



Neale Cousland / Shutterstock.com

CHAPTER 1

INTRODUCTION TO COMPUTING AND PROGRAMMING

IN THIS CHAPTER, YOU WILL:

- Learn about the history of computers
- Learn to differentiate between system and application software
- Learn the steps of software development
- Explore different programming methodologies
- Learn why C# is being used today for software development
- Distinguish between the different types of applications that can be created with C#
- Explore a program written in C#
- Examine the basic elements of a C# program
- Compile, run, build, and debug an application
- Create an application that displays output
- Work through a programming example that illustrates the chapter's concepts

All Microsoft screenshots used with permission from Microsoft Corporation.

Copyright 2013 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Computers have penetrated every aspect of our society and have greatly simplified many tasks. Can you picture yourself typing a paper on an electric typewriter? Would you use an eraser to make your corrections? Would you start from scratch to increase or decrease your margins or line spacing? Can you imagine living in an age without electronic messaging or e-mail capability? What would you do without an automatic teller machine (ATM) in your neighborhood?

Computers have become such an integral part of our lives that many of their functions are taken for granted. Yet, only a few years ago, mobile apps, text messaging and cloud computing were unknown. Social media technologies like internet forums, weblogs, wikis, podcasts and social networks like Facebook were unknown. In 2012 social media became one of the most powerful sources for news updates through platforms like Twitter and Facebook. Advances in computing are occurring every day, and the programs that are loaded on your computer have become very complex. The technology of wireless communication is advancing quickly. Expectations are that tablet sales will grow by 200 percent through 2016. Over 100 million units will be sold in 2012. For most consumers, tablets are not replacements for their conventional computers, but are added devices they'll purchase. Mobile applications for smartphones, pocket and tablet PCs, and other handheld wireless computers are increasingly in demand. To reach this level of complexity, software development has gone through a number of eras, and today technical advances accumulate faster and faster. What new types of computer services and programs will be integral to our daily lives in the future? This book focuses on creating software programs. Before beginning the journey into software development, a historical perspective on computing is included to help you see the potential for advancements that awaits you.

History of Computers

Computing dates back some 5000 years. Many consider the abacus to be the first computer. Used by merchants of the past and present for trading transactions, the abacus is a calculating device that uses a system of sliding beads on a rack for addition and subtraction.

In 1642, another calculating device, called the Pascaline, was created. The Pascaline had eight movable dials on wheels that could calculate sums up to eight figures long. Both the abacus and Pascaline could perform only addition and subtraction. It was not until the 1830s that the first general-purpose computer, the Analytical Engine, was available.

Charles Babbage and his assistant, Lady Augusta Ada Bryon, Countess of Lovelace, designed the Analytical Engine. Although it was very primitive by today's standards, it was the prototype for what is known today as a general-purpose computer. The Analytical Engine included input devices, memory storage, a control unit that allowed processing instructions in any sequence, and output devices.

NOTE

In the 1980s, the U.S. Defense Department named the Ada programming language in honor of Lady Lovelace. She has been called the world's first programmer. Controversy surrounds her title. Lady Byron was probably the fourth or fifth person to write programs. She did programming as a student of Charles Babbage and reworked some of his calculations.

1

Many computer historians believe the present day to be in the fifth generation of modern computing. Each era is characterized by an important advancement. In the mid-1940s, the Second World War, with its need for strategic types of calculations, spurred on the first generation of general-purpose machines. These large, first-generation computers were distinguished by the use of vacuum tubes. They were difficult to program and limited in functionality. The operating instructions were made to order for each specific task.

The invention of the transistor in 1956 led to second-generation computers, which were smaller, faster, more reliable, and more energy efficient than their predecessors. The software industry was born during the second generation of computers with the introduction of FORTRAN and COBOL.

The third generation, 1964–1971, saw computers become smaller, as transistors were squeezed onto small silicon discs (single chips), which were called semiconductors. Operating systems, as they are known today, which allowed machines to run many different programs at once, were also first seen in third-generation systems.

As time passed, chips kept getting smaller and capable of storing more transistors, making computers more powerful and less expensive. The Intel 4004 chip, developed in 1971, placed the most important components of a computer (central processing unit, memory, and input and output controls) on a minuscule chip about half the size of a dime. Many household items such as microwave ovens, television sets, and automobiles benefited from the fourth generation of computing.

During the fourth generation, computer manufacturers tried to bring computing to general consumers. In 1981, IBM introduced its personal computer (PC). The 1980s saw an expansion in computer use as clones of the IBM PC made the personal computer even more affordable. We also saw the development of Graphical User Interfaces (GUIs) and the mouse as a handheld input device. The number of personal computers in use more than doubled from two million in 1981 to 5.5 million in 1982. Ten years later, 65 million PCs were in use.

NOTE

According to the October 2010 *U.S. Census Bureau's Current Population Survey*, released in July 2012, over 76% of households in the United States had computers.

Defining a fifth generation of systems is somewhat difficult because the generation is still young. Computers can now accept spoken word instructions, imitate human reasoning through artificial intelligence, and communicate with devices instantaneously around the

globe by transmitting digital media. Mobile apps are growing. By applying problem-solving steps, expert systems assist doctors in making diagnoses. Healthcare professionals are now using handheld devices in patients' rooms to retrieve and update patient records. Using handheld devices, drivers of delivery trucks are accessing global positioning systems (GPS) to verify locations of customers for pickups and deliveries. Sitting at a traffic light, you can check your e-mail, make airline reservations, remotely monitor and manage household appliances, and access your checking and savings accounts. Using wireless networks, students can access a professor's notes when they enter the classroom.

Major advances in software are anticipated as integrated development environments (IDEs) such as Visual Studio make it easier to develop applications for the Internet rapidly. Because of the programmability of the computer, the imagination of software developers is set free to conjure the computing functions of the future.

The real power of the computer does not lie in the hardware, which comprises the physical components that make up the system. The functionality lies in the software available to make use of the hardware. The hardware processes complex patterns of 0s and 1s. The software actually transposes these 0s and 1s into text, images, and documents that people can read. The next section begins the discussion on software.

System and Application Software

Software consists of **programs**, which are sets of instructions telling the computer exactly what to do. The instructions might tell the computer to add up a set of numbers, compare two names, or make a decision based on the result of a calculation. Just as a cook follows a set of instructions (a recipe) to prepare a dish, the computer follows instructions without adding extra salt to perform a useful task. The next sections describe the two major categories of software: system software and application software.

System Software

System software is loaded when you power on the computer. When thinking of system software, most people think of operating systems. **Operating systems** such as Windows 8, Android, iOS, Windows 7, and Linux are types of programs that oversee and coordinate the resources on the machine. Included are file system utilities, small programs that take care of locating files and keeping up with the details of a file's name, size, and date of creation. System software programs perform a variety of other functions: setting up directories; moving, copying, and deleting files; transferring data from secondary storage to primary memory; formatting media; and displaying data on screens. Operating systems include communication programs for connecting to the Internet or connecting to output devices such as printers. They include user interface subsystems for managing the look and feel of the system.

NOTE

Operating systems are one type of system software. They are utility programs that make it easier for you to use the hardware.

1

Another type of system software includes compilers, interpreters, and assemblers. As you begin learning software development, you will write instructions for the computer using a **programming language**. Modern programming languages are designed to be easy to read and write. They are called **high-level languages** because they are written in English-like statements. The programming language you will be using is **C#** (pronounced *see sharp*). Other high-level computer programming languages include Visual Basic, FORTRAN, Pascal, C, C++, and Java.

Before the computer can execute the instructions written in a programming language such as C#, the instructions must be translated into machine-readable format. A **compiler** makes this conversion. Figure 1-1 shows what a machine language instruction looks like.

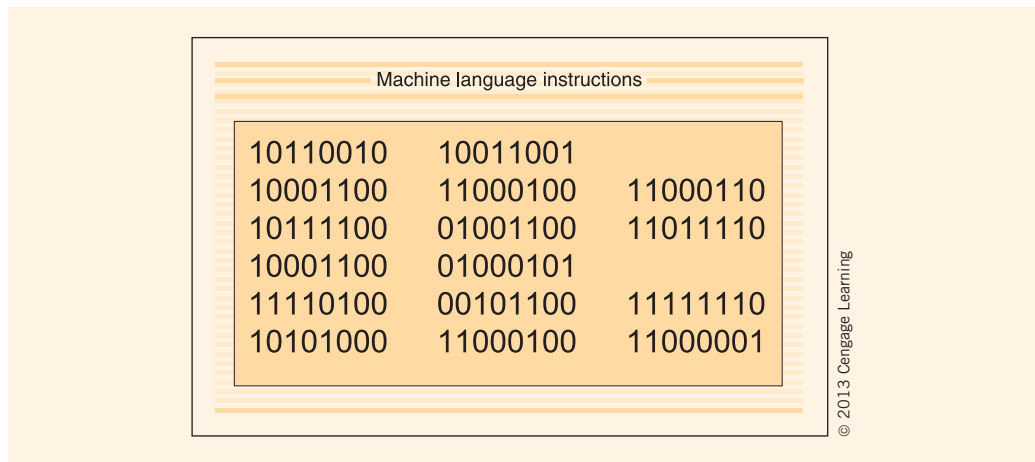


FIGURE 1-1 A machine language instruction

Just as the English language has rules for sentence construction, programming languages such as C# have a set of rules, called **syntax**, that must be followed. Before translating code into machine-readable form, a compiler checks for rule violations. Compilers do not convert any statements into machine language until all syntax errors are removed. Code can be interpreted as well as compiled. Interpreters translate one statement of code into machine-readable form and then they execute that line. They then translate the next instruction, execute it, and so on. Unlike compilers, which look at entire pieces of code, **interpreters** check for rule violations line by line. If the line does not contain an error, it is converted to machine language. Interpreters are normally slower than compilers. Many languages offer both compilers and interpreters, including C, BASIC, Python, and Lisp. **Assemblers** convert the assembly programming language, which is a low-level

programming language, into machine code. Low-level programming languages are closer to hardware. They are not as easy to read or write as high-level programming languages.

Application Software

Application software consists of programs developed to perform a specific task. The games you might play or the search engines you use on the Internet are types of application software. Word processors, such as Microsoft Word, are examples of application software. Word was written to help users create professional looking documents by including a number of editing and formatting options. Spreadsheets, such as Microsoft Excel, are types of application software designed to make numerical calculations and generate charts. Database management systems, such as SQL Server, Oracle, or Microsoft Access, were designed to organize large amounts of data, so that reports could easily be generated. Software that generates payroll checks is considered application software, as is software that helps you register for a class. E-commerce Web sites with database-driven shopping carts, such as eBay, are forms of application software. Application software is used by the banking industry to manage your checking and saving accounts. Programmers use programming languages such as C# to write application software to carry out specific tasks or to solve specific problems. The programs that you write from this book will be application software.

Software Development Process

You will soon be writing programs using C#. How do you start? Many beginning programmers just begin typing without planning or without using any organized sequence of steps. This often leads to increased development time and solutions that might not consistently produce accurate results.

Programming is a process of problem solving. Typing the program statements in a language such as C# is not the hardest part of programming. The most difficult part is coming up with a plan to solve the problem. A number of different approaches, or **methodologies**, are used to solve computer-related problems. Successful problem solvers follow a methodical approach with each programming project. Figure 1-2 illustrates the organized plan, or methodology, that is used to solve the problems presented in this book. The following section describes each step.

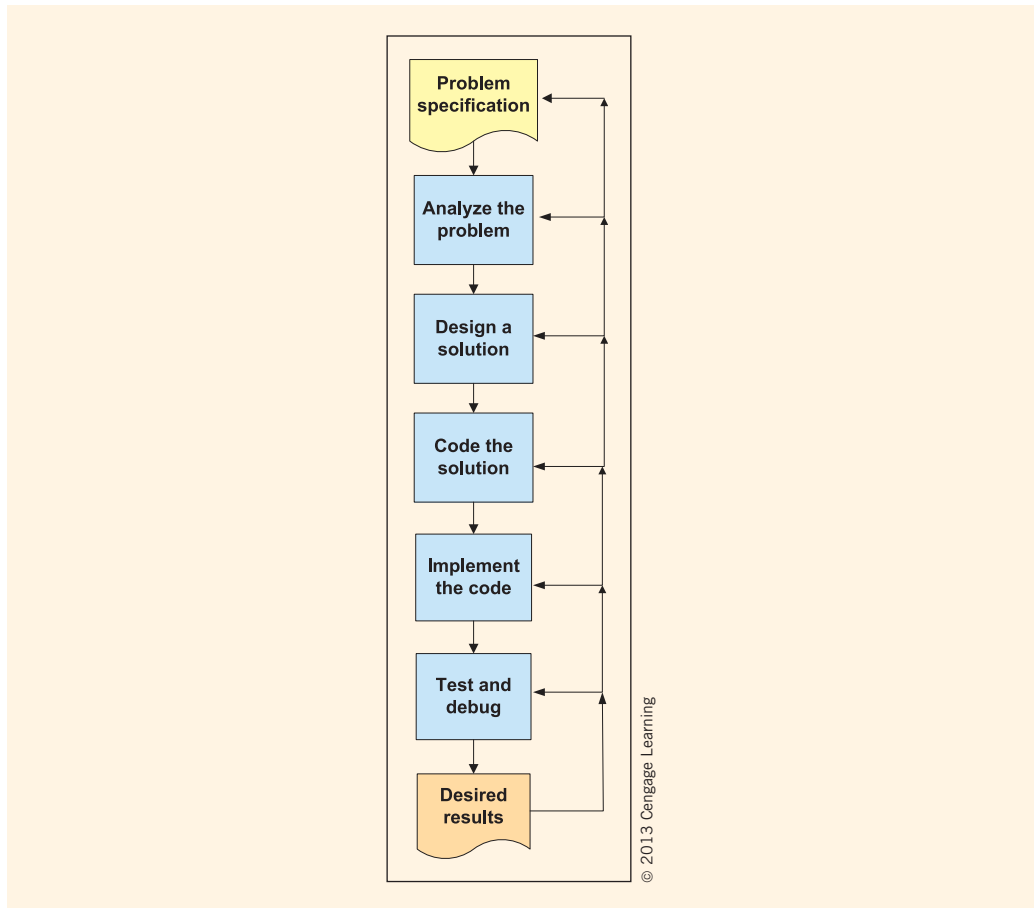


FIGURE 1-2 Steps in the software development process

Steps in the Program Development Process

1. **Analyze the problem.** The first step should be directed toward grasping the problem thoroughly. Analyze precisely what the software is supposed to accomplish. During this phase, you review the problem **specifications**, which describe what the program should accomplish. Specifications often include the desired output of the program in terms of what is to be displayed, saved, or printed. If specifications are ambiguous or need clarification, you might need to ask probing questions. *If you do not know where you are going, how will you know when you have arrived at the correct location?*

NOTE

Sometimes one of the most difficult parts of the process is getting clear specifications from the user. Unless you know what the problem is, there is no way you can solve it. Make sure you understand the problem definition.

A program specification might look like Figure 1-3.

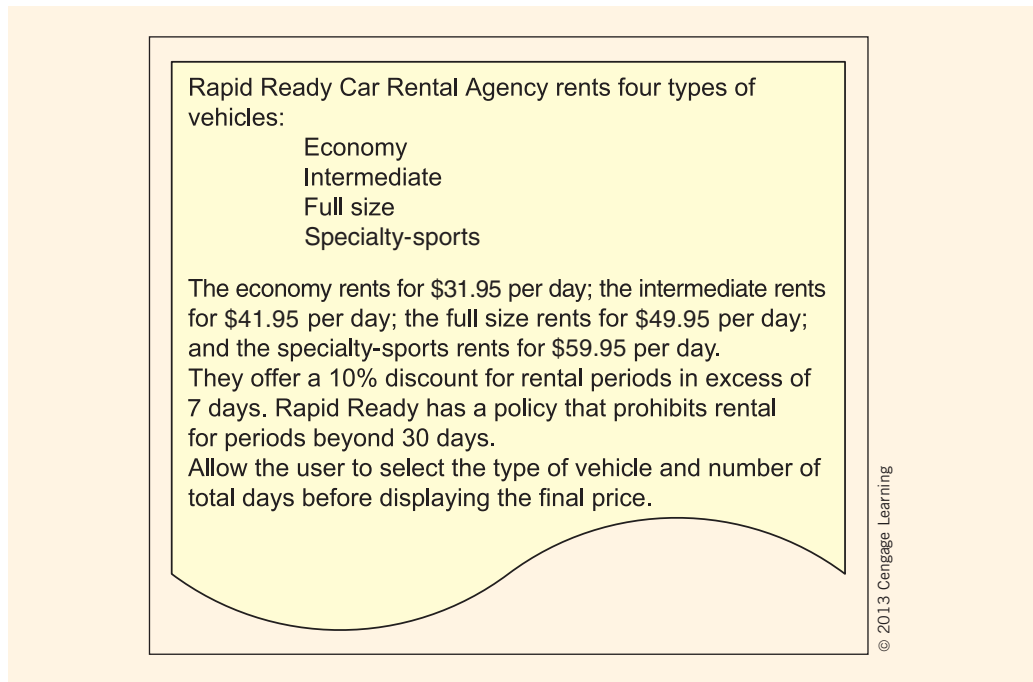


FIGURE 1-3 Program specification sheet for a car rental agency problem

During this first phase, in addition to making sure you understand the problem definition, you must also review the program inputs. You should ask the following types of questions:

- What kind of data will be available for input?
- What types of values (e.g., whole numbers, alphabetic characters, and numbers with a decimal point) will be in each of the identified data items?
- What is the **domain** (range of the values) for each input item?
- Will the user of the program be inputting values?
- If the problem solution is to be used with multiple data sets, are there any data items that stay the same, or remain **constant**, with each set?

Before you move to designing a solution, you should have a thorough understanding of the problem. It might be helpful to verbalize the problem definition. It might help to see sample input for each of the data items. Figure 1-4 illustrates how the input data items would be determined during analysis for the car rental agency problem shown in Figure 1-3. Figure 1-4 shows the identifier, or name of the data item, the type, and the domain of values for each item.

Instead of having the user enter the full words of Economy, Intermediate, Full size, or Speciality Sports, the characters E, I, F, and S could be mapped to those categories.

Data identifier	Data type	Domain of values
kindOfVehicle	char (single coded character)	E, I, F, or S
noOfDays	Integer (whole number)	1...30

© 2013 Cengage Learning

FIGURE 1-4 Data for car rental agency

2. **Design a solution.** Programmers use several approaches, or **methods**, during design. Procedural and object-oriented methodologies are the two most commonly used design methods. Some projects are more easily solved using one approach than the other. Both of these approaches are discussed in the next section. The selection of programming language sometimes weighs in when determining the approach. The C# language was designed to be very object oriented.

No matter what size the project is, careful design always leads to better solutions. In addition, careful design normally leads to solutions that can be produced in shorter amounts of time. A **divide-and-conquer** approach can be used with both methodologies. As the name implies, when you divide and conquer a programming problem, you break the problem into subtasks. Then, you conquer each of the subtasks by further decomposing them. This process is also called **top-down design**. Detailed models should be developed as input to subsequent phases.

Using the **object-oriented approach**, the focus is on determining the data characteristics and the methods or behaviors that operate on the data. These logical groupings of members (data and behavior) are referred to as a **class**. These characteristics are placed in a class diagram. Figure 1-5 contains a class diagram for the problem specification given in Figure 1-3.

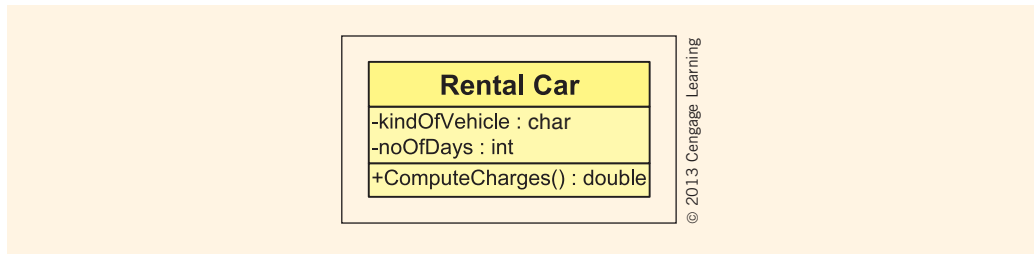


FIGURE 1-5 Class diagram of car rental agency

Figure 1-5 is a class diagram divided into three sections with the top portion identifying the name of the class. The middle portion of a class diagram always lists the data characteristics. Data representing the type of vehicle to rent and the number of days for the rental are important to a rental car agency. The bottom portion of the class diagram shown in Figure 1-5 shows what actions are to be performed with the data items. **ComputeCharges ()** is used to determine the cost of the rental using the type of vehicle and the number of rental days. You will learn more about class diagrams later in this chapter. Procedural designs, which are appropriate for simpler problem definitions, use structure charts to show the hierarchy of modules, and flowcharts or pseudocode listings to detail the steps for each of the modules.

Algorithms for the behaviors (object oriented) or processes (procedural) should be developed for both of these methodologies. An **algorithm** is a clear, unambiguous, step-by-step process for solving a problem. These steps must be expressed so completely and so precisely that all details are included. The instructions in the algorithm should be both simple to perform and capable of being carried out in a finite amount of time. Following the steps blindly should result in the same results every time.

An algorithm for **ComputeCharges ()** multiplies the number of rental days by the rate associated with the type of vehicle rented to produce the rental charge. After the algorithm is developed, the design should be checked for correctness. One way to do this is to use sample data and **desk check** your algorithms by mimicking the computer; in other words, walking through the computer's steps. Follow each step precisely, one step at a time. If the problem involves calculations, use a calculator, and follow your design statements exactly. It is important when you desk check not to add any additional steps, unless you go back and revise the algorithm.

During this phase, it might also be helpful to plan and design the look of the desired output by developing a prototype. A **prototype** is a mock-up of screens depicting the look of the final output.

3. **Code the solution.** After you have completed the design and verified that the algorithm is correct, you translate the design into source code.

Source code consists of program statements written using a programming language, such as C#.

NOTE

Source code statements can be typed into the computer using an editor such as Notepad or an **integrated development environment (IDE)**, such as Visual Studio. IDEs include a number of useful development tools: IntelliSense (pop-up windows with completion options), debugging, color coding of different program sections, online help and documentation, and features for running the program.

You must follow the syntax of the language when you code the solution. Whether you speak English, Spanish, or another language, you are accustomed to following language **syntax**, or rules. For example, the syntax of the English language dictates that statements end with periods and include subjects and verbs. When you write in English, you are expected to follow those rules. When you write in the C# programming language, you are expected to follow the rule that every statement should end with a semicolon. It is at this third phase of the program development process (code the solution) that you must concern yourself with language syntax.

Many programmers use an **iterative approach** in the software development process. This means that you might find it necessary to go back to the design stage to make modifications. There might even be times when additional analysis is necessary. If you analyze and design thoroughly before attempting to code the solution, you usually develop a much better program that is easier to read and modify.

4. **Implement the code.** During this phase, the typed program statements (source code) are compiled to check for rule violations. Integrated development environments (IDEs) such as Visual Studio supply compilers within the development environment. The output of the compiler is a listing of the errors along with a brief description of the violation. Before the implementation can go forward, all the syntax errors must be corrected. When rule violations are eliminated, the source code is converted into the Microsoft **Intermediate Language (IL)**. All languages targeting the .NET (pronounced dot net) platform compile into an IL. The language that you will be using in this book, C#, is a language introduced as part of the .NET platform. Like C#, other languages, such as Java, compile code into an intermediate language. Java's intermediate language is called **bytecode**. Intermediate languages facilitate the use of code that is more platform independent than other languages that compile straight into the machine language of the specific platform.

NOTE

If you are using the Visual Studio IDE, you might not be aware of the IL's presence. You simply select options to compile, build, and execute the program to see the output results.

The IL code is between the high-level source code and the **native code**, which is the machine language code of a particular computer. IL code is not directly executable on any computer. It is not in the language of the computer, which means it is not tied to any specific CPU platform. A second step is required before you see the results of the application.

This second step is managed by .NET's **common language runtime (CLR)**. CLR loads predefined .NET classes used by the program into memory and then performs a second compile, called a **just-in-time (JIT) compilation**. This converts the IL code to the platform's native code. The CLR tool used for this is a just-in-time compiler called **JITer**. JITer reads the IL and produces the machine code that runs on the particular platform. Any computer that executes the code must have the CLR installed. The CLR is included with the .NET Framework. Any computer executing .NET code must have the .NET Framework installed. Figure 1-6 illustrates the steps that must be performed for source code written in C# to be executed.

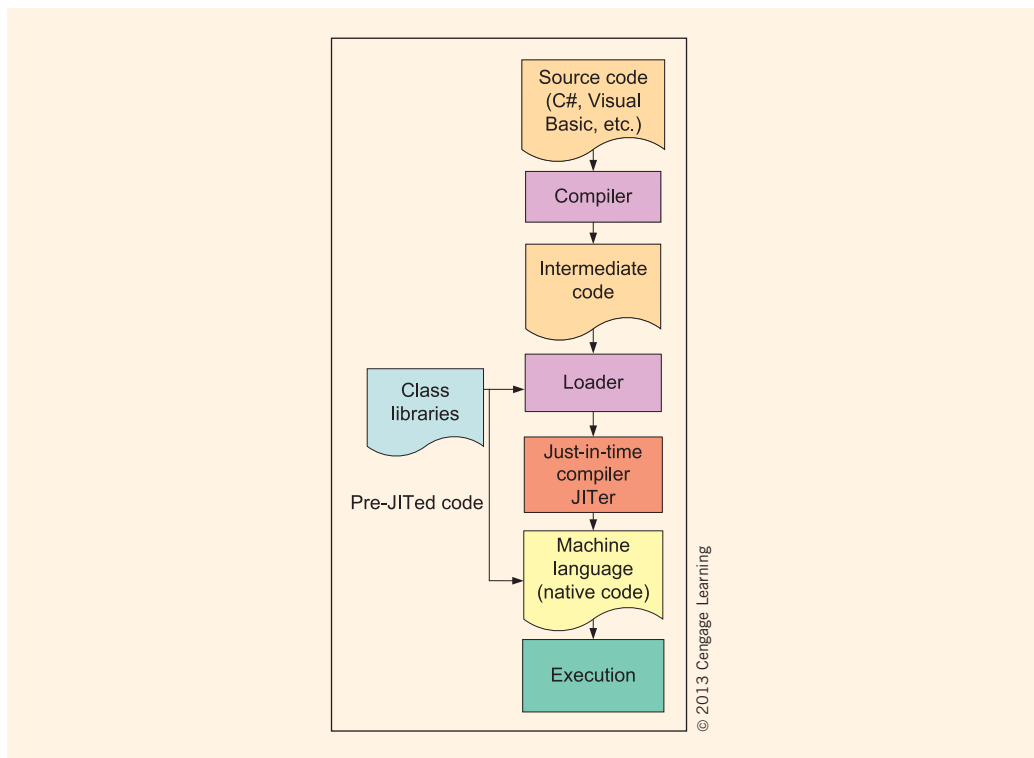


FIGURE 1-6 Execution steps for .NET

5. **Test and debug.** Even though you have no compiler syntax errors and you receive output, your results might be incorrect. You must test the program to ensure that you get consistent results. The **testing** phases are often shortchanged. Only after thoroughly testing can you be sure that your program is running correctly.

Plan your testing. Good programmers often build **test plans** at the same time they are analyzing and designing their solutions. This test plan should include testing extreme values, identifying possible problem cases, and ensuring that these cases are tested. After syntax errors are eliminated and results are being produced, you should implement the test plan verifying that the results are accurate. If the application is interacting with a user for input of data, run the program multiple times with the planned test values. For calculations, perform the same operations using a calculator, much as you did during the design phase when you desk checked your algorithm. There are software development methodologies built around test development. For example, **Test Driven Development** (TDD) is a programming methodology that emphasizes fast, incremental development and writing tests before writing code. With TDD, additional functionality is added only after the first tests are passed. The first cycle normally deals with very simple cases. After you have these very simple tests working, you add more functionality, a bit at a time.

During testing, **logic errors** are often uncovered. Logic errors might cause an abnormal termination of the program or just produce incorrect results. These types of errors are often more difficult to locate and correct than syntax errors. A **run-time error** is one form of logic error. Run-time errors normally cause program crashes (stopping execution) and the reporting of error messages. For example, if you attempt to divide by zero, your program might crash. To further complicate matters, a program might sometimes work properly with most test data, but crash when a certain value is entered. This is why it is so important to make sure you thoroughly test all applications. When a logic error is detected, it might be necessary to go back to Step 1, reanalyze the problem specifications, and redesign a solution. As you look back at Figure 1-2, notice the figure shows that the software development process is iterative. As errors are discovered, it is often necessary to cycle back to a previous phase or step.

Programming Methodologies

How do you ride a bicycle? How do you drive a car? How do you do your laundry? How do you prepare for an exam? As you think about those questions, you probably have an answer, and your answer will differ from those of other readers. But, you have some strategy, a set of steps, which you follow to get the job done. You can think of a methodology as a strategy, a set of steps, or a set of directions. Programmers use a

number of different programming methodologies. The two most popular programming paradigms are structured procedural programming and object-oriented programming. These approaches are discussed in this section.

Structured Procedural Programming

This approach emerged in the 1970s and is still in use today. **Procedural programming** is process oriented focusing on the processes that data undergoes from input until meaningful output is produced. This approach is very effective for small stand-alone applications. The five steps for software development—analyze, design, code, implement, and test and debug—which were identified in the preceding section, work well for the structured procedural approach.

During design, processing steps are defined in terms of an algorithm. Any formulas or special processing steps are included in the algorithm. To think algorithmically, programmers use a number of tools. One such tool used is a flowchart. Figure 1-7 shows some of the symbols used in a flowchart for the construction of an algorithm.

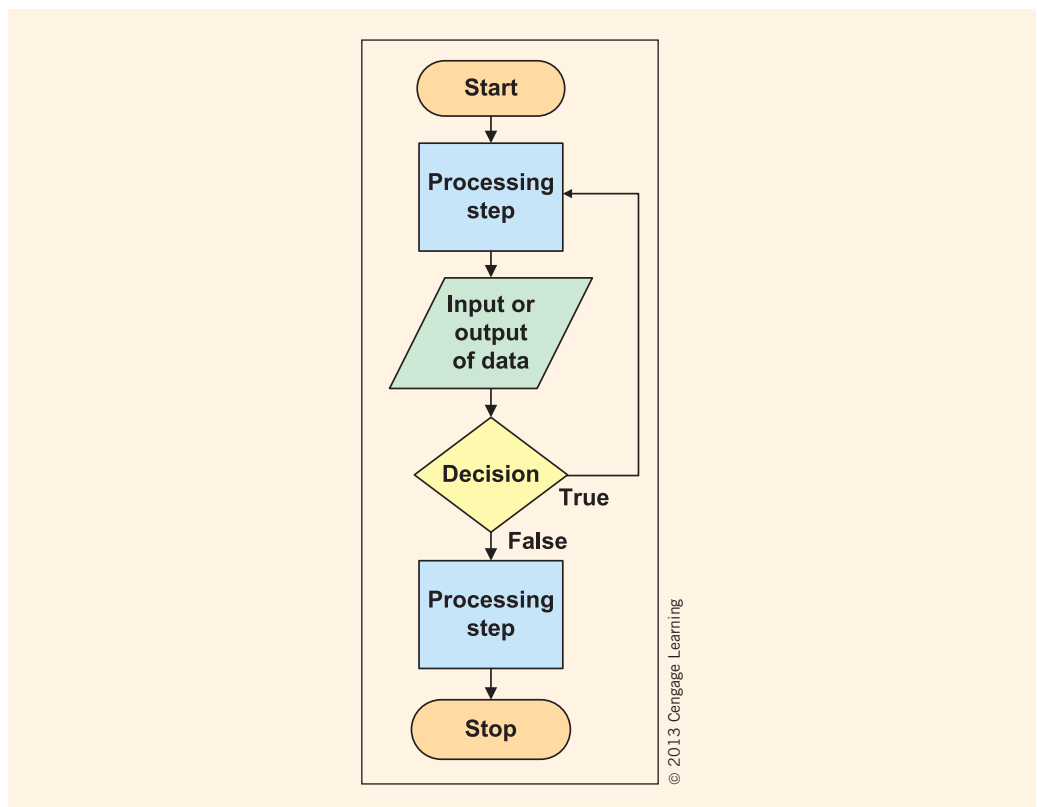


FIGURE 1-7 Flowchart symbols and their interpretation

Another tool used to develop an algorithm during design is **pseudocode**. As the name implies, with pseudocode, steps are written in pseudo or approximate code format, which looks like English statements. The focus is on determining and writing the processes or steps involved to produce the desired output. With pseudocode, the algorithm is written using a combination of English statements and the chosen programming language, such as C#. Verbs like compute, calculate, sum, print, input, and display are used to imply what type of activity is needed to reach the desired result. While, do while, for, and for each are used to imply looping or that steps should be performed more than one time. When decisions or tests are required, if and if else are used. Indentation is used to show which program statements are grouped together. Figure 1-8 shows example pseudocode for the Rapid Ready car rental problem specification.

```

if (desired number of days > 30)
    display message "Sorry can not rent more than 30 days"
else
    {
        if (desired number of days > 7)
            discount percent = 0.10
        else
            discount percent = 0

        if (car type = 'E')
            rate = 31.95
        else
            if (car type = 'I')
                rate = 41.95
            else
                if (car type = 'F')
                    rate = 49.95
                else
                    if (car type = 'S')
                        rate = 59.95

        calculate cost before discount = rate * desired number of days
        calculate discount amount = cost before discount * discount percent
        calculate final charge = cost before discount - discount amount
    }

```

FIGURE 1-8 Pseudocode or Structured English for the Rental Car application

Structured programming is associated with a technique called **top-down design** or **stepwise refinement**. The underlying theme or concept is that given a problem definition, you can refine the logic by dividing and conquering. The problem can be divided into subproblems, or procedures. Then, each of the subproblems is further decomposed. This continues until you reach subproblems that are straightforward enough to be solved easily at the subproblem level. After you arrive at a solution at the lower levels, these solutions are combined to solve the overall problem.

NOTE

Consider the analogy of building a house. Using top-down design, this problem might be decomposed into Prepare the Land, Construct the Foundation, Frame the Structure, Install the Roof, Finish the Interior, and Complete the Exterior. Then, each of these subproblems could be further subdivided. An overall contractor could be hired for the project and then subcontractors assigned to each of the subproblem areas. Within each subproblem, additional problems would be defined and separate workers assigned to that area. For example, Finish the Interior might be further divided into Walls, Floors, and so on. Walls would be further decomposed into Hang Sheet Rock, Mud the Walls, Prepare for Paint, Paint the Walls, Paint the Trim, and so on. Again, each of these areas could be subcontracted out.

As you think about breaking the Rental Car problem into smaller subprograms, you might consider that the data must be entered or inputted, number of days tested to determine whether you can rent the car or not, type of desired vehicle tested to determine the rate, final calculations made, and then the results displayed. Each of those small subprograms could be further divided into multiple program statements.

Programmers using the structured procedural approach normally write each of the subprograms as separate functions or methods that are called by a main controlling function or module. This facilitates the divide-and-conquer approach. With larger problems, the subprograms can be written by different individuals. One of the drawbacks of the procedural approach involves **software maintenance**. When an application is upgraded or changed, programs written using the procedural approach are more difficult to maintain. Even minor modifications can affect multiple functions and require additional modifications to be made. There is also less opportunity to reuse code than with the object-oriented methodology.

Object-Oriented Programming

Object-oriented programming developed as the dominant programming methodology in the early and mid-1990s. Today it is viewed as a newer approach to software development. The concept behind object-oriented programming (OOP) is that applications can be organized around objects rather than processes. This methodology includes a number of powerful design strategies that facilitate construction of more complex systems that model

real-world entities. The C# language was designed to take advantage of the benefits of the object-oriented methodology.

With **object-oriented analysis, design, and programming**, the focus is on determining the objects you want to manipulate rather than the processes or logic required to manipulate the data. Remember that the procedural approach focuses on processes. One of the underlying assumptions of the object-oriented methodology is that the world contains a number of entities that can be identified and described. An **entity** is often defined as a person, place, or thing. It is normally a noun. By **abstracting out the attributes** (data) and the **behaviors** (processes on the data), you can divide complex phenomena into understandable entities. An **abstraction** is simply a description of the essential, relevant properties of an entity. For example, you should easily be able to picture in your mind, or conceptualize, the entities of people, school personnel, faculty, student, undergraduate student, graduate student, vehicle, car, book, school, animal, dog, and poodle by describing **characteristics** or attributes, and **behaviors** or actions, about each of these.

Consider the case of a student. A student has these characteristics or attributes: student ID, name, age, GPA, major, and hometown. The characteristics can be further described by identifying what type or kind of data might exist in each of them. For example, alphabetic characters would be found in the name attribute. Whole numbers without a decimal would be found in the age attribute, and numbers with a decimal would be in the GPA attribute.

In addition to abstracting the data properties, you should also be able to think about some of the actions or behaviors of each of the entities identified in the previous paragraph. Behaviors associated with a student include actions such as Apply for Admission, Enroll as Student, Get Final Grade, Change Name, and Determine GPA. Using the object-oriented methodology, these attributes and actions (or characteristics and behaviors) are **encapsulated**, which means that they are combined together to form a class.

A **class diagram** is one of the primary modeling tools used by object-oriented programmers. Figure 1-9 illustrates the Student class diagram using the Unified Modeling Language (UML) notation. The top portion of the diagram identifies the name of the class, which is Student. The section in the center contains the data members and the type of data that would be found in each data member. **Information hiding** is an important component of object-oriented programming. The minus symbol (–) to the left of the data member’s name refers to the access modifier and indicates the member is private and accessible to that class only. The bottom portion of the diagram in Figure 1-9 shows actions, or methods, of the Student class. The plus symbol (+) access modifier indicates that the behaviors are public and available outside of the class. You will read more about class diagrams in upcoming chapters. This UML notation is used throughout the book for creating class diagrams.

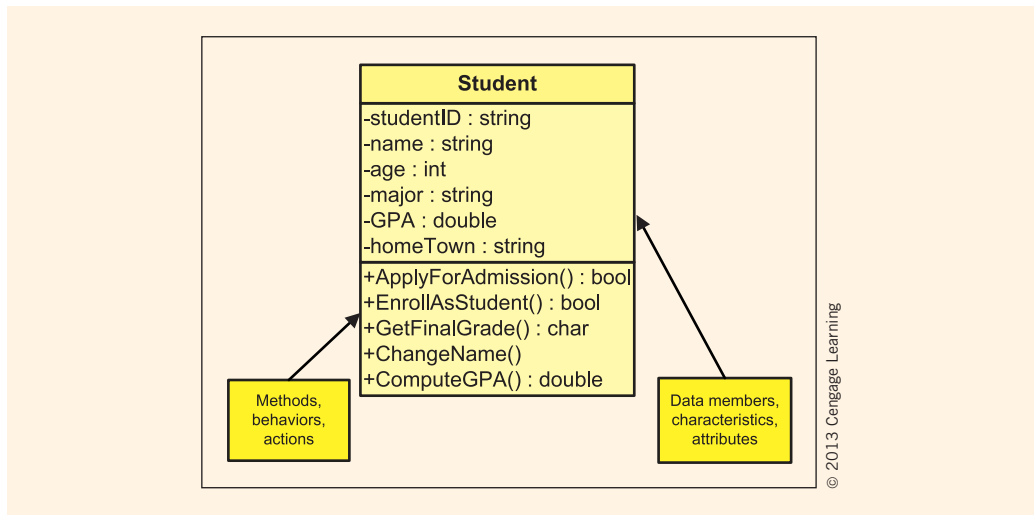


FIGURE 1-9 Student class diagram

A class is like a template. It is similar to a blueprint for a house. Even though you define all the characteristics of a house in the blueprint, the house does not exist until one is built using that template. Many houses can be created using the same template. In object-oriented terminology, the constructed house is one instance (object) of the blueprint or template. You **instantiate** the blueprint by building a house using that template. Many objects of a particular class can be instantiated. Thus, an **object** is an **instance** of the class.

Examine the expanded example of a student with the following data members:

- Student number: 122223
- Student name: Justin Howard
- Age: 18
- GPA: 3.80
- Major: CS
- Hometown: Winchester, Kentucky

When these data members are associated with the class, an object is created or **constructed**. A second object could be instantiated from that class by giving values to its members (e.g., 228221, Elizabeth Czerwinski, 21, 4.0, ENG, Reno, Nevada).

The object-oriented methodology facilitates designing components, which contain code that can be reused by packaging together the attributes of a data type along with the actions of that type. In Chapter 4 you will create your own classes that you instantiate.

Through **inheritance**, it is possible to define subclasses of data objects that share some or all of the parent's class characteristics. This is what enables reuse of code. For example, you should be able to visualize that Student is a subset of Person, just as Teacher is a subset of Person. Using object-oriented design, you could abstract out the common characteristics of all people and place them in a superclass. Through inheritance, Student and Teacher can use the characteristics of Person and add their own unique members.

Another important object-oriented concept is polymorphism. Behaviors or methods of parent and subclasses can have the same name, but offer different functionality. Through **polymorphism**, you are able to invoke methods of the same name on objects of different classes and have the correct method executed. For example, you might have subclasses of `UndergraduateStudent` and `GraduateStudent`. They could both inherit characteristics from the `Student` **class**. Both of the subclasses might have their own method that contains details about how to determine their cost of tuition. Both subclasses might name their method `DetermineTuitionCosts()`. Through polymorphism, the correct method is executed based on which object invoked it. When an object of the `UndergraduateStudent` **class** is used, the `DetermineTuitionCosts()` method of the `UndergraduateStudent` **class** is used. When an object of the `GraduateStudent` **class** is used, the `DetermineTuitionCosts()` method of the `GraduateStudent` **class** is used. You will read much more about object-oriented features in upcoming chapters. In Chapter 11 you will read about advanced object-oriented programming features including inheritance and polymorphism. In that chapter you will write multiclass solutions.

The object-oriented principles are of particular importance when writing software using C#. No program can be written that does not include at least one class. All program statements written using the C# language are placed in a class.

Whether you are using a procedural or object-oriented approach, you should follow the five steps to program development. As with the procedural approach, the object-oriented development process is iterative. During design and subsequent phases, do not hesitate to reconsider analysis and design decisions.

Evolution of C# and .NET

Programming Languages

In the 1940s, programmers toggled switches on the front of computers to enter programs and data into memory. That is how some of the early programming began. Even when they moved to punching 0s and 1s on cards and reading the cards into memory, it could not have been much fun to be a programmer. Coding was very tedious and prone to error. In the 1950s, assembly languages replaced the binary notation by using mnemonic symbols to represent the instructions for the computer. Symbols such as MV were used to represent moving the contents of one value in memory to another memory location. Assembly languages were designed to make the programmer's job easier by using these mnemonic symbols instead of binary numbers. However, the instructions depended on the particular CPU architecture. Statements to perform the same task differed from

computer to computer. Assembly languages are still considered **low-level programming languages**. As you can see from Figure 1-10, these types of instructions are not easy to read or understand. They are not considered close to the English language, as high-level programming languages such as C# and Java are.

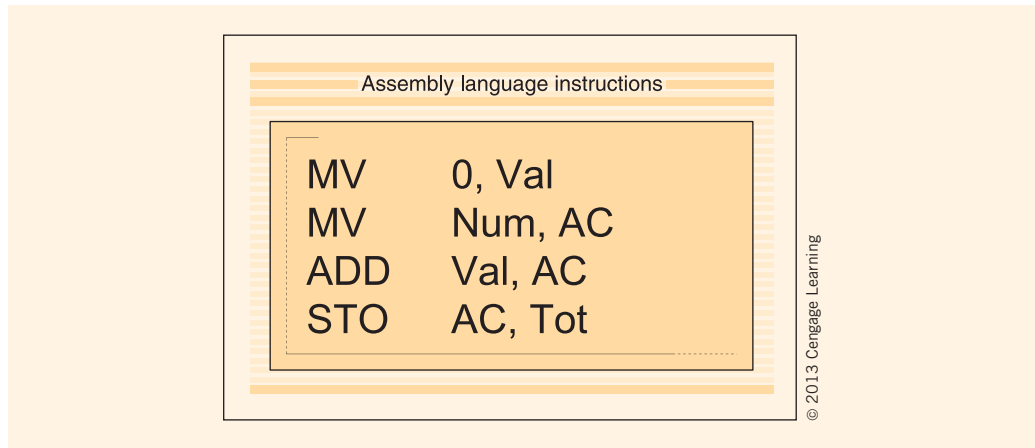


FIGURE 1-10 Assembly language instruction to add two values

High-level languages came into existence in the late 1950s with **FORTRAN** (Formula Translator) and later **COBOL** (Common Business Oriented Language). These languages were considered **high-level languages** because they were designed to be accessible to humans, easy to read and write, and close to the English language. Since then, more than 2000 high-level languages have come into existence. Some have gone by the wayside, while others have evolved and are still widely used today.

Some of the more noteworthy high-level programming languages are C, C++, Visual Basic, Java, and now C#. **C++** is an extension of C, which actually evolved from BCPL and B in 1973. Dennis Ritchie is credited with developing the C programming language; Bjarne Stroustrup at Bell Labs is considered the father of C++ for his design work in the early 1980s. C++ includes features that enabled programmers to perform object-oriented programming. C++ is used heavily in the computer industry today.

Smalltalk, considered a pure object-oriented language, was developed at the Xerox Palo Alto Research Center (PARC). Visual Basic, introduced in 1991, derived from the BASIC (Beginners All Purpose Symbolic Code) programming language, a language developed in the 1960s. The earlier versions of Visual Basic did not facilitate development using an object-oriented approach. Earlier versions of Visual Basic did, however, facilitate easy creation of Windows-based graphical user interfaces (GUIs). Visual Basic has been used for a great deal of application development because of this.

Java was introduced in 1995 and was originally called Oak. It was originally designed to be a language for intelligent consumer-electronic devices such as appliances and microwave ovens. Instead of being used for that purpose, the business community used Java most heavily for Web applications because of the nature of the bytecode, which enabled machine-independent language compiling. Because the bytecode does not target any particular computer platform and must be converted to the language of the system running the application, this facilitates development of code that runs on many types of computers.

C# is one of the newer programming languages. It conforms closely to C and C++, but many developers consider it akin to Java. There are a number of similarities between the languages. It has the rapid graphical user interface (GUI) features of previous versions of Visual Basic, the added power of C++, and object-oriented class libraries similar to Java. C# was designed from scratch by Microsoft to work with the new programming paradigm, .NET, and was the language used most heavily for the development of the .NET Framework class libraries. C# can be used to develop any type of software component, including mobile applications, dynamic Web pages, database access components, Windows desktop applications, Web services, and console-based applications. You will be using C# for the software development in this book; however, the concepts presented can be applied to other languages. The intent of the book is to use the C# language as a tool for learning how to develop software rather than to focus on the syntax of C#.

.NET

When you think about C#, you should also understand its relationship to .NET. **.NET** is an environment in which programs run and was designed to be a new programming paradigm. It is not an operating system, but rather a layer between the operating system and other applications. As such, it provides a platform for developing and running code that is easy to use. .NET is an integral part of many applications running on Windows and provides common functionality for those applications to run. Microsoft stated that the vision for .NET was to provide a new programming platform and a set of development tools. The intent was for developers to be able to build distributed component-based applications. By the year 2000, when Microsoft unveiled .NET, many developers had already experienced .NET because of the massive marketing campaign of Microsoft.

Before its official announcement in 2000, .NET was in development for over three years. Microsoft distributed a number of beta versions before the official release. A **beta version** is a working version that has not been fully tested and may still contain **bugs** or errors. It was not until February 2002 that Microsoft finally released the first version of Visual Studio, the IDE for developing C# applications. Visual Basic, Visual C++, and Visual C# all use this same development environment. Figure 1-11 shows Visual Studio with a C# program running. The output of the program “Welcome to Programming!” is shown in the small message box near the bottom of the figure.

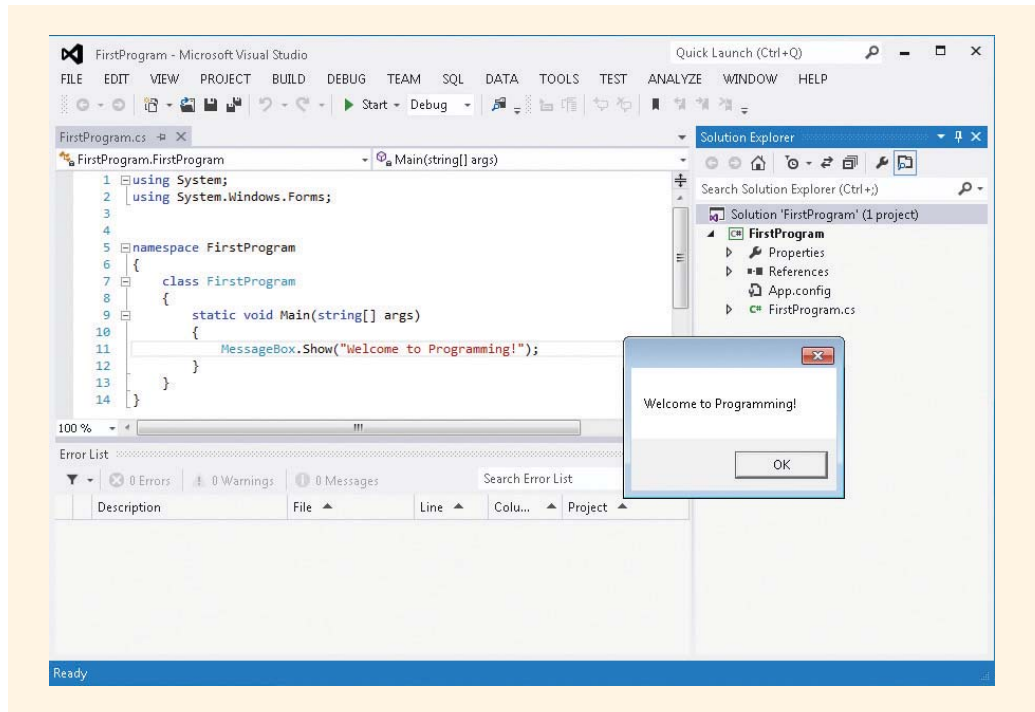


FIGURE 1-11 Visual Studio integrated development environment

NOTE

Microsoft dropped .NET from the name of Visual Studio for the 2005 version.

Included in Visual Studio are tools for typing program statements, and compiling, executing, and debugging applications. The new concepts introduced as part of .NET are outlined in the following paragraphs.

Multilanguage independence: .NET supports development using a number of programming languages: Visual Basic, C#, C++, and a number of third-party languages. All the code from each of the languages can compile to the common Microsoft Intermediate Language (IL).

Part of the application might be developed using Visual Basic and another portion developed using C#. When they are compiled, they translate to a common IL. The following list describes some of the common features of the .NET-supported languages:

Framework base classes: The .NET Framework has a class library, which provides a very large collection of over 2500 reusable types (classes) available to any .NET language.

Dynamic Web pages and Web services: .NET was written with the Internet in mind; thus, deploying applications over intranets or the Internet becomes an ordinary, day-to-day operation. Using a new technology, ASP.NET, a set of components called ADO.NET, and having XML support makes it easier to access relational databases and other data sources as well as to import and export data through the Web.

Scalable component development: .NET not only facilitates object-oriented development, but it takes design one step further. With .NET, true component-based development can be performed better and easier than in the past. Segments of code, created as separate entities, can be stored independently, combined, and reused in many applications. That is the beauty behind .NET—it provides a relatively seamless way for client programs to use components built using different languages. With .NET, component-based development can become a reality.

Why C#?

Compilers targeting the .NET platform are available for the programming languages of Visual Basic, C++, and C#. In addition to Microsoft, a number of third-party vendors are also marketing .NET-compliant languages. Microsoft also introduced a new programming language called Visual F# with Visual Studio 2010. There are also a number of other programming languages, which target the .NET platform. So, why use C#? C# was *the* language created for .NET and was designed from scratch to work with .NET. A large number of classes were included as part of .NET. These classes or templates were designed for reuse by any of the .NET-supported languages. They reduce the amount of programming that needs to be done. These classes are called the .NET Framework classes. Most of the .NET Framework classes were written using the C# programming language.

C#, in conjunction with the .NET Framework classes, offers an exciting vehicle to incorporate and use emerging Web standards, such as Hypertext Markup Language (HTML), Extensible Markup Language (XML), and Simple Object Access Protocol (SOAP). As stated earlier, C# was designed with the Internet in mind. Most of the programming tools that were developed before .NET were designed when the Web was in its infancy and, thus, are not the greatest fit for developing Windows and Web applications.

C# is a simple, object-oriented language. Through using the Visual Studio IDE and the .NET Framework, C# provides an easy way to create graphical user interfaces similar to those Visual Basic programmers have been using for years. C# also provides the pure data-crunching horsepower to which C and C++ programmers are accustomed. All of the looping and selection programming constructs are included. C# enables developers to make their packages available over the Internet using many common features previously found only in Java, Common Gateway Interface (CGI), or PERL.

NOTE

Some characterize C# by saying that Microsoft took the best of Visual Basic and added C++, trimming off some of the more arcane C traditions. The syntax is very close to Java.

Many attribute the success of C++ to its American National Standards Institute (ANSI) standardization. C# is following a similar path toward success. On December 13, 2001, the European Computer Manufacturers Association (ECMA) General Assembly ratified C# and its common language infrastructure (CLI) specifications into international standards. This verification proves that C# is not just a fly-by-night language. Originally, development with C# was possible only for the Windows platform; C# is now available for other platforms including Linux. C# is going to be around for some time. It represents the next generation of languages.

As you read in previous sections, programs are the instructions written to direct a computer to perform a particular task. Each instruction must be written in a specific way. The **syntax** rules for writing these instructions is defined by the language in which the program is written. Before results are obtained, these human-readable instructions, called **source code**, must be translated into the machine language, which is the **native code** of the computer. The translation is a two-step process, which begins with the compiler. In this chapter, you will write your first C# program, learn how it is compiled, and explore how the final output is produced.

Each instruction statement has a **semantic meaning**, a specific way in which it should be used. This chapter highlights the purpose of the program statements as they appear in an application, because many of these program elements will be used in all applications that you develop using C#. You'll start by investigating the types of applications that can be developed using C# and the .NET platform.

Types of Applications Developed with C#

C# can be used to create several different types of software applications. Some of the most common applications are:

- Web applications
- Windows graphical user interface (GUI) applications
- Console-based applications

In addition to these applications, class libraries and stand-alone components (.dlls), smart device applications or apps, and services can also be created using C#.

Web Applications

C# was built from the ground up with the Internet in mind. For this reason, programmers can use C# to quickly build applications that run on the Web for end users to view through browser-neutral user interfaces (UIs). As they program in C#, developers can ignore the unique requirements of the platforms—Macintosh, Windows, and Linux—that

will be executing these applications and end users will still see consistent results. Using **Web forms**, which are part of the ASP.NET technology, programmable Web pages can be built that serve as a user interface (UI) for Web applications. **ASP.NET** is a programming framework that lets you create applications that run on a Web server and delivers functionality through a browser, such as Microsoft Internet Explorer. Although you can use other languages to create ASP.NET applications, C# takes advantage of the .NET Framework and is generally acknowledged to be the language of choice for ASP.NET development. Much of the .NET Framework class library (FCL) is written in the C# programming language. After you learn some problem-solving techniques and the basic features of the C# language, Chapter 15 introduces you to ASP.NET. Figure 1-12 illustrates an ASP.NET Web page you will create with C# in Chapter 15.

Computer Club Inquiry Form

First Name: Alma

Last Name: King

Phone Number: 606-549-2639

Student I.D.: 1122132

Classification:

- ☐ Freshman/Sophomore
- ☒ Junior/Senior
- ☐ Other

Special Interests:

- Programming Contest
- Social Gatherings
- Ask a Techie

Junior & Seniors Always Welcome!

Current Members:

First Name	Last name
Ralph	Abbott
Colleen	Bennett
Linda	Bishop
Brenda	Bowers
Gary	Jones
Rachel	Smith
James	Tuttle
Sara	Winston

Click below to see when the club meets next.

Next Meeting

Meeting next week: Tuesday, 9/3 at 8 P.M.

Thanks Alma! You will be contacted... to discuss joining the "Social Gatherings" team.

FIGURE 1-12 Web application written using C#

Windows Applications

Windows applications are designed for desktop use and for a single platform. They run on PC desktops much like your favorite word-processing program. Writing code using C# and classes from the `System.Windows.Forms` namespace, applications can be developed to include menus, pictures, drop-down controls, and other widgets you have come to expect in a modern desktop application. .NET uses the concept of **namespace**

to group types of similar functionality. The `System.Windows.Forms` namespace is used as an umbrella to organize classes needed to create Windows applications. Using the integrated development environment (IDE) of Visual Studio, GUIs can be developed by dragging and dropping controls such as buttons, text boxes, and labels on the screen. This same drag-and-drop approach is used to create Web applications with Visual Studio. You will begin creating Windows GUI applications in Chapter 9. Figure 1-13 illustrates a Windows application you will create using the C# language in Chapter 10.

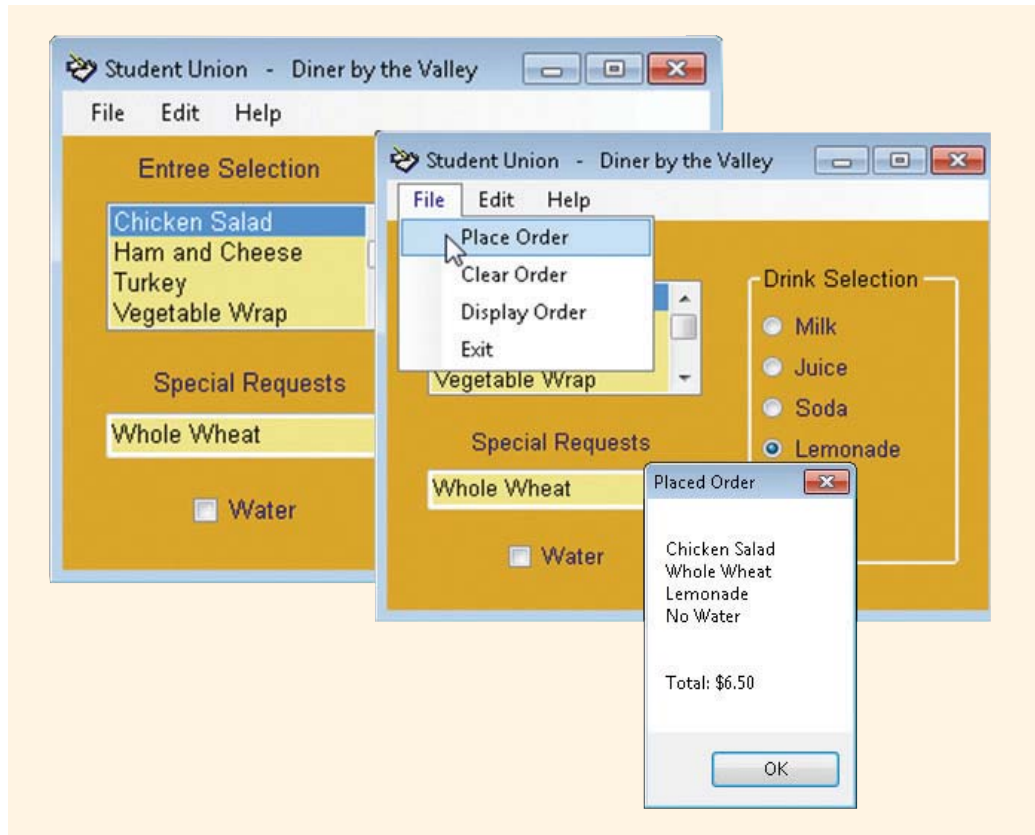


FIGURE 1-13 Windows application written using C#

Console Applications

Console applications normally send requests to the operating system to display text on the command console display or to retrieve data from the keyboard. From a beginners' standpoint, console applications are the easiest to create and represent the simplest approach to learning software development, because you do not have to be concerned with the side issues associated with building GUIs. Values can be entered as input with

minimal overhead, and output is displayed in a console window, as illustrated in Figure 1-14. You will begin by developing console applications, so that you can focus on the details of programming and problem solving in general.

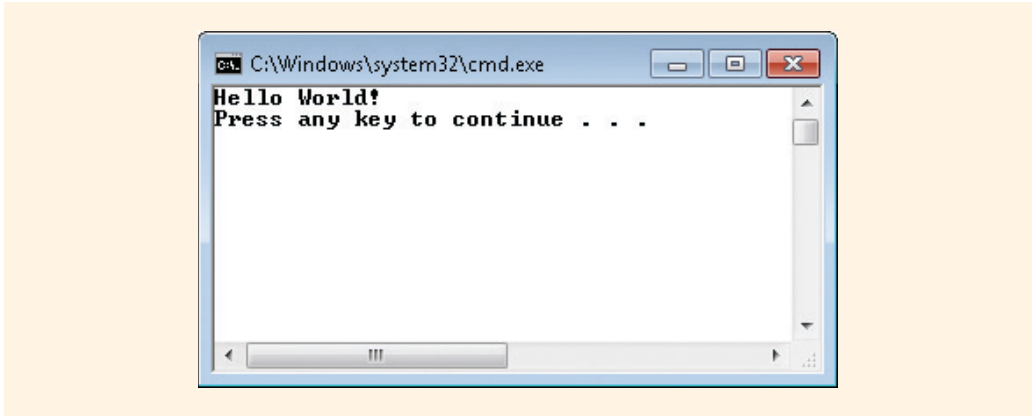


FIGURE 1-14 Output from Example 1-1 console application

Exploring the First C# Program


Since the 1970s when the C language was developed, it has been traditional when learning a new language to display “Hello World!” on the console screen as the result of your first program. The program in Example 1-1 demonstrates a C# program that does exactly that. The sections that follow explain line-by-line the elements that make up this first program.

EXAMPLE 1.1

```

Line 1  // This is traditionally the first program written.
Line 2  using System;
Line 3  namespace HelloWorldProgram
Line 4  {
Line 5      class HelloWorld
Line 6      {
Line 7          static void Main( )
Line 8          {
Line 9              Console.WriteLine("Hello World!");
Line 10         }
Line 11     }
Line 12 }
```

Readability is important. As far as the compiler is concerned, you could actually type the entire program without touching the Enter key. The entire program could be typed as a single line, but it would be very difficult to read and even more challenging to debug. The style in Example 1-1 is a good one to mimic. Notice that curly braces { } are matched and appear on separate lines, by themselves. Statements are grouped together and indented. Indenting is not required, but is a good practice to follow because it makes the code easier to read. When you type your program, you should follow a consistent, similar style. The output produced from compiling and executing the program appears in Figure 1-14.

NOTE  Formatting code is much simpler with Visual Studio. You can preset your text editor preferences from the **Tools, Options** menu. You can also purchase plug-ins, such as ReSharper, which do intelligent code editing and formatting for you as you type.

Elements of a C# Program

Although the program statements in Example 1-1 make up one of the smallest functional programs that can be developed using C#, they include all the major components found in most programs. An understanding of the features that make up this program will help prepare you to begin developing your own applications. Examine each segment line-by-line to analyze the program, beginning with Line 1.

Comments

The first line of Example 1-1 is a comment:

Line 1 `// This is traditionally the first program written.`

NOTE  Comments are displayed in **green** throughout the book.

Writing a comment is like making notes for yourself or for readers of your program. Comments are not considered instructions to the computer and, therefore, have no effect on the running of the program. When the program is compiled, comments are not checked for rule violations; on the contrary, the compiler ignores and bypasses them. Comments do not have to follow any particular rules, with the exception of how they begin and end.

Comments serve two functions: they make the code more readable and they internally document what the program statements are doing. At the beginning of a program, comments are often written to identify who developed the code, when it was developed, and for what purpose the instructions were written. Comments are also used

to document the purpose of different sections of code. You can place comments anywhere in the program. Where a portion of the code might be difficult to follow, it is appropriate to place one or more comments explaining the details. In Example 1-1, the only comment is found on Line 1.

Line 1 `// This is traditionally the first program written.`

NOTE

C# programmers do not normally use line numbers, but they are added here to explain the features.

With C#, three types of commenting syntax can be added to a program: inline, multiline, and XML document comments.

Inline Comments

The comment that appears on Line 1 of Example 1-1 is an inline, or single-line, comment. An **inline comment** is indicated by two forward slashes `//` and is usually considered a one-line comment. The slashes indicate that everything to the right of the slashes, on the same line, is a comment and should be ignored by the compiler. No special symbols are needed to end the comment. The carriage return (Enter) ends the comment.

Multiline Comments

For longer comments, multiline comments are available. A forward slash followed by an asterisk `/*` marks the beginning of a **multiline comment**, and the opposite pattern `*/` marks the end. Everything that appears between the comment symbols is treated as a comment. Multiline comments are also called **block comments**. Although they are called multiline comments, they do not have to span more than one line. Unlike the single-line comment that terminates at the end of the current line when the Enter key is pressed, the multiline comment requires special characters `/*` and `*/` to begin and terminate the comment, even if the comment just appears on a single line. Example 1-2 shows an example of a multiline comment.

EXAMPLE 1.2

`/* This is the beginning of a multiline (block) comment. It can go on for several lines or just be on a single line. No additional symbols are needed after the beginning two characters. Notice there is no space placed between the two characters. To end the comment, use the following symbols. */`

NOTE

C# does not allow you to nest multiline comments. In other words, you cannot place a block comment inside another block comment. The outermost comment is ended at the same time as the inner comment as soon as the first `*/` is typed.

XML Documentation Comments

A third type of comment uses three forward slashes `///`. This is an advanced documentation technique used for XML-style comments. **XML (Extensible Markup Language)** is a markup language that provides a format for describing data using tags similar to HTML tags. A C# compiler reads the **XML documentation comments** and generates XML documentation from them.

You will be using the inline `//` and multiline `/* */` comments for the applications you will develop in this book.

Using Directive

The following statement that appears in Example 1-1, Line 2 permits the use of classes found in the `System namespace` without having to qualify them with the word “System.” This reduces the amount of typing that would be necessary without the directive.

Line2 `using System;`

Using .NET provides the significant benefit of making available more than 2000 classes that make up what is called the Framework class library (FCL). A `class` contains code that can be reused, which makes programming easier and faster because you don’t have to reinvent the wheel for common types of operations. The Framework classes are all available for use by any of the .NET-supported languages, including C#.

NOTE

Keywords after they are introduced, such as `class`, are displayed in blue throughout this book. **Keywords** are words reserved by the system and have special, predefined meanings. The keyword `class`, for instance, defines or references a C# `class`.

With several thousand .NET Framework classes, it is very unlikely that program developers will memorize the names of all of the classes, let alone the additional names of members of the classes. As you could well imagine, it would be easy for you to use a name in your own application that has already been used in one or more of the Framework classes. Another likely occurrence is that one or more of these Framework classes could use the same name. How would you know which `class` was being referenced? .NET eliminates this potential problem by using namespaces.

EXAMPLE 1.3**1**

Assume you have a **class** called **Captain**. You could abstract out characteristics and behavior attributes of a Captain. If you had an application that was being developed for a football team, the characteristics and behaviors of that kind of Captain would differ greatly from those of the Captain in an application designed for a fire department. Moreover, both sets of characteristics differ from those of a boating Captain. The Captain associated with a military unit, such as the Navy, would also be different. With each possible application, you could use the same name for the Captain **class**. If you gave instructions to a program to display information about a Captain, the program would not know which set of characteristics to display. To clarify whether you are talking about the boat Captain, football team Captain, fire station Captain, or the Navy Captain, you could qualify the keyword by preceding the name with its category. To do so, you could write **Boat.Captain**, **Football.Captain**, **Fireman.Captain**, or **Navy.Captain**. You would need to precede Captain with its category type every time reference was made to Captain.

By specifying which **namespace** you are building an application around, you can eliminate the requirement of preceding Captain with a dot (.) and the name of the **namespace**. If you simply “use the boating **namespace**,” you do not have to qualify each statement with the prefix name. That is what the **using** directive accomplishes. It keeps you from having to qualify each **class** by identifying within which **namespace** something appears. A **using** directive enables unqualified use of the types that are members of the **namespace**. By typing the **using-namespace-directive**, all the types contained in the given **namespace** are imported, or available for use within the particular application.

The most important and frequently used **namespace** is **System**. The **System namespace** contains classes that define commonly used types or classes. The **Console class** is defined within the **System namespace**. The **Console class** enables programmers to write to and read from the console window or keyboard. The fully qualified name for **Console** is **System.Console**. If you remove the **using System;** directive in Line 2, it would be necessary for you to replace

Line 9 `Console.WriteLine("Hello World!");`

with

Line 9 `System.Console.WriteLine("Hello World!");`

NOTE

In addition to including the **using** directive, the **System namespace** must be included in the references for a project created using Visual Studio. When a console application is created, Visual Studio automatically references and imports the **System namespace**.

Namespaces provide scope, or an enclosing context, for the names defined within the group. By including the **using** directive to indicate the name of the **namespace** to be used, you can avoid having to precede the **class** names with the category grouping. After you add the **using System;** line, you can use the **Console** **class** name without explicitly identifying that it is in the **System** **namespace**.

Namespace

Lines 3 through 12 define a **namespace** called **HelloWorldProgram**.

```
Line 3    namespace HelloWorldProgram
Line 4    {
Line 12   }
```

The **namespace** does little more than group semantically related types under a single umbrella. Example 1-1 illustrates how a **namespace** called **HelloWorldProgram** is created. **HelloWorldProgram** is an **identifier**, simply a user-supplied or user-created name. As noted in the previous section, you will create many names (identifiers) when programming in C#. Rules for creating identifiers are discussed in Chapter 2. You can define your own **namespace** and indicate that these are names associated with your particular application.

Each **namespace** must be enclosed in curly braces { }. The opening curly brace ({) on Line 4 marks the beginning of the **HelloWorldProgram** **namespace**. The opening curly brace is matched by a closing curly brace (}) at the end of the program on Line 12. Within the curly braces, you write the programming constructs.

In the application in Example 1-1, Lines 3, 4, and 12 could have been omitted. This program did not define any new programming constructs or names. It is merely using the **Console** **class**, which is part of the **System** **namespace**. No errors are introduced by adding the additional umbrella, but it was not necessary. Visual Studio automatically adds a **namespace** umbrella around applications that are created using the IDE.

Class Definition

Lines 5 through 11 make up the **class** definition for this application.

```
Line 5    class HelloWorld
Line 6    {
Line 11   }
```

As C# is an object-oriented language, everything in C# is designed around a **class**, which is the building block of an object-oriented program. C# doesn't allow anything to be defined outside of a **class**. Every program must have at least one **class**. Classes define a category, or type, of object. Many classes are included with the .NET

Framework. Programs written using C# can use these predefined .NET classes or create their own classes.

In Example 1-1, the user-defined **class** is titled **HelloWorld**. The example also uses the **Console class**, one of the .NET predefined classes. Every **class** is named. It is traditional to name the file containing the **class** the same name as the **class** name, except the filename will have a .cs extension affixed to the end of the name. C# allows a single file to have more than one **class**; however, it is common practice to place one user-defined **class** per file for object-oriented development.

NOTE Most object-oriented languages, including Java, restrict a file to one **class**. C#, however, allows multiple classes to be stored in a single file.

Classes are used to define controls such as buttons and labels, as well as types of things such as Student, Captain, and Employee. The word **class** is a keyword. Like namespaces, each **class** definition must be enclosed in curly braces { }. The { on Line 6 is an opening curly brace, which marks the beginning of the **class** definition. The opening curly brace is matched by a closing curly brace at the end of the **class** definition on Line 11. Within the curly braces, you define the **class** members. A **class** member is generally either a member method, which performs some behavior of the **class**, or a data member, which contains a value associated with the state of the **class**.

Main() Method

The definition for the **Main()** method begins on Line 7 and ends with the closing curly brace on Line 10.

```
Line 7      static void Main( )
Line 8      {
Line 10     }
```


The **Main()** method plays a very important role in C#. This is the entry point for all applications. This is where the program begins execution. The **Main()** method can be placed anywhere inside the **class** definition. It begins on Line 7 for this example. When a C# program is launched, the execution starts with the first executable statement found in the **Main()** method and continues to the end of that method. If the application is 3000 lines long with the **Main()** method beginning on Line 2550, the first statement executed for the application is on Line 2550.

NOTE All executable applications must contain a **Main()** method.

The entire contents of Line 7 is the heading for the method. A **method** is a collection of one or more statements combined to perform an action. Methods are similar to C++ functions. The heading for the method contains its **signature**, which includes the name of the method and its argument list. Return types and modifiers, such as **public** and **static**, are part of the heading, but not considered part of the signature. The heading line for the **Main()** method begins with the keyword **static**, which implies that a single copy of the method is created, and that you can access this method without having an object of the **class** available. More details regarding **static** are discussed in subsequent sections. For now, remember that **Main()** should include the **static** keyword as part of its heading.

The second keyword in the heading is **void**. Void is the return type. Typically, a method calls another method and can return a value to the calling method. Remember that a method is a small block of code that performs an action. As a result of this action, a value might be returned. If a method does not return a value, the **void** keyword is included to signal that no value is returned. When the method does return a value, the type of value is included as part of the heading. Chapter 3 introduces you to the different data types in C#.

Main() is the name of the method. Methods communicate with each other by sending arguments inside parentheses or as return values. Sometimes no argument is sent, as is the case when nothing appears inside the parentheses.

NOTE Unlike the lowercase `main()` method that appears in the C++ language,  in C#, `Main()` must begin with an uppercase 'M'.

In Example 1-1, only one executable statement is included in the body of the method **Main()**. The body includes all items enclosed inside opening and closing curly braces. When a program is executed, the statements that appear in the **Main()** method are executed in sequential order. When the closing curly brace is encountered, the entire program ends.

Method Body Statements

The body of this **Main()** method consists of a single, one-line statement found on Line 9.

```
Line 8      {
Line 9      Console.WriteLine("Hello World!");
Line 10     }
```

Remember that the purpose of the program in Example 1-1 is to display **"Hello World!"** on the output screen. The lines of code in Example 1-1, which have been explained on previous pages of this chapter, are common to most applications you will be developing. Line 9, however, is unique to this application. The body for this method begins on Line 8 and ends on Line 10.

The statement in the `Main()` method is a call to another method named `WriteLine()`. A method call is the same as a **method invocation**. Like `Main()`, `WriteLine()` has a signature. The heading along with the complete body of the method is the **definition of the method**. When called, `WriteLine()` writes the string argument that appears inside the parentheses to the standard output device, a monitor. After displaying the string, `WriteLine()` advances to the next line, as if the Enter key had been pressed.

NOTE

A quick way to identify a method is by looking for parentheses; methods always appear with parentheses (). A call to any method, such as `WriteLine()`, always includes a set of parentheses following the method name identifier, as do signatures for methods.

The string of text, "Hello World!", placed inside the parentheses is the method's argument. `WriteLine()` is defined in the `Console` class and can be called with no arguments. To have a blank line displayed on the standard output device, type:

```
Console.WriteLine( );    // No string argument is placed inside ( )
```

The `Console` class contains the standard input and output methods for console applications. Methods in this class include `Read()`, `ReadKey()`, `ReadLine()`, `Write()`, and `WriteLine()`. The method `Write()` differs from `WriteLine()` in that it does not automatically advance the carriage return to the next line when it finishes. The following lines would produce the same result as the single line `Console.WriteLine("Hello World!");`

```
Console.Write("Hello");
Console.Write(" ");
Console.WriteLine("World!");
```

An invisible pointer moves across the output screen as the characters are displayed. As new characters are displayed, it moves to the next position and is ready to print at that location if another output statement is sent. Notice the second statement in the preceding code, `Console.Write(" ");` this places a blank character between the two words. After displaying the space, the pointer is positioned and ready to display the *W* in *World*. The output for both of the preceding segments of code is:

```
Hello World!
```

Usually, the characters inside the double quotes are displayed exactly as they appear when used as an argument to `Write()` or `WriteLine()`. An exception occurs when an escape character is included. The backslash (`'\'`) is called the **escape character**. The escape character is combined with one or more characters to create a special **escape sequence**, such as `'\n'` to represent advance to next line, and `'\t'` for a tab indentation. A number of escape sequences can be used in C#. This is the same set of escape characters found in other languages such as Java and C++. Table 1-1 lists some of the more commonly used escape characters that can be included as part of a string in C#.

TABLE 1-1 Escape sequences

Escape sequence character	Description
\n	Cursor advances to the next line; similar to pressing the Enter key
\t	Cursor advances to the next horizontal tab stop
\"	Double quote is printed
\'	Single quote is printed
\\	Backslash is printed
\r	Cursor advances to the beginning of the current line
\a	Alert signal (short beep) is sounded

© 2013 Cengage Learning

When an escape sequence is encountered inside the double quotes, it signals that a special character is to be produced as output. The output of the statement:

`Console.Write("What goes\nup\nmust come\tdown.");`
is

What goes
up
must come down.

Notice in the `Write()` method that the argument inside the parentheses has three escape sequences. The backslash is not printed. When the `'\n'` is encountered, the output is advanced to the new line. The space between the words “come” and “down” was placed there as a result of the tab escape sequence (`'\t'`).

Three other methods in the `Console class`, `Read()`, `ReadKey()`, and `ReadLine()`, deserve explanation. Visually they differ from the `WriteLine()` method in that they have no arguments, nothing is placed inside the parentheses. `Read()`, `ReadKey()`, and `ReadLine()` methods can all return values and are used for accepting input from a standard input device, such as a keyboard.

NOTE

Notice in Example 1-1 that the statements in the body of methods end in semicolons. But, no semicolon is placed at the end of method headings, **class** definition headings, or **namespace** definition headings. Note that semicolons appear on Lines 2 and 9 in Example 1-1.

ReadKey () is often used in a C# program to keep the output screen displayed until the user presses a key on the keyboard. The **Read ()** method can also be used and like the **ReadKey ()** method accepts any character from the input device, and as soon as a key is pressed, control passes to the next line in the program. Instead of accepting a single character as the **Read ()** and **ReadKey ()** methods do, **ReadLine ()** allows multiple characters to be entered. It accepts characters until the Enter key is pressed. You will use the **ReadLine ()** method in Chapter 4 to input data values.

Now that you know what elements make up a C# program, you are almost ready to begin developing your own programs. Figure 1-15 shows how **namespace**, **class**, method, and statements are related. Notice that the Framework class library (FCL) includes a number of different namespaces, such as **System**. Defined within a **namespace** are a

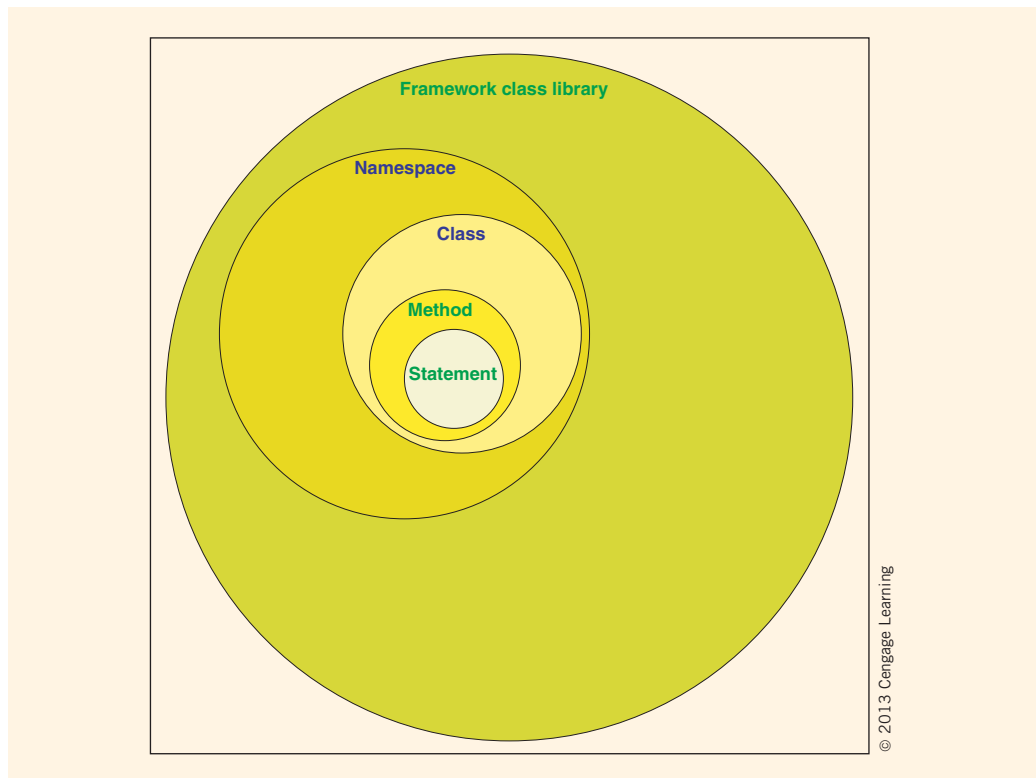


FIGURE 1-15 Relationship among C# elements

number of classes, such as `Console`. Defined within a `class` are a number of methods, such as `WriteLine()` and `Read()`. You did not see the statements that made up the `WriteLine()` method, you saw the `WriteLine()` method being called. A method can have one or more statements. One of those statements can be a call to another method, such as the case of calling on the `WriteLine()` method from within the `Main()` method.

To develop software using C#, the .NET Framework is needed. The Framework includes all the predefined .NET classes. Having so many classes available decreases the amount of new code needed. You can use much of the functionality included in the `class` library. The next section describes how you begin typing your program statements.

Compiling, Building, and Running an Application

Typing Your Program Statements

You have a couple of options for writing code using C#. One approach to developing applications is to type the source code statements using a simple text editor (such as Notepad) and follow that up by using the DOS command line to compile and execute the application. This technique offers the advantage of not requiring significant system resources, but requires that you must know exactly what to type.

A second approach is to use the Visual Studio integrated development environment (IDE), and this chapter introduces you to Visual Studio. The IDE is an interactive environment that enables you to type the source code, compile, and execute without leaving the IDE program. Because of the debugging and editing features that are part of the IDE, you will find it much easier to develop applications if Visual Studio is available. In addition to the rich, integrated development environment, Visual Studio includes a number of tools and wizards. Appendix A, “Visual Studio Configuration,” describes many additional features of Visual Studio, including suggestions for customizing the IDE.

NOTE

You will want to deselect **Hide file extensions for known file types** so that as you create applications, you will be able to see the file extensions in the **Solution Explorer** Window. The file extension includes a dot and two to six characters following the name. Examples are `.cs`, `.csproj`, `.sys`, `.doc`, `.suo`, `.sln`, and `.exe`. The extension identifies the type of information stored in the file. For example, a file ending in `.cs` is a C# source file. It is helpful to be able to identify files by type.

The preceding sections described the program statements required as a minimum in most console-based applications. To see the results of a program, you must type the statements, or source code, into a file, compile that code, and then execute the application. The next sections examine what happens during the compilation and execution process with and without using the Visual Studio IDE.

Compilation and Execution Process

The **compiler** is used to check the grammar. The grammar is the symbols or words used to write the computer instructions. The compiler makes sure there are no rule violations in the program statements or source code. After the code is successfully compiled, the compiler usually generates a file that ends with an .exe extension. The code in this .exe file has not yet been turned into machine code that is targeted to any specific CPU platform. Instead, the code generated by the compiler is **Microsoft Intermediate Language** (MSIL), often referred to simply as **IL**. In the second required step, the **just-in-time** compiler (**JITer**) reads the IL code and translates or produces the machine code that runs on the particular platform. After the code is translated in this second step, results can be seen.

Operations going on behind the scene are not readily apparent. For example, after the compiler creates the IL, the code can be executed on any machine that has the .NET Framework installed. Microsoft offers as a free distribution the .NET Framework Redistributable version for deploying applications only. The **Redistributable version** is a smaller download than the SDK and includes the CLR and **class** libraries. Again, this is available at the Microsoft Web site.

The runtime version of the .NET Framework is similar in concept to the Java Virtual Machine (JVM). Like C#, Java's compiler first translates source code into intermediate code called bytecode. The bytecode must be converted to native machine code before results can be seen from an application. With C#, the CLR actually translates only the parts of the program that are being used. This saves time. In addition, after a portion of your IL file has been compiled on a computer, it never needs to be compiled again because the final compiled portion of the program is saved and used the next time that portion of the program is executed.

Compiling the Source Code Using Visual Studio IDE

You can use the built-in editor available with the Visual Studio IDE to type your program statement. You then compile the source code from one of the pull-down menu options in the IDE and execute the application using another menu option in the IDE. Many shortcuts are available. The next section explores how this is done using the Visual Studio IDE.

Begin by opening Visual Studio. Create a new project by either selecting the **New Project** button on the Start page or by using the **File, New, Project** option.

As shown in Figure 1-16, a list of project types appears in the middle window. There are a number of templates that can be used within the IDE. To develop a C# console application, select **Visual C#** and **Console Application** for the Template. Using the **Browse** button beside the Location text box, navigate to the location where you want to store your projects. The name of the project is **HelloWorldProgram**. You can remove the check mark, if it is present, from the check box beside the **Create directory for**

solution option. Having that option selected creates another directory layer. This extra folder is not necessary for the types of applications you will be creating.

NOTE Whatever name you give the project becomes the namespace's name for this project unless this default setting is changed.

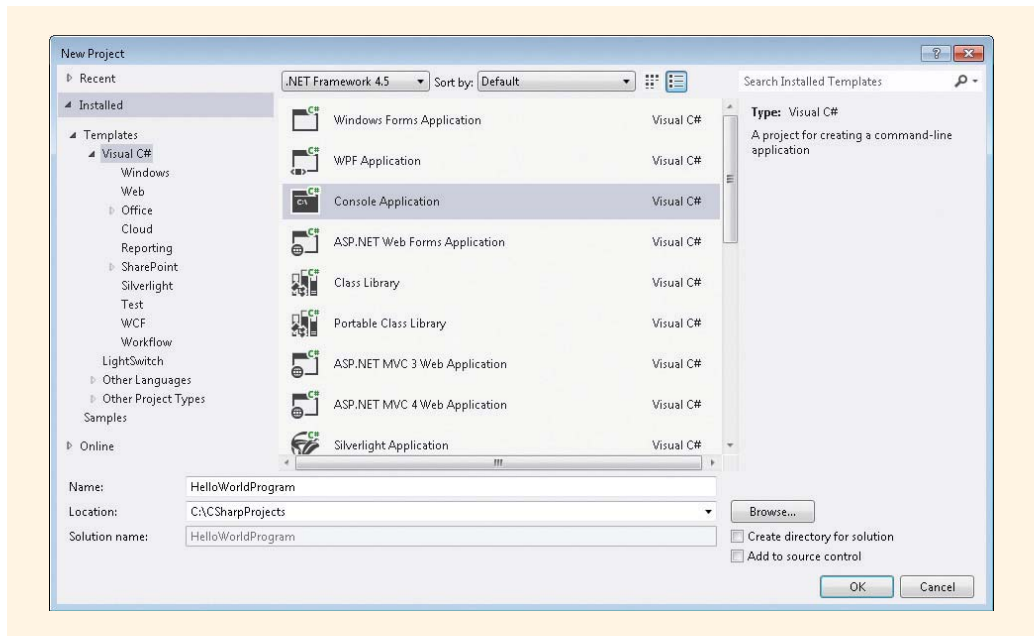


FIGURE 1-16 Creating a console application

Selecting the template determines what type of code is generated by the IDE. Figure 1-17 shows the code that is created automatically when you create a console application.

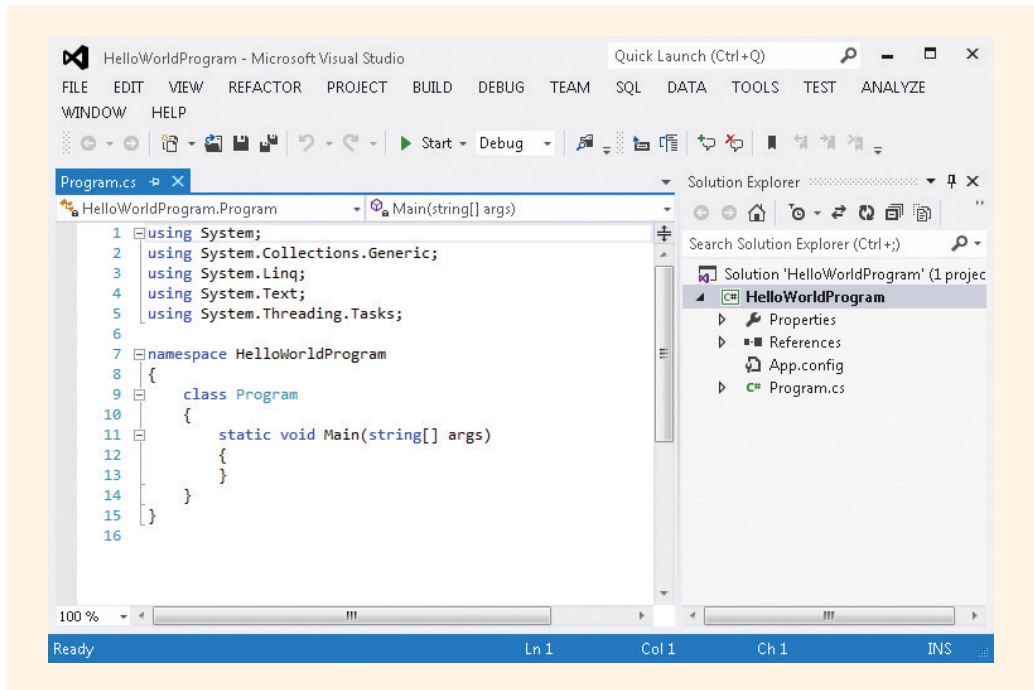


FIGURE 1-17 Code automatically generated by Visual Studio

As you can see from Figure 1-17, having the IDE generate this much code reduces your typing. The only lines that must be added are those specific to the application being developed. To produce the traditional Hello World program, begin by moving the cursor to the end of Line 11 and pressing the Enter key to open up a new blank line. Type the following line between the braces in the `Main()` method:

```
Console.WriteLine("Hello World!");
```

NOTE Notice that as soon as you type the character `C`, a smart window listing pops up with `Console` already selected. This is the **Word Correct** option of the **IntelliSense** feature of the IDE. As the name implies, the feature attempts to sense what you are going to type before you type it. When the window pops up, if it has the correct selection, simply type a period and you will get another IntelliSense pop-up. Again, if it has the correct selection (`WriteLine`), simply type the next separator, which is the left parenthesis. You can also use the arrow keys to select from the list.

Next, change the name of the `class` and the source code filename. Visual Studio names the `class` `Program`, and by default identifies the source code file by that same name.

If you use the **Solution Explorer** window to change the source code filename to `HelloWorld.cs`, a message, as shown in Figure 1-18, will be displayed asking whether you want to change all references to that new name. You can make this change in the **Solution Explorer** window by either right-clicking on the name in the **Solution Explorer** window and selecting the **Rename** option, as is shown in Figure 1-18, or by simply clicking on the name. If the **Solution Explorer** window is not active on your desktop, select **View, Solution Explorer**. Be sure to include the `.cs` file extension when you rename the file.

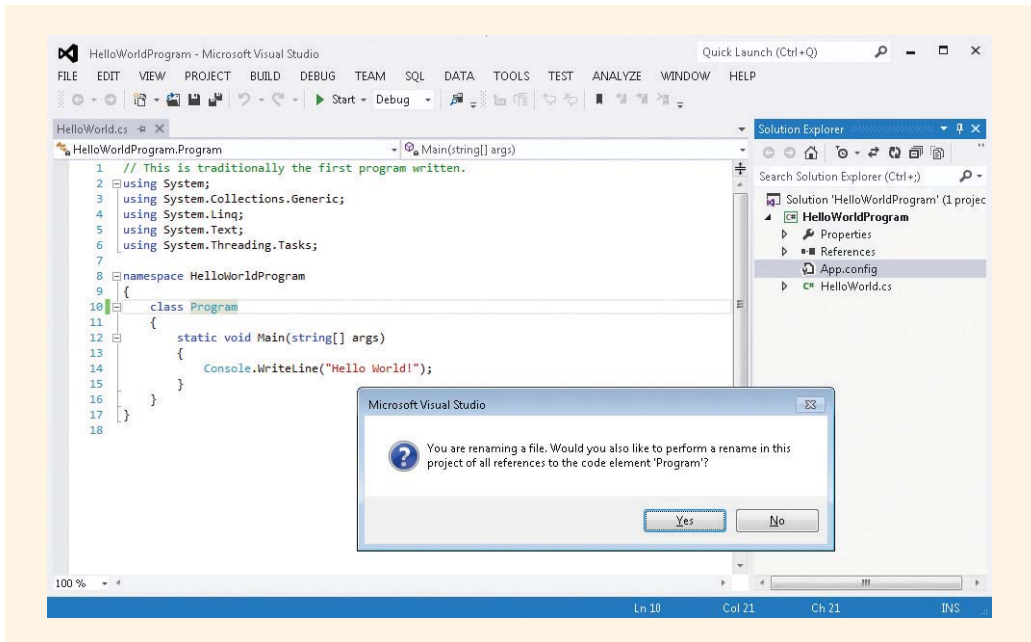


FIGURE 1-18 Changing the source code name from `Program`

As you review the **Solution Explorer** window shown in Figure 1-18, notice the top level reads “**Solution ‘HelloWorldProgram’ (1 project).**” The second line also contains the word “**HelloWorldProgram.**” Visual Studio first creates a solution file. This is the file, `HelloWorldProgram.sln`, which appears on that top line. The solution file may consist of one or more projects. For this application, the solution consists of one project. When you explore the directory where your applications are stored, you will find a folder named using that solution name. Inside the folder there will be a file ending with a `.sln` extension. This is the solution file and it stores information about the solution. In that same folder, you will also see a file ending with a `.csproj` extension. This file stores information about the project. Normally when you reopen your application in Visual Studio, you will open the file ending with the `.sln` extension.

If you answer **Yes** to the question shown in Figure 1-18, the name of the **class** in the source code is replaced with the new name.

NOTE



It is not absolutely necessary to change the names of the source code file and **class**. The application can run without making that change; however, to develop good habits you should change the name. It can save you time and grief when applications involve multiple classes.

The statements in Example 1-4 appeared in Example 1-1 and are repeated here without the line numbers so that you can see the final source listing.

NOTE

Visual Studio generates a couple of other unnecessary lines that can be removed. For example, the extraneous **using** statements were removed. They are not needed for most of the applications you will develop. The arguments inside the parentheses for the `Main()` method were also removed. You will read about these lines later in this book.

EXAMPLE 1.4

```
// This is traditionally the first program written.
using System;
namespace HelloWorldProgram
{
    class HelloWorld
    {
        static void Main( )
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

NOTE



Do remember that C# is case sensitive, meaning the name `HelloWorld` is a totally different name from `helloWorld`, even though only one character is different. So be very careful to type the statements exactly as they are shown.

To compile the `HelloWorldProgram` project, select the **Build HelloWorldProgram** option on the **Build** menu. The name `HelloWorldProgram` follows the **Build** option because `HelloWorldProgram` is the name of the project. Projects that contain more than one `class` are compiled using the **Build Solution** option.

To run the application, you can click **Start Debugging** or **Start Without Debugging** on the **Debug** menu bar, as illustrated in Figure 1-19. If you attempt to execute code that has not been compiled (using **Build**), the smart IDE compiles the code first. Therefore, many developers use the shortcut of bypassing the **Build** step.

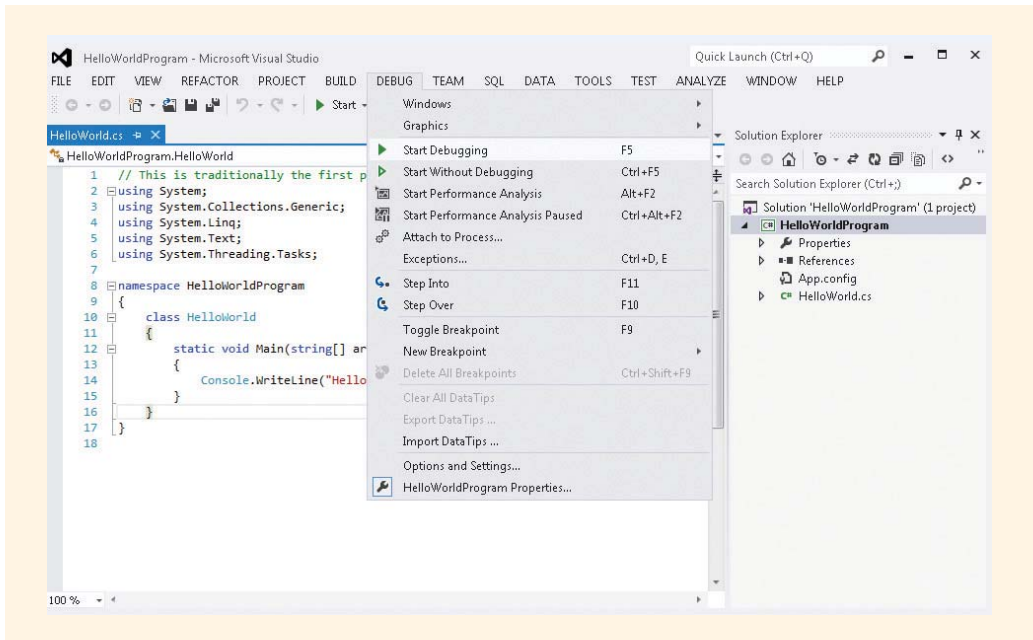


FIGURE 1-19 Execution of an application using Visual Studio

When you ran the application using the **Start Debugging** option, you probably noticed that the output flashes on the screen and then disappears. You can hold the output screen if you include a call to the `Read()` or `ReadKey()` methods. When this line is executed, the program waits for you to press a key before it terminates. To add this, go back into the source code and place this line as the last statement in the `Main()` method:

```
Console.ReadKey();
```

You read earlier that the software development cycle is iterative, and it is sometimes necessary to return to previous phases. Here, it is necessary to return to your source code statements and type the additional statement. You then need to have the code recompiled. When there are no more syntax errors, you will see the output from your program.

Another option to hold the command windows in Visual Studio is to select **Debug, Start Without Debugging** instead of **Debug, Start Debugging** to execute your program. Notice in Figure 1-19 that **Start Without Debugging** is the option immediately below the **Start Debugging** option. If you select **Debug, Start Without Debugging**, it is not necessary to add the additional `Console.Read()` or `ReadKey() ;` statements. When you use the **Start Without Debugging** option for execution, the user is prompted to “Press any key to continue,” as was illustrated earlier in Figure 1-14. This is the preferred method for executing your program.

NOTE



Several other shortcuts are available to run your program. Notice as you look at the menu option under **Debug** that `Ctrl+F5` is listed as a shortcut for **Start Without Debugging**; `F5` is the shortcut for **Start Debugging**. In addition, if you have the **Debug** Toolbars icons on your screen, an open right arrow represents **Start Without Debugging**.

Visual Studio is a highly sophisticated integrated development environment. Appendix A, “Visual Studio Configuration,” includes additional information regarding customizing the development environment and using the debugger for development. Reviewing that material now would be wise.

Debugging an Application

It is inevitable that your programs will not always work properly immediately. Several types of errors can occur and cause you frustration. The major categories of errors—syntax and runtime—are discussed in the next sections.

Syntax Errors

When you type source code statements into the editor, you are apt to make typing errors by misspelling a name or forgetting to end a statement with a semicolon. These types of errors are categorized as **syntax errors** and are caught by the compiler. When you compile, the compiler checks the segment of code to see if you have violated any of the rules of the language. If it cannot recognize a statement, it issues an error message, which is sometimes cryptic, but should help you fix the problem. Error messages in Visual Studio are more descriptive than those issued at the command line. But, be aware that a single typing error can generate several error messages. Figure 1-20 shows a segment of code, in which a single error causes the compiler to generate three error messages.

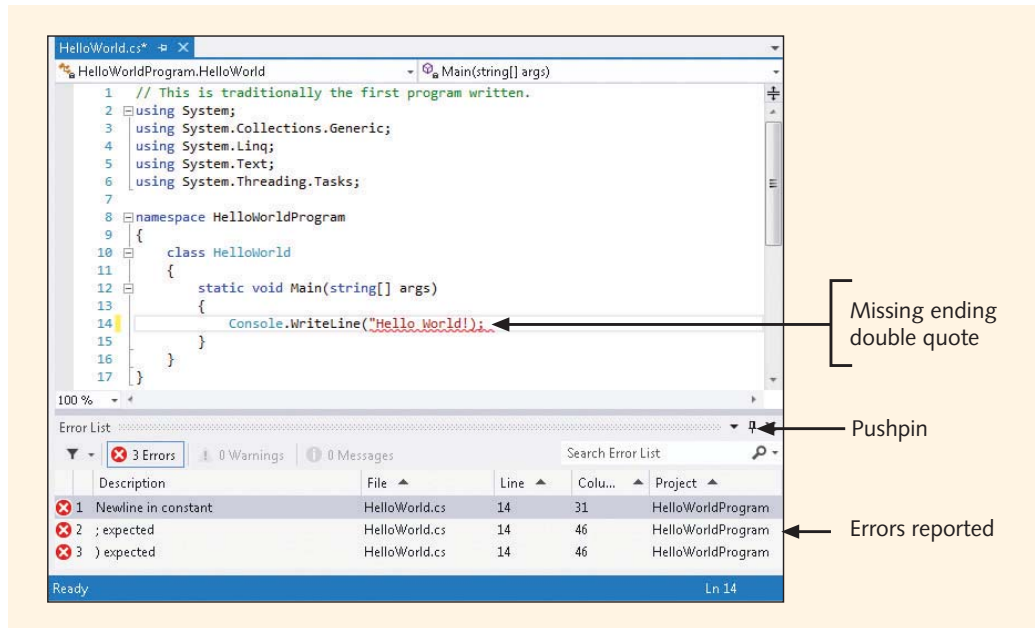


FIGURE 1-20 Syntax error message listing

The error messages are displayed in the **Error List** window found at the bottom of the IDE. The IDE also underlines the supposed location of the problem. You can also double-click on any of the error messages to go directly to the line flagged by the compiler. If the **Error List** tab is not visible on your screen, you can select **Error List** from the **View** menu.

A pushpin icon appears in the upper-right corner of all tool windows such as that shown in the **Error List** window. When the pushpin stands up like a thumbtack, the window is docked in place; thus, the **Error List** window is docked in place in Figure 1-20.

NOTE Tool windows support a feature called Auto Hide. If you click the pushpin so that it appears to be lying on its side, the window minimizes along the edges of the IDE. A small tab with the window name appears along the edge. This frees up space so you can see more of your source code.

The syntax error shown in Figure 1-20 is a common mistake. The double quote was omitted from the end of the argument to the `WriteLine()` method at Line 14. The error message does not say that, however. When you are reviewing error messages, keep in mind that the message might not be issued until the compiler reaches the next line or next block of code. Look for the problem in the general area that is flagged. Some of the most common errors are failing to end with an opening curly brace (`{`) or having an extra closing curly brace (`}`), failing to type a semicolon at the end of a statement (`;`), and misspelling names. As a good exercise, consider purposefully omitting curly braces,

semicolons, and misspelling words. See what kind of error messages each mistake generates. You will then be more equipped to find those errors quickly when you develop more sophisticated applications in the future.

NOTE

Because one error can generate several messages, it is probably best to fix the first error and then recompile rather than trying to fix all the errors in one pass.

Run-time Errors

Run-time errors are much more difficult to detect than syntax errors. A program containing run-time errors might compile without any problems, run, and produce results. Run-time errors can also cause a program to crash. A program might be written to retrieve data from a file. If that file is not available, when the program runs, a run-time error occurs. Another type of run-time error is division by zero.

Many times, a program compiles and executes without errors, but does not perform as expected. If the results are incorrect, the associated error is called a **logic error**. Logic errors are not identified until you look closely at the results and verify their correctness. For example, a value might not be calculated correctly. The wrong formula might be used. Failing to understand the problem specification fully is the most common culprit in logic errors. It is not enough to produce output; the output must be a correct solution to the problem.

Another potential run-time error type will be introduced when you start working with data in Chapter 2. If you are using data for calculations or performing different steps based on the value of data, it is easy to encounter a run-time error. Run-time errors can be minimized during software development by performing a thorough analysis and design before beginning the coding phase. The common strategy of desk checking also leads to more accurate results.

Creating an Application

Now that you understand what is required in most C# programs and how to compile and see your results, work through the following example using the suggested methodology for program development introduced earlier in this chapter. In this section, you will design a solution for the following problem definition.

PROGRAMMING EXAMPLE: ProgrammingMessage

The problem specification is shown in Figure 1-21.

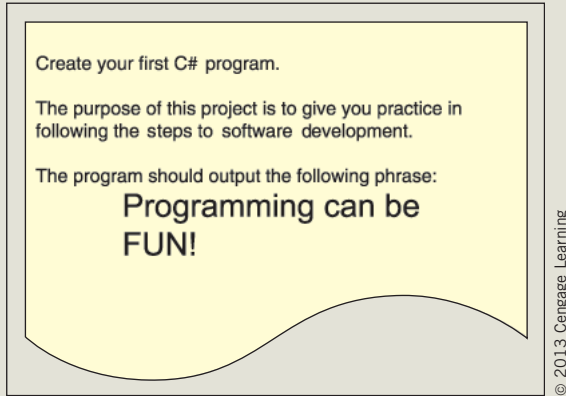


FIGURE 1-21 Problem specification sheet for the ProgrammingMessage example

ANALYZE THE PROBLEM

Do you understand the problem definition? This step is often slighted or glossed over. It is easy to scan the problem definition and think you have a handle on the problem, but miss an important feature. If you do not devote adequate time and energy analyzing the problem definition, much time and expense might be wasted in later steps. This is one, if not the most important, step in the development process. Ask questions to clarify the problem definition if necessary. You want to make sure you fully grasp what is expected.

As you read the problem definition given in Figure 1-21, note that no program inputs are required. The only data needed for this application is a string of characters. This greatly simplifies the analysis phase.

DESIGN A SOLUTION

The desired output, as noted in the problem specification sheet in Figure 1-21, is to display “Programming can be FUN!” on two lines. For this example, as well as any other application you develop, it is helpful to determine what your final output should look like. One way to document your desired output is to construct a **prototype**, or mock-up, of the output. Prototypes range from being elaborate designs created with graphics, word-processing, or paint programs, to being quite cryptic sketches created with paper and pencil. It is crucial to realize the importance of constructing a prototype, no matter which method you use. Developing a prototype helps you construct your algorithm. Prototypes also provide additional documentation detailing the purpose of the application. Figure 1-22 shows a prototype of the final output for the ProgrammingMessage example.

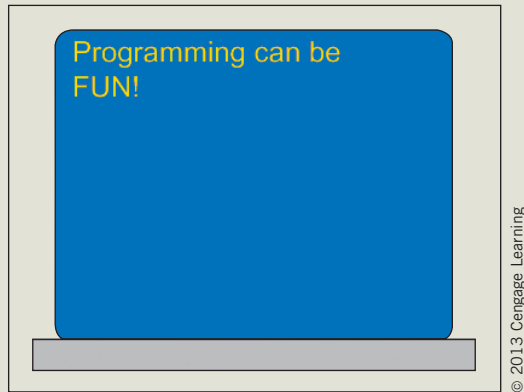


FIGURE 1-22 Prototype for the ProgrammingMessage example

During design, it is important to develop an algorithm. The algorithm for this problem could be developed using a flowchart. The algorithm should include a step-by-step solution for solving the problem, which, in this case, is straightforward and involves merely the output of a string of characters. Figure 1-23 contains a flowchart defining the steps needed for the ProgrammingMessage example.

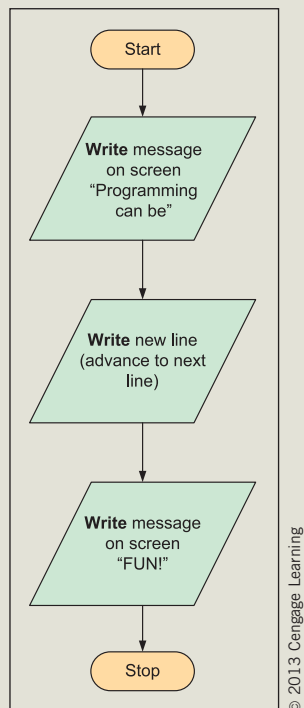


FIGURE 1-23 Algorithm for ProgrammingMessage example

Another option is to use structured English or pseudocode to define the algorithm. The pseudocode for this problem would be very short. It would include a single line to display the message Programming can be FUN! on the output screen.

Using an object-oriented approach to design, the solutions normally entail creating **class** diagrams. No data members would need to be defined for this problem. The **Console class** in the **System namespace** already has methods you can use to display output on a standard output device. Thus, no new methods or behaviors would need to be defined. So, if you were to construct a **class** diagram, it would include only a heading for the **class**. Nothing would appear in the middle or bottom portion of the diagram. Because the diagram does not provide additional documentation or help with the design of this simple problem, it is not drawn.

After the algorithm is developed, the design should be checked for correctness. One way to do this is to desk check your algorithms by mimicking the computer and working through the code step-by-step as if you were the computer. When you step through the flowchart, it should produce the output that appears on the prototype. It is extremely important that you carefully design and check your algorithm for correctness before beginning to write your code. You will spend much less time and experience much less frustration if you do this.

CODE THE SOLUTION

After you have completed the design and verified the algorithm's correctness, it is time to translate the design into source code. You can type source code statements into the computer using a simple editor such as Notepad, or you can create the file using Visual Studio. In this step of the process, you must pay attention to the language syntax rules.

If you create the application using Visual Studio, the IDE automatically generates much of the code for you. Some of that code can be removed or disregarded. For example, did you notice that the four **using** statements (**using System.Collections.Generic;** **using System.Linq;** **using System.Text;** and **using System.Threading.Tasks;**) were removed in the previous example? You can again remove or disregard these clauses. The **using System;** is the only **using** clause that needs to remain with your program statements for most of the applications that you will be developing.

Visual Studio also modifies the **Main()** method's heading from what you saw previously in this chapter. The signature for **Main()** can have an empty argument list or include **string[] args** inside the parentheses. For the types of applications you are developing, you do not need to send the additional argument. So, you can also disregard or completely remove the argument inside the parentheses to **Main()** at this time.

You might want to change the name of the source code file, and allow Visual Studio to change all references to that name. When you typed **ProgrammingMessage** as the project name, the IDE automatically named the **namespace ProgrammingMessage**. The **class** could also be called **ProgrammingMessage**, and this would cause no name clashing problems such as the **namespace** being given the same name as the **class** name. Figure 1-24 illustrates the changes you might want to make to the code generated by Visual Studio.

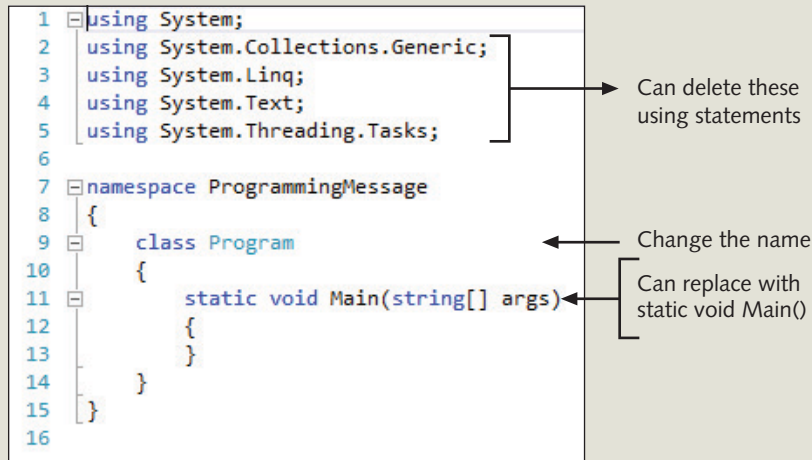


FIGURE 1-24 Recommended deletions

The only lines that you had to type were found in the `Main()` body.

```

Console.WriteLine("Programming can be");
Console.WriteLine("FUN!");
Console.ReadKey( );

```

The final program listing looks like this:

```

/* Programmer:    [supply your name]
   Date:         [supply the current date]
   Purpose:      This class can be used to send messages
                  to the output screen.
*/
using System;
namespace ProgrammingMessage
{
    class ProgrammingMessage
    {
        static void Main( )
        {
            Console.WriteLine("Programming can be");
            Console.WriteLine("FUN!");
            Console.ReadKey( );
        }
    }
}

```

At the beginning of your program, identify the author of the code, and as a minimum specify the purpose of the project. Note that the statements inside the `Main()` method are executed in sequential order. The `ReadKey()` method is executed after the two `WriteLine()` methods. `ReadKey()` is used in a program to keep the

output screen displayed until the user presses a key. After a character is pressed on the keyboard, control passes to the next line that marks the end of the application.

IMPLEMENT THE CODE

During implementation, the source code is compiled to check for rule violations. To compile from within the Visual Studio IDE, use the **Build** menu option. If you have errors, they must be corrected before you can go forward. From within the Visual Studio IDE, select **Start Without Debugging** on the **Debug** menu bar to see the results.

TEST AND DEBUG

Just because you have no compiler syntax errors and receive output does not mean the results are correct. During this final step, test the program and ensure you have the correct result. The output should match your prototype. Is your spacing correct?

By following the steps outlined in this chapter, you have officially become a C# programmer. The program you created was a simple one. However, the concepts you learned in this chapter are essential to your progress. In Chapter 2, you will begin working with data.

Coding Standards

It is important to follow coding standards when you design classes. Doing so will lead to better solutions and reduce the amount of time needed when you make changes to your program statements. You should also follow standards while initially designing your algorithms using flowcharts and pseudocode. Following are some suggested coding standards or guidelines as they relate to pseudocode.

Pseudocode

- For arithmetic operations, use verbs such as compute or calculate to imply some form of arithmetic is needed.
- Use words such as set, reset, or increment to imply what type of actions should be performed.
- Use print or display to indicate what should be shown on the screen.
- Group items and add indentation to imply they belong together.
- For statements that should be performed more than one time, use keywords like while or do while to imply looping.
- Use if or if/else for testing the contents of memory locations.

Developing standards that you consistently adhere to will increase your coding efficiency and make your code more maintainable.

Resources

There are enormous numbers of sites devoted to just C# on the Web. You might start your exploring at one or more of the following sites:

- Current C# Language Specifications – <http://www.microsoft.com/en-us/download/details.aspx?id=7029>
- Visual C# Express download – <http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-csharp-express>
- History of computing project – <http://www.thocp.net/>
- Intel processor information – <http://www.intel.com>
- Pascaline – <http://www.thocp.net/hardware/pascaline.htm>
- The Microsoft .NET Web site – <http://www.microsoft.com/net>
- The MSDN Visual C# home page – <http://msdn2.microsoft.com/en-us/vcsharp/default.aspx>
- U.S. Census Data on Computer and Internet Use – <http://www.census.gov/cps/>
- Wikipedia Free Encyclopedia – <http://wikipedia.org>
- Mono cross platform open source .NET framework – <http://www.mono-project.com>
- Microsoft Developer Network – <http://msdn.microsoft.com/en-us/>

QUICK REVIEW

1. The power of the computer rests with software, which is the set of instructions or programs that give the hardware functionality.
2. Many consider today's computer technology to be in the fifth generation of modern computing. Each era is characterized by an important advancement.
3. Mobile applications for smart devices today are increasing in demand.
4. Software can be divided into two categories: system software and application software. Application software is defined as the programs developed to perform a specific task.

5. The type of software most often associated with system software is the operating system. The operating system software is loaded when you turn on the computer. Other types of system software are compilers, interpreters, and assemblers.
6. Programming is a process of problem solving. The hardest part is coming up with a plan to solve the problem.
7. The five problem-solving steps for software development introduced in this chapter include analyzing, designing, coding, implementing, and testing and debugging the solution.
8. Procedural programming is process oriented and focuses on the processes that data undergoes from input until meaningful output is produced.
9. The underlying theme of top-down design or stepwise refinement is that given any problem definition, the logic can be refined by using the divide-and-conquer approach.
10. Software maintenance refers to upgrading or changing applications.
11. Using an object-oriented analysis approach, the focus is on determining the objects you want to manipulate rather than the logic required to manipulate them.
12. Encapsulation refers to combining attributes and actions or characteristics and behaviors to form a class.
13. An object is an instance of a class.
14. Through inheritance, it is possible to define subclasses of data objects that share some or all of the main class characteristics of their parents or super-classes. Inheritance enables reuse of code.
15. C# was designed from scratch to work with the new programming paradigm, .NET, and was the language used for development of much of .NET.
16. .NET is a software environment in which programs run. It is not the operating system. It is a layer between the operating system and other applications, providing an easier framework for developing and running code.
17. Through using Visual Studio (which is an IDE) and the .NET Framework classes, C# provides an easy way to create graphical user interfaces.
18. Originally, development with C# was possible only for the Windows platform; however, a number of projects are in development that are porting C# to other platforms such as Linux.
19. C# can be used to create Web, Windows, and console applications.
20. Web pages are created using Web forms, which are part of the ASP.NET technology.
21. Windows applications are considered desktop bound and designed for a single platform.
22. Console applications are the easiest to create. Values can be entered and produced with minimal overhead.

23. C# programs usually begin with a comment or **using** directives, followed by an optional **namespace** grouping and then the required **class** definition.
24. All C# programs must define a **class**.
25. Comments are written as notes to yourself or to readers of your program. The compiler ignores comments.
26. It is not necessary to end single inline comments (**//**); they end when the Enter key is pressed. Comments that span more than one line should be enclosed between **/* */**. These are considered block or multiline comments.
27. Over 2000 classes make up the Framework class library. A **class** contains data members and methods or behaviors that can be reused.
28. The using-namespace-directive imports all the types contained in the given **namespace**. By specifying the **namespace** around which you are building an application, you can eliminate the need of preceding a **class** with a dot (**.**) and the name of the **namespace**.
29. Everything in C# is designed around a **class**. Every program must have at least one **class**.
30. A method is a collection of one or more statements taken together that perform an action. In other words, a method is a small block of code that performs an action.
31. The **Main()** method is the entry point for every C# console application. It is the point at which execution begins.
32. The keyword **static** indicates that a single copy of the method is created.
33. The keyword **void** is included to signal that no value is returned. The complete signature of a method starts with the return type, followed by the name of the method, and finally a parenthesized list of arguments. One signature for **Main()** is **void static Main()**.
34. **WriteLine()** writes a string message to the monitor or a standard output device.
35. Methods communicate with each other through arguments placed inside parentheses.
36. Readability is important. Indenting is not required, but it is a good practice because it makes the code easier to read.
37. To see the results of a program, you must type the statements (source code) into a file, compile that code, and then execute the application.
38. Visual Studio integrated development environment (IDE) is an interactive development environment that enables you to type the source code, compile, and execute without leaving the IDE program.
39. One way to document your desired output is to construct a prototype, or mock-up, of your output.
40. The **Read()** method accepts any character from a standard input device, such as a keyboard. It does nothing with the character.