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, iPods®, 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.
Chapter 1 Introduction to Computers and Programming

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 on. 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.
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
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

![Image of a lab technician holding a modern microprocessor](image)

**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 disk drive stores data by magnetically encoding it onto a circular disk. Most computers have a disk drive mounted inside their case. External disk drives, which connect to one of the computer’s
communication ports, are also available. External disk drives can be used to create backup copies of important data or to move data to another computer.

In addition to external disk drives, many types of devices have been created for copying data and for moving it to other computers. For many years floppy disk drives were popular. A floppy disk drive records data onto a small floppy disk, which can be removed from the drive. Floppy disks have many disadvantages, however. For example, they have limited storage capacity and are slow to access data. The use of floppy disk drives has declined dramatically in recent years in favor of superior devices such as USB drives. Universal serial bus (USB) drives are small devices that plug into the computer’s USB port and appear to the system as disk drives. These drives do not actually contain a disk, however. They store data in a special type of memory known as flash memory. USB drives, which are also known as memory sticks and flash drives, are inexpensive, reliable, and small enough to be carried in a 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, 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 video displays and printers. Disk drives and CD or DVD recorders 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
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.

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 1-11 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

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](image)

**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**.)
1.4 How a Program Works

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. 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.

---

1The example shown is an actual instruction for an Intel microprocessor. It tells the microprocessor to move a value into the CPU.
1.4 How a Program Works

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.

**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

---

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

*Figure 1-16 The fetch-decode-execute cycle*
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

```
mov eax, 2
add eax, 2
mov Y, eax
and so forth...
```

```
10100001
10111000
10011110
and so forth...
```

### 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

<table>
<thead>
<tr>
<th>Language</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada</td>
<td>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.</td>
</tr>
<tr>
<td>BASIC</td>
<td>Beginners All-purpose Symbolic Instruction Code 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.</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>FORmula TRANslator was the first high-level programming language. It was designed in the 1950s for performing complex mathematical calculations.</td>
</tr>
<tr>
<td>COBOL</td>
<td>Common Business-Oriented Language was created in the 1950s and was designed for business applications.</td>
</tr>
<tr>
<td>Pascal</td>
<td>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.</td>
</tr>
<tr>
<td>C and C++</td>
<td>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.</td>
</tr>
<tr>
<td>C#</td>
<td>Pronounced “c sharp,” this language was created by Microsoft around the year 2000 for developing applications based on the Microsoft .NET platform.</td>
</tr>
<tr>
<td>Java</td>
<td>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.</td>
</tr>
<tr>
<td>JavaScript</td>
<td>JavaScript, created in the 1990s, can be used in Web pages. Despite its name, JavaScript is not related to Java.</td>
</tr>
<tr>
<td>Python</td>
<td>Python is a general-purpose language created in the early 1990s. It has become popular in business and academic applications.</td>
</tr>
<tr>
<td>Ruby</td>
<td>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.</td>
</tr>
<tr>
<td>Visual Basic</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 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.
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.

   - Display "Hello, Earthling" and so forth...

   - Machine language program
     - 10100001
     - 10111000
     - 10011110
     - and so forth...

   - Compiler

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 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.
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.

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. GUls also popularized the use of the mouse as an input device. Instead of requiring the user to type commands on the keyboard, GUls 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 1-20 A command line interface](image1.png)

![Figure 1-21 A window in a graphical user interface](image2.png)
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

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 re-type 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

![Diagram of objects used in a GUI](image-url)
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 graphical user interface 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 graphical user interface.

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 graphical user interface, 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 9 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?

**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.
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 Graphical User Interface (GUI)**

   Once you clearly understand what the program is supposed to do, you can begin designing its graphical user interface. 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 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.

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](image)

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 and Visual Studio Express for Windows Desktop consist 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 either Visual Studio 2012 or Visual Studio 2012 Express for Windows Desktop on your computer. Visual Studio 2012 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 C++. If you are using a school’s computer lab, there’s a good chance that Visual Studio 2012 has been installed.
If you do not have access to Visual Studio 2012, you can install Visual Studio 2012 Express for Windows Desktop, a free programming environment that is available for download from the Microsoft Web site. (When this book is purchased new, it has an accompanying Microsoft DVD that contains Visual Studio 2012 Express for Windows Desktop.)

For the purposes of this book, it does not matter whether you are using Visual Studio 2012 or Visual Studio 2012 Express for Windows Desktop. Both products look very similar and work in a similar manner. When there are differences, the book will alert you. To keep things simple, this book will use the term Visual Studio to refer to either Visual Studio 2012 or Visual Studio 2012 Express for Windows Desktop. When you are instructed to use Visual Studio to perform some task, use the system that is installed on your computer.

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

**Step 1:** Find out from your instructor whether you are using Visual Studio 2012 or Visual Studio 2012 Express for Windows Desktop. Then, click the Start button, open the All Programs menu, and perform one of the following:

- If you are using Visual Studio, open the Microsoft Visual Studio 2012 program group and then execute Visual Studio 2012.
- If you are using Visual Studio 2012 Express for Windows Desktop, open the Microsoft Visual Studio 2012 Express program group and then execute VS Express for Desktop.

**NOTE:** The first time you run Visual Studio, you will see a window entitled Choose Default Environment Settings. Select Visual C# Development Settings from the list and click the Start Visual Studio button.

**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.
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. If you are using Visual Studio Express, click the Finish button at this point, and proceed to Step 4 of the tutorial. If you are using Visual Studio, click the Next > button.
- If you are using Visual Studio, the window shown in Figure 1-31 should appear next. Select Visual C# Development Settings 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.

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 1-30 Selecting Tools and then Import and Export Settings …

Figure 1-31 Selecting Visual C# Development Settings
1.8 Getting Started with the Visual Studio Environment

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

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

Step 2: If you are using Visual Studio 2012: 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.

Figure 1-32 Resetting the window layout

Figure 1-33 The New Project window
If you are using Visual Studio 2012 Express: Click FILE on the menu bar at the top of the screen and then select New Project. After doing this, a New Project window similar to Figure 1-33 should be displayed. (With Visual Studio Express, the window will have fewer items than shown in the figure.)

Step 3: At the left side of the window, under Installed Templates, make sure Visual C# is selected. Then, select Windows Forms Application, 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 WindowsApplication1. Change the project name to My First Project, as shown in Figure 1-34.

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

If you are using Visual Studio 2012: Just below the Name text box you will see a Location text box and a Solution name text box.

- 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 WindowsApplication1 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.
1.8 Getting Started with the Visual Studio Environment

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

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 graphical user interface. 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.

**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. You might see this tab if you are using Visual Studio 2012. We do not discuss the Team Explorer in this book.)
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

![Visual Studio toolbar buttons](image)

**Table 1-3** Visual Studio toolbar buttons

<table>
<thead>
<tr>
<th>Toolbar Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigate Backward</td>
<td>Moves to the previously active tab in the <strong>Designer</strong> window</td>
</tr>
<tr>
<td>Navigate Forward</td>
<td>Moves to the next active tab in the <strong>Designer</strong> window</td>
</tr>
<tr>
<td>New Project</td>
<td>Starts a new project</td>
</tr>
<tr>
<td>Open File</td>
<td>Opens an existing file</td>
</tr>
<tr>
<td>Save</td>
<td>Saves the file named by <strong>filename</strong></td>
</tr>
<tr>
<td>Save All</td>
<td>Saves all the files in the current project</td>
</tr>
<tr>
<td>Undo</td>
<td>Undoes the most recent operation</td>
</tr>
<tr>
<td>Redo</td>
<td>Redoes the most recently undone operation</td>
</tr>
<tr>
<td>Start Debugging</td>
<td>Starts debugging (running) your program</td>
</tr>
<tr>
<td>Solution Configurations</td>
<td>Configures your project’s executable code</td>
</tr>
<tr>
<td>Find</td>
<td>Searches for text in your application code</td>
</tr>
</tbody>
</table>

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

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. (In Visual Studio Express, click VIEW on the menu bar, then click Other Windows, and then click Toolbox.)

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 1-41  The Toolbox opened (Auto Hide turned off)
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](image)

**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 around the screen. You may use whichever style you prefer—docked or floating. When windows are floating, they behave as normal windows. You may move or resize them to suit your preference.

**NOTE:** A window cannot float if its Auto Hide feature is turned on.

**TIP:** Remember, you can always reset the window layout by clicking WINDOW on the menu bar and then selecting Reset Window Layout. If you accidentally close the Designer window, the Solution Explorer window, or the Properties window, you can use the VIEW menu to redisplay them.
1.8 Getting Started with the Visual Studio Environment

Projects and Solutions

As you learn to program in Visual C#, you will see the terms *project* and *solution* used often. These terms do not mean the same thing, but they are sometimes used interchangeably. Let’s briefly discuss the difference between a project and a solution.

Each Visual C# application that you create is called a project. A Visual C# project consists of several files. You can think of a project as a collection of files that belong to a single application.

A *solution* is a container that holds one or more Visual C# projects. If you are developing applications for a large organization, you might find it convenient to store several related projects together in the same solution.

Although it is possible for a solution to hold more than one project, each project that you will create in this book will be saved in its own solution. Each time you create a new project, you will also create a new solution to hold it. Figure 1-44 illustrates this concept. Typically, the solution will be given the same name as the project.

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

Figure 1-44 Solution and project organization
Typical Organization of Solutions and Projects on the Disk

When you create a new project, you specify the project’s name, the solution’s name, and a location on the disk where the solution should be stored. If you are using Visual Studio, you specify this information at the bottom of the New Project window, as shown in Figure 1-45. If you are using Visual Studio Express, you specify this information in the Save Project window the first time you save the project.

<table>
<thead>
<tr>
<th>Figure 1-45</th>
<th>Specifying the project name, solution name, and location</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Diagram" /></td>
<td>This is where the solution folder will be created</td>
</tr>
</tbody>
</table>

Let’s use Figure 1-45 to see an example of how the files for the My First Project solution and project will be organized on the disk. Notice that in Figure 1-45, the following location is shown for the solution:

`C:\Users\Tony\Documents\Visual Studio 2012\Projects\`

On your system, the location will not be exactly the same as this, but it will be something similar. At this location, a solution folder named My First Project will be created. If we use Windows to look inside that folder, we will see the two items shown in Figure 1-46. Notice that one of the items is another folder named My First Project. That is the project folder, which contains various files related to the project. The other item is the solution file. In Windows, you can double-click the solution file to open the project in Visual Studio.

<table>
<thead>
<tr>
<th>Figure 1-46</th>
<th>Contents of the My First Project solution folder</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image2.png" alt="Windows Explorer" /></td>
<td></td>
</tr>
</tbody>
</table>
Opening an Existing Project

If Visual Studio is already running, you can perform the following steps to open an existing project:

- Click **FILE** on the Visual Studio menu bar, then select **Open**, and then select **Project/Solution**. . .
- The **Open Project** window will appear. Navigate to the desired solution folder, select the solution file, and click **Open**.

In Visual Studio Express, perform the following steps to open an existing project:

- Click **FILE** on the Visual Studio menu bar and then select **Open Project**. . .
- The **Open Project** window will appear. Navigate to the desired solution folder, select the solution file, and click **Open**.

Tutorial 1-4:
Opening an Existing Project

In this tutorial you will reopen the *My First Project* application that you created in Tutorial 1-2.

**Step 1:** Visual Studio should still be running from the previous tutorial. Perform one of the following operations to reopen *My First Project*:

If you are using Visual Studio:

Click **FILE** on the menu bar; select **Open** and then select **Project/Solution**. . .

The **Open Project** window will appear. Navigate to the *My First Project* solution, select the solution file, and click the **Open** button.

If you are using Visual Studio Express:

Click **FILE** on the menu bar, and then select **Open Project**. . . The **Open Project** window will appear. Navigate to the *My First Project* solution, select the solution file, and click the **Open** button.

After performing this step, *My First Project* should be opened. If you plan to complete the next tutorial, leave Visual Studio running with *My First Project* opened.

Displaying the Designer (When It Does Not Automatically Appear)

Sometimes when you open an existing project, the project’s form will not be automatically displayed in the **Designer**. Figure 1-47 shows an example of the Visual Studio environment with an opened project but no form displayed in the **Designer**. When this happens, perform the following steps to display the project’s form in the **Designer**:

- As shown in Figure 1-48, right-click Form1.cs in the **Solution Explorer**.
- Click **View Designer** in the pop-up menu.

Accessing the Visual Studio Documentation

You can access the documentation for Visual Studio by Clicking **HELP** on the menu bar, and then selecting **View Help**. (Or, you can press **Ctrl**+**F1**, and then press **V** on the keyboard.) This launches your Web browser and opens the online Microsoft Developer
Network (MSDN) Library. The MSDN Library provides complete documentation for Visual C# as well as the other programming languages included in Visual Studio. You will also find code samples, tutorials, articles, and access to tutorial videos.
1.8 Getting Started with the Visual Studio Environment

**Tutorial 1-5:**
Getting Familiar with the Visual Studio Environment

This exercise will give you practice interacting with the *Solution Explorer* window, the *Properties* window, and the *Toolbox*.

**Step 1:** If Visual Studio is still running on your computer from the previous tutorial, continue to Step 2. If Visual Studio is not running on your computer, repeat the steps in Tutorial 1-4 to open *My First Project*.

**Step 2:** Practice turning the Auto Hide feature on and off for the *Solution Explorer* window, the *Properties* window, and the *Toolbox*. Recall from our previous discussion that clicking the pushpin icon in each window’s title bar turns Auto Hide on and off. When you are finished practicing, make sure Auto Hide is turned off for each of these windows. Your screen should look like Figure 1-41.

**Step 3:** Practice floating and docking the *Solution Explorer* window, the *Properties* window, and the *Toolbox*. Recall from our previous discussion that you can make any of these windows float by right-clicking its title bar and selecting *Float*. You dock a floating window by right-clicking its title bar and selecting *Dock*.

**Step 4:** The *Toolbox*, *Solution Explorer*, and *Properties* windows each have a *Close* button (x) in their upper-right corner. Close each of these windows by clicking its *Close* button.

**Step 5:** Do you remember which buttons on the toolbar restore the *Solution Explorer*, *Properties* window, and *Toolbox*? If not, move your mouse cursor over any button on the toolbar and leave it there until the ToolTip appears. Repeat this procedure on different buttons until you find the ones whose ToolTips read *Solution Explorer*, *Properties Window*, and *Toolbox*. (Refer to Figure 1-39 and Table 1-3 for further assistance.)

**Step 6:** Click the appropriate buttons on the toolbar to restore the *Solution Explorer*, the *Properties* window, and the *Toolbox*.

**Step 7:** Exit Visual Studio by clicking *FILE* on the menu bar and then clicking *Exit*. You may see a dialog box asking whether you wish to save changes to a number of items. Click *Yes*.

---

**Checkpoint**

1.44 Briefly describe the purpose of the *Solution Explorer* window.
1.45 Briefly describe the purpose of the *Properties* window.
1.46 Briefly describe the purpose of the standard toolbar.
1.47 What is the difference between the toolbar and the *Toolbox*?
1.48 What is a ToolTip?
1.49 What is a project?
1.50 What is a solution?
Key Terms

algorithm
American Standard Code for Information Interchange (ASCII)
application software
assembler
assembly language
Auto Hide
binary
binary digit
binary numbering system
bit
Button
byte
central processing unit (CPU)
class
code
command line interface
compact disc (CD)
compiler
console interface
controls
debug
Designer window
digital
digital data
digital device
digital versatile disc (DVD)
disk drive
docked (window)
End terminal
ENIAC
event driven
executing
fetch-decode-execute cycle
fields
floating (window)
floating-point notation
floppy disk drive
flowchart
Form
graphical user interface (GUI)
hardware
high-level languages
input
input device
input symbols
instruction set
integrated development environment (IDE)
interpreter
keywords
Label
logic error
low-level language
machine language
main memory
methods
microprocessors
mnemonics
.NET Framework
object oriented
operating system
operators
output
output device
output symbols
picture element
pixel
processing symbols
program
program development cycle
programmer
project
project folder
properties
Properties window
pseudocode
random-access memory (RAM)
reserved words
running
samples
secondary storage
software
software developer
software development tools
solution
Solution Explorer window
solution file
solution folder
source code
standard toolbar
start terminal
statements
syntax
syntax error
system software
terminal symbol
TextBox
Toolbox
ToolTip
two's complement
Unicode
**Review Questions**

**Multiple Choice**

1. A(n) __________ is a set of instructions that a computer follows to perform a task.
   - a. compiler
   - b. program
   - c. interpreter
   - d. programming language

2. The physical devices that a computer is made of are referred to as __________.
   - a. hardware
   - b. software
   - c. the operating system
   - d. tools

3. The part of a computer that runs programs is called __________.
   - a. RAM
   - b. secondary storage
   - c. main memory
   - d. the CPU

4. Today, CPUs are small chips known as __________.
   - a. ENIACs
   - b. microprocessors
   - c. memory chips
   - d. operating systems

5. The computer stores a program while the program is running, as well as the data that the program is working with, in __________.
   - a. secondary storage
   - b. the CPU
   - c. main memory
   - d. the microprocessor

6. __________ is a volatile type of memory that is used only for temporary storage while a program is running.
   - a. RAM
   - b. secondary storage
   - c. the disk drive
   - d. the USB drive

7. A type of memory that can hold data for long periods of time—even when there is no power to the computer—is called __________.
   - a. RAM
   - b. main memory
   - c. secondary storage
   - d. CPU storage
8. A component that collects data from people or other devices and sends it to the computer is called __________.
   a. an output device
   b. an input device
   c. a secondary storage device
   d. main memory

9. A video display is a(n) __________ device.
   a. output device
   b. input device
   c. secondary storage device
   d. main memory

10. A __________ is enough memory to store a letter of the alphabet or a small number.
    a. byte
    b. bit
    c. switch
    d. transistor

11. A byte is made up of eight __________.
    a. CPUs
    b. instructions
    c. variables
    d. bits

12. In the __________ numbering system, all numeric values are written as sequences of 0s and 1s.
    a. hexadecimal
    b. binary
    c. octal
    d. decimal

13. A bit that is turned off represents the following value: __________.
    a. 1
    b. −1
    c. 0
    d. “no”

14. A set of 128 numeric codes that represent the English letters, various punctuation marks, and other characters is __________.
    a. binary numbering
    b. ASCII
    c. Unicode
    d. ENIAC

15. An extensive encoding scheme that can represent the characters of many of the languages in the world is __________.
    a. binary numbering
    b. ASCII
    c. Unicode
    d. ENIAC

16. Negative numbers are encoded using the __________ technique.
    a. two’s complement
    b. floating point
    c. ASCII
    d. Unicode
17. Real numbers are encoded using the __________ technique.
   a. two’s complement
   b. floating point
   c. ASCII
   d. Unicode

18. The tiny dots of color that digital images are composed of are called __________.
   a. bits
   b. bytes
   c. color packets
   d. pixels

19. If you were to look at a machine language program, you would see __________.
   a. C# code
   b. a stream of binary numbers
   c. English words
   d. circuits

20. In the __________ part of the fetch-decode-execute cycle, the CPU determines which
    operation it should perform.
    a. fetch
    b. decode
    c. execute
    d. immediately after the instruction is executed

21. Computers can execute only programs that are written in __________.
    a. C#
    b. assembly language
    c. machine language
    d. Java

22. The __________ translates an assembly language program to a machine language
    program.
    a. assembler
    b. compiler
    c. translator
    d. interpreter

23. The words that make up a high-level programming language are called __________.
    a. binary instructions
    b. mnemonics
    c. commands
    d. keywords

24. The rules that must be followed when writing a program are called __________.
    a. syntax
    b. punctuation
    c. keywords
    d. operators

25. A(n) __________ is a program that translates a high-level language program into a
    separate machine language program.
    a. assembler
    b. compiler
    c. translator
    d. utility
26. A ________ is any hypothetical person using a program and providing input for it.
   a. designer
   b. user
   c. guinea pig
   d. test subject

27. A ________ error does not prevent the program from running but causes it to produce incorrect results.
   a. syntax
   b. hardware
   c. logic
   d. fatal

28. A(n) ________ is a set of well-defined logical steps that must be taken to perform a task.
   a. logarithm
   b. plan of action
   c. logic schedule
   d. algorithm

29. An informal language that has no syntax rules and is not meant to be compiled or executed is called ________.
   a. faux code
   b. pseudocode
   c. C#
   d. a flowchart

30. A ________ is a diagram that graphically depicts the steps that take place in a program.
   a. flowchart
   b. step chart
   c. code graph
   d. program graph

31. Objects that are visible in a program’s graphical user interface are commonly referred to as ________.
   a. buttons
   b. controls
   c. forms
   d. windows

32. A ________ is code that describes a particular type of object.
   a. namespace
   b. blueprint
   c. schema
   d. class

33. The ________ 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.
   a. .NET framework
   b. Standard Template Library
   c. GUI framework
   d. MSDN Library
34. The ________ is the part of a computer with which the user interacts.
   a. central processing unit
   b. user interface
   c. control system
   d. interactivity system

35. Before GUIs became popular, the ________ interface was the most commonly used.
   a. command line
   b. remote terminal
   c. sensory
   d. event-driven

36. ________ programs are usually event driven.
   a. command line
   b. text-based
   c. GUI
   d. procedural

True or False
1. Today, CPUs are huge devices made of electrical and mechanical components such as vacuum tubes and switches.
2. Main memory is also known as RAM.
3. Any piece of data that is stored in a computer’s memory must be stored as a binary number.
4. Images, such as the ones you make with your digital camera, cannot be stored as binary numbers.
5. Machine language is the only language that a CPU understands.
6. Assembly language is considered a high-level language.
7. An interpreter is a program that both translates and executes the instructions in a high-level language program.
8. A syntax error does not prevent a program from being compiled and executed.
9. Windows, Linux, UNIX, and Mac OS are all examples of application software.
10. Word processing programs, spreadsheet programs, e-mail programs, Web browsers, and games are all examples of utility programs.
11. Programmers must be careful not to make syntax errors when writing pseudocode programs.
12. C# provides only the basic keywords and operators that you need to construct a program.

Short Answer
1. Why is the CPU the most important component in a computer?
2. What number does a bit that is turned on represent? What number does a bit that is turned off represent?
3. What would you call a device that works with binary data?
4. What are the words that make up a high-level programming language called?
5. What are the short words that are used in assembly language called?

6. What is the difference between a compiler and an interpreter?

7. What type of software controls the internal operations of the computer’s hardware?

8. What is pseudocode? What is a flowchart?

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

10. What does a class specify about an object?

11. Can you use C# alone to perform advanced operations such as creating GUIs, reading data from a file, or working with databases? Why or why not?

12. Figure 1-49 shows the Visual Studio IDE. What are the names of the four areas that are indicated in the figure?

13. What is the purpose of the Toolbox in the Visual Studio environment?

14. How can you access the documentation for Visual Studio? What resources are provided by the MSDN Library?

15. What steps must you take to open an existing project?

16. How can you view the project’s form if it is not automatically displayed in the Designer?

---

**Figure 1-49** The Visual Studio IDE

---

**Programming Problems**

1. Use what you’ve learned about the binary numbering system in this chapter to convert the following decimal numbers to binary:
   - 11
   - 65
   - 100
   - 255
2. Use what you’ve learned about the binary numbering system in this chapter to convert the following binary numbers to decimal:
   1101
   1000
   101011

3. Look at the ASCII chart in Appendix C and determine the codes for each letter of your first name.

4. Suppose your instructor gives three exams during the semester and you want to write a program that calculates your average exam score. Answer the following:
   a. What items of input must the user enter?
   b. Once the input has been entered, how will the program determine the average?
   c. What output will the program display?
This page intentionally left blank
TOPICS

2.1 Getting Started with Forms and Controls
2.2 Creating the GUI for Your First Visual C# Application: The Hello World Application
2.3 Introduction to C# Code
2.4 Writing Code for the Hello World Application
2.5 Label Controls
2.6 Making Sense of IntelliSense
2.7 PictureBox Controls
2.8 Comments, Blank Lines, and Indentation
2.9 Writing the Code to Close an Application's Form
2.10 Dealing with Syntax Errors

CONCEPT: The first step in creating a Visual C# application is creating the application’s GUI. You use the Visual Studio Designer, Toolbox, and Properties window to build the application’s form with the desired controls and set each control’s properties.

In this chapter you will create your first Visual C# application. Before you start, however, you need to learn some fundamental concepts about creating a GUI in Visual Studio. This section shows the basics of editing forms and creating controls.

The Application’s Form

When you start a new Visual C# project, Visual Studio automatically creates an empty form and displays it in the Designer. Figure 2-1 shows an example. Think of the empty form as a blank canvas that can be used to create the application’s user interface. You can add controls to the form, change the form’s size, and modify many of its characteristics. When the application runs, the form will be displayed on the screen.
If you take a closer look at the form, you will notice that it is enclosed by a thin dotted line, known as a bounding box. As shown in Figure 2-2, the bounding box has small sizing handles, which appear on the form’s right edge, bottom edge, and lower-right corner. When a bounding box appears around an object in the Designer, it indicates that the object is selected and is ready for editing.

Initially the form’s size is 300 pixels wide by 300 pixels high. You can easily resize the form with the mouse. When you position the mouse cursor over any edge or corner that has a sizing handle, the cursor changes to a two-headed arrow (\(\rightarrow\)). Figure 2-3 shows examples. When the mouse cursor becomes a two-headed arrow, you can click and drag the mouse to resize form.
The Properties Window

The appearance and other characteristics of a GUI object are determined by the object’s properties. When you select an object in the Designer, that object’s properties are displayed in the Properties window. For example, when the Form1 form is selected, its properties are displayed in the Properties window, as shown in Figure 2-4.

The area at the top of the Properties window shows the name of the object that is currently selected. You can see in Figure 2-4 that the name of the selected object is Form1. Below that is a scrollable list of properties. The list of properties has two columns: The left column shows each property’s name, and the right column shows each property’s value. For example, look at the form’s Size property in Figure 2-4. Its value is 300, 300. This means that the form’s size is 300 pixels wide by 300 pixels high. Next, look at the form’s Text property. The Text property determines the text that is displayed in the form’s title bar (the bar that appears at the top of the form). Its current value is Form1, so the text Form1 is displayed in the form’s title bar.

When a form is created, its Text property is initially set to the same value as the form’s name. When you start a new project, the blank form that appears in the Designer will always be named Form1, so the text Form1 will always appear in the form’s title bar. In most cases you want to change the value of the form’s Text property to something more meaningful. For example, assume the Form1 form is currently selected. You can perform the following steps to change its Text property to My First Program.

Identifying Forms and Controls by Their Names

An application’s GUI is made of forms and various controls. Each form and control in an application’s GUI must have a name that identifies it. The blank form that Visual Studio initially creates in a new project is named Form1.

NOTE: Later in this book you will learn how to change a form’s name, but for now, you will keep the default name, Form1.
Step 1: In the Properties window, locate the Text property.
Step 2: Double-click the word Form1 that currently appears as the Text property’s value, and then use the [Delete] key to delete it.
Step 3: Type My First Program in its place and press the [Enter] key. The text My First Program will now appear in the form’s title bar, as shown in Figure 2-5.

**NOTE:** Changing an object’s Text property does not change the object’s name. For example, if you change the Form1 form’s Text property to My First Program, the form’s name is still Form1. You have changed only the text that is displayed in the form’s title bar.
Earlier we discussed how to use the mouse to resize a form in the **Designer**. An alternative method is to change the form’s Size property in the **Properties** window. For example, assume the Form1 form is currently selected. You can perform the following steps to change its size to 400 pixels wide by 100 pixels high.

**Step 1:** In the **Properties** window, locate the Size property.
**Step 2:** Click inside the area that holds the Size property’s value, and then delete the current value.
**Step 3:** Type 400, 100 in its place and press the **Enter** key. The form will be resized as shown in Figure 2-6.

**Figure 2-6** The form’s size changed to 400 by 100

---

**NOTE:** Notice in Figure 2-6 that the **Alphabetical button** (¶) is selected near the top of the **Properties** window. This causes the properties to be displayed in alphabetical order. Alternatively, the **Categorized button** (¶) can be selected, which causes the properties to be displayed in groups. The alphabetical listing is the default selection, and most of the time, it makes it is easier to locate specific properties.

---

**Adding Controls to a Form**

When you are ready to create controls on the application’s form, you use the **Toolbox**. Recall from Chapter 1 that the **Toolbox** usually appears on the left side of the Visual Studio environment. If the **Toolbox** is in Auto Hide mode, you can click its tab to open it. Figure 2-7 shows an example of how the **Toolbox** typically appears when it is open.

**TIP:** Recall from Chapter 1 that if you do not see the **Toolbox** or its tab, click **VIEW** on the menu bar and then click **Toolbox**.

The **Toolbox** shows a scrollable list of controls that you can add to a form. To add a control to a form, you simply find it in the **Toolbox** and then double-click it. The control will be created on the form. For example, suppose you want to create a Button control on the form. You find it in the **Toolbox**, as shown in Figure 2-8, double-click it, and a Button control will appear on the form.
Figure 2-7 The Toolbox

Figure 2-8 Creating a Button control

**TIP:** You can also click and drag controls from the Toolbox onto the form.

**Resizing and Moving Controls**

Take a closer look at the Button control that is shown on the form in Figure 2-8. Notice that it is enclosed in a bounding box with sizing handles. This indicates that the control is currently selected. When a control is selected, you can use the mouse to resize it in the same way that you learned to resize a form earlier. You can also use the mouse to move a control to a
new location on the form. Position the mouse cursor inside the control, and when the mouse
cursor becomes a four-headed arrow ( ), you can click and drag the control to a new
location. Figure 2-9 shows a form with a Button control that has been enlarged and moved.

Figure 2-9 A Button control resized and moved

Deleting a Control
Deleting a control is simple: you select it and then press the key on the keyboard.

More about Button Controls
You learned earlier that each form and each control in an application’s GUI must have
a name that identifies it. When you create Button controls, they are automatically given
default names such as button1, button2, and so forth.

Button controls have a Text property, which holds the text that is displayed on the face
of the button. When a Button control is created, its Text property is initially set to the
same value as the Button control’s name. As a result, when you create a Button control,
its name will be displayed on the face of the button. For example, the form in Figure 2-10
contains three Button controls named button1, button2, and button3.

Figure 2-10 A form with three Button controls

After you create a Button control, you should always change its Text property. The text
that is displayed on a button should indicate what the button will do when it is clicked.
For example, a button that calculates an average might have the text Calculate Average
displayed on it, and a button that prints a report might have the text Print Report dis-
played on it. Here are the steps you perform to change a Button control’s Text property:

Step 1: Make sure the Button control is selected. (If you don’t see the bounding box
and sizing handles around the control, just click the control to select it.)
Step 2: In the Properties window, locate the Text property.
Step 3: Click inside the area that holds the Text property’s value, and then delete the
current value. Then, type the new text in its place and press the key. The
new text will be displayed on the button.
Figure 2-11 shows an example of how changing a Button control’s Text property changes the text displayed on the face of the button.

**Figure 2-11 A Button control’s Text property changed**

![Image of a Button control with its Text property changed from 'Click Me' to 'A Button control’s Text property changed']

**Changing a Control’s Name**

A control’s name identifies the control in the application’s code and in the Visual Studio environment. When you create a control on an application’s form, you should always change the control’s name to something that is more meaningful than the default name that Visual Studio gives it. A control’s name should reflect the purpose of the control.

For example, suppose you’ve created a Button control to calculate an amount of tax. A default name such as `button1` does not convey the button’s purpose. A name such as `calculateTaxButton` would be much better. When you are working with the application’s code and you see the name `calculateTaxButton`, you will know precisely which button the code is referring to.

You can change a control’s name by changing its Name property. Here are the steps:

**Step 1:** Make sure the control is selected. (If you do not see the bounding box and sizing handles around the control, just click the control to select it.)

**Step 2:** In the Properties window, scroll up to the top of the list of properties. You should see the Name property, as shown in Figure 2-12. (The Name property is enclosed in parentheses to make it appear near the top of the alphabetical list of properties. This makes it easier to find.)

**Step 3:** Click inside the area that holds the Name property’s value and then delete the current name. Then, type the new name in its place and press the Enter key. You have successfully changed the name of the control.

**Figure 2-12 The Name property**

![Image of the Properties window with the Name property selected and the text 'button1' highlighted]
Figure 2-13 shows the Properties window after a Button control’s name has been changed to `calculateTaxButton`.

**Figure 2-13** The Name property changed to `calculateTaxButton`

### Rules for Naming Controls

Control names are also known as identifiers. When naming a control, you must follow these rules for C# identifiers:

- The first character must be one of the letters `a` through `z` or `A` through `Z` or an underscore character `_`.
- After the first character, you may use the letters `a` through `z` or `A` through `Z`, the digits 0 through 9, or underscores.
- The name cannot contain spaces.

Table 2-1 lists some identifiers that might be used for Button control names and indicates whether each is a legal or illegal identifier in C#.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Legal or Illegal?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>showDayOfWeekButton</code></td>
<td>Legal</td>
</tr>
<tr>
<td><code>3rdQuarterButton</code></td>
<td>Illegal because identifiers cannot begin with a digit.</td>
</tr>
<tr>
<td><code>change*color*Button</code></td>
<td>Illegal because the <code>*</code> character is not allowed.</td>
</tr>
<tr>
<td><code>displayTotalButton</code></td>
<td>Legal</td>
</tr>
<tr>
<td><code>calculate Tax Button</code></td>
<td>Illegal because identifiers cannot contain spaces.</td>
</tr>
</tbody>
</table>

Because a control’s name should reflect the control’s purpose, programmers often find themselves creating names that are made of multiple words. For example, consider the following Button control names:

- `calculatetaxbutton`
- `printreportbutton`
- `displayanimationbutton`

Unfortunately, these names are not easily read by the human eye because the words are not separated. Because we cannot have spaces in control names, we need to find another way to separate the words in a multiword control name to make it more readable to the human eye.
Most C# programmers address this problem by using the camelCase naming convention for controls. camelCase names are written in the following manner:

- You begin writing the name with lowercase letters.
- The first character of the second and subsequent words is written in uppercase.

For example, the following control names are written in camelCase:

- calculateTaxButton
- printReportButton
- displayAnimationButton

**NOTE:** This style of naming is called camelCase because the uppercase characters that appear in a name are sometimes reminiscent of a camel’s humps.

**Checkpoint**

2.1 When you start a new Visual C# project, what is automatically created and displayed in the Designer?

2.2 How can you tell that an object is selected and ready for editing in the Designer?

2.3 What is the purpose of an object’s sizing handles?

2.4 What must each form and control in an application’s GUI have to identify it?

2.5 What is the purpose of the Properties window?

2.6 What does the Alphabetical button do when it is selected in the Properties window?

2.7 What does the Categorized button do when it is selected in the Properties window?

2.8 What does a form’s Text property determine?

2.9 What does a form’s Size property determine?

2.10 What is shown in the Toolbox?

2.11 How do you add a control to a form?

2.12 What should the text that is displayed on a button indicate?

2.13 What are the rules for naming controls?

2.14 What naming convention do most C# programmers use to separate words in a multiword identifier?

**2.2 Creating the GUI for Your First Visual C# Application: The Hello World Application**

When a student is learning computer programming, it is traditional to start by learning to write a Hello World program. A Hello World program is a simple program that merely displays the words “Hello World” on the screen. In this chapter you will create your first Visual C# application, which will be an event-driven Hello World program. When the finished application runs, it will display the form shown on the left in Figure 2-14. Notice that the form contains a button that reads Display Message. When you click the button, the window shown on the right in the figure will appear.
The process of creating this application is divided into two parts. First, you will create the application’s GUI, and second, you will write the code that causes the Hello World message to appear when the user clicks the Display Message button. Tutorial 2-1 leads you through the process of creating the GUI.

**Tutorial 2-1:**
Creating the GUI for the Hello World Application

**Step 1:** Start Visual Studio (or Visual Studio Express).

**Step 2:** Start a new project by performing one of the following actions:
- If you are using Visual Studio, click FILE on the menu bar, then select New, and then select Project…
- If you are using Visual Studio Express, click FILE on the menu bar and then select New Project…

**Step 3:** The New Project window should appear. At the left side of the window, under Installed > Templates, make sure Visual C# is selected. Then, select Windows Forms Application as the type of application. In the Name text box (at the bottom of the window), change the name of the project to Hello World, and then click the Ok button.

**Step 4:** Make sure the Toolbox, the Solution Explorer, and the Properties window are visible and that Auto Hide is turned off for each of these windows. The Visual Studio environment should appear as shown in Figure 2-15.

**Step 5:** Change the Form1 form’s Text property to My First Program, as shown in Figure 2-16.

**Step 6:** The form’s default size is too large for this application, so you need to make it smaller. Use the technique discussed in the previous section to adjust the form’s size with the mouse. The form should appear similar to that shown in Figure 2-17. (Don’t worry about the form’s exact size. Just make it appear similar to Figure 2-17.)

**Step 7:** Now you are ready to add a Button control to the form. Locate the Button tool in the Toolbox and double-click it. A Button control should appear on the form, as shown in Figure 2-18. Move the Button control so it appears approximately in the center of the form, as shown in Figure 2-19.
**Figure 2-15** The Visual Studio environment

**Figure 2-16** The form’s Text property changed to *My First Program*

**Figure 2-17** The form resized
2.2 Creating the GUI for Your First Visual C# Application: The *Hello World* Application

**Figure 2-18** A Button control created on the form

**Figure 2-19** The Button control moved

**Step 8:** Change the value of the Button control’s Text property to *Display Message*. After doing this, notice that the text displayed on the button has changed, as shown in Figure 2-20.

**Figure 2-20** The Button control’s Text property changed

**Step 9:** The Button control isn’t quite large enough to accommodate all of the text that you typed into its Text property, so enlarge the Button control, as shown in Figure 2-21.

**Figure 2-21** The Button control enlarged
Step 10: As discussed in the previous section, a control’s name should reflect the purpose of the control. The Button control that you created in this application will cause a message to be displayed when it is clicked. The name button1 does not convey that purpose, however. Change the Button control’s Name property to messageButton. The Properties window should appear as shown in Figure 2-22.

Figure 2-22 The Button control’s Name property changed to messageButton

Step 11: Click FILE on the Visual Studio menu bar and then click Save All to save the project.

Step 12: You’re only partially finished with the application, but you can run it now to see how the GUI looks on the screen. To run the application, press the F5 key on the keyboard or click the Start Debugging button ( ) on the toolbar. This causes the application to be compiled and executed. You will notice the appearance of the Visual Studio environment change somewhat, and you will see the application’s form appear on the screen as shown in Figure 2-23.

Figure 2-23 The application running

Although the application is running, it is not capable of doing anything other than displaying the form. If you click the Display Message button, nothing will happen. That is because you have not yet written the code that executes when the button is clicked. You will do that in the next tutorial. To end the application, click the standard Windows close button ( ) in the form’s upper-right corner.
CONCEPT: You use the Visual Studio code editor to write an application’s code. Much of the code that you will write in an application will be event handlers. Event handlers respond to specific events that take place while an application is running.

In the previous sections of this chapter, you learned the basics of creating an application’s GUI. An application is more than a user interface, however. If you want your application to perform any meaningful actions, you have to write code. This section introduces you to Visual C# code and shows how to program an application to respond to button clicks.

A file that contains program code is called a source code file. When you start a C# Windows Forms Application project, Visual Studio automatically creates several source code files and adds them to the project. If you look at the Solution Explorer, as shown in Figure 2-24, you will see the names of two source code files: Form1.cs and Program.cs. (C# source code files always end with the .cs extension.)

Figure 2-24 Source code files shown in the Solution Explorer

Here is a brief description of the two files:

- The Program.cs file contains the application’s start-up code, which executes when the application runs. The code in this file performs behind-the-scenes initialization tasks that are necessary to get the application up and running. It is important that you do not modify the contents of this file because doing so could prevent the application from executing.
- The Form1.cs file contains code that is associated with the Form1 form. When you write code that defines some action related to Form1 (such as responding to a button click), you will write the code in this file.

NOTE: You might see additional source code files in the Solution Explorer, other than those shown in Figure 2-24.

The Form1.cs file already contains code that was generated by Visual Studio when the project was created. You can think of this auto-generated code as an outline to which you can add your own code as you develop the application.
Let’s take a look at the code. If you still have the Hello World project open from the previous tutorial, right-click Form1.cs in the Solution Explorer. A pop-up menu will appear, as shown in Figure 2-25. On the pop-up menu, click View Code. The file’s contents will be displayed in the Visual Studio code editor, as shown in Figure 2-26.

**Figure 2-25** Opening Form1.cs in the code editor

![Opening Form1.cs in the code editor](image)

**Figure 2-26** Form1.cs code displayed in the Visual Studio code editor

![Form1.cs code displayed in the Visual Studio code editor](image)
At this point, it’s not necessary for you to understand the meaning of the statements that you see in this code. It will be helpful for you to know how this code is organized, however, because later you will add your own code to this file. C# code is primarily organized in three ways: namespaces, classes, and methods. Here’s a summary:

- A **namespace** is a container that holds classes.
- A class is a container that holds methods (among other things).
- A method is a group of one or more programming statements that performs some operation.

So, C# code is organized as methods, which are contained inside classes, which are contained inside namespaces. With this organizational structure in mind, look at Figure 2-27.

**Figure 2-27** Organization of the Form1.cs code

![Figure 2-27 Organization of the Form1.cs code](image)

The figure shows four different sections of the code, marked with the numbers 1, 2, 3, and 4. Let’s discuss each section of code.

1. Recall from Chapter 1 that C# applications rely heavily on the .NET Framework, which is a collection of classes and other code. The code in the .NET Framework is organized into namespaces. The series of `using` directives that appears at the top of a C# source code file indicate which namespaces in the .NET Framework the program will use.

2. This section of code creates a namespace for the project. The line that reads `namespace Hello_World` marks the beginning of a namespace named `Hello_World`. Notice that the next line contains an opening brace (`{`) and that the last line in the file contains a corresponding closing brace (`}`). All the code that appears between these braces is inside the `Hello_World` namespace.

3. This section of code is a class declaration. The line that reads `public partial class Form1` marks the beginning of the class. The next line contains an opening brace (`{`), and the last line in this section of code contains a corresponding closing brace (`}`). All the code that appears between these braces is inside the class.

4. This section of code is a method. The line that reads `public Form1()` marks the beginning of the method. The next line contains an opening brace (`{`), and the last line in this section of code contains a corresponding closing brace (`}`). The code that appears between these braces is inside the method.

It’s important to point out that code containers, such as namespaces, classes, and methods, use **braces** (`{`) to enclose code. Each opening brace (`{`) must have a corresponding
closing brace (}) at some later point in the program. Figure 2-28 shows how the braces in Form1.cs are paired.

**Figure 2-28** Corresponding braces

![Corresponding braces](image)

Switching between the Code Editor and the Designer

When you open the code editor, it appears in the same part of the screen as the Designer. While developing a Visual C# application, you will often find yourself needing to switch back and forth between the Designer and the code editor. One way to quickly switch between the two windows is to use the tabs shown in Figure 2-29. In the figure, notice that the leftmost tab reads Form1.cs. That is the tab for the code editor. The rightmost tab reads Form1.cs [Design]. That is the tab for the Designer. (The tabs may not always appear in this order.) To switch between the Designer and the code editor, you simply click the tab for the desired window.

**Figure 2-29** Code editor and Designer tabs

![Code editor and Designer tabs](image)

You can also detach the code editor and move it to another part of the screen. This allows you to see the code editor and the Designer at the same time. As shown in Figure 2-30, click the code editor tab and drag it to the desired location on the screen. (If you have multiple monitors connected to your computer, you can even drag the code editor to a different monitor.) To return the code editor to its position within the IDE, right-click the tab for the source code file in the code editor window and select Move to Main Document Group. This is shown in Figure 2-31.
2.3 Introduction to C# Code

Figure 2-30 Detaching the code editor by clicking and dragging

Figure 2-31 Returning the code editor to its docked position

Adding Your Own Code to a Project

Now you are ready to learn how to add your own code to a project. Suppose you have created a project named Code Demo and set up the project’s form with a Button control, as shown in Figure 2-32. The Button control’s name is myButton, and its Text property is set to Click Me!

Suppose you want the application to display the message Thanks for clicking the button! when the user clicks the button. To accomplish that, you need to write a special type of method known as an event handler. An event handler is a method that executes when a specific event takes place while an application is running. In this project you need to write an event handler that will execute when the user clicks the myButton control. To create the event handler, you double-click the myButton control in the Designer. This opens the Form1.cs file in the code editor, as shown in Figure 2-33, with some new code added to it.
Figure 2-32 A form with a Button control

Figure 2-33 The code window opened with event handler code generated
2.3 Introduction to C# Code

When an application is running and the user clicks a control, we say that a **Click event** has occurred on the control. The code that has been added to the Form1.cs file (shown in Figure 2-33) is an event handler that will execute when a Click event occurs on the myButton control. For now you do not need to understand all parts of the event handler code. At this point you need to understand only the following concepts:

- As shown in Figure 2-34, the event handler’s name is **myButton_Click**. The "myButton" portion of the name indicates that the event handler is associated with the **myButton** control, and the "Click" portion of the name indicates that the event handler responds to Click events. This is the typical naming convention that Visual Studio uses when it generates event handler code. When you see the name **myButton_Click**, you understand that it is an event handler that executes when a Click event occurs on the **myButton** control.

![Figure 2-34 A closer look at the event handler code](image)

Now you know how to create an empty Click event handler for a Button control. But what code do you write inside the event handler? In this example we write code that displays the message **Thanks for clicking the button!** in a message box, which is a small pop-up window.

**Message Boxes**

A **message box** is a small window, sometimes referred to as a **dialog box**, that displays a message. Figure 2-35 shows an example of a message box displaying the message **Thanks for clicking the button!** Notice that the message box also has an **OK** button. When the user clicks the **OK** button, the message box closes.

![Figure 2-35 A message box](image)
The .NET Framework provides a method named `MessageBox.Show` that you can use in Visual C# to display a message box. If you want to execute the `MessageBox.Show` method, you write a statement known as a method call. (Programmers refer to the act of executing a method as calling the method.) The following statement shows an example of how you would call the `MessageBox.Show` method to display the message box shown in Figure 2-35:

```csharp
MessageBox.Show("Thanks for clicking the button!");
```

When you call the `MessageBox.Show` method, you write a string of characters inside the parentheses. (In programming we use the term string to mean string of characters.) The string that is written inside the parentheses will be displayed in the message box. In this example the string "Thanks for clicking the button!" is written inside the parentheses.

Notice that the string is enclosed in double quotation marks in the code. When the message is displayed (as shown in Figure 2-35), however, the double quotation marks do not appear. The double quotation marks are required in the code to indicate the beginning and the end of the string.

Also notice that a semicolon appears at the end of the statement. This is required by C# syntax. Just as a period marks the end of a sentence, a semicolon marks the end of a programming statement in C#.

Getting back to our Code Demo example project, Figure 2-36 shows how you can call the `MessageBox.Show` method from the `myButton_Click` event handler. After typing the statement as shown in the figure, you can press the `F5` key on the keyboard, or click the Start Debugging button on the toolbar to compile and run the application. When the application runs, it will display the form shown on the left in Figure 2-37. When you click the button, the message box shown on the right in the figure will appear. You can click the OK button on the message box to close it.

**Figure 2-36** Event handler code for displaying a message box

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Code_Demo
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void myButton_Click(object sender, EventArgs e)
        {
            MessageBox.Show("Thanks for clicking the button!");
        }
    }
}
```
Programs almost always work with data of some type. For example, the code shown in Figure 2-36 uses the following string when it calls the `MessageBox.Show` method:

"Thanks for clicking the button!"

This string is the data that is displayed by the program. When a piece of data is written into a program’s code, it’s called a literal (because the data is literally written into the program). When a string is written into a program’s code, it’s called a string literal. In C#, string literals must be enclosed in double quotation marks.

NOTE: Programmers sometimes say that literals are values that are hard coded into a program because the value of a literal cannot change while the program is running.

Multiple Buttons with Event Handlers

The `Code Demo` project previously shown has only one button with a Click event handler. Many of the applications that you will develop will have multiple buttons, each with its own Click event handler. For example, the form shown in Figure 2-38 has three Button controls. As shown in the figure, the controls are named `firstButton`, `secondButton`, and `thirdButton`. 
To create Click event handlers for the buttons, you simply double-click each Button control in the *Designer* and an empty event handler will be created in the form’s source code file. The names of the Click event handlers will be *firstButton_Click*, *secondButton_Click*, and *thirdButton_Click*. Figure 2-39 shows an example of the form’s source code after the three event handlers have been created and a *MessageBox.Show* statement has been added to each one.

**Figure 2-39** Source code with three Click event handlers

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Multiple.Buttons
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void firstButton_Click(object sender, EventArgs e)
        {
            MessageBox.Show("You clicked the first button.");
        }

        private void secondButton_Click(object sender, EventArgs e)
        {
            MessageBox.Show("You clicked the second button.");
        }

        private void thirdButton_Click(object sender, EventArgs e)
        {
            MessageBox.Show("You clicked the third button.");
        }
    }
}
```

**Design Time and Run Time**

When you have a project open in Visual Studio, the time during which you build the GUI and write the application’s code is referred to as *design time*. During design time, you can use the *Designer* and the *Toolbox* to place controls on the form, use the *Properties* window to set property values, use the code editor to write code, and so forth. This is the phase during which you create or modify the application.

When you are ready to run a project that you have open in Visual Studio, you press the *F5* key on the keyboard or click the *Start Debugging* button (¶) on the toolbar. The project will be compiled, and if there were no errors, it will be executed. The time during which an application is executing is referred to as *run time*. During run time, you can interact with the running application, but you cannot use the *Designer*, the *Toolbox*, the *Properties* window, the code editor, or parts of Visual Studio to make changes to it.
NOTE: In computing literature and on the Web, you will see the term run time also spelled as runtime or run-time. All these variations typically mean the same thing.

Checkpoint

2.15 A file that contains program code is known as what type of file?
2.16 What must you do if you want your application to perform any meaningful actions?
2.17 What does the Program.cs file contain?
2.18 What does the Form1.cs file contain?
2.19 How is C# code organized?
2.20 What is a namespace?
2.21 What characters do code containers, such as namespaces, classes, and methods, use to enclose code?
2.22 How do you switch between the Designer and the code editor?
2.23 How do you create an event handler for a button?
2.24 What is a Click event?
2.25 What method do you use in Visual C# to display a message box?
2.26 What is a literal?
2.27 What are string literals enclosed in?
2.28 How do you run a project that you have open in Visual Studio?

2.4 Writing Code for the Hello World Application

Now you know everything necessary to complete the Hello World project. In Tutorial 2-2 you will open the project and add a Click event handler for the messageButton control. The event handler will call the MessageBox.Show method to display a message box with the message Hello World.

Tutorial 2-2:
Writing Code for the Hello World Application

Step 1: If Visual Studio (or Visual Studio Express) is not already running, start it. Open the Hello World project that you started in Tutorial 2-1.

Step 2: Make sure the Form1 form is visible in the Designer, as shown in Figure 2-40. If it is not, right-click Form1.cs in the Solution Explorer and then select View Designer from the pop-up menu.

Step 3: In the Designer, double-click the messageButton control. This should cause the code editor to appear as shown in Figure 2-41. Notice that an empty event handler named messageButton_Click has been created.
Figure 2-40 The *Hello World* project loaded with Form1 shown in the Designer

![Diagram of the Designer](image1)

Figure 2-41 Code editor with an empty event handler

![Diagram of the Code editor](image2)
Step 4: Inside the messageButton_Click event handler, type the following statement exactly as it is shown:

```
MessageBox.Show("Hello World");
```

Don’t forget to type the semicolon at the end of the statement! When you have finished, the code window should look like Figure 2-42.

Figure 2-42 Statement written inside the event handler

Step 5: Save the project.

Step 6: Press the F5 key on the keyboard, or click the Start Debugging button (F5) on the toolbar to compile and run the application.

NOTE: If you typed the statement correctly inside the messageButton_Click event handler (in Step 4), the application should run. If you did not type the statement correctly, however, a window will appear reporting build errors. If that happens, click the No button in the window and then correct the statement so it appears exactly as shown in Figure 2-42.

When the application runs, it will display the form shown on the left in Figure 2-43. When you click the Display Message button, the message box shown on the right in the figure will appear. You can click the OK button on the message box to close it.
Label Controls

CONCEPT: A label control displays text on a form. Label controls have various properties that affect the control’s appearance. Label controls can be used to display unchanging text, or program output.

When you want to display text on a form, you use a Label control. Figure 2-44 shows an example of a form with two Label controls. Once you have placed a Label control on a form, you set its Text property to the text that you want to display. For example, in Figure 2-44, the upper Label control’s Text property is set to Number of Hours Worked, and the lower Label control’s Text property is set to Hourly Pay Rate.

You’ll find the Label control tool in the Common Controls group of the Toolbox, as shown in Figure 2-45. To create a Label control on a form, you double-click the Label control tool in the Toolbox. As shown in Figure 2-45, a Label control will be created on the form. (Alternatively, you can click and drag the Label control tool from the Toolbox onto the form.) Notice that a bounding box appears around the Label control in the figure. This indicates that the control is currently selected.

When you create Label controls, they are automatically given default names such as label1, label2, and so forth. A Label control’s Text property is initially set to the same value as the Label control’s name. So, a Label control will display its own name when it is created, as shown by the example in Figure 2-45. When a Label control is selected in the Designer, you can use the Properties window to change its Text property. Figure 2-46 shows a Label control after its Text property has been changed to Programming in Visual C# is fun!
You can also use the Properties window to change a Label control’s name. It’s always a good idea to change a control’s name to something that is more meaningful than the default name that Visual Studio gives it.

**The Font Property**

If you want to change the appearance of a Label control’s text, you can change the control’s Font property. The Font property allows you to set the font, font style, and size of the control’s text. When you select the Font property in the Properties window, you will notice that an ellipses button (…) appears next to the property’s value, as shown in Figure 2-47. When you click the ellipses button, the Font dialog box appears, as shown...
in Figure 2-48. Select a font, font style, and size, and click OK. The text displayed by the control will be updated with the selected attributes. For example, Figure 2-49 shows a Label control with the following Font property attributes:

- **Font**: Lucida Handwriting
- **Font Style**: Italic
- **Size**: 10 point

**Figure 2-48** The Font dialog box

**Figure 2-49** A label’s appearance with altered font attributes

### The BorderStyle Property

Label controls have a **BorderStyle property** that allows you to display a border around the control’s text. The BorderStyle property may have one of three values: None, FixedSingle, or Fixed3D. The property is set to None by default, which means that no border will appear around the control’s text. If the BorderStyle property is set to FixedSingle, the control’s text will be outlined with a thin border. If the BorderStyle property is set to Fixed3D, the control’s text will have a recessed 3D appearance. Figure 2-50 shows an example of Label controls with each BorderStyle setting.

**Figure 2-50** BorderStyle examples
To change the BorderStyle property, select it in the Properties window and then click the down-arrow button (▼) that appears next to the property’s value. As shown in Figure 2-51, a drop-down list will appear containing the three possible values for this property. Select the desired value and the control’s text will be updated.

Figure 2-51  BorderStyle selections

The AutoSize Property

Label controls have an AutoSize property that controls the way they can be resized. The AutoSize property is a Boolean property, which means that it can be set to one of two possible values: True or False. By default, a Label control’s AutoSize property is set to True, which means that the control automatically resizes itself to accommodate the size of the text it displays. For example, look at the three Label controls in Figure 2-52. Each of the controls displays different amounts of text at different font sizes. Because each control’s BorderStyle property is set to FixedSingle, you can see that each control is just large enough to accommodate its text.

Figure 2-52  Label controls with AutoSize set to True

When a Label control’s AutoSize property is set to True, you cannot manually change the size of the control by clicking and dragging its bounding box. If you want to manually change the size of a Label control, you have to set its AutoSize property to False. When AutoSize is set to False, sizing handles will appear around the control, allowing you to click and drag the bounding box to resize the control. Figure 2-53 shows an example. In the figure, the Label control has been resized so it is much larger than the text it displays.
When a Label control’s AutoSize property is set to True, the label’s text will always appear on one line. When the AutoSize property is set to False, the label’s text will wrap across multiple lines if it is too long to fit on one line.

The TextAlign Property

When you set a Label control’s AutoSize property to False and then manually resize the control, it sometimes becomes necessary to change the way the label’s text is aligned. By default, a label’s text is aligned with the top and left edges of the label’s bounding box. For example, look at the label shown in Figure 2-53. Notice how the text is positioned in the label’s upper-left corner.

What if we want the text to be aligned differently within the label? For example, what if we want the text to be centered in the label or positioned in the lower-right corner? We can change the text’s alignment in the label with the TextAlign property. The TextAlign property may be set to any of the following values: TopLeft, TopCenter, TopRight, MiddleLeft, MiddleCenter, MiddleRight, BottomLeft, BottomCenter, or BottomRight. Figure 2-54 shows nine Label controls, each with a different TextAlign value.

To change the TextAlign property, select it in the Properties window and then click the down-arrow button (▼) that appears next to its value. This causes a dialog box with nine buttons, as shown in the left image in Figure 2-55, to appear. As shown in the right image in the figure, the nine buttons represent the valid settings of the TextAlign property.

Using Code to Display Output in a Label Control

In addition to displaying unchanging text on a form, Label controls are also useful for displaying output while an application is running. For example, suppose you are creating an application that performs a calculation and you want to display the result of the calcula-
2.5 Label Controls

When outputting results at a specific location on the form. Using a Label control to display the output would be an ideal solution. Here are the general steps that you would follow:

**Step 1:** While creating the application’s GUI, you place a Label control on the form at the location where you want the result to be displayed. Then, in the Properties window, you erase the contents of the Label control’s Text property. Because the control’s Text property is empty, the control will not initially display anything when the application runs.

**Step 2:** In the application’s code, you write the necessary statements to perform the calculation and then you store the result of the calculation in the Label control’s Text property. This causes the result to be displayed on the form in the Label control.

**NOTE:** We do not discuss calculations until Chapter 3, so in this chapter we look at examples that display nonmathematical data as output in Label controls.

In code, you use an assignment statement to store a value in a control’s property. For example, suppose you have created a Label control and named it `outputLabel`. The following assignment statement stores the string “Thank you very much” in the control’s Text property.

```
outputLabel.Text = "Thank you very much";
```

The equal sign (=) is known as the assignment operator. It assigns the value that appears on its right side to the item that appears on its left side. In this example, the item on the left side of the assignment operator is the expression `outputLabel.Text`. This is simply the `outputLabel` control’s Text property. The value on the right side of the assignment operator is the string “Thank you very much”. When this statement executes, the string “Thank you very much” is assigned to the `outputLabel` control’s Text property. When this statement executes, the text *Thank you very much* is displayed in the Label control.

**WARNING!** When writing assignment statements, remember that the item receiving the value must be on the left side of the = operator. The following statement, for example, is wrong and will cause an error when you compile the program:

```
"Thank you very much" = outputLabel; ← ERROR!
```

**NOTE:** The standard notation for referring to a control’s property in code is:

```
ControlName.PropertyName
```
Let’s look at an example application that uses a Label control to display output. Make sure you have downloaded the student sample programs from the book’s companion Web site (at www.pearsonhighered.com/gaddis). In the Chap02 folder, you will find a project named Presidential Trivia. The purpose of the application is to display a trivia question about a former U.S. president. When the user clicks a button, the answer to the trivia question is displayed on the form. The project’s form appears as shown in Figure 2-56.

**Figure 2-56** Presidential Trivia form

As shown in the figure, the form has the three controls:

- A Label control named `questionLabel`. This label displays the trivia question.
- A Label control named `answerLabel`. This label initially appears empty, but will be used to display the answer to the trivia question.
- A Button control named `showAnswerButton`. When the user clicks this button, the answer to the trivia question is displayed.

Table 2-2 lists the property settings for each control of which you should take note.

<table>
<thead>
<tr>
<th>Control Name</th>
<th>Control Type</th>
<th>Property Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>questionLabel</code></td>
<td>Label</td>
<td>AutoSize: False</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Font</strong>: Microsoft Sans Serif (Style: Regular, Size: 10 point)</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Text</strong>: What former U.S. president is known for going on an African safari?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TextAlign: MiddleCenter</td>
</tr>
<tr>
<td><code>answerLabel</code></td>
<td>Label</td>
<td>AutoSize: False</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Font</strong>: Microsoft Sans Serif (Style: Bold, Size: 10 point)</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Text</strong>: (The contents of the Text property have been erased.)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TextAlign: MiddleCenter</td>
</tr>
<tr>
<td><code>showAnswerButton</code></td>
<td>Button</td>
<td>Size: 110, 23</td>
</tr>
</tbody>
</table>

If we open the Form1.cs file in the code editor, we see the code shown in Figure 2-57. (To open the file in the code window, right-click Form1.cs in the Solution Explorer and then select View Code.) Notice the method named `showAnswerButton_Click`.  

This is the Click event handler for the `showAnswerButton` control. It contains the following statement:

```
answerLabel.Text = "Theodore Roosevelt";
```

When this statement executes, it assigns the string "Theodore Roosevelt" to the `answerLabel` control’s Text property. As a result, Theodore Roosevelt is displayed in the label control.

When you run the application, the form appears as shown on the left in Figure 2-58. Click the `Show the Answer` button and the answer to the trivia question appears as shown on the right in the figure.

**Figure 2-57** Form1.cs code

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Presidential_Trivia
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void showAnswerButton_Click(object sender, EventArgs e)
        {
            answerLabel.Text = "Theodore Roosevelt";
        }
    }
}
```

**The Text Property Accepts Strings Only**

It is important to point out that the Label control’s Text property can accept strings only. You cannot assign a number to the Text property. For example, let’s assume that an
application has a Label control named `resultLabel`. The following statement will cause an error because it is attempting to store the number 5 in the `resultLabel` control’s Text property:

```csharp
resultLabel.Text = 5; ← ERROR!
```

This does not mean that you cannot display a number in a label, however. If you put quotation marks around the number, it becomes a string. The following statement will work:

```csharp
resultLabel.Text = "5";
```

### Clearing a Label

In code, if you want to clear the text that is displayed in a Label control, simply assign an empty string ("") to the control’s Text property, as shown here:

```csharp
answerLabel.Text = "";
```

In Tutorial 2-3 you will work with some of the Label control properties that we have discussed in this section.

---

**Tutorial 2-3: Creating the Language Translator Application**

In this tutorial you will create an application that displays the phrase “Good Morning” in different languages. The form will have three buttons: one for Italian, one for Spanish, and one for German. When the user clicks any of these buttons, the translated phrase will appear in a Label control.

**Step 1:** Start Visual Studio (or Visual Studio Express) and begin a new Windows Forms Application project named `Language Translator`.

**Step 2:** Set up the application’s form as shown in Figure 2-59. Notice that the form’s Text property is set to `Language Translator`. The form has two Label controls and three Button controls. The names of the controls are shown in the figure. As you place each of the controls on the form, refer to Table 2-3 for the relevant property settings.

**Figure 2-59** The Language Translator form

---

**Step 3:** Once you have the form and its controls set up, you can create the Click event handlers for the Button controls. In the `Designer`, double-click the `italianButton` control. This will open the code editor, and you will see an empty event handler named `italianButton_Click`. Write the following statement inside the event handler:

```csharp
translationLabel.Text = "Buongiorno";
```
Table 2-3 Control property settings

<table>
<thead>
<tr>
<th>Control Name</th>
<th>Control Type</th>
<th>Property Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>instructionLabel</td>
<td>Label</td>
<td>Text: <em>Select a language and I will say Good Morning.</em></td>
</tr>
<tr>
<td>translationLabel</td>
<td>Label</td>
<td>AutoSize: False</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BorderStyle: FixedSingle</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Font: Microsoft Sans Serif (Style: Bold, Size: 10 point)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Text: <em>(The contents of the Text property have been erased.)</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TextAlign: MiddleCenter</td>
</tr>
<tr>
<td>italianButton</td>
<td>Button</td>
<td>Text: <em>Italian</em></td>
</tr>
<tr>
<td>spanishButton</td>
<td>Button</td>
<td>Text: <em>Spanish</em></td>
</tr>
<tr>
<td>germanButton</td>
<td>Button</td>
<td>Text: <em>German</em></td>
</tr>
</tbody>
</table>

**Step 4:** Switch your view back to the Designer and double-click the **spanishButton** control. In the code editor you will see an empty event handler named **spanishButton_Click**. Write the following statement inside the event handler:

```
translationLabel.Text = “Buenos Dias”;
```

**Step 5:** Switch your view back to the Designer and double-click the **germanButton** control. In the code editor you will see an empty event handler named **germanButton_Click**. Write the following statement inside the event handler:

```
translationLabel.Text = “Guten Morgen”;
```

**Step 6:** The form’s code should now appear as shown in Program 2-1. Note that the line numbers are not part of the code. The line numbers are shown so that you and your instructor can more easily refer to different parts of the program. The lines that appear in boldface are the ones that you typed. Make sure the code you typed appears exactly as shown here. (Don’t forget the semicolons!)

**Program 2-1** Completed Form1 code for the Language Translator application

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Language_Translator
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```
private void italianButton_Click(object sender, EventArgs e)
{
    translationLabel.Text = "Buongiorno";
}

private void spanishButton_Click(object sender, EventArgs e)
{
    translationLabel.Text = "Buenos Dias";
}

private void germanButton_Click(object sender, EventArgs e)
{
    translationLabel.Text = "Guten Morgen";
}

Step 7: Save the project. Then, press the \[F5\] key on the keyboard or click the Start Debugging button (\(\square\)) on the toolbar to compile and run the application.

**NOTE:** If you typed the statements correctly inside the event handlers, the application should run. If you did not type the statements correctly inside the event handlers, a window will appear reporting build errors. If that happens, click the No button in the window and then correct the code so it appears exactly as previously shown.

Figure 2-60 shows the application’s form when it starts running and after you have clicked each of the Button controls. After you have tested each button, close the application’s form.

**Figure 2-60** The Language Translator application running
Checkpoint

2.29 In which group of the Toolbox can you find the Label control tool?
2.30 Once you have placed a Label control on a form, which property do you use to set the text that you want to display?
2.31 What property can you use to change the appearance of a Label control’s text?
2.32 What is the default value of a label’s BorderStyle property?
2.33 How do you change the BorderStyle property of a control in the Properties window?
2.34 What property determines whether a label can be resized?
2.35 What property determines the way text is aligned in a Label control?
2.36 How can you use a Label control to display output while a program is running?
2.37 What happens if you assign an empty string to a control’s Text property in code?

2.6 Making Sense of IntelliSense

CONCEPT: As you type code in the Visual Studio code editor, IntelliSense boxes pop up to assist you. You can use the IntelliSense boxes to automatically complete some programming statements after typing only the first few characters.

IntelliSense is a feature of Visual Studio that provides automatic code completion as you write programming statements. Once you learn how to use IntelliSense, it helps you write code faster. If you’ve worked through the previous tutorials in this chapter, you’ve already encountered IntelliSense. For example, in Step 3 of Tutorial 2-3, you were instructed to write the following statement in the italianButton_Click event handler:

```csharp
translationLabel.Text = "Buongiorno";
```

Did you notice that as soon as you started typing the statement, a box popped up on the screen? This is known as an IntelliSense list box. The contents of the list box changes as you type. Figure 2-61 shows the IntelliSense list box after you have typed the characters tra.

Figure 2-61 IntelliSense list box displayed
The IntelliSense system is anticipating what you are about to type, and as you type characters, the content of the list box is reduced. The list box shown in Figure 2-61 shows all the names starting with tra that might be a candidate for the statement you are typing. Notice that translationLabel is selected in the list box. With that item selected, you can press the Tab key on the keyboard, and the tra that you previously typed becomes translationLabel.

Next, when you type a period, an IntelliSense list pops up showing every property and method belonging to the translationLabel control. Type te and the Text property becomes selected, as shown in Figure 2-62. When you press the Tab key to select the Text property, your statement automatically becomes translationLabel.Text. At this point, you can continue typing until you have completed the statement.

Now that you have an idea of how IntelliSense works, you are encouraged to experiment with it as you write code in future projects. With a little practice, it will become intuitive.

2.7 PictureBox Controls

CONCEPT: A PictureBox control displays a graphic image on a form. PictureBox controls have properties for controlling the way the image is displayed. A PictureBox control can have a Click event handler that responds when the user clicks the control at run time.

You can use a PictureBox control to display a graphic image on a form. A PictureBox control can display images that have been saved in the bitmap, GIF, JPEG, metafile, or icon graphics formats.

In the Toolbox, the PictureBox tool is located in the Common Controls group. When you double-click the tool, an empty PictureBox control is created on the form, as shown in Figure 2-63. Although the control does not yet display an image, it has a bounding box that shows its size and location, as well as sizing handles. When you create PictureBox controls, they are automatically given default names such as pictureBox1, pictureBox2, and so forth. You should always change the default name to something more meaningful.
Once you have created a PictureBox control, you use its **Image property** to specify the image that it will display. Follow these steps:

**Step 1:** Click the Image property in the *Properties* window. An ellipses button (…) will appear, as shown on the left in Figure 2-64.

**Step 2:** Click the ellipses button and the *Select Resource* window, shown on the right in Figure 2-64, will appear.

**Step 3:** In the *Select Resource* window, click the *Import* button. An *Open* dialog box will appear. Use the dialog box to locate and select the image file that you want to display.

**Step 4:** After you select an image file, you will see its contents displayed in the *Select Resource* window. This indicates that the image has been imported into the project. Figure 2-65 shows an example of the *Select Resource* window after we have selected and imported an image.

**Step 5:** Click the OK button in the *Select Resource* window, and the selected image will appear in the PictureBox control. Figure 2-66 shows an example. Depending on the size of the image, you might see only part of it displayed. This is the case in Figure 2-66 because the image is larger than the PictureBox control. Your next step is to set the SizeMode property and adjust the size of the control.
The SizeMode Property

The PictureBox control’s SizeMode property specifies how the control’s image is to be displayed. It can be set to one of the following values:

- **Normal**
  Normal is the default value. The image will be positioned in the upper-left corner of the PictureBox control. If the image is too big to fit in the PictureBox control, it will be clipped.

- **StretchImage**
  StretchImage resizes the image both horizontally and vertically to fit in the PictureBox control. If the image is resized more in one direction than the other, it will appear stretched.

- **AutoSize**
  With AutoSize, the PictureBox control is automatically resized to fit the size of the image.
• **CenterImage**
  CenterImage centers the image in the PictureBox control without resizing it.

• **Zoom**
  Zoom uniformly resizes the image to fit in the PictureBox without losing its original aspect ratio. *(Aspect ratio is the image’s width to height ratio.)* This causes the image to be resized without appearing stretched.

Figure 2-67 shows an example of an image displayed in a PictureBox control. The control’s SizeMode is set to Zoom, so it can be resized without appearing stretched.

**Figure 2-67** An image resized with SizeMode set to Zoom

---

**NOTE:** PictureBox controls also have a BorderStyle property that works just like a Label control’s BorderStyle property.

---

### Creating Clickable Images

Buttons aren’t the only controls that can respond to Click events. PictureBox controls can, too. That means an application can display an image and perform some action when the user clicks the image.

To make an image clickable, you simply have to create a Click event handler for the PictureBox control that displays the image. You create a Click event handler for a PictureBox control in the same way that you create a Click event handler for a Button control:

- You double-click the PictureBox control in the *Designer*. This creates an empty Click event handler in the form’s source code file.
- In the code editor you write statements inside the event handler that you want to execute when the image is clicked.

As an example, look at the *Cat* project that is in the *Chap02* folder of the Student Sample Programs that accompany this textbook. Figure 2-68 shows the application’s form. The PictureBox control’s name is `catPictureBox`. Its image is the *Cat.jpg* file, which is also found in the *Chap02* folder of the Student Sample Programs. The SizeMode property is set to Zoom, and the BorderStyle property is set to FixedSingle.

Open the Form1.cs file in the code editor and you will see that we have already created a Click event handler for the `catPictureBox` control, as shown in Figure 2-69. If you run the application and click the PictureBox, a message box will appear displaying the string *Meow.*
Tutorial 2-4: Creating the Flags Application

In this tutorial you will create an application that displays the flags of Finland, France, and Germany in PictureBox controls. When the user clicks any of these PictureBoxes, the name of that flag’s country will appear in a Label control.

Step 1: Start Visual Studio (or Visual Studio Express) and begin a new Windows Forms Application project named Flags.
Step 2: Set up the application’s form as shown in Figure 2-70. Notice that the form’s Text property is set to Flags. The names of the controls are shown in the figure. Refer to Table 2-4 for each control’s relevant property settings.

**Figure 2-70** The Flags form

![The Flags form](image)

**Table 2-4** Control property settings

<table>
<thead>
<tr>
<th>Control Name</th>
<th>Control Type</th>
<th>Property Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>instructionLabel</td>
<td>Label</td>
<td>Text: Click a flag to see the name of the country.</td>
</tr>
<tr>
<td>finlandPictureBox</td>
<td>PictureBox</td>
<td>Image: Select and import the Finland.bmp file from the Chap02 folder of the Student Sample Programs. BorderStyle: FixedSingle SizeMode: AutoSize</td>
</tr>
<tr>
<td>francePictureBox</td>
<td>PictureBox</td>
<td>Image: Select and import the France.bmp file from the Chap02 folder of the Student Sample Programs. BorderStyle: FixedSingle SizeMode: AutoSize</td>
</tr>
<tr>
<td>germanyPictureBox</td>
<td>PictureBox</td>
<td>Image: Select and import the Germany.bmp file from the Chap02 folder of the Student Sample Programs. BorderStyle: FixedSingle SizeMode: AutoSize</td>
</tr>
<tr>
<td>countryLabel</td>
<td>Label</td>
<td>AutoSize: False BorderStyle: FixedSingle Font: Microsoft Sans Serif (Style: Bold, Size: 10 point) Text: (The contents of the Text property have been erased.) TextAlign: MiddleCenter</td>
</tr>
</tbody>
</table>

Step 3: Once you have the form and its controls set up, you can create the Click event handlers for the PictureBox controls. In the Designer, double-click the finlandPictureBox control. This will open the code editor, and you will see an empty event handler named finlandPictureBox_Click. Write the following statement inside the event handler:

countryLabel.Text = "Finland";
Step 4: Switch your view back to the Designer and double-click the francePictureBox control. This will open the code editor, and you will see an empty event handler named francePictureBox_Click. Write the following statement inside the event handler:

countryLabel.Text = "France";

Step 5: Switch your view back to the Designer and double-click the germanyPictureBox control. This will open the code editor, and you will see an empty event handler named germanyPictureBox_Click. Write the following statement inside the event handler:

countryLabel.Text = "Germany";

Step 6: The form’s code should now appear as shown in Program 2-2. As was mentioned in the previous tutorial, the line numbers are shown for reference only, and are not part of the code. The lines that appear in boldface are the ones that you typed. Make sure the code you typed appears exactly as shown here. (Don’t forget the semicolons!)

**Program 2-2  Completed Form1 code for the Flags application**

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Flags
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void finlandPictureBox_Click(object sender, EventArgs e)
        {
            countryLabel.Text = "Finland";
        }

        private void francePictureBox_Click(object sender, EventArgs e)
        {
            countryLabel.Text = "France";
        }

        private void germanyPictureBox_Click(object sender, EventArgs e)
        {
            countryLabel.Text = "Germany";
        }
    }
}
```
Step 7: Save the project. Then, press the **F5** key on the keyboard, or click the Start Debugging button ( ) on the toolbar to compile and run the application.

**NOTE:** If you typed the statements correctly inside the event handlers, the application should run. If you did not type the statements inside the event handlers correctly, a window will appear reporting build errors. If that happens, click the No button in the window, then correct the code so it appears exactly as previously shown.

Figure 2-71 shows the application’s form when it starts running and then after you have clicked each of the PictureBox controls. After you have clicked each flag to make sure the application works correctly, close the form.

![Figure 2-71 The Flags application running](image)

The Visible Property

Most controls have a **visible property** that determines whether the control is visible on the form at run time. The Visible property is a Boolean property, which means it can be set only to the values True or False. If a control’s Visible property is set to True, the control will be visible on the form at run time. If a control’s Visible property is set to False, however, the control will not be visible at run time. By default, the Visible property is set to True.

When you use the Properties window to change a control’s Visible property at design time, the control will still be visible in the Designer. When you run the application, however, the control will not be visible on the form. For example, the image on the left in Figure 2-72 shows a form in the Designer. The PictureBox control’s Visible property is set to False, but the control can still be seen in the Designer. The image on the right shows the form while the application is running. At run time, the control is not visible.

A control’s Visible property can also be modified in code by an assignment statement, which makes it possible to hide or display a control while the application is running. For example, the PictureBox control shown in Figure 2-72 is named spiderPictureBox. The following statement sets the control’s Visible property to True:

```csharp
spiderPictureBox.Visible = true;
```
When this statement executes, the `spiderPictureBox` control will become visible. Likewise, the following statement sets the control’s Visible property to `false`:

```csharp
spiderPictureBox.Visible = false;
```

When this statement executes, the `spiderPictureBox` control will become invisible.

**NOTE:** When you write the values `true` and `false` in code, as shown in the previous assignment statement, they must be written in all lowercase letters. The words `true` and `false` are C# keywords, and an error will occur if you don’t write them in lowercase. However, when you use the Properties window to set a Boolean property, such as Visible, the values True and False will be shown with an initial capital. Try not to let this inconsistency confuse you!

In Tutorial 2-5 you will create an application that uses the Visible property of two PictureBox controls to simulate a card being flipped over.

**Tutorial 2-5:**
Creating the Card Flip Application

In this tutorial you will create an application that simulates a card being flipped over. When the application runs, it will display the form shown on the left in Figure 2-73. The form initially displays the back of a poker card. When the user clicks the `Show the Card Face` button, the card will be flipped over to show its face, as shown in the form on the right. When the user clicks the `Show the Card Back` button, the card is flipped back over to show its back.
The simulation of the card being flipped will be accomplished using the following logic:

- When the user clicks the *Show the Card Face* button, the PictureBox showing the card’s back will be made invisible and the PictureBox showing the card’s face will be made visible.
- When the user clicks the *Show the Card Back* button, the PictureBox showing the card’s face will be made invisible and the PictureBox showing the card’s back will be made visible.

**Figure 2-73** The Card Flip application

**Step 1:** Start Visual Studio (or Visual Studio Express) and begin a new Windows Forms Application project named *Card Flip*.

**Step 2:** Set up the application’s form as shown in Figure 2-74. Notice that the form’s Text property is set to *Card Flip*. The names of the controls are shown in the figure. Use the Properties window to make the property settings shown in Table 2-5. (In particular, note that the *cardBackPictureBox* control’s Visible property is set to True, and the *cardFacePictureBox* control’s Visible property is set to False.)

**Figure 2-74** The Card Flip form
**Table 2-5** Control property settings

<table>
<thead>
<tr>
<th>Control Name</th>
<th>Control Type</th>
<th>Property Settings</th>
</tr>
</thead>
</table>
| cardBackPictureBox   | PictureBox   | **Image:** Select and import the Backface_Blue.jpg file from the Chap02 folder of the Student Sample Programs.  
Size: 100, 140  
SizeMode: Zoom  
Visible: True |
| cardFacePictureBox   | PictureBox   | **Image:** Select and import the Ace_Spades.jpg file from the Chap02 folder of the Student Sample Programs.  
Size: 100, 140  
SizeMode: Zoom  
Visible: False |
| showBackButton       | Button       | **Text:** Show the Card Back  
(Manually resize the button to accommodate the text, as shown in Figure 2-74.) |
| showFaceButton       | Button       | **Text:** Show the Card Face (Manually resize the button to accommodate the text, as shown in Figure 2-74.) |

**Step 3:** Once you have the form and its controls set up, you can create the Click event handlers for the Button controls. In the Designer, double-click the showBackButton control. This will open the code editor, and you will see an empty event handler named showBackButton_Click. Write the following statements inside the event handler:

```csharp
cardBackPictureBox.Visible = true;
cardFacePictureBox.Visible = false;
```

**Step 4:** Switch your view back to the Designer and double-click the showFaceButton control. This will open the code editor, and you will see an empty event handler named showFaceButton_Click. Write the following statements inside the event handler:

```csharp
cardBackPictureBox.Visible = false;
cardFacePictureBox.Visible = true;
```

**Step 5:** The form’s code should now appear as shown in Program 2-3. Remember, the line numbers are shown for reference only and are not part of the code. The lines that appear in boldface are the ones that you typed. Make sure the code you typed appears exactly as shown here. (Don’t forget the semicolons!)

**Program 2-3** Completed Form1 code for the Card Flip application

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```
using System.Windows.Forms;

namespace Card_Flip
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void showBackButton_Click(object sender, EventArgs e)
        {
            cardBackPictureBox.Visible = true;
            cardFacePictureBox.Visible = false;
        }

        private void showFaceButton_Click(object sender, EventArgs e)
        {
            cardBackPictureBox.Visible = false;
            cardFacePictureBox.Visible = true;
        }
    }
}

Step 6: Save the project. Then, press the [F5] key on the keyboard, or click the Start Debugging button ( ) on the toolbar to compile and run the application.

Test the application by clicking the buttons. When you click the Show the Card Face button you should see the card’s face (and the back of the card should be invisible). When you click the Show the Card Back button you should see the card’s back (and the card’s face should be invisible). When you are finished, close the application.

NOTE: If you typed the statements correctly inside the event handlers, the application should run. If you did not type the statements inside the event handlers correctly, a window will appear reporting build errors. If that happens, click the No button in the window and then correct the code so it appears exactly as previously shown.

NOTE: In addition to PictureBoxes, many other types of controls have a Visible property. For example, you can make a Label control visible or invisible by setting the value of its Visible property.

Sequential Execution of Statements

In Tutorial 2-5, the event handlers that you created each contained more than one statement. For example, here is the showBackButton_Click method:

private void showBackButton_Click(object sender, EventArgs e)
{
    cardBackPictureBox.Visible = true;
    cardFacePictureBox.Visible = false;
}
This method has two assignment statements. When the method executes, the statements in the method execute in the order that they appear, from the beginning of the method to the end of the method. This statement executes first:

    cardBackPictureBox.Visible = true;

Then this statement executes:

    cardFacePictureBox.Visible = false;

When the application is running, however, you can't really tell that the statements are executing in this order simply by watching the action take place on the screen. When you click the showBackButton control, the Click event handler executes so quickly that it appears as though both statements execute simultaneously. It’s important for you to understand, however, that the statements execute one at a time, in the order that they appear in the method.

In this particular method, it doesn’t really matter which assignment statement is written first. If we reverse the order of the statements, we will not be able to see the difference on the screen because the application executes so quickly. In most applications, however, the order in which you write the statements in the event handlers is critically important. In Chapter 3 you will start writing event handlers that perform several steps, and in most cases, the steps must be performed in a specific order. Otherwise, the program will not produce the correct results.

Checkpoint

2.38 What is a PictureBox control used for?
2.39 Where is the PictureBox tool located in the Toolbox?
2.40 How do you display an image in the PictureBox?
2.41 What is the default value of the PictureBox control’s SizeMode property?
2.42 How does setting the SizeMode property to Zoom affect the image that is to be displayed in the PictureBox control?
2.43 How do you create a clickable image?
2.44 Does the value of a control’s Visible property change how the image appears at run time and design time?

2.8 Comments, Blank Lines, and Indentation

CONCEPT: Comments are brief notes that are placed in a program’s source code to explain how parts of the program work. Programmers commonly use blank lines and indentation in program code to give the code visual organization and make it easier to read.

Comments

Comments are short notes that are placed in different parts of a program, explaining how those parts of the program work. Comments are not intended for the compiler. They are intended for any person who is reading the code and trying to understand what it does.

In C# there are three types of comments: line comments, block comments, and documentation comments. A line comment appears on one line in a program. You begin a line
2.8 Comments, Blank Lines, and Indentation

A line comment does not have to occupy an entire line. Anything appearing after the // symbol, to the end of the line, is ignored. So, a comment can appear after an executable statement. The following code sample shows an example.

```csharp
private void showBackButton_Click(object sender, EventArgs e)
{
    cardBackPictureBox.Visible = true; // Show the card back.
    cardFacePictureBox.Visible = false; // Hide the card face.
}
```

A line comment does not have to occupy an entire line. Anything appearing after the // symbol, to the end of the line, is ignored. So, a comment can appear after an executable statement. The following code sample shows an example.

A block comment can occupy multiple consecutive lines in a program. A block comment starts with /* (a forward slash followed by an asterisk) and ends with */ (an asterisk followed by a forward slash). Everything between these markers is ignored. The following code sample shows how block comments may be used.

```csharp
/* Click event handler for the showBackButton control. This method makes the image of the back of the card visible and makes the image of the card's face invisible. */
private void showBackButton_Click(object sender, EventArgs e)
{
    cardBackPictureBox.Visible = true; // Show the card back.
    cardFacePictureBox.Visible = false; // Hide the card face.
}
```

The first five lines in this code sample are a block comment that explains what the showBackButton_Click method does. Block comments make it easier to write long explanations because you do not have to mark every line with a comment symbol.

Remember the following advice when using block comments:

- Be careful not to reverse the beginning symbol (/*) with the ending symbol (*/).
- Do not forget the ending symbol.

Each of these mistakes can be difficult to track down and will prevent the program from compiling correctly.

The third type of comment is known as a documentation comment. Documentation comments are used by professional programmers to embed extensive documentation in a program’s source code. Visual Studio can extract information from the documentation comments and generate external documentation files. Single-line documentation comments begin with three forward slashes ///. Block documentation comments begin with /// and end with */. Although documentation comments are useful for professional programmers, we do not use them in this book.

As a beginning programmer, you might resist the idea of writing a lot of comments in your programs. After all, it’s a lot more interesting to write code that actually does something. However, it’s crucial that you take the extra time to write comments. They
will almost certainly save you time in the future when you have to modify or debug the program. Even large and complex programs can be made easy to read and understand if they have proper comments.

**Using Blank Lines and Indentation to Make Your Code Easier to Read**

Programmers commonly use blank lines and indentations in their code to create a sense of visual organization. This is similar to the way that authors visually arrange the text on the pages of a book. Instead of writing each chapter as one long series of sentences, they break the text into paragraphs that are visually separated on the page. This does not change the information in the book, but it makes it easier to read.

For example, in the following code sample, we have inserted a blank line inside the method to visually separate the code into two sets of statements. The blank line is not required, but it makes the code easier for humans to read.

```csharp
private void showBackButton_Click(object sender, EventArgs e)
{
    // Make the image of the back of the card visible.
    cardBackPictureBox.Visible = true;
    // Make the image of the face of the card invisible.
    cardFacePictureBox.Visible = false;
}
```

Programmers also use indentation to visually organize code. You may have noticed that in the code editor, all the statements that appear inside a set of braces ({}) are indented. For example, all the statements inside a namespace are indented, all the statements inside a class are indented, and all the statements inside a method are indented. In fact, Visual Studio is normally set up to automatically indent the code that you write in this fashion.

Although the indentation is not required, it makes your code much easier to read. By indenting the statements inside a method, you visually set them apart. As a result, you can tell at a glance which statements are inside the method. The same is true for classes and namespaces. This practice of indentation is a convention that virtually all programmers follow.

**Checkpoint**

2.45 What purpose do comments serve?

2.46 How are line comments and block comments different?

2.47 What should you be careful to remember about the beginning and ending symbols of block comments?

2.48 Why do programmers insert blank lines and indentations in their code?

**Writing the Code to Close an Application’s Form**

**CONCEPT:** To close an application’s form in code, you use the statement `this.Close();`

All the applications that you created in this chapter’s tutorials required the user to click the standard Windows close button (X) to close the application. The standard Windows close button appears in the upper-right corner of almost every window. In many
2.10 Dealing with Syntax Errors

applications, however, you will want to give the user an alternative way to close the application. For example, you might want to create an Exit button that closes the application when it is clicked.

To close an application’s form, you execute the following statement:

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

Let’s look at an example of how this statement can be used. Figure 2-75 shows the form and code from a project named Exit Button Demo. The Button control that you see on the form is named exitButton. In the form’s code you can see that we have created a Click event handler for the button. When the user clicks the button, it closes the form, thus closing the application.

**Figure 2-75** A form with an Exit button

2.10 Dealing with Syntax Errors

**CONCEPT:** The Visual Studio code editor examines each statement as you type it and reports any syntax errors that are found. This allows you to quickly correct syntax errors.

Writing code requires a lot of precision. Even small errors, such as using an uppercase letter where you are supposed to use a lowercase letter or forgetting to end a statement with a semicolon, will prevent an application’s code from compiling and executing. Recall from Chapter 1 that these types of mistakes are known as syntax errors.

The Visual Studio code editor does a good job of reporting syntax errors soon after you type them. When you enter a statement into the editor, Visual Studio analyzes it, and if a syntax error is found, it is underlined with a jagged line. Figure 2-76 shows an example. If you hold the mouse cursor over the jagged line, a description of the error will pop up in a ToolTip window. The description usually gives you enough information to determine the cause of the error and how to fix it.
If a syntax error exists in a project’s code and you attempt to compile and execute it (by pressing the \( F5 \) key on the keyboard, or clicking the Start Debugging button on the toolbar), you will see the window shown in Figure 2-77, reporting build errors. Click the No button to close the window, and you will see the Error List shown in Figure 2-78.

Notice that the Error List window shows a description of the error, the source code file that contains the error, the line number and column number of the error, and the name of the project. If you double-click the error message that is displayed in the Error List window, the code editor will highlight the code that caused the error.

**Checkpoint**

2.49 What statement do you use to close an application’s form in code?

2.50 How can you tell that Visual Studio has found a syntax error?

2.51 What happens if you hold the mouse cursor over a jagged line in the code editor?

2.52 What happens if you attempt to compile and execute a program that contains syntax errors?
Review Questions

Multiple Choice

1. A(n) __________ is the thin dotted line that encloses an object in the Designer.
   a. selection marker
   b. control binder
   c. bounding box
   d. object container

2. The small squares that appear on the right edge, bottom edge, and lower-right corner of a form’s bounding box are called __________.
   a. sizing hooks
   b. form edges
   c. bounding tags
   d. sizing handles

3. __________ is the name of the blank form that Visual Studio initially creates in a new project.
   a. Form1
   b. Main
   c. New1
   d. Blank

4. The __________ property holds the text that is displayed on the face of the button.
   a. Name
   b. Text
   c. Tag
   d. Face

5. A file that contains program code is called a(n) __________.
   a. destination code file
   b. executable file
6. A namespace is container that holds
   a. methods
   b. names
   c. spaces
   d. classes

7. A(n) _______ is a method that executes when a specific event takes place while
   an application is running.
   a. action process
   b. event handler
   c. runtime procedure
   d. event method

8. The statement `MessageBox.Show("Hello World");` is an example of a(n) _______.
   a. method call
   b. namespace
   c. Click event
   d. event handler

9. In programming we use the term `string` to mean _______.
   a. many lines of code
   b. parallel memory locations
   c. string of characters
   d. virtually anything

10. A(n) _______ marks the end of a programming statement in C#.
    a. semicolon
    b. period
    c. hyphen
    d. underscore

11. A piece of data that is written into a program’s code is a(n) _______.
    a. identifier
    b. specifier
    c. keyword
    d. literal

12. The time during which you build the GUI and write the application’s code is re-
    ferred to as _______.
    a. run time
    b. design time
    c. code time
    d. planning

13. The time during which an application is executing is referred to as _______.
    a. go time
    b. design time
    c. execution
    d. run time

14. When you want to display text on a form, you use a _______ control.
    a. Button
    b. PictureBox
c. Label  
d. TextBox

15. The ________ property allows you to set the font, font style, and size of the control’s text.
   a. Style  
b. AutoSize  
c. Text  
d. Font

16. A ________ property can be set to one of two possible values: True or False.
   a. Boolean  
b. Logical  
c. Binary  
d. Dual

17. Label controls have a(n) ________ property that controls the way they can be resized.
   a. Stretch  
b. AutoSize  
c. Dimension  
d. Fixed

18. The ________ property can be used to change the text’s alignment in the label.
   a. TextPosition  
b. AutoAlign  
c. TextCenter  
d. TextAlign

19. In code, you use a(n) ________ to store a value in a control’s property.
   a. Click event  
b. method call  
c. assignment statement  
d. Boolean value

20. The equal sign (=) is known as the ________.
   a. equality symbol  
b. assignment operator  
c. equality operator  
d. property position

21. The standard notation for referring to a control’s property in code is ________.
   a. ControlName.PropertyName  
b. ControlName=PropertyName  
c. PropertyName.ControlName  
d. PropertyName=ControlName

22. ________ is a feature of Visual Studio that provides automatic code completion as you write programming statements.
   a. AutoCode  
b. AutoComplete  
c. IntelliSense  
d. IntelliCode

23. You can use a(n) ________ control to display a graphic image on a form.
   a. Graphics  
b. PictureBox
24. Once you have created a PictureBox control, you use its __________ property to specify the image that it will display.
   a. Image
   b. Source
   c. DrawSource
   d. ImageList

25. The PictureBox control’s __________ property specifies how the control’s image is to be displayed.
   a. RenderMode
   b. DrawMode
   c. SizeMode
   d. ImageMode

26. __________ is the image’s width to height ratio.
   a. Aspect ratio
   b. Size ratio
   c. Projection ratio
   d. Area ratio

27. Most controls have a __________ property that determines whether the control is visible on the form at run time.
   a. Render
   b. Viewable
   c. Visible
   d. Draw

28. A(an) __________ appears on one line in a program.
   a. inline comment
   b. line comment
   c. forward comment
   d. block comment

29. A __________ can occupy multiple consecutive lines in a program.
   a. block comment
   b. square comment
   c. multiline comment
   d. machine comment

30. Programmers commonly use blank lines and indentations in their code to create a sense of __________.
   a. logic
   b. visual organization
   c. documentation
   d. program flow

31. To close an application’s form in code, you use the statement __________.
   a. Close();
   b. Close.This();
   c. Close()
   d. this.Close();
**True or False**

1. Changing an object’s Text property also changes the object’s name.
2. When a form is created, its Text property is initially set to the same value as the form’s name.
3. The form’s title is displayed in the bar along the top of a form.
4. C# source code files always end with the .cs extension.
5. You add your own code to the Program.cs file as you develop an application.
6. C# code is organized as methods, which are contained inside classes, which are contained inside namespaces.
7. In C# code, each opening brace must have a corresponding closing brace at some point later in the program.
8. When you double-click a control in the Designer, Visual Studio not only creates an empty event handler, but it also writes some code that you don’t see, elsewhere in the project that is necessary for the event handler to properly function.
9. A Label control’s Text property is initially set to the same value as the Label control’s name.
10. When a Label control’s AutoSize property is set to True, you cannot manually change the size of the control by clicking and dragging its bounding box.
11. By default, a label’s text is aligned with the bottom and right edges of the label’s bounding box.
12. Label controls are useful for displaying output while an application is running.
13. The assignment operator assigns the value that appears on its left side to the item that appears on its right side.
14. PictureBox controls also have a BorderStyle property that works just like a Label control’s BorderStyle property.
15. Buttons are the only controls that can respond to Click events.
16. The Visible property is a Binary property, which means it can be set only to the values 1 and 0.
17. When you write the values true or false in code, they must be written in all lowercase letters.
18. In C# there are three types of comments: line comments, block comments, and documentation comments.
19. To close an application’s form in code, you use the statement `Close.This();`
20. The Visual Studio code editor examines each statement as you type it, and reports any syntax errors that are found.

**Short Answer**

1. What does a bounding box indicate about an object in the Designer?
2. What happens when you position the mouse cursor over an edge or corner of a bounding box that has sizing handles?
3. What determines an object’s appearance and other characteristics?
114  Chapter 2  Introduction to Visual C#

4. What is shown by each column in the Properties window?

5. What steps must you perform to change a form’s Text property?

6. What steps must you perform to change a form’s Size property in the Properties window?

7. How do you move a control to a new location on the form using the mouse?

8. What steps do you perform to change a Button control’s Text property?

9. Briefly describe the contents of the Form1.cs file.

10. In code, what characters do you enclose a string literal in?

11. When creating an event handler for a button, is it possible to skip a step by opening the code editor and writing all the event handler code yourself? Why or why not?

12. Briefly describe the difference between design time and run time.

13. Describe the appearance of a Label control that’s BorderStyle property is set to Fixed3D.

14. What does it mean when a Label control’s AutoSize property is set to True?

15. What are the values that the TextAlign property may be set to?

16. How do you clear the text that is displayed in a Label control in code?

17. What are the different image formats that a PictureBox control can display?

18. List the values that the SizeMode property of a PictureBox control can be set to.

19. What are the three types of comments you can use in Visual C#?

20. How does Visual Studio help you to quickly correct syntax errors?

Algorithm Workbench

1. What statement would you write to display Good Afternoon in a message box?

2. What statement would you write to display your name in a message box?

3. Suppose an application’s GUI has a Label control named dogLabel. Write a statement that causes Fido to be displayed in the dogLabel control.

4. Suppose an application’s GUI has a Label control named outputLabel. Write a statement that clears any text that happens to be displayed by the control.

5. Suppose an application’s GUI has a PictureBox control named myPicture. Write a statement that makes the control invisible.

Programming Problems

1. Latin Translator

   Look at the following list of Latin words and their meanings.

<table>
<thead>
<tr>
<th>Latin</th>
<th>English</th>
</tr>
</thead>
<tbody>
<tr>
<td>sinister</td>
<td>left</td>
</tr>
<tr>
<td>dexter</td>
<td>right</td>
</tr>
<tr>
<td>medium</td>
<td>center</td>
</tr>
</tbody>
</table>
Create an application that translates the Latin words to English. The form should have three buttons, one for each Latin word. When the user clicks a button, the application should display the English translation in a Label control.

2. **Clickable Number Images**

In the Chap02 folder, in the Student Sample Program files, you will find the image files shown in Figure 2-79. Create an application that displays these images in PictureBox controls. The application should perform the following actions:

- When the user clicks the 1 image, the application should display the word *One* in a message box.
- When the user clicks the 2 image, the application should display the word *Two* in a message box.
- When the user clicks the 3 image, the application should display the word *Three* in a message box.
- When the user clicks the 4 image, the application should display the word *Four* in a message box.
- When the user clicks the 5 image, the application should display the word *Five* in a message box.

![Image files](Image files)

3. **Card Identifier**

In the Student Sample Programs that accompany this book, you will find a folder named Images\Cards\Poker Large. In that folder you will find JPEG image files for a complete deck of poker cards. Create an application with five PictureBox controls. Each PictureBox should display a different card from the set of images. When the user clicks any of the PictureBox controls, the name of the card should be displayed in a Label control. Figure 2-80 shows an example of the application running. The image on the left shows the application’s form when it starts running. The image on the right shows the form after the user has clicked the two of clubs card.

![Card Identifier application](Card Identifier application)
4. **Joke and Punch line**
   A joke typically has two parts: a setup and a punch line. For example, this might be the setup for a joke:
   
   *How many programmers does it take to change a lightbulb?*
   
   And this is the punch line:
   
   *None. That’s a hardware problem.*
   
   Think of your favorite joke and identify its setup and punch line. Then, create an application that has a Label and two buttons on a form. One of the buttons should read “Setup” and the other button should read “Punch line.” When the **Setup** button is clicked, display the joke’s setup in the Label control. When the **Punch line** button is clicked, display the joke’s punch line in the Label control.

5. **Heads or Tails**
   In the Student Sample Programs that accompany this book you will find a folder named *Images\Coins* that contains images showing the heads and tails sides of a coin. Create an application with a **Show Heads** button and a **Show Tails** button. When the user clicks the **Show Heads** button, an image of the heads side of a coin should appear. When the user clicks the **Show Tails** button, an image of the tails side of a coin should appear. Figure 2-81 shows examples of how the application’s form might appear.

![Figure 2-81 The Heads or Tails application](image)

6. **Orion Constellation**
   Orion is one of the most famous constellations in the night sky. In the *Chap02* folder of the Student Sample Programs that accompany this book, you will find an image file named Orion.bmp, which contains a diagram of the Orion constellation. Create an application that displays the Orion image in a PictureBox control, as shown on the left in Figure 2-82. The application should have a button that, when clicked, displays the names of each of the stars, as shown on the right in Figure 2-82. The application should have another button that, when clicked, hides the star names. The names of the stars are *Betelgeuse, Meissa, Alnitak, Alnilam, Mintaka, Saiph,* and *Rigel.*

   **Hint:** Place the PictureBox control with the Orion image on the form. Then, place Label controls containing the star names on top of the PictureBox. Use the *Properties* window to set each of the Label control’s Visible property to False. That will cause the labels to be invisible when the application runs. The **Show Star Names** button will set each of the Label control’s Visible property to true, and the **Hide Star Names** button will set each of the Label control’s Visible property to false.
Figure 2-82 The Orion Constellation application
This page intentionally left blank
CHAPTER 3  Processing Data

TOPICS

3.1  Reading Input with TextBox Controls  
3.2  A First Look at Variables  
3.3  Numeric Data Types and Variables  
3.4  Performing Calculations  
3.5  Inputting and Outputting Numeric Values  
3.6  Formatting Numbers with the ToString Method  
3.7  Simple Exception Handling  
3.8  Using Named Constants  
3.9  Declaring Variables as Fields  
3.10 Using the Math Class  
3.11 More GUI Details

3.1 Reading Input with TextBox Controls

CONCEPT: The TextBox control is a rectangular area that can accept keyboard input from the user.

Many of the programs that you will write from this point forward will require the user to enter data. The data entered by the user will then be used in some sort of operation. One of the primary controls that you will use to get data from the user is the TextBox control.

A TextBox control appears as a rectangular area on a form. When the application is running, the user can type text into a TextBox control. The program can then retrieve the text that the user entered and use that text in any necessary operations.

In the Toolbox, the TextBox tool is located in the Common Controls group. When you double-click the tool, a TextBox control is created on the form, as shown in Figure 3-1. When you create TextBox controls, they are automatically given default names such as textBox1, textBox2, and so forth. As you learned in Chapter 2, you should always change a control’s default name to something more meaningful.

When the user types into a TextBox control, the text is stored in the control’s Text property. In code, if you want to retrieve the data that has been typed into a TextBox, you simply retrieve the contents of the control’s Text property.
Let’s look at an example. Make sure you have downloaded the student sample programs from the book’s companion Web site (at www.pearsonhighered.com/gaddis). In the Chap03 folder, you will find a project named TextBox Demo. Figure 3-2 shows the form, with most of the control names specified, and Figure 3-3 shows the form’s code. (In Figure 3-3, to conserve space on the page, we have scrolled past the using directives that appear at the top of the code file.)

Notice in Figure 3-3 that the readInputButton control’s Click event handler performs the following assignment statement:

```csharp
outputLabel.Text = nameTextBox.Text;
```

This statement assigns the value of the nameTextBox control’s Text property to the outputLabel control’s Text property. In other words, it gets any text that has been entered by the user into the nameTextBox control and displays it in the outputLabel control. If you run the application, Figure 3-4 shows an example of how the form appears after you have entered Kathryn Smith and clicked the readInputButton control.

### Clearing the Contents of a TextBox Control

You can clear the contents of a TextBox control in the same way that you clear the contents of a Label control: you assign an empty string ("") to the control’s Text property. For example, the following statement clears the contents of the nameTextBox control:

```csharp
nameTextBox.Text = "";
```
When this statement executes, the `nameTextBox` control will appear empty on the application’s form.

Checkpoint

3.1 What control can be used to gather text input from the user?
3.2 In code, how do you retrieve data that has been typed into a TextBox control?
3.3 What type of data does a control’s Text property always contain?
3.4 How do you clear the contents of a TextBox control?

3.2 A First Look at Variables

**CONCEPT:** A variable is a storage location in memory that is represented by a name.

Most programs store data in the computer’s memory and perform operations on that data. For example, consider the typical online shopping experience: you browse a Web
site and add the items that you want to purchase to the shopping cart. As you add items
to the shopping cart, data about those items is stored in memory. Then, when you click
the checkout button, a program running on the Web site’s computer calculates the cost
of all the items you have in your shopping cart, applicable sales taxes, shipping costs, and
the total of all these charges. When the program performs these calculations, it stores the
results in the computer’s memory.

Programs use variables to store data in memory. A variable is a storage location in mem-
ory that is represented by a name. For example, a program that manages a company’s cus-
tomer mailing list might use a variable named lastName to hold a customer’s last name, a
variable named firstName to hold the customer’s first name, a variable named address
to hold the customer’s mailing address, and so on.

In C#, you must declare a variable in a program before you can use it to store data. You
do this with a variable declaration, which specifies two things about the variable:

1. The variable’s data type, which is the type of data the variable will hold
2. The variable’s name

A variable declaration statement is written in this general format:

    dataTypeVariableName;

Let’s take a closer look at each of these.

**Data Type**

A variable’s data type indicates the type of data that the variable will hold. Before you
declare a variable, you need to think about the type of value that will be stored in the vari-
able. For example, will the variable hold a string or a number? If it will hold a number,
what kind of number will it be, an integer or a real number? When you have determined
the kind of data that the variable will hold, you select one of the data types that C# pro-
vides for a variable.

The C# language provides many data types for storing fundamental types of data, such as
strings, integers, and real numbers. These data types are known as primitive data types.
We will look at several of them in this chapter.

**Variable Name**

A variable name identifies a variable in the program code. When naming a variable, you
should always choose a meaningful name that indicates what the variable is used for.
For example, a variable that holds the temperature might be named temperature, and
a variable that holds a car’s speed might be named speed. You may be tempted to give
variables short, nondescript names such as x or b2, but names such as these give no clue
as to the purpose of the variable.

In addition, there are some specific rules that you must follow when naming a variable.
The same rules for identifiers that apply to control names also apply to variable names.
We discussed these rules in Chapter 2, but we review them now:

- The first character must be one of the letters a through z or A through Z or an
  underscore character (_).
- After the first character, you may use the letters a through z or A through Z, the
digits 0 through 9, or underscores.
- The name cannot contain spaces.

When naming variables, we use the same camelCase naming convention that we intro-
duced in Chapter 2 for control names. For example, if we are declaring a variable to hold
3.2 A First Look at Variables

an employee's gross pay, we might name it grossPay. Or, if we are declaring a variable to a
customer number, we might name it customerNumber.

**string Variables**

The first primitive data type we consider is the string data type. A variable of the string
data type can hold any string of characters, such as a person's name, address, password,
and so on. Here is an example of a statement that declares a string variable named
productDescription:

```csharp
string productDescription;
```

After the variable has been declared, you can use the assignment operator (=) to store a
value in the variable. Here is an example:

```csharp
productDescription = "Italian Espresso Machine";
```

When this statement executes, the string literal "Italian Espresso Machine" is as-
signed to the productDescription variable. When writing an assignment statement,
remember that the assignment operator assigns the value that appears on its right side to
the variable that appears on its left side.

Once you have assigned a value to a variable, you can use the variable in other opera-
tions. For example, assume productLabel is the name of a Label control. The following
statement assigns the productDescription string to the productLabel control's Text
property:

```csharp
productLabel.Text = productDescription;
```

After this statement executes, the string that is stored in the productDescription variable
is displayed in the productLabel control. The following statement shows another example:

```csharp
MessageBox.Show(productDescription);
```

When this statement executes, the string that is stored in the productDescription vari-
able is displayed in a message box.

**String Concatenation**

A common operation that performed on strings is concatenation, or appending one string
to the end of another string. In C# you use the + operator to concatenate strings. The +
operator produces a string that is the combination of the two strings used as its operands.
The following code shows an example:

```csharp
string message;
message = "Hello " + "world";
MessageBox.Show(message);
```

The first statement declares a string variable named message. The second statement
combines the strings "Hello " and "world" to produce the string "Hello world". The
string "Hello world" is then assigned to the message variable. The third statement dis-
plays the contents of the message variable in a message box. When the message box is
displayed, it shows the string Hello world.

Let's look at an application that further demonstrates string concatenation. In the
Chap03 folder of this book's student sample programs (available for download at www.
pearsonhighered.com/gaddis), you will find a project named String Variable Demo. Fig-
ure 3-5 shows the form, with most of the control names specified, and Figure 3-6 shows
the form's code. (In Figure 3-6, to conserve space on the page, we have scrolled past the
using directives that appear at the top of the code file.)
In Figure 3-6, three statements in the `showNameButton_Click` event handler are pointed out:

1. This statement is a variable declaration. It declares a `string` variable named `fullName`.

2. This statement assigns the result of a string concatenation to the `fullName` variable. The string that is assigned to the variable begins with the value of the `firstNameTextBox` control’s Text property, followed by a space (" "), followed by the value of the `lastNameTextBox` control’s Text property. For example, if the user has entered `Joe` into the `firstNameTextBox` control and `Smith` into the `lastNameTextBox` control, this statement will assign the string "Joe Smith" to the `fullName` variable.

3. This statement assigns the `fullName` variable to the `fullNameLabel` control’s Text property. As a result, the string that is stored in the `fullName` variable is displayed in the `fullNameLabel` control.
If you run the application, Figure 3-7 shows an example of how the form appears after you have entered *Chris* for the first name and *Jones* for the last name and clicked the *showNameButton* control.

**Figure 3-7** The user’s full name displayed in the label

---

### Declaring Variables before Using Them

The purpose of a variable declaration statement is to tell the compiler that you plan to use a variable of a specified name to store a particular type of data in the program. A variable declaration statement causes the variable to be created in memory. For this reason, a variable’s declaration statement must appear *before* any other statements in the method that use the variable. This makes perfect sense because you cannot store a value in a variable if the variable has not been created in memory.

### Local Variables

Notice that the `fullName` variable in Figure 3-6 is declared inside the event handler method. Variables that are declared inside a method are known as local variables. A local variable belongs to the method in which it is declared, and only statements inside that method can access the variable. (The term *local* is meant to indicate that the variable can be used only locally, within the method in which it is declared.)

An error will occur if a statement in one method tries to access a local variable that belongs to another method. For example, let’s go over the sample code shown in Figure 3-8:

1. This statement declares a `string` variable named `myName`. The variable is declared inside the `firstButton_Click` event handler, so it is local to that method.
2. This statement, which is also in the `firstButton_Click` event handler, assigns the `nameTextBox` control’s `Text` property to the `myName` variable.

**Figure 3-8** One method trying to access a variable that is local to another method
This statement, which is in the secondButton_Click event handler, attempts to assign the myName variable to the outputLabel control’s Text property. This statement will not work, however, because the myName variable is local to the firstButton_Click event handler, and statements in the secondButton_Click event handler cannot access it.

Scope of a Variable

Programmers use the term scope to describe the part of a program in which a variable may be accessed. A variable is visible only to statements inside the variable’s scope.

A local variable’s scope begins at the variable’s declaration and ends at the end of the method in which the variable is declared. As you saw in the previous example, a local variable cannot be accessed by statements that are outside the method. In addition, a local variable cannot be accessed by code that is inside the method but before the variable’s declaration.

Lifetime of a Variable

A variable’s lifetime is the time period during which the variable exists in memory while the program is executing. A local variable is created in memory when the method in which it is declared starts executing. When the method ends, all the method’s local variables are destroyed. So, a local variable’s lifetime is the time during which the method in which it is declared is executing.

Duplicate Variable Names

You cannot declare two variables with the same name in the same scope. For example, if you declare a variable named productDescription in an event handler, you cannot declare another variable with that name in the same event handler. You can, however, have variables of the same name declared in different methods.

Assignment Compatibility

You can assign a value to a variable only if the value is compatible with the variable’s data type. Only strings are compatible with the string data type, so all the assignments in the following code sample work:

```csharp
1 // Declare and initialize a string variable.
2 string productDescription = "Chocolate Truffle";
3
4 // Declare another string variable.
5 string myFavoriteProduct;
6
7 // Assign a value to a string variable.
8 myFavoriteProduct = productDescription;
9
10 // Assign a value from a TextBox to a string variable.
11 productDescription = userInputTextBox.Text;
```

The following comments explain these lines of code:

- In line 2 we initialize a string variable with a string literal. This works because string literals are assignment compatible with string variables.
- In line 8 we assign a string variable to another string variable. This works for the obvious reason that string variables are compatible with other string variables.
• Assume that the application has a TextBox control named `userInputTextBox`. In line 11 we assign the value of the TextBox control’s Text property to a `string` variable. This works because the value in a control’s Text property is always a string.

The following code will not work, however, because it attempts to assign a nonstring value to a `string` variable:

```csharp
1 // Declare a string variable.
2 string employeeID;
3
4 // Assign a value to the variable. Will this work?
5 employeeID = 125; ← ERROR!
```

In line 5 we are attempting to assign the number 125 to a `string` variable. Numbers are not assignment compatible with `string` variables, so this statement will cause an error when the code is compiled.

**NOTE:** Although you cannot store the number 125 in a `string` variable, you can store the string literal "125" in a `string` variable.

---

**A Variable Holds One Value at a Time**

Variables can hold different values while a program is running, but they can hold only one value at a time. When you assign a value to a variable, that value will remain in the variable until you assign a different value to the variable. For example, look at the following code sample:

```csharp
1 // Declare a string variable.
2 string productDescription;
3
4 // Assign a value to the variable.
5 productDescription = "Large Medium-Roast Coffee";
6
7 // Display the variable's value.
8 MessageBox.Show(productDescription);
9
10 // Assign a different value to the variable.
11 productDescription = "Chocolate Truffle";
12
13 // Display the variable's value.
14 MessageBox.Show(productDescription);
```

The following comments explain what we did:

- Line 2 declares a `string` variable named `productDescription`.
- Line 5 assigns the string "Large Medium-Roast Coffee" to the `productDescription` variable.
- Line 8 displays the value of the `productDescription` variable in a message box. (The message box will display Large Medium-Roast Coffee.)
- Line 11 assigns a different value to the `productDescription` variable. After this statement executes, the `productDescription` variable will hold the string "Chocolate Truffle".
- Line 14 displays the value of the `productDescription` variable in a message box. (The message box will display Chocolate Truffle.)
This code sample illustrates two important characteristics of variables:

- A variable holds only one value at a time.
- When you store a value in a variable, that value replaces the previous value that was in the variable.

Tutorial 3-1 gives you some practice using variables. You will create an application that uses TextBox controls to get input values, stores those values in variables, and uses the variables in operations.

**Tutorial 3-1:**

The *Birth Date String* Application

In this tutorial you create an application that lets the user enter the following information about his or her birthdate:

- The day of the week (Monday, Tuesday, etc.)
- The name of the month (January, February, etc.)
- The numeric day of the month
- The year

Figure 3-9 shows the application’s form, along with the names of all the controls. When the application runs, the user enters each piece of data into a separate TextBox. When the user clicks the *Show Date* button, the application concatenates the contents of the TextBoxes into a string such as *Friday, June 1, 1990*. The string is displayed in the *dateOutputLabel* control. When the user clicks the *Clear* button, the contents of the TextBoxes and the *dateOutputLabel* control are cleared. The *Exit* button closes the application’s form.

**Figure 3-9** The *Birth Date String* form

---

**Step 1:** Start Visual Studio (or Visual Studio Express) and begin a new Windows Forms Application project named *Birth Date String*.

**Step 2:** Set up the application’s form as shown in Figure 3-9. Notice that the form’s Text property is set to *Birth Date String*. The names of the controls are shown in the figure. As you place each control on the form, refer to Table 3-1 for the relevant property settings.

**Step 3:** Once you have set up the form with its controls, you can create the Click event handlers for the Button controls. At the end of this tutorial, Program 3-1 shows the completed code for the form. You will be instructed to refer to Program 3-1.
Table 3-1  Control property settings

<table>
<thead>
<tr>
<th>Control Name</th>
<th>Control Type</th>
<th>Property Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>dayOfWeekPromptLabel</td>
<td>Label</td>
<td>Text: Enter the day of the week</td>
</tr>
<tr>
<td>monthPromptLabel</td>
<td>Label</td>
<td>Text: Enter the name of the month</td>
</tr>
<tr>
<td>dayOfMonthPromptLabel</td>
<td>Label</td>
<td>Text: Enter the numeric day of the month</td>
</tr>
<tr>
<td>yearPromptLabel</td>
<td>Label</td>
<td>Text: Enter the year</td>
</tr>
<tr>
<td>dayOfWeekTextBox</td>
<td>TextBox</td>
<td>No properties changed</td>
</tr>
<tr>
<td>monthTextBox</td>
<td>TextBox</td>
<td>No properties changed</td>
</tr>
<tr>
<td>dayOfMonthTextBox</td>
<td>TextBox</td>
<td>No properties changed</td>
</tr>
<tr>
<td>yearTextBox</td>
<td>TextBox</td>
<td>No properties changed</td>
</tr>
<tr>
<td>dateOutputLabel</td>
<td>Label</td>
<td>AutoSize: False  BorderStyle: FixedSingle  Text: (The contents of the Text property have been erased.)  TextAlign: MiddleCenter</td>
</tr>
<tr>
<td>showDateButton</td>
<td>Button</td>
<td>Text: Show Date</td>
</tr>
<tr>
<td>clearButton</td>
<td>Button</td>
<td>Text: Clear</td>
</tr>
<tr>
<td>exitButton</td>
<td>Button</td>
<td>Text: Exit</td>
</tr>
</tbody>
</table>

as you write the event handlers. (Remember, the line numbers that are shown in Program 3-1 are not part of the program. They are shown for reference only.)

In the Designer, double-click the showDateButton control. This will open the code editor, and you will see an empty event handler named showDateButton_Click. Complete the showDateButton_Click event handler by typing the code shown in lines 22–32 in Program 3-1.

Let’s take a closer look at the code:

**Line 23:** This statement declares a string variable named output.

**Lines 26–29:** These lines are actually one long statement, broken up into multiple lines. The statement concatenates the Text properties of the TextBox controls, along with appropriately placed commas and spaces, to create the date string. The resulting string is assigned to the output variable.

For example, suppose the user has entered the following input:

- Friday in the dayOfWeekTextBox control.
- June in the monthTextBox control.
- 1 in the dayOfMonthTextBox control.
- 1990 in the yearTextBox control.

The concatenation in this statement produces the string “June 1, 1990”; it is assigned to the output variable.

**Line 32:** This statement assigns the output variable to the dateOutputLabel control’s Text property. When this statement executes, the contents of the output variable are displayed in the dateOutputLabel control.

**Step 4:** Switch your view back to the Designer and double-click the clearButton control. In the code editor, you will see an empty event handler named...
clearButton_Click. Complete the clearButton_Click event handler by typing the code shown in lines 37–44 in Program 3-1.

Let's take a closer look at the code:

Lines 38–41: Each statement assigns an empty string ("") to the Text property of one of the TextBox controls. When these statements have finished executing, the TextBox controls will appear empty.

Line 44: This statement assigns an empty string ("") to the dateOutputLabel control’s Text property. After the statement has executed, the label appears empty.

Step 5: Switch your view back to the Designer and double-click the exitButton control. In the code editor, you will see an empty event handler named exitButton_Click. Complete the exitButton_Click event handler by typing the code shown in lines 49–50 in Program 3-1.

Step 6: Save the project. Then, press the [F5] key on the keyboard, or click the Start Debugging button ( ) on the toolbar to compile and run the application. The form will appear as shown in the image on the left in Figure 3-10. Test the application by entering values into the TextBoxes and clicking the Show Date button. The date should be displayed, similar to the image shown on the right in the figure. Click the Clear button, and the contents of the TextBoxes and the Label control should clear. Click the Exit button and the form should close.

Figure 3-10 The Birth Date String application

Program 3-1 Completed Form1 code for the Birth Date String application

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Birth_Date_String
{
    public partial class Form1 : Form
    {
        public Form1()
        {
```
NOTE: In Tutorial 3-1, the statement in lines 26–29 shows an example of how you can break up a statement into multiple lines. Quite often, you will find yourself writing statements that are too long to fit entirely inside the Code window. Your code will be hard to read if you have to horizontally scroll the Code window to view long statements. In addition, if you or your instructor chooses to print your code, the statements that are too long to fit on one line of the page will wrap around to the next line and make your code look unorganized. For these reasons, it is usually best to break a long statement into multiple lines.

When typing most statements, you can simply press the Enter key when you reach an appropriate point to continue the statement on the next line. Remember, however, that you cannot break up a keyword, a quoted string, or an identifier (such as a variable name or a control name).
### Initializing Variables

In C#, a variable must be assigned a value before it can be used. For example, look at this code:

```csharp
string productDescription;
MessageBox.Show(productDescription);
```

This code declares a `string` variable named `productDescription` and then tries to display the variable’s value in a message box. The only problem is that we have not assigned a value to the variable. When we compile the application containing this code, we will get an error message such as `Use of unassigned local variable ‘productDescription’`. The C# compiler will not compile code that tries to use an unassigned variable.

One way to make sure that a variable has been assigned a value is to **initialize** the variable with a value when you declare it. For example, the following statement declares a `string` variable named `productDescription` and immediately assigns the string literal "Chocolate Truffle" to it:

```csharp
string productDescription = "Chocolate Truffle";
```

We say that this statement **initializes** the `productDescription` variable with the string "Chocolate Truffle". Here is another example:

```csharp
string lastName = lastNameTextBox.Text;
```

Assume that this statement belongs to an application that has a `TextBox` named `lastNameTextBox`. The statement declares a `string` variable named `lastName` and initializes it with the value of the `lastNameTextBox` control’s `Text` property.

### Declaring Multiple Variables with One Statement

You can declare multiple variables of the same data type with one declaration statement. Here is an example:

```csharp
string lastName, firstName, middleName;
```

This statement declares three `string` variables named `lastName`, `firstName`, and `middleName`. Notice that commas separate the variable names. Here is an example of how we can declare and initialize the variables with one statement:

```csharp
string lastName = "Jones", firstName = "Jill", middleName = "Rebecca";
```

Remember, you can break up a long statement so it spreads across two or more lines. Sometimes you will see long variable declarations written across multiple lines, like this:

```csharp
string lastName = "Jones",
firstName = "Jill",
middleName = "Rebecca";
```

### Checkpoint

3.5 What is the purpose of a variable?

3.6 Give an example of a variable declaration that will store the name of your favorite food.
3.7 For each of the following items, determine whether the data type should be an integer, string, or real number.
   a. pet name
   b. sales tax
   c. mailing address
   d. video game score

3.8 Indicate whether each of the following is a legal variable name. If it is not, explain why.
   a. pay_Rate
   b. speed_of_sound
   c. totalCost
   d. 2ndPlaceName

3.9 What will be stored in the message variable after the following statement is executed?
   ```
   string message = "He" + "ll" + "o!";
   ```

3.10 What is the lifetime of a variable that is declared inside of a Click event handler?

3.11 Assuming the variable greeting has not been assigned a value, what will be the result of the following statement?
   ```
   MessageBox.Show(greeting);
   ```

3.12 Will the following statement cause an error? Why or why not?
   ```
   string luckyNumber = 7;
   ```

3.13 Write a single declaration statement for the variables name, city, and state.

---

**3.3 Numeric Data Types and Variables**

**CONCEPT:** If you need to store a number in a variable and use that number in a mathematical operation, the variable must be of a numeric data type. You select a numeric data type that is appropriate for the type of number that you need to store.

In the previous section you read about string variables. Variables of the string data type can be used to store text, but they cannot store numeric data for the purpose of performing mathematical operations. If you need to store numbers and perform mathematical operations on them, you have to use a numeric data type.

The C# language provides several primitive data types. You can read about all the C# primitive data types in Appendix A. Many of the data types provided by C# are for specialized purposes beyond the scope of this book. When it comes to numeric data, most of the time you will use the three numeric primitive data types described in Table 3-2.

Here are examples of declaring variables of each data type:

```
int speed;
double distance;
decimal grossPay;
```

The first statement declares an int variable named speed. The second example declares a double variable named distance. The third statement declares a decimal variable named grossPay.
Chapter 3  Processing Data

Numeric Literals

You learned in Chapter 2 that a literal is a piece of data written into a program’s code. When you know, at the time that you are writing a program’s code, that you want to store a specific value in a variable, you can assign that value as a literal to the variable.

A numeric literal is a number that is written into a program’s code. For example, the following statement declares an `int` variable named `hoursWorked` and initializes it with the value 40:

```
int hoursWorked = 40;
```

In this statement, the number 40 is a numeric literal. The following shows another example:

```
double temperature = 87.6;
```

This statement declares a `double` variable named `temperature` and initializes it with the value 87.6. The number 87.6 is a numeric literal.

### Table 3-2  The primitive numeric data types that you will use most often

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int</code></td>
<td>A variable of the <code>int</code> data type can hold whole numbers only. For example, an <code>int</code> variable can hold values such as 42, 0, and −99. An <code>int</code> variable cannot hold numbers with a fractional part, such as 22.1 or −4.9. The <code>int</code> data type is the primary data type for storing integers. We use it in this book any time we need to store and work with integers. An <code>int</code> variable uses 32 bits of memory and can hold an integer number in the range of −2,147,483,648 through 2,147,483,647.</td>
</tr>
<tr>
<td><code>double</code></td>
<td>A variable of the <code>double</code> data type can hold real numbers, such as 3.5, −87.95, or 3.0. A number that is stored in a <code>double</code> variable is rounded to 15 digits of precision. We use variables of the <code>double</code> data type to store any number that might have a fractional part. The <code>double</code> data type is especially useful for storing extremely great or extremely small numbers. In memory a <code>double</code> variable uses 64 bits of memory. It is stored in a format that programmers call <code>double precision floating-point notation</code>. Variables of the <code>double</code> data type can hold numbers in the range of ( \pm 5.0 \times 10^{−324} ) to ( \pm 1.7 \times 10^{308} ).</td>
</tr>
<tr>
<td><code>decimal</code></td>
<td>A variable of the <code>decimal</code> data type can hold real numbers with greater precision than the <code>double</code> data type. A number that is stored in a <code>decimal</code> variable is rounded to 28 digits of precision. Because <code>decimal</code> variables store real numbers with a great deal of precision, they are most commonly used in financial applications. In this book we typically use the <code>decimal</code> data type when storing amounts of money. In memory a <code>decimal</code> variable uses 128 bits of memory. It is stored in a format that programmers call <code>decimal notation</code>. Variables of the <code>decimal</code> data type can hold numbers in the range of ( \pm 1.0 \times 10^{−228} ) to ( \pm 7.9 \times 10^{28} ).</td>
</tr>
</tbody>
</table>
When you write a numeric literal in a program's code, the numeric literal is assigned a data type. In C#, if a numeric literal is an integer (not written with a decimal point) and it fits within the range of an `int` (see Table 3-2 for the minimum and maximum values), then the numeric literal is treated as an `int`. A numeric literal that is treated as an `int` is called an integer literal. For example, each of the following statements initializes a variable with an integer literal:

```csharp
int hoursWorked = 40;
int unitsSold = 650;
int score = -23;
```

If a numeric literal is written with a decimal point and it fits within the range of a `double` (see Table 3-2 for the minimum and maximum values), then the numeric literal is treated as a `double`. A numeric literal that is treated as a `double` is called a double literal. For example, each of the following statements initializes a variable with a double literal:

```csharp
double distance = 28.75;
double speed = 87.3;
double temperature = -10.0;
```

When you append the letter `M` or `m` to a numeric literal, it is treated as a decimal and is referred to as a decimal literal. Here are some examples:

```csharp
decimal payRate = 28.75m;
decimal price = 8.95M;
decimal profit = -50m;
```

**TIP:** Because `decimal` is the preferred data type for storing monetary amounts, remembering that “m” stands for “money” might help you to remember that decimal literals must end with the letter `M` or `m`.

### Assignment Compatibility for `int` Variables

You can assign `int` values to `int` variables, but you cannot assign `double` or `decimal` values to `int` variables. For example, look at the following declarations.

```csharp
int hoursWorked = 40; // This works
int unitsSold = 650m; // ERROR!
int score = -25.5; // ERROR!
```

The first declaration works because we are initializing an `int` variable with an `int` value. The second declaration causes an error, however, because you cannot assign a `decimal` value to an `int` variable. The third declaration also causes an error because you cannot assign a `double` value to an `int` variable.

You cannot assign a `double` or a `decimal` value to an `int` variable because such an assignment could result in a loss of data. Here are the reasons:

- The `double` and `decimal` values may be fractional, but `int` variables can hold only integers. If you were allowed to store a fractional value in an `int` variable, the fractional part of the value would have to be discarded.
• The double and decimal values may be much larger or much smaller than allowed by the range of an int variable. A double or a decimal number can potentially be so large or so small that it will not fit in an int variable.

**Assignment Compatibility for double Variables**

You can assign either double or int values to double variables, but you cannot assign decimal values to double variables. For example, look at the following declarations.

```csharp
double distance = 28.75;  // This works
double speed = 75;        // This works
double sales = 6500.0m;   // ERROR!
```

The first declaration works because we are initializing a double variable with a double value. The second declaration works because we are initializing a double variable with an int value. The third declaration causes an error, however, because you cannot assign a decimal value to a double variable.

It makes sense that you are allowed to assign an int value to a double variable because any number that can be stored as an int can be converted to a double with no loss of data. When you assign an int value to a double variable, the int value is implicitly converted to a double.

You cannot assign a decimal value to a double variable because the decimal data type allows for much greater precision than the double data type. A decimal value can have up to 28 digits of precision, whereas a double can provide only 15 digits of precision. Storing a decimal value in a double variable could potentially result in a loss of data.

**Assignment Compatibility for decimal Variables**

You can assign either decimal or int values to decimal variables, but you cannot assign double values to decimal variables. For example, look at the following declarations.

```csharp
decimal balance = 9280.73m; // This works
decimal price = 50;         // This works
decimal sales = 6500.0;     // ERROR!
```

The first declaration works because we are initializing a decimal variable with a decimal value. The second declaration works because we are initializing a decimal variable with an int value. When you assign an int value to a decimal variable, the int value is implicitly converted to a decimal with no loss of data. The third declaration causes an error, however, because you cannot assign a double value to a decimal variable. A double value can potentially be much larger or much smaller than allowed by the range of a decimal.

**Explicitly Converting Values with Cast Operators**

Let’s consider a hypothetical situation. Suppose you’ve written an application that uses a double variable, and for some reason, you need to assign the contents of the double variable to an int variable. In this particular situation, you know that the double variable’s value is something that can be safely converted to an int without any loss of data (such as 3.0, or 98.0). However, the C# compiler will not allow you to make the assignment because double values are not assignment compatible with int variables. Isn’t there a way to override the C# rules in this particular situation and make the assignment anyway?
The answer is yes, there is a way. You can use a **cast operator** to explicitly convert a value from one numeric data type to another, even if the conversion might result in a loss of data. A cast operator is the name of the desired data type, written inside parentheses and placed to the left of the value that you want to convert. The following code sample demonstrates:

```java
1 // Declare an int variable.
2 int wholeNumber;
3
4 // Declare a double variable.
5 double realNumber = 3.0;
6
7 // Assign the double to the int.
8 wholeNumber = (int)realNumber;
```

The following points describe the code:

- Line 2 declares an `int` variable named `wholeNumber`.
- Line 5 declares a `double` variable named `realNumber`, initialized with the value 3.0.
- Line 8 uses a cast operator to convert the value of `realNumber` to an `int` and assigns the converted value to `wholeNumber`. After this statement executes, the `wholeNumber` variable is assigned the value 3.

Table 3-3 shows other code examples involving different types of cast operators.

<table>
<thead>
<tr>
<th>Code Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int wholeNumber;</code></td>
<td>The <code>(int)</code> cast operator converts the value of the <code>wholeNumber</code> variable to an <code>int</code>. The converted value is assigned to the <code>wholeNumber</code> variable.</td>
</tr>
<tr>
<td><code>decimal moneyNumber = 4500m;</code> <code>wholeNumber = (int)moneyNumber;</code></td>
<td>The <code>(int)</code> cast operator converts the value of the <code>moneyNumber</code> variable to an <code>int</code>. The converted value is assigned to the <code>wholeNumber</code> variable.</td>
</tr>
<tr>
<td><code>double realNumber;</code> <code>decimal moneyNumber = 625.70m;</code> <code>realNumber = (double)moneyNumber;</code></td>
<td>The <code>(double)</code> cast operator converts the value of the <code>moneyNumber</code> variable to a <code>double</code>. The converted value is assigned to the <code>realNumber</code> variable.</td>
</tr>
<tr>
<td><code>decimal moneyNumber;</code> <code>double realNumber = 98.9;</code> <code>moneyNumber = (decimal)realNumber;</code></td>
<td>The <code>(decimal)</code> cast operator converts the value of the <code>realNumber</code> variable to a <code>decimal</code>. The converted value is assigned to the <code>moneyNumber</code> variable.</td>
</tr>
</tbody>
</table>

When you use a cast operator, you are essentially telling the compiler that you know what you are doing and you are willing to accept the consequences of the conversion. It is still possible that a loss of data can occur. For example, look at the following code sample:

```java
int wholeNumber;
double realNumber = 8.9;
wholeNumber = (int)realNumber;
```

In this example, the `double` variable contains a fractional number. When the cast operator converts the fractional number to an `int`, the part of the number that appears after the decimal point is dropped. The process of dropping a number's fractional part is called **truncation**. After this code executes, the `wholeNumber` variable contains the value 8.

It's important to realize that when a cast operator is applied to a variable, it does not change the contents of the variable. The cast operator merely returns the value that is stored in the variable, converted to the specified data type. In the previous code sample, when the `(int)` cast operator is applied to the `realNumber` variable, the cast operator returns the value 8. The `realNumber` variable remains unchanged, however, still containing the value 8.9.
Checkpoint

3.14 Specify the appropriate primitive numeric data type to use for each of the following values.
   a. 24 dollars
   b. 12 bananas
   c. 14.5 inches
   d. 83 cents
   e. 2 concert tickets

3.15 Which of the following variable declarations will cause an error? Why?
   a. decimal payRate = 24m;
   b. int playerScore = 1340.5;
   c. double boxWidth = 205.25;
   d. string lastName = "Holm";

3.16 Write a programming statement that will convert the following decimal variable to an int and store the result in an int variable named dollars:
   decimal deposit = 976.54m;

3.17 What value will the wholePieces variable contain after the following code executes?
   double totalPieces = 6.5;
   int wholePieces = (int)totalPieces;

3.4 Performing Calculations

CONCEPT: You can use math operators to perform simple calculations. Math expressions can be written using the math operators and parentheses as grouping symbols. The result of a math expression can be assigned to a variable.

Most programs require calculations of some sort to be performed. A programmer’s tools for performing calculations are math operators. C# provides the math operators shown in Table 3-4.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name of the Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>Adds two numbers</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>Subtracts one number from another</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>Multiplies one number by another</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>Divides one number by another and gives the quotient</td>
</tr>
<tr>
<td>%</td>
<td>Modulus</td>
<td>Divides one integer by another and gives the remainder</td>
</tr>
</tbody>
</table>

Programmers use the operators shown in Table 3-4 to create math expressions. A math expression performs a calculation and gives a value. The following is an example of a simple math expression:

$$12 \times 2$$
The values on the right and left of the * operator are called operands. These are values that the * operator multiplies together. The value that is given by this expression is 24.

Variables may also be used in a math expression. For example, suppose we have two variables named hoursWorked and payRate. The following math expression uses the * operator to multiply the value in the hoursWorked variable by the value in the payRate variable:

```
hoursWorked * payRate
```

When we use a math expression to calculate a value, we have to do something with the value. Normally we want to save the value in memory so we can use it again in the program. We do this with an assignment statement. For example, suppose we have another variable named grossPay. The following statement assigns the value hoursWorked times payRate to the grossPay variable:

```
grossPay = hoursWorked * payRate;
```

Here are some other examples of statements that assign the result of a math expression to a variable:

```
total = price + tax;
sale = price - discount;
commission = sales * percent;
half = number / 2;
```

The modulus operator (%) performs division between two integers, but instead of returning the quotient, it returns the remainder. The following statement assigns 2 to leftover:

```
leftOver = 17 % 3;
```

This statement assigns 2 to leftover because 17 divided by 3 is 5 with a remainder of 2. You will not use the modulus operator frequently, but it is useful in some situations. It is commonly used in calculations that detect odd or even numbers, determine the day of the week, or measure the passage of time and in other specialized operations.

**The Order of Operations**

You can write mathematical expressions with several operators. The following statement assigns the sum of 17, the variable x, 21, and the variable y to the variable answer.

```
answer = 17 + x + 21 + y;
```

Some expressions are not that straightforward, however. Consider the following statement:

```
outcome = 12 + 6 / 3;
```

What value will be stored in outcome? The number 6 is used as an operand for both the addition and division operators. The outcome variable could be assigned either 6 or 14, depending on when the division takes place. The answer is 14 because the order of operations dictates that the division operator works before the addition operator does.

The order of operations can be summarized as follows:

1. Perform any operations that are enclosed in parentheses.
2. Perform any multiplications, divisions, or modulus operations as they appear from left to right.
3. Perform any additions or subtractions as they appear from left to right.
Mathematical expressions are evaluated from left to right. Multiplication and division are always performed before addition and subtraction, so the statement

\[
\text{outcome} = 12 + 6 / 3;
\]

works like this:

1. 6 is divided by 3, yielding a result of 2.
2. 12 is added to 2, yielding a result of 14.

It could be diagrammed as shown in Figure 3-11.

Table 3-5 shows some other sample expressions with their values.

**Figure 3-11** The order of operations at work

```
outcome = 12 + 6 / 3;
outcome = 12 + 2;
outcome = 14;
```

**Table 3-5** Some math expressions and their values

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(5 + 2 \times 4)</td>
<td>13</td>
</tr>
<tr>
<td>(10 / 2 - 3)</td>
<td>2</td>
</tr>
<tr>
<td>(8 + 12 \times 2 - 4)</td>
<td>28</td>
</tr>
<tr>
<td>(6 - 3 \times 2 + 7 - 1)</td>
<td>6</td>
</tr>
</tbody>
</table>

**Grouping with Parentheses**

Parts of a mathematical expression may be grouped with parentheses to force some operations to be performed before others. In the following statement, the variables \(a\) and \(b\) are added together, and their sum is divided by 4:

\[
\text{result} = (a + b) / 4;
\]

But what if we left the parentheses out, as shown here?

\[
\text{result} = a + b / 4;
\]

We would get a different result. Without the parentheses, \(b\) would be divided by 4 and the result added to \(a\). Table 3-6 shows some math expressions that use parentheses and their values.

**Table 3-6** More expressions and their values

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>((5 + 2) \times 4)</td>
<td>28</td>
</tr>
<tr>
<td>(10 / (5 - 3))</td>
<td>5</td>
</tr>
<tr>
<td>(8 + 12 \times (6 - 2))</td>
<td>56</td>
</tr>
<tr>
<td>((6 - 3) \times (2 + 7) / 3)</td>
<td>9</td>
</tr>
</tbody>
</table>
Mixing Data Types in a Math Expression

When you perform a math operation on two operands, the data type of the result will depend on the data type of the operands. If the operands are of the same data type, the result will also be of that data type. For example:

- When an operation is performed on two \texttt{int} values, the result will be an \texttt{int}.
- When an operation is performed on two \texttt{double} values, the result will be a \texttt{double}.
- When an operation is performed on two \texttt{decimal} values, the result will be a \texttt{decimal}.

It's not uncommon, however, for a math expression to have operands of different data types. C# handles operations involving \texttt{int}, \texttt{double}, and \texttt{decimal} operands in the following ways:

- When a math expression involves an \texttt{int} and a \texttt{double}, the \texttt{int} is temporarily converted to a \texttt{double}, and the result is a \texttt{double}.
- When a math expression involves an \texttt{int} and a \texttt{decimal}, the \texttt{int} is temporarily converted to a \texttt{decimal}, and the result is a \texttt{decimal}.
- Math expressions involving a \texttt{double} and a \texttt{decimal} are not allowed unless a cast operator is used to convert one of the operands.

For example, suppose a pay-calculating program has the following variable declarations:

\begin{verbatim}
int hoursWorked; // To hold the number of hours worked
decimal payRate; // To hold the hourly pay rate
decimal grossPay; // To hold the gross pay
\end{verbatim}

Then, later in the program this statement appears:

\begin{verbatim}
grossPay = hoursWorked \times payRate;
\end{verbatim}

The math expression on the right side of the \texttt{=} operator multiplies an \texttt{int} by a \texttt{decimal}. When the statement executes, the value of the \texttt{hoursWorked} variable is temporarily converted to a \texttt{decimal} and then multiplied by the \texttt{payRate} variable. The result is a \texttt{decimal} and is assigned to the \texttt{grossPay} variable.

When possible, you should avoid math operations that use a mixture of \texttt{double} and \texttt{decimal} operands. C# does not allow operations involving these two types unless you use a cast operator to explicitly convert one of the operands. For example, suppose a program that calculates the cost of a product has the following variable declarations:

\begin{verbatim}
double weight;         // The product weight
decimal pricePerPound; // The price per pound
decimal total;         // The total cost
\end{verbatim}

Later in the program you need to calculate the total cost, like this:

\begin{verbatim}
total = weight \times pricePerPound; \leftarrow \text{ERROR!}
\end{verbatim}

The compiler will not allow this statement because \texttt{weight} is a \texttt{double} and \texttt{pricePerPound} is a \texttt{decimal}. To fix the statement, you can insert a cast operator, as shown here:

\begin{verbatim}
total = (decimal)weight \times pricePerPound;
\end{verbatim}

The cast operator converts the value of the \texttt{weight} variable to a \texttt{decimal}, and the converted value is multiplied by \texttt{pricePerPound}. The result of the expression is a \texttt{decimal} and is assigned to \texttt{total}. 
**Integer Division**

When you divide an integer by an integer in C#, the result is always given as an integer. If the result has a fractional part, it is truncated. For example, look at the following code:

```csharp
int length;        // Declare length as an int
double half;       // Declare half as a double
length = 75;       // Assign 75 to length
half = length / 2; // Calculate half the length
```

The last statement divides the value of `length` by 2 and assigns the result to `half`. Mathematically, the result of 75 divided by 2 is 37.5. However, that is not the result that we get from the math expression. The `length` variable is an `int`, and it is being divided by the numeric literal 2, which is also treated as an `int`. The result of the division is truncated, giving the value 37. This is the value that is assigned to the `half` variable. It does not matter that the `half` variable is declared as a `double`. The fractional part of the result is truncated before the assignment takes place.

**Combined Assignment Operators**

Sometimes you want to increase a variable’s value by a certain amount. For example, suppose you have a variable named `number` and you want to increase its value by 1. You can accomplish that with the following statement:

```csharp
number = number + 1;
```

The expression on the right side of the assignment operator calculates the value of `number` plus 1. The result is then assigned to `number`, replacing the value that was previously stored there. This statement effectively adds 1 to `number`. For example, if `number` is equal to 6 before this statement executes, it is equal to 7 after the statement executes.

Similarly, the following statement subtracts 5 from `number`:

```csharp
number = number - 5;
```

If `number` is equal to 15 before this statement executes, it is equal to 10 after the statement executes. Here’s another example. The following statement doubles the value of the `number` variable:

```csharp
number = number * 2;
```

If `number` is equal to 4 before this statement executes, it is equal to 8 after the statement executes.

These types of operations are very common in programming. For convenience, C# offers a special set of operators known as **combined assignment operators** that are designed specifically for these jobs. Table 3-7 shows the combined assignment operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example Usage</th>
<th>Equivalence</th>
</tr>
</thead>
<tbody>
<tr>
<td>+=</td>
<td><code>x += 5;</code></td>
<td><code>x = x + 5;</code></td>
</tr>
<tr>
<td>-=</td>
<td><code>y -= 2;</code></td>
<td><code>y = y - 2;</code></td>
</tr>
<tr>
<td>*=</td>
<td><code>z *= 10;</code></td>
<td><code>z = z * 10;</code></td>
</tr>
<tr>
<td>/=</td>
<td><code>a /= b;</code></td>
<td><code>a = a / b;</code></td>
</tr>
<tr>
<td>%=</td>
<td><code>c %= 3;</code></td>
<td><code>c = c % 3;</code></td>
</tr>
</tbody>
</table>
As you can see, the combined assignment operators do not require the programmer to type the variable name twice. Also, they give a clear indication of what is happening in the statement.

**Checkpoint**

3.18 List the operands for the following math expression.

    length * width

3.19 Summarize the mathematical order of operations.

3.20 Rewrite the following code segment so that it does not cause an error.

```csharp
decimal pricePerFoot = 2.99m;
double boardLength = 10.5;
decimal totalCost = boardLength * pricePerFoot;
```

3.21 Assume `result` is a double variable. When the following statement executes, what value will be stored in `result`?

```csharp
result = 4 + 10 / 2;
```

3.22 Assume `result` is an int variable. When the following statement executes, what value will be stored in `result`?

```csharp
result = (2 + 5) * 10;
```

3.23 Assume `result` is a double variable. When the following statement executes, what value will be stored in `result`?

```csharp
result = 5 / 2;
```

3.24 Rewrite the following statements using combined assignment operators:

a. count = count + 1;
b. amount = amount – 5;
c. radius = radius * 10;
d. length = length / 2;

### 3.5 Inputting and Outputting Numeric Values

**CONCEPT:** If the user has entered a number into a TextBox, the number will be stored as a string in the TextBox’s Text property. If you want to store that number in a numeric variable, you have to convert it to the appropriate numeric data type. When you want to display the value of a numeric variable in a Label control or a message box, you have to convert it to a string.

**Getting a Number from a TextBox**

GUI applications typically use TextBox controls to read keyboard input. Any data that the user enters into a TextBox control is stored in the control’s Text property as a string, even if it is a number. For example, if the user enters the number 72 into a TextBox control, the input is stored as the string “72” in the control’s Text property.

If the user has entered a numeric value into a TextBox control and you want to assign that value to a numeric variable, you have to convert the control’s Text property to the desired numeric data type. Unfortunately, you cannot use a cast operator to convert a string to a numeric type.
To convert a string to any of the numeric data types, we use a family of methods in the .NET Framework known as the Parse methods. In computer science, the term parse typically means to analyze a string of characters for some purpose. The Parse methods are used to convert a string to a specific data type. There are several Parse methods in the .NET Framework, but because we are primarily using the int, double, and decimal numeric data types, we need only three of them:

- We use the int.Parse method to convert a string to an int.
- We use the double.Parse method to convert a string to a double.
- We use the decimal.Parse method to convert a string to a decimal.

When you call one of the Parse methods, you pass a piece of data known as an argument into the method, and the method returns a piece of data back to you. The argument that you pass to the method is the string that you want to convert, and the piece of data that the method returns back to you is the converted value. Figure 3-12 illustrates this concept using the int.Parse method as an example.

The following code sample shows how to use the int.Parse method to convert a control’s Text property to an int. Assume that hoursWorkedTextBox is the name of a TextBox control.

```csharp
1 // Declare an int variable to hold the hours worked.
2 int hoursWorked;
3
4 // Get the hours worked from the TextBox.
5 hoursWorked = int.Parse(hoursWorkedTextBox.Text);
```

Let’s assume that the user has entered the value 40 into the hoursWorkedTextBox control. In line 5 of the code sample, on the right side of the = operator is the expression int.Parse(hoursWorkedTextBox.Text). This expression calls the int.Parse method, passing the value of hoursWorkedTextBox.Text as an argument. Because the user has entered 40 into the TextBox, the string "40" is the value that is passed as the argument. The method converts the string "40" to the int value 40. The int value 40 is returned from the method and the = operator assigns it to the hoursWorked variable. Figure 3-13 illustrates this process.

The following code sample demonstrates the double.Parse method. Assume that temperatureTextBox is the name of a TextBox control.

```csharp
1 // Declare a double variable to hold the temperature.
2 double temperature;
3
4 // Get the temperature from the TextBox.
5 temperature = double.Parse(temperatureTextBox.Text);
```

Line 5 passes temperatureTextBox.Text as an argument to the double.Parse method. That value is converted to a double, returned from the double.Parse method, and assigned to the temperature variable.

The following code sample demonstrates the decimal.Parse method. Assume that moneyTextBox is the name of a TextBox control.
### 3.5 Inputting and Outputting Numeric Values

#### Figure 3-13 Converting TextBox input to an int

The user enters 40 into the hoursWorkedTextBox control.

```csharp
// Declare a decimal variable to hold an amount of money.
decimal money;

// Get an amount from the TextBox.
money = decimal.Parse(moneyTextBox.Text);
```

The int value 40 is returned from the int.Parse method and assigned to the hoursWorked variable.

1 // Declare a decimal variable to hold an amount of money.
2 decimal money;
3
4 // Get an amount from the TextBox.
5 money = decimal.Parse(moneyTextBox.Text);

Line 5 passes moneyTextBox.Text as an argument to the decimal.Parse method. That value is converted to a decimal, returned from the decimal.Parse method, and assigned to the money variable.

**NOTE:** If you look at the top of a form's source code in the code editor, you should see a directive that reads `using System;`. That directive is required for any program that uses the Parse methods.

### Invalid Conversions

The Parse methods work only if the string that is being converted contains a valid numeric value. If the string contains invalid characters or contains a number that cannot be converted to the specified data type, an error known as an exception occurs. An exception is an unexpected error that occurs while a program is running, causing the program to halt if the error is not properly dealt with.

For example, assume that hoursWorked is an int variable and hoursWorkedTextBox is a TextBox control. Suppose the user has entered `xyz` into the TextBox and the following statement executes:

```csharp
hoursWorked = int.Parse(hoursWorkedTextBox.Text);
```

Obviously, the string "xyz" cannot be converted to an int, so an exception occurs. (When an exception occurs, programmers say an exception is “thrown.”) Depending on how you execute the application, you will see one of the windows displayed in Figure 3-14 or Figure 3-15.

- If you see the window in Figure 3-14, you can stop the application by clicking the Stop Debugging button ( ), or by pressing `Shift + F5`, or by clicking Debug and then Stop Debugging.
- When you see the window shown in Figure 3-15, in most situations you should click the Quit button to stop the application.
Later in this chapter you will learn how to catch errors like this and prevent the program from halting.

**Displaying Numeric Values**

Suppose an application has a `decimal` variable named `grossPay` and a Label control named `grossPayLabel`. You want to display the variable’s value in the Label control. To accomplish this, you must somehow get the value of the `grossPay` variable into the `grossPayLabel` control’s Text property. The following assignment statement will not work, however:

```csharp
grossPayLabel.Text = grossPay;
```

**ERROR!**

You cannot assign a numeric value to a control’s Text property because only strings can be assigned to the Text property. If you want to display the value of a numeric variable in a Label control, you have to convert the variable’s value to a string.

Luckily, all variables have a `ToString` method that you can call to convert the variable’s value to a string. You call the `ToString` method using the following general format:

```csharp
variableName.ToString()
```

In the general format, `variableName` is the name of any variable. The expression returns the variable’s value as a string. Here is a code sample that demonstrates:

```csharp
decimal grossPay = 1550.0m;
grossPayLabel.Text = grossPay.ToString();
```

The first statement declares a `decimal` variable named `grossPay` initialized with the value 1,550.0. In the second statement, the expression on the right side of the `=` operator calls the `grossPay` variable’s `ToString` method. The method returns the string
"1550.0". The = operator then assigns the string "1550.0" to the grossPayLabel control’s Text property. As a result, the value 1550.0 is displayed in the grossPayLabel control. This process is illustrated in Figure 3-16.

**Figure 3-16 Displaying numeric data in a Label control**

You must also convert a numeric variable to a string before passing it to the MessageBox.Show method. The following example shows how an int variable’s value can be converted to a string and displayed in a message box:

```csharp
int myNumber = 123;
MessageBox.Show(myNumber.ToString());
```

The first statement declares an int variable named myNumber, initialized with the value 123. In the second statement the following takes place:

- The myNumber variable’s ToString method is called. The method returns the string "123".
- The string "123" is passed to the MessageBox.Show method. As a result, the value 123 is displayed in a message box.

**Implicit String Conversion with the + Operator**

In this chapter you’ve learned that the + operator has two uses: string concatenation and numeric addition. If you write an expression using the + operator and both operands are strings, the + operator concatenates the strings. If both operands are numbers of compatible types, then the + operator adds the two numbers. But what happens if one operand is a string and the other operand is a number? The number will be implicitly converted to a string, and both operands will be concatenated. Here’s an example:

```csharp
int idNumber = 1044;
string output = "Your ID number is " + idNumber;
```

In the second statement, the string variable output is initialized with the string "Your ID number is 1044". Here is another example:

```csharp
double testScore = 88.5;
MessageBox.Show("Your test score is " + testScore);
```

The second statement displays a message box showing the string "Your test score is 88.5".

In Tutorial 3-2 you will use some of the techniques discussed in this section. You will create an application that reads numeric input from TextBox controls, and displays numeric output in a Label control.
Chapter 3 Processing Data

Tutorial 3-2: Calculating Fuel Economy

In the United States, a car’s fuel economy is measured in miles per gallon, or MPG. You use the following formula to calculate a car’s MPG:

\[
MPG = \frac{\text{Miles driven}}{\text{Gallons of gas used}}
\]

In this tutorial you will create an application that lets the user enter the number of miles he or she has driven and the gallons of gas used. The application will calculate and display the car’s MPG.

Figure 3-17 shows the application’s form, with the names of all the controls. When the application runs, the user enter the number of miles driven into the milesTextBox control and the gallons of gas used into the gallonsTextBox control. When the user clicks the calculateButton control, the application calculates the car’s MPG and displays the result in the mpgLabel control. The exitButton control closes the application’s form.

Figure 3-17 The Fuel Economy form

Step 1: Start Visual Studio (or Visual Studio Express) and begin a new Windows Forms Application project named Fuel Economy.

Step 2: Set up the application’s form as shown in Figure 3-17. Notice that the form’s Text property is set to Fuel Economy. The names of the controls are shown in the figure. As you place each of the controls on the form, refer to Table 3-8 for the relevant property settings.

Step 3: Once you have set up the form with its controls, you can create the Click event handlers for the Button controls. At the end of this tutorial, Program 3-2 shows the completed code for the form. You will be instructed to refer to Program 3-2 as you write the event handlers. (Remember, the line numbers that are shown in Program 3-2 are not part of the program. They are shown for reference only.)

In the Designer, double-click the calculateButton control. This opens the code editor, and you will see an empty event handler named calculateButton_Click. Complete the calculateButton_Click event handler by typing the code shown in lines 22–38 in Program 3-2.

Let’s take a closer look at the code:

Line 22: This statement declares a double variable named miles. This variable is used to hold the number of miles driven.

Line 23: This statement declares a double variable named gallons. This variable is used to hold the number of gallons used.
3.5 Inputting and Outputting Numeric Values

**Table 3-8** Control property settings

<table>
<thead>
<tr>
<th>Control Name</th>
<th>Control Type</th>
<th>Property Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>milesPromptLabel</td>
<td>Label</td>
<td>Text: Enter the number of miles driven:</td>
</tr>
<tr>
<td>gasPromptLabel</td>
<td>Label</td>
<td>Text: Enter the gallons of gas used:</td>
</tr>
<tr>
<td>outputDescriptionLabel</td>
<td>Label</td>
<td>Text: Your car’s MPG:</td>
</tr>
<tr>
<td>milesTextBox</td>
<td>TextBox</td>
<td>No properties changed</td>
</tr>
<tr>
<td>gallonsTextBox</td>
<td>TextBox</td>
<td>No properties changed</td>
</tr>
<tr>
<td>mpgLabel</td>
<td>Label</td>
<td>AutoSize: False</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BorderStyle: FixedSingle</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Text: (The contents of the Text property</td>
</tr>
<tr>
<td></td>
<td></td>
<td>have been erased.)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TextAlign: MiddleCenter</td>
</tr>
<tr>
<td>calculateButton</td>
<td>Button</td>
<td>Text: Calculate MPG</td>
</tr>
<tr>
<td>exitButton</td>
<td>Button</td>
<td>Text: Exit</td>
</tr>
</tbody>
</table>

**Line 24:** This statement declares a double variable named mpg. This variable is used to hold the MPG, which will be calculated.

**Line 28:** This statement converts the milesTextBox control’s Text property to a double and assigns the result to the miles variable.

**Line 32:** This statement converts the gallonsTextBox control’s Text property to a double and assigns the result to the gallons variable.

**Line 35:** This statement calculates MPG. It divides the miles variable by the gallons variable and assigns the result to the mpg variable.

**Line 38:** This statement converts the mpg variable to a string and assigns the result to the mpgLabel control’s Text property. This causes the value of the mpg variable to be displayed in the mpgLabel control.

**Step 4:** Switch your view back to the Designer and double-click the exitButton control. In the code editor you will see an empty event handler named exitButton_Click. Complete the exitButton_Click event handler by typing the code shown in lines 43–44 in Program 3-2.

**Step 5:** Save the project. Then, press the **F5** key on the keyboard or click the Start Debugging button ( ) on the toolbar to compile and run the application. Test the application by entering values into the TextBoxes and clicking the Calculate MPG button. The MPG should be displayed, similar to Figure 3-18. Click the Exit button and the form should close.

**Figure 3-18** The Fuel Economy application
namespace Fuel_Economy
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void calculateButton_Click(object sender, EventArgs e)
        {
            double miles;     // To hold miles driven
            double gallons;   // To hold gallons used
            double mpg;       // To hold MPG

            // Get the miles driven and assign it to
            // the miles variable.
            miles = double.Parse(milesTextBox.Text);

            // Get the gallons used and assign it to
            // the gallons variable.
            gallons = double.Parse(gallonsTextBox.Text);

            // Calculate MPG.
            mpg = miles / gallons;

            // Display the MPG in the mpgLabel control.
            mpgLabel.Text = mpg.ToString();
        }

        private void exitButton_Click(object sender, EventArgs e)
        {
            // Close the form.
            this.Close();
        }
    }
}
Checkpoint

3.25 What method converts the string literal "40" to a value of the int data type?

3.26 Write a statement that converts each of the following string values to the decimal data type using the decimal.Parse method.
   a. "9.05"
   b. grandTotal
   c. "50"
   d. priceTextBox.Text

3.27 Suppose an application has a decimal variable named total and a Label control named totalLabel. What will be the result when the following assignment statement is executed?
   totalLabel.Text = total;

3.28 Write a statement that displays each of the following numeric variables in a message box.
   a. grandTotal
   b. highScore
   c. sum
   d. width

3.29 Write a statement that will store the value of an int variable named result in the Text property of a Label control named resultLabel.

---

3.6 Formatting Numbers with the ToString Method

CONCEPT: The ToString method can optionally format a number to appear in a specific way.

When you display large numbers, you usually want to format them with commas so they are easy to read. For example, 487,634,789.0 is easier to read than 487634789.0. Also, when you display amounts of money, you usually want to round them to two decimal places and display a currency symbol, such as a dollar sign ($).

When you call the ToString method, you can optionally pass a formatting string as an argument to the method. The formatting string indicates that you want the number to appear formatted in a specific way when it is returned as a string from the method. For example, when you pass the formatting string "c" to the ToString method, the number is returned formatted as currency. Assuming that you are in the United States, numbers formatted as currency are preceded by a dollar sign ($), are rounded to two decimal places, and have comma separators inserted as necessary. The following code sample demonstrates:

```
decimal amount = 123456789.45678m;
MessageBox.Show(amount.ToString("c"));
```

Notice in the second statement that the "c" formatting string is passed to the amount variable's ToString method. The message box that the statement displays appears as shown in Figure 3-19.

There are several other format strings that you can use with the ToString method, and each produces a different type of formatting. Table 3-9 shows a few of them.
Chapter 3 Processing Data

Number Format
Number format ("n" or "N") displays numeric values with comma separators and a decimal point. By default, two digits display to the right of the decimal point. Negative values are displayed with a leading minus sign. An example is $-2,345.67.

Fixed-Point Format
Fixed-point format ("f" or "F") displays numeric values with no thousands separator and a decimal point. By default, two digits display to the right of the decimal point. Negative values are displayed with a leading minus (−) sign. An example is $-2345.67.

Exponential Format
Exponential format ("e" or "E") displays numeric values in scientific notation. The number is displayed with a single digit to the left of the decimal point. The letter e appears in front of the exponent, and the exponent has a leading + or − sign. By default, six digits display to the right of the decimal point, and a leading minus sign is used if the number is negative. An example is $-2.345670e+003.

Currency Format
Currency format ("c" or "C") displays a leading currency symbol (such as $), digits, comma separators, and a decimal point. By default, two digits display to the right of the decimal point. Negative values are surrounded by parentheses. An example of a negative value is ($2,345.67).

Using Percent Format
Percent format ("p" or "P") causes the number to be multiplied by 100 and displayed with a trailing space and % sign. By default, two digits display to the right of the decimal point.
3.6 Formatting Numbers with the **ToString** Method

Point. Negative values are displayed with a leading minus sign. For example, the number 0.125 would be formatted as 12.5 % and the number −0.2345 would be formatted as −23.45 %.

**Specifying the Precision**

Each numeric format string can optionally be followed by an integer that indicates how many digits to display after the decimal point. For example, the format "n3" displays three digits after the decimal point. Table 3-10 shows a variety of numeric formatting examples, based on the North American locale.

**Table 3-10** Numeric formatting examples (North American locale)

<table>
<thead>
<tr>
<th>Number</th>
<th>Format String</th>
<th><strong>ToString( )</strong> Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.3</td>
<td>&quot;n3&quot;</td>
<td>12.300</td>
</tr>
<tr>
<td>12.348</td>
<td>&quot;n2&quot;</td>
<td>12.35</td>
</tr>
<tr>
<td>1234567.1</td>
<td>&quot;N&quot;</td>
<td>1,234,567.10</td>
</tr>
<tr>
<td>123456.0</td>
<td>&quot;f2&quot;</td>
<td>123456.00</td>
</tr>
<tr>
<td>123456.0</td>
<td>&quot;e3&quot;</td>
<td>1.235e+005</td>
</tr>
<tr>
<td>.234</td>
<td>&quot;P&quot;</td>
<td>23.40 %</td>
</tr>
<tr>
<td>−1234567.8</td>
<td>&quot;C&quot;</td>
<td>($1,234,567.80)</td>
</tr>
</tbody>
</table>

**Rounding**

Rounding can occur when the number of digits you have specified after the decimal point in the format string is smaller than the precision of the numeric value. Suppose, for example, that the value 1.235 were displayed with a format string of "n2". Then the displayed value would be 1.24. If the next digit after the last displayed digit is 5 or higher, the last displayed digit is rounded away from zero. Table 3-11 shows examples of rounding using a format string of "n2".

**Table 3-11** Rounding examples using the "n2" display format string

<table>
<thead>
<tr>
<th>Number</th>
<th>Formatted As</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.234</td>
<td>1.23</td>
</tr>
<tr>
<td>1.235</td>
<td>1.24</td>
</tr>
<tr>
<td>1.238</td>
<td>1.24</td>
</tr>
<tr>
<td>−1.234</td>
<td>−1.23</td>
</tr>
<tr>
<td>−1.235</td>
<td>−1.24</td>
</tr>
<tr>
<td>−1.238</td>
<td>−1.24</td>
</tr>
</tbody>
</table>

**Using Leading Zeros with Integer Values**

You can use the "d" or "D" formatting strings with integers to specify the minimum width for displaying the number. Leading zeros are inserted if necessary. Table 3-12 shows examples.

In Tutorial 3-3 you will create an application that uses currency formatting to display a dollar amount.
### Tutorial 3-3:
Creating the *Sale Price Calculator* Application with Currency Formatting

If you are writing a program that works with a percentage, you have to make sure that the percentage’s decimal point is in the correct location before doing any math with the percentage. This is especially true when the user enters a percentage as input. Most users will enter the number 50 to mean 50 percent, 20 to mean 20 percent, and so forth. Before you perform any calculations with such a percentage, you have to divide it by 100 to move its decimal point to the left two places.

Suppose a retail business is planning to have a storewide sale in which the prices of all items will be reduced by a specified percentage. In this tutorial you will create an application to calculate the sale price of an item after the discount is subtracted. Here is the algorithm, expressed as pseudocode:

1. Get the original price of the item.
2. Get the discount percentage. (For example, 20 is entered for 20 percent.)
3. Divide the percentage amount by 100 to move the decimal point to the correct location.
4. Multiply the percentage by the original price. This is the amount of the discount.
5. Subtract the discount from the original price. This is the sale price.
6. Display the sale price.

Figure 3-20 shows the application’s form, with the names of all the controls. When the application runs, the user enters an item’s original price into the `originalPriceTextBox` control and the discount percentage into the `discountPercentageTextBox` control. When the user clicks the `calculateButton` control, the application calculates the item’s sale price and displays the result in the `salePriceLabel` control. The `exitButton` control closes the application’s form.
Step 1: Start Visual Studio (or Visual Studio Express) and begin a new Windows Forms Application project named *Sale Price Calculator*.

Step 2: Set up the application’s form, as shown in Figure 3-20. Notice that the form’s Text property is set to *Sale Price Calculator*. The names of the controls are shown in the figure. As you place each of the controls on the form, refer to Table 3-13 for the relevant property settings.

<table>
<thead>
<tr>
<th>Control Name</th>
<th>Control Type</th>
<th>Property Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>originalPricePromptLabel</td>
<td>Label</td>
<td>Text: Enter the item’s original price:</td>
</tr>
<tr>
<td>discPercentagePromptLabel</td>
<td>Label</td>
<td>Text: Enter the discount percentage:</td>
</tr>
<tr>
<td>outputDescriptionLabel</td>
<td>Label</td>
<td>Text: Sale price:</td>
</tr>
<tr>
<td>originalPriceTextBox</td>
<td>TextBox</td>
<td>No properties changed</td>
</tr>
<tr>
<td>discountPercentageTextBox</td>
<td>TextBox</td>
<td>No properties changed</td>
</tr>
<tr>
<td>salePriceLabel</td>
<td>Label</td>
<td>AutoSize: False</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BorderStyle: FixedSingle</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Text: (The contents of the Text</td>
</tr>
<tr>
<td></td>
<td></td>
<td>property have been erased.)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TextAlign: MiddleCenter</td>
</tr>
<tr>
<td>calculateButton</td>
<td>Button</td>
<td>Text: Calculate Sale Price</td>
</tr>
<tr>
<td>exitButton</td>
<td>Button</td>
<td>Text: Exit</td>
</tr>
</tbody>
</table>

Step 3: Once you have set up the form with its controls, you can create the Click event handlers for the Button controls. At the end of this tutorial, Program 3-3 shows the completed code for the form. You will be instructed to refer to Program 3-3 as you write the event handlers. (Remember, the line numbers that are shown in Program 3-3 are not part of the program. They are shown for reference only.)

In the Designer, double-click the calculateButton control. This will open the code editor, and you will see an empty event handler named calculateButton_Click. Complete the calculateButton_Click event handler by typing the code shown in lines 22–43 in Program 3-3.

Let’s take a closer look at the code:

Line 22: This statement declares a decimal variable named originalPrice. This variable will hold the item’s original price.

Line 23: This statement declares a decimal variable named discountPercentage. This variable will hold the discount percentage.

Line 24: This statement declares a decimal variable named discountAmount. This variable will hold the amount of discount that will be taken from the item’s original price. This amount will be calculated.

Line 25: This statement declares a decimal variable named salePrice. This variable will hold the item’s sale price. This amount will be calculated.

Line 28: This statement converts the originalPriceTextBox control’s Text property to a decimal and assigns the result to the originalPrice variable.
Line 31: This statement converts the discountPercentageTextBox control’s Text property to a decimal and assigns the result to the discountPercentage variable.

Line 34: This statement divides discountPercentage by 100 and stores the result back in discountPercentage. This moves the decimal point in the discountPercentage variable to the left two places.

Line 37: This statement calculates the amount of the discount. It multiplies originalPrice by discountPercentage and assigns the result to discountAmount.

Line 40: This statement calculates the item’s sale price. It subtracts the discountAmount variable from the originalPrice variable and assigns the result to the salePrice variable.

Line 43: This statement displays the item’s sale price as a currency amount. It converts the salePrice variable to a string and assigns the result to the salePriceLabel control’s Text property. Notice that the format string “c” is passed to the salePrice variable’s ToString method.

Step 4: Switch your view back to the Designer and double-click the exitButton control. In the code editor you will see an empty event handler named exitButton_Click. Complete the exitButton_Click event handler by typing the code shown in lines 48–49 in Program 3-3.

Step 5: Save the project. Then, press the F5 key on the keyboard or click the Start Debugging button ( ) on the toolbar to compile and run the application. Test the application by entering values into the TextBoxes and clicking the Calculate Sale Price button. The sale price is displayed, similar to Figure 3-21. Click the Exit button and the form closes.

Figure 3-21 The Sale Price Calculator application

Program 3-3 Completed Form1 code for the Sale Price Calculator application

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Sale_Price_Calculator
{
    // Code for the Sale Price Calculator application
}
```
3.6 Formatting Numbers with the ToString Method

Checkpoint

3.30 Write a programming statement that displays the string value of a variable named salary in a message box using currency format.

3.31 The following variable names give an indication of the data each stores. For each variable, specify the format string that you think is most appropriate.

   a. discountPercentage
   b. atomicWeight
   c. retailPrice
   d. quantityPurchased
   e. degreesKelvin

```csharp
class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }

    private void calculateButton_Click(object sender, EventArgs e)
    {
        decimal originalPrice;       // The item's original price
        decimal discountPercentage;  // The discount percentage
        decimal discountAmount;     // The amount of discount
        decimal salePrice;          // The item's sale price

        // Get the item's original price.
        originalPrice = decimal.Parse(originalPriceTextBox.Text);

        // Get the discount percentage.
        discountPercentage = decimal.Parse(discountPercentageTextBox.Text);

        // Move the percentage's decimal point left two spaces.
        discountPercentage = discountPercentage / 100;

        // Calculate the amount of the discount.
        discountAmount = originalPrice * discountPercentage;

        // Calculate the sale price.
        salePrice = originalPrice - discountAmount;

        // Display the sale price.
        salePriceLabel.Text = salePrice.ToString("c");
    }

    private void exitButton_Click(object sender, EventArgs e)
    {
        // Close the form.
        this.Close();
    }
}
```
3.32 What value will be displayed in the message box when the following code segment is executed?
```csharp
double apples = 12.0;
MessageBox.Show(apples.ToString("n0"));
```

3.33 Examine the following integer variables and specify the number of leading zeros to use with the `d` or `D` format strings so that all the numbers are equal in width.
```csharp
int valueA = 234, valueB = 56, valueC = 7, valueD = 89123;
```

3.34 Write a programming statement that uses the `ToString` method of a variable named `millimeters` so that it displays a precision of four digits after the decimal point in fixed-point scientific format.

### 3.7 Simple Exception Handling

**CONCEPT:** An exception is an unexpected error that happens while a program is running. If an exception is not handled by the program, the program will abruptly halt.

An exception is an unexpected error that occurs while a program is running, causing the program to halt if the error is not properly dealt with. Exceptions are usually caused by circumstances that are beyond the programmer’s control. For example, suppose the user has entered a value into a TextBox, and that value is expected to be a numeric data type, but the string contains invalid characters and it cannot be converted. The `Parse` method cannot continue, so an exception occurs. (To use the proper terminology, we say that a method throws an exception when an unexpected error occurs and it cannot continue operating.)

Let’s look at an example. If you have completed the Fuel Economy project from Tutorial 3-2, open it in Visual Studio and then either click the Start Debugging button or press `F5` to run the application. On the application’s form, enter 300 for the number of miles driven and then enter a nonnumeric sequence of characters for the gallons of gas used. Figure 3-22 shows an example where the user has entered `wxyz`. Then, click the Calculate MPG button. Because the invalid string that you entered for the gallons of gas cannot be converted to a `double`, an exception is thrown. The application stops running and Visual Studio goes into a special mode known as break mode. The window shown in Figure 3-23 is displayed, and the line of code that was executing when the exception was thrown is highlighted.

**Figure 3-22 Invalid data entered for gallons**
The exception window that is shown in Figure 3-23 displays a lot of information, but if you look at the line just below the window’s title bar, you see the message Input string was not in a correct format. That is a description of what happened to cause the exception. To get out of break mode, click the Stop Debugging button ( ), or press Shift + F5. Visual Studio then returns to its normal mode.

Handling Exceptions

C#, like most modern programming languages, allows you to write code that responds to exceptions when they are thrown and prevents the program from abruptly crashing. Such code is called an exception handler, and is written with the try-catch statement. There are several ways to write a try-catch statement, but the following general format is the simplest variation:

```csharp
try
{
    statement;
    statement;
    etc.
}
catch
{
    statement;
    statement;
    etc.
}
```

First the key word try appears, followed by a group of one or more statements that appears inside a set of braces. This group of statements is known as a try block. One or more of the statements inside the try block can potentially throw an exception.

After the try block, a catch clause appears. The catch clause is followed by a group of one or more statements enclosed inside a set of braces. This group of statements is known as a catch block.

When a try-catch statement executes, the statements in the try block are executed in the order that they appear. If a statement in the try block throws an exception, the program immediately jumps to the catch clause and executes the statements in the catch block. If all the statements in the try block execute with no exception, the catch block is skipped.
Let’s see how a try-catch statement can be used in the Fuel Economy application. Here is a modified version of the application’s calculateButton_Click event handler:

```csharp
private void calculateButton_Click(object sender, EventArgs e)
{
    try
    {
        double miles;   // To hold miles driven
        double gallons; // To hold gallons used
        double mpg;     // To hold MPG

        // Get the miles driven and assign it to
        // the miles variable.
        miles = double.Parse(milesTextBox.Text);

        // Get the gallons used and assign it to
        // the gallons variable.
        gallons = double.Parse(gallonsTextBox.Text);

        // Calculate MPG.
        mpg = miles / gallons;

        // Display the MPG in the mpgLabel control.
        mpgLabel.Text = mpg.ToString();
    }
    catch
    {
        // Display an error message.
        MessageBox.Show("Invalid data was entered.");
    }
}
```

When you write a try-catch statement, you put all the code that might throw an exception inside the try block. In this version of the event handler, the try block appears in lines 5–21. (In this example, we have put all the statements that previously appeared in the event handler inside the try block.) If any statement inside the try block throws an exception, the program will immediately jump to the catch clause in line 23. Then, the statements in the catch block (lines 25–26) will execute.

Let’s say that the application is running and the user enters invalid input into the milesTextBox control. When the event handler executes, the statement in line 11 throws an exception because the double.Parse method is not able to convert the control’s Text property to a double. The program will immediately jump to the catch clause in line 23 and then execute the statements inside the catch block. Line 26 displays a message box with an error message. Figure 3-24 illustrates this process.

On the other hand, if all the statements inside the try block execute and no exceptions are thrown, the catch block will be skipped.

**Displaying an Exception’s Default Error Message**

When an exception is thrown, an object known as an exception object is created in memory. The exception object has various properties that contain data about the exception. When you write a catch clause, you can optionally assign a name to the exception object, as shown here:

```csharp
    catch (Exception ex)
```

The expression that appears inside the parentheses specifies that we are assigning the name `ex` to the exception object. (There is nothing special about the name `ex`. That is simply the name that we’ve chosen for the examples. You can use any name that you wish.)
Inside the catch block, we can use the name `ex` to access the exception object’s properties. One of these is the `Message` property, which contains the exception’s default error message. The following code shows how this can be done. This is another modification of the `Fuel Economy` project’s `calculateButton_Click` event handler.

```csharp
private void calculateButton_Click(object sender, EventArgs e) {
    try {
        double miles;   // To hold miles driven
        double gallons;  // To hold gallons used
        double mpg;     // To hold MPG

        // Get the miles driven and assign it to
        // the miles variable.
        miles = double.Parse(milesTextBox.Text);

        // Get the gallons used and assign it to
        // the gallons variable.
        gallons = double.Parse(gallonsTextBox.Text);

        // Calculate MPG.
        mpg = miles / gallons;

        // Display the MPG in the mpgLabel control.
        mpgLabel.Text = mpg.ToString();
    }
    catch (Exception ex) {
        // Display the default error message.
        MessageBox.Show(ex.Message);
    }
}
```

Figure 3-24 Handling an exception

The program jumps to the catch clause and executes the statements in the catch block.

If this statement throws an exception....

The program jumps to the catch clause and executes the statements in the catch block.
The statement in line 26 simply passes the exception object’s Message property to the `MessageBox.Show` method. This causes the default error message to be displayed in a message box. Figure 3-25 shows an example of the message that is displayed as a result of the user entering invalid input for either the `milesTextBox` or the `gallonsTextBox` controls.

**Figure 3-25** A message box showing an exception’s default error message

![Message box showing an exception's default error message](image)

In Tutorial 3-4 you create an application that uses a `try-catch` statement to handle exceptions that are thrown when the user enters invalid data into a TextBox control.

---

**Tutorial 3-4:**

**Creating the Test Average Application with Exception Handling**

Determining the average of a group of values is a simple calculation: You add all the values and then divide the sum by the number of values. Although this is a straightforward calculation, it is easy to make a mistake when writing a program that calculates an average. For example, let’s assume that the variables `a`, `b`, and `c` each hold a value and we want to calculate the average of those values. If we are careless, we might write a statement such as the following to perform the calculation:

```csharp
average = a + b + c / 3.0;
```

Can you see the error in this statement? When it executes, the division will take place first. The value in `c` will be divided by 3, and then the result will be added to `a + b`. That is not the correct way to calculate an average. To correct this error we need to put parentheses around `a + b + c`, as shown here:

```csharp
average = (a + b + c) / 3.0;
```

In this tutorial you will create an application that calculates the average of three test scores. Figure 3-26 shows the application’s form, with the names of all the controls. When

**Figure 3-26** The Test Average form

![Test Average form](image)
the application runs, the user will enter the test scores into the TextBox controls. When
the user clicks the calculateButton control, the application will calculate the average
test score and display the result in the averageLabel control. The clearButton control
will clear the contents of the TextBoxes and the averageLabel control. The exitButton
control closes the application’s form.

Step 1: Start Visual Studio (or Visual Studio Express) and begin a new Windows Forms
Application project named Test Average.

Step 2: Set up the application’s form as shown in Figure 3-26. Notice that the form’s
Text property is set to Test Average. The names of the controls are shown in the
figure. As you place each of the controls on the form, refer to Table 3-14 for the
relevant property settings.

Table 3-14 Control property settings

<table>
<thead>
<tr>
<th>Control Name</th>
<th>Control Type</th>
<th>Property Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>test1PromptLabel</td>
<td>Label</td>
<td>Text: Test 1:</td>
</tr>
<tr>
<td>test2PromptLabel</td>
<td>Label</td>
<td>Text: Test 2:</td>
</tr>
<tr>
<td>test3PromptLabel</td>
<td>Label</td>
<td>Text: Test 3:</td>
</tr>
<tr>
<td>outputDescriptionLabel</td>
<td>Label</td>
<td>Text: Average Test Score:</td>
</tr>
<tr>
<td>test1TextBox</td>
<td>TextBox</td>
<td>No properties changed</td>
</tr>
<tr>
<td>test2TextBox</td>
<td>TextBox</td>
<td>No properties changed</td>
</tr>
<tr>
<td>test3TextBox</td>
<td>TextBox</td>
<td>No properties changed</td>
</tr>
<tr>
<td>averageLabel</td>
<td>Label</td>
<td>AutoSize: False</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BorderStyle: FixedSingle</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Text: (The contents of the Text</td>
</tr>
<tr>
<td></td>
<td></td>
<td>property have been erased.)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TextAlign: MiddleCenter</td>
</tr>
<tr>
<td>calculateButton</td>
<td>Button</td>
<td>Text: Calculate Sale Price</td>
</tr>
<tr>
<td>clearButton</td>
<td>Button</td>
<td>Text: Clear</td>
</tr>
<tr>
<td>exitButton</td>
<td>Button</td>
<td>Text: Exit</td>
</tr>
</tbody>
</table>

Step 3: Once you have set up the form with its controls, you can create the Click event
handlers for the Button controls. At the end of this tutorial, Program 3-4 shows
the completed code for the form. You will be instructed to refer to Program 3-4
as you write the event handlers. (Remember, the line numbers that are shown
in Program 3-4 are not part of the program. They are shown for reference
only.)

In the Designer, double-click the calculateButton control. This will open the
code editor, and you will see an empty event handler named calculateButton_Click.
Complete the calculateButton_Click event handler by typing the code shown in lines
22–45 in Program 3-4. Let’s take a closer look at the code:

Line 22: This is the beginning of a try-catch statement. The try block appears
in lines 24–39, and the catch block appears in lines 43–44.

Lines 24–27: These statements declare the following double variables: test1, test2, test3, and average. The variables will hold the three test scores and
the average test score.
Line 30: This statement converts the `test1TextBox` control’s Text property to a double and assigns the result to the `test1` variable.

Line 31: This statement converts the `test2TextBox` control’s Text property to a double and assigns the result to the `test2` variable.

Line 32: This statement converts the `test3TextBox` control’s Text property to a double and assigns the result to the `test3` variable.

Line 35: This statement calculates the average of the `test1`, `test2`, and `test3` variables and assigns the result to the `average` variable.

Line 39: This statement converts the `average` variable to a string and assigns the result to the `averageLabel` control’s Text property. Notice that the “n1” format string is passed as an argument to the `ToString` method. This causes the number to be rounded to one decimal point.

**Step 4:** Switch your view back to the Designer and double-click the `clearButton` control. In the code editor you will see an empty event handler named `clearButton_Click`. Complete the `clearButton_Click` event handler by typing the code shown in lines 50–54 in Program 3-4.

Lines 51–53: Each of these statements assigns an empty string (“”) to the Text property of one of the TextBox controls. When these statements have finished executing, the TextBox controls appear empty.

Line 54: This statement assigns an empty string (“”) to the `averageLabel` control’s Text property. After the statement has executed, the label appears empty.

**Step 5:** Switch your view back to the Designer and double-click the `exitButton` control. In the code editor you will see an empty event handler named `exitButton_Click`. Complete the `exitButton_Click` event handler by typing the code shown in lines 59–60 in Program 3-4.

**Step 6:** Save the project. Then, press the `F5` key on the keyboard or click the Start Debugging button ( ) on the toolbar to compile and run the application.

First, test the application by entering valid numeric values into the TextBoxes and clicking the Calculate Average button. A test average should be displayed, similar to the image shown on the left in Figure 3-27.

**Figure 3-27** The Test Average application
Next, click the **Clear** button to clear the TextBoxes and the average. Then enter a nonnumeric value for test 1, and click the **Calculate Average** button. An exception will be thrown, and you should see the message box shown in the image on the right in Figure 3-27.

Continue to test the application as you wish. When you are finished, click the **Exit** button and the form should close.

---

**Program 3-4** Completed Form1 code for the *Test Average* application

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Test_Average
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void calculateButton_Click(object sender, EventArgs e)
        {
            try
            {
                double test1;    // To hold test score #1
                double test2;    // To hold test score #2
                double test3;    // To hold test score #3
                double average;  // To hold the average test score

                // Get the three test scores.
                test1 = double.Parse(test1TextBox.Text);
                test2 = double.Parse(test2TextBox.Text);
                test3 = double.Parse(test3TextBox.Text);

                // Calculate the average test score.
                average = (test1 + test2 + test3) / 3.0;

                // Display the average test score, with
                // the output rounded to 1 decimal point.
                averageLabel.Text = average.ToString("n1");
            }
            catch (Exception ex)
            {
                // Display the default error message.
                MessageBox.Show(ex.Message);
            }
        }

        private void clearButton_Click(object sender, EventArgs e)
        {
            // Clear the TextBoxes and the average.
            test1TextBox.Text = "";
            test2TextBox.Text = "";
            test3TextBox.Text = "";
            averageLabel.Text = ""
        }
    }
}
```
Checkpoint

3.35 What can cause an application to throw an exception?

3.36 How do you get out of break mode when an exception is thrown?

3.37 What kind of code does the try block of a try-catch statement contain?

3.38 What causes the program to jump to the catch clause and execute the catch block of a try-catch statement?

3.39 How can you display the default error message when an exception is thrown?

3.40 Write a try-catch statement for an application that calculates the sum of two whole numbers and displays the result. The application uses two TextBox controls named value1TextBox and value2TextBox to gather the input and a Label control named sumLabel to display the result.

3.8 Using Named Constants

CONCEPT: A named constant is a name that represents a value that cannot be changed during the program’s execution.

Assume that the following statement appears in a banking program that calculates data pertaining to loans:

\[
\text{amount} = \text{balance} \times 0.069;
\]

In such a program, two potential problems arise. First, it is not clear to anyone other than the original programmer what 0.069 is. It appears to be an interest rate, but in some situations there are fees associated with loan payments. How can the purpose of this statement be determined without painstakingly checking the rest of the program?

The second problem occurs if this number is used in other calculations throughout the program and must be changed periodically. Assuming the number is an interest rate, what if the rate changes from 6.9 percent to 7.2 percent? The programmer would have to search through the source code for every occurrence of the number.
Both these problems can be addressed by using named constants. A **named constant** is a name that represents a value that cannot be changed during the program’s execution. The following is an example of how you can declare a named constant in C#:

```csharp
const double INTEREST_RATE = 0.129;
```

This statement declares a named constant named `INTEREST_RATE` initialized with the value 0.129. It looks like a regular variable declaration, except that the word `const` appears before the data type name and the name of the variable is written in uppercase characters. The keyword `const` is a qualifier that tells the compiler to make the variable read only. If a statement attempts to change the constant’s value, an error will occur when the program is being compiled. When you declare a named constant, an initialization value is required.

It is not required that the constant name be written in uppercase letters, but many programmers prefer to write them this way so they are easily distinguishable from regular variable names. When you are reading a program’s code and you see an uppercase identifier, you know instantly that it is a constant.

**NOTE:** Writing the names of constants in uppercase letters is traditional in many programming languages, and that practice is followed in this book. Within the C# community, some programmers adhere to this practice and some do not. In the classroom, you should use the naming convention that your instructor prefers.

An advantage of using named constants is that they make programs more self-explanatory. The statement

```csharp
amount = balance * 0.069;
```

can be changed to read

```csharp
amount = balance * INTEREST_RATE;
```

A new programmer can read the second statement and know what is happening. It is evident that `balance` is being multiplied by the interest rate. Another advantage to this approach is that widespread changes can easily be made to the program. Let’s say the interest rate appears in a dozen different statements throughout the program. When the rate changes, the initialization value in the declaration of the named constant is the only value that needs to be modified. If the rate increases to 7.2 percent, the declaration can be changed to the following:

```csharp
const double INTEREST_RATE = 0.072;
```

The new value of 0.072 will then be used in each statement that uses the `INTEREST_RATE` constant. In Tutorial 3-5 you will create an application that uses named constants.

### 3.9 Declaring Variables as Fields

**CONCEPT:** A field is a variable that is declared at the class level. A field’s scope is the entire class.

So far in this chapter, all the variables with which we have worked have been local variables. A local variable is declared inside a method and is visible only to statements in that method. Another type of variable is a **field**, which is a variable that is declared inside a
class but not inside of any method. A field’s scope is the entire class, so when you declare a field, all the methods in the class can access the variable.

Typically, fields are declared at the top of a class declaration, before any methods. Figure 3-28 shows where you would write field declarations inside a form class. When you are about to write a field declaration, you can insert some blank lines after the class’s opening brace ({) and write the field declaration in that space.

![Figure 3-28](image.png)

Let’s look at an example of a field declaration. Assume that the following statement appears inside a class declaration but not inside any methods. This statement declares an int field named `number`, initialized with the value 0:

```csharp
private int number = 0;
```

Field declarations are written like any other variable declaration, except that an access modifier usually appears before the data type. In this example, the keyword `private` is the access modifier. An access modifier specifies how a class member can be accessed by code outside the class. When you use the `private` access modifier in a field declaration, the field cannot be accessed by code outside the class. It can be accessed only by the methods that are inside the class.

It is a good programming practice to make fields `private` because `private` fields are hidden from code outside the class. That prevents code outside the class from changing the values of a class’s fields and helps prevent bugs from creeping into your program. You will learn more about this in Chapter 10. Until then, if you declare fields in a class, you should get in the habit of making them `private`.

**NOTE:** There are other access modifiers, as you will learn later in this book. If you don’t write an access modifier in a field declaration, C# will automatically make the field `private`. It is still a good idea to write the `private` access modifier because it makes it evident to anyone reading the code that the field is indeed `private`. 
In the previous field-declaration example, the `number` field is initialized with the value 0. If a field is a variable of a numeric data type (such as `int`, `double`, or `decimal`), it will be initialized to 0 by default if you do not explicitly initialize it with a value. It is always a good idea to explicitly initialize a field, however, even if you want it to begin with the value 0. This clearly indicates the field’s starting value for anyone reading the code.

![WARNING!](warning.png)

If you do not initialize a `string` field, it begins with a special value known as `null`. An error will occur if you attempt to use a `string` that is set to `null`.

In a form, fields are useful for storing pieces of data that must be shared among the form’s event handlers. For example, in the `Chap03` folder of this book’s student sample programs (available for download at www.pearsonhighered.com/gaddis), you will find a project named `Field Demo`. Figure 3-29 shows the application’s form, along with the names of the Button controls.

**Figure 3-29** The Field Demo form

Program 3-5 shows the Form1 code. Notice that in line 16 a `string` variable named `name` is declared as a field and initialized with the value “Charles”. Next look at the button event handlers:

- In the `showNameButton_Click` event handler, line 25 displays a message box showing the value of the `name` variable.
- In the `chrisButton_Click` event handler, line 30 changes the value of the `name` variable to “Chris”.
- In the `carmenButton_Click` event handler, line 35 changes the value of the `name` variable to “Carmen”.

As you can see, all of the event handlers in the Form1 class have access to the `name` variable. If you run the application and click the `Show Name` button, a message box will appear displaying `Charles`, which is the name field’s initial value. If you click the `Change Name to Chris` button and then click the `Show Name` button, a message box will appear displaying `Chris`. If you click the `Change Name to Carmen` button and then click the `Show Name` button, a message box will appear displaying `Carmen`.

**Program 3-5** Form1 code for the Field Demo application

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
```
namespace Field_Demo
{
    public partial class Form1 : Form
    {
        // Declare a private field to hold a name.
        private string name = "Charles";

        public Form1()
        {
            InitializeComponent();
        }

        private void showNameButton_Click(object sender, EventArgs e)
        {
            MessageBox.Show(name);
        }

        private void chrisButton_Click(object sender, EventArgs e)
        {
            name = "Chris";
        }

        private void carmenButton_Click(object sender, EventArgs e)
        {
            name = "Carmen";
        }
    }
}

The Lifetime of a Field in a Form Class

When you declare a field in a form class, the field’s lifetime is the time during which the form exists. This means that the field will exist in memory as long as the form exists. This is different than the lifetime of a local variable, which exists only while the method in which it is declared is executing. Local variables come and go in memory, but a form’s fields exist as long as the form exists.

You can see this in the Field Demo application. The name field is created in memory when the Form1 form is created, and it continues to exist as long as Form1 exists. When one of the event handlers stores a value in the field, that value remains in the field until it is changed again, perhaps by a different event handler. So, fields give you a way of storing values that must not disappear when a particular method ends.

Precautions

Although fields make it easy to share data among the methods in a class, you should be careful about using them. The overuse of fields can make debugging a class’s code difficult, especially if the class has many methods. If an incorrect value is being stored in a field, you will have to track down every statement in the class that accesses the field to determine where the incorrect value is coming from. In most cases, fields should be used only for data that must be shared among multiple methods and must continue to exist in memory when the methods are not executing.

Constant Fields

A constant field is a field that cannot be changed by any statement in the class. An error will occur if the compiler finds a statement that tries to change the value of a constant
field. A constant field is declared with the `const` keyword and initialized with a value. Here is an example:

```csharp
private const decimal INTEREST_RATE = 0.075m;
```

This statement declares a constant `decimal` field named `INTEREST_RATE`, initialized with the value 0.075. Constant fields are typically used to represent unchanging values that are needed by many of a class's methods. For example, suppose a banking program uses a constant field to represent an interest rate. If the interest rate is used in several methods, it is easier to create a constant field, rather than a local named constant in each method. This also simplifies maintenance of the code. If the interest rate changes, only the declaration of the constant field has to be changed, instead of several local declarations.

**NOTE:** Because a constant field’s value cannot be changed by other statements in the class, you do not have to worry about many of the potential debugging problems that are associated with the overuse of regular, nonconstant fields.

In Tutorial 3-5 you will create an application that uses a field in a form class to hold data, as well as constant fields to represent nonchanging values.

---

**Tutorial 3-5:**

**Creating the Change Counter Application**

In this tutorial you will create the Change Counter application. The application displays images of four coins, having the values 5 cents, 10 cents, 25 cents, and 50 cents. Each time the user clicks on a coin image, the value of that coin is added to a total, and the total is displayed. Figure 3-30 shows the application’s form, with the names of all the controls.

**Figure 3-30** The Change Counter form

---

**Step 1:** Start Visual Studio (or Visual Studio Express) and begin a new Windows Forms Application project named Sale Price Calculator.

**Step 2:** Set up the application’s form as shown in Figure 3-30. Notice that the form’s Text property is set to Change Counter. The names of the controls are shown in...
Table 3-15  Control property settings

<table>
<thead>
<tr>
<th>Control Name</th>
<th>Control Type</th>
<th>Property Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>instructionLabel</td>
<td>Label</td>
<td><strong>Font</strong>: Microsoft Sans Serif (Style: Bold, Size: 10 point) <strong>Text</strong>: Click the Coins</td>
</tr>
<tr>
<td>fiveCentsPictureBox</td>
<td>PictureBox</td>
<td><strong>Image</strong>: Select and import the 5cents.png file from the Chap02 folder of the Student Sample Programs. <strong>SizeMode</strong>: AutoSize</td>
</tr>
<tr>
<td>tenCentsPictureBox</td>
<td>PictureBox</td>
<td><strong>Image</strong>: Select and import the 10cents.png file from the Chap02 folder of the Student Sample Programs. <strong>SizeMode</strong>: AutoSize</td>
</tr>
<tr>
<td>twentyFiveCentsPictureBox</td>
<td>PictureBox</td>
<td><strong>Image</strong>: Select and import the 25cents.png file from the Chap02 folder of the Student Sample Programs. <strong>SizeMode</strong>: AutoSize</td>
</tr>
<tr>
<td>fiftyCentsPictureBox</td>
<td>PictureBox</td>
<td><strong>Image</strong>: Select and import the 50cents.png file from the Chap02 folder of the Student Sample Programs. <strong>SizeMode</strong>: AutoSize</td>
</tr>
<tr>
<td>outputDescriptionLabel</td>
<td>Label</td>
<td><strong>Font</strong>: Microsoft Sans Serif (Style: Bold, Size: 10 point) <strong>Text</strong>: Total</td>
</tr>
<tr>
<td>totalLabel</td>
<td>Label</td>
<td><strong>AutoSize</strong>: False <strong>BorderStyle</strong>: FixedSingle <strong>Text</strong>: (The contents of the Text property have been erased.) <strong>TextAlign</strong>: MiddleCenter</td>
</tr>
<tr>
<td>exitButton</td>
<td>Button</td>
<td><strong>Text</strong>: Exit</td>
</tr>
</tbody>
</table>

**Step 3:** Once you have set up the form with its controls, you can begin writing code. At the end of this tutorial, Program 3-6 shows the completed code for the form. You will be instructed to refer to Program 3-6 as you write the form’s code. (Remember, the line numbers that are shown in Program 3-6 are not part of the program. They are shown for reference only.)

First, you write the declarations for the fields. Switch your view to the code editor (right-click **Form1.cs** in the **Solution Explorer** and select **View Code** from the pop-up menu). Write the declarations shown in lines 16–23 in Program 3-6. Let’s take a closer look at the code:

**Line 16:** This statement declares a constant **decimal** field named **FIVE_CENTS_VALUE**, initialized with the value 0.05. This constant represents the value of the 5-cent coin.

**Line 17:** This statement declares a constant **decimal** field named **TEN_CENTS_VALUE**, initialized with the value 0.10. This constant represents the value of the 10-cent coin.
Line 18: This statement declares a constant `decimal` field named `TWENTY_FIVE_CENTS_VALUE`, initialized with the value 0.25. This constant represents the value of the 25-cent coin.

Line 19: This statement declares a constant `decimal` field named `FIFTY_CENTS_VALUE`, initialized with the value 0.50. This constant represents the value of the 50-cent coin.

Line 23: This statement declares a `decimal` field named `total`, initialized with the value 0. This field is used to keep the total value of the coins that the user clicks.

Step 4: Now you can create the Click event handlers for the PictureBox controls. Switch your view back to the `Designer` and double-click the `fiveCentsPictureBox` control. This opens the code editor, and you will see an empty event handler named `fiveCentsPictureBox_Click`. Complete the `fiveCentsPictureBox_Click` event handler by typing the code shown in lines 32–36 in Program 3-6. Let’s take a closer look at the code:

Line 33: This statement adds the value of the `FIVE_CENTS_VALUE` constant to the `total` field.

Line 36: This statement converts the `total` variable to a string and assigns the result to the `totalLabel` control’s Text property. The “c” format string causes the number to be formatted as currency.

Step 5: Switch your view back to the `Designer` and double-click the `tenCentsPictureBox` control. This opens the code editor, and you will see an empty event handler named `tenCentsPictureBox_Click`. Complete the `tenCentsPictureBox_Click` event handler by typing the code shown in lines 41–45 in Program 3-6. Let’s take a closer look at the code:

Line 42: This statement adds the value of the `TEN_CENTS_VALUE` constant to the `total` field.

Line 45: This statement converts the `total` variable to a string and assigns the result to the `totalLabel` control’s Text property. The “c” format string causes the number to be formatted as currency.

Step 6: Switch your view back to the `Designer` and double-click the `twentyFiveCentsPictureBox` control. This opens the code editor, and you will see an empty event handler named `twentyFiveCentsPictureBox_Click`. Complete the `twentyFiveCentsPictureBox_Click` event handler by typing the code shown in lines 50–54 in Program 3-6. Let’s take a closer look at the code:

Line 51: This statement adds the value of the `TWENTY_FIVE_CENTS_VALUE` constant to the `total` field.

Line 54: This statement converts the `total` variable to a string and assigns the result to the `totalLabel` control’s Text property. The “c” format string causes the number to be formatted as currency.

Step 7: Switch your view back to the `Designer` and double-click the `fiftyCentsPictureBox` control. This opens the code editor, and you will see an empty event handler named `fiftyCentsPictureBox_Click`. Complete the `fiftyCentsPictureBox_Click` event handler by typing the code shown in lines 59–63 in Program 3-6. Let’s take a closer look at the code:

Line 60: This statement adds the value of the `FIFTY_CENTS_VALUE` constant to the `total` field.

Line 63: This statement converts the `total` variable to a string and assigns the result to the `totalLabel` control’s Text property. The “c” format string causes the number to be formatted as currency.
Step 8: Now you write the event handler for the Exit button. Switch your view back to the Designer and double-click the exitButton control. This opens the code editor, and you will see an empty event handler named exitButton_Click. Complete the exitButton_Click event handler by typing the code shown in lines 68–69 in Program 3-6.

Step 9: Save the project. Then, press the F5 key on the keyboard or click the Start Debugging button ( ) on the toolbar to compile and run the application. Test the application by clicking the coin images in any order you wish. The total shown on the form should update by the correct amount each time you click a coin. When you are finished, click the Exit button and the form should close.

Program 3-6 Completed Form1 code for the Change Counter application

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Change_Counter
{
    public partial class Form1 : Form
    {
        // Constant fields
        const decimal FIVE_CENTS_VALUE = 0.05m;
        const decimal TEN_CENTS_VALUE = 0.10m;
        const decimal TWENTY_FIVE_CENTS_VALUE = 0.25m;
        const decimal FIFTY_CENTS_VALUE = 0.50m;

        // Field variable to hold the total,
        // initialized with 0.
        private decimal total = 0m;

        public Form1()
        {
            InitializeComponent();
        }

        private void fiveCentsPictureBox_Click(object sender, EventArgs e)
        {
            // Add the value of 5 cents to the total.
            total += FIVE_CENTS_VALUE;

            // Display the total, formatted as currency.
            totalLabel.Text = total.ToString("c");
        }

        private void tenCentsPictureBox_Click(object sender, EventArgs e)
        {
            // Add the value of 10 cents to the total.
            total += TEN_CENTS_VALUE;
        }
    }
}
```
3.10 Using the Math Class

CONCEPT: The .NET Framework’s Math class provides several methods for performing complex mathematical calculations.

The .NET Framework provides a class named Math, which contains numerous methods that are useful for performing advanced mathematical operations. Table 3-16 gives a summary of several of the Math class methods. (For a comprehensive list of all the methods...
These methods typically accept one or more values as arguments, perform a mathematical operation using the arguments, and return the result. For example, the `Math.Pow` method raises a number to a power. Here is an example of how the method is called:

```csharp
double result;
result = Math.Sqrt(4.0, 2.0);
```

The method takes two `double` arguments. It raises the first argument to the power of the second argument and returns the result as a `double`. In this example, 4.0 is raised to the power of 2.0. This statement is equivalent to the following algebraic statement:

\[ result = 4^2 \]
The following code sample shows another example of a statement using the `Math.Pow` method. It assigns 3 times 6³ to `x`:

```java
double x;
x = 3 * Math.pow(6.0, 3.0);
```

The `Math.Sqrt` method accepts an argument and returns the square root of the argument. Here is an example of how it is used:

```java
double result;
result = Math.Sqrt(16.0);
```

The statement that calls the `Math.Sqrt` method passes 16.0 as an argument. The method returns the square root of 16.0 (as a `double`), which is then assigned to the `result` variable.

**The `Math.PI` and `Math.E` Named Constants**

The `Math` class also provides two predefined named constants, `Math.PI` and `Math.E`, which are assigned mathematical values for π and e. You can use these variables in equations that require their values. For example, the following statement, which calculates the area of a circle, uses `Math.PI`.

```java
area = Math.PI * radius * radius;
```

**NOTE:** If you look at the top of a form’s source code in the code editor, you should see a directive that reads `using System;`. That directive is required for any program that uses the `Math` class.

**Checkpoint**

3.47 Write a programming statement that uses the `Math.Pow` method to square the number 12 and store the result in a `double` variable named `product`.

3.48 What method of the `Math` class can be used to determine the larger of two values?

3.49 What method of the `Math` class can be used to determine the smaller of two values?

**3.11 More GUI Details**

In Chapter 2 you learned the basics of creating a GUI by placing controls on a form and setting various properties. In this section you learn to fine-tune many aspects of an application’s GUI.

**Controlling a Form’s Tab Order**

When an application is running and a form is displayed, one of the form’s controls always has the focus. The control having the focus is the one that receives the user’s keyboard input. For example, when a TextBox control has the focus, it receives the characters that the user enters on the keyboard. When a button has the focus, pressing the Enter key executes the button’s Click event handler.
You can tell which control has the focus by looking at the form at run time. When a TextBox control has the focus, a blinking text cursor appears inside it, or the text inside the TextBox control might appear highlighted. When a button has the focus, a thin dotted line usually appears around the control.

When an application is running, pressing the Tab key changes the focus from one control to another. The order in which controls receive the focus is called the **tab order**. When you place controls on a form in Visual C#, the tab order is in the same sequence in which you created the controls. In many cases this is the tab order you want, but sometimes you rearrange controls on a form, delete controls, and add new ones. These modifications often lead to a disorganized tab order, which can confuse and irritate the users of your application.

Users want to tab smoothly from one control to the next, in a logical sequence. You can modify the tab order by changing a control’s TabIndex property. The **TabIndex property** contains a numeric value, which indicates the control's position in the tab order. When you create a control, Visual C# automatically assigns a value to its TabIndex property. The first control you create on a form has a TabIndex of 0, the second has a TabIndex of 1, and so on. The control with a TabIndex of 0 is the first control in the tab order. The next control in the tab order is the one with a TabIndex of 1. The tab order continues in this sequence.

You may change the tab order of a form's controls by selecting them, one by one, and changing their TabIndex property in the **Properties** window. An easier method, however, is to click **VIEW** on the Visual Studio menu bar and then click **Tab Order**. This causes the form to be displayed in **tab order selection mode**. The image on the left in Figure 3-31 shows a form in the normal view, and the image on the right shows the form in tab order selection mode. We have also inserted the names of the TextBox and Button controls in the image on the right for reference purposes.

**Figure 3-31** A form displayed in tab order selection mode

In tab order selection mode, each control’s existing TabIndex value is displayed in a small box in the control’s upper-left corner. Notice the following in the image on the right in Figure 3-31:
• The nameTextBox control’s TabIndex is 2.
• The addressTextBox control’s TabIndex is 3.
• The cityTextBox control’s TabIndex is 0.
• The stateTextBox control’s TabIndex is 4.
• The zipTextBox control’s TabIndex is 1.
• The clearButton control’s TabIndex is 8.

**NOTE:** Although the Label controls have TabIndex values, those values are irrelevant in this example because Label controls cannot receive the focus.

As you look at Figure 3-31, think about the order in which the controls will receive the focus when the application runs.

• The cityTextBox control has the lowest TabIndex value (0), so it will receive the focus first.
• If you press the Tab key, the focus will jump to the zipTextBox control because it has the next lowest TabIndex value (1).
• Press the Tab key again and the focus will jump to the nameTextBox control (TabIndex is set to 2).
• Press the Tab key again and the focus will jump to the addressTextBox control (TabIndex is set to 3).
• Press the Tab key again and the focus will jump to the stateTextBox control (TabIndex is set to 4).
• Press the Tab key again and the focus will jump to the clearButton control (TabIndex is set to 8).

This is a very confusing tab order and should be rearranged. When a form is displayed in tab order selection mode, you establish a new tab order by simply clicking the controls with the mouse in the order you want. To fix the disorganized tab order shown in Figure 3-31, we perform the following:

• First, click the nameTextBox control. The control’s TabIndex value changes to 0.
• Next, click the addressTextBox control. The control’s TabIndex value changes to 1.
• Then, click the cityTextBox control. The control’s TabIndex value changes to 2.
• Next, click the stateTextBox control. The control’s TabIndex value changes to 3.
• Then, click the zipTextBox control. The control’s TabIndex value changes to 4.
• Finally, click the clearButton control. The control’s TabIndex value changes to 5.

When you are finished, exit tab order selection mode by pressing the Esc key. Now when the application runs, the focus will shift smoothly in an order that makes sense to the user.

### Changing the Focus with the **Focus** Method

Often, you want to make sure a particular control has the focus. For example, look at the form shown in Figure 3-31. The purpose of the Clear button is to clear any input that the user has entered and reset the form so it is ready to accept a new set of input. When the Clear button is clicked, the TextBox controls should be cleared and the focus should return to the nameTextBox control. This would make it unnecessary for the user to click the TextBox control in order to start entering another set of information.

In code, you move the focus to a control by calling the **Focus** method. The method’s general syntax is:

```csharp
ControlName.Focus();
```
where ControlName is the name of the control. For instance, you move the focus to the nameTextBox control with this statement:

```csharp
nameTextBox.Focus();
```

After the statement executes, the nameTextBox control will have the focus. Here is an example of how the clearButton control’s Click event handler could be written:

```csharp
private void clearButton_Click(object sender, EventArgs e)
{
    // Clear the TextBox controls.
    nameTextBox.Text = "";
    addressTextBox.Text = "";
    cityTextBox.Text = "";
    stateTextBox.Text = "";
    zipTextBox.Text = "";

    // Set the focus to nameTextBox.
    nameTextBox.Focus();
}
```

The statements in lines 4–8 clear the contents of the TextBox controls. Then, the statement in line 11 sets the focus to the nameTextBox control.

### Assigning Keyboard Access Keys to Buttons

An access key, also known as a mnemonic, is a key that is pressed in combination with the Alt key to access quickly a control such as a button. When you assign an access key to a button, the user can trigger a Click event either by clicking the button with the mouse or by using the access key. Users who are quick with the keyboard prefer to use access keys instead of the mouse.

You assign an access key to a button through its Text property. For example, assume an application has a button whose Text property is set to Exit. You wish to assign the access key Alt + X to the button so the user may trigger the button’s Click event by pressing Alt + X on the keyboard. To make the assignment, place an ampersand (&) before the letter x in the button’s Text property: E&xit. Figure 3-32 shows how the Text property appears in the Properties window.

**Figure 3-32** Text property E&amp;xit
Although the ampersand is part of the Button control’s Text property, it is not displayed on the button. With the ampersand in front of the letter \textit{x}, the letter will appear underlined as shown in Figure 3-33. This indicates that the button may be clicked by pressing \texttt{Alt} + \texttt{x} on the keyboard. (You will see the underlining at design time. At run time, however, the underlining may not appear until the user presses the \texttt{Alt} key.)

**Figure 3-33** Button control with \texttt{Exit} Text property

NOTE: Access keys do not distinguish between uppercase and lowercase characters. There is no difference between \texttt{Alt} + \texttt{X} and \texttt{Alt} + \texttt{X}.

Suppose we store the value \texttt{Exit} in the button’s Text property. The ampersand is in front of the letter \texttt{E}, so \texttt{Alt} + \texttt{E} becomes the access key. The button will appear as shown in Figure 3-34.

**Figure 3-34** Button control with \texttt{Exit} Text property

**Assigning the Same Access Key to Multiple Buttons**

Be careful not to assign the same access key to two or more buttons on the same form. If two or more buttons share the same access key, a Click event is triggered for the first button created when the user presses the access key.

**Displaying the & Character on a Button**

If you want to display an ampersand character on a button, use two ampersands (&&) in the Text property. Using two ampersands causes a single ampersand to display and does not define an access key. For example, if a button’s Text property is set to \texttt{Save} \texttt{&&} \texttt{Exit}, the button will appear as shown in Figure 3-35.

**Figure 3-35** Button control with \texttt{Save \& Exit} Text property

**Accept Buttons and Cancel Buttons**

An \textit{accept button} is a button on a form that is automatically clicked when the user presses the Enter key. A \textit{cancel button} is a button on a form that is automatically clicked when the user presses the Esc key. Forms have two properties, AcceptButton and CancelButton, which allow you to designate an accept button and a cancel button. When you select these properties in the Properties window, a down-arrow button (▼) appears, which displays a drop-down list when clicked. The list contains the names of all the buttons on the form. You select the button that you want to designate as the accept button or cancel button.

Any button that is frequently clicked should probably be selected as the accept button. This will allow keyboard users to access the button quickly and easily. \textit{Exit} or \textit{Cancel} buttons are likely candidates to become cancel buttons.
The BackColor Property

Forms and most controls have a BackColor property that allows you to change the object’s background color. When you select an object’s BackColor property in the Properties window, a down-arrow button appears, which displays a drop-down list of available colors when clicked, as shown in Figure 3-36.

The drop-down list has three tabs: Custom, Web, and System. The System tab lists colors defined in the current Windows configuration. The Web tab lists colors displayed with consistency in Web browsers. The Custom tab displays a color palette. Select a color from one of the tabs and the object’s background color will be set to that color.

The ForeColor Property

Controls that display text have a ForeColor property that allows you to change the color of the text. When you select a control’s ForeColor property in the Properties window, a down-arrow button appears, which displays the drop-down list of available colors shown in Figure 3-36 when clicked. Select a color from one of the tabs and the text that is displayed by the control will be set to that color.

Figure 3-36  Drop-down list of colors

Setting Colors in Code

In addition to using the Properties window, you can also set the values of the BackColor and ForeColor properties with code. The .NET Framework provides numerous values that represent colors and can be assigned to the ForeColor and BackColor properties in code. The following are a few of the values:

- Color.Black
- Color.Blue
- Color.Cyan
- Color.Green
- Color.Magenta
- Color.Red
- Color.White
- Color.Yellow

For example, assume an application has a Label control named messageLabel. The following code sets the label’s background color to black and foreground color to yellow:

```csharp
messageLabel.BackColor = Color.Black;
messageLabel.ForeColor = Color.Yellow;
```
The .NET Framework also provides values that represent default colors on your system. For example, the value `SystemColors.Control` represents the default control background color and `SystemColors.ControlText` represents the default control text color. The following statements set the `messageLabel` control's background and foreground to the default colors.

```csharp
messageLabel.BackColor = SystemColors.Control
messageLabel.ForeColor = SystemColors.ControlText
```

**NOTE:** If you have an event handler in a form's source code file and you want the event handler to change the form's `BackColor` property, use the `this` keyword to refer to the form. For example, the following statement changes the color of the form to blue:

```csharp
this.BackColor = Color.Blue;
```

### Background Images for Forms

In Chapter 2 you learned about displaying images with PictureBox controls. An image can also be displayed as the background for a form. Forms have a property named `BackColorImage` that allows you to import and display an image on the form. If you know how to use the PictureBox control's `Image` property, then you already know how to use a form's `BackColorImage` property. They both work the same way:

- Click the `BackColorImage` property in the *Properties* window. An ellipses button will appear.
- Click the ellipses button and the *Select Resource* window will appear.
- In the *Select Resource* window, click the *Import* button. An *Open* dialog box will appear. Use the dialog box to locate and select the image file that you want to display.
- Click the *OK* button in the *Select Resource* window, and the selected image will appear as the form's background.

A form's `BackColorImageLayout` property is similar to the PictureBox control's `SizeMode` property. It specifies how the background image is to be displayed. It can be set to one of the following values:

- **None**
  The image is positioned in the upper-left corner of the form. If the image is too big to fit in the form, it is clipped.
- **Tile**
  This is the default value. The image is tiled (repeatedly displayed) across the form.
- **Center**
  The image is centered in the form without being resized.
- **Stretch**
  The image is resized both horizontally and vertically to fit in the form. If the image is resized more in one direction than the other, it appears stretched.
- **Zoom**
  The image is uniformly resized to fit in the form without losing its original aspect ratio. This causes the image to be resized without appearing stretched.

Figure 3-37 shows examples of each of these settings.
Chapter 3  Processing Data

Organizing Controls with GroupBoxes and Panels

A GroupBox control is a rectangular control that appears with a thin border and an optional title in its upper-left corner. It is a container that can hold other controls. You can use GroupBoxes to create a sense of visual organization on a form.

The GroupBox control is found in the Toolbox, in the Containers section. When you create a GroupBox control, you can set its Title property to the text that you want displayed in the GroupBox’s upper-left corner. If you don’t want a title displayed on the GroupBox, you can clear the contents of its Text property.

Figure 3-38 shows a GroupBox control. The control’s Text property is set to Personal Data, and several other controls are inside the GroupBox.

Figure 3-37 Different settings for the BackgroundImageLayout property

![Figure 3-37](image)

BackgroundImageLayout set to None

BackgroundImageLayout set to Tile

BackgroundImageLayout set to Center

BackgroundImageLayout set to Stretch

BackgroundImageLayout set to Zoom
Creating a Group Box and Adding Controls to It

Suppose you've just created a GroupBox control. To add another control to the GroupBox, select the GroupBox control and then double-click the desired tool in the Toolbox to place another control inside the group box.

Moving an Existing Control to a Group Box

If an existing control is not inside a GroupBox but you want to move it to the GroupBox, follow these steps:

1. Select the control you wish to add to the GroupBox.
2. Cut the control to the clipboard.
3. Select the GroupBox.
4. Paste the control.

Moving and Resizing a GroupBox

If a GroupBox is selected in the Designer, a four-headed arrow (↑↓←→) will appear in the GroupBox's upper-left corner. Click and drag the four-headed arrow to move the GroupBox. Any controls inside the GroupBox move with it.

Deleting a GroupBox

To delete a GroupBox, simply select it in the Designer and then press the [Delete] key. Any controls inside the GroupBox are deleted as well.

Group Box Tab Order

The value of a control's TabIndex property is handled differently when the control is placed inside a GroupBox control. GroupBox controls have their own TabIndex property, and the TabIndex value of the controls inside the group box are relative to the GroupBox control's TabIndex property. For example, Figure 3-39 shows a GroupBox control displayed in tab order selection mode. As you can see, the GroupBox control's TabIndex is set to 0. The TabIndex of the controls inside the group box is displayed as 0.0, 0.1, 0.2, and so on.

Figure 3-39 GroupBox TabIndex values

NOTE: The TabIndex properties of the controls inside the group box will not appear this way in the Properties window. They will appear as 0, 1, 2, and so on.
A Panel control is a rectangular container for other controls, like a GroupBox. There are several primary differences between a Panel and GroupBox:

- A Panel cannot display a title and does not have a Text property.
- A Panel’s border can be specified by its BorderStyle property. The available settings are None, FixedSingle, and Fixed3D. The property is set to None by default, which means that no border will appear. If the BorderStyle property is set to FixedSingle, the control will be outlined with a thin border. If the BorderStyle property is set to Fixed3D, the control will have a recessed 3D appearance.

Figure 3-40 shows an example of a form with a Panel. The Panel’s BorderStyle property is set to Fixed3D.

**Figure 3-40** A Panel containing other controls

---

**Checkpoint**

3.50 What happens if you press the Enter key while a Button control has the focus?

3.51 How do you display a form in tab order selection mode? How do you exit tab order selection mode?

3.52 Write a programming statement that gives the focus to a TextBox control named `numberTextBox`.

3.53 How do you assign an access key to a Button control?

3.54 How do you display an ampersand (&) character on a Button control?

3.55 Write the code that will change the BackColor property of a Label control named `resultLabel` to the color white and the ForeColor property to the color red.

3.56 List the different values of a form’s BackgroundImageLayout property.

3.57 When a GroupBox control is deleted, what happens to the controls that are inside?

3.58 How are the TabIndex properties of the controls inside the group box organized?

3.59 How is a Panel control different from a GroupBox control?
Key Terms

accept button
accept key
access modifier
argument
BackColor property
break mode
cancel button
cast operator
catch block
catch clause
combined assignment operators
concatenation
constant field
data type
decimal literal
decimal.Parse method
double literal
double.Parse method
exception
exception handler
exception object
field
focus
Focus method
ForeColor
formatting string
GroupBox Control
initialize
int.Parse method
integer literal
lifetime
local variable
math expression
math operators
mnemonic
named constant
numeric literal
operands
order of operations
Panel control
parse
Parse methods
primitive data types
scope
tab order
tab order selection mode
TabIndex property
TextBox control
ToString method
truncation
try block
try-catch statement
variable
variable declaration
variable name

Review Questions

1. When the user types into a TextBox control, the text is stored in the control’s ________ property.
   a. Input
   b. Text
   c. String
   d. Data

2. A ________ is a storage location in memory that is represented by a name.
   a. mnemonic
   b. data type
   c. namespace
   d. variable

3. In C#, you must ________ a variable before you can use it to store data.
   a. cite
   b. associate
   c. declare
   d. instance
4. A variable’s __________ indicates the type of data that the variable will hold.
   a. name
   b. data type
   c. scope
   d. value

5. Fundamental types of data, such as strings, integers, and real numbers, are known as __________.
   a. primitive data types
   b. fundamental variables
   c. logical digits
   d. literal data types

6. A __________ identifies a variable in the program code.
   a. binary number
   b. variable name
   c. unique global identifier
   d. hexadecimal value

7. A common operation performed on strings is __________, or appending one string to the end of another string.
   a. addition
   b. merging
   c. concatenation
   d. tying

8. A __________ belongs to the method in which it is declared, and only statements inside that method can access the variable.
   a. method variable
   b. primitive variable
   c. temporary variable
   d. local variable

9. Programmers use the term __________ to describe the part of a program in which a variable may be accessed.
   a. range
   b. scope
   c. focus
   d. field

10. A variable’s __________ is the time period during which the variable exists in memory while the program is executing.
    a. lifetime
    b. run time
    c. time to live
    d. half life

11. One way to make sure that a variable has been assigned a value is to __________ the variable with a value when you declare it.
    a. concatenate
    b. initialize
    c. delimit
    d. restrict
12. You can use a __________ to explicitly convert a value from one numeric data type to another, even if the conversion might result in a loss of data.
   a. transpose statement
   b. cast operator
   c. conversion operator
   d. literal conversion

13. The process of dropping a number’s fractional part is called __________.
   a. shifting
   b. twos complement
   c. numeric rounding
   d. truncation

14. A programmer’s tools for performing calculations are __________.
   a. math operators
   b. numeric literals
   c. local variables
   d. parsed literals

15. A __________ performs a calculation and gives a value.
    a. numeric literal
    b. math expression
    c. machine instruction
    d. programming statement

16. C# offers a special set of operators known as __________ that are designed specifically for changing the value of a variable without having to type the variable name twice.
    a. combined assignment operators
    b. advanced math operators
    c. variable modifiers
    d. assignment sequencers

17. In computer science, the term __________ typically means to analyze a string of characters for some purpose.
    a. compile
    b. compute
    c. debug
    d. parse

18. A(n) __________ is a piece of data that is passed into a method.
    a. variable
    b. argument
    c. string
    d. literal

19. A(n) __________ is an unexpected error that occurs while a program is running, causing the program to halt if the error is not properly dealt with.
    a. breakpoint
    b. bug
    c. syntax error
    d. exception
20. The ________ indicates that you want the number to appear formatted in a specific way when it is returned as a string from the `ToString` method.
   a. formatting string
   b. `insert` method
   c. data type
   d. variable name

21. You have started an application by clicking the start `Debugging` button (⿴) or by pressing `F5` on the keyboard. If an exception is thrown, the application stops running and Visual Studio goes into a special mode known as ________.
   a. exception mode
   b. break mode
   c. debug mode
   d. crash mode

22. Code that responds to exceptions when they are thrown and prevents the program from abruptly crashing is called a(n) ________.
   a. exit strategy
   b. fail safe
   c. event handler
   d. exception handler

23. A ________ is a name that represents a value that cannot be changed during the program’s execution.
   a. named literal
   b. named constant
   c. variable signature
   d. key term

24. A ________ is a variable that is declared inside a class but not inside any method.
   a. term
   b. class variable
   c. field
   d. mnemonic

25. A(n) ________ specifies how a class member can be accessed by code outside the class.
   a. namespace
   b. access modifier
   c. scope delimiter
   d. class directive

26. A ________ is a field that cannot be changed by any statement in the class.
   a. static field
   b. class name
   c. key field
   d. constant field

27. The .NET Framework provides a class named ________, which contains numerous methods that are useful for performing advanced mathematical operations.
   a. `Math`
   b. `Calc`
   c. `Trig`
   d. `Linq`
28. When a control has the __________, it receives the user’s keyboard input.
   a.  text  
   b.  tab order  
   c.  focus  
   d.  input allocator

29. The order in which controls receive the focus is called the __________.
   a.  order of operations  
   b.  program flow  
   c.  execution sequence  
   d.  tab order

30. The __________ contains a numeric value, which indicates the control’s position in
    the tab order.
   a.  IndexOf property  
   b.  TabIndex property  
   c.  ControlOrder property  
   d.  TabOrder property

**True or False**

1. You can clear the contents of a TextBox control in the same way that you clear
   the contents of a Label control.

2. In C#, you must declare a variable in a program before you can use it to store data.

3. You can declare multiple variables of different data types with one declaration.

4. When you append the letter D or d to a numeric literal, it is treated as a decimal
   and is referred to as a decimal literal.

5. The order of operations dictates that the division operator works before the addition
   operator does.

6. All variables have a `ToString` method that you can call to convert the variable’s
   value to a string.

7. When you pass the formatting string “C” or “c” to the `ToString` method, the
   number is returned formatted as currency.

8. When you declare a named constant, an initialization value is required.

9. An error will occur if the compiler finds a statement that tries to change the value of
   a constant field.

10. Forms and most controls have a Preferences property that allows you to change the
    object’s background color.

**Short Answer**

1. In the Toolbox, in which group is the TextBox tool located?

2. What two things does a variable declaration specify about a variable?

3. Give an example of a programming statement that uses string concatenation.

4. What is the term used for a number that is written into a program’s code?

5. Write a programming statement that assigns an integer literal to a variable.

6. What are the values on the right and left of an operator called?

7. Name the family of methods in the .NET Framework that can be used to convert a
   string to any of the numeric data types.
8. What object is created in memory when an exception is thrown and has various properties that contain data about the exception?

9. What is the purpose of a `try-catch` statement?

10. Which class in the .NET Framework provides predefined named constants that are assigned the mathematical values for \( \pi \) and \( e \)?

11. In code, what function do you call to move the focus to a control?

12. What property allows you to change the color of a control’s text?

## Programming Problems

1. **Name Formatter**

   Create an application that lets the user enter the following pieces of data:
   - The user’s first name
   - The user’s middle name
   - The user’s last name
   - The user’s preferred title (Mr., Mrs., Ms., Dr., etc.)

   Assume the user has entered the following data:
   - First name: Kelly
   - Middle name: Jane
   - Last name: Smith
   - Title: Ms.

   The application should have buttons that display the user’s name formatted in the following ways:
   - Ms. Kelly Jane Smith
   - Kelly Jane Smith
   - Kelly Smith
   - Smith, Kelly Jane, Ms.
   - Smith, Kelly Jane
   - Smith, Kelly

2. **Tip, Tax, and Total**

   Create an application that lets the user enter the food charge for a meal at a restaurant. When a button is clicked, the application should calculate and display the amount of a 15 percent tip, 7 percent sales tax, and the total of all three amounts.

3. **Distance Traveled**

   Assuming there are no accidents or delays, the distance that a car travels down an interstate highway can be calculated with the following formula:

   \[
   \text{Distance} = \text{Speed} \times \text{Time}
   \]

   Create an application that allows the user to enter a car’s speed in miles per hour. The application should have buttons that display the following:
   - The distance the car will travel in 5 hours
   - The distance the car will travel in 8 hours
   - The distance the car will travel in 12 hours

4. **Sales Tax and Total**

   Create an application that allows the user to enter the amount of a purchase. The program should then calculate the state and county sales tax. Assume the state sales tax is 4 percent and the county sales tax is 2 percent. The program should display
the amount of the purchase, the state sales tax, the county sales tax, the total sales tax, and the total of the sale (which is the sum of the amount of purchase plus the total sales tax).

5. **Celsius and Fahrenheit Temperature Converter**
Assuming that $C$ is a Celsius temperature, the following formula converts the temperature to Fahrenheit:

$$F = \frac{9}{5} C + 32$$

Assuming that $F$ is a Fahrenheit temperature, the following formula converts the temperature to Celsius:

$$C = \frac{5}{9} (F - 32)$$

Create an application that allows the user to enter a temperature. The application should have Button controls described as follows:
- A button that reads _Convert to Fahrenheit_. If the user clicks this button, the application should treat the temperature that is entered as a Celsius temperature and convert it to Fahrenheit.
- A button that reads _Convert to Celsius_. If the user clicks this button, the application should treat the temperature that is entered as a Fahrenheit temperature, and convert it to Celsius.

6. **Body Mass Index**
Create an application that lets the user enter his or her weight (in pounds) and height (in inches). The application should display the user's body mass index (BMI). The BMI is often used to determine whether a person is overweight or underweight for his or her height. A person’s BMI is calculated with the following formula:

$$BMI = \frac{weight \times 703}{height^2}$$

7. **Sentence Builder**
The form in Figure 3-41 contains buttons showing various words, phrases, and punctuation. Create an application with a form similar to this one. When the application runs, the user clicks the buttons to build a sentence, which is shown in a Label control. You can use the same buttons as shown in the figure or make up your own. The _Reset_ button should clear the sentence so the user can start over.

**Figure 3-41** The _Sentence Builder_ form
8. **How Much Insurance?**
Many financial experts advise that property owners should insure their homes or buildings for at least 80 percent of the amount it would cost to replace the structure. Create an application that lets the user enter the replacement cost of a building and then displays the minimum amount of insurance he or she should buy for the property.

9. **Cookie Calories**
A bag of cookies holds 40 cookies. The calorie information on the bag claims that there are 10 servings in the bag and that a serving equals 300 calories. Create an application that lets the user enter the number of cookies he or she actually ate and then reports the number of total calories consumed.

10. **Calorie Counter**
Create an application with a form that resembles Figure 3-42. The PictureBox controls display the images of four fruits (a banana, an apple, an orange, and a pear) and each fruit’s calories. You can find these images in the Chap03 folder of the Student Sample Programs.

When the application starts, the total calories displayed should be zero. Each time the user clicks one of the PictureBoxes, the calories for that fruit should be added to the total calories, and the total calories should be displayed. When the user clicks the **Reset** button, the total calories should be reset to zero.

**Figure 3-42 Calorie Counter form**

11. **Automobile Costs**
Create an application that lets the user enter the monthly costs for the following expenses incurred from operating his or her automobile: loan payment, insurance, gas, oil, tires, and maintenance. The program should then display the total monthly cost of these expenses and the total annual cost of these expenses.

12. **Paint Job Estimator**
A painting company has determined that for every 115 square feet of wall space, 1 gallon of paint and 8 hours of labor will be required. The company charges $20.00 per hour for labor. Create an application that allows the user to enter the square feet of wall space to be painted and the price of the paint per gallon. The program should display the following data:
- The number of gallons of paint required
- The hours of labor required
Programming Problems

13. **Property Tax**

If you own real estate in a particular county, the property tax that you owe each year is calculated as 64 cents per $100 of the property’s value. For example, if the property’s value is $10,000, then the property tax is calculated as follows:

\[ \text{Tax} = \frac{10,000}{100} \times 0.64 \]

Create an application that allows the user to enter a property’s value and displays the sales tax on that property.

14. **Stadium Seating**

There are three seating categories at an athletic stadium. For a baseball game, Class A seats cost $15 each, Class B seats cost $12 each, and Class C seats cost $9 each. Create an application that allows the user to enter the number of tickets sold for each class. The application should be able to display the amount of income generated from each class of ticket sales and the total revenue generated. The application’s form should resemble the one shown in Figure 3-43.

**Figure 3-43 Stadium Seating form**

Use the following sets of test data to determine if the application is calculating properly:

<table>
<thead>
<tr>
<th>Ticket Sales</th>
<th>Revenue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class A: 320</td>
<td>Class A: $4,800.00</td>
</tr>
<tr>
<td>Class B: 570</td>
<td>Class B: $6,840.00</td>
</tr>
<tr>
<td>Class C: 890</td>
<td>Class C: $8,010.00</td>
</tr>
<tr>
<td>Total Revenue: $19,650.00</td>
<td></td>
</tr>
<tr>
<td>Class A: 500</td>
<td>Class A: $7,500.00</td>
</tr>
<tr>
<td>Class B: 750</td>
<td>Class B: $9,000.00</td>
</tr>
<tr>
<td>Class C: 1,200</td>
<td>Class C: $10,800.00</td>
</tr>
<tr>
<td>Total Revenue: $27,300.00</td>
<td></td>
</tr>
<tr>
<td>Class A: 100</td>
<td>Class A: $1,500.00</td>
</tr>
<tr>
<td>Class B: 300</td>
<td>Class B: $3,600.00</td>
</tr>
<tr>
<td>Class C: 500</td>
<td>Class C: $4,500.00</td>
</tr>
<tr>
<td>Total Revenue: $9,600.00</td>
<td></td>
</tr>
</tbody>
</table>
This page intentionally left blank
Decision Structures and the if Statement

CONCEPT: A decision structure allows a program to perform actions only under certain conditions. In code, you can use the if statement to write a simple decision structure.

A control structure is a logical design that controls the order in which a set of statements execute. So far in this book we have used only the simplest type of control structure: the sequence structure. A sequence structure is a set of statements that execute in the order that they appear. For example, the following code sample is a sequence structure because the statements execute from top to bottom.

```csharp
int ageInYears, ageInDays;
ageInYears = int.Parse(ageTextBox.Text);
ageInDays = ageInYears * 365;
daysLabel = ageInDays.ToString();
```

Although the sequence structure is heavily used in programming, it cannot handle every type of task. Some problems simply cannot be solved by performing a set of ordered steps, one after the other. For example, consider a pay-calculating program that determines whether an employee has worked overtime. If the employee has worked more than 40 hours, he or she gets paid extra for all hours over 40. Otherwise, the overtime calculation should be skipped. Programs like this require a different type of control structure: one that can execute a set of statements only under certain circumstances. This can be accomplished with a decision structure. (Decision structures are also known as selection structures.)
In a decision structure’s simplest form, a specific action is performed only if a certain condition exists. If the condition does not exist, the action is not performed. The flowchart shown in Figure 4-1 shows how the logic of an everyday decision can be diagrammed as a decision structure. The diamond symbol represents a true-false condition. If the condition is true, we follow one path, which leads to an action being performed. If the condition is false, we follow another path, which skips the action.

**Figure 4-1** A simple decision structure

![Decision Structure Diagram](image)

In the flowchart, the diamond symbol indicates some condition that must be tested. In this case, we are determining whether the condition *Cold outside* is true or false. If this condition is true, the action *Wear a coat* is performed. If the condition is false, the action is skipped. The action is **conditionally executed** because it is performed only when a certain condition is true.

Programmers call the type of decision structure shown in Figure 4-1 a **single-alternative decision structure** because it provides only one alternative path of execution. If the condition in the diamond symbol is true, we take the alternative path. Otherwise, we exit the structure. Figure 4-2 shows a more elaborate example, where three actions are taken only when it is cold outside.

**Figure 4-2** A decision structure that performs three actions if it is cold outside

![Decision Structure Diagram](image)
In C#, you use the `if` statement to write a single-alternative decision structure. Here is the general format of the `if` statement:

```csharp
if (expression)
{
    statement;
    statement;
    etc.
}
```

The statement begins with the word `if`, followed by an expression enclosed in a set of parentheses. Beginning on the next line is a set of statements enclosed in curly braces.

The expression that appears inside the parentheses is a Boolean expression. A **Boolean expression** is an expression that can be evaluated as either true or false. When the `if` statement executes, the Boolean expression is tested. If it is true, the statements that appear inside the curly braces are executed. If the Boolean expression is false, however, the statements inside the curly braces are skipped. We say that the statements inside the curly braces are **conditionally executed** because they are executed only if the Boolean expression is true.

If you are writing an `if` statement that has only one conditionally executed statement, you do not have to enclose the conditionally executed statement inside curly braces. Such an `if` statement can be written in the following general format:

```csharp
if (expression)
    statement;
```

When an `if` statement written in this format executes, the Boolean expression is tested. If it is true, the one statement that appears on the next line is executed. If the Boolean expression is false, however, that one statement is skipped.

Although the curly braces are not required when there is only one conditionally executed statement, it is still a good idea to use them, as shown in the following general format:

```csharp
if (expression)
{
    statement;
}
```

This is a good style for writing `if` statements because it minimizes errors. Remember, if you have more than one conditionally executed statement, those statements must be enclosed in curly braces. If you get into the habit of always enclosing the conditionally executed statements in a set of curly braces, it’s less likely that you will forget them.

**Boolean Expressions and Relational Operators**

Boolean expressions are named in honor of the English mathematician George Boole. In the 1800s, Boole invented a system of mathematics in which the abstract concepts of true and false can be used in computations.

Typically, the Boolean expression that is tested by an `if` statement is formed with a relational operator. A **relational operator** determines whether a specific relationship exists between two values. For example, the greater than operator (`>`), determines whether one value is greater than another. The equal to operator (`==`) determines whether two values are equal. Table 4-1 lists the relational operators that are available in C#.

The following is an example of an expression that uses the greater than (`>`) operator to compare two variables, `length` and `width`:

```csharp
length > width
```
Chapter 4  Making Decisions

This expression determines whether the value of the length variable is greater than the value of the width variable. If length is greater than width, the value of the expression is true. Otherwise, the value of the expression is false. The following expression uses the less than operator (<) to determine whether length is less than width:

\[ \text{length} < \text{width} \]

Table 4-2 shows examples of several Boolean expressions that compare the variables \( x \) and \( y \).

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x &gt; y )</td>
<td>Is ( x ) greater than ( y )?</td>
</tr>
<tr>
<td>( x &lt; y )</td>
<td>Is ( x ) less than ( y )?</td>
</tr>
<tr>
<td>( x &gt;= y )</td>
<td>Is ( x ) greater than or equal to ( y )?</td>
</tr>
<tr>
<td>( x &lt;= y )</td>
<td>Is ( x ) less than or equal to ( y )?</td>
</tr>
<tr>
<td>( x == y )</td>
<td>Is ( x ) equal to ( y )?</td>
</tr>
<tr>
<td>( x != y )</td>
<td>Is ( x ) not equal to ( y )?</td>
</tr>
</tbody>
</table>

The >= and <= Operators

Two of the operators, \( >= \) and \( <= \), test for more than one relationship. The \( >= \) operator determines whether the operand on its left is greater than or equal to the operand on its right. The \( <= \) operator determines whether the operand on its left is less than or equal to the operand on its right.

For example, assume the variable \( a \) is assigned 4. All the following expressions are true:

\[
\begin{align*}
  a & >= 4 \\
  a & >= 2 \\
  8 & >= a \\
  a & <= 4 \\
  a & <= 9 \\
  4 & <= a
\end{align*}
\]

The == Operator

The == operator determines whether the operand on its left is equal to the operand on its right. If the values of both operands are the same, the expression is true. Assuming that \( a \) is 4, the expression \( a == 4 \) is true and the expression \( a == 2 \) is false.
### The `!=` Operator

The `!=` operator is the not equal to operator. It determines whether the operand on its left is not equal to the operand on its right, which is the opposite of the `==` operator. As before, assuming `a` is 4, `b` is 6, and `c` is 4, both `a != b` and `b != c` are true because `a` is not equal to `b` and `b` is not equal to `c`. However, `a != c` is false because `a` is equal to `c`.

### Putting It All Together

Let’s look at the following example of the `if` statement:

```csharp
if (sales > 50000)
{
    bonus = 500;
}
```

This statement uses the `>` operator to determine whether `sales` is greater than 50,000. If the expression `sales > 50000` is true, the variable `bonus` is assigned 500. If the expression is false, however, the assignment statement is skipped. Figure 4-3 shows a flowchart for this section of code.

#### Figure 4-3 Example decision structure

The following code sample conditionally executes three statements. Figure 4-4 shows a flowchart for this section of code.

```csharp
if (sales > 50000)
{
    bonus = 500;
    commissionRate = 0.12;
    MessageBox.Show("You met your sales quota!");
}
```

When you write an `if` statement, Visual Studio automatically indents the conditionally executed statements, as shown in the previous examples. The indentation is not required, but it makes the code easier to read and debug. By indenting the conditionally executed statements, you visually set them apart from the surrounding code. This allows you to tell at a glance what part of the program is controlled by the `if` statement. Most programmers use this style of indentation when writing `if` statements.
In this tutorial you will complete an application that allows the user to enter three test scores and calculates the average of the test scores. If the average is greater than 95, the application also displays a message congratulating the user.

To save time, the project has already been started for you, and the application’s form has already been created. To complete the project, follow the steps in this tutorial.

**Step 1:** Start Visual Studio (or Visual Studio Express). Open the project named *Test Score Average* in the Chap04 folder of this book’s Student Sample Programs, available for download at www.pearsonhighered.com/gaddis.

**Step 2:** Open the Form1 form in the Designer. The form is shown, along with the names of the important controls, in Figure 4-5.

**Tutorial 4-1:**
Completing the *Test Score Average* Application

**Figure 4-5** The *Test Score Average* form
Step 3: Now you will create the Click event handlers for the Button controls. At the end of this tutorial, Program 4-1 shows the completed code for the form. You will be instructed to refer to Program 4-1 as you write the event handlers.

In the Designer, double-click the calculateButton control. This will open the code editor, and you see an empty event handler named calculateButton_Click. Complete the calculateButton_Click event handler by typing the code shown in lines 22–49 in Program 4-1.

Let’s take a closer look at the code:

Line 22: This is the beginning of a try-catch statement. The try block appears in lines 24–43, and the catch block appears in lines 47–48. The purpose of this try-catch statement is to gracefully respond if the user enters invalid input for any of the test scores. If any of the statements in lines 28, 29, and 30 throw an exception, the program does not crash. Instead, it jumps to the catch block, and line 48 displays an error message.

Line 24: This statement declares a constant double named HIGH_SCORE, set to the value 95.0. We use this constant to determine whether the average is high. If the average is greater than this constant, the program displays a message congratulating the user.

Line 25: This statement declares the following double variables: test1, test2, test3, and average. The variables hold the three test scores and the average test score.

Line 28: This statement converts the test1TextBox control’s Text property to a double and assigns the result to the test1 variable.

Line 29: This statement converts the test2TextBox control’s Text property to a double and assigns the result to the test2 variable.

Line 30: This statement converts the test3TextBox control’s Text property to a double and assigns the result to the test3 variable.

Line 33: This statement calculates the average of the test1, test2, and test3 variables and assigns the result to the average variable.

Line 36: This statement converts the average variable to a string (rounded to 1 decimal place) and assigns the result to the averageLabel control’s Text property.

Line 40: This if statement determines whether average is greater than HIGH_SCORE. If it is, the statement in line 42 is executed, displaying a message box with a congratulatory message. If average is not greater than HIGH_SCORE, the statement in line 42 is skipped.

Step 4: Switch your view back to the Designer and double-click the clearButton control. In the code editor you will see an empty event handler named clearButton_Click. Complete the clearButton_Click event handler by typing the code shown in lines 54–61 in Program 4-1.

Lines 55–57: Each of these statements assigns an empty string ("") to the Text property of one of the TextBox controls. When these statements have finished executing, the TextBox controls appear empty.

Line 58: This statement assigns an empty string ("") to the averageLabel control’s Text property. After the statement has executed, the label appears empty.

Line 61: This statement sets the focus to the test1TextBox control. This makes it more convenient for the user to start entering a new set of test scores.

Step 5: Switch your view back to the Designer and double-click the exitButton control. In the code editor you will see an empty event handler named exitButton_Click. Complete the exitButton_Click event handler by typing the code shown in lines 66–67 in Program 4-1.
Step 6: Save the project. Then, press F5 on the keyboard or click the Start Debugging button on the toolbar to compile and run the application.

First, enter the following test scores in the TextBoxes: 80, 90, and 75. Click the Calculate Average button and the average should appear as shown in Figure 4-6.

**Figure 4-6 Average displayed**

Next, click the Clear button to clear the TextBoxes and the average. Now, enter the following test scores in the TextBoxes: 100, 97, and 99. Click the Calculate Average button. This time, in addition to displaying the average, the application displays the message box shown in Figure 4-7.

**Figure 4-7 Average and message displayed**

Continue to test the application as you wish. When you are finished, click the Exit button, and the form should close.

**Program 4-1 Completed Form1 code for the Test Score Average application**

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
```
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Test_Score_Average
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        private void calculateButton_Click(object sender, EventArgs e)
        {
            try
            {
                const double HIGH_SCORE = 95.0;       // High Score value
double test1, test2, test3, average; // Variables

                // Get the test scores from the Textboxes.
test1 = double.Parse(test1TextBox.Text);
test2 = double.Parse(test2TextBox.Text);
test3 = double.Parse(test3TextBox.Text);

                // Calculate the average test score.
average = (test1 + test2 + test3) / 3.0;

                // Display the average, rounded to 2 decimal places.
averageLabel.Text = average.ToString("n1");

                // If the average is a high score, congratulate
                // the user with a message box.
if (average > HIGH_SCORE)
                {
                    MessageBox.Show("Congratulations! Great job!");
                }
            }
            catch (Exception ex)
            {
                // Display the default error message.
MessageBox.Show(ex.Message);
            }
        }
        private void clearButton_Click(object sender, EventArgs e)
        {
            // Clear the Textboxes and the averageLabel control.
test1TextBox.Text = "";
test2TextBox.Text = "";
test3TextBox.Text = "";
averageLabel.Text = "";

                // Reset the focus to test1.
test1TextBox.Focus();
        }
        private void exitButton_Click(object sender, EventArgs e)
        {
            // Close the form.
Checkpoint

4.1 What is a control structure?
4.2 What is a decision structure?
4.3 What is a single-alternative decision structure?
4.4 What is a Boolean expression?
4.5 What types of relationships between numeric values can you test with relational operators?
4.6 Write an if statement that determines whether the variable \( y \) is equal to 20. If it is, assign 0 to the variable \( x \).
4.7 Write an if statement that determines whether the variable \( \text{sales} \) is greater than or equal to 10,000. If it is, assign 0.2 to the variable \( \text{commissionRate} \).

4.2 The if-else Statement

CONCEPT: An if-else statement will execute one block of statements if its Boolean expression is true or another block if its Boolean expression is false.

The previous section introduced the single-alternative decision structure (the if statement), which has one alternative path of execution. Now we will look at the dual-alternative decision structure, which has two possible paths of execution—one path is taken if the Boolean expression is true, and the other path is taken if the Boolean expression is false. Figure 4-8 shows an example flowchart for a dual-alternative decision structure.

Figure 4-8 A dual-alternative decision structure

```csharp
67     this.Close();
68     }
69   }
70 }
```
The decision structure in the flowchart tests the expression \( \text{temperature} < 40 \). If this expression is true, the message “A little cold, isn’t it?” is displayed. If the expression is false, the message “Nice weather we’re having.” is displayed.

In code we write a dual-alternative decision structure as an \textit{if-else} statement. Here is the general format of the \textit{if-else} statement:

\begin{verbatim}
if (expression)
{
    statement;
    statement;
    etc.
}
else
{
    statement;
    statement;
    etc.
}
\end{verbatim}

An \textit{if-else} statement has two parts: an if clause and an else clause. Just like a regular \textit{if} statement, the \textit{if-else} statement tests a Boolean expression. If the Boolean expression is true, the set of statements following the \textit{if} clause is executed. If the Boolean expression is false, the set of statements following the \textit{else} clause is executed.

The \textit{if-else} statement has two sets of conditionally executed statements. One set is executed only under the condition that the Boolean expression is true, and the other set is executed only under the condition that the Boolean expression is false. Under no circumstances are both sets of conditionally executed statements executed.

If either set of conditionally executed statements contains only one statement, the curly braces are not required. For example, the following general format shows only one statement following the \textit{if} clause and only one statement following the \textit{else} clause:

\begin{verbatim}
if (expression)
    statement;
else
    statement;
\end{verbatim}

Although the curly braces are not required when there is only one conditionally executed statement, it is still a good idea to use them, as shown in the following general format:

\begin{verbatim}
if (expression)
{
    statement;
}
else
{
    statement;
}
\end{verbatim}

When we discussed the regular \textit{if} statement, we mentioned that this is a good style of programming because it cuts down on errors. If there is more than one conditionally executed statement following either the \textit{if} clause or the \textit{else} clause, those statements \textit{must} be enclosed in curly braces. If you get into the habit of always enclosing the conditionally executed statements in a set of curly braces, it’s less likely that you will forget them.

In Tutorial 4-2 you will complete an application that uses an \textit{if-else} statement.
Tutorial 4-2: Completing the Payroll with Overtime Application

At a particular business, if an employee works more than 40 hours in a week, it is said that the employee has worked overtime. For example, an employee that has worked 45 hours in a week has worked 5 overtime hours. Employees that work overtime get paid their regular hourly pay rate for the first 40 hours plus 1.5 times their regular hourly pay rate for all hours over 40. In this tutorial you will complete a payroll application that calculates an employee’s gross pay, including overtime pay.

The application allows the user to enter the number of hours worked and the hourly pay rate into TextBoxes. When the user clicks a button, the gross pay is calculated in the following manner:

If the hours worked is greater than 40:

- base pay = hourly pay rate × 40
- overtime hours = hours worked − 40
- overtime pay = overtime hours × hourly pay rate × 1.5
- gross pay = base pay + overtime pay

Else:

- gross pay = hours worked × hourly pay rate

To save time, the project has already been started for you, and the application’s form has already been created. To complete the project, follow the steps in this tutorial.

Step 1: Start Visual Studio (or Visual Studio Express). Open the project named Payroll with Overtime in the Chap04 folder of this book’s Student Sample Programs.

Step 2: Open the Form1 form in the Designer. The form is shown, along with the names of the important controls, in Figure 4-9.

Figure 4-9 The Payroll with Overtime form

Step 3: Now you will create the Click event handlers for the Button controls. At the end of this tutorial, Program 4-2 shows the completed code for the form. You will be instructed to refer to Program 4-2 as you write the event handlers.

In the Designer, double-click the calculateButton control. This opens the code editor, and you see an empty event handler named calculateButton_Click. Complete the calculateButton_Click event handler by typing the code shown in lines 22–69 in Program 4-2.
4.2 The if-else Statement

Let’s take a closer look at the code:

**Line 22:** This is the beginning of a try-catch statement. The try block appears in lines 24–63, and the catch block appears in lines 67–68. The purpose of this try-catch statement is to gracefully respond if the user enters invalid input. If an exception is thrown by any statement inside the try block, the program jumps to the catch block, and line 68 displays an error message.

**Lines 25–26:** These statements declare the following named constants:
- BASE_HOURS, a constant decimal set to the value 40. This is the number of hours an employee can work in a week without getting overtime pay.
- OT_MULTIPLIER, a constant decimal set to the value 1.5. This is the pay rate multiplier for overtime hours.

**Lines 29–34:** These statements declare the following variables:
- hoursWorked, a decimal variable to hold the number of hours worked
- hourlyPayRate, a decimal variable to hold the hourly pay rate
- basePay, a decimal variable to hold the pay for 40 or less hours
- overtimeHours, a decimal variable to hold the number of overtime hours worked
- overtimePay, a decimal variable to hold the amount of overtime pay
- grossPay, a decimal variable to hold the gross pay

**Line 37:** This statement converts the hoursWorkedTextBox control’s Text property to a decimal and assigns the result to the hoursWorked variable.

**Line 38:** This statement converts the hourlyPayRateTextBox control’s Text property to a decimal and assigns the result to the hourlyPayRate variable.

**Line 41:** This if statement determines whether hoursWorked is greater than BASE_HOURS (40). If so, the statements in lines 43–54 are executed. Otherwise, the statements in lines 58–59 are executed.

**Lines 43–54:** These statements, which are executed only if the hours worked are greater than 40, make all the necessary calculations to determine gross pay with overtime:
- Line 44 calculates the base pay, which is the amount of pay for the first 40 hours.
- Line 47 calculates the number of overtime hours, which is the number of hours over 40.
- Lines 50 and 51 calculate the amount of overtime pay, which is the pay for the hours over 40.
- Line 54 calculates the gross pay, which is the amount of base pay plus the amount of overtime pay. The result is assigned to the grossPay variable.

**Line 59:** This statement, which is executed only if the hours worked are 40 or less, calculates the gross pay and assigns the result to the grossPay variable.

**Line 63:** This statement converts the value of the grossPay variable to a string, formatted as currency, and assigns the result to the grossPayLabel control’s Text property.

**Step 4:** Switch your view back to the Designer and double-click the clearButton control. In the code editor you see an empty event handler named clearButton_Click. Complete the clearButton_Click event handler by typing the code shown in lines 74–80 in Program 4-2. These statements clear the TextBoxes and the grossPayLabel control and sets the focus to the hoursWorkedTextBox control.
**Step 5:** Switch your view back to the Designer and double-click the exitButton control. In the code editor you see an empty event handler named exitButton_Click. Complete the exitButton_Click event handler by typing the code shown in lines 85–86 in Program 4-2.

**Step 6:** Save the project and run the application. First, enter 40 for the number of hours worked and 20 for the hourly pay rate. Click the Calculate Gross Pay button, and the application should display $800.00 as the gross pay. No overtime hours were worked, so the gross pay is simply calculated as hours worked × hourly pay rate.

Click the Clear button. Enter 50 for the number of hours worked and 20 for the hourly pay rate. Click the Calculate Gross Pay button, and the application should display $1,100.00 as the gross pay. This time, more than 40 hours were worked, so the application calculated the gross pay to include overtime pay.

Continue to test the application as you wish. When you are finished, click the Exit button and the form should close.

---

**Program 4-2 Completed Form1 code for the Payroll with Overtime application**

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Payroll_with_Overtime
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void calculateButton_Click(object sender, EventArgs e)
        {
            try
            {
                // Named constants
                const decimal BASE_HOURS = 40m;
                const decimal OT_MULTIPLIER = 1.5m;

                // Local variables
                decimal hoursWorked; // Number of hours worked
                decimal hourlyPayRate; // Hourly pay rate
                decimal basePay; // Pay not including overtime
                decimal overtimeHours; // overtime hours worked
                decimal overtimePay; // overtime pay
                decimal grossPay; // total gross pay

                // Get the hours worked and hourly pay rate.
                hoursWorked = decimal.Parse(hoursWorkedTextBox.Text);
                hourlyPayRate = decimal.Parse(hourlyPayRateTextBox.Text);
```
4.2 The if-else Statement

```csharp
// Determine the gross pay.
if (hoursWorked > BASE_HOURS)
{
    // Calculate the base pay (without overtime).
    basePay = hourlyPayRate * BASE_HOURS;

    // Calculate the number of overtime hours.
    overtimeHours = hoursWorked - BASE_HOURS;

    // Calculate the overtime pay.
    overtimePay = overtimeHours * hourlyPayRate * 
    OT_MULTIPLIER;

    // Calculate the gross pay.
    grossPay = basePay + overtimePay;
}
else
{
    // Calculate the gross pay.
    grossPay = hoursWorked * hourlyPayRate;
}

// Display the gross pay.
grossPayLabel.Text = grossPay.ToString("c");

``` catch (Exception ex)
{
    // Display an error message.
    MessageBox.Show(ex.Message);
}

private void clearButton_Click(object sender, EventArgs e)
{
    // Clear the TextBoxes and gross pay label.
    hoursWorkedTextBox.Text = "";
    hourlyPayRateTextBox.Text = "";
    grossPayLabel.Text = "";

    // Reset the focus.
    hoursWorkedTextBox.Focus();
}

private void exitButton_Click(object sender, EventArgs e)
{
    // Close the form.
    this.Close();
}

Checkpoint

4.8 Describe how a dual alternative decision structure works.

4.9 In an if-else statement, under what circumstances do the statements that appear after the else clause execute?
4.10 Write an if-else statement that works like this: If the \texttt{sales} variable is greater-than or equal-to 50,000, the \texttt{commissionRate} variable should be assigned the value 0.2. Otherwise, the \texttt{commissionRate} variable should be assigned the value 0.1.

### 4.3 Nested Decision Structures

**CONCEPT:** To test more than one condition, a decision structure can be nested inside another decision structure.

In Section 4.1, we mentioned that a control structure determines the order in which a set of statements execute. Programs are usually designed as combinations of different control structures. For example, Figure 4-10 shows a flowchart that combines a decision structure with two sequence structures.

**Figure 4-10** Combining sequence structures with a decision structure
The flowchart in Figure 4-10 starts with a sequence structure. Assuming you have an outdoor thermometer in your window, the first step is *Go to the window*, and the next step is *Read thermometer*. A decision structure appears next, testing the condition *Cold outside*. If this is true, the action *Wear a coat* is performed. Another sequence structure appears next. The step *Open the door* is performed, followed by *Go outside*.

Quite often, structures must be nested inside other structures. For example, look at the partial flowchart in Figure 4-11. It shows a decision structure with a sequence structure nested inside. The decision structure tests the condition *Cold outside*. If that condition is true, the steps in the sequence structure are executed.

**Figure 4-11** A sequence structure nested inside a decision structure

You can also have nested decision structures, which are decision structures that appear inside other decision structures. This is commonly done in programs that need to test more than one condition. For example, consider a program that determines whether a bank customer qualifies for a loan. To qualify, two conditions must exist: (1) The customer must earn at least $40,000 per year, and (2) the customer must have been employed at his or her current job for at least 2 years. Figure 4-12 shows a flowchart for an algorithm that could be used in such a program. Assume that the `salary` variable contains the customer’s annual salary, and the `yearsOnJob` variable contains the number of years that the customer has worked on his or her current job.

If we follow the flow of execution, we see that the Boolean expression `salary >= 40000` is tested. If this expression is false, there is no need to perform further tests; we know that the customer does not qualify for the loan. If the expression is true, however, we need to test the second condition. This is done with a nested decision structure that tests the Boolean expression `yearsOnJob >= 2`. If this expression is true, then the customer qualifies for the loan. If this expression is false, then the customer does not qualify. In Tutorial 4-3 you create an application that performs this algorithm.
In this tutorial you complete an application that determines whether a person qualifies for a loan. To qualify for the loan, the person must earn a salary of at least $40,000 and must have been employed at his or her current job for at least 2 years.

To save time, the project has already been started for you, and the application’s form has already been created. To complete the project, follow the steps in this tutorial.

**Step 1:** Start Visual Studio (or Visual Studio Express). Open the project named *Loan Qualifier* in the Chap04 folder of this book’s Student Sample Programs.

**Step 2:** Open the Form1 form in the Designer. The form is shown, along with the names of the important controls, in Figure 4-13.

**Step 3:** Now you will create the Click event handlers for the Button controls. At the end of this tutorial, Program 4-3 shows the completed code for the form. You will be instructed to refer to Program 4-3 as you write the event handlers.
4.3 Nested Decision Structures

In the Designer, double-click the checkButton control. This opens the code editor, and you see an empty event handler named checkButton_Click. Complete the checkButton_Click event handler by typing the code shown in lines 22–62 in Program 4-3. Let’s take a closer look at the code:

**Line 22:** This is the beginning of a try-catch statement. The try block appears in lines 24–56, and the catch block appears in lines 60–61. The purpose of this try-catch statement is to gracefully respond if the user enters invalid input. If an exception is thrown by any statement inside the try block, the program jumps to the catch block, and line 61 displays an error message.

**Lines 25–26:** These statements declare the following named constants:
- **MINIMUM_SALARY**, a constant decimal set to the value 40,000, which is the minimum salary a person must earn to qualify for the loan
- **MINIMUM_YEARS_ON_JOB**, a constant int set to the value 2, which is the minimum number of years a person must have been at his or her current job to qualify for the loan

**Lines 29–30:** These statements declare the following variables:
- **salary**, a decimal variable to hold the salary
- **yearsOnJob**, an int variable to hold the number of years at the current job

**Lines 33–34:** These statements get the salary and years at the current job from the TextBox controls and assign those values to the salary and yearsOnJob variables.

**Line 37:** This if statement determines whether salary is greater than or equal to **MINIMUM_SALARY**. If so, the program continues at line 39. Otherwise, the program jumps to the else clause in line 51, and in lines 54–55 the string “Minimum salary requirement not met.” is assigned to the decisionLabel control’s Text property.

**Line 39:** This if statement determines whether yearsOnJob is greater than or equal to **MINIMUM_YEARS_ON_JOB**. If so, the program continues at line 42, where the string “You qualify for the loan.” is assigned to the decisionLabel control’s Text property. Otherwise, the program jumps to the else clause in line 44, and in lines 47–48 the string “Minimum years at current job not met.” is assigned to the decisionLabel control’s Text property.

**Step 4:** Switch your view back to the Designer and double-click the clearButton control. In the code editor you see an empty event handler named clearButton_Click. Complete the clearButton_Click event handler by typing the code shown in lines 67–73 in Program 4-3.
Step 5: Switch your view back to the Designer and double-click the exitButton control. In the code editor you see an empty event handler named exitButton_Click. Complete the exitButton_Click event handler by typing the code shown in lines 78–79 in Program 4-3.

Step 6: Save the project and run the application. First, enter 45000 for the salary and 1 for the years at current job. Click the Check Qualifications button, and the application should display the message “Minimum years at current job not met.”

Click the Clear button. Enter 35000 for the salary and 5 for the years at current job. Click the Check Qualifications button, and the application should display the message “Minimum salary requirement not met.”

Click the Clear button. Enter 45000 for the salary and 5 for the years at current job. Click the Check Qualifications button, and the application should display the message “You qualify for the loan.”

Continue to test the application as you wish. When you are finished, click the Exit button and the form should close.

Program 4-3  Completed Form1 code for the Loan Qualifier application

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Loan_Qualifier
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void checkButton_Click(object sender, EventArgs e)
        {
            try
            {
                // Names constants
                const decimal MINIMUM_SALARY = 40000m;
                const int MINIMUM_YEARS_ON_JOB = 2;

                // Local variables
                decimal salary;
                int yearsOnJob;

                // Get the salary and years on the job.
                salary = decimal.Parse(salaryTextBox.Text);
                yearsOnJob = int.Parse(yearsTextBox.Text);

                // Determine whether the user qualifies.
                if (salary >= MINIMUM_SALARY) 
                {
                    // Additional code...
                }
            }
        }
    }
}```
Indentation and Alignment in Nested Decision Structures

For debugging purposes, it’s important to use proper alignment and indentation in a nested if statement. This makes it easier to see which actions are performed by each part of the structure. For example, the following code is functionally equivalent to lines 37–56 in Program 4-3. Although this code is logically correct, it would be very difficult to debug because it is not properly indented.
if (salary >= MINIMUM_SALARY)
{
    if (yearsOnJob >= MINIMUM_YEARS_ON_JOB)
    {
        // The user qualifies.
        decisionLabel.Text = "You qualify for the loan.";
    }
    else
    {
        // The user does not qualify.
        decisionLabel.Text = "Minimum years at current " +
                             "job not met.";
    }
}
else
{
    // The user does not qualify.
    decisionLabel.Text = "Minimum salary requirement " +
                         "not met.";
}

Fortunately, Visual Studio automatically indents and aligns the statements in a decision structure. Proper indentation and alignment makes it easier to see which if and else clauses belong together, as shown in Figure 4-14.

**Figure 4-14** Alignment of if and else clauses

![Alignment of if and else clauses](image)

Testing a Series of Conditions

In Tutorial 4-3 you saw how a program can use nested decision structures to test more than one Boolean expression. It is not uncommon for a program to have a series of Boolean expressions to test and then perform an action, depending on which expression is true. One way to accomplish this is to have a decision structure with numerous other decision structures nested inside it. For example, look at the Grader application in the Chap04 folder of this book’s Student Sample Programs.

Figure 4-15 shows the application’s form, with the names of several controls. When you run the application, you enter a numeric test score into the testScoreTextBox control and click the determineGradeButton control; a grade is then displayed in the gradeLabel control.
4.3 Nested Decision Structures

The following 10-point grading scale is used to determine the grade:

<table>
<thead>
<tr>
<th>Test Score</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>90 and above</td>
<td>A</td>
</tr>
<tr>
<td>80–89</td>
<td>B</td>
</tr>
<tr>
<td>70–79</td>
<td>C</td>
</tr>
<tr>
<td>60–69</td>
<td>D</td>
</tr>
<tr>
<td>Below 60</td>
<td>F</td>
</tr>
</tbody>
</table>

The logic of determining the grade can be expressed like this:

If the test score is less than 60, then the grade is “F.”

Otherwise, if the test score is less than 70, then the grade is “D.”

Otherwise, if the test score is less than 80, then the grade is “C.”

Otherwise, if the test score is less than 90, then the grade is “B.”

Otherwise, the grade is “A.”

This logic requires several nested decision structures, as shown in the flowchart in Figure 4-16.
Open the code editor and look at the determineGradeButton_Click event handler, shown in the following code sample. The nested decision structure appears in lines 12–41.

```csharp
private void determineGradeButton_Click(object sender, EventArgs e)
{
    try
    {
        // Variable to hold the test score.
        double testScore;

        // Get the test score.
        testScore = double.Parse(testScoreTextBox.Text);

        // Determine the grade.
        if (testScore < 60)
        {
            gradeLabel.Text = "F";
        }
        else
        {
            if (testScore < 70)
            {
                gradeLabel.Text = "D";
            }
            else
            {
                if (testScore < 80)
                {
                    gradeLabel.Text = "C";
                }
                else
                {
                    if (testScore < 90)
                    {
                        gradeLabel.Text = "B";
                    }
                    else
                    {
                        gradeLabel.Text = "A";
                    }
                }
            }
        }
    }
    catch (Exception ex)
    {
        // Display an error message.
        MessageBox.Show(ex.Message);
    }
}
```

**The if-else-if Statement**

Even though the Grader application previously shown is a simple example, the logic of the nested decision structure is fairly complex. C# provides a special version of the decision structure known as the **if-else-if statement**, which makes this type of logic simpler to write. You write the **if-else-if** statement using the following general format:
4.3 Nested Decision Structures

When the statement executes, BooleanExpression_1 is tested. If BooleanExpression_1 is true, the set of statements that immediately follows is executed, and the rest of the structure is skipped. If BooleanExpression_1 is false, however, the program jumps to the very next else if clause and tests BooleanExpression_2. If it is true, the set of statements that immediately follows is executed, and the rest of the structure is then skipped. This process continues until a Boolean expression is found to be true, or no more else if clauses are left. If none of the Boolean expressions are true, the set of statements following the final else clause is executed.

For example, look at the Grader2 application in the Chap04 folder of this book’s Student Sample Programs. This application works just like the Grader application that was previously discussed. The user enters a numeric test score, and the application displays a grade. Its form is identical to the form shown in Figure 4-15. The Grader2 application, however, uses an if-else-if statement to determine the grade instead of nested if-else statements. The Grader2 application’s determineGradeButton_Click event handler is shown here:

```csharp
private void determineGradeButton_Click(object sender, EventArgs e)
{
    try
    {
        // Variable to hold the test score.
        double testScore;

        // Get the test score.
        testScore = double.Parse(testScoreTextBox.Text);

        // Determine the grade.
        if (testScore < 60)
        {
            gradeLabel.Text = "F";
        }
        else if (testScore < 70)
        {
            gradeLabel.Text = "D";
        }
        else if (testScore < 80)
        {
            gradeLabel.Text = "C";
        }
        else if (testScore < 90)
        {
            gradeLabel.Text = "B";
        }
        else
```

When the statement executes, BooleanExpression_1 is tested. If BooleanExpression_1 is true, the set of statements that immediately follows is executed, and the rest of the structure is skipped. If BooleanExpression_1 is false, however, the program jumps to the very next else if clause and tests BooleanExpression_2. If it is true, the set of statements that immediately follows is executed, and the rest of the structure is then skipped. This process continues until a Boolean expression is found to be true, or no more else if clauses are left. If none of the Boolean expressions are true, the set of statements following the final else clause is executed.

For example, look at the Grader2 application in the Chap04 folder of this book’s Student Sample Programs. This application works just like the Grader application that was previously discussed. The user enters a numeric test score, and the application displays a grade. Its form is identical to the form shown in Figure 4-15. The Grader2 application, however, uses an if-else-if statement to determine the grade instead of nested if-else statements. The Grader2 application’s determineGradeButton_Click event handler is shown here:
{ gradeLabel.Text = "A";
}

catch (Exception ex)
{
    // Display an error message.
    MessageBox.Show(ex.Message);
}

Notice the alignment and indentation that is used with the if-else-if statement: The if, else if, and else clauses are all aligned, and the conditionally executed statements are indented.

You never have to use the if-else-if statement because its logic can be coded with nested if-else statements. However, a long series of nested if-else statements has two particular disadvantages when you are debugging code:

- The code can grow complex and become difficult to understand.
- Because indenting is important in nested statements, a long series of nested if-else statements can become too long to be displayed on the computer screen without horizontal scrolling. Also, long statements tend to wrap around when printed on paper, making the code even more difficult to read.

The logic of an if-else-if statement is usually easier to follow than a long series of nested if-else statements. And, because all the clauses are aligned in an if-else-if statement, the lengths of the lines in the statement tend to be shorter.

**Checkpoint**

4.11 Convert the following set of nested if-else statements to an if-else if statement:

```csharp
if (number == 1)
{
    MessageBox.Show("One");
}
else
{
    if (number == 2)
    {
        MessageBox.Show("Two");
    }
    else
    {
        if (number == 3)
        {
            MessageBox.Show("Three");
        }
        else
        {
            MessageBox.Show("Unknown");
        }
    }
}
```
4.4 Logical Operators

CONCEPT: The logical AND operator (&&) and the logical OR operator (||) allow you to connect multiple Boolean expressions to create a compound expression. The logical NOT operator (!) reverses the truth of a Boolean expression.

The C# language provides a set of operators known as logical operators, which you can use to create complex Boolean expressions. Table 4-3 describes these operators.

Table 4-3 Logical operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>This is the logical AND operator. It connects two Boolean expressions into one compound expression. Both subexpressions must be true for the compound expression to be true.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>This is the logical NOT operator. It is a unary operator, meaning it works with only one operand. The operand must be a Boolean expression. The not operator reverses the truth of its operand. If it is applied to an expression that is true, the operator returns false. If it is applied to an expression that is false, the operator returns true.</td>
</tr>
</tbody>
</table>

Table 4-4 shows examples of several compound Boolean expressions that use logical operators.

Table 4-4 Compound Boolean expressions using logical operators

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>x &gt; y &amp;&amp; a &lt; b</td>
<td>Is x greater than y AND is a less than b?</td>
</tr>
<tr>
<td>x == y</td>
<td></td>
</tr>
<tr>
<td>!(x &gt; y)</td>
<td>Is the expression x &gt; y NOT true?</td>
</tr>
</tbody>
</table>

The && Operator

The && operator is the logical AND operator. It takes two Boolean expressions as operands and creates a compound Boolean expression that is true only when both subexpressions are true. The following is an example of an if statement that uses the && operator:

```csharp
if (temperature < 20 && minutes > 12)
{
    MessageBox.Show("The temperature is in the danger zone.");
}
```

In this statement, the two Boolean expressions temperature < 20 and minutes > 12 are combined into a compound expression. The MessageBox.Show statement is executed only if temperature is less than 20 and minutes is greater than 12. If either of the Boolean subexpressions is false, the compound expression is false and the message is not displayed.
Table 4-5 shows a truth table for the $\&\&$ operator. The truth table lists expressions showing all the possible combinations of true and false connected with the $\&\&$ operator. The resulting values of the expressions are also shown.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value of the Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>true $&amp;&amp;$ false</td>
<td>false</td>
</tr>
<tr>
<td>false $&amp;&amp;$ true</td>
<td>false</td>
</tr>
<tr>
<td>false $&amp;&amp;$ false</td>
<td>false</td>
</tr>
<tr>
<td>true $&amp;&amp;$ true</td>
<td>true</td>
</tr>
</tbody>
</table>

As the table shows, both sides of the $\&\&$ operator must be true for the operator to return a true value.

**The $\|\|$ Operator**

The $\|\|$ operator is the logical OR operator. It takes two Boolean expressions as operands and creates a compound Boolean expression that is true when either of the subexpressions is true. The following is an example of an if statement that uses the $\|\|$ operator:

```csharp
if (temperature < 20 || temperature > 100)
{
    MessageBox.Show("The temperature is in the danger zone.");
}
```

The `MessageBox.Show` statement executes only if `temperature` is less than 20 or `temperature` is greater than 100. If either subexpression is true, the compound expression is true. Table 4-6 shows a truth table for the $\|\|$ operator.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value of the Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>true $||$ false</td>
<td>true</td>
</tr>
<tr>
<td>false $||$ true</td>
<td>true</td>
</tr>
<tr>
<td>false $||$ false</td>
<td>false</td>
</tr>
<tr>
<td>true $||$ true</td>
<td>true</td>
</tr>
</tbody>
</table>

All it takes for an $\|\|$ expression to be true is for one side of the $\|\|$ operator to be true. It doesn’t matter if the other side is false or true.

**Short-Circuit Evaluation**

Both the $\&\&$ and $\|\|$ operators perform short-circuit evaluation. Here is how it works with the $\&\&$ operator: if the expression on the left side of the $\&\&$ operator is false, the expression on the right side is not checked. Because the compound expression is false if only one of the subexpressions is false, it would waste CPU time to check the remaining expression. So, when the $\&\&$ operator finds that the expression on its left is false, it short-circuits and does not evaluate the expression on its right.

Here’s how short-circuit evaluation works with the $\|\|$ operator: if the expression on the left side of the $\|\|$ operator is true, the expression on the right side is not checked. Because it is necessary for only one of the expressions to be true, it would waste CPU time to check the remaining expression.
The ! Operator

The ! operator is the logical NOT operator. It is a unary operator that takes a Boolean expression as its operand and reverses its logical value. In other words, if the expression is true, the ! operator returns false, and if the expression is false, the ! operator returns true. The following is an if statement using the NOT operator:

```csharp
if (! (temperature > 100))
{
    MessageBox.Show("This is below the maximum temperature.");
}
```

First, the expression `(temperature > 100)` is tested and a value of either true or false is the result. Then the ! operator is applied to that value. If the expression `(temperature > 100)` is true, the ! operator returns false. If the expression `(temperature > 100)` is false, the ! operator returns true. The previous code is equivalent to asking "Is the temperature not greater than 100?"

Notice that in this example, we have put parentheses around the expression `temperature > 100`. This is necessary because the ! operator has higher precedence than the relational operators. If we do not put the parentheses around the expression `temperature > 100`, the ! operator will be applied just to the `temperature` variable.

Table 4-7 shows a truth table for the ! operator.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value of the Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>! true</td>
<td>false</td>
</tr>
<tr>
<td>! false</td>
<td>true</td>
</tr>
</tbody>
</table>

Precedence of the Logical Operators

We mentioned earlier that the ! operator has higher precedence than the relational operators. The && and || logical operators have lower precedence than the relational operators. For example, look at the following expression:

```csharp
creditScore > 700 || accountBalance > 9000
```

When this expression is evaluated, the > operators work first, and then the || operator works. The expression is the same as the following:

```csharp
(creditScore > 700) || (accountBalance > 9000)
```

Many programmers choose to enclose the expressions that are to the left and the right of a logical operator in parentheses, as shown here. Even though the parentheses are not required in many situations, using them makes the compound expression easier to understand.

Checking Numeric Ranges with Logical Operators

Sometimes you need to write code that determines whether a numeric value is within a specific range of values or outside a specific range of values. When determining whether a number is inside a range, it is best to use the && operator. For example, the following if statement checks the value in x to determine whether it is in the range of 20 through 40:

```csharp
if (x > 20 && x < 40)
{
    MessageBox.Show("The value is in the acceptable range.");
}
```
Chapter 4  Making Decisions

The compound Boolean expression being tested by this statement is true only when x is greater than 20 and less than 40. The value in x must be between the values of 20 and 40 for this compound expression to be true.

When determining whether a number is outside a range, it is best to use the || operator. The following statement determines whether x is outside the range of 20 through 40:

```csharp
if (x < 20 || x > 40)
{
    MessageBox.Show("The value is outside the acceptable range.");
}
```

It is important not to get the logic of the logical operators confused when testing for a range of numbers. For example, the compound Boolean expression in the following code would never test true:

```csharp
// This is an error!
if (x < 20 && x > 40)
{
    MessageBox.Show("The value is outside the acceptable range.");
}
```

Obviously, x cannot be less than 20 and at the same time be greater than 40.

Let’s look at an example application that checks the range of a value entered by the user. Open the Range Checker application in the Chap04 folder of this book’s Student Sample Programs. Figure 4-17 shows the application’s form, along with the names of some of the controls. When you run the application, you enter an integer into the inputTextBox control and click the checkButton control. If you enter a number in the range of 1 through 10, a message box appears letting you know that the number is acceptable. Otherwise, a message box appears letting you know that the number is not acceptable.

![Figure 4-17 The Range Checker application’s form](image)

The following code sample shows the checkButton_Click event handler. Line 7 declares an int variable named number, initialized with the value that has been entered into the inputTextBox control. The if statement that begins in line 10 determines whether number is greater than or equal to 1 AND number is less than or equal to 10. If the Boolean expression is true, the statement in line 12 executes. Otherwise, the statement in line 16 executes.

```csharp
private void checkButton_Click(object sender, EventArgs e)
{
    try
    {
        // Declare a variable and initialize it with
        // the user's input.
```
```csharp
    int number = int.Parse(inputTextBox.Text);

    // Check the number's range.
    if (number >= 1 && number <= 10)
    {
        MessageBox.Show("That number is acceptable.");
    }
    else
    {
        MessageBox.Show("That number is NOT acceptable.");
    }

    catch (Exception ex)
    {
        // Display an error message.
        MessageBox.Show(ex.Message);
    }
```

**Checkpoint**

4.12 What is a compound Boolean expression?

4.13 The following truth table shows various combinations of the values `true` and `false` connected by a logical operator. Complete the table by circling T or F to indicate whether the result of such a combination is `true` or `false`.

<table>
<thead>
<tr>
<th>Logical Expression</th>
<th>Result (circle T or F)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>true &amp;&amp; false</code></td>
<td>T F</td>
</tr>
<tr>
<td><code>true &amp;&amp; true</code></td>
<td>T F</td>
</tr>
<tr>
<td><code>false &amp;&amp; true</code></td>
<td>T F</td>
</tr>
<tr>
<td><code>false &amp;&amp; false</code></td>
<td>T F</td>
</tr>
<tr>
<td>`true</td>
<td></td>
</tr>
<tr>
<td>`true</td>
<td></td>
</tr>
<tr>
<td>`false</td>
<td></td>
</tr>
<tr>
<td>`false</td>
<td></td>
</tr>
<tr>
<td><code>! true</code></td>
<td>T F</td>
</tr>
<tr>
<td><code>! false</code></td>
<td>T F</td>
</tr>
</tbody>
</table>

4.14 Assume the variables `a = 2`, `b = 4`, and `c = 6`. Circle T or F for each of the following conditions to indicate if it is true or false.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Result (circle T or F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>`a == 4</td>
<td></td>
</tr>
<tr>
<td><code>6 &lt;= c &amp;&amp; a &gt; 3</code></td>
<td>T F</td>
</tr>
<tr>
<td><code>! = b &amp;&amp; c != 3</code></td>
<td>T F</td>
</tr>
<tr>
<td>`a &gt;= -1</td>
<td></td>
</tr>
<tr>
<td><code>!(a &gt; 2)</code></td>
<td>T F</td>
</tr>
</tbody>
</table>

4.15 Explain how short-circuit evaluation works with the `&&` and `||` operators.

4.16 Write an `if` statement that displays the message “The number is valid” in a message box if the variable `speed` is within the range 0 through 200.

4.17 Write an `if` statement that displays the message “The number is not valid” in a message box if the variable `speed` is outside the range 0 through 200.
4.5 bool Variables and Flags

**CONCEPT:** You can store the values true and false in bool variables, which are commonly used as flags.

The C# language provides the bool data type that you can use to create variables that hold true or false values. Here is an example of the declaration of a bool variable:

```csharp
bool grandMaster;
```

This declares a bool variable named grandMaster. In the program we can assign the values true or false to the variable, as shown here:

```csharp
if (points > 5000)
{   
    grandMaster = true;
}
else
{
    grandMaster = false;
}
```

Variables of the bool data type are commonly used as flags. A flag is a variable that signals when some condition exists in the program. When the flag variable is set to false, it indicates that the condition does not yet exist. When the flag variable is set to true, it means the condition does exist. For example, the previous code might be used in a game to determine whether the user is a “grand master.” If he or she has earned more than 5,000 points, we set the grandMaster variable to true. Otherwise, we set the variable to false. Later in the program we can test the grandMaster variable, like this:

```csharp
if (grandMaster)
{   
    powerLevel += 500;
}
```

This code performs the following: if grandMaster is true, add 500 to powerLevel. Here is another example:

```csharp
if (!grandMaster)
{
    powerLevel = 100;
}
```

This code performs the following: if grandMaster is not true, set powerLevel to 100.

**Checkpoint**

4.18 What special values can you store in a bool variable?

4.19 What is a flag variable?

4.6 Comparing Strings

**CONCEPT:** You can use certain relational operators and methods to compare strings.
You can use the `==` operator to compare two strings. For example, look at the following code sample:

```csharp
string name1 = "Mary";
string name2 = "Mark";
if (name1 == name2)
{
    MessageBox.Show("The names are the same." Usuarios);
} else
{
    MessageBox.Show("The names are NOT the same.");
}
```

The `==` operator compares `name1` and `name2` to determine whether they are equal. Because the strings “Mary” and “Mark” are not equal, the `else` clause displays the message “The names are NOT the same.”

You can compare string variables with string literals as well. Assume `month` is a string variable. The following code sample uses the `!=` operator to determine whether `month` is not equal to "October".

```csharp
if (month != "October")
{
    statement;
}
```

Look at the Secret Word application in the Chap04 folder of this book’s Student Sample Programs. Figure 4-18 shows the application’s form, with the names of some of the controls. The form prompts you to enter the secret word into the `inputTextBox` control. When you click the `checkButton` control, the application compares the string that you entered to "Ariel."

![Figure 4-18 The Secret Word application’s form](image)

The following code sample shows the `checkButton_Click` event handler. Line 5 declares a string variable named `secretWord`, initialized with the value that has been entered into the `inputTextBox` control. The `if` statement that begins in line 8 compares the `secretWord` variable to the string literal "Ariel". If the two are equal, the statement in line 10 executes. Otherwise, the statement in line 14 executes.

```csharp
private void checkButton_Click(object sender, EventArgs e)
{
    // Declare a string variable and initialize it with
    // the user's input.
    string secretWord = inputTextBox.Text;
    
    // Did the user enter the correct secret word?
    if (secretWord == "Ariel")
    {
        // Statement 10
    } else
    {
        // Statement 14
    }
```
Other String Comparisons

In addition to determining whether strings are equal or not equal, you can use the `String.Compare` method to determine whether one string is greater than or less than another string. This is a useful capability because sometimes you need to sort strings in some order. Before we look at how the method works, we should review how characters are stored in memory.

Recall from Chapter 1 that computers do not actually store characters, such as A, B, C, and so on, in memory. Instead, they store numeric codes that represent the characters. We mentioned in Chapter 1 that C# uses Unicode to represent characters. Here are some facts about the Unicode system:

- The uppercase characters A through Z are represented by the numbers 65 through 90.
- The lowercase characters a through z are represented by the numbers 97 through 122.
- When the digits 0 through 9 are stored in memory as characters, they are represented by the numeric codes 48 through 57. (For example, the string “abc123” is stored in memory as the codes 97, 98, 99, 49, 50, and 51.)
- A blank space is represented by the number 32.

In addition to establishing a set of numeric codes to represent characters in memory, Unicode also establishes an order for characters. The character A comes before the character B, which comes before the character C, and so on.

When a program compares characters, it actually compares the codes for the characters. The character A would be considered less than the character B because the character A’s numeric code is less than the character B’s numeric code.

Let’s look at how strings containing more than one character are compared. Suppose we have the strings “Mary” and “Mark” stored in memory, as follows:

```csharp
string name1 = "Mary";
string name2 = "Mark";
```

Figure 4-19 shows how the strings “Mary” and “Mark” are stored in memory using character codes.

![Figure 4-19](image-url)

When you compare these strings in a program, they are compared character-by-character, beginning with the first, or leftmost, characters. This is shown in Figure 4-20.
Here is how the comparison takes place:

1. The M in “Mary” is compared with the M in “Mark.” These are the same, so the next characters are compared.
2. The a in “Mary” is compared with the a in “Mark.” Because these are the same, the next characters are compared.
3. The r in “Mary” is compared with the r in “Mark.” These are the same, so the next characters are compared.
4. The y in “Mary” is compared with the k in “Mark.” Because these are not the same, the two strings are not equal. The character y has a higher character code (121) than k (107), so it is determined that the string “Mary” is greater than the string “Mark.”

If one of the strings in a comparison is shorter than the other, only the corresponding characters are compared. If the corresponding characters are identical, then the shorter string is considered less than the longer string. For example, suppose the strings “High” and “Hi” are compared. The string “Hi” is considered less than “High” because it is shorter.

In C# you cannot use relational operators to determine whether one string is greater than or less than another string. Instead, you use the `String.Compare` method. You use the following general format to call the method:

```
String.Compare(string1, string2)
```

In the general format, `string1` and `string2` are the strings that are being compared. The method returns an integer value indicating the result of the comparison. The integer value will be one of the following:

- Greater than zero if `string1` is greater than `string2`.
- Zero if `string1` is equal to `string2`.
- Less than zero if `string1` is less than `string2`.

Here is a code sample that uses the method to display two names in alphabetical order:

```
1 string str1 = "Joe";
2 string str2 = "Kerry";
3
4 if (String.Compare(str1, str2) < 0)
5 {
6     MessageBox.Show(str1 + " " + str2);
7 }
8 else
9 {
10    MessageBox.Show(str2 + " " + str1);
11 }
```

The `if` statement in line 4 calls the `String.Compare` method, passing `str1` and `str2` as arguments. If we execute this code, the method will return a value that is less than 0 because the string “Joe” is less than the string “Kerry”. As a result, the statement in line 6 will display Joe Kerry.

The `String.Compare` method performs a case sensitive comparison, which means that uppercase characters are not considered the same as their lowercase counterparts. For
example, the strings “Joe” and “joe” are not equal because the case of the first character
is different in each. You can pass the Boolean value true as an optional third argument
to the String.Compare method if you want it to perform a case insensitive comparison.
Here is an example:

```csharp
1 string str1 = "JOE";
2 string str2 = "joe";
3 if (String.Compare(str1, str2, true) == 0)
4 {
5     MessageBox.Show(str1 + " and " + str2 + " are equal.");
6 }
7 else
8 {
9     MessageBox.Show(str1 + " and " + str2 + " are NOT equal.");
10 }
```

Notice that the if statement in line 4 passes true as the third argument to the String.
Compare method. This specifies that we want a case insensitive comparison. As a re-
sult, the method will return 0 and the statement in line 6 will display JOE and joe are
equal.

**Checkpoint**

4.20 If the following code were part of a complete program, what would it display?
```csharp
if (String.Compare("z", "a") < 0)
{
    MessageBox.Show("z is less than a.");
}
else
{
    MessageBox.Show("z is not less than a.");
}
```

4.21 If the following code were part of a complete program, what would it display?
```csharp
string s1 = "New York";
string s2 = "Boston";
if (String.Compare(s1, s2) > 0)
{
    MessageBox.Show(s2);
    MessageBox.Show(s1);
}
else
{
    MessageBox.Show(s1);
    MessageBox.Show(s2);
}
```

### 4.7 Preventing Data Conversion Exceptions with the TryParse Methods

**CONCEPT:** Exceptions should be prevented when possible. You can use the
TryParse methods to prevent exceptions as a result of the user entering
invalid data.
In Chapter 3 you learned that the **Parse** methods throw an exception when you try to use them to convert nonnumeric data to a numeric data type. If you use one of the **Parse** methods to convert a TextBox control’s Text property to a number, there is always the possibility of an exception being thrown. After all, the user is free to enter anything he or she wants into a TextBox control. To handle the exceptions that are caused by the **Parse** methods, we have been using the **try-catch** statement.

Although many exceptions happen for reasons that the programmer cannot anticipate (such as a system malfunction), some exceptions are predictable. For example, you know that using a **Parse** method to convert nonnumeric input to a numeric data type will throw an exception. In situations like that, where an exception is predictable, you should write your code to prevent the exception. It is a better programming practice to prevent an exception instead of allowing it to happen and then letting a **try-catch** statement react to it. You should use **try-catch** statements primarily for those exceptions that are beyond your control.

**NOTE:** Until now, we’ve simply been allowing exceptions to happen and letting a **try-catch** statement respond to them. After reading the previous paragraph, you might be wondering why we haven’t been preventing exceptions all along. The reason is that you need to know how to write **if** statements to perform the techniques that we discuss in this section. Now that you know how to write **if** statements, you can add more sophistication to your code.

Now that you know how to write **if** statements, you can use a family of methods in the .NET Framework known as the **TryParse** methods. With the **TryParse** methods, you can determine whether a string (such as a control’s Text property) contains a value that can be converted to a specific data type before it is converted to that data type. The **TryParse** methods do not throw an exception, so you can use them without a **try-catch** statement.

There are several **TryParse** methods in the .NET Framework. For now, we are using the **int**, **double**, and **decimal** numeric data types, so we will discuss three of them:

- We use the **int.TryParse** method to convert a string to an **int**.
- We use the **double.TryParse** method to convert a string to a **double**.
- We use the **decimal.TryParse** method to convert a string to a **decimal**.

When you call one of the **TryParse** methods, you pass two arguments: (1) the string that you want to convert, and (2) the name of the variable in which you want to store the converted value. First, let’s look at the **int.TryParse** method. Here is the general format of how the **int.TryParse** method is called:

```csharp
int.TryParse(string, out targetVariable)
```

In the general format, **string** is the string that you want to convert, and **targetVariable** is the name of an **int** variable. The method tries to convert the **string** argument to an **int**. If the conversion is successful, the converted value is stored in the **targetVariable**, and the method returns the Boolean value **true** to indicate that the conversion was successful. If the conversion is not successful, the method does not throw an exception. Instead, it stores 0 in the **targetVariable** and returns the Boolean value **false** to indicate that the **string** could not be converted.

Look carefully at the general format and notice that the word **out** appears before the **targetVariable**. The **out** keyword is required, and it specifies that the **targetVariable** is an output variable. An **output variable** is a variable that is passed as an argument to a method, and when the method is finished, a value is stored in the variable.

Because the **TryParse** methods return either **true** or **false**, they are commonly called as the Boolean expression in an **if** statement. The following code shows an example using...
the `int.TryParse` method. In the example, assume that `inputTextBox` is the name of a TextBox control.

```csharp
1 int number;
2 if (int.TryParse(inputTextBox.Text, out number))
3 {
4     MessageBox.Show("Success!");
5 }
6 else
7 {
8     MessageBox.Show("Enter a valid integer.");
9 }
```

The purpose of this code sample is to convert the value of the `inputTextBox` control's Text property to an `int` and assign that value to the `number` variable. In line 3, the `if` statement calls the `int.TryParse` method, passing `inputTextBox.Text` as argument 1 and `number` as argument 2. Here's what happens:

- If `inputTextBox.Text` is successfully converted to an `int`, the resulting value is assigned to the `number` variable, and the method returns `true`. That causes the statement in line 5 to execute.
- If `inputTextBox.Text` cannot be converted to an `int`, the value 0 is assigned to the `number` variable, and the method returns `false`. That causes the statement in line 9 (after the `else` clause) to execute.

The other `TryParse` methods work in a similar manner. Here is the general format of how the `double.TryParse` method is called:

```csharp
double.TryParse(string, out targetVariable)
```

In the general format, `string` is the string that you want to convert, and `targetVariable` is the name of a `double` variable. If the `string` can be converted to a `double`, its value is stored in the `targetVariable`, and the method returns the Boolean value `true` to indicate that the conversion was successful. If the conversion was not successful, the method stores 0 in the `targetVariable` and returns the Boolean value `false` to indicate that the `string` could not be converted.

The following code shows an example using the `double.TryParse` method. In the example, assume that `inputTextBox` is the name of a TextBox control.

```csharp
1 double number;
2 if (double.TryParse(inputTextBox.Text, out number))
3 {
4     MessageBox.Show("Success!");
5 }
6 else
7 {
8     MessageBox.Show("Enter a valid double.");
9 }
```

Here is the general format of how the `decimal.TryParse` method is called:

```csharp
decimal.TryParse(string, out targetVariable)
```

In the general format, `string` is the string that you want to convert, and `targetVariable` is the name of a `decimal` variable. If the `string` can be converted to a `decimal`, its value is stored in the `targetVariable`, and the method returns the Boolean value `true` to indicate that the conversion was successful. If the conversion was not successful, the method stores 0 in the `targetVariable` and returns the Boolean value `false` to indicate that the `string` could not be converted.
The following code shows an example using the `decimal.TryParse` method. In the example, assume that `inputTextBox` is the name of a TextBox control.

```csharp
decimal number;
if (decimal.TryParse(inputTextBox.Text, out number))
{
    MessageBox.Show("Success!");
} else
{
    MessageBox.Show("Enter a valid decimal.");
}
```

**Validating the Data in Multiple TextBoxes**

If a form has multiple TextBoxes, then the user has multiple opportunities to enter an invalid piece of data. A well-designed program should validate the contents of each TextBox individually. When a piece of invalid data is found, the program should display an error message that tells the user specifically which TextBox contains the bad input.

This technique requires a set of nested `if` statements. For example, suppose a form has two TextBoxes. The following pseudocode shows the logic for validating each TextBox. (In the pseudocode, a set of dotted lines connects each `If` statement with its corresponding `Else` clause and its ending.)

```
If the data in the first TextBox is good, then
  If the data in the second TextBox is good, then
    Process the data in both TextBoxes
  Else
    Display an error message about the second TextBox
  End if
Else
  Display an error message about the first TextBox
End if
```

Let's see how that logic looks in actual C# code. In the `Chap04` folder of this book’s Student Sample Programs, you will find a project named `Add Two Numbers`. The application’s form is shown in Figure 4-21. When you run the application, enter an integer into each of the TextBox controls and then click the `Add` button. A message box will appear showing the sum of the two numbers. If you enter anything other than an integer into either TextBox, an error message will appear telling you which TextBox contains the invalid data.

**Figure 4-21** The `Add Two Numbers` form
Here is the code for the addButton_Click event handler:

```csharp
private void addButton_Click(object sender, EventArgs e)
{
    // Local variables
    int first, second, sum;

    if (int.TryParse(firstTextBox.Text, out first))
    {
        if (int.TryParse(secondTextBox.Text, out second))
        {
            // Add the two numbers and display the sum.
            sum = first + second;
            MessageBox.Show(sum.ToString());
        }
        else
        {
            // Display an error message about the second TextBox.
            MessageBox.Show("The second TextBox contains invalid data.");
        }
    }
    else
    {
        // Display an error message about the first TextBox.
        MessageBox.Show("The first TextBox contains invalid data.");
    }
}
```

Let’s take a closer look:

- Line 4 declares three int variables: first, second, and sum.
- The if statement in line 6 tries to convert firstTextBox.Text to an int. If the conversion is successful, the result is stored in the first variable, and the program continues executing at line 8. If the conversion is not successful, the program jumps to the else clause in line 20, and line 23 displays an error message regarding the first TextBox control.
- The if statement in line 8 tries to convert secondTextBox.Text to an int. If the conversion is successful, the result is stored in the second variable, and the program continues executing at line 10. If the conversion is not successful, the program jumps to the else clause in line 14, and line 17 displays an error message regarding the second TextBox control.
- The statements in lines 11 and 12 execute only if both TextBox controls contain valid integer values. These statements add the first and second variables and display their sum.

If you need to validate three TextBox controls, you will write a set of three nested if statements. Here’s the pseudocode:

- If the data in the first TextBox is good, then
  - If the data in the second TextBox is good, then
    - If the data in the third TextBox is good, then
      Process the data in all three TextBoxes
    - Else
      Display an error message about the third TextBox
  - Else
    Display an error message about the second TextBox
  - End if
- Else
  Display an error message about the first TextBox
- End if
- Else
  Display an error message about the first TextBox
- End if
In Tutorial 4-4 you will complete an application that uses the TryParse methods to validate data entered into two TextBox controls.

**Tutorial 4-4:**
Calculating Fuel Economy

In Tutorial 3-2, you created an application that calculates a car’s fuel economy in miles per gallon (MPG). Recall that the application lets the user enter the number of miles he or she has driven and the gallons of gas used. The application calculates and displays the car’s MPG. In this tutorial you will create a new version of the application that validates the data entered by the user.

To save time, the project has already been started for you, and the application’s form has already been created. To complete the project, follow the steps in this tutorial.

**Step 1:** Start Visual Studio (or Visual Studio Express). Open the project named Fuel Economy with TryParse in the Chap04 folder of this book’s Student Sample Programs.

**Step 2:** Open the Form1 form in the Designer. The form is shown, along with the names of the important controls, in Figure 4-22.

**Figure 4-22** The Fuel Economy form

**Step 3:** Now you will create the Click event handlers for the Button controls. At the end of this tutorial, Program 4-4 shows the completed code for the form. You will be instructed to refer to Program 4-4 as you write the event handlers.

In the Designer, double-click the calculateButton control. This will open the code editor, and you see an empty event handler named calculateButton_Click. Complete the calculateButton_Click event handler by typing the code shown in lines 22–48 in Program 4-4.

Let’s take a closer look at the code:

**Lines 22–24:** These lines declare the double variables miles, gallons, and mpg. The variables hold the miles driven, the gallons of gas used, and the MPG, respectively.

**Line 27:** This if statement tries to convert milesTextBox.Text to a double. If the conversion is successful, the result is stored in the miles variable, and the program continues executing at line 29. If the conversion is not successful, the program jumps to the else clause in line 44, and line 47 displays the error message “Invalid input for miles.”
Line 30: This if statement tries to convert `gallonsTextBox.Text` to a double. If the conversion is successful, the result is stored in the `gallons` variable, and the program continues executing at line 32. If the conversion is not successful, the program jumps to the else clause in line 38, and line 41 displays the error message “Invalid input for gallons.”

Lines 32–36: These lines are executed only if both the `milesTextBox` and `gallonsTextBox` contain valid data. Line 33 calculates MPG and assigns the result to the `mpg` variable, and line 36 displays the value of the `mpg` variable in the `mpgLabel` control.

**Step 4:** Switch your view back to the Designer and double-click the `exitButton` control. In the code editor you see an empty event handler named `exitButton_Click`. Complete the `exitButton_Click` event handler by typing the code shown in lines 53–54 in Program 4-4.

**Step 5:** Save the project and run the application. First, enter 300 for the miles and 10 for the gallons. Click the Calculate MPG button, and the application should display 30.0 as the MPG.

Now change the miles to an invalid entry, such as 123xyz, and click the Calculate MPG button. The message “Invalid input for miles.” should appear in a message box, as shown on the left in Figure 4-23.

Now change the miles back to 300, change the gallons to an invalid entry, such as 123xyz, and click the Calculate MPG button. The message “Invalid input for gallons.” should appear in a message box, as shown on the right in Figure 4-23. Continue to test the application as you wish. When you are finished, click the Exit button and the form should close.

**Program 4-4** Completed Form1 code for the Fuel Economy with TryParse application

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Fuel_Economy_with_TryParse
{
```
4.7 Preventing Data Conversion Exceptions with the TryParse Methods

```csharp
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }
}

private void calculateButton_Click(object sender, EventArgs e)
{
    double miles;     // To hold miles driven
    double gallons;   // To hold gallons used
    double mpg;       // To hold MPG

    // Validate the milesTextBox control.
    if (double.TryParse(milesTextBox.Text, out miles))
    {
        // Validate the gallonsTextBox control.
        if (double.TryParse(gallonsTextBox.Text, out gallons))
        {
            // Calculate MPG.
            mpg = miles / gallons;

            // Display the MPG in the mpgLabel control.
            mpgLabel.Text = mpg.ToString("n1");
        }
        else
        {
            // Display an error message for gallonsTextBox.
            MessageBox.Show("Invalid input for gallons.");
        }
    }
    else
    {
        // Display an error message for milesTextBox.
        MessageBox.Show("Invalid input for miles.");
    }
}

private void exitButton_Click(object sender, EventArgs e)
{
    // Close the form.
    this.Close();
}
```

**Checkpoint**

4.22 What value does a TryParse method return if the string argument is successfully converted? What value does it return if the string is not converted?

4.23 If a TryParse method successfully converts the string argument, where is the result stored?

4.24 If a TryParse method cannot convert the string argument, what is stored in the second argument?

4.25 What does the keyword out mean when it is written before an argument to a method call?
4.8 Input Validation

CONCEPT: Input validation is the process of inspecting data that has been input to a program to make sure it is valid before it is used in a computation.

In the previous section you learned about using the `TryParse` methods to validate the type of data entered by the user. You should also validate the accuracy of the data that is entered by the user. One of the most famous sayings among computer programmers is “garbage in, garbage out.” This saying, sometimes abbreviated as GIGO, refers to the fact that computers cannot tell the difference between good input and bad input. If a user provides bad data as input to a program, the program will process that bad data and, as a result, will produce bad data as output.

For example, consider a payroll program that accepts the number of hours that an employee has worked in a given week as input. If the payroll clerk accidentally enters 400 hours instead of 40 hours, an unusually large check will be written because there are less than 400 hours in a week! The computer, however, is unaware of this fact, and unless the program is written to catch such errors, it will process the bad data just as if it were good data.

Sometimes stories are reported in the news about computer errors that mistakenly cause people to be charged thousands of dollars for small purchases or to receive large tax refunds to which they were not entitled. These “computer errors” are rarely caused by a computer, however; they are more commonly caused by software bugs or bad data that was read into a program as input.

The integrity of a program’s output is only as good as the integrity of its input. For this reason, you should write your programs in such a way that bad input is never accepted. When input is given to a program, it should be inspected before it is processed. If the input is invalid, the program should discard it and prompt the user to enter the correct data. This process is known as input validation.

For example, in a payroll program we might validate the number of hours worked like this:

```csharp
if (int.TryParse(hoursWorkedTextBox.Text, out hours))
{
    if (hours > 0 && hours <= 168)
    {
        // Continue to process the input.
    }
    else
    {
        MessageBox.Show("Invalid number of hours entered.");
    }
}
else
{
    MessageBox.Show("The hours worked must be an integer.");
}
```

Let’s assume the application uses a TextBox named `hoursWorkedTextBox` to get the hours worked. Also assume that the variable `hours` has already been declared as an `int`. The outer `if` statement (line 1) uses the `int.TryParse` method to make sure the user has entered an integer. If so, the value is stored in the `hours` variable and the program continues to the inner `if` statement (line 3). The inner `if` statement ensures that we process the input only if `hours` is greater than 0 and `hours` is less than or equal to 168. This is because we cannot write a paycheck for 0 hours worked, and 168 is the maximum number of hours in a week.
Let’s look at another example. The following code comes from an application that gets a test score as input. A valid test score is an integer in the range of 0 through 100.

```csharp
if (int.TryParse(testScoreTextBox.Text, out testScore))
    if (testScore >= 0 && testScore <= 100)
        // Continue to process the input.
    else
        MessageBox.Show("Test score must be in the range 0 - 100.");
else
    MessageBox.Show("The test score must be an integer.");
```

Let’s assume the application uses a TextBox named testScoreTextBox to get the test score. Also assume that the variable `testScore` has already been declared as an `int`. The outer `if` statement (line 1) uses the `int.TryParse` method to make sure the user has entered an integer. If so, the value is stored in the `testScore` variable and the program continues to the inner `if` statement (line 3). The inner `if` statement ensures that we process the input only if `testScore` is greater than or equal to 0 and `testScore` is less than or equal to 100.

---

**4.9 Radio Buttons and Check Boxes**

**CONCEPT:** GUIs commonly use radio buttons and check boxes to let the user select items.

**Radio Buttons**

Radio buttons are useful when you want the user to select one choice from several possible choices. Figure 4-24 shows a form with a group of three radio buttons. The radio buttons in the figure allow the user to select Coffee, Tea, or Soft Drink.

**Figure 4-24** Radio buttons

A radio button may be either selected or deselected. Each radio button has a small circle that appears filled in when the radio button is selected, and appears empty when the radio button is deselected. In Figure 4-24, the Coffee radio button is selected and the other radio buttons are deselected.

At run time, only one radio button in a group may be selected at a time. Clicking on a radio button selects it, and automatically deselects any other radio button in the same group. We call this **mutually exclusive selection.**
When you want to create a group of radio buttons on a form, you use the **RadioButton** control, which is found in the *Common Controls* section of the *Toolbox*. RadioButton controls are normally grouped in one of the following ways:

- You place them inside a GroupBox control. All RadioButton controls that are inside a GroupBox are members of the same group.
- You place them inside a Panel control. All RadioButton controls that are inside a Panel are members of the same group.
- You place them on a form but not inside a GroupBox or a Panel. All RadioButton controls that are on a form but not inside a GroupBox or Panel are members of the same group.

Figure 4-25 shows a form with two groups of RadioButton controls. The group on the left is inside a GroupBox control, and the group on the right is inside a Panel control. When the application runs, the user will be able to select only one RadioButton from each group. In the figure, *Coffee* is selected in the left group and *Lunch* is selected in the right group.

![Image](Image.png)

**Figure 4-25** A form with two groups of RadioButton controls

### The RadioButton Control’s Text Property

RadioButton controls have a Text property, which holds the text that is displayed next to the radio button’s circle. For example, the radio buttons shown in Figure 4-24 have their Text properties set to *Coffee*, *Tea*, and *Soft Drink*.

### The RadioButton Control’s Checked Property

RadioButton controls have a Checked property that determines whether the control is selected or deselected. The Checked property is a Boolean property, which means that it may be set to either True or False. When the Checked property is set to True, the RadioButton is selected, and when the Checked property is set to False, the RadioButton is deselected. By default, the Checked property is set to False.

You can use the *Properties* window to set the initial value of a RadioButton control’s Checked property. Keep in mind that the Checked property of only one RadioButton in a group can be set to True at a given time. When you set a RadioButton control’s Checked property to True in the *Properties* window, the Checked properties of all the other RadioButton controls in the same group automatically are set to False.

---

**NOTE:** The name *radio button* refers to the old car radios that had push buttons for selecting stations. Only one button could be pushed in at a time. When you pushed a button, it automatically popped out the currently selected button.
Working with Radio Buttons in Code

In code, you can determine whether a RadioButton control is selected by testing its Checked property. For example, suppose a form has a RadioButton control named choice1RadioButton. The following if statement determines whether it is selected:

```csharp
if (choice1RadioButton.Checked)
{
    MessageBox.Show("You selected Choice 1");
}
```

Notice that we did not have to use the `==` operator to explicitly compare the Checked property to the value `true`. This code is equivalent to the following:

```csharp
if (choice1RadioButton.Checked == true)
{
    MessageBox.Show("You selected Choice 1.");
}
```

Let’s look at an example using multiple RadioButton controls. Open the RadioButton project in the Chap04 folder of this book’s Student Sample Programs. The application’s form is shown in Figure 4-26. When you run the application, select one of the radio buttons and then click the OK button. A message box appears showing the sport that you selected.

Figure 4-26 The RadioButton Example form

Here is the code for the okButton_Click event handler:

```csharp
private void okButton_Click(object sender, EventArgs e)
{
    if (footballRadioButton.Checked)
    {
        MessageBox.Show("You selected Football.");
    }
    else if (basketballRadioButton.Checked)
    {
        MessageBox.Show("You selected Basketball.");
    }
    else if (baseballRadioButton.Checked)
```
When the event handler executes, the `if` statement in line 3 determines whether the `footballRadioButton` control's Checked property is true. If it is, the message `You selected Football.` is displayed in line 5. Otherwise, line 7 determines whether the `basketballRadioButton` control's Checked property is true. If it is, the message `You selected Basketball.` is displayed in line 9. Otherwise, line 11 determines whether the `baseballRadioButton` control's Checked property is true. If it is, the message `You selected Baseball.` is displayed in line 13.

Check Boxes

A check box appears as a small box with some accompanying text. Figure 4-27 shows an example. They are called check boxes because clicking on an empty check box causes a check mark to appear in it. If a check mark already appears in a check box, clicking it removes the check mark.

![Figure 4-27 A check box](image)

Check boxes are similar to radio buttons, except that check boxes are not mutually exclusive. You can have one or more check boxes in a group, and any number of them can be selected at any given time. When you want to create a check box on a form, you use the `CheckBox` control, which is found in the Common Controls section of the Toolbox.

The CheckBox Control’s Text Property

CheckBox controls have a Text property, which holds the text that is displayed next to the check box. For example, the CheckBox control shown in Figure 4-27 has its Text property set to `Pepperoni`.

The CheckBox Control’s Checked Property

Like radio buttons, CheckBox controls have a Checked property. When a CheckBox control is selected, or checked, its Checked property is set to True. When a CheckBox control is deselected, or unchecked, its Checked property is set to False.

Working with CheckBox Controls in Code

In code, you can determine whether a CheckBox control is selected by testing its Checked property. For example, suppose a form has a CheckBox control named `option1CheckBox`. The following `if` statement determines whether it is selected:

```csharp
if (option1CheckBox.Checked)
{
    MessageBox.Show("You selected Option 1.");
}
```

Let’s look at an example program. Open the `CheckBox Example` project in the `Chap04` folder of this book’s Student Sample Programs. The application’s form is shown in Figure 4-28. When you run the application, select any of the check boxes and then click the OK button. One or more message boxes will appear, showing you the items that you selected.
Here is the code for the okButton_Click event handler:

```csharp
private void okButton_Click(object sender, EventArgs e)
{
    if (pepperoniCheckBox.Checked)
    {
        MessageBox.Show("You selected Pepperoni.");
    }

    if (cheeseCheckBox.Checked)
    {
        MessageBox.Show("You selected Cheese.");
    }

    if (anchoviesCheckBox.Checked)
    {
        MessageBox.Show("You selected Anchovies.");
    }
}
```

Notice that we have three separate if statements. The if statement in line 3 determines whether the pepperoniCheckBox control is selected. If so, line 5 displays the message You selected Pepperoni. The if statement in line 8 determines whether the cheeseCheckBox control is selected. If so, line 10 displays the message You selected Cheese. The if statement in line 13 determines whether the anchoviesCheckBox control is selected. If so, line 15 displays the message You selected Anchovies.

## The CheckedChanged Event

Any time a RadioButton or a CheckBox control’s Checked property changes, a CheckedChanged event happens for that control. If you want some action to immediately take place when the user selects (or deselects) a RadioButton or CheckBox control, you can create a CheckedChanged event handler for the control and write the desired code in that event handler.

To create a CheckedChanged event handler for a RadioButton or a CheckBox, simply double-click the control in the Designer. An empty CheckedChanged event handler is created in the code editor. You can then write code inside the event handler. Tutorial 4-5 leads you through the process.
Tutorial 4-5: Creating the Color Theme Application

In this tutorial you create an application that allows the user to select a color using RadioButton controls. When the user selects a color, the form’s background color is changed to that color immediately. Figure 4-29 shows the application’s form, with the names of all the controls.

Figure 4-29 The Color Theme form

Step 1: Start Visual Studio (or Visual Studio Express) and begin a new Windows Forms Application project named Color Theme.

Step 2: Set up the application’s form as shown in Figure 4-29. Notice that the form’s Text property is set to Color Theme. The names of the controls are shown in the figure. As you place each of the controls on the form, refer to Table 4-8 for the relevant property settings.

Table 4-8 Control property settings

<table>
<thead>
<tr>
<th>Control Name</th>
<th>Control Type</th>
<th>Property Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>colorGroupBox</td>
<td>GroupBox</td>
<td>Text: Select a Background Color</td>
</tr>
<tr>
<td>yellowRadioButton</td>
<td>RadioButton</td>
<td>Text: Yellow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Checked: False</td>
</tr>
<tr>
<td>redRadioButton</td>
<td>RadioButton</td>
<td>Text: Red</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Checked: False</td>
</tr>
<tr>
<td>whiteRadioButton</td>
<td>RadioButton</td>
<td>Text: White</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Checked: False</td>
</tr>
<tr>
<td>normalRadioButton</td>
<td>RadioButton</td>
<td>Text: Back to Normal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Checked: True</td>
</tr>
<tr>
<td>exitButton</td>
<td>Button</td>
<td>Text: Exit</td>
</tr>
</tbody>
</table>

Step 3: Once you have set up the form with its controls, you can create the Checked-Changed event handlers for the RadioButton controls. At the end of this tutorial, Program 4-5 shows the completed code for the form. You will be instructed to refer to Program 4-5 as you write the event handlers.

In the Designer, double-click the yellowRadioButton control. This opens the code editor, and you see an empty event handler named yellowRadioButton_CheckedChanged. Complete the yellowRadioButton_CheckedChanged event handler by typing the code shown in lines 22–25 in Program 4-5.
The event handler is easy to understand. The if statement in line 22 determines whether the yellowRadioButton control is checked. If so, line 24 sets the form’s background to yellow.

**Step 4:** Switch your view back to the Designer and double-click the redRadioButton control. This opens the code editor, and you see an empty event handler named redRadioButton_CheckedChanged. Complete the redRadioButton_CheckedChanged event handler by typing the code shown in lines 30–33 in Program 4-5.

**Step 5:** Switch your view back to the Designer and double-click the whiteRadioButton control. This opens the code editor, and you see an empty event handler named whiteRadioButton_CheckedChanged. Complete the whiteRadioButton_CheckedChanged event handler by typing the code shown in lines 38–41 in Program 4-5.

**Step 6:** Switch your view back to the Designer and double-click the normalRadioButton control. This opens the code editor, and you see an empty event handler named normalRadioButton_CheckedChanged. Complete the normalRadioButton_CheckedChanged event handler by typing the code shown in lines 46–49 in Program 4-5.

**Step 7:** Switch your view back to the Designer and double-click the exitButton control. In the code editor you see an empty event handler named exitButton_Click. Complete the exitButton_Click event handler by typing the code shown in lines 54–55 in Program 4-5.

**Step 8:** Save the project and run the application. Notice that the Back to Normal radio button is initially selected. That’s because you set its Checked property to True in the Properties window. Click the other Radio buttons and notice that the form’s background color changes immediately. When you are finished testing the application, click the Exit button to close it.

---

Program 4-5 Completed Form1 code for the Color Theme application

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Color_Theme
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void yellowRadioButton_CheckedChanged(object sender, EventArgs e)
        {
            if (yellowRadioButton.Checked)
```
Checkpoint

4.26 If several RadioButton controls have been created in the same GroupBox, how many of them may be selected at one time?

4.27 If several CheckBox controls have been created in the same GroupBox, how many of them may be selected at one time?

4.28 In code, how do you determine whether a RadioButton or a CheckBox control has been selected?

4.10 The switch Statement

CONCEPT: The switch statement lets the value of a variable or an expression determine which path of execution the program will take.

The switch statement is a multiple-alternative decision structure. It allows you to test the value of a variable or an expression and then use that value to determine which statement
or set of statements to execute. Figure 4-30 shows an example of how a multiple alternative decision structure looks in a flowchart.

**Figure 4-30** A multiple alternative decision structure

In the flowchart, the diamond symbol shows month, which is the name of a variable. If the month variable contains the value 1, the program displays “January.” If the month variable contains the value 2, the program displays “February.” If the month variable contains the value 3, the program displays “March.” If the month variable contains none of these values, the action that is labeled Default is executed. In this case, the program displays “Error: Invalid month.”

Here is the general format of a switch statement in C# code:

```csharp
switch (testExpression)
{
    case value_1:
        statement;
        statement;
        etc.
        break;

    case value_2:
        statement;
        statement;
        etc.
        break;

    case value_N:
        statement;
        statement;
        etc.
        break;

    default:
        statement;
        statement;
        etc.
        break;
}
```

The testExpression is a variable or expression.

These statements are executed if the testExpression is equal to value_1.

These statements are executed if the testExpression is equal to value_2.

Insert as many case sections as necessary.

These statements are executed if the testExpression is equal to value_N.

These statements are executed if the testExpression is not equal to any of the case values.
The first line of the statement starts with the word `switch`, followed by a `testExpression`, which is enclosed in parentheses. The `testExpression` is a variable or an expression that gives an integer, string, or `bool` value. (Several other data types that we have not discussed yet are also permissible. The important thing to remember is that the `testExpression` cannot be a floating-point or decimal value.)

Beginning at the next line is a block of code enclosed in curly braces. Inside this block of code are one or more `case` sections. A `case` section begins with the word `case`, followed by a value, followed by a colon. Each `case` section contains one or more statements, followed by a `break` statement. Each `case` section must end with a `break` statement. At the end is an optional `default` section. The `default` section must also end with a `break` statement.

When the `switch` statement executes, it compares the value of the `testExpression` with the values that follow each of the `case` statements (from top to bottom). When it finds a `case` value that matches the `testExpression`'s value, the program branches to the `case` statement. The statements that follow the `case` statement are executed, until a `break` statement is encountered. At that point the program jumps out of the `switch` statement. If the `testExpression` does not match any of the `case` values, the program branches to the `default` statement and executes the statements that immediately follow it.

For example, the following code performs the same operation as the flowchart in Figure 4-30:

```csharp
switch (month)
{
    case 1:
        MessageBox.Show("January");
        break;
    case 2:
        MessageBox.Show("February");
        break;
    case 3:
        MessageBox.Show("March");
        break;
    default:
        MessageBox.Show("Error: Invalid month");
        break;
}
```

In this example the `testExpression` is the `month` variable. If the value in the `month` variable is 1, the program branches to the `case 1:` section and executes the `MessageBox.Show("January")` statement that immediately follows it. If the value in the `month` variable is 2, the program branches to the `case 2:` section and executes the `MessageBox.Show("February")` statement that immediately follows it. If the value in the `month` variable is 3, the program branches to the `case 3:` section and executes the `MessageBox.Show("March")` statement that immediately follows it. If the value in the `month` variable is not 1, 2, or 3, the program branches to the `default:` section and executes the `MessageBox.Show("Error: Invalid month")` statement that immediately follows it.

The `switch` statement can be used as an alternative to an `if-else-if` statement that tests the same variable or expression for several different values. For example, the previously shown `switch` statement works like this `if-else-if` statement:
if (month == 1)
{
    MessageBox.Show("January");
}
else if (month == 2)
{
    MessageBox.Show("February");
}
else if (month == 3)
{
    MessageBox.Show("March");
}
else
{
    MessageBox.Show("Error: Invalid month");
}

To see an application that uses a switch statement, look at the Switch Example project in the Chap04 folder of this book’s Student Sample Programs.

Checkpoint

4.29 Convert the following if-else-if code to a switch statement.
if (choice == 1)
{
    MessageBox.Show("You chose 1.");
}
else if (choice == 2)
{
    MessageBox.Show("You chose 2.");
}
else if (choice == 3)
{
    MessageBox.Show("You chose 3.");
}
else
{
    MessageBox.Show("Make another choice.");
}

4.11 Introduction to List Boxes

CONCEPT: List boxes display a list of items and allow the user to select an item from the list.

A list box displays a list of items and allows the user to select one or more items from the list. In Visual C# you use the ListBox control to create a list box on an application’s form. Figure 4-31 shows a form with two ListBox controls. At run time, the user may select one of the items, causing the item to appear selected.

The topmost ListBox in Figure 4-31 does not have a scroll bar, but the bottom one does. A scroll bar appears when a ListBox contains more items than can be displayed in the
space provided. In the figure, the top ListBox has four items (Poodle, Great Dane, German Shepherd, and Terrier), and all items are displayed. The bottom ListBox shows four items (Siamese, Persian, Bobtail, and Burmese), but because it has a scroll bar, we know there are more items in the ListBox than those four.

You will find the ListBox control in the Common Controls section of the Toolbox. Once you create a ListBox control, you add items to its Items property. The items that you add to a ListBox’s Items property are displayed in the ListBox.

To store values in the Items property at design time, follow these steps:

1. Select the ListBox control in the Designer window.
2. In the Properties window, the setting for the Items property is displayed as (Collection). When you select the Items property, an ellipsis button ( ) appears.
3. Click the ellipsis button. The String Collection Editor dialog box appears, as shown in Figure 4-32.
4. Type the values that are to appear in the ListBox into the String Collection Editor dialog box. Type each value on a separate line by pressing the Enter key after each entry.
5. When you have entered all the values, click the OK button.

NOTE: Once you acquire the necessary skills, you will be able to fill the Items collection of list boxes from external data sources (such as databases).
The SelectedItem Property

When the user selects an item in a ListBox, the item is stored in the ListBox’s SelectedItem property. For example, suppose an application has a ListBox control named fruitListBox and a string variable named selectedFruit. The fruitListBox control contains the items Apples, Pears, and Bananas. If the user has selected Pears, the following statement assigns the string “Pears” to the variable selectedFruit:

```csharp
selectedFruit = fruitListBox.SelectedItem.ToString();
```

Notice that you have to call the SelectedItem property’s ToString method to retrieve the value as a string.

Determining Whether an Item Is Selected

An exception will occur if you try to get the value of a ListBox’s SelectedItem property when no item is selected in the ListBox. For that reason, you should always make sure that an item is selected before reading the SelectedItem property. You do this with the SelectedIndex property.

The items that are stored in a ListBox each have an index. The index is simply a number that identifies the item’s position in the ListBox. The first item has the index 0, the next has the index 1, and so on. The last index value is \( n - 1 \), where \( n \) is the number of items in the ListBox. When the user selects an item in a ListBox, the item’s index is stored in the ListBox’s SelectedIndex property. If no item is selected in the ListBox, the SelectedIndex property is set to –1.

You can use the SelectedIndex property to make sure that an item is selected in a ListBox before you try to get the value of the SelectedItem property. You simply make sure the SelectedIndex property is not set to –1 before trying to read the SelectedItem property. Here is an example:

```csharp
if (fruitListBox.SelectedIndex != -1)
{
    selectedFruit = fruitListBox.SelectedItem.ToString();
}
```

In Tutorial 4-6 you will create an application that lets the user select an item from a ListBox control.

**Tutorial 4-6: Creating the Time Zone Application**

In this tutorial you create an application that allows the user to select a city from a ListBox control. When the user clicks a button, the application displays the name of the city’s time zone. Figure 4-33 shows the application’s form, with the names of all the controls.

**Step 1:** Start Visual Studio (or Visual Studio Express) and begin a new Windows Forms Application project named Time Zone.

**Step 2:** Set up the application’s form, as shown in Figure 4-33. Notice that the form’s Text property is set to Time Zone. The names of the controls are shown in the figure. As you place each of the controls on the form, refer to Table 4-9 for the relevant property settings.
Step 3: Once you have set up the form with its controls, you can create the Click event handlers for the Button controls. At the end of this tutorial, Program 4-6 shows the completed code for the form. You will be instructed to refer to Program 4-6 as you write the event handlers.

In the Designer, double-click the okButton control. This opens the code editor, and you see an empty event handler named okButton_Click. Complete the okButton_Click event handler by typing the code shown in lines 22–53 in Program 4-6. Let’s take a closer look at the code:

**Line 22:** This line declares a string variable named city. It is used to hold the name of the city that the user selects from the ListBox.

**Line 24:** This if statement determines whether the user has selected an item in the cityListBox control. If an item is selected, the control’s SelectedIndex property is set to the item’s index (a value of 0 or greater), and the program continues to line 26. If no item is selected, however, the control’s SelectedIndex property is set to −1, and the program jumps to the else clause in line 49.
4.11 Introduction to List Boxes

Line 27: This statement gets the selected item from the ListBox and assigns it to the city variable.

Line 30: This switch statement tests the city variable and branches to one of its case statements, depending on the variable’s value:

• If the city variable equals "Honolulu", the program jumps to the case statement in line 32.
• If the city variable equals "San Francisco", the program jumps to the case statement in line 35.
• If the city variable equals "Denver", the program jumps to the case statement in line 38.
• If the city variable equals "Minneapolis", the program jumps to the case statement in line 41.
• If the city variable equals "New York", the program jumps to the case statement in line 44.

Step 4:  Switch your view back to the Designer and double-click the exitButton control. In the code editor you see an empty event handler named exitButton_Click. Complete the exitButton_Click event handler by typing the code shown in lines 58–59 in Program 4-6.

Step 5:  Save the project and run the application. Select a city in the ListBox control and click the OK button to see its time zone. Test each city, and when you are finished, click the Exit button and the form should close.

Program 4-6  Completed Form1 code for the Time Zone application

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Time_Zone
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void okButton_Click(object sender, EventArgs e)
        {
            string city; // To hold the name of a city

            if (cityListBox.SelectedIndex != -1)
            {
                // Get the selected item.
                city = cityListBox.SelectedItem.ToString();
                // Determine the time zone.
            }
        }
    }
}
```
Checkpoint

4.30 How do you add items to a ListBox control using the Properties window?
4.31 How do you get the item that is selected in a ListBox?
4.32 How can you determine whether an item has been selected in a ListBox?
Key Terms

- \&\& operator
- || operator
- bool data type
- Boolean expression
- check box
- CheckBox control
- Checked property
- CheckedChanged event
- conditionally executed
- control structure
- decimal.TryParse method
- decision structure
- double.TryParse method
- dual-alternative decision structure
- flag
- if-else statement
- if-else-if statement
- index
- input validation
- int.TryParse method
- Items property
- ListBox control

Review Questions

Multiple Choice

1. A _______ structure executes a set of statements only under certain circumstances.
   a. sequence
   b. circumstantial
   c. decision
   d. Boolean

2. A _______ structure provides one alternative path of execution.
   a. sequence
   b. single-alternative decision
   c. one-path alternative
   d. single-execution decision

3. A(n) _______ expression has a value of either true or false.
   a. binary
   b. decision
   c. unconditional
   d. Boolean

4. The symbols >, <, and == are all _______ operators.
   a. relational
   b. logical
   c. conditional
   d. ternary
5. A __________ structure tests a condition and then takes one path if the condition is true or another path if the condition is false.
   a. multibranch statement
   b. single-alternative decision
   c. dual-alternative decision
   d. sequence

6. You use a(n) __________ statement to write a single-alternative decision structure.
   a. test-jump
   b. if
   c. if-else
   d. if-call

7. You use a(n) __________ statement to write a dual alternative decision structure.
   a. test-jump
   b. if
   c. if-else
   d. if-call

8. A __________ decision structure is written inside another decision structure.
   a. nested
   b. tiered
   c. dislodged
   d. hierarchical

9. &&, ||, and ! are __________ operators.
   a. relational
   b. logical
   c. conditional
   d. ternary

10. A compound Boolean expression created with the __________ operator is true only if both of its subexpressions are true.
    a. &&
    b. ||
    c. !
    d. both

11. A compound Boolean expression created with the __________ operator is true if either of its subexpressions is true.
    a. &&
    b. ||
    c. !
    d. either

12. The __________ operator takes a Boolean expression as its operand and reverses its logical value.
    a. &&
    b. ||
    c. !
    d. either

13. A __________ is a Boolean variable that signals when some condition exists in the program.
    a. flag
    b. signal
14. The ________ family of methods can be used to convert a string to a specific data type without throwing an exception.
   a. TryConvert
   b. Parse
   c. TryParse
   d. SafeConvert

15. If several ________ controls exist in a GroupBox, only one of them may be selected at a time.
   a. CheckBox
   b. RadioButton
   c. ListBox
   d. SelectionButton

16. You use the ________ statement to create a multiple alternative decision structure.
   a. menu
   b. branch
   c. select
   d. switch

17. The ________ section of a switch statement is branched to if none of the case values match the test expression.
   a. else
   b. default
   c. case
   d. otherwise

18. A ListBox’s index numbering starts at ________.
   a. 0
   b. 1
   c. −1
   d. any value you specify

19. You can use the ________ property to determine whether an item is selected in a ListBox.
   a. Index
   b. SelectedItem
   c. SelectedIndex
   d. Items.SelectedIndex

20. The ________ property holds the item that is selected in a ListBox control.
   a. Index
   b. SelectedItem
   c. SelectedIndex
   d. Items.SelectedIndex

**True or False**

1. You can write any program using only sequence structures.

2. A single-alternative decision structure tests a condition and then takes one path if the condition is true or another path if the condition is false.

3. The if-else statement is a dual-alternative decision structure.
4. A decision structure can be nested inside another decision structure.

5. A compound Boolean expression created with the && operator is true only when both subexpressions are true.

6. The TryParse methods throw an exception if the string argument cannot be converted.

7. Multiple CheckBox controls in the same GroupBox can be selected at the same time.

8. The test expression in a switch statement can be a double or a decimal value.

9. If an item is not selected in a ListBox, the control’s SelectedIndex property will be set to 0.

10. To store items in a ListBox, you add them to the control’s Text property.

**Short Answer**

1. What is meant by the term conditionally executed?

2. You need to test a condition and then execute one set of statements if the condition is true. If the condition is false, you need to execute a different set of statements. What structure will you use?

3. Briefly describe how the && operator works.

4. Briefly describe how the || operator works.

5. When determining whether a number is inside a range, which logical operator is it best to use?

6. What is a flag and how does it work?

7. What are the two arguments that you pass to a TryParse method?

8. How do you determine in code whether a RadioButton control or a CheckBox control is selected?

9. How do you add items to a ListBox using the Properties window?

10. How can you read the selected item from a ListBox while preventing an exception from occurring if no item is selected?

**Algorithm Workbench**

1. Write an if statement that assigns 20 to the variable y and assigns 40 to the variable z if the variable x is greater than 100.

2. Write an if statement that assigns 0 to the variable b and assigns 1 to the variable c if the variable a is less than 10.

3. Write an if-else statement that assigns 0 to the variable b if the variable a is less than 10. Otherwise, it should assign 99 to the variable b.

4. Write nested decision structures that perform the following: if amount1 is greater than 10 and amount2 is less than 100, display the greater of amount1 and amount2.

5. Write an if-else statement that displays “Speed is normal” if the value of the speed variable is at least 24 but no more than 56. If the speed variable’s value is outside this range, display “Speed is abnormal.”

6. Write an if-else statement that determines whether the value of the points variable is less than 9 or greater than 51. If this is true, display “Invalid points.” Otherwise, display “Valid points.”
7. Assume `pointsTextBox` is the name of a TextBox control and `points` is the name of an `int` variable. Write an `if-else` statement that uses one of the `TryParse` methods to convert the `pointsTextBox` control’s `Text` property to an `int` and stores the result in the `points` variable. If the conversion is not successful, display an error message in a message box.

8. Rewrite the following `if-else-if` statement as a `switch` statement.

```csharp
if (selection == 1)
{
    MessageBox.Show("You selected 1.");
}
else if (selection == 2)
{
    MessageBox.Show("You selected 2.");
}
else if (selection == 3)
{
    MessageBox.Show("You selected 3.");
}
else if (selection == 4)
{
    MessageBox.Show("You selected 4.");
}
else
{
    MessageBox.Show("Not good with numbers, eh?");
}
```

9. Assume `nameListBox` is a ListBox control. Write code that reads the selected item from the ListBox. Be sure to prevent an exception from occurring in case no item has been selected.

---

**Programming Problems**

1. **Roman Numeral Converter**

Create an application that allows the user to enter an integer between 1 and 10 into a TextBox control. The program should display the Roman numeral version of that number. If the number is outside the range of 1 through 10, the program should display an error message.

The following table lists the Roman numerals for the numbers 1 through 10.

<table>
<thead>
<tr>
<th>Number</th>
<th>Roman Numeral</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I</td>
</tr>
<tr>
<td>2</td>
<td>II</td>
</tr>
<tr>
<td>3</td>
<td>III</td>
</tr>
<tr>
<td>4</td>
<td>IV</td>
</tr>
<tr>
<td>5</td>
<td>V</td>
</tr>
<tr>
<td>6</td>
<td>VI</td>
</tr>
<tr>
<td>7</td>
<td>VII</td>
</tr>
<tr>
<td>8</td>
<td>VIII</td>
</tr>
<tr>
<td>9</td>
<td>IX</td>
</tr>
<tr>
<td>10</td>
<td>X</td>
</tr>
</tbody>
</table>
2. **Mass and Weight**

Scientists measure an object’s mass in kilograms and its weight in Newtons. If you know the amount of mass of an object, you can calculate its weight, in Newtons, with the following formula:

\[ \text{Weight} = \text{Mass} \times 9.8 \]

Create an application that lets the user enter an object’s mass and then calculates its weight. If the object weighs more than 1000 Newtons, display a message indicating that it is too heavy. If the object weighs less than 10 Newtons, display a message indicating that it is too light.

3. **Magic Dates**

The date June 10, 1960, is special because when it is written in the following format, the month times the day equals the year:

6/10/60

Create an application that lets the user enter a month (in numeric form), a day, and a two-digit year. The program should then determine whether the month times the day equals the year. If so, it should display a message saying the date is magic. Otherwise, it should display a message saying the date is not magic.

4. **Color Mixer**

The colors red, blue, and yellow are known as the primary colors because they cannot be made by mixing other colors. When you mix two primary colors, you get a secondary color, as shown here:

- When you mix red and blue, you get purple.
- When you mix red and yellow, you get orange.
- When you mix blue and yellow, you get green.

Create an application that lets the user select two primary colors from two different sets of Radio buttons. The form should also have a Mix button. When the user clicks the Mix button, the form’s background should change to the color that you get when you mix the two selected primary colors. Figure 4-34 shows an example of how the form should appear.

**Figure 4-34** The Color Mixer form

![Color Mixer form](image)

*Note: If the user picks the same color from both sets of Radio buttons, set the form’s background to that color.*

5. **Distance Converter**

In the English measurement system, 1 yard equals 3 feet and 1 foot equals 12 inches. Use this information to create an application that lets the user convert distances to and from inches, feet, and yards.
Figure 4-35 shows an example of how the application’s form might appear. In the example, the user enters the distance to be converted into a TextBox. A ListBox allows the user to select the units being converted from, and another ListBox allows the user to select the units being converted to.

![Distance Converter form](image)

Note: Be sure to handle the situation where the user picks the same units from both list boxes. The converted value will be the same as the value entered.

6. **Book Club Points**

   Serendipity Booksellers has a book club that awards points to its customers based on the number of books purchased each month. The points are awarded as follows:
   - If a customer purchases 0 books, he or she earns 0 points.
   - If a customer purchases 1 book, he or she earns 5 points.
   - If a customer purchases 2 books, he or she earns 15 points.
   - If a customer purchases 3 books, he or she earns 30 points.
   - If a customer purchases 4 or more books, he or she earns 60 points.

   Create an application that lets the user enter the number of books that he or she has purchased this month and displays the number of points awarded.

7. **Software Sales**

   A software company sells a package that retails for $99. Quantity discounts are given according to the following table:

<table>
<thead>
<tr>
<th>Quantity</th>
<th>Discount</th>
</tr>
</thead>
<tbody>
<tr>
<td>10–19</td>
<td>20%</td>
</tr>
<tr>
<td>20–49</td>
<td>30%</td>
</tr>
<tr>
<td>50–99</td>
<td>40%</td>
</tr>
<tr>
<td>100 or more</td>
<td>50%</td>
</tr>
</tbody>
</table>

   Create an application that lets the user enter the number of packages purchased. The program should then display the amount of the discount (if any) and the total amount of the purchase after the discount.

8. **Body Mass Index Program Enhancement**

   In Programming Problem 6 in Chapter 3, you were asked to create an application that calculates a person’s body mass index (BMI). Recall from that exercise that the BMI is often used to determine whether a person is overweight or underweight for their height. A person’s BMI is calculated with the following formula:

   $$BMI = \frac{Weight \times 703}{Height^2}$$

   In the formula, weight is measured in pounds and height is measured in inches. Enhance the program so it displays a message indicating whether the person has optimal weight, is underweight, or is overweight. A person’s weight is considered
to be optimal if his or her BMI is between 18.5 and 25. If the BMI is less than 18.5, the person is considered to be underweight. If the BMI value is greater than 25, the person is considered to be overweight.

9. **Change for a Dollar Game**

Create a change-counting game that gets the user to enter the number of coins required to make exactly one dollar. The program should let the user enter the number of pennies, nickels, dimes, and quarters. If the total value of the coins entered is equal to one dollar, the program should congratulate the user for winning the game. Otherwise, the program should display a message indicating whether the amount entered was more than or less than one dollar.

10. **Fat Percentage Calculator**

One gram of fat has 9 calories. If you know the number of fat grams in a particular food, you can use the following formula to calculate the number of calories that come from fat in that food:

\[
\text{Calories from fat} = \text{Fat grams} \times 9
\]

If you know the food’s total calories, you can use the following formula to calculate the percentage of calories from fat:

\[
\text{Percentage of calories from fat} = \frac{\text{Calories from fat}}{\text{Total calories}}
\]

Create an application that allows the user to enter:
- The total number of calories for a food item
- The number of fat grams in that food item

The application should calculate and display:
- The number of calories from fat
- The percentage of calories that come from fat

Also, the application’s form should have a CheckBox that the user can check if he or she wants to know whether the food is considered low fat. (If the calories from fat are less than 30% of the total calories of the food, the food is considered low fat.)

Use the following test data to determine if the application is calculating properly:

<table>
<thead>
<tr>
<th>Calories and Fat</th>
<th>Percentage Fat</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 calories, 8 fat grams</td>
<td>Percentage of calories from fat: 36%</td>
</tr>
<tr>
<td>150 calories, 2 fat grams</td>
<td>Percentage of calories from fat: 12% (a low-fat food)</td>
</tr>
<tr>
<td>500 calories, 30 fat grams</td>
<td>Percentage of calories from fat: 54%</td>
</tr>
</tbody>
</table>

Note: Make sure the number of calories and fat grams are not less than 0. Also, the number of calories from fat cannot be greater than the total number of calories. If that happens, display an error message indicating that either the calories or fat grams were incorrectly entered.

11. **Time Calculator**

Create an application that lets the user enter a number of seconds and works as follows:
- There are 60 seconds in a minute. If the number of seconds entered by the user is greater than or equal to 60, the program should display the number of minutes in that many seconds.
- There are 3,600 seconds in an hour. If the number of seconds entered by the user is greater than or equal to 3,600, the program should display the number of hours in that many seconds.
- There are 86,400 seconds in a day. If the number of seconds entered by the user is greater than or equal to 86,400, the program should display the number of days in that many seconds.
12. **Workshop Selector**

The following table shows a training company’s workshops, the number of days of each, and their registration fees.

<table>
<thead>
<tr>
<th>Workshop</th>
<th>Number of Days</th>
<th>Registration Fee</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handling Stress</td>
<td>3</td>
<td>$1,000</td>
</tr>
<tr>
<td>Time Management</td>
<td>3</td>
<td>$800</td>
</tr>
<tr>
<td>Supervision Skills</td>
<td>3</td>
<td>$1,500</td>
</tr>
<tr>
<td>Negotiation</td>
<td>5</td>
<td>$1,300</td>
</tr>
<tr>
<td>How to Interview</td>
<td>1</td>
<td>$500</td>
</tr>
</tbody>
</table>

The training company conducts its workshops in the six locations shown in the following table. The table also shows the lodging fees per day at each location.

<table>
<thead>
<tr>
<th>Location</th>
<th>Lodging Fees per Day</th>
</tr>
</thead>
<tbody>
<tr>
<td>Austin</td>
<td>$150</td>
</tr>
<tr>
<td>Chicago</td>
<td>$225</td>
</tr>
<tr>
<td>Dallas</td>
<td>$175</td>
</tr>
<tr>
<td>Orlando</td>
<td>$300</td>
</tr>
<tr>
<td>Phoenix</td>
<td>$175</td>
</tr>
<tr>
<td>Raleigh</td>
<td>$150</td>
</tr>
</tbody>
</table>

When a customer registers for a workshop, he or she must pay the registration fee plus the lodging fees for the selected location. For example, here are the charges to attend the Supervision Skills workshop in Orlando:

**Registration:** $1,500  
**Lodging:** $300 × 3 days = $900  
**Total:** $2,400

Create an application that lets the user select a workshop from one ListBox and a location from another ListBox. When the user clicks a button, the application should calculate and display the registration cost, the lodging cost, and the total cost.
This page intentionally left blank
More about ListBoxes

CONCEPT: ListBox controls have various methods and properties that you can use in code to manipulate the ListBox’s contents.

In Chapter 4 we introduced the ListBox control, which displays a list of items and allows the user to select one or more items from the list. In this chapter we use ListBox controls to display output. Many of the algorithms that you will see in this chapter generate lists of data and then display those lists in ListBox controls.

Recall from Chapter 4 that you add items to a ListBox control’s Items property, and those items are displayed in the ListBox. At design time, you can use the Properties window to add items to the control’s Items property. You can also write code that adds items to a ListBox control at run time. To add an item to a ListBox control with code, you call the control’s Items.Add method. Here is the method’s general format:

```
ListBoxName.Items.Add(Item);
```

ListBoxName is the name of the ListBox control. Item is the value to be added to the Items property. For example, in the Chap05 folder of this book’s Student Sample Programs, you will find a project named Name List. Figure 5-1 shows the application’s form. As shown in the image on the left, the ListBox’s name is nameListBox and the Button control’s name is addButton. At run time, when you click the addButton control, the names shown in the image on the right are added to the nameListBox control.
Here is the code for the addButton_Click event handler:

```csharp
1 private void addButton_Click(object sender, EventArgs e)
2 {
3     namesListBox.Items.Add("Chris");
4     namesListBox.Items.Add("Alicia");
5     namesListBox.Items.Add("Justin");
6     namesListBox.Items.Add("Holly");
7 }
```

You can add values of other types as well. In the Chap05 folder of the book’s Student Sample Programs, you will find a project named Number List. Figure 5-2 shows the application’s form. As shown in the image on the left, the ListBox’s name is numberListBox and the Button control’s name is addButton. At run time, when you click the addButton control, the numbers shown in the image on the right are added to the numberListBox control.

Here is the code for the addButton_Click event handler:

```csharp
1 private void addButton_Click(object sender, EventArgs e)
2 {
3     numberListBox.Items.Add(10);
4     numberListBox.Items.Add(20);
5     numberListBox.Items.Add(30);
6     numberListBox.Items.Add(40);
7 }
```

**The Items.Count Property**

ListBox controls have an Items.Count property that reports the number of items stored in the ListBox. If the ListBox is empty, the Items.Count property equals 0. For example,
5.2 The while Loop

CONCEPT: The while loop causes a statement or set of statements to repeat as long as a Boolean expression is true.

The while loop gets its name from the way it works: While a Boolean expression is true, do some task. The loop has two parts: (1) a Boolean expression that is tested for a true or false value and (2) a statement or set of statements that is repeated as long as the Boolean expression is true. Figure 5-3 shows the logic of a while loop.

Figure 5-3 The logic of a while loop

assume an application has a ListBox control named employeesListBox. The following if statement displays a message box if there are no items in the ListBox:

```csharp
if (employeesListBox.Items.Count == 0)
{
    MessageBox.Show("There are no items in the list!");
}
```

The Items.Count property holds an integer value. Assuming numEmployees is an int variable, the following statement assigns the number of items in the employeesListBox to the numEmployees variable:

```csharp
numEmployees = employeesListBox.Items.Count;
```

The Items.Clear Method

ListBox controls have an Items.Clear method that erases all the items in the Items property. Here is the method’s general format:

```csharp
ListBoxName.Items.Clear();
```

For example, assume an application has a ListBox control named employeesListBox. The following statement clears all the items in the list.

```csharp
employeesListBox.Items.Clear();
```

Checkpoint

5.1 In code, how do you add an item to a ListBox control?
5.2 How do you determine the number of items that are stored in a ListBox control?
5.3 How do you erase the contents of a ListBox control?
The diamond symbol represents the Boolean expression that is tested. Notice what happens if the expression is true: One or more statements are executed and the program’s execution flows back to the point just above the diamond symbol. The Boolean expression is tested again, and if it is true, the process repeats. If the Boolean expression is false, the program exits the loop. Each time the loop executes its statement or statements, we say the loop is iterating, or performing an iteration.

Here is the general format of the while loop:

```java
while (BooleanExpression)
{
    statement;
    statement;
    etc.
}
```

We refer to the first line as the while clause. The while clause begins with the word while, followed by a Boolean expression that is enclosed in parentheses. Beginning on the next line is a set of statements enclosed in curly braces. This block of statements is known as the body of the loop.

When the while loop executes, the Boolean expression is tested. If the Boolean expression is true, the statements that appear in the body of the loop are executed, and then the loop starts over. If the Boolean expression is false, the loop ends and the program resumes execution at the statement immediately following the loop.

We say that the statements in the body of the loop are conditionally executed because they are executed only under the condition that the Boolean expression is true. If you are writing a while loop that has only one statement in its body, you do not have to enclose the statement inside curly braces. Such a loop can be written in the following general format:

```java
while (BooleanExpression)
    statement;
```

When a while loop written in this format executes, the Boolean expression is tested. If it is true, the one statement that appears on the next line is executed, and then the loop starts over. If the Boolean expression is false, however, the loop ends.

Although the curly braces are not required when there is only one statement in the loop’s body, it is still a good idea to use them, as shown in the following general format:

```java
while (BooleanExpression)
{
    statement;
}
```

When we discussed the various if statements in Chapter 4, we mentioned that this is a good style of programming because it cuts down on errors. If you have more than one statement in the body of a loop, those statements must be enclosed in curly braces. If you get into the habit of always enclosing the conditionally executed statements in a set of curly braces, it’s less likely that you will forget them.

You should also notice that the statements in the body of the loop are indented. As with if statements, this indentation makes the code easier to read and debug. By indenting the statements in the body of the loop, you visually set them apart from the surrounding code.

Let’s look at an example. In the Chap05 folder of this book’s Student Sample Programs, you will find a project named while Loop Demo. Figure 5-4 shows the application’s form. As shown in the image on the left, the Button control’s name is goButton. At run time, when you click the goButton control, the message box shown in the image on the right
The while Loop

5.2 The while Loop

is displayed. When you click the OK button to close the message box, another identical message box is displayed. The message box is displayed a total of five times.

Here is the code for the goButton_Click event handler:

```csharp
private void goButton_Click(object sender, EventArgs e)
{
    // Declare a variable to count the loop iterations.
    int count = 1;

    // Display “Hello” in a message box five times.
    while (count <= 5)
    {
        // Display the message box.
        MessageBox.Show("Hello");

        // Add one to count.
        count = count + 1;
    }
}
```

Let’s take a closer look at this code. In line 4 an int variable named count is declared and initialized with the value 1. A while loop begins in line 7. Notice that the while loop tests the expression count <= 5. The statements in the body of the while loop repeat as long as the count variable is less than or equal to 5. Inside the body of the loop, line 10 displays “Hello” in a message box, and then line 13 adds one to the count variable. This is the last statement in the body of the loop, so after it executes, the loop starts over. It tests the Boolean expression again, and if it is true, the statements in the body of the loop are executed. This cycle repeats until the Boolean expression count <= 5 is false, as illustrated in Figure 5-5. A flowchart for the loop is shown in Figure 5-6.
Chapter 5 Loops, Files, and Random Numbers

The while Loop Is a Pretest Loop

The while loop is known as a pretest loop, which means it tests its condition before performing an iteration. Because the test is done at the beginning of the loop, you usually have to perform some steps prior to the loop to make sure that the loop executes at least once. Notice the declaration of the count variable in the while Loop Demo program:

```csharp
int count = 1;
```

The count variable is initialized with the value 1. If count had been initialized with a value that is greater than 5, as shown in the following program sample, the loop would never execute:

```csharp
private void goButton_Click(object sender, EventArgs e)
{
    // Declare a variable to count the loop iterations.
    int count = 6;

    // This loop will never iterate!
    while (count <= 5)
    {
        // Display the message box.
        MessageBox.Show("Hello");

        // Add one to count.
        count = count + 1;
    }
}
```

An important characteristic of the while loop is that the loop will never iterate if the Boolean expression is false to start with. If you want to be sure that a while loop executes the first time, you must initialize the relevant data in such a way that the Boolean expression starts out as true.

Counter Variables

In the while Loop Demo application, the variable count is initialized with the value 1, and then 1 is added to the variable count during each loop iteration. The loop executes as long as count is less than or equal to 5. The variable count is used as a counter variable, which means it is regularly incremented in each loop iteration. In essence, the count variable keeps count of the number of iterations the loop has performed. Counter variables are commonly used to control the number of times that a loop iterates.

Tutorial 5-1 will give you some practice writing a loop and using a counter variable. In the tutorial you will write a while loop that calculates the amount of interest earned by a bank account each month for a number of months.
5.2 The while Loop

Tutorial 5-1:
Using a Loop to Calculate an Account Balance

In this tutorial you complete the Ending Balance application. The project has already been started for you and is located in the Chap05 folder of this book’s Student Sample Programs. The application’s form is shown in Figure 5-7.

Figure 5-7 The Ending Balance form

When you complete the application, it will allow the user to enter an account’s starting balance into the startingBalTextBox control and the number of months that the account will be left to earn interest into the monthsTextBox control. When the user clicks the calculateButton control, the application calculates the account’s balance at the end of the time period. The account’s monthly interest rate is 0.005, and the interest is compounded monthly.

Step 1: Start Visual Studio (or Visual Studio Express). Open the project named Ending Balance in the Chap05 folder of this book’s Student Sample Programs.

Step 2: Open the Form1 form in the Designer. The form is shown, along with the names of the important controls, in Figure 5-7.

Step 3: Now you will create the Click event handlers for the Button controls. At the end of this tutorial, Program 5-1 shows the completed code for the form. You will be instructed to refer to Program 5-1 as you write the event handlers.

In the Designer, double-click the calculateButton control. This opens the code editor, and you will see an empty event handler named calculateButton_Click. Complete the calculateButton_Click event handler by typing the code shown in lines 22–59 in Program 5-1. Let's take a closer look at the code:

Line 23: This statement declares a constant decimal named INTEREST_RATE, set to the value 0.005. This is the monthly interest rate.

Lines 26–28: These statements declare the following variables:
• balance, a decimal variable to hold the account balance.
• months, an int variable to hold the number of months that the account will be left to earn interest.
• count, an int that is used to count the months as a loop iterates. Notice that the count variable is initialized with the value 1.
Line 31: This if statement tries to convert startingBalTextBox.Text to a decimal. If the conversion is successful, the result is stored in the balance variable, and the program continues executing at line 33. If the conversion is not successful, the program jumps to the else clause in line 55, and line 58 displays the error message Invalid value for starting balance.

Line 34: This if statement tries to convert monthsTextBox.Text to an int. If the conversion is successful, the result is stored in the months variable, and the program continues executing at line 36. If the conversion is not successful, the program jumps to the else clause in line 49, and line 52 displays the error message Invalid value for months.

Line 37: This is the beginning of a while loop. The loop executes as long as the expression count <= months is true.

Lines 39–43: These statements are the body of the loop. Line 40 calculates the monthly interest and adds it to the balance variable. Line 43 adds 1 to the count variable.

Line 47: This statement executes after the loop has finished all of its iterations. It converts the value of the balance variable to a string (formatted as currency) and assigns the resulting string to the endingBalanceLabel control’s Text property.

Step 4: Switch your view back to the Designer and double-click the clearButton control. In the code editor you will see an empty event handler named clearButton_Click. Complete the clearButton_Click event handler by typing the code shown in lines 64–70 in Program 5-1.

Step 5: Switch your view back to the Designer and double-click the exitButton control. In the code editor you will see an empty event handler named exitButton_Click. Complete the exitButton_Click event handler by typing the code shown in lines 75–76 in Program 5-1.

Step 6: Save the project. Then, press [F5] on the keyboard, or click the Start Debugging button ( ) on the toolbar to compile and run the application.

First, enter 1000 as the starting balance and 48 as the number of months. Click the Calculate Average button and $1,270.49 should appear as the ending balance. Think about the value that you entered for the number of months. How many times did the while loop in line 36 iterate? (Answer: 48 times.)

Next, click the Clear button to clear the TextBoxes and the ending balance. Now, enter 100 as the starting balance and 1 as the number of months. Click the Calculate Average button and $100.50 should appear as the ending balance. How many times did the while loop iterate this time? (Answer: 1 time.)

Continue to test the application as you wish. When you are finished, click the Exit button and the form should close. (If you plan to continue to the next tutorial, leave this project open in Visual Studio.)

Program 5.1 Completed Form1 code for the Ending Balance application

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
```
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Ending_Balance
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void calculateButton_Click(object sender, EventArgs e)
        {
            // Constant for the monthly interest rate.
            const decimal INTEREST_RATE = 0.005m;

            // Local variables
            decimal balance;    // The account balance
            int months;        // The number of months
            int count = 1;      // Loop counter, initialized with 1

            // Get the starting balance.
            if (decimal.TryParse(startingBalTextBox.Text, out balance))
            {
                // Get the number of months.
                if (int.TryParse(monthsTextBox.Text, out months))
                {
                    // The following loop calculates the ending balance.
                    while (count <= months)
                    {
                        // Add this month's interest to the balance.
                        balance = balance + (INTEREST_RATE * balance);
                        // Add one to the loop counter.
                        count = count + 1;
                    }

                    // Display the ending balance.
                    endingBalanceLabel.Text = balance.ToString("c");
                }
                else
                {
                    // Invalid number of months was entered.
                    MessageBox.Show("Invalid value for months.");
                }
            }
            else
            {
                // Invalid starting balance was entered.
                MessageBox.Show("Invalid value for starting balance.");
            }

            private void clearButton_Click(object sender, EventArgs e)
            {
                // Clear the TextBoxes and the endingBalanceLabel control.
startingBalTextBox.Text = "";
monthsTextBox.Text = "";
endingBalanceLabel.Text = "";

// Reset the focus.
startingBalTextBox.Focus();

private void exitButton_Click(object sender, EventArgs e)
{
    // Close the form.
    this.Close();
}

Tutorial 5-2:
Enhancing the Ending Balance Application

In this tutorial you enhance the Ending Balance application that you created in Tutorial 5-1. First, add a ListBox control to the application’s form, as shown in Figure 5-8. Then modify the calculateButton_Click event handler so it displays each month’s ending balance in the ListBox. Figure 5-9 shows an example of how the form will appear when the user has entered 1000 for the starting balance and 8 for the months.

Figure 5-8 The modified Ending Balance form

Step 1: Start Visual Studio (or Visual Studio Express) and open the Ending Balance project that you completed in Tutorial 5-1.

Step 2: Enlarge the form so it is roughly the size shown in Figure 5-8. (310 pixels wide by 325 pixels high should be sufficient.)

Step 3: Create a ListBox control named detailListBox. Resize the ListBox as shown in Figure 5-8.
5.2 The while Loop

Step 4: Switch to the code editor and insert the code shown in lines 42–45 in Program 5.2. (The new lines of code are shown in bold.) The statement in lines 43–45 adds a string to the detailListBox control. If you examine the statement carefully, you will see that it uses concatenation to create a string in the following format:

The ending balance for month count is balance.

In the actual string that is created, count will be the value of the count variable and balance will be the value of the balance variable, formatted as currency.

Step 5: Find the clearButton_Click event handler in the code editor. Update the comment as shown in lines 69–70, and insert the line of code shown in line 74. (The lines are shown in bold.) The statement in line 74 clears the contents of the detailListBox control.

Step 6: Save the project. Then, press F5 on the keyboard or click the Start Debugging button ( ) on the toolbar to compile and run the application.

As shown in Figure 5-9, enter 1000 as the starting balance and 8 as the number of months. Click the Calculate Average button. Your output should look like that shown in Figure 5-9. Click the Clear button and enter any other values you wish to test the application further. When you are finished, click the Exit button and the form should close.

Program 5.2 Completed Form1 code for the Ending Balance application

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Ending_Balance
{
    // Code for the Ending Balance application
} 
```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }

    private void calculateButton_Click(object sender, EventArgs e)
    {
        // Constant for the monthly interest rate.
        const decimal INTEREST_RATE = 0.005m;

        // Local variables
        decimal balance; // The account balance
        int months; // The number of months
        int count = 1; // Loop counter, initialized with 1

        // Get the starting balance.
        if (decimal.TryParse(startingBalTextBox.Text, out balance))
        {
            // Get the number of months.
            if (int.TryParse(monthsTextBox.Text, out months))
            {
                // The following loop calculates the ending balance.
                while (count <= months)
                {
                    // Add this month’s interest to the balance.
                    balance = balance + (INTEREST_RATE * balance);

                    // Display this month’s ending balance.
                    detailListBox.Items.Add("The ending balance " +
                        "for month " + count + " is " +
                        balance.ToString("c"));

                    // Add one to the loop counter.
                    count = count + 1;
                }

                // Display the ending balance.
                endingBalanceLabel.Text = balance.ToString("c");
            }
            else
            {
                // Invalid number of months was entered.
                MessageBox.Show("Invalid value for months.");
            }
        }
        else
        {
            // Invalid starting balance was entered.
            MessageBox.Show("Invalid value for starting balance.");
        }

    }

    private void clearButton_Click(object sender, EventArgs e)
    {
        // Clear the TextBoxes, the endingBalanceLabel control,
        // and the ListBox.
    }
In all but rare cases, loops must contain a way to terminate within themselves. This means that something inside the loop must eventually make the loop’s Boolean expression false. The loop in Program 5-2 stops when the expression `count <= months` is false. If a loop does not have a way of stopping, it is called an infinite loop. An infinite loop continues to repeat until the program is interrupted. Infinite loops usually occur when the programmer forgets to write code inside the loop that makes the test condition false. In most circumstances you should avoid writing infinite loops.

The following code sample demonstrates an infinite loop. In line 1 the `count` variable is declared and initialized with the value 1. The `while` loop that begins in line 5 executes as long as `count <= 5`. There is no code inside the loop to change the `count` variable’s value, so the expression `count <= 5` in line 5 is always true. As a consequence, the loop has no way of stopping.

```csharp
1 // Declare a variable to count the loop iterations.
2 int count = 1;
3
4 // How many times will this loop iterate?
5 while (count <= 5)
6 {
7    // Display the message box.
8    MessageBox.Show("Hello");
9 }
```

**Checkpoint**

5.4 What is a loop iteration?

5.5 What is a counter variable?

5.6 What is a pretest loop?

5.7 Does the `while` loop test its condition before or after it performs an iteration?

5.8 What is an infinite loop?
The ++ and -- operators

CONCEPT: To increment a variable means to increase its value, and to decrement a variable means to decrease its value. C# provides special operators to increment and decrement variables.

To **increment** a variable means to increase its value and to **decrement** a variable means to decrease its value. Both of the following statements increment the variable `num` by 1:

```csharp
cnum = num + 1;
cnum += 1;
```

And `num` is decremented by 1 in both the following statements:

```csharp
cnum = num - 1;
cnum -= 1;
```

Incrementing and decrementing is so commonly done in programs that C# provides a set of simple unary operators designed just for incrementing and decrementing variables. The increment operator is `++`, and the decrement operator is `--`. The following statement uses the `++` operator to add 1 to `num`:

```csharp
cnum++;
```

After this statement executes, the value of `num` is increased by 1. The following statement uses the `--` operator to subtract 1 from `num`:

```csharp
cnum--;
```

**NOTE:** The `++` operator is pronounced “plus plus,” and the `--` operator is pronounced “minus minus.” The expression `num++` is pronounced “num plus plus,” and the expression `num--` is pronounced “num minus minus.”

In these examples, we have written the `++` and `--` operators after their operands (or, on the right side of their operands). This is called **postfix mode**. The operators can also be written before (or, on the left side) of their operands, which is called **prefix mode**. Here are examples:

```csharp
++num;
--num;
```

When you write a simple statement to increment or decrement a variable, such as the ones shown here, it doesn’t matter if you use prefix mode or postfix mode. The operators do the same thing in either mode. However, if you write statements that mix these operators with other operators or with other operations, there is a difference in the way the two modes work. Such complex code can be difficult to understand and debug. When we use the increment and decrement operators, we will do so only in ways that are straightforward and easy to understand, such as the statements previously shown.

We introduce these operators at this point because they are commonly used in loops. The following code segment shows an example. In the code, the `count` variable is initialized with the value 1. The `while` loop that begins in line 5 iterates as long as `count` is less than or equal to 5. The statement in line 11 increments `count`. The loop will iterate 5 times.

```csharp
1 // Declare a variable to count the loop iterations.
2 int count = 1;
```
5.4 The for Loop

CONCEPT: The for loop is ideal for performing a known number of iterations.

The for loop is specifically designed for situations requiring a counter variable to control the number of times that a loop iterates. When you write a for loop, you specify three actions:

- **Initialization**: This action takes place when the loop begins. It happens only once.
- **Test**: A Boolean expression is tested. If the expression is true, the loop iterates. Otherwise, the loop stops.
- **Update**: This action takes place at the end of each loop iteration.

Figure 5-10 shows how these three actions are used in the logic of a for loop.

Here is the general format of the for loop:

```
for (InitializationExpression; TestExpression; UpdateExpression)
{
    statement;
    statement;
    etc.
}
```

The statements that appear inside the curly braces are the body of the loop. These are the statements that are executed each time the loop iterates. As with other control structures,
the curly braces are optional if the body of the loop contains only one statement, as shown in the following general format:

\[
\text{for (InitializationExpression; TestExpression; UpdateExpression)}
\]

\[
\text{statement;}
\]

The first line of the \texttt{for} loop is the \textit{loop header}. After the key word \texttt{for}, there are three expressions inside the parentheses, separated by semicolons. (Notice that there is not a semicolon after the third expression.)

The first expression is the \textit{initialization expression}. It is normally used to initialize a counter variable to its starting value. This is the first action performed by the loop, and it is only done once. The second expression is the \textit{test expression}. This is a \textit{Boolean expression} that controls the execution of the loop. As long as this expression is true, the body of the \texttt{for} loop will repeat. The \texttt{for} loop is a pretest loop, so it evaluates the test expression before each iteration. The third expression is the \textit{update expression}. It executes at the end of each iteration. Typically, this is a statement that increments the loop’s counter variable.

Let’s assume that \texttt{count} is an \texttt{int} variable that has already been declared. Here is an example of a simple \texttt{for} loop that displays “Hello” in a message box 5 times:

```csharp
for (count = 1; count <= 5; count++)
{
    MessageBox.Show("Hello");
}
```

In this loop, the initialization expression is \texttt{count = 1}, the test expression is \texttt{count <= 5}, and the increment expression is \texttt{count++}. The body of the loop has one statement, which
is the call to MessageBox.Show method. This is a summary of what happens when this loop executes:

1. The initialization expression count = 1 is executed. This assigns 1 to the count variable.
2. The expression count <= 5 is tested. If the expression is true, continue with Step 3. Otherwise, the loop is finished.
3. The statement MessageBox.Show("Hello"); is executed.
4. The update expression count++ is executed. This adds 1 to the count variable.
5. Go back to Step 2.

Figure 5-11 illustrates this sequence of events. Notice that Steps 2–4 are repeated as long as the test expression is true. Figure 5-12 shows the logic of the loop as a flowchart.

Let’s look at a complete application that uses a for loop. In the Chap05 folder of this book’s Student Sample Programs, you will find a project named Squares. The purpose of the application is to display the numbers 1–10 and their squares. Figure 5-13 shows the application’s form. As shown in the image on the left, the ListBox’s name is outputListBox and the Button control’s name is goButton. At run time, when you click the goButton control, the outputListBox control displays the program’s output, as shown in the image on the right.
Chapter 5 Loops, Files, and Random Numbers

Here is the code for the `goButton_Click` event handler:

```csharp
private void goButton_Click(object sender, EventArgs e)
{
    // Constant for the maximum number
    const int MAX_VALUE = 10;

    // Loop counter
    int number;

    // Display the list of numbers and their squares.
    for (number = 1; number <= MAX_VALUE; number++)
    {
        outputListBox.Items.Add("The square of " +
                               number + " is " + (number * number));
    }
}
```

Let's take a closer look at the code:

- Line 4 declares an `int` constant named `MAX_VALUE`, set to the value 10. This is the maximum number that we will use to calculate a square.
- Line 7 declares an `int` variable named `number`. This variable is used both as a counter variable and in the calculation of squares.
- Line 10 is the beginning of a `for` loop. You can see from this line that the loop works in the following way:
  - **Initialization:** The `number` variable is initialized with the value 1.
  - **Test:** The expression `number <= MAX_VALUE` is tested at the beginning of each iteration.
  - **Update:** The expression `number++` is executed at the end of each iteration.

  Since the `MAX_VALUE` constant is set to the value 10, the `number` variable will be assigned the values 1 through 10 as the loop iterates.
- Lines 12 and 13: This statement adds a line to the ListBox showing the current value of the `number` variable, and the square of that value.

**The `for` Loop Is a Pretest Loop**

Because the `for` loop tests its Boolean expression before it performs an iteration, it is a pretest loop. It is possible to write a `for` loop in such a way that it will never iterate. Here is an example:

```csharp
for (count = 6; count <= 5; count++)
{
    MessageBox.Show("Hello");
}
```
Because the variable count is initialized to a value that makes the Boolean expression false from the beginning, this loop terminates as soon as it begins.

**Declaring the Counter Variable in the Initialization Expression**

Not only may the counter variable be initialized in the initialization expression, but it may also be declared there. The following code shows an example:

```csharp
for (int count = 1; count <= 5; count++)
{
    MessageBox.Show("Hello");
}
```

In this loop, the count variable is both declared and initialized in the initialization expression. If the variable is used only in the loop, it makes sense to define it in the loop header. This makes the variable’s purpose clearer.

When a variable is declared in the initialization expression of a for loop, the scope of the variable is limited to the loop. This means you cannot access the variable in statements outside the loop. For example, the following code would cause a compiler error because the statement in line 6 cannot access the count variable.

```csharp
1 for (int count = 1; count <= 5; count++)
2 {
3     MessageBox.Show("Hello");
4 }
5
6 MessageBox.Show("The value of count is " + count);
```

**Other Forms of the Update Expression**

In the update expression, the counter variable is typically incremented by 1. This makes it convenient to use the ++ operator in the increment expression. This is not a requirement, however. You can write virtually any expression you wish as the update expression. For example, the following loop increments count by 10.

```csharp
for (int count = 0; count <= 100; count += 10)
{
    MessageBox.Show(count.ToString());
}
```

Notice that in this example the increment expression is count += 10. This means that at the end of each iteration, 10 is added to count. During the first iteration count is set to 0, during the second iteration count is set to 10, during the third iteration count is set to 20, and so forth.

**Counting Backward by Decrementing the Counter Variable**

Although the counter variable is usually incremented in a count-controlled loop, you can alternatively decrement the counter variable. For example, look at the following code:

```csharp
for (int count = 10; count >= 0; count--)
{
    MessageBox.Show(count.ToString());
}
MessageBox.Show("Blastoff!");
```

In this loop the count variable is initialized with the value 10. The loop iterates as long as count is greater than or equal to 0. At the end of each iteration, count is decremented by 1.
During the first iteration count is 10, during the second iteration count is 9, and so forth. If this were in an actual program, it would display the numbers 10, 9, 8, and so forth, down to 0, and then display Blastoff!

**Avoiding Modifying the Counter Variable in the Body of the for Loop**

Be careful not to place a statement that modifies the counter variable in the body of the for loop. All modifications of the control variable should take place in the update expression, which is automatically executed at the end of each iteration. If a statement in the body of the loop also modifies the counter variable, the loop probably will not terminate when you expect it to. The following loop, for example, increments count twice for each iteration:

```csharp
for (int count = 1; count <= 10; count++)
{
    MessageBox.Show(count.ToString());
    count++; // Wrong!
}
```

You have seen several examples of the for loop. Tutorial 5-3 gives you an opportunity to write one. In the tutorial you will complete an application that uses a for loop to convert a series of measurements from the metric system to the English system.

**Tutorial 5-3:**

**Using the for Loop**

Your friend Amanda just inherited a European sports car from her uncle. Amanda lives in the United States, and she is afraid she will get a speeding ticket because the car’s speedometer works in kilometers per hour. She has asked you to write a program that displays a table of speeds in kilometers per hour with their values converted to miles per hour. The formula for converting kilometers per hour to miles per hour is:

\[
MPH = KPH \times 0.6214
\]

In the formula, MPH is the speed in miles per hour and KPH is the speed in kilometers per hour.

The table that your program displays should show speeds from 60 kilometers per hour through 130 kilometers per hour, in increments of 10, along with their values converted to miles per hour. The table should look something like this:

<table>
<thead>
<tr>
<th>KPH</th>
<th>MPH</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>37.284</td>
</tr>
<tr>
<td>70</td>
<td>43.498</td>
</tr>
<tr>
<td>80</td>
<td>49.712</td>
</tr>
<tr>
<td>etc . . .</td>
<td></td>
</tr>
</tbody>
</table>

After thinking about this table of values, you decide that you will write a for loop that uses a counter variable to hold the kilometers-per-hour speeds. The counter’s starting value will be 60, its ending value will be 130, and you will increase its value by 10 in the update expression. Inside the loop you will use the counter variable to calculate a speed in miles per hour.
The project, which is named *Speed Converter*, has already been started for you. It is located in the *Chap05* folder of this book’s Student Sample Programs. The application’s form is shown in Figure 5-14. The image on the left in the figure shows the names of the controls. The image on the right shows how the form appears after the user clicks the *Display Speeds* button.

**Figure 5-14** The *Speed Converter* form

---

**Step 1:** Start Visual Studio (or Visual Studio Express). Open the project named *Speed Converter* in the *Chap05* folder of this book’s Student Sample Programs.

**Step 2:** Open the Form1 form in the *Designer*.

**Step 3:** In the *Designer*, double-click the *displayButton* control. This opens the code editor, and you will see an empty event handler named *displayButton_Click*. Complete the *displayButton_Click* event handler by typing the code shown in lines 22–41 in Program 5-3 (at the end of this tutorial). Let’s take a closer look at the code:

**Lines 23–26:** These statements declare the following named constants:

- **START_SPEED**, an *int* constant set to 60. This is the starting speed for the list of conversions and the value with which the loop’s counter variable is initialized.
- **END_SPEED**, an *int* constant set to 130. This is the ending speed for the list of conversions. When the counter variable exceeds this value, the loop stops.
- **INTERVAL**, an *int* constant set to 10. This is the amount that you add to loop’s counter variable after each iteration.
- **CONVERSION_FACTOR**, a *double* constant set to 0.6214. This is the conversion factor that you use in the formula to convert KPH to MPH.

**Lines 29–30:** These statements declare the following variables:

- **kph**, an *int* variable to hold the speed in kilometers per hour.
- **mph**, a *double* variable to hold the speed in miles per hour.

**Line 33:** This is the beginning of a *for* loop that works in the following way:

**Initialization:** The *kph* variable is initialized with the value of **START_SPEED**, which is 60.

**Test:** The expression *kph <= END_SPEED* is tested at the beginning of each iteration.
Update: The expression `kph += INTERVAL` is executed at the end of each iteration. This adds the value of `INTERVAL` (which is 10) to the `kph` variable.

As the loop iterates, the `kph` variable is assigned the values 60, 70, 80, and so forth, through 130.

Line 36: This statement converts the value of the `kph` variable to miles per hour and assigns the result to the `mph` variable.

Lines 39–40: This statement adds a line to the `outputListBox` control showing the current value of the `kph` variable and the equivalent value in miles per hour.

**Step 4:** Switch your view back to the Designer and double-click the `exitButton` control. In the code editor you will see an empty event handler named `exitButton_Click`. Complete the `exitButton_Click` event handler by typing the code shown in lines 46–47 in Program 5-3.

**Step 5:** Save the project. Then, press [F5] on the keyboard or click the Start Debugging button ( ) on the toolbar to compile and run the application. Click the Display Speeds button, and you should see the output shown in the image on the right in Figure 5-14. Click the Exit button to close the form.

**Program 5-3** Completed Form1 code for the Speed Converter application

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Speed_Converter
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void displayButton_Click(object sender, EventArgs e)
        {
            // Constants
            const int START_SPEED = 60;
            const int END_SPEED = 130;
            const int INTERVAL = 10;
            const double CONVERSION_FACTOR = 0.6214;

            // Variables
            int kph;        // Kilometers per hour
            double mph;     // Miles per hour

            // Display the table of speeds.
            for (kph = START_SPEED; kph <= END_SPEED; kph += INTERVAL)
            {
```
5.11 Name the three expressions that appear inside the parentheses in the first line of a for loop.

5.12 You want to write a for loop that displays I love to program 50 times. Assume that you will use a variable named count as the counter variable.
   a. What initialization expression will you use?
   b. What test expression will you use?
   c. What update expression will you use?
   d. Write the loop.

5.13 What would the following code display?
   ```csharp
   for (int count = 1; count <= 5; count++)
   {
     MessageBox.Show(count.ToString());
   }
   ```

5.14 What would the following code display?
   ```csharp
   for (int count = 0; count <= 500; count += 100)
   {
     MessageBox.Show(count.ToString());
   }
   ```

### The do–while Loop

**CONCEPT:** The do–while loop is a posttest loop, which means it performs an iteration before testing its Boolean expression.

You have learned that the while loop and the for are pretest loops, which means they test their Boolean expressions before performing an iteration. The do–while loop is a posttest loop. This means it performs an iteration before testing its Boolean expression. As a result, the do–while loop always performs at least one iteration, even if its Boolean expression is false to begin with. The logic of a do–while loop is shown in Figure 5-15.
In the flowchart, one or more statements are executed, and then a Boolean expression is tested. If the Boolean expression is true, the program's execution flows back to the point just above the first statement in the body of the loop, and this process repeats. If the Boolean expression is false, the program exits the loop.

In code, the do-while loop looks something like an inverted while loop. Here is the general format of the do-while loop:

```
do
  { statement; statement; etc. }
while (BooleanExpression);
```

As with the while loop, the braces are optional if there is only one statement in the body of the loop. This is the general format of the do-while loop with only one conditionally executed statement:

```
do
  statement;
while (BooleanExpression);
```

Notice that a semicolon appears at the very end of the do-while statement. This semicolon is required; leaving it out is a common error.

The do-while loop is a posttest loop. This means it does not test its Boolean expression until it has completed an iteration. As a result, the do-while loop always performs at least one iteration, even if the expression is false to begin with. This differs from the behavior of a while loop. For example, in the following while loop, the statement that calls MessageBox.Show will not execute at all:

```
int number = 1;
while (number < 0)
{
  MessageBox.Show(number.ToString());
}
```

But, the statement that calls MessageBox.Show in the following do-while loop executes one time because the do-while loop does not test the expression number < 0 until the end of the iteration.

```
int number = 1;
while (number < 0)
{
  MessageBox.Show(number.ToString());
}
```
5.6 Using Files for Data Storage

CONCEPT: When a program needs to save data for later use, it writes the data in a file. The data can be read from the file at a later time.

The programs you have written so far require the user to reenter data each time the program runs because data kept in variables and control properties is stored in RAM and disappears once the program stops running. If a program is to retain data between the times it runs, it must have a way of saving it. Data is saved in a file, which is usually stored on a computer’s disk. Once the data is saved in a file, it will remain there after the program stops running. Data that is stored in a file can be retrieved and used at a later time.

Most of the commercial software that you use on a day-to-day basis store data in files. The following are a few examples.

- **Word processors**: Word processing programs are used to write letters, memos, reports, and other documents. The documents are then saved in files so they can be edited and printed.
- **Image editors**: Image-editing programs are used to draw graphics and edit images, such as the ones that you take with a digital camera. The images that you create or edit with an image editor are saved in files.
- **Spreadsheets**: Spreadsheet programs are used to work with numerical data. Numbers and mathematical formulas can be inserted into the rows and columns of the spreadsheet. The spreadsheet can then be saved in a file for use later.
- **Games**: Many computer games keep data stored in files. For example, some games keep a list of player names with their scores stored in a file. These games typically display the players’ names in order of their scores, from highest to lowest. Some games also allow you to save your current game status in a file so you can quit the game and then resume playing it later without having to start from the beginning.
- **Web browsers**: Sometimes when you visit a Web page, the browser stores a small file known as a cookie on your computer. Cookies typically contain information about the browsing session, such as the contents of a shopping cart.

Programs that are used in daily business operations rely extensively on files. Payroll programs keep employee data in files, inventory programs keep data about a company’s products in files, accounting systems keep data about a company’s financial operations in files, and so on.
Programmers usually refer to the process of saving data in a file as writing data to the file. When a piece of data is written to a file, it is copied from a variable in RAM to the file. This is illustrated in Figure 5-16. The term output file is used to describe a file to which data is written. It is called an output file because the program stores output in it.

**Figure 5-16 Writing data to a file**

![Diagram showing data copied from variables to a file]

The process of retrieving data from a file is known as reading data from the file. When a piece of data is read from a file, it is copied from the file into a variable in RAM. Figure 5-17 illustrates this. The term input file is used to describe a file from which data is read. It is called an input file because the program gets input from the file.

**Figure 5-17 Reading data from a file**

![Diagram showing data copied from a file to variables]

In this section we discuss ways to create programs that write data to files and read data from files. There are always three steps that must be taken when a file is used by a program.

1. **Open the file**—Opening a file creates a connection between the file and the program. Opening an output file usually creates the file on the disk and allows the program to write data to it. Opening an input file allows the program to read data from the file.
2. **Process the file**—In this step data is either written to the file (if it is an output file) or read from the file (if it is an input file).
3. **Close the file**—When the program is finished using the file, the file must be closed. Closing a file disconnects the file from the program.
5.6 Using Files for Data Storage

**Types of Files**

In general, there are two types of files: text and binary. A **text file** contains data that has been encoded as text using a scheme such as Unicode. Even if the file contains numbers, those numbers are stored in the file as a series of characters. As a result, the file may be opened and viewed in a text editor such as Notepad. A **binary file** contains data that has not been converted to text. As a consequence, you cannot view the contents of a binary file with a text editor. In this chapter we work only with text files.

**File Access Methods**

Most programming languages provide two different ways to access data stored in a file: sequential access and direct access. When you work with a **sequential access file**, you access data from the beginning of the file to the end of the file. If you want to read a piece of data that is stored at the very end of the file, you have to read all the data that comes before it—you cannot jump directly to the desired data. This is similar to the way cassette tape players work. If you want to listen to the last song on a cassette tape, you have to either fast-forward over all of the songs that come before it or listen to them. There is no way to jump directly to a specific song.

When you work with a **direct access file** (which is also known as a **random access file**), you can jump directly to any piece of data in the file without reading the data that comes before it. This is similar to the way a CD player or an MP3 player works. You can jump directly to any song you want to listen to.

This chapter focuses on sequential access files. Sequential access files are easy to work with, and you can use them to gain an understanding of basic file operations.

**Filenames and File Objects**

Files on a disk are identified by a **filename**. For example, when you create a document with a word processor and then save the document in a file, you have to specify a filename. When you use a utility such as Windows Explorer to examine the contents of your disk, you see a list of filenames. Figure 5-18 shows how three files named cat.jpg, notes.txt, and resume.doc might be represented in Windows Explorer.

![Figure 5-18 Three files](image)

Each operating system has its own rules for naming files. Many systems, including Windows, support the use of **filename extensions**, which are short sequences of characters that appear at the end of a filename and are preceded by a period (which is known as a “dot”). For example, the files depicted in Figure 5-18 have the extensions .jpg, .txt, and .doc. The extension usually indicates the type of data stored in the file. For example, the .jpg extension usually indicates that the file contains a graphic image that is compressed according to the JPEG image standard. The .txt extension usually indicates that the file contains text. The .doc extension usually indicates that the file contains a Microsoft Word document.

In order for a program to work with a file on the computer’s disk, the program must create a file object in memory. A **file object** is an object that is associated with a specific file and provides a way for the program to work with that file. In the program, a variable is linked with the file object. We say that the variable references the object. This variable is used to carry out any operations that are performed on the file. This concept is shown in Figure 5-19.
Chapter 5 Loops, Files, and Random Numbers

You will be using two classes from the .NET Framework to create file objects. When you want to write data to a text file, you use the `StreamWriter` class, and when you want to read data from a text file you use the `StreamReader` class. These classes are in the `System.IO` namespace in the .NET Framework, so you will need to write the following directive at the top of your programs:

```csharp
using System.IO;
```

**NOTE:** In programming terminology, files are considered streams of data. In C# you use a `StreamWriter` object to open a stream, such as a file, and write data to it. You use a `StreamReader` object to open a stream, such as a file, and read data from it.

### Writing Data to a File with a StreamWriter Object

You can use the `StreamWriter` class’s `WriteLine` method to write a line of text to a file. Let’s suppose you want to write a program that creates a text file named `courses.txt` and writes the names of the courses you are taking to the file. The following code sample shows how you can do this:

```csharp
StreamWriter outputFile;
outputFile = File.CreateText("courses.txt");
outputFile.WriteLine("Intro to Computer Science");
outputFile.WriteLine("English Composition");
outputFile.WriteLine("Calculus I");
outputFile.WriteLine("Music Appreciation");
outputFile.Close();
```

Let’s look at each line of code.

**Line 1:** This statement declares a variable named `outputFile`, which can be used to reference a `StreamWriter` object.

**Line 2:** In a nutshell, this statement opens the file to which you will be writing data. It does so by calling the `File.CreateText` method, passing the string “courses.txt” as an argument. The `File.CreateText` method does the following:
- It creates a text file with the name specified by the argument. If the file already exists, its contents are erased.
- It creates a `StreamWriter` object in memory, associated with the file.
- It returns a reference to the `StreamWriter` object.
Notice that an assignment operator assigns the value returned from the `File.CreateText` method to the `outputFile` variable. This causes the `outputFile` variable to reference the `StreamWriter` object that was created by the method.

After the statement in line 2 executes, the `courses.txt` file is created on the disk, a `StreamWriter` object associated with the file exists in memory, and the `outputFile` variable references that object.

**Line 4:** This statement writes the string "Intro to Computer Science" to the `courses.txt` file. It does that by calling the `StreamWriter` class's `WriteLine` method, passing the string that is to be written to the file as an argument. When the `WriteLine` method writes data to a file, it writes a newline character immediately following the data. A newline character is an invisible character that specifies the end of a line of text.

**Line 5:** This statement writes the string "English Composition" to the `courses.txt` file.

**Line 6:** This statement writes the string "Calculus I" to the `courses.txt` file.

**Line 7:** This statement writes the string "Music Appreciation" to the `courses.txt` file.

**Line 9:** This statement closes the `courses.txt` file. It does that by calling the `StreamWriter` class's `Close` method.

After this code has executed, we can open the `courses.txt` file using a text editor and look at its contents. Figure 5-20 show how the file's contents will appear in Notepad.

**Figure 5-20** Contents of the `courses.txt` file shown in Notepad

---

**Writing Data with the `Write` Method**

Earlier you read that the `StreamWriter` class's `WriteLine` method writes an item of data to a file and then writes a newline character. The newline character specifies the end of a line of text. For example, the following code sample opens a file named `Example.txt` and then uses the `WriteLine` method to write the strings "One", "Two", and "Three" to the file. Because a newline character is written after each string, the strings appear on separate lines when viewed in a text editor. The screen shown on the left in Figure 5-21 shows how the file would appear in Notepad.

```csharp
StreamWriter outputFile;
outputFile = File.CreateText("Example.txt");
outputFile.WriteLine("One");
outputFile.WriteLine("Two");
outputFile.WriteLine("Three");
outputFile.Close();
```
In some situations you might want to write an item to a file without a newline character immediately following it. The `StreamWriter` class provides the `Write` method for this purpose. It writes an item of data to a text file without writing a newline character. The following code sample demonstrates. The screen shown on the right in Figure 5-21 shows how the resulting file would appear in Notepad.

```csharp
StreamWriter outputFile;
outputFile = File.CreateText("Example.txt");
outputFile.Write("One");
outputFile.Write("Two");
outputFile.Write("Three");
outputFile.Close();
```

**Handling File-Related Exceptions**

Unexpected problems can potentially occur when working with files. For example, your program might not have sufficient rights to create a file when it calls the `File.CreateText` method, or the disk might be full when you call the `StreamWriter` class’s `WriteLine` method. When unexpected errors such as these occur, an exception is thrown. To handle such exceptions, you can write a try-catch statement, with the code that performs file operations placed in the try block. Here is an example:

```csharp
try
{
    StreamWriter outputFile;
    outputFile = File.CreateText("courses.txt");
    outputFile.WriteLine("Intro to Computer Science");
    outputFile.WriteLine("English Composition");
    outputFile.WriteLine("Calculus I");
    outputFile.WriteLine("Music Appreciation");
    outputFile.Close();
}
catch (Exception ex)
{
    // Display an error message.
    MessageBox.Show(ex.Message);
}
```

In Tutorial 5-4 you will complete an application that reads input from a TextBox control and writes the input to a file.
Tutorial 5-4: Writing Data to a Text File

In this tutorial you complete the *Friend File* application. The project has already been started for you and is located in the *Chap05* folder of this book’s Student Sample Programs. The application’s form is shown in Figure 5-22.

**Figure 5-22** The *Friend File* form

When you complete the application, it will allow the user to enter a name into the `nameTextBox` control. When the user clicks the `writeNameButton` control, the application opens a text file named Friend.txt, writes the name that was entered into the `TextBox` control to the file, and then closes the file.

**Step 1:** Start Visual Studio (or Visual Studio Express). Open the project named *Ending Balance* in the *Chap05* folder of this book’s Student Sample Programs.

**Step 2:** Open the Form1 form’s code in the code editor. Insert the `using System.IO;` directive shown in line 10 of Program 5-4 at the end of this tutorial. This statement is necessary because the `StreamWriter` class is part of the `System.IO` namespace in the .NET Framework.

**Step 3:** Open the Form1 form in the Designer. The form is shown, along with the names of the important controls, in Figure 5-22. Double-click the `writeNameButton` control. This opens the code editor, and you will see an empty event handler named `writeNameButton_Click`. Complete the `writeNameButton_Click` event handler by typing the code shown in lines 23–44 in Program 5-4. Let’s take a closer look at the code:

**Line 23:** This is the beginning of a `try-catch` statement. The `try` block appears in lines 25–38, and the `catch` block appears in lines 42–43. If an exception is thrown by any statement in the `try` block, the program jumps to the `catch` block, and line 43 displays an error message.

**Line 26:** This statement declares a `StreamWriter` variable named `outputFile`. You use this variable to reference a `StreamWriter` object.

**Line 29:** This statement calls the `File.CreateText` method to create a text file named Friend.txt. The method also creates a `StreamWriter` object in memory associated with the file. The method returns a reference to that object, which is assigned to the `outputFile` variable. As a result, the `outputFile` variable references the `StreamWriter` object. You will be able to use the `outputFile` variable to perform operations on the Friend.txt file.

**Line 32:** This statement uses the `outputFile` variable to call the `StreamWriter` class’s `WriteLine` method. The `nameTextBox` control’s `Text` property is passed as an argument. As a result, the value entered into the `TextBox` is written to the Friend.txt file.
Line 35: This statement closes the Friend.txt file.

Line 38: This statement displays a message box to let the user know that the name was written to the file.

**Step 4:** Switch your view back to the *Designer* and double-click the `exitButton` control. In the code editor you will see an empty event handler named `exitButton_Click`. Complete the `exitButton_Click` event handler by typing the code shown in lines 49–50 in Program 5-4.

**Step 5:** Save the project. Then, press **F5** on the keyboard or click the *Start Debugging* button ( ) on the toolbar to compile and run the application.

Enter a name into the `nameTextBox` control, and then click the *Write Name* button. You should see a message box appear letting you know that the name was written to the file. Click the *OK* button to dismiss the message box; then click the *Exit* button on the application’s form to end the application.

**Step 6:** Now you will look at the contents of the Friend.txt file that the application created. Perform one of the following, depending on whether you are using Visual Studio or Visual Studio Express:

- If you are using Visual Studio, click *FILE* on the menu bar, then click *Open*, and then click *File*.
- If you are using Visual Studio Express, click *FILE* on the menu bar, and then click *Open File*.

**Step 7:** You should now see the *Open File* window, viewing the contents of the *Friend File* project folder. As shown in Figure 5-23, open the *bin* folder, then open the

**Figure 5-23** Opening the Friend.txt file in the *Open File* window

1. Open the bin folder.
2. Open the Debug folder.
3. Select Friend.txt. (You might not see the .txt extension.)
Debug folder, and then select the file Friend.txt. (You might not see the .txt extension, depending on how your system is set up.) Click the Open button.

**Step 8:** You should now see the contents of the Friend.txt file in Visual Studio, as shown in Figure 5-24. (The example in the figure shows the contents of the file after the user has written Tim Owens to the file.) When you are finished examining the contents of the file, you can close its tab. We come back to this project in the next tutorial.

**Figure 5-24** Sample contents of the Friend.txt file shown in Visual Studio

---

**Program 5-4** Completed Form1 code for the Friend File application

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.IO;

namespace Friend_File
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void writeNameButton_Click(object sender, EventArgs e)
        {
            try
            {
                // Declare a StreamWriter variable.
                StreamWriter outputFile;

                // Create a file and get a StreamWriter object.
                outputFile = File.CreateText("Friend.txt");

                // Write the friend's name to the file.
                outputFile.WriteLine(nameTextBox.Text);

                // Close the file.
                outputFile.Close();
            }
            catch
            {
                MessageBox.Show("Could not write to file."");
            }
        }
    }
}
```
Writing Numeric Data to a Text File

You can use the StreamWriter class’s WriteLine or Write method to write numbers (such as ints, doubles, and decimals) to a text file, but the numbers are converted to strings. For example, look at the following sample code (taken from the Number File project in the Chap05 folder of the Student Sample Programs):

```csharp
private void writeNumbersButton_Click(object sender, EventArgs e)
{
    try
    {
        // Declare a StreamWriter variable.
        StreamWriter outputFile;

        // Create a file and get a StreamWriter object.
        outputFile = File.CreateText("Numbers.txt");

        // Write the numbers 1 through 10 to the file.
        for (int count = 1; count <= 10; count++)
        {
            outputFile.WriteLine(count);
        }

        // Close the file.
        outputFile.Close();
    }
    catch (Exception ex)
    {
        // Display an error message.
        MessageBox.Show(ex.Message);
    }
}
```

When this event handler executes, line 9 creates a text file named Numbers.txt, and the loop in lines 12–15 writes the numbers 1–10 to the file. Figure 5-25 shows how the file appears when opened with Notepad.
Appending Data to an Existing File

When you call the `File.CreateText` method to open a file and the file specified by the argument already exists, it is erased and a new empty file with the same name is created. For example, when you run the `Friend File` application that you completed in Tutorial 5-4, each time you click the `Write Name` button, the Friend.txt file is erased and a new file is created.

Sometimes you want to preserve an existing file and append new data to its current contents. To append data to an existing file, you open it with the `File.AppendText` method. It works like the `File.CreateText` method, but the file is not erased if it already exists. Any data written to the file is appended to the file’s existing contents.

For example, assume the file Names.txt exists and contains the data shown in Figure 5-26:

The following code opens the file and appends additional data to its existing contents:

```csharp
StreamWriter outputFile;
outputFile = File.AppendText("Names.txt");

outputFile.WriteLine("Lynn");
outputFile.WriteLine("Steve");
outputFile.WriteLine("Bill");

outputFile.Close();
```

After this code executes, the Names.txt file contains the data shown in Figure 5-27:
Tutorial 5-5:  
Appending Data to the Friend.txt File

In this tutorial you will modify the Friend File application so it appends data to the Friend. txt file. When the user clicks the Write Name button, instead of erasing the file’s current contents, the application adds the contents of the nameTextBox control to the Friend.txt file.

Step 1:  If the Friend File project from Tutorial 5-4 is not currently open in Visual Studio (or Visual Studio Express), open it now.

Step 2:  Open the Form1 form’s code in the code editor. You will make modifications to the writeNameButton_Click event handler. Program 5-5, at the end of this tutorial, shows how the event handler code will appear after you make the following changes:
- Lines 8 and 9: Change the comments as shown to reflect the way that the Friend. txt file will be opened in line 10.
- Lines 21–25: Add the new comments and statements shown in these lines. Line 22 clears the nameTextBox control’s contents, and line 25 gives the focus to the nameTextBox control. This makes the application more convenient for adding several names to the file.

Step 3:  Save the project. Then, press F5 on the keyboard or click the Start Debugging button ( ) on the toolbar to compile and run the application.

Enter a name into the nameTextBox control and then click the Write Name button. You should see a message box letting you know that the name was written to the file. When you click the OK button to dismiss the message box, notice that the nameTextBox is cleared, and the focus is given to the TextBox. Enter another name, and click the Write Name button. Again, you see the message box. Click the OK button to dismiss the message box and then click the Exit button to end the application.

Step 4:  Now you will look at the contents of the Friend.txt file.
- If you are using Visual Studio, click FILE on the menu bar, then click Open, and then click File.
- If you are using Visual Studio Express, click FILE on the menu bar and then click Open File.

You should now see the Open File window, viewing the contents of the Friend File project folder. Open the bin folder, then open the Debug folder, and then
select the file Friend.txt. (You might not see the .txt extension, depending on how your system is set up.) Click the Open button.

**Step 5:** You should now see the contents of the Friend.txt file in Visual Studio. Figure 5-28 shows an example. Notice that the names that you entered were appended to the file each time you clicked the Write Name button. When you are finished examining the contents of the file, you can close its tab. We come back to this project in the next tutorial.

**Figure 5-28** Sample contents of the Friend.txt file shown in Visual Studio

![Sample contents of the Friend.txt file shown in Visual Studio](image)

**Program 5-5** Partial code for Form1 in the Friend File application

```csharp
private void writeNameButton_Click(object sender, EventArgs e)
{
    try
    {
        // Declare a StreamWriter variable.
        StreamWriter outputFile;

        // Open the Friend.txt file for appending,
        // and get a StreamWriter object.
        outputFile = File.AppendText("Friend.txt");

        // Write the friend's name to the file.
        outputFile.WriteLine(nameTextBox.Text);

        // Close the file.
        outputFile.Close();

        // Let the user know the name was written.
        MessageBox.Show("The name was written.");

        // Clear the nameTextBox control.
        nameTextBox.Text = "";

        // Give the focus to the nameTextBox control.
        nameTextBox.Focus();
    }
    catch (Exception ex)
    {
        // Display an error message.
        MessageBox.Show(ex.Message);
    }
}
```
304

Chapter 5

Loops, Files, and Random Numbers

Specifying the Location of an Output File
When you call the File.CreateText or File.AppendText methods to open a file, you
pass the filename as a string argument. If the filename that you pass as an argument does
not contain the file’s path, the file’s location will be the bin\Debug folder, under the application’s project folder. You saw this in Tutorials 5-4 and 5-5 when you opened the
Friend.txt file in Visual Studio.
If you want to open a file in a different location, you can specify a path as well as a
filename in the argument that you pass to the File.CreateText or File.AppendText
method. If you specify a path in a string literal, be sure to prefix the string with the @
character. Here is an example:
StreamWriter outputFile;
outputFile = File.CreateText(@"C:\Users\Chris\Documents\Names.txt");

T IP: You can also let the user specify the file location. See Section 5.7 for more information on the SaveFileDialog control.

Reading Data from a File with a StreamReader Object
To read data from a text file, you create a StreamReader object. You can then use the
StreamReader class’s ReadLine method to read a line of text from a file. For example,
suppose a file named Students.txt exists and contains the four names shown in Figure 5-29.

Figure 5-29 Contents of the Students.txt file

Let’s suppose you want to write a program that reads the four names from the Students.
txt file and displays them in message boxes. The following code sample shows how you
can do this. (This code sample is taken from the Student Names project, in the Chap05
folder of this book’s Student Sample Programs.)
1 try
2 {
3
// Declare a variable to hold an item read from the file.
4
string studentName;
5
6
// Declare a StreamReader variable.
7
StreamReader inputFile;
8
9
// Open the file and get a StreamReader object.
10
inputFile = File.OpenText("Students.txt");
11


5.6 Using Files for Data Storage 305

// Read and display the first name.
studentName = inputFile.ReadLine();
MessageBox.Show(studentName);

// Read and display the second name.
studentName = inputFile.ReadLine();
MessageBox.Show(studentName);

// Read and display the third name.
studentName = inputFile.ReadLine();
MessageBox.Show(studentName);

// Read and display the fourth name.
studentName = inputFile.ReadLine();
MessageBox.Show(studentName);

// Close the file.
inputFile.Close();

} catch (Exception ex)
{
    // Display an error message.
    MessageBox.Show(ex.Message);
}

Let’s take a closer look at the code.

**Line 1:** This is the beginning of a try-catch statement. An exception will be thrown in the try block if a problem occurs while the file is being opened or while an item is being read from the file. If that happens, the program jumps to the catch clause in line 31.

**Line 4:** This statement declares a string variable named studentName. Each time we read a line of text from the file, we assign it to this variable.

**Line 7:** This statement declares a variable named inputFile that can be used to reference a StreamReader object.

**Line 10:** This statement opens the file from which we will be reading data. It does so by calling the File.OpenText method, passing the string "Students.txt" as an argument. The File.OpenText method does the following:
- It opens an existing text file with the name specified by the argument. If the file does not exist, an exception is thrown.
- It creates a StreamReader object in memory associated with the file.
- It returns a reference to the StreamReader object.

Notice that an assignment operator assigns the value returned from the File.OpenText method to the inputFile variable. This causes the inputFile variable to reference the StreamReader object that was created by the method.

**Line 13:** This statement calls the inputFile.ReadLine method, which reads a line of text from the file. The line of text is returned as a string from the method and assigned to the studentName variable. Since this statement reads the first line of text from the file, the studentName variable is assigned the string “Joe Merrell”.

**Line 14:** This statement displays the contents of the studentName variable in a message box.

**Lines 17 and 18:** The statement in line 17 reads the next line of text from the file and assigns it to the studentName variable. After this line executes, the studentName variable is assigned the string “Chris Rich”. The statement in line 18 displays the contents of the studentName variable in a message box.
Lines 21 and 22: The statement in line 21 reads the next line of text from the file and assigns it to the `studentName` variable. After this line executes, the `studentName` variable is assigned the string “Kathryn Stevens”. The statement in line 22 displays the contents of the `studentName` variable in a message box.

Lines 25 and 26: The statement in line 25 reads the next line of text from the file and assigns it to the `studentName` variable. After this line executes, the `studentName` variable is assigned the string “Carly Colombo”. The statement in line 26 displays the contents of the `studentName` variable in a message box.

Line 29: This statement closes the Students.txt file.

The Read Position

When a program works with an input file, a special value known as a read position is internally maintained for that file. A file’s read position marks the location of the next item that will be read from the file. When an input file is opened, its read position is initially set to the first item in the file. As items are read from the file, the read position moves forward, toward the end of the file. Let’s see how this works in the previous code sample (from the Student Names project) After the statement in line 10 executes, the read position for the Students.txt file is positioned as shown in Figure 5-30.

![Figure 5-30 The initial read position](image)

The `ReadLine` method call in line 13 reads an item from the file’s current read position and assigns that item to the `studentName` variable. Once this statement executes, the `studentName` variable is assigned the string “Joe Merrell”. In addition, the file’s read position is advanced to the next item in the file, as shown in Figure 5-31.

![Figure 5-31 Read position after the first ReadLine method call](image)

The `ReadLine` method call in line 17 reads an item from the file’s current read position and assigns that value to the `studentName` variable. Once this statement executes, the `studentName` variable is assigned the string “Chris Rich”. The file’s read position is advanced to the next item, as shown in Figure 5-32.

![Figure 5-32 Read position after the second ReadLine method call](image)
The `ReadLine` method call in line 21 reads an item from the file's current read position and assigns that value to the `studentName` variable. Once this statement executes, the `studentName` variable is assigned the string “Kathryn Stevens”. The file’s read position is advanced to the next item, as shown in Figure 5-33.

**Figure 5-33** Read position after the third `ReadLine` method call

<table>
<thead>
<tr>
<th>Joe Merrell newline</th>
<th>Chris Rich newline</th>
<th>Kathryn Stevens newline</th>
<th>Carly Colombo newline</th>
</tr>
</thead>
</table>

The last `ReadLine` method call appears in line 25. It reads an item from the file’s current read position and assigns that value to the `studentName` variable. Once this statement executes, the `studentName` variable is assigned the string “Carly Colombo”. The file’s read position is advanced to the end of the file, as shown in Figure 5-34.

**Figure 5-34** Read position after the fourth `ReadLine` method call

<table>
<thead>
<tr>
<th>Joe Merrell newline</th>
<th>Chris Rich newline</th>
<th>Kathryn Stevens newline</th>
<th>Carly Colombo newline</th>
</tr>
</thead>
</table>

**NOTE:** Did you notice that the previous code sample read the items in the Students.txt file in sequence, from the beginning of the file to the end of the file? Recall from our discussion at the beginning of the section that this is the nature of a sequential access file.

**Reading Numeric Data from a Text File**

Remember that when data is stored in a text file, it is encoded as text, using a scheme such as Unicode. Even if the file contains numbers, those numbers are stored in the file as a series of characters. Furthermore, when you read an item from a text file with the `StreamReader` class’s `ReadLine` method, that item is returned as a string.

Suppose a text file contains numeric data, such as that shown in Figure 5-35. When we use the `ReadLine` method to read the items from the file, we get the strings “10”, “20”, and “30”. If we need to perform math with these values, we must convert each value from a string to a numeric data type. We can use the `Parse` or `TryParse` families of methods that you already know about to perform this conversion.

**Figure 5-35** A text file containing numeric data
Let’s suppose you want to write a program that reads the three numbers from the NumericData.txt file shown in Figure 5-35 and displays their total in a message box. The following code sample shows a simple demonstration. (This code sample is taken from the Numeric Data project in the Chap05 folder of this book’s Student Sample Programs.)

```csharp
try
{
    // Variables to hold the numbers read from the file
    // and their total
    int number1, number2, number3, total;

    // A StreamReader variable.
    StreamReader inputFile;

    // Open the file and get a StreamReader object.
    inputFile = File.OpenText("NumericData.txt");

    // Read three numbers from the file.
    number1 = int.Parse(inputFile.ReadLine());
    number2 = int.Parse(inputFile.ReadLine());
    number3 = int.Parse(inputFile.ReadLine());

    // Calculate the total of the numbers.
    total = number1 + number2 + number3;

    // Display the total.
    MessageBox.Show("The total is "+ total);

    // Close the file.
    inputFile.Close();
}

catch (Exception ex)
{
    // Display an error message.
    MessageBox.Show(ex.Message);
}
```

Let’s take a closer look at the code.

**Line 1:** This is the beginning of a try-catch statement. Various exceptions can be thrown by the code in the try block, which cause the program to jump to the catch clause in line 27.

**Line 5:** This statement declares the int variables `number1`, `number2`, `number3`, and `total`. These variables hold the three values read from the file and their total.

**Lines 8–11:** After these statements have executed, the NumericData.txt file is opened for reading, and the `inputFile` variable references a StreamReader object that is associated with the file.

**Line 14:** This statement does the following:
- It calls the `inputFile.ReadLine` method to read a line of text from the file.
- The value that is returned from the `inputFile.ReadLine` method (a string) is passed as an argument to the `int.Parse` method.
- The value that is returned from the `int.Parse` method is assigned to the `number1` variable.

After this statement executes, the `number1` variable is assigned the first value read from the file, converted to an int. (The `number1` variable is assigned the value 10.)
Line 15: This statement reads the next value from the file, converts it to an int, and assigns the result to the number2 variable. (The number2 variable is assigned the value 20.)

Line 16: This statement reads the next value from the file, converts it to an int, and assigns the result to the number3 variable. (The number3 variable is assigned the value 30.)

Line 19: This statement calculates the sum of number1, number2, and number3, and assigns the result to total.

Line 22: This statement displays the sum of the numbers in a message box.

Line 25: This statement closes the file.

Reading a File with a Loop and Detecting the End of the File

Quite often a program must read the contents of a file without knowing the number of items that are stored in the file. For example, suppose you need to write a program that displays all the items in a file, but you do not know how many items the file contains. You can open the file and then use a loop to repeatedly read an item from the file and display it. However, an exception will be thrown if the program attempts to read beyond the end of the file. The program needs some way of knowing when the end of the file has been reached so it will not try to read beyond it. The following pseudocode shows the logic:

Open the file
While not at the end of the file:
   Read an item from the file
   Display the item
End While
Close the file

StreamReader objects have a Boolean property named EndOfStream that signals whether the end of the file has been reached. If the file’s read position is at the end of the file (and there is no more data to read), the EndOfStream property is set to true. Otherwise, it is set to false. When you need to read all the items in a file without knowing how many items the file contains, you can write a loop that iterates as long as the EndOfStream property is false.

Let’s assume inputFile references a StreamReader object that is associated with a file that is already open. You can write the loop in the following manner:

while (inputFile.EndOfStream == false)
{
   // Read an item from the file.
   // Process the item.
}

However, most programmers prefer the following logic, which uses the ! operator:

while (!inputFile.EndOfStream)
{
   // Read an item from the file.
   // Process the item.
}

Recall that the ! operator is the logical NOT operator. When you read the first line of this loop, you naturally think while NOT at the end of the stream. In Tutorial 5-6 you will complete an application that uses this technique to display all the items in a file.
Tutorial 5-6:  
Using a Loop to Read to the End of a File

In this tutorial you complete the South America application that is found in the Chap05 folder of this book’s Student Sample Programs. The application’s form has already been created and is shown in Figure 5-36. The application also has an accompanying text file named Countries.txt that is stored in the bin\Debug folder under the project folder. The Countries.txt file contains the names of the countries of South America. Figure 5-37 shows the file as it appears in Notepad.

**Figure 5-36** The South America application’s form

![South America application's form](image)

**Figure 5-37** The Countries.txt file

![Countries.txt file in Notepad](image)

When the completed application runs and the user clicks the Get Countries button, the application reads each country name from the file and adds each one to the countriesListBox control.

**Step 1:** Start Visual Studio (or Visual Studio Express). Open the project named South America in the Chap05 folder of this book’s Student Sample Programs.

**Step 2:** Open the Form1 form’s code in the code editor. Insert the using System.IO; directive shown in line 10 of Program 5-6 at the end of this tutorial. This statement is necessary because the StreamReader class is part of the System.IO namespace in the .NET Framework.
Step 3: Open the Form1 form in the Designer. The form is shown, along with the names of the important controls, in Figure 5-36. Double-click the getCountriesButton control. This opens the code editor, and you will see an empty event handler named getCountriesButton_Click. Complete the getCountriesButton_Click event handler by typing the code shown in lines 23–46 in Program 5-6. Let's take a closer look at the code:

Line 23: This is the beginning of a try-catch statement, which handles any exceptions that are thrown while the file is being processed. If an exception is thrown by any statement in the try block, the program jumps to the catch clause in line 50.

Line 26: This statement declares the string variable countryName, which holds the lines of text that are read from the file.

Lines 29–32: After these statements have executed, the Countries.txt file is opened for reading, and the inputFile variable references a StreamReader object that is associated with the file.

Line 35: This statement clears anything that might be displayed in the countriesListBox control. (This prevents the names of the countries from appearing multiple times in the ListBox if the user clicks the Get Countries button multiple times.)

Line 38: This is the beginning of a while loop that iterates as long as the end of the Countries.txt file has not been reached.

Line 41: This statement reads a line of text from the file and assigns it to the countryName variable.

Line 44: This statement adds the contents of the countryName variable to the ListBox.

Line 48: This statement closes the file.

Step 4: Switch your view back to the Designer and double-click the exitButton control. In the code editor you will see an empty event handler named exitButton_Click. Complete the exitButton_Click event handler by typing the code shown in lines 59–60 in Program 5-6.

Step 5: Save the project. Then, press F5 on the keyboard or click the Start Debugging button ( ) on the toolbar to compile and run the application. When the application runs, click the Get Countries button. This should fill the ListBox with the names of the countries from the Countries.txt file, as shown in Figure 5-38. Click the Exit button to exit the application.

Figure 5-38 The South America application displaying the list of countries
namespace South_America
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void getCountriesButton_Click(object sender, EventArgs e)
        {
            try
            {
                // Declare a variable to hold a country name.
                string countryName;

                // Declare a StreamReader variable.
                StreamReader inputFile;

                // Open the file and get a StreamReader object.
                inputFile = File.OpenText("Countries.txt");

                // Clear anything currently in the ListBox.
                countriesListBox.Items.Clear();

                // Read the file's contents.
                while (!inputFile.EndOfStream)
                {
                    // Get a country name.
                    countryName = inputFile.ReadLine();

                    // Add the country name to the ListBox.
                    countriesListBox.Items.Add(countryName);
                }

                // Close the file.
                inputFile.Close();
            }
            catch (Exception ex)
            {
                // Display an error message.
                MessageBox.Show(ex.Message);
            }
        }

        private void exitButton_Click(object sender, EventArgs e)
        {
            // Close the application.
            this.Close();
        }
    }
}
Calculating a Running Total

Many programming tasks require you to calculate the total of a series of numbers. In this section you learn how to calculate the total of a series of numbers that are stored in a file. For example, suppose you have a file that contains a business’s sales for each day of a week and you need to write a program that calculates the total of all the amounts in the file. The program would read the values in the file and keep a total of the values as they are read.

Programs that calculate the total of a series of numbers typically use two elements:

- A loop that reads each number in the series
- A variable that accumulates the total of the numbers as they are read

The variable that is used to accumulate the total of the numbers is called an accumulator. It is often said that the loop keeps a running total because it accumulates the total as it reads each number in the series. Figure 5-39 shows the general logic of a loop that calculates a running total.

Figure 5-39 Logic for calculating a running total

![Flowchart for calculating a running total]

When the loop finishes, the accumulator will contain the total of the numbers that were read by the loop. Notice that the first step in the flowchart is to set the accumulator variable to 0. This is a critical step. Each time the loop reads a number, it adds it to the accumulator. If the accumulator starts with any value other than 0, it will not contain the correct total when the loop finishes.

In Tutorial 5-7 you will complete an application that calculates a running total of the values in a file.
Tutorial 5-7: Calculating a Running Total

In this tutorial you complete the Total Sales application that is found in the Chap05 folder of this book’s Student Sample Programs. The application’s form has already been created and is shown in Figure 5-40. The application also has an accompanying text file named Sales.txt that is stored in the bin\Debug folder, under the project folder. The Sales.txt file contains the amounts shown in Figure 5-41.

Figure 5-40 The Total Sales application’s form

Figure 5-41 Contents of the Sales.txt file

When the completed application runs and the user clicks the Read Sales & Calculate Total button, the application calculates the total of the values in the Sales.txt file and displays the total in the totalLabel control.

Step 1: Start Visual Studio (or Visual Studio Express). Open the project named Total Sales in the Chap05 folder of this book’s Student Sample Programs.

Step 2: Open the Form1 form’s code in the code editor. Insert the using System.IO; directive shown in line 10 of Program 5-7 at the end of this tutorial.

Step 3: Open the Form1 form in the Designer. The form is shown, along with the names of the important controls, in Figure 5-40. Double-click the calculateButton control. This opens the code editor, and you will see an empty event handler named calculateButton_Click. Complete the calculateButton_Click event handler by typing the code shown in lines 23–56 in Program 5-7. Let’s take a closer look at the code:
Line 23: This is the beginning of a try-catch statement, which handles any exceptions that are thrown while the file is being processed. If an exception is thrown by any statement in the try block, the program jumps to the catch clause in line 52.

Lines 26–27: These statements declare the decimal variables sales and total. The sales variable holds each value that is read from the file, and the total variable is used as an accumulator. Notice that the total variable is explicitly initialized to 0.

Lines 30–33: After these statements have executed, the Sales.txt file is opened for reading, and the inputFile variable references a StreamReader object that is associated with the file.

Line 36: This is the beginning of a while loop that iterates as long as the end of the Countries.txt file has not been reached.

Line 39: This statement reads a line of text from the file, converts it to a decimal, and assigns the result to the sales variable.

Line 42: This statement adds the sales variable to the total variable.

Line 46: This statement closes the file.

Line 49: This statement displays the total, formatted as currency, in the totalLabel control.

Step 4: Switch your view back to the Designer and double-click the exitButton control. In the code editor you will see an empty event handler named exitButton_Click. Complete the exitButton_Click event handler by typing the code shown in lines 61–62 in Program 5-7.

Step 5: Save the project. Then, press F5 on the keyboard or click the Start Debugging button on the toolbar to compile and run the application. When the application runs, click the Read Sales & Calculate Total button. The total sales should be calculated and displayed, as shown in Figure 5-42. Click the Exit button to exit the application.

Figure 5-42 The Total Sales application displaying the total sales

Program 5-7 Completed code for Form1 in the Total Sales application

1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel;
4 using System.Data;
5 using System.Drawing;
6 using System.Linq;
7 using System.Text;
namespace Total_Sales
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void calculateButton_Click(object sender, EventArgs e)
        {
            try
            {
                // Variables
                decimal sales;        // To hold a sales amount
                decimal total = 0m;   // Accumulator, set to 0

                // Declare a StreamReader variable.
                StreamReader inputFile;

                // Open the file and get a StreamReader object.
                inputFile = File.OpenText("Sales.txt");

                // Read the file’s contents.
                while (!inputFile.EndOfStream)
                {
                    // Get a sales amount.
                    sales = decimal.Parse(inputFile.ReadLine());

                    // Add the sales amount to total.
                    total += sales;
                }

                // Close the file.
                inputFile.Close();

                // Display the total.
                totalLabel.Text = total.ToString("C");

            }
            catch (Exception ex)
            {
                // Display an error message.
                MessageBox.Show(ex.Message);
            }
        }

        private void exitButton_Click(object sender, EventArgs e)
        {
            // Close the form.
            this.Close();
        }
    }
}
5.7 The OpenFileDialog and SaveFileDialog Controls

**Checkpoint**

5.18 What is an output file?
5.19 What is an input file?
5.20 What three steps must be taken by a program when it uses a file?
5.21 What is the difference between a text file and a binary file?
5.22 What are the two types of file access? What is the difference between these two?
5.23 What type of object do you create if you want to write data to a text file?
5.24 What type of object do you create if you want to read data from a text file?
5.25 If you call the `File.CreateText` method and the specified file already exists, what happens to the existing file?
5.26 If you call the `File.AppendText` method and the specified file already exists, what happens to the existing file?
5.27 What is the difference between the `WriteLine` and `Write` methods discussed in this chapter?
5.28 What method do you call to open a text file to read data from it?
5.29 What is a file’s read position? Initially, where is the read position when an input file is opened?
5.30 How do you read a line of text from a text file?
5.31 How do you close a file?
5.32 Assume `inputFile` references a `StreamReader` object that is associated with an open file. Which of the following loops is written in the correct general format to read all of the items from the file?

```csharp
Loop A:
while (inputFile.EndOfStream)
{
    // Read an item from the file.
}

Loop B:
while (!inputFile.EndOfStream)
{
    // Read an item from the file.
}
```

**The OpenFileDialog and SaveFileDialog Controls**

**CONCEPT:** The OpenFileDialog and SaveFileDialog controls allow your application to display standard Windows dialog boxes for opening and saving files. These allow the user to easily specify a file’s name and location.

So far, the applications in this chapter that open a file specify the filename as a string literal. Most Windows users, however, are accustomed to using a dialog box to browse their disk for a file to open or for a location to save a file. You can use the OpenFileDialog and SaveFileDialog controls to equip applications with standard Windows dialog boxes for these purposes.
The OpenFileDialog Control

The OpenFileDialog control displays a standard Windows Open dialog box, such as the one shown in Figure 5-43. The Open dialog box is useful in applications that must open an existing file because it allows the user to browse the system and select the file.

Figure 5-43 Windows Open dialog box

Adding the OpenFileDialog Control to Your Project

To add an OpenFileDialog control to a form, double-click the OpenFileDialog tool under the Dialogs group in the Toolbox window. When the control is created, it does not appear on the form, but in an area at the bottom of the Designer known as the component tray. Figure 5-44 shows an example of how an OpenFileDialog control appears in the component tray. The control’s default name is openFileDialog1. As with other controls, you can change the control’s Name property to change its name.

Displaying an Open Dialog Box

In code, you can display an Open dialog box by calling the OpenFileDialog control’s ShowDialog method. For example, assume that we have created an OpenFileDialog control and changed its name to openFile. The following statement calls the control’s ShowDialog method:

```
openFile.ShowDialog();
```

In most cases, however, you will want to know whether the user clicked the Open button or the Cancel button to dismiss the Open dialog box. If the user clicked the Open button, it means that the user has selected a file and he or she can open it. If the user clicked the Cancel button, it means that the user does not want to proceed.

The ShowDialog method returns a value that indicates which button the user clicked to dismiss the dialog box. If the user clicked the Open button, the value DialogResult.OK is returned. If the user clicked the Cancel button, the value DialogResult.Cancel is returned. Assuming openFile is the name of an OpenFileDialog control, the following is
an example of an if-else statement that calls the ShowDialog method and determines whether the user clicked the Open button or the Cancel button.

```csharp
if (openFile.ShowDialog() == DialogResult.OK)
{
    MessageBox.Show("You clicked the Open button.");
}
else
{
    MessageBox.Show("You clicked the Cancel button.");
}
```

**The Filename Property**

When the user selects a file with the Open dialog box, the file’s path and filename are stored in the control’s Filename property. Assume openFile is the name of an OpenFileDialog control. The following code is an example of how you can display an Open dialog box and, if the user clicks the Open button to dismiss the dialog box, open the selected file.

```csharp
StreamReader inputFile;
if (openFile.ShowDialog() == DialogResult.OK)
{
    // Open the selected file.
    inputFile = File.OpenText(openFile.Filename);
    // Continue processing the file...
}
else
{
    MessageBox.Show("Operation canceled.");
}
```
Let’s take a closer look at the code. The statement in line 1 declares a `StreamReader` variable named `inputFile`. The `if` statement in line 3 calls the `openFile` control’s `ShowDialog` method to display an `Open` dialog box. If the user clicks the `Open` button to dismiss the dialog box, the program continues to line 6, where the name of the selected file is retrieved from the control’s Filename property and that file is opened. Otherwise (if the user clicks the `Cancel` button), the program jumps to the `else` clause in line 10.

**TIP:** When you create an OpenFileDialog control, its Filename property is initially set to the control’s default name. For example, if the control’s default name is `openFileDialog1`, then the Filename property is also set to `openFileDialog1`. Always be sure to delete the default value of the Filename property.

**The InitialDirectory Property**

By default, the `Open` dialog box displays the contents of the user’s `Documents` directory (or folder). You can specify another directory to be initially displayed by storing its path in the `InitialDirectory` property. For example, the following code stores the path `C:\Data` in the `openFile` control’s `InitialDirectory` property before displaying an `Open` dialog box:

```csharp
openFile.InitialDirectory = "C:\Data";

if (openFile.ShowDialog() == DialogResult.OK)
{
    // Continue to process the selected file...
}
else
{
    // The operation was canceled.
}
```

In this example, when the `Open` dialog box is displayed it shows the contents of the directory `C:\Data`.

**The Title Property**

By default, the word `Open` is displayed in an `Open` dialog box’s title bar. You can change the default text displayed in the title bar by changing the control’s `Title` property.

**The SaveFileDialog Control**

The `SaveFileDialog` control displays a standard Windows `Save As` dialog box, such as the one shown in Figure 5-45. The `Save As` dialog box allows the user to browse the system and select a location and name for a file that is about to be saved.

**Adding the SaveFileDialog Control to Your Project**

The SaveFileDialog control has much in common with the OpenFileDialog control. To add a SaveFileDialog control to a form, double-click the SaveFileDialog tool under the Dialogs group in the Toolbox window. When the control is created, it appears in the component tray at the bottom of the Designer. The control will be given a default name such as `saveFileDialog1`, but you can change the name with the Name property.

**Displaying a Save As Dialog Box**

In code, you can display a `Save As` dialog box by calling the SaveFileDialog control’s `ShowDialog` method. For example, assume that we have created a SaveFileDialog
control and changed its name to `saveFile`. The following statement calls the control’s ShowDialog method:

```
saveFile.ShowDialog();
```

The method returns a value indicating whether the user clicked the `Save` button or the `Cancel` button to dismiss the `Save As` dialog box. If the user clicks the `Save` button, the value `DialogResult.OK` is returned. If the user clicks the `Cancel` button, the value `DialogResult.Cancel` is returned. Assume `saveFile` is the name of a `SaveFileDialog` control. The following is an example of an `if-else` statement that calls the `ShowDialog` method and determines whether the user clicked the `Save` button or the `Cancel` button.

```
if (saveFile.ShowDialog() == DialogResult.OK)
{
    MessageBox.Show("You clicked the Save button.");
}
else
{
    MessageBox.Show("You clicked the Cancel button.");
}
```

**The Filename Property**

When the user specifies a location and filename with the `Save As` dialog box, the file’s path and filename are stored in the control’s Filename property. Assume `saveFile` is the name of a `SaveFileDialog` control. The following code is an example of how you can display a `Save As` dialog box, and if the user clicks the `Save` button to dismiss the dialog box, open the selected file.

```csharp
StreamWriter outputFile;

if (saveFile.ShowDialog() == DialogResult.OK)
{
    outputFile = new StreamWriter(saveFile.FileName);
    // Write text to file...
    outputFile.Close();
}
else
{
    MessageBox.Show("You clicked the Cancel button.");
}
```