Locations of **VideoNotes**

www.pearson.com/cs-resources

VideoNote

Brief Contents

Contents

Preface

Welcome to *Starting Out with Visual C#*, Fifth Edition. This book is intended for an introductory programming course and is ideal for students with no prior experience. Students who are new to programming will appreciate the clear, down-to-earth explanations and the detailed walk-throughs that are provided by the hands-on tutorials. More experienced students will appreciate the depth of detail as they learn about the .NET Framework, databases, Language-Integrated Query, and other topics.

As with all the books in the *Starting Out With* series, the hallmark of this text is its clear, friendly, and easy-to-understand writing. In addition, it is rich in example programs that are concise and practical. The programs in this book include short examples that highlight specific programming topics, as well as more involved examples that focus on problem solving. Each chapter provides numerous hands-on tutorials that guide the student through each step of the development of an application. In addition to detailed, step-by-step instructions, the tutorials also provide the application's completed code and screen captures of the completed forms.

New to This Edition

The biggest changes in this edition appear in the last half of the book. Some of the chapters have been reorganized, and two new chapters have been added. Here is a summary of the reorganization:

- **The ImageList control is now covered along with Arrays and Lists in Chapter 7.** Since an ImageList is a collection of images, it is fitting that this control be covered in the same chapter as Arrays and Lists.
- **Chapter 8 has been split into two chapters, with new material added to both of the resulting chapters.** In this edition, Chapter 8 is titled *Text Processing* and Chapter 9 is titled *Structures, Enumerated Types, and Dictionaries*.
- **Two new chapters have been added.** In this edition, Chapter 13 is titled *Delegates and Lambda Expressions* and Chapter 14 is titled *Language-Integrated Query (LINQ)*.

An abundance of new topics and improvements has been added to this edition:

- **`var` Keyword:** This edition introduces the `var` keyword for variable declaration.
- **Advanced String Formatting:** Chapter 8 includes a new section on formatting strings with the `String.Format` method.
- **Working with Dates and Times:** Chapter 9 includes a new section about working with dates and **times**. The section covers the `DateTime` and `TimeSpan` data types, as well as the DateTimePicker control.
- **Dictionaries:** Chapter 9 includes a new section on the `Dictionary` collection. The student learns to store and work with data as key–value pairs in dictionaries.
- **More Use of Auto-Properties:** In the previous editions, property declarations were almost always written with explicitly declared backing fields, even when the properties were simply used to set and get a value. Where possible, those property declarations have been rewritten as auto-properties.
- **Object Initializer Syntax:** Object initializer syntax is introduced in Chapters 8 and 10 as an alternative way to **declare** and initialize structures and class instances.
- **Static Members:** Chapter 10 has a new section on static fields, properties, methods, and classes.
- **Sealed Classes and Methods:** Chapter 11 has a new discussion on using the `sealed` keyword to prevent classes from being inherited from, and to prevent methods from being overridden.

- **Extension Methods:** Chapter 11 covers extension methods, which allow you to extend classes that you cannot inherit from, as well as the primitive data types.
- **Interfaces**: Chapter 11 now includes a section on writing and using interfaces.
- **Delegates:** Chapter 13 introduces the student to delegates.
- **Anonymous Methods:** Chapter 13 also introduces anonymous methods, which go hand-in-hand with delegates.
- **Lambda Expressions:** Chapter 13 concludes by showing the student how to use a lambda expression to concisely create a delegate and anonymous method.
- **Language-Integrated Query (LINQ):** Chapter 14 introduces LINQ, and shows how to query simple data structures such as arrays and Lists using **LINQ to Objects**, and how to query databases using **LINQ to SQL**.

A GUI-Based Approach

Beginning students are more motivated to learn programming when their applications have some sort of graphical element, such as a graphical user interface (GUI). Students using this book will learn to create GUI-based, event-driven, Visual C# applications. The Visual Studio environment is used to create forms that are rich with user interface controls and graphical images.

Learn to Use Objects Early, Learn to Write Classes Later

This book explains what an object is very early and shows the student how to create objects from classes that are provided by the .NET Framework. It then introduces the student to the fundamentals of input and output, control structures, methods, arrays and lists, and file operations. Then, the student learns to write his or her own classes and explores the topics of inheritance and polymorphism.

Brief Overview of Each Chapter

Chapter 1: Introduction to Computers and Programming. This chapter begins by giving a very concrete and easy-to-understand explanation of how computers work, how data is stored and manipulated, and why we write programs in high-level languages. In this chapter, the student learns what an object is and sees several examples by studying the objects that make up a program's GUI. The chapter discusses steps in the programming development cycle. It also gives an introduction to the Visual Studio environment.

Chapter 2: Introduction to Visual C#. In this chapter, the student learns to create forms with labels, buttons, and picture boxes and learns to modify control properties. The student is introduced to C# code and learns the organizational structure of namespaces, classes, and methods. The student learns to write simple event-driven applications that respond to button clicks or provide interaction through clickable images. The importance of commenting code is also discussed.

Chapter 3: Processing Data. This chapter introduces variables and data types. It discusses the use of local variables and variables declared as fields within a form class. The student learns to create applications that read input from TextBox controls, perform mathematical operations, and produce formatted output. The student learns about the exceptions that can occur when the user enters invalid data into a TextBox and learns to write simple exception-handling code to deal with those problems. Named constants are introduced as a way of representing unchanging values and creating self-documenting, maintainable code. The student also learns more intricacies of creating graphical user interfaces. The chapter concludes with a discussion and tutorial on using the Visual Studio debugger to locate logic errors by single-stepping through an application's code.

Chapter 4: Making Decisions. In this chapter, the student learns about relational operators and Boolean expressions and is shown how to control the flow of a program with decision structures. The `if`, `if-else`, and `if-else-if` statements are covered. Nested decision structures, logical operators, and the `switch` statement are also discussed. The student learns to use the `TryParse` family of methods to validate input and prevent exceptions. Radio buttons, check boxes, and list boxes are introduced as ways to let the user select items in a GUI.

Chapter 5: Loops, Files, and Random Numbers. This chapter shows the student how to use loops to create repetition structures. The `while` loop, the `for` loop, and the `do-while` loop are presented. Counters, accumulators, and running totals are also discussed. This chapter also introduces sequential file input and output and using text files. The student learns various programming techniques for writing data to text files and reading the contents of test files. The chapter concludes with a discussion of pseudorandom numbers, their applications, and how to generate them.

Chapter 6: Modularizing Your Code with Methods. In this chapter, the student first learns how to write and call `void` methods as well as value-returning methods. The chapter shows the benefits of using methods to modularize programs and discusses the top-down design approach. Then, the student learns to pass arguments to methods. Passing by value, by reference, and output parameters are discussed. The chapter concludes with a discussion and tutorial on debugging methods with the Visual Studio step-into, step-over, and step-out, commands.

Chapter 7: Arrays and Lists. Arrays and lists are reference-type objects in C# so this chapter begins by discussing the difference between value type and reference type objects in the C# language. Then, the student learns to create and work with single-dimensional and two-dimensional arrays. The student learns to pass arrays as arguments to methods, transfer data between arrays and files, work with partially filled arrays, and create jagged arrays. Many examples of array processing are provided, including examples of finding the sum, average, highest, and lowest values in an array. Finally, the student learns to create `List` objects and store data in them, and use the ImageList control, a data structure for storing and retrieving images.

Chapter 8: Text Processing. This chapter discusses various techniques for working with text. The topics include working with characters, working with substrings, tokenizing strings, and formatting strings.

Chapter 9: Structures, Enumerated Types, and Dictionaries. In this chapter, the student learns to use structures to encapsulate several variables into a single item. The student next learns to create and use enumerated types. Last, the student learns to use the Dictionary collection from the .NET Framework to store and work with data as key–value pairs.

Chapter 10: Introduction to Classes. Up to this point, the student has extensively used objects that are instances of .NET Framework classes. In this chapter, the student learns to write classes to create his or her own objects with fields, methods, and constructors. The student learns how to implement various types of properties within a class, including auto-properties and read-only auto-properties. Creating arrays of objects and storing objects in a `List` are also discussed. A primer on finding the classes in a problem as well as their responsibilities is provided. The chapter shows the student how to create multiple form classes in a project, instantiate those classes, and display them. A tutorial is given where the student creates a multiform application in which the code in one form accesses controls on another form. The chapter concludes by discussing static members and static classes.

Chapter 11: Inheritance, Polymorphism, and Interfaces. The study of classes continues in this chapter with the subjects of inheritance and polymorphism. The topics covered include base classes, derived classes, how constructors functions work in inheritance, method overriding, and polymorphism. Abstract classes and abstract methods are also discussed. The chapter discusses extension methods, and concludes with a section on interfaces.

Chapter 12: Databases. This chapter introduces the student to basic database concepts. The student first learns about tables, rows, and columns and how to create an SQL Server database in Visual Studio. The student then learns how to connect a database to a Visual C# application and display a table in a DataGridView control, a Details view, and other data-bound controls. Finally, the student learns how to write SQL Select statements to retrieve data from a table.

Chapter 13: Delegates and Lambda Expressions. Lambda expressions have become commonplace in C# programming. Before the student can really understand lambda expressions, he or she must first understand delegates and anonymous methods. The goal of this chapter is to build a foundation of knowledge about how delegates and anonymous methods work, and then build an understanding of lambda expressions.

Chapter 14: Language-Integrated Query (LINQ). This chapter begins by introducing the student to LINQ as a tool for querying the data in common data structures such as arrays and Lists, using LINQ to Objects. It discusses both query syntax and method syntax. Several useful LINQ extension methods are also discussed. The chapter concludes with a section on LINQ to SQL and discusses how to use LINQ to query a database.

Appendix A: C# Primitive Data Types. This appendix gives an overview of the primitive data types available in C#.

Appendix B: Additional User Interface Controls. This appendix shows how to create a variety of controls such as ToolTips, combo boxes, scroll bars, TabControls, WebBrowser controls, ErrorProvider components, and menu systems.

Appendix C: ASCII/Unicode Characters. This appendix lists the ASCII (American Standard Code for Information Interchange) character set, which is also the Latin Subset of Unicode.

Appendix D: Answers to Checkpoint Questions. This appendix provides the answers to the Checkpoint questions that appear throughout each chapter in the book.

Appendix E: Installing LINQ to SQL. This appendix shows how to use the Visual Studio Installer to download and install LINQ to SQL.

Organization of the Text

The text teaches Visual C# step by step. Each chapter covers a major set of programming topics, introduces controls and GUI elements, and builds knowledge as the student progresses through the book. Although the chapters can be easily taught in their existing sequence, there is some flexibility. Figure P-1 shows the chapter dependencies. As shown in the figure, Chapters 1–7 present the fundamentals of Visual C# programming and should be covered in sequence. Then, you can move directly to Chapter 8, 9, 10, or 12. Chapter 11 should be covered after Chapter 10. Then Chapter 13 and 14 can be covered in order.

Features of the Text

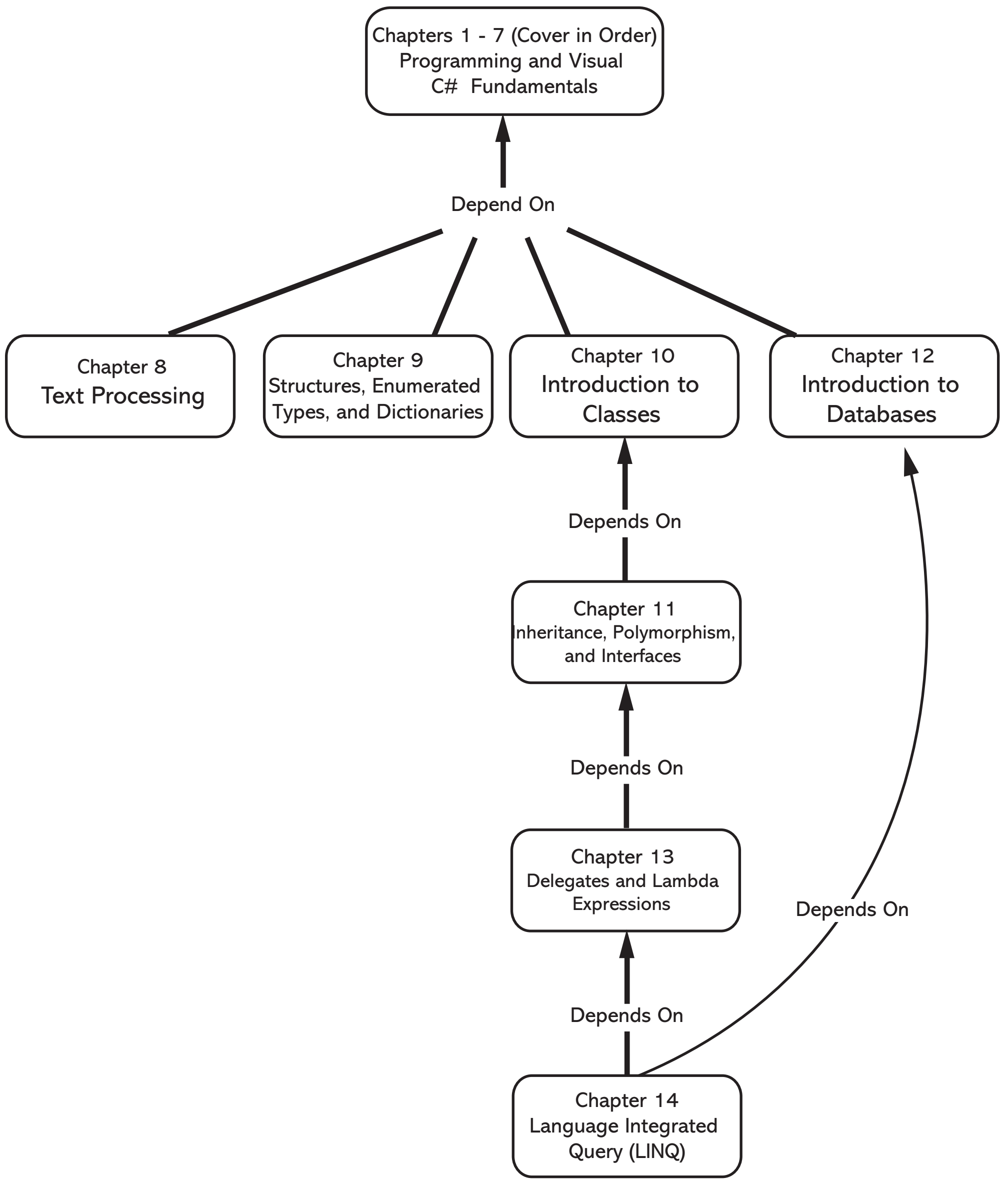
Concept Statements. Each major section of the text starts with a concept statement. This statement concisely summarizes the main point of the section.

Tutorials. Each chapter has several hands-on tutorials that guide the student through the development of an application. Each tutorial provides detailed, step-by-step instructions, as well as the application's completed code and screen captures of the completed forms.

Example Programs. Each chapter has an abundant number of code examples designed to highlight the current topic.

Notes. Notes appear at several places throughout the text. They are short explanations of interesting or often misunderstood points relevant to the topic at hand.

Figure P-1 Chapter dependencies


Chapters 1 - 7 (Cover in Order) Programming and Visual C# Fundamentals
Depend On
Chapter 8 Text Processing
Chapter 9 Structures, Enumerated Types, and Dictionaries
Chapter 10 Introduction to Classes
Chapter 12 Introduction to Databases
Depends On
Chapter 11 Inheritance, Polymorphism, and Interfaces
Depends On
Chapter 13 Delegates and Lambda Expressions
Depends On
Depends On
Chapter 14 Language Integrated Query (LINQ)

Tips. Tips advise the student on the best techniques for approaching different programming or animation problems.

Warnings. Warnings caution students about programming techniques or practices that can lead to malfunctioning programs or lost data.

Checkpoints. Checkpoints are questions placed at intervals throughout each chapter. They are designed to query the student's knowledge quickly after learning a new topic. The answers to the Checkpoint questions can be found in Appendix D.

Review Questions. Each chapter presents a thorough and diverse set of Review Questions. They include Multiple Choice, True/False, Algorithm Workbench, and Short Answer.

Programming Problems. Each chapter offers a pool of Programming Problems designed to solidify the student's knowledge of the topics currently being studied.

VideoNote

VideoNotes. Each tutorial in the book has an accompanying online VideoNote that can be accessed on the book's companion Web site www.pearson.com/gaddis. Students can follow along with the author as he or she works through each tutorial in the videos. Also, one programming problem at the end of each chapter has an accompanying VideoNote that shows the student how to create the solution.

Supplements

Student The following supplementary material is available with the book:

- Source code and files required for the chapter tutorials are available at www.pearson.com/cs-resources
- A complete set of online VideoNotes that take the student through each tutorial in the book. Also, one programming problem from each chapter has an accompanying VideoNote that helps the student create a solution. You may access the VideoNotes by going to www.pearson.com/cs-resources.

Instructor The following supplements are available to qualified instructors:

- Answers to all Review Questions in the text
- Solutions for all Programming Problems in the text
- Completed versions of all tutorials
- PowerPoint presentation slides for every chapter
- Test bank

For information on how to access these supplements, visit the Pearson Education Instructor Resource Center at www.pearsonhighered.com/irc.

Acknowledgments

There were many helping hands in the development and publication of this text. I would like to thank the following faculty reviewers for their helpful suggestions and expertise:

Matthew Alimagham
Spartanburg Community College

Carolyn Borne
Louisiana State University

Arthur E. Carter
Radford University

Sallie B. Dodson
Radford University

Elizabeth Freije
Indiana University—Purdue University, Indianapolis

Bettye J. Parham
Daytona State College

Wendy L. Payne
Gulf Coast State College

Jason Sharp
Tarleton State University

John Van Assen
York Technical College

Reginald White
Black Hawk College

Dawn R. Wick
Southwestern Community College

I also want to thank everyone at Pearson for making the *Starting Out With* ... series so successful. I have worked so closely with the team at Pearson that I consider them among my closest friends. I am extremely fortunate to have Matt Goldstein as my editor, and Meghan Jacoby as Editorial Assistant. They have guided me through the process of revising this book, as well as many others. I am also fortunate to have Demetrius Hall and Yvonne Vannatta as my marketing team. Their hard work is truly inspiring, and they do a great job of getting this book out to the academic community. The production team, led by Carole Snyder, worked tirelessly to make this book a reality. Thanks to you all!

About the Author

Tony Gaddis is the principal author of the *Starting Out With* series of textbooks. Tony has nearly 20 years experience teaching computer science courses at Haywood Community College in North Carolina. He is a highly acclaimed instructor who was previously selected as the North Carolina Community College Teacher of the Year and has received the Teaching Excellence Award from the National Institute for Staff and Organizational Development.

The *Starting Out With* series includes introductory books using the C++ programming language, the Java™ programming language, Microsoft® Visual Basic®, Microsoft® C#®, Python, Programming Logic and Design, MIT App Inventor, and Alice, all published by Pearson Education.

Attention Students

Installing Visual Studio

To complete the tutorials and programming problems in this book, you need to install Visual Studio 2017, or a later version, on your computer. We recommend that you download Visual Studio Community 2017 from the following website, and install it on your system:

www.visualstudio.com

Visual Studio Community 2017 is a free, full-featured development environment, and is a perfect companion for this textbook.

NOTE: If you are working in your school's computer lab, there is a good chance that **Microsoft Visual Studio** has already been installed. If this is the case, your instructor will show you how to start Visual Studio.

Installing the Student Sample Program Files

The Student Sample Program files that accompany this book are available for download from the book's companion Web site at:

www.pearson.com/cs-resources

These files are required for many of the book's tutorials. Simply download the Student Sample Program files to a location on your hard drive where you can easily access them.

CHAPTER 1

Introduction to Computers and Programming

TOPICS

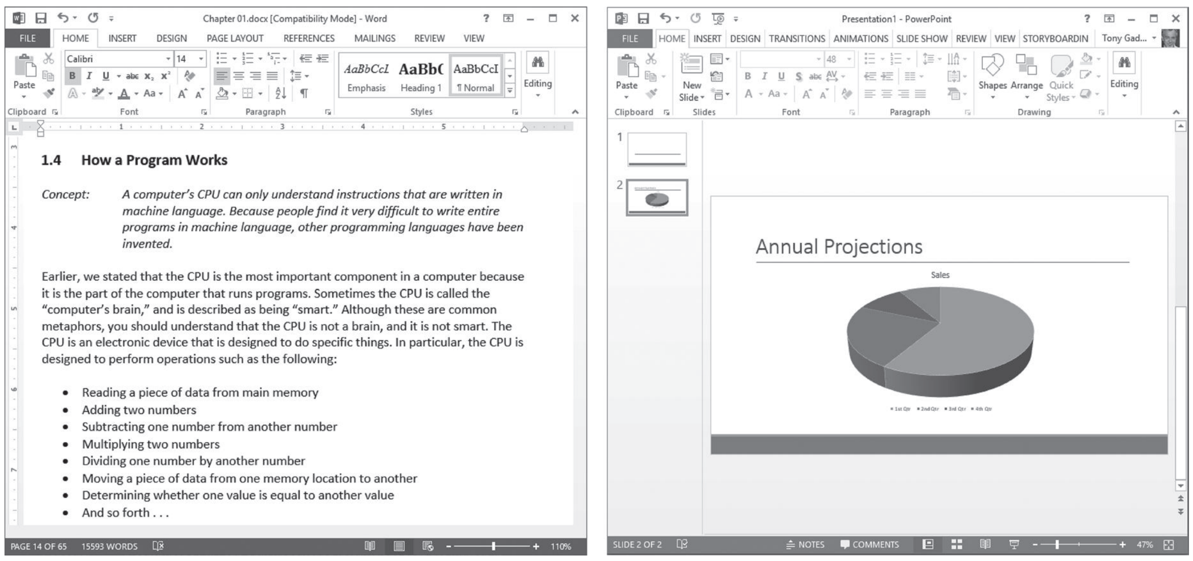
1.1 Introduction

Think about some of the different ways that people use computers. In school, students use computers for tasks such as writing papers, searching for articles, sending e-mail, and participating in online classes. At work, people use computers to analyze data, make presentations, conduct business transactions, communicate with customers and coworkers, control machines in manufacturing facilities, and do many other things. At home, people use computers for tasks such as paying bills, shopping online, staying connected with friends and family, and playing computer games. And don't forget that smart phones, tablets, car navigation systems, and many other devices are computers as well. The uses of computers are almost limitless in our everyday lives.

Computers can do such a wide variety of things because they can be programmed, which means that computers are designed not to do just one job, but to do any job that their programs tell them to do. A **program** is a set of instructions that a computer follows to perform a task. For example, Figure 1-1 shows screens from two commonly used Microsoft programs: Word and PowerPoint. Word is a word processing program that allows you to create, edit, and print documents. PowerPoint allows you to create graphical slides and use them as part of a presentation.

Programs are commonly referred to as **software**. Software is essential to a computer because without software, a computer can do nothing. All the software that makes our computers useful is created by individuals known as programmers, or software developers. A **programmer**, or **software developer**, is a person with the training and skills necessary to design, create, and test computer programs. Computer programming is an exciting and rewarding career. Today, programmers work in business, medicine, government, law enforcement, agriculture, academics, entertainment, and almost every other field.

Figure 1-1 A word processing program and a presentation program


Chapter 01.docx [Compatibility Mode] - Word
FILE HOME INSERT DESIGN PAGE LAYOUT REFERENCES MAILINGS REVIEW VIEW
Calibri 14
Emphasis Heading 1 ¶ Normal
Editing
Clipboard Font Paragraph Styles
PAGE 14 OF 65 15593 WORDS 110%
Presentation1 - PowerPoint
FILE HOME INSERT DESIGN TRANSITIONS ANIMATIONS SLIDE SHOW REVIEW VIEW STORYBOARDIN Tony Gad...
Paste New Slide Shapes Arrange Quick Styles Editing
Clipboard Slides Font Paragraph Drawing
Annual Projections
Sales
1st Qtr 2nd Qtr 3rd Qtr 4th Qtr
SLIDE 2 OF 2 NOTES COMMENTS 47%

1.4 How a Program Works

Concept: *A computer's CPU can only understand instructions that are written in machine language. Because people find it very difficult to write entire programs in machine language, other programming languages have been invented.*

Earlier, we stated that the CPU is the most important component in a computer because it is the part of the computer that runs programs. Sometimes the CPU is called the "computer's brain," and is described as being "smart." Although these are common metaphors, you should understand that the CPU is not a brain, and it is not smart. The CPU is an electronic device that is designed to do specific things. In particular, the CPU is designed to perform operations such as the following:

- Reading a piece of data from main memory
- Adding two numbers
- Subtracting one number from another number
- Multiplying two numbers
- Dividing one number by another number
- Moving a piece of data from one memory location to another
- Determining whether one value is equal to another value
- And so forth . . .

This book introduces you to the fundamental concepts of computer programming using the C# programming language. Before we begin exploring those concepts, you need to understand a few basic things about computers and how they work. This chapter provides a solid foundation of knowledge that you will continually rely on as you study computer science. First, we discuss the physical components that computers are commonly made of. Then, we look at how computers store data and execute programs. Next, we introduce you to two fundamental elements of modern software design: graphical user interfaces and objects. Finally, we give a quick introduction to the software used to write C# programs.

1.2 Hardware and Software

CONCEPT: The physical devices that a computer is made of are referred to as the computer's hardware. The programs that run on a computer are referred to as software.

Hardware

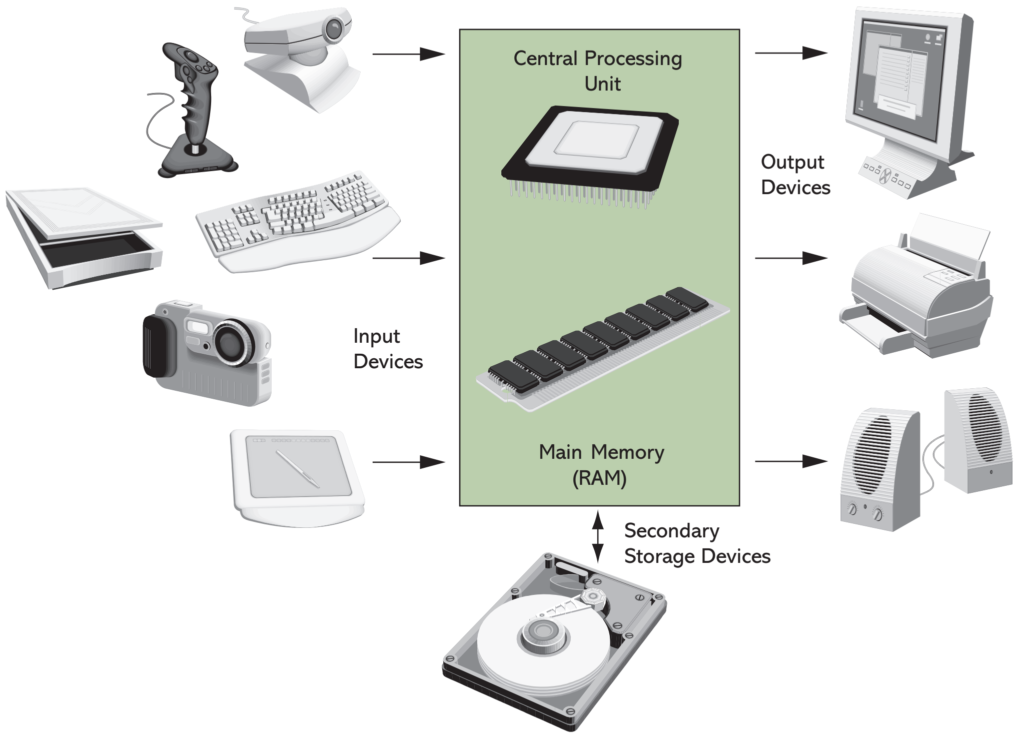
Hardware refers to all the physical devices, or components, of which a computer is made. A computer is not one single device but is a system of devices that all work together. Like the different instruments in a symphony orchestra, each device in a computer plays its own part.

If you have ever shopped for a computer, you have probably seen sales literature listing components such as microprocessors, memory, disk drives, video displays, graphics cards, and so forth. Unless you already know a lot about computers or at least have a friend who does, understanding what these different components do can be confusing. As shown in Figure 1-2, a typical computer system consists of the following major components:

- The central processing unit (CPU)
- Main memory
- Secondary storage devices
- Input devices
- Output devices

Let's take a closer look at each of these components.

Figure 1-2 Typical components of a computer system

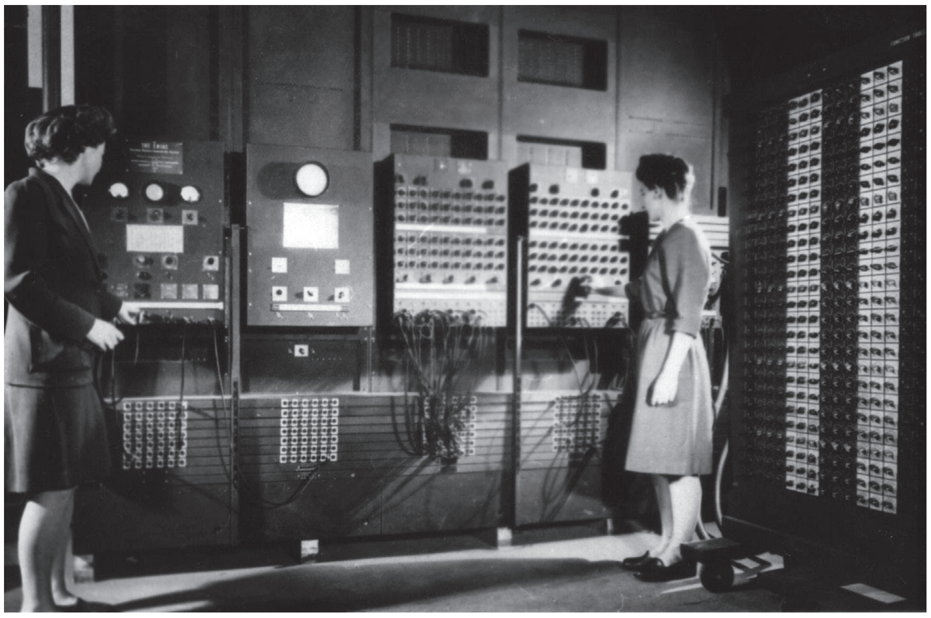


The CPU

When a computer is performing the tasks that a program tells it to do, we say that the computer is **running** or **executing** the program. The **central processing unit**, or **CPU**, is the part of a computer that actually runs programs. The CPU is the most important component in a computer because without it, the computer could not run software.

In the earliest computers, CPUs were huge devices made of electrical and mechanical components such as vacuum tubes and switches. Figure 1-3 shows such a device. The two

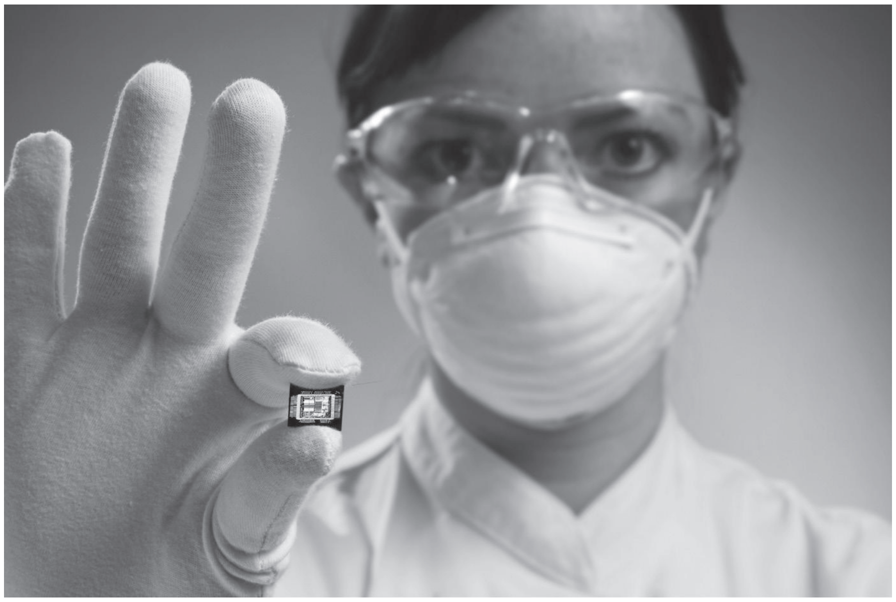
Figure 1-3 The ENIAC computer



women in the photo are working with the historic ENIAC computer. The **ENIAC**, considered by many to be the world's first programmable electronic computer, was built in 1945 to calculate artillery ballistic tables for the U.S. Army. This machine, which was primarily one big CPU, was 8 feet tall and 100 feet long and weighed 30 tons.

Today, CPUs are small chips known as **microprocessors**. Figure 1-4 shows a photo of a lab technician holding a modern-day microprocessor. In addition to being much smaller than the old electromechanical CPUs in early computers, microprocessors are also much more powerful.

Figure 1-4 A lab technician holds a modern microprocessor



Main Memory

You can think of **main memory** as the computer's work area. This is where the computer stores a program while the program is running, as well as the data that the program is working with. For example, suppose you are using a word processing program to write an essay for one of your classes. While you do this, both the word processing program and the essay are stored in main memory.

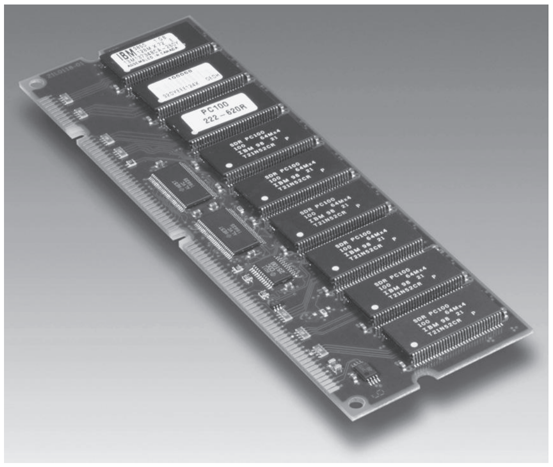
Main memory is commonly known as **random access memory**, or **RAM**. It is called this because the CPU is able to quickly access data stored at any random location in RAM. RAM is usually a **volatile** type of memory that is used only for temporary storage while a program is running. When the computer is turned off, the contents of RAM are erased. Inside your computer, RAM is stored in chips, similar to the ones shown in Figure 1-5.

Secondary Storage Devices

Secondary storage is a type of memory that can hold data for long periods of time, even when there is no power to the computer. Programs are normally stored in secondary memory and loaded into main memory as needed. Important data, such as word processing documents, payroll data, and inventory records, is saved to secondary storage as well.

The most common type of secondary storage device is the **disk drive**. A traditional disk drive stores data by magnetically encoding it onto a spinning circular disk. **Solid-state drives**, which store data in solid-state memory, are increasingly becoming popular. A solid-state drive has no moving parts and operates faster than a traditional disk drive.

Figure 1-5 Memory chips



Most computers have some sort of secondary storage device, either a traditional disk drive or a solid-state drive, mounted inside their case. External storage devices are also available, which connect to one of the computer's communication ports, or plug into a memory slot. External storage devices can be used to create backup copies of important data or to move data to another computer. For example, **USB** (**Universal Serial Bus**) **drives** and **SD** (**Secure Digital**) **memory cards** are small devices that appear in the system as disk drives. They are inexpensive, reliable, and small enough to be carried in your pocket.

Optical devices such as the **compact disc** (**CD**) and the **digital versatile disc** (**DVD**) are also popular for data storage. Data is not recorded magnetically on an optical disc but is encoded as a series of pits on the disc surface. CD and DVD drives use a laser to detect the pits and thus read the encoded data. Optical discs hold large amounts of data, and because recordable CD and DVD drives are now commonplace, they are good mediums for creating backup copies of data.

Input Devices

Input is any data the computer collects from people and from other devices. The component that collects the data and sends it to the computer is called an **input device**. Common input devices are the keyboard, mouse, touchscreen, scanner, microphone, and digital camera. Disk drives and optical drives can also be considered input devices because programs and data are retrieved from them and loaded into the computer's memory.

Output Devices

Output is any data the computer produces for people or for other devices. It might be a sales report, a list of names, or a graphic image. The data is sent to an **output device**, which formats and presents it. Common output devices are screens and printers. Storage devices can also be considered output devices because the system sends data to them in order to be saved.

Software

If a computer is to function, software is not optional. Everything that a computer does, from the time you turn the power switch on until you shut the system down, is under the control of software. There are two general categories of software: system software and application software. Most computer programs clearly fit into one of these two categories. Let's take a closer look at each.

System Software

The programs that control and manage the basic operations of a computer are generally referred to as **system software**. System software typically includes the following types of programs:

Operating Systems

An **operating system** is the most fundamental set of programs on a computer. The operating system controls the internal operations of the computer's hardware, manages all the devices connected to the computer, allows data to be saved to and retrieved from storage devices, and allows other programs to run on the computer.

Utility Programs

A **utility program** performs a specialized task that enhances the computer's operation or safeguards data. Examples of utility programs are virus scanners, file-compression programs, and data-backup programs.

Software Development Tools

The software tools that programmers use to create, modify, and test software are referred to as **software development tools**. Assemblers, compilers, and interpreters, which are discussed later in this chapter, are examples of programs that fall into this category.

Application Software

Programs that make a computer useful for everyday tasks are known as **application software**. These are the programs that people normally spend most of their time running on their computers. Figure 1-1, at the beginning of this chapter, shows screens from two commonly used applications—Microsoft Word, a word processing program, and Microsoft Powerpoint, a presentation program. Some other examples of application software are spreadsheet programs, e-mail programs, Web browsers, and game programs.

Checkpoint

1.1 What is a program?

1.2 What is hardware?

1.3 List the five major components of a computer system.

1.4 What part of the computer actually runs programs?

1.5 What part of the computer serves as a work area to store a program and its data while the program is running?

1.6 What part of the computer holds data for long periods of time, even when there is no power to the computer?

1.7 What part of the computer collects data from people and from other devices?

1.8 What part of the computer formats and presents data for people or other devices?

1.9 What fundamental set of programs control the internal operations of the computer's hardware?

1.10 What do you call a program that performs a specialized task, such as a virus scanner, a file-compression program, or a data-backup program?

1.11 Word processing programs, spreadsheet programs, e-mail programs, Web browsers, and game programs belong to what category of software?

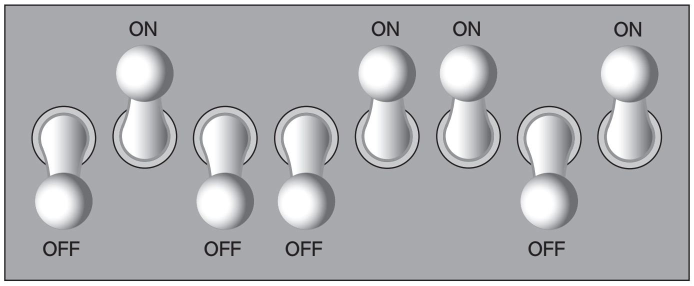
1.3 How Computers Store Data

CONCEPT: All data stored in a computer is converted to sequences of 0s and 1s.

A computer's memory is divided into tiny storage locations known as bytes. One **byte** is enough memory to store only a letter of the alphabet or a small number. In order to do anything meaningful, a computer has to have lots of bytes. Most computers today have millions, or even billions, of bytes of memory.

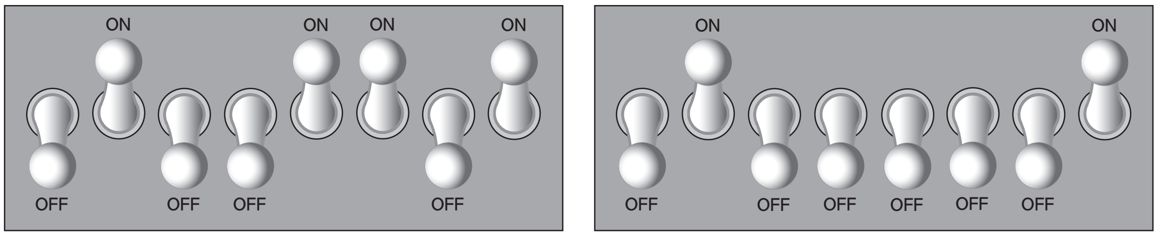
Each byte is divided into eight smaller storage locations known as bits. The term **bit** stands for **binary digit**. Computer scientists usually think of bits as tiny switches that can be either on or off. Bits aren't actual "switches," however, at least not in the conventional sense. In most computer systems, bits are tiny electrical components that can hold either a positive or a negative charge. Computer scientists think of a positive charge as a switch in the *on* position and a negative charge as a switch in the *off* position. Figure 1-6 shows the way that a computer scientist might think of a byte of memory: as a collection of switches that are each flipped to either the on or the off position.

Figure 1-6 A byte thought of as eight switches



When a piece of data is stored in a byte, the computer sets the eight bits to an on/off pattern that represents the data. For example, the pattern shown on the left in Figure 1-7 shows how the number 77 would be stored in a byte, and the pattern on the right shows how the letter A would be stored in a byte. In a moment, you will see how these patterns are determined.

Figure 1-7 Bit patterns for the number 77 and the letter A



The number 77 stored in a byte.

The letter A stored in a byte.

Storing Numbers

A bit can be used in a very limited way to represent numbers. Depending on whether the bit is turned on or off, it can represent one of two different values. In computer systems, a bit that is turned off represents the number 0 and a bit that is turned on represents the number 1. This corresponds perfectly to the **binary numbering system**. In the binary numbering system (or **binary**, as it is usually called), all numeric values are written as sequences of 0s and 1s. Here is an example of a number that is written in binary:

```
10011101
```

The position of each digit in a binary number has a value assigned to it. Starting with the rightmost digit and moving left, the position values are 2^0, 2^1, 2^2, 2^3, and so forth, as shown in Figure 1-8. Figure 1-9 shows the same diagram with the position values calculated. Starting with the rightmost digit and moving left, the position values are 1, 2, 4, 8, and so forth.

Figure 1-8 The values of binary digits as powers of 2

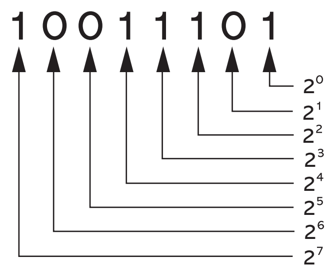
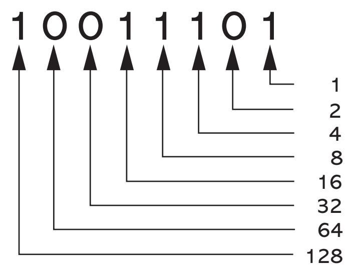


Figure 1-9 The values of binary digits



To determine the value of a binary number, you simply add up the position values of all the 1s. For example, in the binary number 10011101, the position values of the 1s are 1, 4, 8, 16, and 128. This is shown in Figure 1-10. The sum of all these position values is 157. So, the value of the binary number 10011101 is 157.

Figure 1-10 Determining the value of 10011101

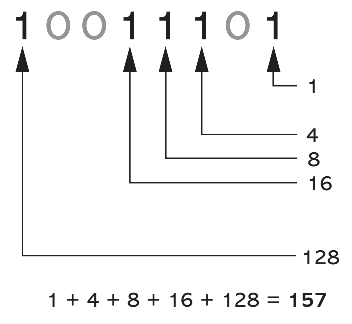
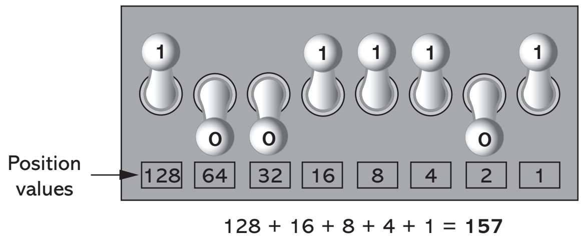


Figure 1-11 shows how you can picture the number 157 stored in a byte of memory. Each 1 is represented by a bit in the on position, and each 0 is represented by a bit in the off position.

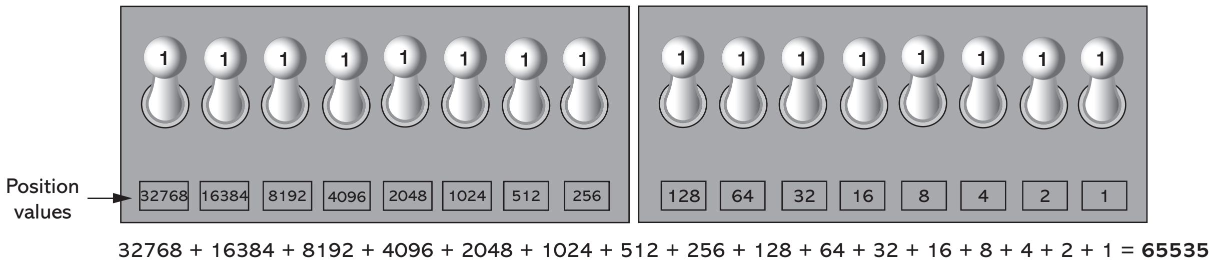
Figure 1-11 The bit pattern for 157



When all the bits in a byte are set to 0 (turned off), then the value of the byte is 0. When all the bits in a byte are set to 1 (turned on), then the byte holds the largest value that can be stored in it. The largest value that can be stored in a byte is 1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 = 255. This limit exists because there are only eight bits in a byte.

What if you need to store a number larger than 255? The answer is simple: use more than 1 byte. For example, suppose we put 2 bytes together. That gives us 16 bits. The position values of those 16 bits would be 2^0, 2^1, 2^2, 2^3, and so forth, up through 2^{15}. As shown in Figure 1-12, the maximum value that can be stored in 2 bytes is 65,535. If you need to store a number larger than this, then more bytes are necessary.

Figure 1-12 Two bytes used for a large number



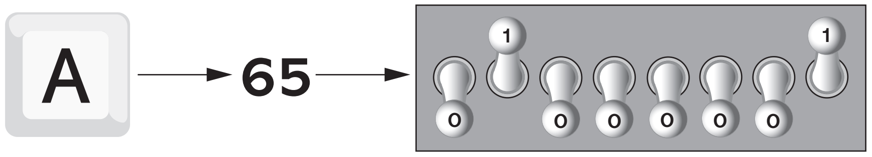
TIP: In case you're feeling overwhelmed by all this, relax! You will not have to actually convert numbers to binary while programming. Knowing that this process is taking place inside the computer will help you as you learn, and in the long term this knowledge will make you a better programmer.

Storing Characters

Any piece of data that is stored in a computer's memory must be stored as a binary number. That includes characters such as letters and punctuation marks. When a character is stored in memory, it is first converted to a numeric code. The numeric code is then stored in memory as a binary number.

Over the years, different coding schemes have been developed to represent characters in computer memory. Historically, the most important of these coding schemes is **ASCII**, which stands for the **American Standard Code for Information Interchange**. ASCII is a set of 128 numeric codes that represent the English letters, various punctuation marks, and other characters. For example, the ASCII code for the uppercase letter A is 65. When you type an uppercase A on your computer keyboard, the number 65 is stored in memory (as a binary number, of course). This is shown in Figure 1-13.

Figure 1-13 The letter A stored in memory as the number 65



TIP: The acronym ASCII is pronounced "askee."

In case you are curious, the ASCII code for uppercase B is 66, for uppercase C is 67, and so forth. Appendix C shows all the ASCII codes and the characters they represent.

The ASCII character set was developed in the early 1960s and was eventually adopted by almost all computer manufacturers. ASCII is limited, however, because it defines codes for only 128 characters. To remedy this, the Unicode character set was developed in the early 1990s. **Unicode** is an extensive encoding scheme that is compatible with ASCII and can also represent the characters of many of the world's languages. Today, Unicode is quickly becoming the standard character set used in the computer industry.

Advanced Number Storage

Earlier you saw how numbers are stored in memory. Perhaps it occurred to you then that the binary numbering system can be used to represent only integer numbers, beginning with 0. Negative numbers and real numbers (such as 3.14159) cannot be represented using the simple binary numbering technique we discussed.

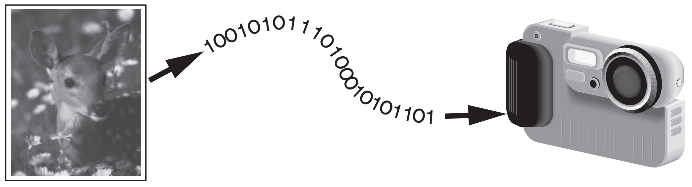
Computers are able to store negative numbers and real numbers in memory, but to do so they use encoding schemes along with the binary numbering system. Negative numbers are encoded using a technique known as **two's complement**, and real numbers are encoded in **floating-point notation**. You don't need to know how these encoding schemes work, only that they are used to convert negative numbers and real numbers to binary format.

Other Types of Data

Computers are often referred to as digital devices. The term **digital** can be used to describe anything that uses binary numbers. **Digital data** is data that is stored in binary, and a **digital device** is any device that works with binary data. In this section, we have discussed how numbers and characters are stored in binary, but computers also work with many other types of digital data.

For example, consider the pictures that you take with your digital camera. These images are composed of tiny dots of color known as **pixels**. (The term pixel stands for **picture element**.) As shown in Figure 1-14, each pixel in an image is converted to a numeric code that represents the pixel's color. The numeric code is stored in memory as a binary number.

Figure 1-14 A digital image stored in binary format



The music that you play on your CD player, iPod, or MP3 player is also digital. A digital song is broken into small pieces known as **samples**. Each sample is converted to a binary number, which can be stored in memory. The more samples that a song is divided into, the more it sounds like the original music when it is played back. A CD-quality song is divided into more than 44,000 samples per second!

Checkpoint

1.12 What amount of memory is enough to store a letter of the alphabet or a small number?

1.13 What do you call a tiny "switch" that can be set to either on or off?

1.14 In what numbering system are all numeric values written as sequences of 0s and 1s?

1.15 What is the purpose of ASCII?

1.16 What encoding scheme is extensive enough to represent all the characters of many of the languages in the world?

1.17 What do the terms *digital data* and *digital device* mean?

1.4 How a Program Works

CONCEPT: A computer's CPU can understand only instructions written in machine language. Because people find it very difficult to write entire programs in machine language, other programming languages have been invented.

Earlier, we stated that the CPU is the most important component in a computer because it is the part of the computer that runs programs. Sometimes the CPU is called the "computer's brain" and is described as being "smart." Although these are common metaphors, you should understand that the CPU is not a brain, and it is not smart. The CPU is an electronic device that is designed to do specific things. In particular, the CPU is designed to perform operations such as the following:

- Reading a piece of data from main memory
- Adding two numbers
- Subtracting one number from another number
- Multiplying two numbers
- Dividing one number by another number
- Moving a piece of data from one memory location to another
- Determining whether one value is equal to another value

As you can see from this list, the CPU performs simple operations on pieces of data. The CPU does nothing on its own, however. It has to be told what to do, which is the purpose of a program. A program is nothing more than a list of instructions that cause the CPU to perform operations.

Each instruction in a program is a command that tells the CPU to perform a specific operation. Here's an example of an instruction that might appear in a program:

```
10110000
```

To you and me, this is only a series of 0s and 1s. To a CPU, however, this is an instruction to perform an operation.[1] It is written in 0s and 1s because CPUs understand only instructions that are written in **machine language**, and machine language instructions are always written in binary.

A machine language instruction exists for each operation that a CPU is capable of performing. For example, there is an instruction for adding numbers; there is an instruction for subtracting one number from another; and so forth. The entire set of instructions that a CPU can execute is known as the CPU's **instruction set**.

NOTE: There are several microprocessor companies today that manufacture CPUs. Some of the more well-known microprocessor companies are Intel, AMD, and Motorola. If you look carefully at your computer, you might find a tag showing a logo for its microprocessor.

Each brand of microprocessor has its own unique instruction set, which is typically understood only by microprocessors of the same brand. For example, Intel microprocessors understand the same instructions, but they do not understand instructions for Motorola microprocessors.

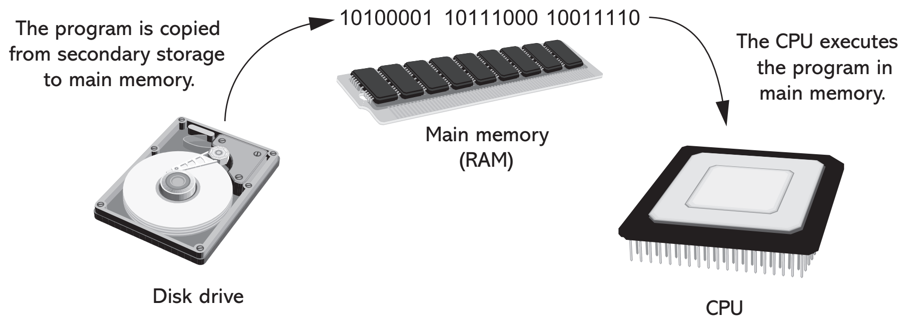
The machine language instruction that was previously shown is an example of only one instruction. It takes a lot more than one instruction, however, for the computer to do anything meaningful. Because the operations that a CPU knows how to perform are so basic in nature, a meaningful task can be accomplished only if the CPU performs many operations. For example, if you want your computer to calculate the amount of interest that you will earn from your savings account this year, the CPU will have to perform a large number of instructions, carried out in the proper sequence. It is not unusual for a program to contain thousands or even a million or more machine language instructions.

Programs are usually stored on a secondary storage device such as a disk drive. When you install a program on your computer, the program is typically copied to your computer's disk drive from a CD-ROM or perhaps downloaded from a Web site.

Although a program can be stored on a secondary storage device such as a disk drive, it has to be copied into main memory, or RAM, each time the CPU executes it. For example, suppose you have a word processing program on your computer's disk. To execute the program, you use the mouse to double-click the program's icon. This causes the program to be copied from the disk into main memory. Then, the computer's CPU executes the copy of the program that is in main memory. This process is illustrated in Figure 1-15.

[1]The example shown is an actual instruction for an Intel microprocessor. It tells the microprocessor to move a value into the CPU.

Figure 1-15 A program being copied into main memory and then executed

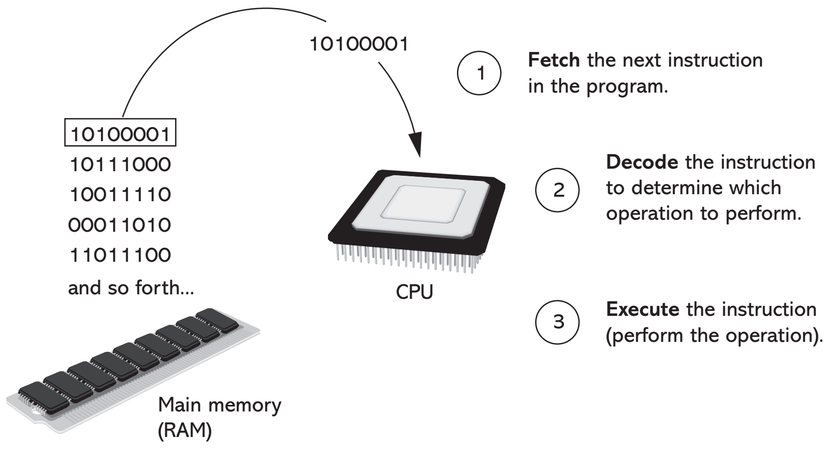

The program is copied from secondary storage to main memory.
10100001 10111000 10011110
The CPU executes the program in main memory.
Main memory (RAM)
Disk drive
CPU

When a CPU executes the instructions in a program, it is engaged in a process that is known as the **fetch-decode-execute cycle**. This cycle, which consists of three steps, is repeated for each instruction in the program. The steps are as follows:

1. **Fetch.** A program is a long sequence of machine language instructions. The first step of the cycle is to fetch, or read, the next instruction from memory into the CPU.
2. **Decode.** A machine language instruction is a binary number that represents a command that tells the CPU to perform an operation. In this step the CPU decodes the instruction that was just fetched from memory, to determine which operation it should perform.
3. **Execute.** The last step in the cycle is to execute, or perform, the operation.

Figure 1-16 illustrates these steps.

Figure 1-16 The fetch-decode-execute cycle


10100001
1 Fetch the next instruction in the program.
10100001
10111000
10011110
00011010
11011100
and so forth...
2 Decode the instruction to determine which operation to perform.
CPU
3 Execute the instruction (perform the operation).
Main memory (RAM)

From Machine Language to Assembly Language

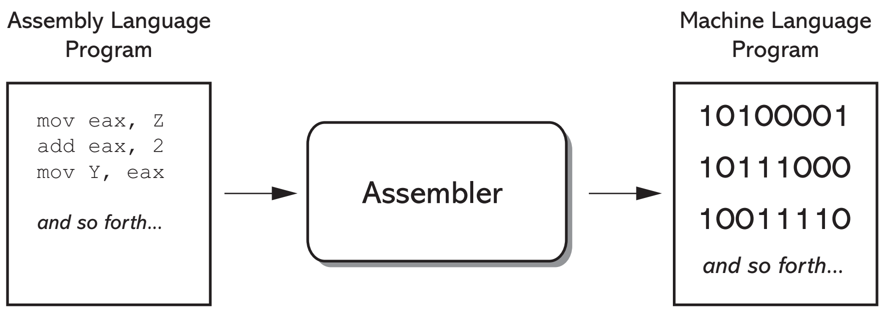
Computers can execute only programs that are written in machine language. As previously mentioned, a program can have thousands or even a million or more binary instructions, and writing such a program would be very tedious and time consuming. Programming in machine language would also be very difficult because putting a 0 or a 1 in the wrong place would cause an error.

Although a computer's CPU understands only machine language, it is impractical for people to write programs in machine language. For this reason, **assembly language** was created in the early days of computing[2] as an alternative to machine language. Instead of using binary numbers for instructions, assembly language uses short words that are known as **mnemonics**. For example, in assembly language, the mnemonic `add` typically means to add numbers, `mul` typically means to multiply numbers, and `mov` typically means to move a value to a location in memory. When a programmer uses assembly language to write a program, he or she can write short mnemonics instead of binary numbers.

> **NOTE:** There are many different versions of assembly language. It was mentioned earlier that each brand of CPU has its own machine language instruction set. Each brand of CPU typically has its own assembly language as well.

Assembly language programs cannot be executed by the CPU, however. The CPU understands only machine language, so a special program known as an **assembler** is used to translate an assembly language program to a machine language program. This process is shown in Figure 1-17. The CPU can then execute the machine language program that the assembler creates.

Figure 1-17 An assembler translating an assembly language program to a machine language program



High-Level Languages

Although assembly language makes it unnecessary to write binary machine language instructions, it is not without difficulties. Assembly language is primarily a direct substitute for machine language, and like machine language, it requires that you know a lot about the CPU. Assembly language also requires that you write a large number of instructions for even the simplest program. Because assembly language is so close in nature to machine language, it is referred to as a **low-level language**.

In the 1950s, a new generation of programming languages known as high-level languages began to appear. A **high-level language** allows you to create powerful and complex programs without knowing how the CPU works and without writing large numbers of low-level instructions. In addition, most high-level languages use words that are easy to understand. For example, if a programmer were using COBOL (which was one of the early high-level languages created in the 1950s), he or she would write the following instruction to display the message *Hello world* on the computer screen:

```
DISPLAY "Hello world"
```

[2]The first assembly language was most likely developed in the 1940s at Cambridge University for use with a historical computer known as the EDSAC.

Doing the same thing in assembly language would require several instructions and an intimate knowledge of how the CPU interacts with the computer's video circuitry. As you can see from this example, high-level languages allow programmers to concentrate on the tasks they want to perform with their programs rather than the details of how the CPU will execute those programs.

Since the 1950s, thousands of high-level languages have been created. Table 1-1 lists several of the more well-known languages.

Table 1-1 Programming languages

Language	Description
Ada	Ada was created in the 1970s, primarily for applications used by the U.S. Department of Defense. The language is named in honor of Countess Ada Lovelace, an influential and historical figure in the field of computing.
BASIC	**B**eginners **A**ll-purpose **S**ymbolic **I**nstruction **C**ode is a general-purpose language that was originally designed in the early 1960s to be simple enough for beginners to learn. Today, there are many different versions of BASIC.
FORTRAN	**FOR**mula **TRAN**slator was the first high-level programming language. It was designed in the 1950s for performing complex mathematical calculations.
COBOL	**Co**mmon **B**usiness-**O**riented **L**anguage was created in the 1950s and was designed for business applications.
Pascal	Pascal was created in 1970 and was originally designed for teaching programming. The language was named in honor of the mathematician, physicist, and philosopher Blaise Pascal.
C and C++	C and C++ (pronounced "c plus plus") are powerful, general-purpose languages developed at Bell Laboratories. The C language was created in 1972, and the C++ language was created in 1983.
C#	Pronounced "c sharp," this language was created by Microsoft around the year 2000 for developing applications based on the Microsoft .NET platform.
Java	Java was created by Sun Microsystems in the early 1990s. It can be used to develop programs that run on a single computer or over the Internet from a Web server.
JavaScript	JavaScript, created in the 1990s, can be used in Web pages. Despite its name, JavaScript is not related to Java.
Python	Python is a general-purpose language created in the early 1990s. It has become popular in business and academic applications.
Ruby	Ruby is a general-purpose language that was created in the 1990s. It is increasingly becoming a popular language for programs that run on Web servers.
Visual Basic	Visual Basic (commonly known as VB) is a Microsoft programming language and software development environment that allows programmers to create Windows-based applications quickly. VB was originally created in the early 1990s.

Keywords, Operators, and Syntax: An Overview

Each high-level language has its own set of predefined words that the programmer must use to write a program. The words that make up a high-level programming language are known as **keywords** or **reserved words**. Each keyword has a specific meaning and cannot be used for any other purpose. Table 1-2 shows the keywords in the C# programming language.

Table 1-2 The C# keywords

abstract	as	base	bool
break	byte	case	catch
char	checked	class	const
continue	decimal	default	delegate
do	double	else	enum
event	explicit	extern	false
finally	fixed	float	for
foreach	goto	if	implicit
in	int	interface	internal
is	lock	long	namespace
new	null	object	operator
out	override	params	private
protected	public	readonly	ref
return	sbyte	sealed	short
sizeof	stackalloc	static	string
struct	switch	this	throw
true	try	typeof	uint
ulong	unchecked	unsafe	ushort
using	virtual	void	volatile
while			

In addition to keywords, programming languages have **operators** that perform various operations on data. For example, all programming languages have math operators that perform arithmetic. In C#, as well as most other languages, the + sign is an operator that adds two numbers. The following adds 12 and 75:

```
12 + 75
```

There are numerous other operators in the C# language, many of which you will learn about as you progress through this text.

In addition to keywords and operators, each language also has its own **syntax**, which is a set of rules that must be strictly followed when writing a program. The syntax rules dictate how keywords, operators, and various punctuation characters must be used in a program. When you are learning a programming language, you must learn the syntax rules for that particular language.

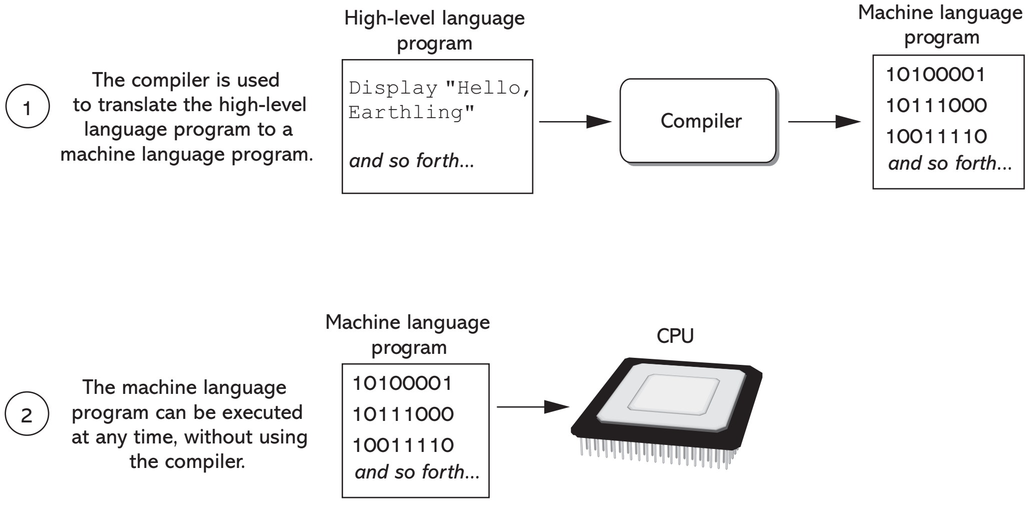
The individual instructions that you use to write a program in a high-level programming language are called **statements**. A programming statement can consist of keywords, operators, punctuation, and other allowable programming elements, arranged in the proper sequence to perform an operation.

Compilers and Interpreters

Because the CPU understands only machine language instructions, programs that are written in a high-level language must be translated into machine language. Depending on the language in which a program has been written, the programmer will use either a compiler or an interpreter to make the translation.

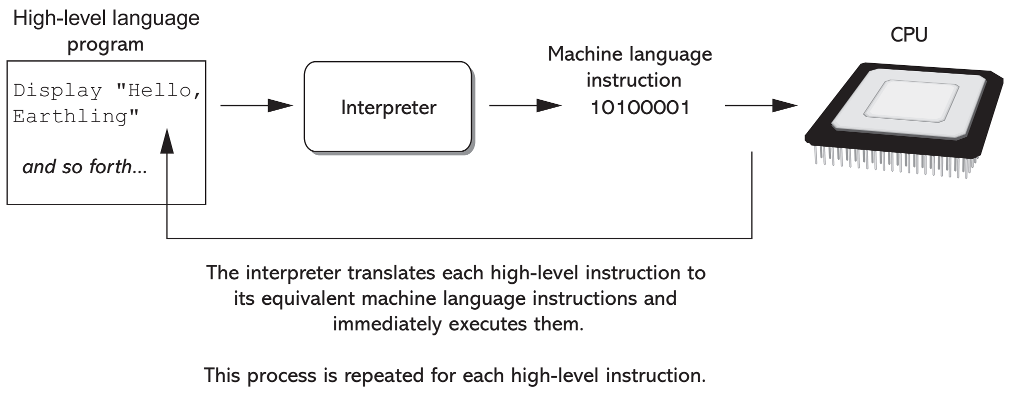
A **compiler** is a program that translates a high-level language program into a separate machine language program. The machine language program can then be executed any time it is needed. This is shown in Figure 1-18. As shown in the figure, compiling and executing are two different processes.

Figure 1-18 Compiling a high-level program and executing it


1
The compiler is used to translate the high-level language program to a machine language program.
High-level language program
Display "Hello, Earthling"
and so forth...
Compiler
Machine language program
10100001
10111000
10011110
and so forth...
2
The machine language program can be executed at any time, without using the compiler.
Machine language program
10100001
10111000
10011110
and so forth...
CPU

Some programming languages use an **interpreter**, which is a program that both translates and executes the instructions in a high-level language program. As the interpreter reads each individual instruction in the program, it converts it to a machine language instruction and then immediately executes it. This process repeats for every instruction in the program. This process is illustrated in Figure 1-19. Because interpreters combine translation and execution, they typically do not create separate machine language programs.

Figure 1-19 Executing a high-level program with an interpreter


High-level language program
Display "Hello, Earthling"
and so forth...
Interpreter
Machine language instruction
10100001
CPU
The interpreter translates each high-level instruction to its equivalent machine language instructions and immediately executes them.
This process is repeated for each high-level instruction.

The statements that a programmer writes in a high-level language are called **source code**, or simply **code**. Typically, the programmer types a program's code into a text editor and then saves the code in a file on the computer's disk. Next, the programmer uses a compiler to translate the code into a machine language program or an interpreter to translate and execute the code. If the code contains a syntax error, however, it cannot be translated. A **syntax error** is a mistake such as a misspelled keyword, a missing punctuation character, or the incorrect use of an operator. When this happens, the compiler or interpreter displays an error message, indicating that the program contains a syntax error. The programmer corrects the error and then attempts once again to translate the program.

NOTE: Human languages also have syntax rules. Do you remember when you took your first English class and you learned all those rules about commas, apostrophes, capitalization, and so forth? You were learning the syntax of the English language.

Although people commonly violate the syntax rules of their native language when speaking and writing, other people usually understand what they mean. Unfortunately, compilers and interpreters do not have this ability. If even a single syntax error appears in a program, the program cannot be compiled or executed.

Checkpoint

1.18 A CPU understands instructions that are written only in what language?

1.19 A program has to be copied into what type of memory each time the CPU executes it?

1.20 When a CPU executes the instructions in a program, it is engaged in what process?

1.21 What is assembly language?

1.22 What type of programming language allows you to create powerful and complex programs without knowing how the CPU works?

1.23 Each language has a set of rules that must be strictly followed when writing a program. What is this set of rules called?

1.24 What do you call a program that translates a high-level language program into a separate machine language program?

1.25 What do you call a program that both translates and executes the instructions in a high-level language program?

1.26 What type of mistake is usually caused by a misspelled keyword, a missing punctuation character, or the incorrect use of an operator?

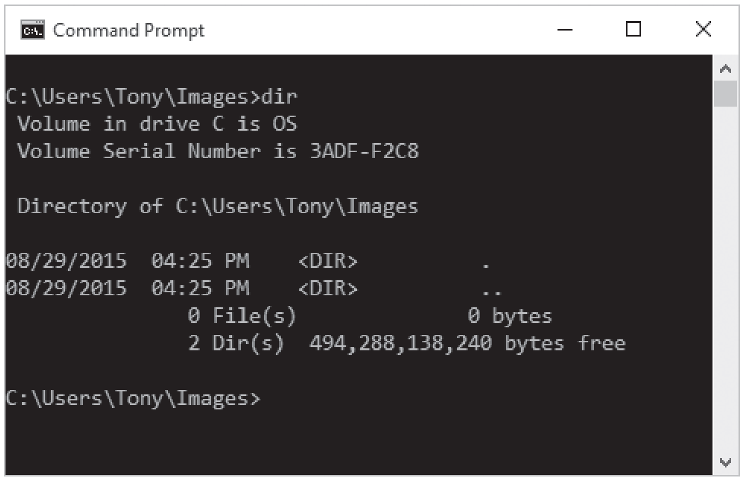
1.5 Graphical User Interfaces

CONCEPT: A graphical user interface allows the user to interact with a program using graphical elements such as icons, buttons, and dialog boxes.

Programmers commonly use the term **user** to describe any hypothetical person that might be using a computer and its programs. A computer's **user interface** is the part of the computer with which the user interacts. One part of the user interface consists of hardware devices, such as the keyboard and the video display. Another part of the user interface involves the way that the computer's operating system and application software accepts commands from the user. For many years, the only way that the user could interact with a computer was through a command line interface. A **command line interface**, which is also known as a **console interface**, requires the user to type commands. If a command is typed correctly, it is executed and the results are displayed. If a command is not typed correctly, an error message is displayed. Figure 1-20 shows the Windows command prompt window, which is an example of a command line interface.

Many computer users, especially beginners, find command line interfaces difficult to use. This is because there are many commands to be learned, and each command has its own syntax, much like a programming statement. If a command isn't entered correctly, it will not work.

Figure 1-20 A command line interface

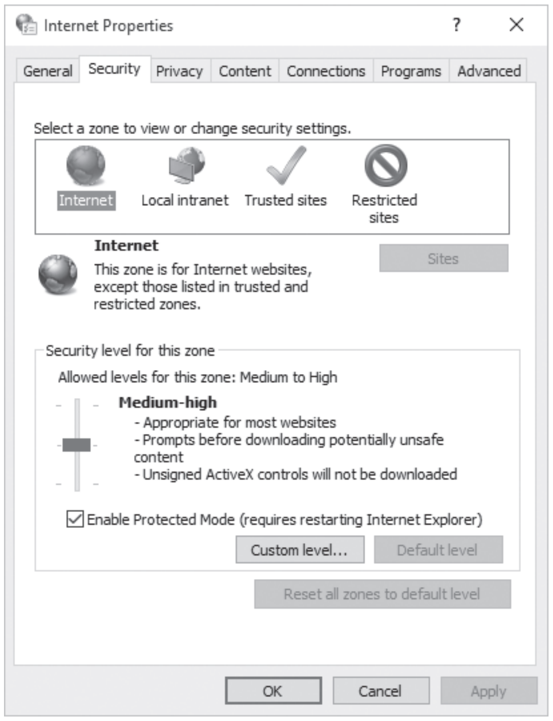


```
C:\Users\Tony\Images>dir
 Volume in drive C is OS
 Volume Serial Number is 3ADF-F2C8

 Directory of C:\Users\Tony\Images

08/29/2015  04:25 PM    <DIR>          .
08/29/2015  04:25 PM    <DIR>          ..
               0 File(s)              0 bytes
               2 Dir(s)  494,288,138,240 bytes free

C:\Users\Tony\Images>
```

In the 1980s, a new type of interface known as a graphical user interface came into use in commercial operating systems. A **graphical user interface**, or **GUI** (pronounced "gooey"), allows the user to interact with the operating system and application programs through graphical elements on the screen. GUIs also popularized the use of the mouse as an input device. Instead of requiring the user to type commands on the keyboard, GUIs allow the user to point at graphical elements and click the mouse button to activate them.

Much of the interaction with a GUI is done through windows that display information and allow the user to perform actions. Figure 1-21 shows an example of a window that allows the user to change the system's Internet settings. Instead of typing cryptic commands, the user interacts with graphical elements such as icons, buttons, and slider bars.

Figure 1-21 A window in a graphical user interface



Select a zone to view or change security settings.

Internet
This zone is for Internet websites, except those listed in trusted and restricted zones.

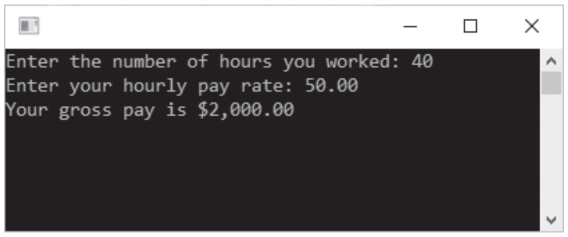
Security level for this zone
Allowed levels for this zone: Medium to High
Medium-high
- Appropriate for most websites
- Prompts before downloading potentially unsafe content
- Unsigned ActiveX controls will not be downloaded

☑ Enable Protected Mode (requires restarting Internet Explorer)

Event-Driven GUI Programs

In a text-based environment, such as a command line interface, programs determine the order in which things happen. For example, Figure 1-22 shows the interaction that has taken place in a text environment with a program that calculates an employee's gross pay. First, the program told the user to enter the number of hours worked. In the figure, the user entered 40 and pressed the Enter key. Next, the program told the user to enter his or her hourly pay rate. In the figure, the user entered 50.00, and pressed the Enter key. Then, the program displayed the user's gross pay. As the program was running, the user had no choice but to enter the data in the order requested.

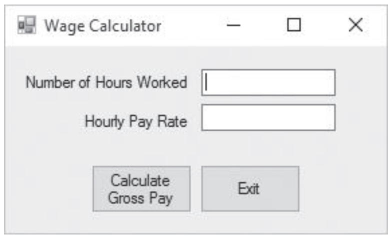
Figure 1-22 Interaction with a program in a text environment



```
Enter the number of hours you worked: 40
Enter your hourly pay rate: 50.00
Your gross pay is $2,000.00
```

In a GUI environment, however, the user determines the order in which things happen. For example, Figure 1-23 shows a GUI program that calculates an employee's gross pay. Notice that there are boxes in which the user enters the number of hours worked and the hourly pay rate. The user can enter the hours and the pay rate in any order he or she wishes. If the user makes a mistake, the user can erase the data that was entered and retype it. When the user is ready to calculate the area, he or she uses the mouse to click the *Calculate Gross Pay* button and the program performs the calculation.

Figure 1-23 A GUI program



Because GUI programs must respond to the actions of the user, they are said to be **event driven**. The user causes events, such as the clicking of a button, and the program responds to those events.

This book focuses exclusively on the development of GUI applications using the C# programming language. As you work through this book, you will learn to create applications that interact with the user through windows containing graphical objects. You will also learn how to program your applications to respond to the events that take place as the user interacts with them.

Checkpoint

1.27 What is a user interface?

1.28 How does a command line interface work?

1.29 When the user runs a program in a text-based environment, such as the command line, what determines the order in which things happen?

1.30 What is an event-driven program?

1.6 Objects

CONCEPT: An object is a program component that contains data and performs operations. Programs use objects to perform specific tasks.

Have you ever driven a car? If so, you know that a car is made of a lot of components. A car has a steering wheel, an accelerator pedal, a brake pedal, a gear shifter, a speedometer, and numerous other devices with which the driver interacts. There are also a lot of components under the hood, such as the engine, the battery, the radiator, and so forth. A car is not just one single object, but rather a collection of objects that work together.

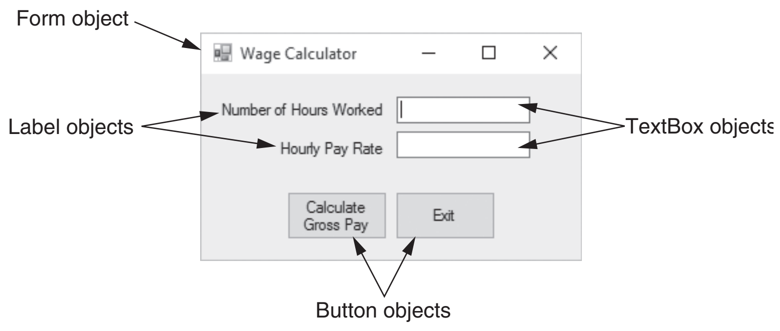
This same notion also applies to computer programming. Most programming languages that are used today are **object oriented**. When you use an object-oriented language, you create programs by putting together a collection of objects. In programming, an object is not a physical device, however, like a steering wheel or a brake pedal. Instead, it is a software component that exists in the computer's memory. In software, an object has two general capabilities:

- An object can store data. The data stored in an object are commonly called **fields**, or **properties**.
- An object can perform operations. The operations that an object can perform are called **methods**.

When you write a program using an object-oriented language, you use objects to accomplish specific tasks. Some objects have a visual part that can be seen on the screen. For example, Figure 1-24 shows the wage-calculator program that we discussed in the previous section. The graphical user interface is made of the following objects:

Form object	A window that is displayed on the screen is called a **Form object**. Figure 1-24 shows a Form object that contains several other graphical objects.
Label objects	A **Label** object displays text on a form. The form shown in Figure 1-24 contains two Label objects. One of the Label objects displays the text *Number of Hours Worked* and the other Label object displays the text *Hourly Pay Rate*.

Figure 1-24 Objects used in a GUI



TextBox objects	A **TextBox** object appears as a rectangular region that can accept keyboard input from the user. The form shown in Figure 1-24 has two TextBox objects: one in which the user enters the number of hours worked and another in which the user enters the hourly pay rate.
Button objects	A **Button** object appears on a form as a button with a caption written across its face. When the user clicks a Button object with the mouse, an action takes place. The form in Figure 1-24 has two Button objects. One shows the caption *Calculate Gross Pay*. When the user clicks this button, the program calculates and displays the gross pay. The other button shows the caption *Exit*. When the user clicks this button, the program ends.

Forms, Labels, TextBoxes, and Buttons are just a few of the objects that you will learn to use in C#. As you study this book, you will create applications that incorporate many different types of objects.

Visible versus Invisible Objects

Objects that are visible in a program's GUI are commonly referred to as **controls**. We could say that the form shown in Figure 1-24 contains two Label controls, two TextBox controls, and two Button controls. When an object is referred to as a control, it simply means that the object plays a role in a program's GUI.

Not all objects can be seen on the screen, however. Some objects exist only in memory for the purpose of helping your program perform some task. For example, there are objects that read data from files, objects that generate random numbers, objects that store and sort large collections of data, and so forth. These types of objects help your program perform tasks, but they do not directly display anything on the screen. When you are writing a program, you will use objects that can help your program perform its tasks. Some of the objects that you use will be controls (visible in the program's GUI), and other objects will be invisible.

Classes: Where Objects Come From

Objects are very useful, but they don't just magically appear in your program. Before a specific type of object can be used, that object has to be created in memory. And, before an object can be created in memory, you must have a class for the object.

A **class** is code that describes a particular type of object. It specifies the data that an object can hold (the object's fields and properties), and the actions that an object can perform (the object's methods). You will learn much more about classes as you progress through this book, but for now, just think of a class as a code "blueprint" that can be used to create a particular type of object.

The .NET Framework

C# is a very popular programming language, but there are a lot of things it cannot do by itself. For example, you cannot use C# alone to create a GUI, read data from files, work with databases, or many of the other things that programs commonly need to do. C# provides only the basic keywords and operators that you need to construct a program.

So, if the C# language doesn't provide the classes and other code necessary for creating GUIs and performing many other advanced operations, where do those classes and code come from? The answer is the .NET Framework. The **.NET Framework** is a collection of classes and other code that can be used, along with a programming language such as C#, to create programs for the Windows operating system. For example, the .NET Framework provides classes to create Forms, TextBoxes, Labels, Buttons, and many other types of objects.

When you use Visual C# to write programs, you are using a combination of the C# language and the .NET Framework. As you work through this book you will not only learn C#, but you will also learn about many of the classes and other features provided by the .NET Framework.

Writing Your Own Classes

The .NET Framework provides many prewritten classes ready for use in your programs. There will be times, however, that you will wish you had an object to perform a specific task, and no such class will exist in the .NET Framework. This is not a problem because in C# you can write your own classes that have the specific fields, properties, and methods that you need for any situation. In Chapter 10, you will learn to create classes for the specific objects that you need in your programs.

Checkpoint

1.31 What is an object?

1.32 What type of language is used to create programs by putting together a collection of objects?

1.33 What two general capabilities does an object have?

1.34 What term is commonly used to refer to objects such as TextBoxes, Labels, and Buttons that are visible in a program's graphical user interface?

1.35 What is the purpose of an object that cannot be seen on the screen and exists only in memory?

1.36 What is a class?

1.37 What is the .NET Framework?

1.38 Why might you need to write your own classes?

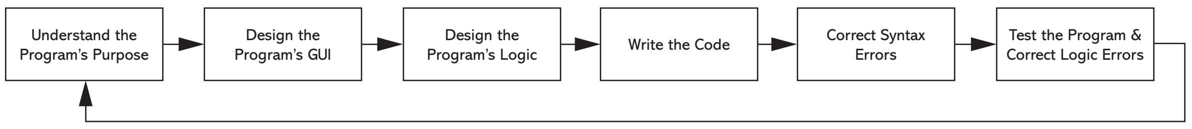
1.7 The Program Development Process

CONCEPT: Creating a program requires several steps, which include designing the program's logic, creating the user interface, writing code, testing, and debugging.

The Program Development Cycle

Previously in this chapter, you learned that programmers typically use high-level languages such as C# to create programs. There is much more to creating a program than writing code, however. The process of creating a program that works correctly typically requires the six phases shown in Figure 1-25. The entire process is known as the **program development cycle**.

Figure 1-25 The program development cycle


Understand the Program's Purpose
Design the Program's GUI
Design the Program's Logic
Write the Code
Correct Syntax Errors
Test the Program & Correct Logic Errors

Let's take a closer look at each stage in the cycle.

1 **Understand the Program's Purpose**

When beginning a new programming project, it is essential that you understand what the program is supposed to do. Most programs perform the following three-step process:

Step 1. Input is received.
Step 2. Some process is performed on the input.
Step 3. Output is produced.

Input is any data that the program receives while it is running. Once input is received, some process, such as a mathematical calculation, is usually performed on it. The results of the process are then sent out of the program as output. If you can identify these three elements of a program (input, process, and output), then you are on your way to understanding what the program is supposed to do.

For example, suppose you have been asked to write a program to calculate and display the gross pay for an hourly paid employee. Here is a summary of the program's input, process, and output:

Input:

- Input the number of hours that the employee worked.
- Input the employee's hourly pay rate.

Process:

- Multiply the number of hours worked by the hourly pay rate. The result is the employee's gross pay.

Output:

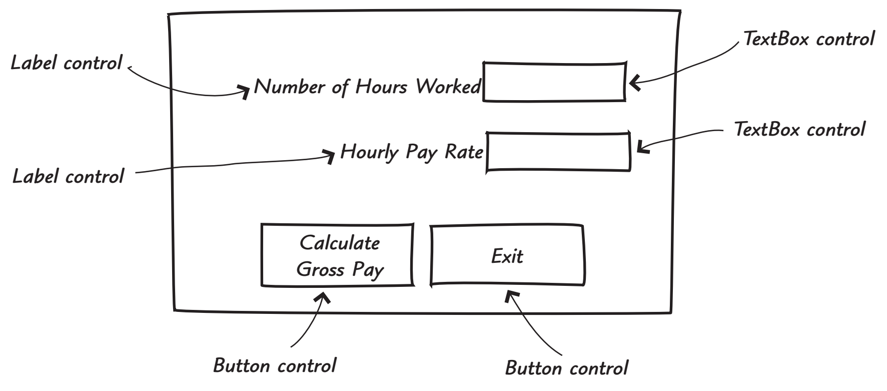
- Display the employee's gross pay on the screen.

2. **Design the GUI**

Once you clearly understand what the program is supposed to do, you can begin designing its GUI. Often, you will find it helpful to draw a sketch of each form that the program displays. For example, if you are designing a program that calculates gross pay, Figure 1-26 shows how you might sketch the program's form.

Notice that the sketch identifies each type of control (GUI object) that will appear on the form. The TextBox controls will allow the user to enter input. The user will type the number of hours worked into one of the TextBoxes and the employee's hourly pay rate into the other TextBox. Notice that Label controls are placed on the form to tell the user what data to enter. When the user clicks the Button control that reads *Calculate*

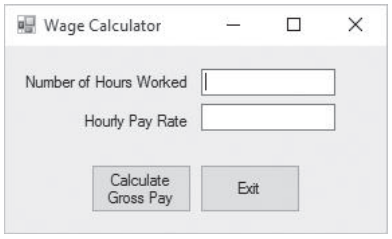
Figure 1-26 Form sketch



Gross Pay, the program will display the employee's gross pay on the screen in a pop-up window. When the user clicks the Button control that reads *Exit*, the program will end.

Once you are satisfied with the sketches that you have created for the program's forms, you can begin creating the actual forms on the computer. As a Visual C# programmer, you have a powerful environment known as Visual Studio at your disposal. Visual Studio gives you a "what you see is what you get" editor that allows you to visually design a program's forms. You can use Visual Studio to create the program's forms, place all the necessary controls on the forms, and set each control's properties so it has the desired appearance. For example, Figure 1-27 shows the actual form that you might create for the wage-calculator program, which calculates gross pay.

Figure 1-27 Form for the wage-calculator program



3. **Design the Program's Logic**

   In this phase you break down each task that the program must perform into a series of logical steps. For example, if you look back at Figure 1-27, notice that the pay-calculating program's form has a Button control that reads *Calculate Gross Pay*. When the user clicks this button, you want the program to display the employee's gross pay. Here are the steps that the program should take to perform that task:

   Step 1. Get the number of hours worked from the appropriate TextBox.
   Step 2. Get the hourly pay rate from the appropriate TextBox.
   Step 3. Calculate the gross pay as the number of hours worked times the hourly pay rate.
   Step 4. Display the gross pay in a pop-up window.

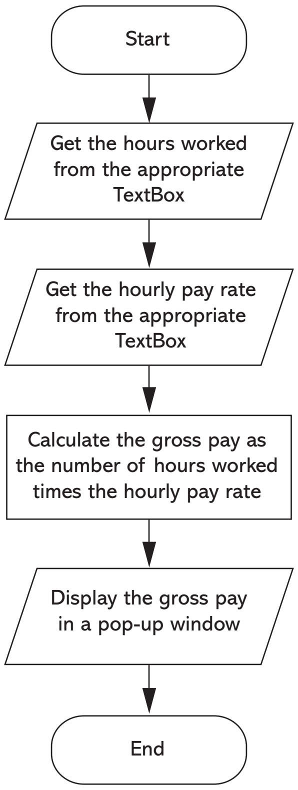
   This is an example of an **algorithm**, which is a set of well-defined, logical steps that must be taken to perform a task. An algorithm that is written out in this manner, in plain English statements, is called **pseudocode**. (The word *pseudo* means fake, so pseudocode is fake code.) The process of informally writing out the steps of an algorithm in pseudocode before attempting to write any actual code is very helpful when you are designing a program. Because you do not have to worry about breaking any syntax rules, you can focus on the logical steps that the program must perform.

   Flowcharting is another tool that programmers use to design programs. A **flowchart** is a diagram that graphically depicts the steps of an algorithm. Figure 1-28 shows how you might create a flowchart for the wage-calculator algorithm. Notice that there are three types of symbols in the flowchart: ovals, parallelograms, and a rectangle. Each of these symbols represents a step in the algorithm, as described here:

   - The ovals, which appear at the top and bottom of the flowchart, are called **terminal symbols**. The **Start terminal** symbol marks the program's starting point and the **End terminal** symbol marks the program's ending point.

- Parallelograms are used as **input symbols** and **output symbols**. They represent steps in which the program reads input or displays output.
- Rectangles are used as **processing symbols**. They represent steps in which the program performs some process on data, such as a mathematical calculation.

Figure 1-28 Flowchart for the wage-calculator program


Start
Get the hours worked from the appropriate TextBox
Get the hourly pay rate from the appropriate TextBox
Calculate the gross pay as the number of hours worked times the hourly pay rate
Display the gross pay in a pop-up window
End

The symbols are connected by arrows that represent the "flow" of the program. To step through the symbols in the proper order, you begin at the *Start* terminal and follow the arrows until you reach the *End* terminal.

4. **Write the Code**

   Once you have created a program's GUI and designed algorithms for the program's tasks, you are ready to start writing code. During this process, you will refer to the pseudocode or flowcharts that you created in Step 3 and use Visual Studio to write C# code.

5. **Correct Syntax Errors**

   You previously learned in this chapter that a programming language such as C# has rules, known as syntax, that must be followed when writing a program. A language's syntax rules dictate things such as how keywords, operators, and punctuation characters can be used. A syntax error occurs if the programmer violates any of these rules. If the program contains a syntax error or even a simple mistake such as a misspelled keyword, the program cannot be compiled or executed.

   Virtually all code contains syntax errors when it is first written, so the programmer will typically spend some time correcting these. Once all the syntax errors and simple typing mistakes have been corrected, the program can be compiled and translated into an executable program.

6 **Test the Program and Correct Logic Errors**

Once the code is in an executable form, you must then test it to determine whether any logic errors exist. A **logic error** is a mistake that does not prevent the program from running but causes it to produce incorrect results. (Mathematical mistakes are common causes of logic errors.) If the program produces incorrect results, the programmer must **debug** the code. This means that the programmer finds and corrects logic errors in the program. Sometimes, during this process, the programmer discovers that the program's original design must be changed. In this event, the program development cycle starts over and continues until no errors can be found.

Checkpoint

1.39 List the six steps in the program development cycle.

1.40 What is an algorithm?

1.41 What is pseudocode?

1.42 What is a flowchart?

1.43 What do each of the following symbols mean in a flowchart?
- Oval
- Parallelogram
- Rectangle

1.8 Getting Started with the Visual Studio Environment

CONCEPT: Visual Studio provides a collection of tools that you use to build Visual C# applications. The first step in using Visual C# is learning about these tools.

To follow the tutorials in this book, and create Visual C# applications, you will need to install Visual Studio on your computer. **Visual Studio** is a professional **integrated development environment** (IDE), which means that it provides all the necessary tools for creating, testing, and debugging software. It can be used to create applications not only with Visual C#, but also with other languages such as Visual Basic and Visual C++. If you are using a school's computer lab, there's a good chance that Visual Studio has been installed.

If you do not have access to Visual Studio, you can install **Visual Studio Community Edition,** a free programming environment that is available for download from Microsoft at www.visualstudio.com.

Visual Studio is a customizable environment. If you are working in your school's computer lab, there's a chance that someone else has customized the programming environment to suit his or her own preferences. If this is the case, the screens that you see may not match exactly the ones shown in this book. For that reason it's a good idea to reset the programming environment before you create a Visual C# application. Tutorial 1-1 guides you through the process.

Tutorial 1-1:
Starting Visual Studio and Setting Up the Environment

VideoNote

Tutorial 1-1: Starting Visual Studio and Setting Up the Environment

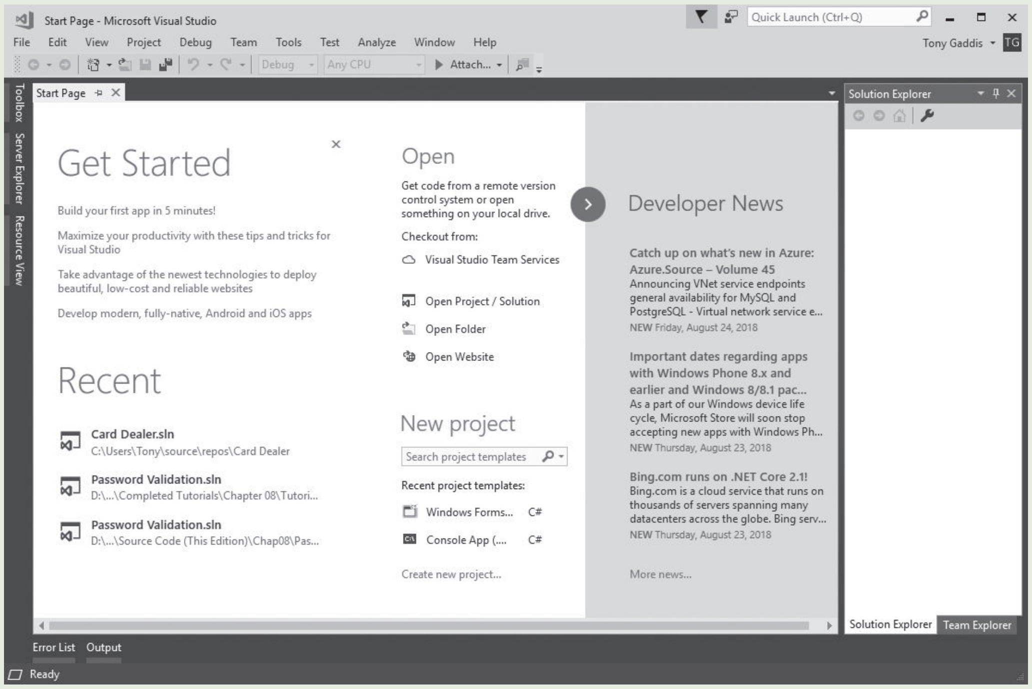
Step 1: Depending on your operating system, use one of the following procedures to start Visual Studio:

- Windows 10: In the Windows search bar, start typing Visual Studio. When you see *Visual Studio 2017* (or another version) appear in the search results, click it.
- Windows 8: On the Start screen, simply start typing Visual Studio. As you type, the search results will appear on the right edge of the screen. When you see *Visual Studio 2017* (or another version) appear, click it.

Step 2: Figure 1-29 shows the Visual Studio environment. The screen shown in the figure is known as the *Start Page*. By default, the *Start Page* is displayed when you start Visual Studio, but you may not see it because it can be disabled.

Notice the check box in the bottom left corner of the *Start Page* that reads *Show page on startup*. If this box is not checked, the *Start Page* will not be displayed when you start Visual Studio. If you do not see the *Start Page*, you can always display it by clicking *View* on the menu bar at the top of the screen and then clicking *Start Page*.

Figure 1-29 Visual Studio *Start Page*


Start Page - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Project Debug Team Tools Test Analyze Window Help
Tony Gaddis TG
Debug Any CPU Attach...
Toolbox Server Explorer Resource View
Start Page
Solution Explorer
Get Started
Recent
Open
New project
Developer News
Solution Explorer Team Explorer
Error List Output
Ready

Build your first app in 5 minutes!

Maximize your productivity with these tips and tricks for Visual Studio

Take advantage of the newest technologies to deploy beautiful, low-cost and reliable websites

Develop modern, fully-native, Android and iOS apps

Card Dealer.sln
C:\Users\Tony\source\repos\Card Dealer

Password Validation.sln
D:\...\Completed Tutorials\Chapter 08\Tutori...

Password Validation.sln
D:\...\Source Code (This Edition)\Chap08\Pas...

Get code from a remote version control system or open something on your local drive.

Checkout from:

Visual Studio Team Services

Open Project / Solution

Open Folder

Open Website

Search project templates

Recent project templates:

Windows Forms... C#

Console App (.... C#

Create new project...

Catch up on what's new in Azure: Azure.Source – Volume 45
Announcing VNet service endpoints general availability for MySQL and PostgreSQL - Virtual network service e...
NEW Friday, August 24, 2018

Important dates regarding apps with Windows Phone 8.x and earlier and Windows 8/8.1 pac...
As a part of our Windows device life cycle, Microsoft Store will soon stop accepting new apps with Windows Ph...
NEW Thursday, August 23, 2018

Bing.com runs on .NET Core 2.1!
Bing.com is a cloud service that runs on thousands of servers spanning many datacenters across the globe. Bing serv...
NEW Thursday, August 23, 2018

More news...

Step 3: In a school computer lab, it is possible that the Visual Studio environment has been set up for a programming language other than Visual C#. To make sure that Visual Studio looks and behaves as described in this book, you should make sure that Visual C# is selected as the programming environment. Perform the following:

- As shown in Figure 1-30, click *Tools* on the menu bar and then click *Import and Export Settings. . . .*
- On the screen that appears next, select *Reset all settings* and click the *Next* > button.
- On the screen that appears next, select *No, just reset settings, overwriting my current settings*. Click the *Next* > button.
- The window shown in Figure 1-31 should appear next. Select *Visual C#* and then click the *Finish* button. After a moment you should see a *Reset Complete* window. Click the *Close* button and continue with the next step in the tutorial.

Figure 1-30 Selecting *Tools* and then *Import and Export Settings . . .*

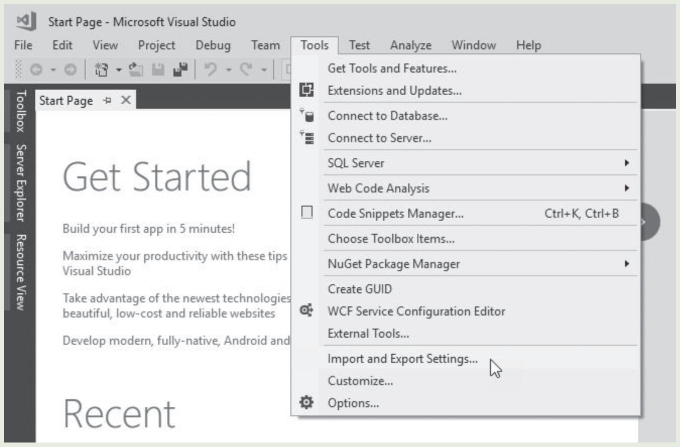
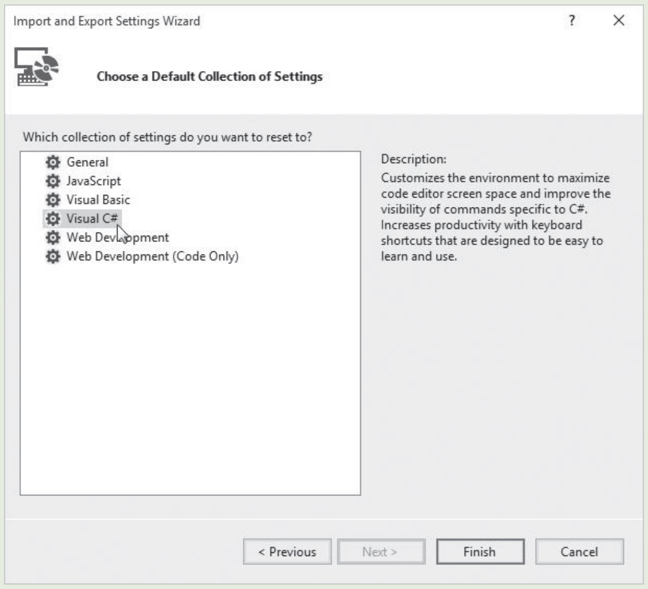

Start Page - Microsoft Visual Studio
File
Edit
View
Project
Debug
Team
Tools
Test
Analyze
Window
Help
Toolbox
Server Explorer
Resource View
Start Page
Get Started
Build your first app in 5 minutes!
Maximize your productivity with these tips Visual Studio
Take advantage of the newest technologies beautiful, low-cost and reliable websites
Develop modern, fully-native, Android and
Recent
Get Tools and Features...
Extensions and Updates...
Connect to Database...
Connect to Server...
SQL Server
Web Code Analysis
Code Snippets Manager... Ctrl+K, Ctrl+B
Choose Toolbox Items...
NuGet Package Manager
Create GUID
WCF Service Configuration Editor
External Tools...
Import and Export Settings...
Customize...
Options...

Figure 1-31 Selecting *Visual C#* Development Settings


Import and Export Settings Wizard

Choose a Default Collection of Settings

Which collection of settings do you want to reset to?

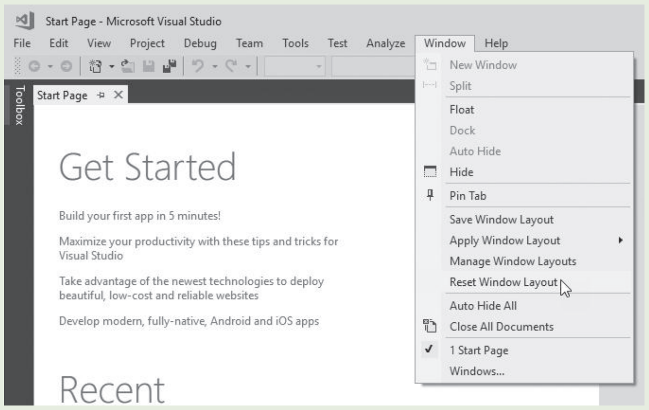
- General
- JavaScript
- Visual Basic
- Visual C#
- Web Development
- Web Development (Code Only)

Description:
Customizes the environment to maximize code editor screen space and improve the visibility of commands specific to C#. Increases productivity with keyboard shortcuts that are designed to be easy to learn and use.

Step 4: Now you will reset Visual Studio's window layout to the default configuration. As shown in Figure 1-32, click *Window* on the menu bar and then click *Reset Window Layout*. Next you will see a dialog box asking *Are you sure you want to restore the default window layout for the environment?* Click *Yes*.

The Visual Studio environment is now set up so you can follow the remaining tutorials in this book. If you are working in your school's computer lab, it is probably a good idea to go through these steps each time you start Visual Studio. If you are continuing with the next tutorial, leave Visual Studio running. You can exit Visual Studio at any time by clicking *File* on the menu bar and then clicking *Exit*.

Figure 1-32 Resetting the window layout



Build your first app in 5 minutes!

Maximize your productivity with these tips and tricks for Visual Studio

Take advantage of the newest technologies to deploy beautiful, low-cost and reliable websites

Develop modern, fully-native, Android and iOS apps

Starting a New Project

Each Visual C# application that you create is called a **project**. When you are ready to create a new application, you start a new project. Tutorial 1-2 leads you through the steps of starting a new Visual C# project.

Tutorial 1-2: Starting a New Visual C# Project

VideoNote
Tutorial 1-2: Starting a New Visual C# Project

Step 1: If Visual Studio is not already running, start it as you did in Tutorial 1-1.

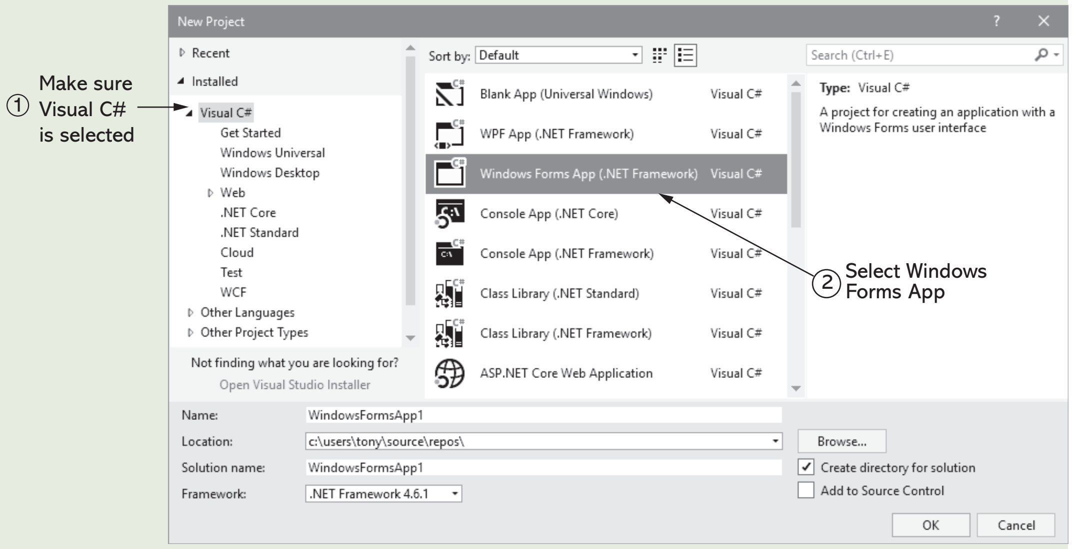
Step 2: Click *File* on the menu bar at the top of the screen, then select *New*, and then select *Project*. After doing this, the *New Project* window shown in Figure 1-33 should be displayed.

Step 3: At the left side of the window, under *Installed Templates*, make sure *Visual C#* is selected. Then, select *Windows Forms App* (*.NET Framework*), as shown in Figure 1-33.

Step 4: At the bottom of the *New Project* window, you see a *Name* text box. This is where you enter the name of your project. The *Name* text box will be automatically filled in with a default name. In Figure 1-33 the default name is *WindowsFormsApp1*. Change the project name to *My First Project*, as shown in Figure 1-34.

Just below the *Name* text box you will see a *Location* text box and a *Solution name* text box.

Figure 1-33 The *New Project* window



New Project

- Recent
- Installed
  - Visual C#
    - Get Started
    - Windows Universal
    - Windows Desktop
    - Web
    - .NET Core
    - .NET Standard
    - Cloud
    - Test
    - WCF
  - Other Languages
  - Other Project Types

Not finding what you are looking for?
Open Visual Studio Installer

Sort by: Default

Search (Ctrl+E)

Blank App (Universal Windows)	Visual C#
WPF App (.NET Framework)	Visual C#
Windows Forms App (.NET Framework)	Visual C#
Console App (.NET Core)	Visual C#
Console App (.NET Framework)	Visual C#
Class Library (.NET Standard)	Visual C#
Class Library (.NET Framework)	Visual C#
ASP.NET Core Web Application	Visual C#

Type: Visual C#

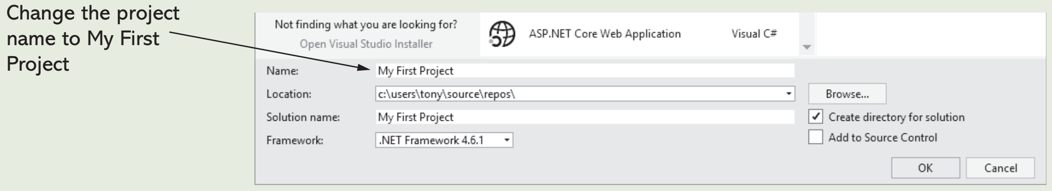
A project for creating an application with a Windows Forms user interface

Name: WindowsFormsApp1
Location: c:\users\tony\source\repos\
Solution name: WindowsFormsApp1
Framework: .NET Framework 4.6.1

Browse...
✓ Create directory for solution
Add to Source Control

OK Cancel

Figure 1-34 Changing the project name to *My First Project*



Not finding what you are looking for?
Open Visual Studio Installer

ASP.NET Core Web Application Visual C#

Name: My First Project
Location: c:\users\tony\source\repos\
Solution name: My First Project
Framework: .NET Framework 4.6.1

Browse...
✓ Create directory for solution
Add to Source Control

OK Cancel

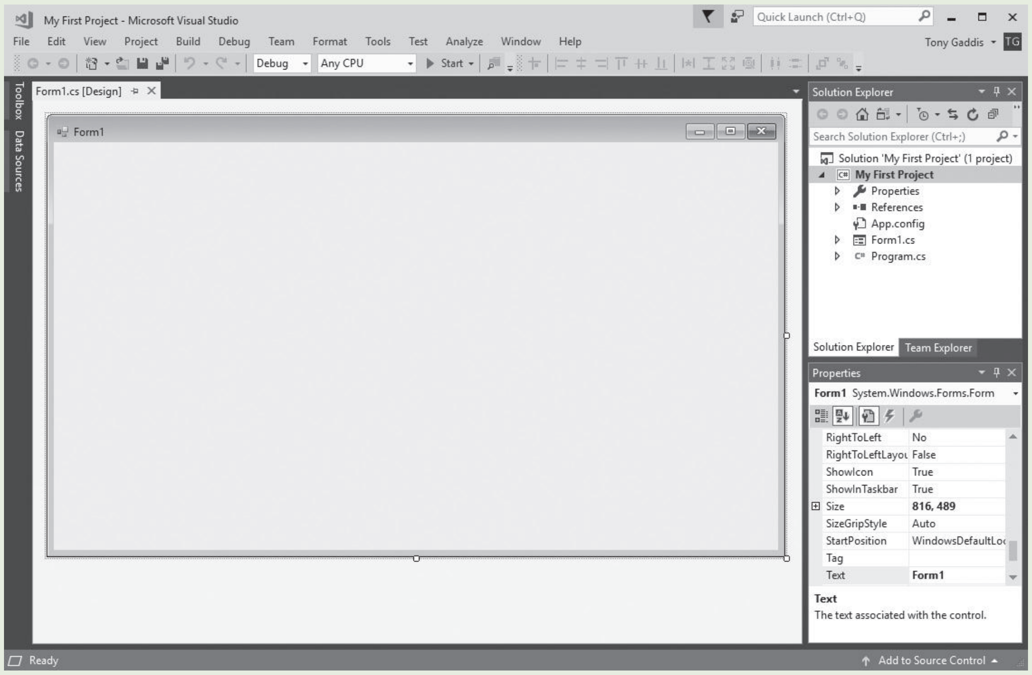
- The *Location* text box shows where a folder will be created to hold the project. If you wish to change the location, click the *Browse* button and select the desired location.
- A solution is a container that holds a project, and the *Solution name* text box shows the name of the solution that will hold this project. By default, the solution name is the same as the project name. For all the projects that you create in this book, you should keep the solution name the same as the project name.

NOTE: As you work through this book, you will create a lot of Visual C# projects. As you do, you will find that default names such as *WindowsFormsApp1* do not help you remember what each project does; therefore, you should always change the name of a new project to something that describes the project's purpose.

Step 5. Click the *Ok* button to create the project. It might take a moment for the project to be created. Once it is, the Visual Studio environment should appear, similar to Figure 1-35. Notice that the name of the project, *My First Project*, is displayed in the title bar at the top of the Visual Studio window.

Leave Visual Studio running and complete the next tutorial.

Figure 1-35 The Visual Studio environment with a new project open



Form1 System.Windows.Forms.Form

RightToLeft	No
RightToLeftLayou	False
ShowIcon	True
ShowInTaskbar	True
Size	816, 489
SizeGripStyle	Auto
StartPosition	WindowsDefaultLoc
Tag	
Text	Form1

Text
The text associated with the control.

VideoNote
Tutorial 1-3: Saving and Closing a Project

Tutorial 1-3:
Saving and Closing a Project

As you work on a project, you should get into the habit of saving it often. In this tutorial you will save the *My First Project* application and then close it.

Step 1: Visual Studio should still be running from the previous tutorial. To save the project that is currently open, click *File* on the menu bar and then select *Save All*.

Step 2: To close the project, click *File* on the menu bar and then click *Close Solution*.

The Visual Studio Environment

The Visual Studio environment consists of a number of windows that you will use on a regular basis. Figure 1-36 shows the locations of the following windows that appear within the Visual Studio environment: the ***Designer* window**, the ***Solution** Explorer* **window**, and the ***Properties* window**. Here is a brief summary of each window's purpose:

- **The *Designer* Window**

  You use the *Designer* window to create an application's GUI. The *Designer* window shows the application's form and allows you to visually design its appearance by placing the desired controls that will appear on the form when the application executes.

- **The *Solution Explorer* Window**

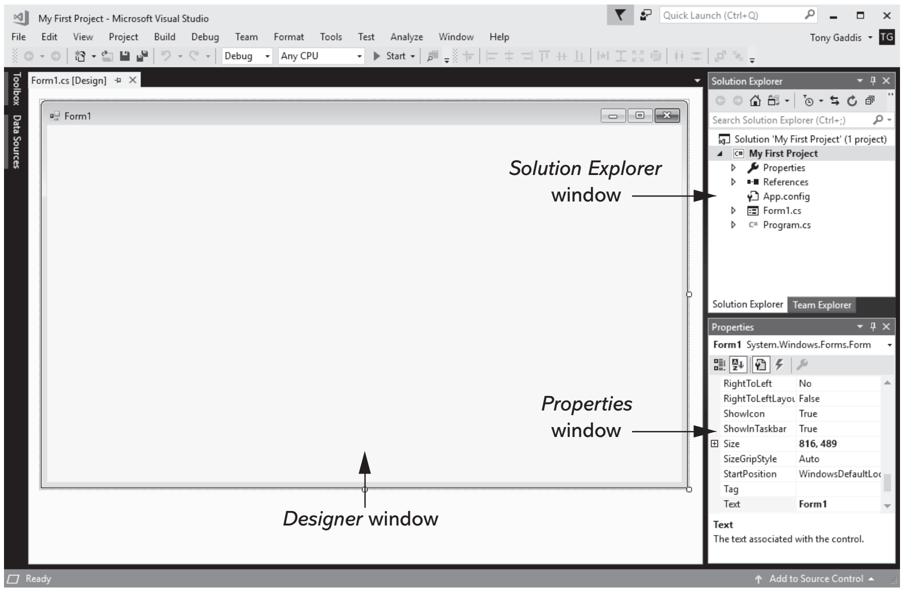
  A solution is a container for holding Visual C# projects. (We discuss solutions in greater detail in a moment.) When you create a new C# project, a new solution is automatically created to contain it. The *Solution Explorer* window allows you to navigate among the files in a Visual C# project.

- **The *Properties* Window**

  A control's appearance and other characteristics are determined by the control's properties. When you are creating a Visual C# application, you use the *Properties* window to examine and change a control's properties.

  Remember that Visual Studio is a customizable environment. You can move the various windows around, so they may not appear in the exact locations shown in Figure 1-36 on your system.

Figure 1-36 The *Designer* window, *Solution Explorer* window, and *Properties* window


My First Project - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Project Build Debug Team Format Tools Test Analyze Window Help
Tony Gaddis TG
Debug Any CPU Start
Toolbox Data Sources
Form1.cs [Design]
Form1
Solution Explorer window
Properties window
Designer window
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'My First Project' (1 project)
My First Project
Properties
References
App.config
Form1.cs
Program.cs
Solution Explorer Team Explorer
Properties
Form1 System.Windows.Forms.Form
Ready
Add to Source Control

RightToLeft	No
RightToLeftLayou	False
ShowIcon	True
ShowInTaskbar	True
Size	816, 489
SizeGripStyle	Auto
StartPosition	WindowsDefaultLoc
Tag	
Text	Form1

Text
The text associated with the control.

Displaying the *Solution Explorer* and *Properties* Windows

If you do not see the *Solution Explorer* or the *Properties* window, you can follow these steps to make them visible:

- If you do not see the *Solution Explorer* window, click *View* on the menu bar. On the *View* menu, click *Solution Explorer*.
- If you do not see the *Properties* window, click *View* on the menu bar. On the *View* menu, click *Properties*.

Using Auto Hide

Many windows in Visual Studio have a feature known as **Auto Hide**. When you see the pushpin icon in a window's title bar, as shown in Figure 1-37, you know that the window has Auto Hide capability. You click the pushpin icon to turn Auto Hide on or off for a window.

When Auto Hide is turned on, the window is displayed only as a tab along one of the edges of the Visual Studio environment. This feature gives you more room to view your application's forms and code. Figure 1-38 shows how the *Solution Explorer* and *Properties* windows appear when their Auto Hide feature is turned on. Notice the tabs that read *Solution Explorer* and *Properties* along the right edge of the screen. (Figure 1-38 also shows a *Team Explorer* tab. We do not discuss the *Team Explorer* in this book.)

Figure 1-37 *Auto Hide* pushpin icon

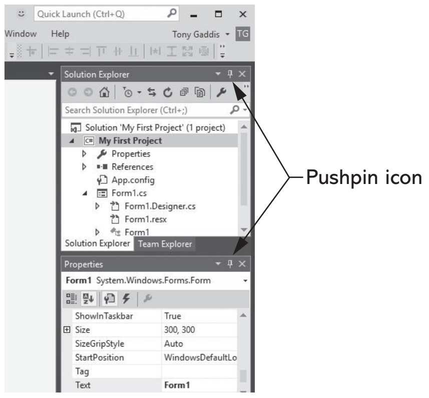
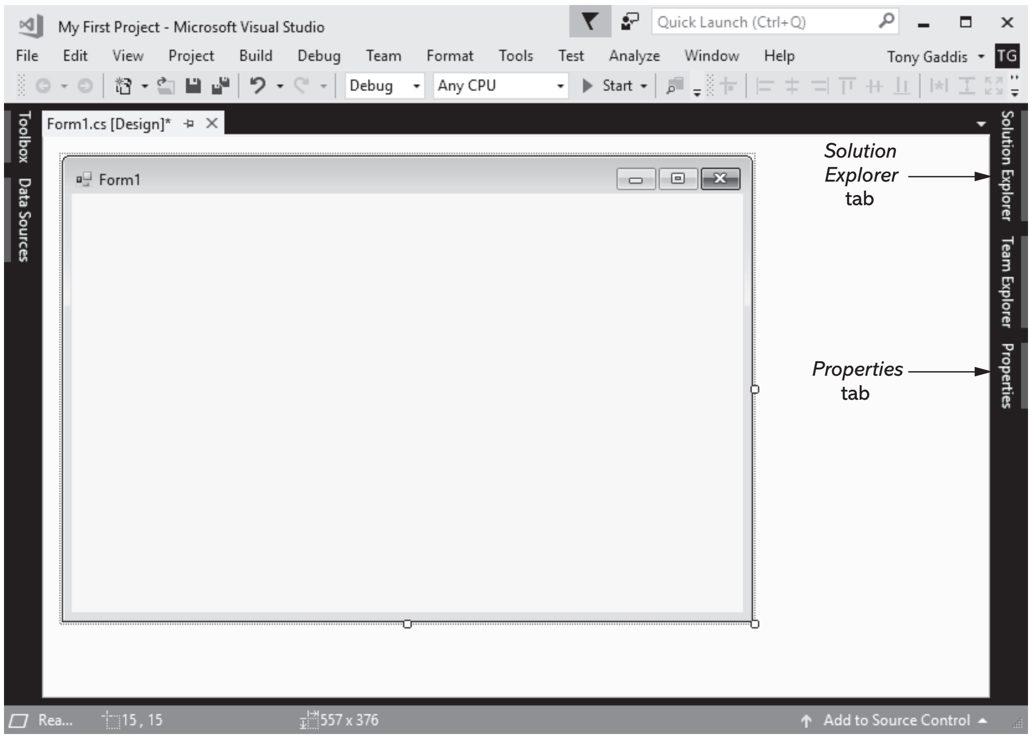


Figure 1-38 The *Solution Explorer* and *Properties* windows hidden



The Menu Bar and the Standard Toolbar

You've already used the Visual Studio menu bar several times. This is the bar at the top of the Visual Studio window that provides menus such as *File*, *Edit*, *View*, *Project*, and so forth. As you progress through this book, you will become familiar with many of the menus.

Below the menu bar is the standard toolbar. The **standard toolbar** contains buttons that execute frequently used commands. All commands that are displayed on the toolbar may also be executed from a menu, but the standard toolbar gives you quicker access to them. Figure 1-39 identifies the standard toolbar buttons that you will use most often, and Table 1-3 gives a brief description of each.

Figure 1-39 Visual Studio toolbar buttons

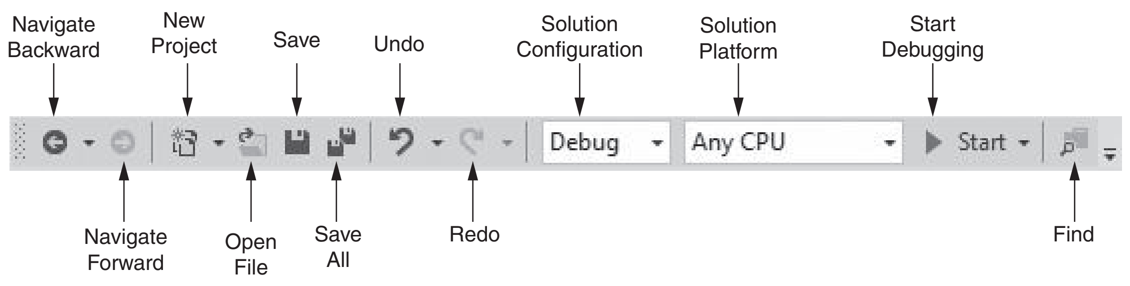


Table 1-3 Visual Studio toolbar buttons

Toolbar Button	Description
Navigate Backward	Moves to the previously active tab in the *Designer* window
Navigate Forward	Moves to the next active tab in the *Designer* window
New Project	Starts a new project
Open File	Opens an existing file
Save	Saves the file named by *filename*
Save All	Saves all the files in the current project
Undo	Undoes the most recent operation
Redo	Redoes the most recently undone operation
Solution Configurations	Configures your project's executable code
Solution Platform	Lets you select the platform on which the application will run
Start Debugging	Starts debugging (running) your program
Find	Searches for text in your application code

The *Toolbox*

The ***Toolbox*** is a window that allows you to select the controls that you want to use in an application's user interface. When you want to place a Button, Label, TextBox, or other control on an application's form, you select it in the *Toolbox*. You will use the *Toolbox* extensively as you develop Visual C# applications.

The *Toolbox* typically appears on the left side of the Visual Studio environment. If the *Toolbox* is in Auto Hide mode, its tab will appear as shown in Figure 1-40. Figure 1-41 shows the *Toolbox* opened, with Auto Hide turned off.

The *Toolbox* is divided into sections, and each section has a name. In Figure 1-41 you can see the *All Windows Forms* and *Common Controls* sections. If you scroll the *Toolbox*, you will see many other sections. Each section can be opened or closed.

Figure 1-40 The *Toolbox* tab (Auto Hide turned on)

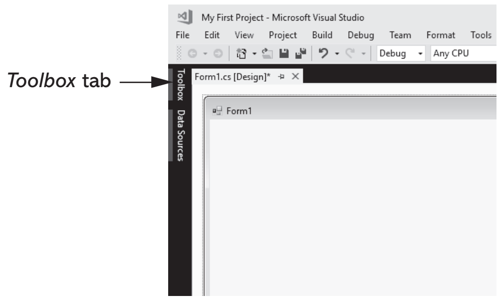
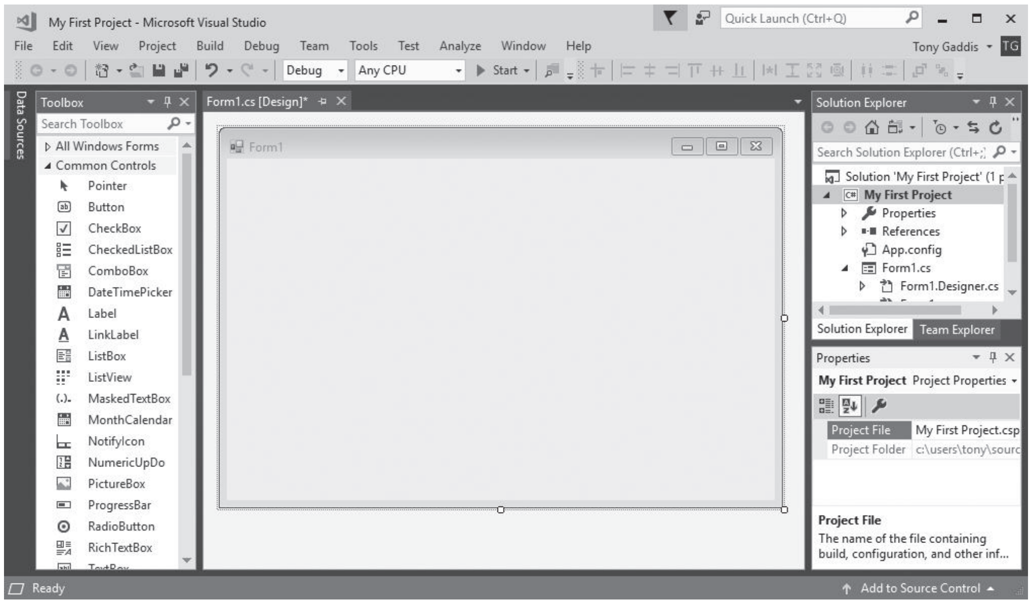

Toolbox tab
My First Project - Microsoft Visual Studio
File Edit View Project Build Debug Team Format Tools
Debug
Any CPU
Toolbox
Data Sources
Form1.cs [Design]*
Form1

Figure 1-41 The *Toolbox* opened (Auto Hide turned off)



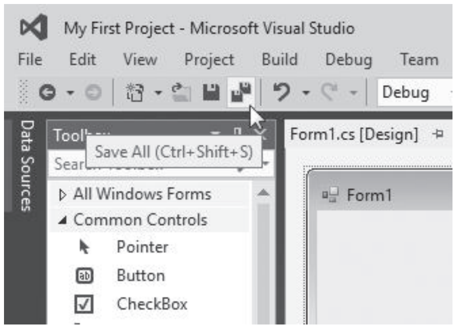
NOTE: If you do not see the *Toolbox* or its tab along the side of the Visual Studio environment, click *View* on the menu bar and then click *Toolbox*.

If you want to open a section of the *Toolbox*, you simply click on its name tab. To close the section, click on its name tab again. In Figure 1-41, the *Common Controls* section is open. You use the *Common Controls* section to access controls that you frequently need, such as Buttons, Labels, and TextBoxes. You can move any section to the top of the list by dragging its name with the mouse.

Using ToolTips

A **ToolTip** is a small rectangular box that pops up when you hover the mouse pointer over a button on the toolbar or in the *Toolbox* for a few seconds. The ToolTip box contains a short description of the button's purpose. Figure 1-42 shows the ToolTip that appears when the cursor is left sitting on the *Save All* button. Use a ToolTip whenever you cannot remember a particular button's function.

Figure 1-42 *Save All* ToolTip



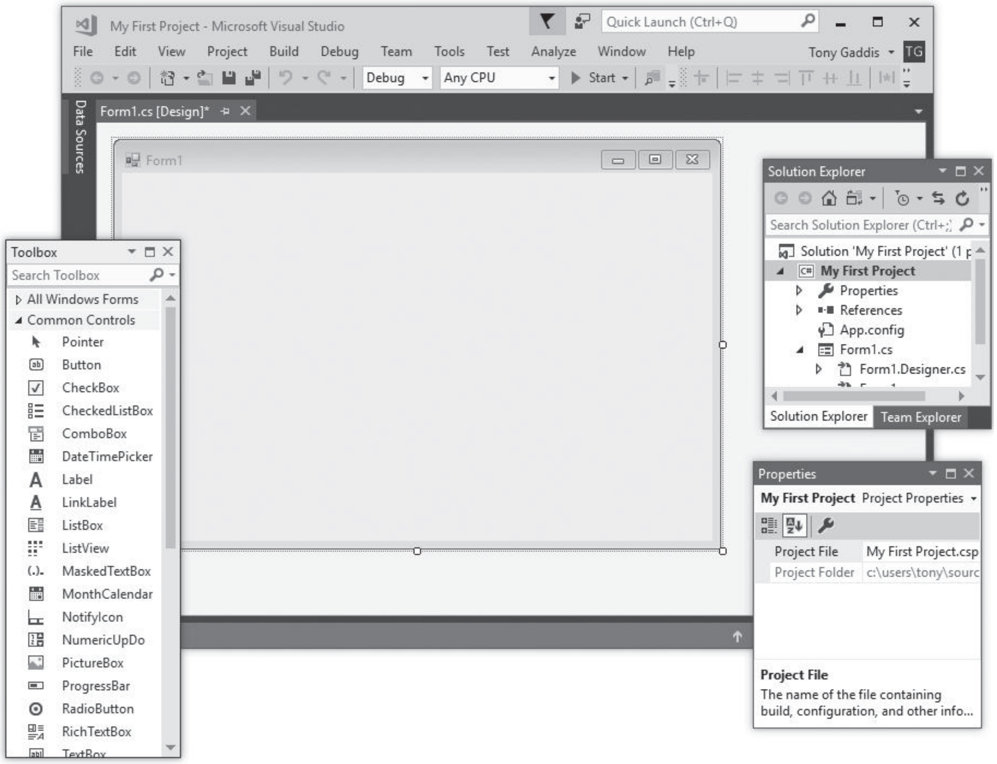
Docked and Floating Windows

Figure 1-41 shows the *Toolbox*, *Solution Explorer*, and *Properties* windows when they are **docked**, which means they are attached to one of the edges of the Visual Studio window. Alternatively, the windows can be **floating**. You can control whether a window is docked or floating as follows:

- To change a window from docked to floating, right-click its title bar and select *Float*.
- To change a window from floating to docked, right-click its title bar and select *Dock*.

Figure 1-43 shows Visual Studio with the *Toolbox*, *Solution Explorer*, and *Properties* windows floating. When a window is floating, you can click and drag it by its title bar

Figure 1-43 *Toolbox, Solution Explorer,* and *Properties* windows floating


My First Project - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Tony Gaddis
Debug
Any CPU
Start
Form1.cs [Design]*
Data Sources
Form1
Toolbox
Search Toolbox
All Windows Forms
Common Controls
Pointer
Button
CheckBox
CheckedListBox
ComboBox
DateTimePicker
Label
LinkLabel
ListBox
ListView
MaskedTextBox
MonthCalendar
NotifyIcon
NumericUpDo
PictureBox
ProgressBar
RadioButton
RichTextBox
TextBox
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'My First Project' (1 p
My First Project
Properties
References
App.config
Form1.cs
Form1.Designer.cs
Solution Explorer
Team Explorer
Properties
My First Project Project Properties
Project File
My First Project.csp
Project Folder
c:\users\tony\sourc
Project File
The name of the file containing build, configuration, and other info...