capitals game, Tic-Tac-Toe, a simple drawing program, and even a basic video game. Designed for kids ages 12 and up.



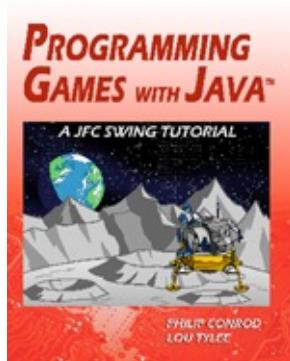
**Learn Java™ GUI Applications** is a 9 lesson Tutorial covering object-oriented programming concepts, using an integrated development environment to create and test Java projects, building and distributing GUI applications, understanding and using the Swing control library, exception handling, sequential file access, graphics, multimedia, advanced topics such as printing, and help system authoring. Our **Beginning Java** or **Java For Kids** tutorial is a pre-requisite for this tutorial.



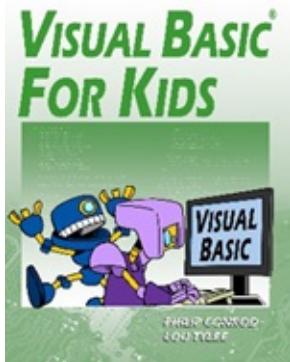
**Java™ Homework Projects** is a Java GUI Swing tutorial covering object-oriented programming concepts. It explains (in simple, easy-to-follow terms) how to build Java GUI project to use around the home. Students learn about project design, the Java Swing controls, many elements of the Java language, and how to distribute finished projects. The projects built include a Dual-Mode Stopwatch, Flash Card Math Quiz, Multiple Choice Exam, Blackjack Card Game, Weight Monitor, Home Inventory Manager and a Snowball Toss Game. Our **Learn Java GUI Applications** tutorial is a pre-requisite for this tutorial.



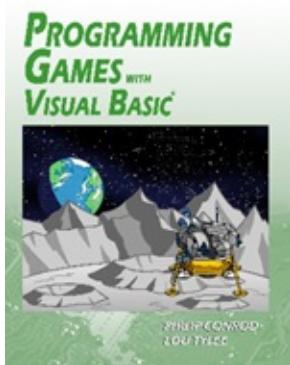
**Beginning Java™** is a semester long "beginning" programming tutorial consisting of 10 chapters explaining (in simple, easy-to-follow terms) how to build a Java application. The tutorial includes several detailed computer projects for students to build and try. These projects include a number guessing game, card game, allowance calculator, drawing program, state capitals game, and a couple of video games like Pong. We also include several college prep bonus projects including a loan calculator, portfolio manager, and checkbook balancer. Designed for students age 15 and up.



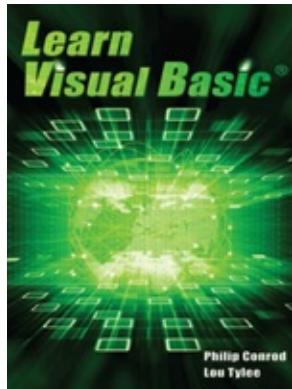
**Programming Games with Java™** is a semester long "intermediate" programming tutorial consisting of 10 chapters explaining (in simple, easy-to-follow terms) how to build a Visual C# Video Games. The games built are non-violent, family-friendly and teach logical thinking skills. Students will learn how to program the following Visual C# video games: Safecracker, Tic Tac Toe, Match Game, Pizza Delivery, Moon Landing, and Leap Frog. This intermediate level self-paced tutorial can be used at home or school. The tutorial is simple enough for kids yet engaging enough for beginning adults. Our **Learn Java GUI Applications** tutorial is a required pre-requisite for this tutorial.



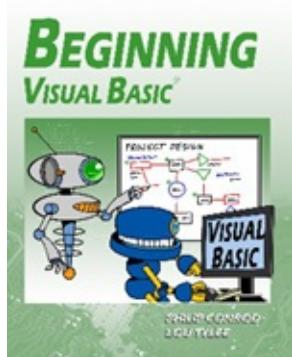
**Visual Basic® For Kids** is a beginning programming tutorial consisting of 10 chapters explaining (in simple, easy-to-follow terms) how to build a Visual Basic Windows application. Students learn about project design, the Visual Basic toolbox, and many elements of the BASIC language. The tutorial also includes several detailed computer projects for students to build and try. These projects include a number guessing game, a card game, an allowance calculator, a drawing program, a state capitals game, Tic-Tac-Toe and even a simple video game. Designed for kids ages 12 and up.



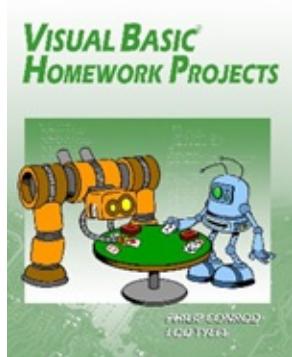
**Programming Games with Visual Basic®** is a semester long "intermediate" programming tutorial consisting of 10 chapters explaining (in simple, easy-to-follow terms) how to build Visual Basic Video Games. The games built are non-violent, family-friendly, and teach logical thinking skills. Students will learn how to program the following Visual Basic video games: Safecracker, Tic Tac Toe, Match Game, Pizza Delivery, Moon Landing, and Leap Frog. This intermediate level self-paced tutorial can be used at home or school. The tutorial is simple enough for kids yet engaging enough for beginning adults.



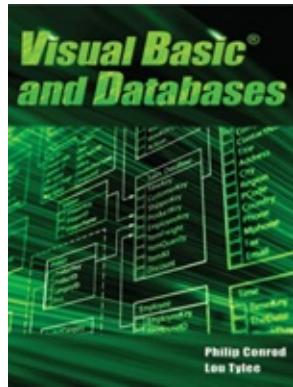
**LEARN VISUAL BASIC** is a comprehensive college level programming tutorial covering object-oriented programming, the Visual Basic integrated development environment, building and distributing Windows applications using the Windows Installer, exception handling, sequential file access, graphics, multimedia, advanced topics such as web access, printing, and HTML help system authoring. The tutorial also introduces database applications (using ADO .NET) and web applications (using ASP.NET).



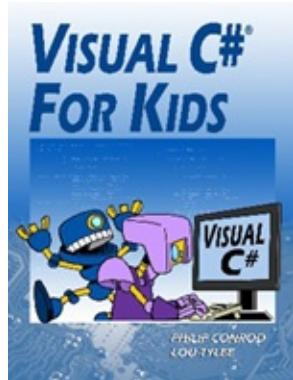
**Beginning Visual Basic®** is a semester long self-paced "beginner" programming tutorial consisting of 10 chapters explaining (in simple, easy-to-follow terms) how to build a Visual Basic Windows application. The tutorial includes several detailed computer projects for students to build and try. These projects include a number guessing game, card game, allowance calculator, drawing program, state capitals game, and a couple of video games like Pong. We also include several college prep bonus projects including a loan calculator, portfolio manager, and checkbook balancer. Designed for students age 15 and up.



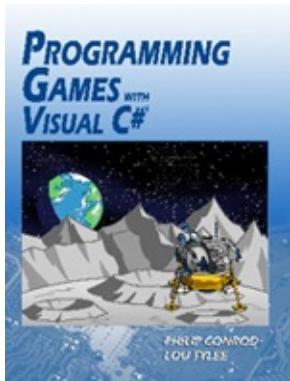
**Visual Basic® Homework Projects** is a semester long self-paced programming tutorial explains (in simple, easy-to-follow terms) how to build a Visual Basic Windows project. Students learn about project design, the Visual Basic toolbox, many elements of the Visual Basic language, and how to debug and distribute finished projects. The projects built include a Dual-Mode Stopwatch, Flash Card Math Quiz, Multiple Choice Exam, Blackjack Card Game, Weight Monitor, Home Inventory Manager and a Snowball Toss Game.



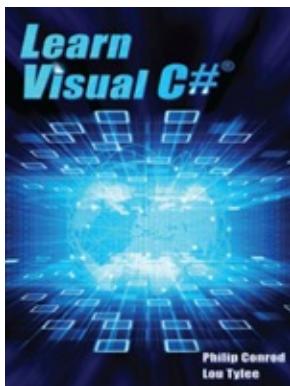
**VISUAL BASIC AND DATABASES** is a tutorial that provides a detailed introduction to using Visual Basic for accessing and maintaining databases for desktop applications. Topics covered include: database structure, database design, Visual Basic project building, ADO .NET data objects (connection, data adapter, command, data table), data bound controls, proper interface design, structured query language (SQL), creating databases using Access, SQL Server and ADOX, and database reports. Actual projects developed include a book tracking system, a sales invoicing program, a home inventory system and a daily weather monitor.



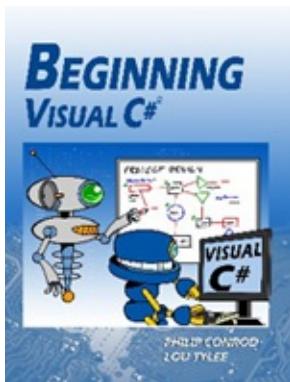
**Visual C#® For Kids** is a beginning programming tutorial consisting of 10 chapters explaining (in simple, easy-to-follow terms) how to build a Visual C# Windows application. Students learn about project design, the Visual C# toolbox, and many elements of the C# language. Numerous examples are used to demonstrate every step in the building process. The projects include a number guessing game, a card game, an allowance calculator, a drawing program, a state capitals game, Tic-Tac-Toe and even a simple video game. Designed for kids ages 12+.



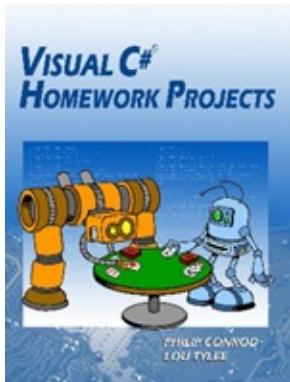
**Programming Games with Visual C#®** is a semester long "intermediate" programming tutorial consisting of 10 chapters explaining (in simple, easy-to-follow terms) how to build a Visual C# Video Games. The games built are non-violent, family-friendly and teach logical thinking skills. Students will learn how to program the following Visual C# video games: Safecracker, Tic Tac Toe, Match Game, Pizza Delivery, Moon Landing, and Leap Frog. This intermediate level self-paced tutorial can be used at home or school. The tutorial is simple enough for kids yet engaging enough for beginning adults



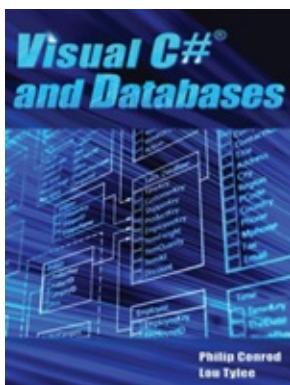
**LEARN VISUAL C#** is a comprehensive college level computer programming tutorial covering object-oriented programming, the Visual C# integrated development environment and toolbox, building and distributing Windows applications (using the Windows Installer), exception handling, sequential file input and output, graphics, multimedia effects (animation and sounds), advanced topics such as web access, printing, and HTML help system authoring. The tutorial also introduces database applications (using ADO .NET) and web applications (using ASP.NET).



**Beginning Visual C#®** is a semester long “beginning” programming tutorial consisting of 10 chapters explaining (in simple, easy-to-follow terms) how to build a C# Windows application. The tutorial includes several detailed computer projects for students to build and try. These projects include a number guessing game, card game, allowance calculator, drawing program, state capitals game, and a couple of video games like Pong. We also include several college prep bonus projects including a loan calculator, portfolio manager, and checkbook balancer. Designed for students ages 15+.



**Visual C#® Homework Projects** is a semester long self-paced programming tutorial explains (in simple, easy-to-follow terms) how to build a Visual C# Windows project. Students learn about project design, the Visual C# toolbox, many elements of the Visual C# language, and how to debug and distribute finished projects. The projects built include a Dual-Mode Stopwatch, Flash Card Math Quiz, Multiple Choice Exam, Blackjack Card Game, Weight Monitor, Home Inventory Manager and a Snowball Toss Game.



**VISUAL C# AND DATABASES** is a tutorial that provides a detailed introduction to using Visual C# for accessing and maintaining databases for desktop applications. Topics covered include: database structure, database design, Visual C# project building, ADO .NET data objects (connection, data adapter, command, data table), data bound controls, proper interface design, structured query language (SQL), creating databases using Access, SQL Server and ADOX, and database reports. Actual projects developed include a book tracking system, a sales invoicing program, a home inventory system and a daily weather monitor.

This book was downloaded from AvaxHome!

Visit my blog for more new books:

[www.avxhm.se/blogs/AlenMiler](http://www.avxhm.se/blogs/AlenMiler)