CHAPTER ONE

INTRODUCTION TO C#

1.1 Introduction

In this chapter, we introduce console applications—these input and output text in a console window, which in Windows Operating System(s) is known as the Command Prompt. We use live-code examples to demonstrate input/output, text formatting and arithmetic, equality and relational operators.

1.2 A Simple C# Application: Displaying a Line of Text

Let's consider an application that displays a line of text. (Later in this section we discuss how to compile and run an application.) The application and its output are shown in Fig. 1.1. The application illustrates several important C# language features. For your convenience, each program we present includes line numbers, which are not part of actual C# code. In Section 1.3 we show how to display line numbers for your C# code in the IDE. We'll soon see that line 10 does the real work of the application—namely, displaying the phrase Welcome to C# Programming! on the screen. We now do a code walkthrough.

Line 1 begins with //, indicating that the remainder of the line is a comment. We begin every application with a comment indicating the figure number and the name of the file in which the application is stored.

```
1    // Fig. 1.1: Welcome1.cs
2    // Text-displaying application.
3    using System;
4
5    public class Welcome1
6    {
7       // Main method begins execution of C# application
8       public static void Main( string[] args )
9       {
10          Console.WriteLine( "Welcome to C# Programming!" );
11      } // end Main
12   } // end class Welcome1
```

```
Welcome to C# Programming!
```

Fig. 1.1 | Text-displaying application.

A comment that begins with // is called a single-line comment, because it terminates at the end of the line on which it appears. A // comment also can begin in the middle of a line and continue until the end of that line (as in lines 7, 11 and 12).Delimited comments such as

```
/* This is a delimited comment.
   It can be split over many lines */
```

can be spread over several lines. This type of comment begins with the delimiter / and ends with the delimiter */. All text between the delimiters is ignored by the compiler. C# incorporated delimited comments and single-line comments from the C and C++ programming languages, respectively. In this book, we use only single-line comments in our programs.*

Line 2 is a single-line comment that describes the purpose of the application. Line 3 is a using directive that tells the compiler where to look for a class that is used in this application. A great strength of Visual C# is its rich set of predefined classes that you can reuse rather than "reinventing the wheel." These classes are organized under namespaces—named collections of related classes. Collectively, .NET's namespaces are referred to as the .NET Framework Class Library. Each using directive identifies a namespace containing pre- defined classes that a C# application should be able to use. The using directive in line 3 indicates that this example uses classes from the System namespace, which contains the predefined Console class (discussed shortly) used in line 10, and many other useful classes.

Line 5 begins a class declaration for the class Welcome1. Every application you create consists of at least one class declaration that is defined by you. These are known as user-defined classes. The class keyword introduces a class declaration and is immediately followed by the class name (Welcome1). Keywords (also called reserved words) are reserved for use by C# and are always spelled with all lowercase letters. The complete list of C# keywords is shown in Fig. 1.2.

C# Keywords and contextual keywords

break	bool	Base	as	abstract
checked	char	catch	case	byte
default	decimal	continue	const	class
enum	else	double	do	delegate
finally	false	extern	explicit	event
goto	foreach	For	float	fixed
interface	int	In	implicit	if
namespace	long	Lock	is	internal
out	operator	object	null	new
public	protected	private	params	override
sealed	sbyte	return	ref	readonly
string	static	stackalloc	sizeof	short
true	throw	This	switch	struct
unchecked	ulong	Uint	typeof	try
void	virtual	using	ushort	unsafe
			while	volatile
Contextual Keywords				
descending	by	ascending	alias	add
group	global	Get	from	equals
orderby	on	Let	join	into
value	set	select	remove	partial
		yield	where	var

Fig. 1.2 | C# keywords and contextual keywords.

By convention, all class names begin with a capital letter and capitalize the first letter of each word they include (e.g., SampleClassName). This is frequently referred to as Pascal casing. A class name is an identifier—a series of characters consisting of letters, digits and underscores (_) that does not begin with a digit and does not contain spaces. Some valid identifiers are

Welcome1, identifier, _value and m_inputField1. The name 7button is not a valid identifier because it begins with a digit, and the name input field is not a valid identifier because it contains a space. Normally, an identifier that does not begin with a capital letter is not the name of a class. C# is case sensitive—that is, uppercase and low- ercase letters are distinct, so a1 and A1 are different (but both valid) identifiers. Identifiers may also be preceded by the @ character. This indicates that a word should be interpreted as an identifier, even if it's a keyword (e.g., @int). This allows C# code to use code written in other .NET languages where an identifier might have the same name as a C# keyword.

The contextual keywords in Fig. 1.2 can be used as identifiers outside the contexts in which they're keywords, but for clarity this is not recommended. In Chapters 3–9, every class we define begins with the keyword public. When you save your public class declaration in a file, the file name is usually the class name followed by the .cs file-name extension. For our application, the file name is Welcome1.cs.

A left brace (in line 6 in Fig. 1.1), {, begins the body of every class declaration. A corresponding right brace (in line 12), }, must end each class declaration. Lines 7–11 are indented. This indentation is one of the spacing conventions mentioned earlier. We define each spacing convention as a Good Programming Practice.

For each application, one of the methods in a class must be called Main (which is typically defined as shown in line 8); otherwise, the application will not execute. Methods are able to perform tasks and return information when they complete their tasks. Keyword void (line 8) indicates that this method will not return any information after it completes its task. Later, we'll see that many methods do return information.

Line 10 displays the string of characters contained between the double quotation marks. Whitespace characters in strings are not ignored by the compiler. Class Console provides standard input/output capabilities that enable applications to read and display text in the console window from which the application executes. The ***Console.WriteLine*** *method displays a line of text in the console window.*

The string in the parentheses in line 10 is the argument to the method. Method Console.WriteLine displays its argument in the console window. When Console.WriteLine completes its task, it positions the screen cursor at the beginning of the next line in the console window.

The entire line 10, including Console.WriteLine, the parentheses, the argument "Welcome to C# Programming!" in the parentheses and the semicolon (;), is called a statement. Most statements end with a semicolon. When the statement in line 10 executes, it displays the string Welcome to C# Programming! in the console window. A method is typically composed of one or more statements that perform the method's task.

1.3 Creating a Simple Application in Visual C# Express

Now that we have presented our first console application (Fig. 1.1), we provide a step-by- step explanation of how to compile and execute it using Visual C#.

Creating the Console Application

After opening Visual C# 2010 , select File > New Project… to display the New Project dialog (Fig. 1.3), then select the Console Application template. In the dialog's Name field, type Welcome1. Click OK to create the project. The IDE now contains the open console application, as shown in Fig. 1.4. The editor window already contains some code provided by the IDE. Some of this code is similar to that of Fig. 1.1. Some is not, and uses features that we have not yet discussed. The IDE inserts this extra code to help organize the application and to provide access to some common classes in the .NET Framework Class Library—at this point in the book, this code is neither required nor relevant to the discussion of this application; delete all of it.

The code coloring scheme used by the IDE is called syntax-color highlighting and helps you visually differentiate application elements. For example, keywords appear in blue, and comments appear in green. When present, comments are green. One example of a literal is the string passed to Console.WriteLine in line 10 of Fig. 1.1. You can customize the colors shown in the code editor by selecting Tools > Options…. This displays the Options dialog. Then expand the Environment node and select Fonts and Colors. Here you can change the colors for various code elements.

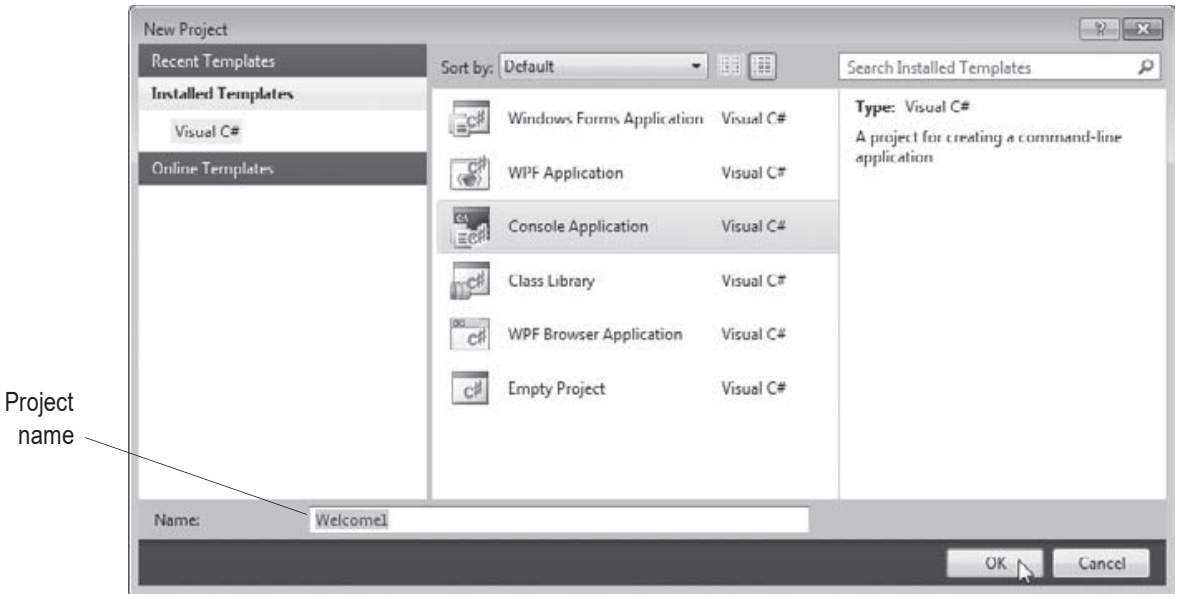

Fig. 1.3 | Creating a Console Application with the New Project dialog.

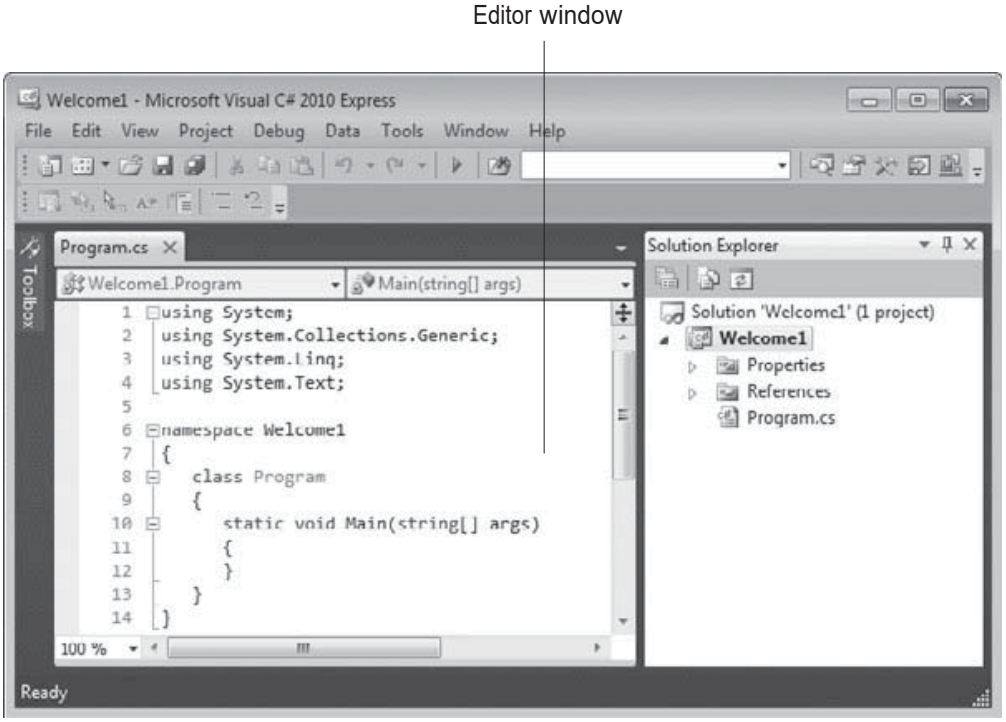

Fig. 1.4 | IDE with an open console application.

Modifying the Editor Settings to Display Line Numbers

Visual C# provides many ways to personalize your coding experience. In this step, you'll change the settings so that your code matches that of this book. To have the IDE display line numbers, select Tools > Options…. In the dialog that appears (Fig. 1.5), click the Show all settings checkbox on the lower left of the dialog, then expand the Text Editor node in the left pane and select All Languages. On the right, check the Line numbers check-box. Keep the Options dialog open.

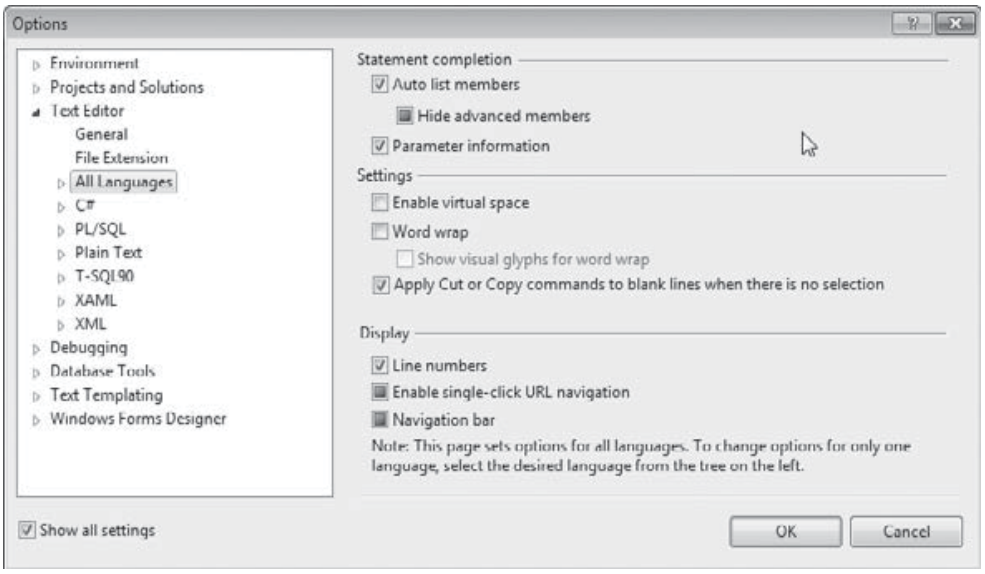

Fig. 1.5 | Modifying the IDE settings.

Changing the Name of the Application File

For applications we create in this book, we change the default name of the application file (i.e., Program.cs) to a more descriptive name. To rename the file, click Program.cs in the Solution Explorer window. This displays the application file's properties in the Properties window (Fig. 1.6). Change the File Name property to Welcome1.cs.

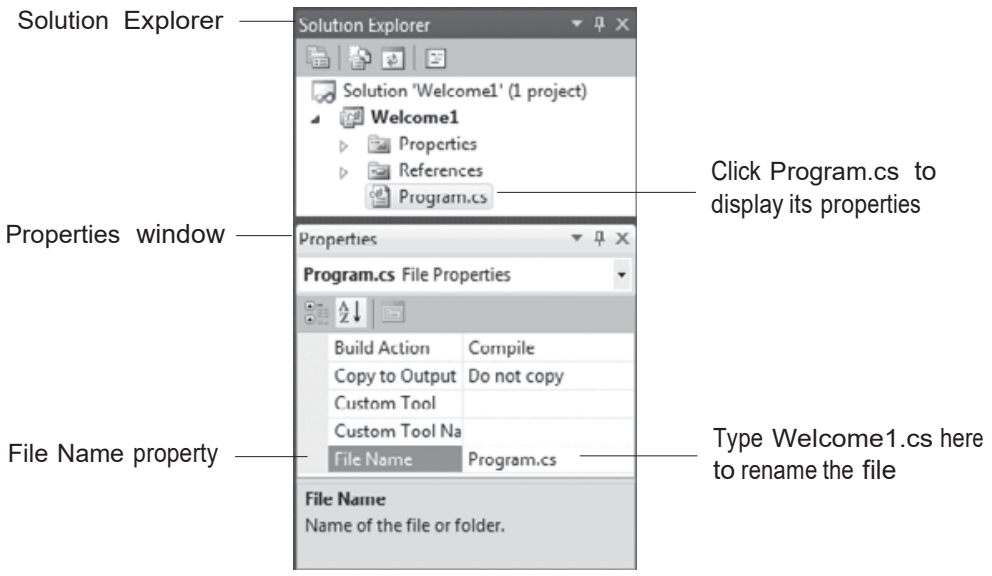

Fig. 1.6 | Renaming the program file in the Properties window.

Writing Code and Using IntelliSense

In the editor window (Fig. 1.4), type the code from Fig. 1.1. As you begin typing (in line 10) the class name class name Console, an IntelliSense window containing a scrollbar is displayed as shown in Fig. 1.7. This IDE feature lists a class's members, which include method names. When you type the dot (.) after Console, the IntelliSense window reappears and shows only the members of class Console that can be used on the right side of the dot (Fig. 1.7, part 1). When you type the open parenthesis character, (, after Console.WriteLine, the Parameter Info window is displayed (Fig. 1.8). This window contains information about the method's parameters.

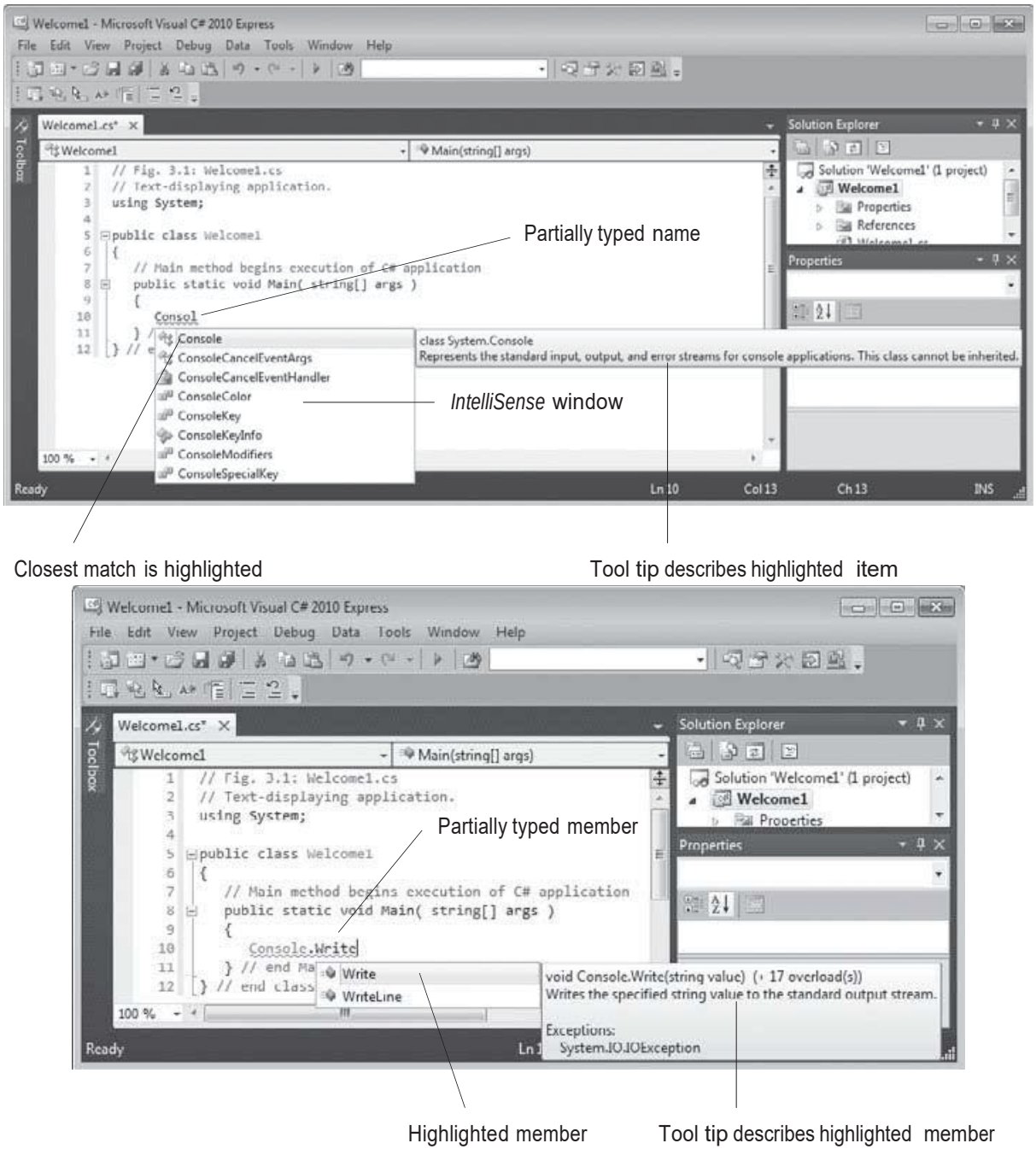

Fig. 1.7 | *IntelliSense* feature of Visual C#.

That is, a class can define several methods that have the same name, as long as they have different numbers and/or types of parameters—a concept known as overloaded methods. These methods normally all perform similar tasks. The Parameter Info window indicates how many versions of the selected method are available and provides up and down arrows for scrolling through the different versions. For example, there are 19 versions of the WriteLine method—we use one of these 19 versions in our application. The Parameter Info window is one of many features provided by the IDE to facilitate application development. From the code in Fig. 1.1, we already know that we intend to display one string with WriteLine, so, because you know exactly which version of WriteLine you want to use, you can simply close the Parameter Info window by pressing the Esc key.

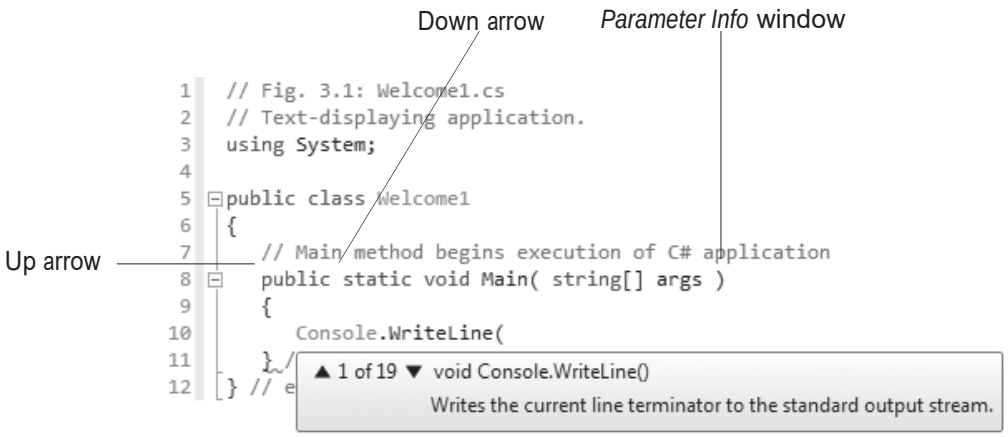

Fig. 1.8 | *Parameter Info* window.

Compiling and Running the Application

You're now ready to compile and execute your application. Depending on its type, the compiler may compile the code into files with a .exe (executable) extension, a .dll (dynamically linked library) extension or one of several other extensions. Such files are called assemblies and are the packaging units for compiled C# code. These assemblies contain the Microsoft Intermediate Language (MSIL) code for the application.

To compile the application, select Debug > Build Solution. If the application contains no syntax errors, this will compile your application and build it into an executable file (named Welcome1.exe, in one of the project's subdirectories). To execute it, type Ctrl + F5, which invokes the Main method (Fig. 1.1). (If you attempt to run the application before building it, the

IDE will build the application first, then run it only if there are no compilation errors.) The statement in line 10 of Main displays Welcome to C# Programming!. Figure 1.9 shows the results of executing this application, displayed in a console (Command Prompt) window. Leave the application's project open in Visual C#; we'll go back to it later in this section.

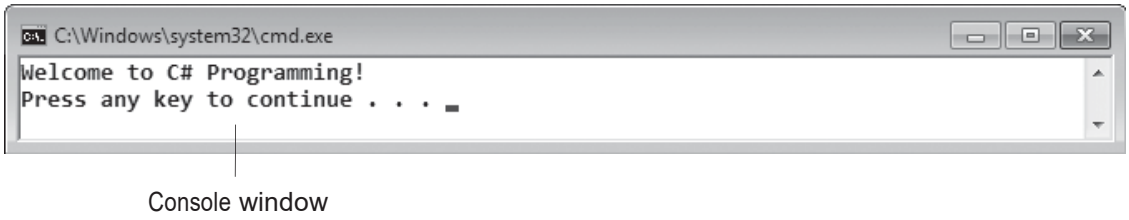

Fig. 1.9 | Executing the application shown in Fig. 1.1.

Syntax Errors, Error Messages and the Error List Window

Go back to the application in Visual C#. As you type code, the IDE responds either by applying syntax-color highlighting or by generating a syntax error, which indicates a violation of Visual C#'s rules for creating correct applications (i.e., one or more statements are not written correctly). Syntax errors occur for various reasons, such as missing parentheses and misspelled keywords.

When a syntax error occurs, the IDE underlines the error in red and provides a description of it in the Error List window (Fig. 1.10). If the Error List window is not visible in the IDE, select View > Error List to display it. In Figure 1.10, we intentionally omitted the comma between "Welcome to" and "C# Programming!" in line 10. The first error is simply indicating that line 10 is not a valid statement. The second error indicates that a right parenthesis is expected at character position 51 in the statement, because the compiler is confused by the unmatched left parenthesis from earlier in line 10. The third error has the text "Invalid expression term ')'", because the compiler thinks the closing right parenthesis should have appeared earlier in the line. The fourth error has the text"; expected", because the prior errors make the compiler think that the statement should have been terminated with a semicolon earlier in the line. Although we deleted only one comma in line 10, this caused the compiler to misinterpret several items in this line and to generate four error messages.

You can double click an error message in the Error List to jump to the place in the code that caused the error.

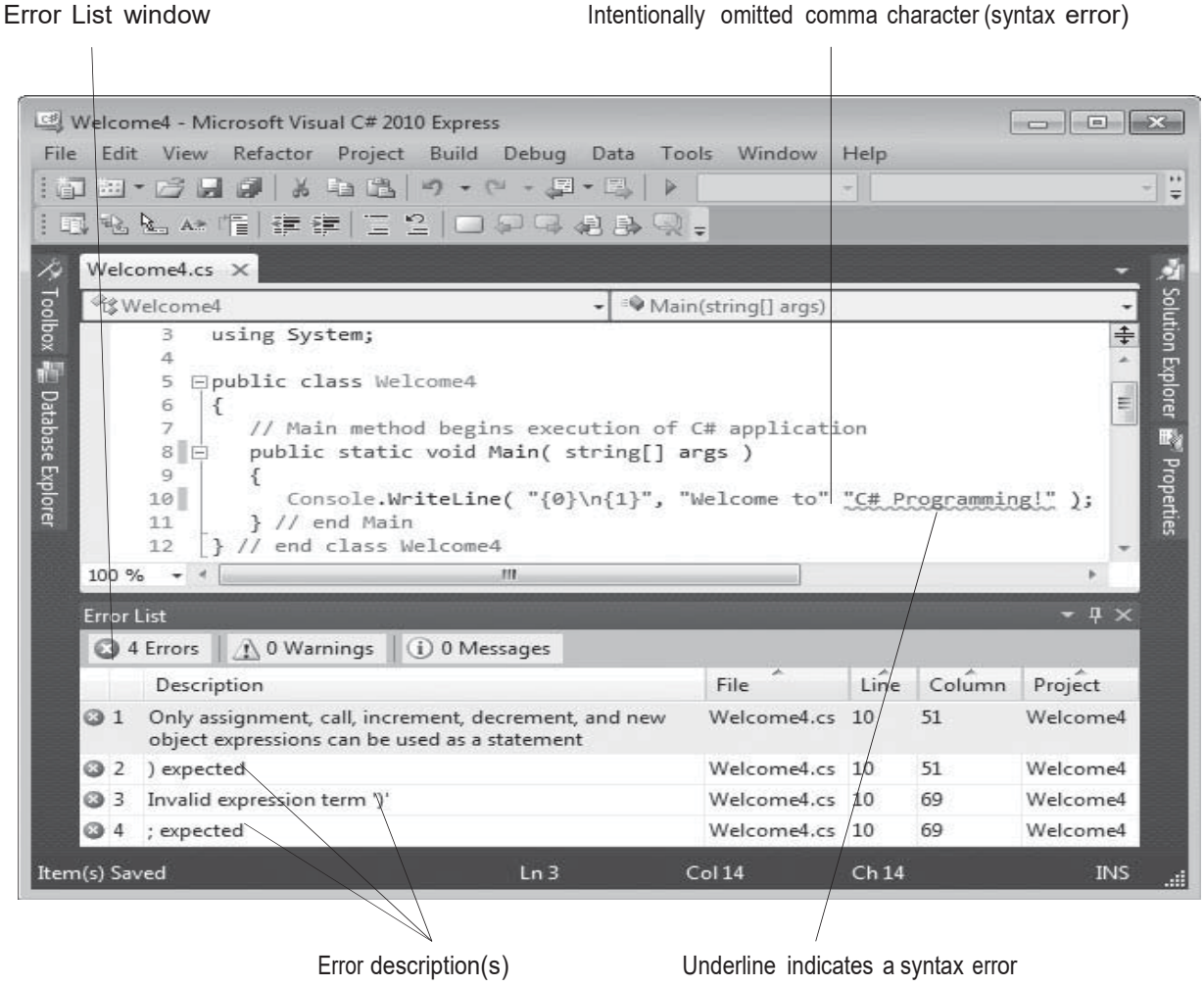

Fig. 1.10 | Syntax errors indicated by the IDE.

1.4 Modifying Your Simple C# Application

This section continues our introduction to C# programming with two examples that modify the example of Fig. 1.1 to display text on one line by using several statements and to display text on several lines by using only one statement.

Displaying a Single Line of Text with Multiple Statements

"Welcome to C# Programming!" can be displayed several ways. Class Welcome2, shown in Fig. 1.11, uses two statements to produce the same output as that shown in Fig. 1.1. From this point forward, we highlight the new and key features in each code listing, as shown in lines 10–11 of Fig. 1.11.

```
1  // Fig. 1.11: Welcome2.cs
2  // Displaying one line of text with multiple statements.
3  using System;
4
5  public class Welcome2
6  {
7     // Main method begins execution of C# application
8     public static void Main( string[] args )
9     {
10       Console.Write( "Welcome to " );
11       Console.WriteLine( "C# Programming!" );
12    } // end Main
13 } // end class Welcome2
```

```
Welcome to C# Programming!
```

Fig. 1.11 | Displaying one line of text with multiple statements.

The application is almost identical to Fig. 1.1. We discuss only the changes here. Lines 10–11 display one line of text in the console window. The first statement uses Console's method Write to display a string. Unlike WriteLine, after displaying its argument, Write does not position the screen cursor at the beginning of the next line in the console window—the next character the application displays will appear immediately after the last character that Write displays. Thus, line 11 positions the first character in its argument (the letter "C") immediately after the last character that line 10 displays. Each Write statement resumes displaying characters from where the last Write statement displayed its last character.

Displaying Multiple Lines of Text with a Single Statement

A single statement can display multiple lines by using newline characters, which indicate to Console methods Write and WriteLine when they should position the screen cursor to the beginning of the next line in the console window. Fig. 1.12 outputs four lines of text, using newline characters to indicate when to begin each new line.

```
1  // Fig. 1.12: Welcome1.cs
2  // Displaying multiple lines with a single statement.
3  using System;
4
5  public class Welcome3
6  {
7  // Main method begins execution of C# application
8     public static void Main( string[] args )
9     {
10       Console.WriteLine( "Welcome\nto\nC#\nProgramming!" );
11    } // end Main
12 } // end class Welcome3
```

```
Welcome to
C#
Programming!
```

Fig. 1.12 | Displaying multiple lines with a single statement. (Part 2 of 2.)

Most of the application is identical to the applications of Fig. 1.1 and Fig. 1.11, so we discuss only the changes here. Line 10 displays four separate lines of text in the console window. Normally, the characters in a string are displayed exactly as they appear in the double quotes. Note, however, that the two characters \ and n (repeated three times in the statement) do not appear on the screen. The backslash (\) is called an escape character. It indicates to C# that a "special character" is in the string. When a backslash appears in a string of characters, C# combines the next character with the backslash to form an escape sequence.

Escape sequence	Description
\n	Newline. Positions the screen cursor at the beginning of the next line.
\t	Horizontal tab. Moves the screen cursor to the next tab stop.
\r	Carriage return. Positions the screen cursor at the beginning of the current line—does not advance the cursor to the next line. Any characters output after the carriage return overwrite the characters previously output on that line.
\\	Backslash. Used to place a backslash character in a string.

\"	Double quote. Used to place a double-quote character (") in a string—e,g.,
	`Console.Write( "\"in quotes\"" );`
	displays
	`"in quotes"`

Fig. 1.13 | Some common escape sequences.

1.5 Formatting Text with `Console.Write` and `Console.WriteLine`

Console methods Write and WriteLine also have the capability to display formatted data. Figure 1.14 outputs the strings "Welcome to" and "C# Programming!" with WriteLine.

```
1  // Fig. 1.14: Welcome4.cs
2  // Displaying multiple lines of text with string formatting.
3  using System;
4
5  public class Welcome4
6  {
7     // Main method begins execution of C# application
8     public static void Main( string[] args )
9     {
10       Console.WriteLine( "{0}\n{1}", "Welcome to", "C# Programming!" );
11    } // end Main
12 } // end class Welcome4
```

```
Welcome to
C# Programming!
```

Fig. 1.14 | Displaying multiple lines of text with string formatting.

Line 10 calls method Console.WriteLine to display the application's output. The method call specifies three arguments. When a method requires multiple arguments, the arguments appear in a comma-separated list.

Most statements end with a semicolon (;). Therefore, line 10 represents only one statement. Large statements can be split over many lines, but there are some restrictions. Method WriteLine's first argument is a format string that may consist of fixed text and format items. Fixed text is output by WriteLine, as in Fig. 1.1. Each format item is a placeholder for a value. Format items also may include optional formatting information.

Format items are enclosed in curly braces and contain a sequence of characters that tell the method which argument to use and how to format it. For example, the format item {0} is a placeholder for the first additional argument (because C# starts counting from 0), {1} is a placeholder for the second, and so on. The format string in line 10 specifies that WriteLine should output two arguments and that the first one should be followed by a newline character. So this example substitutes "Welcome to" for the {0} and "C# Programming!" for the {1}. The output shows that two lines of text are

displayed. Because braces in a formatted string normally indicate a placeholder for text substitution, you must type two left braces ({{) or two right braces (}}) to insert a single left or right brace into a formatted string, respectively. We introduce additional formatting features as they're needed in our examples.

1.6 Another C# Application: Adding Integers

Our next application reads (or inputs) two integers (whole numbers, like –22, 7, 0 and 1024) typed by a user at the keyboard, computes the sum of the values and displays the result. This application keeps track of the numbers supplied by the user in variables. The application of Fig. 1.15 demonstrates these concepts. In the sample output, we highlight data the user enters at the keyboard in bold.

```
1  // Fig. 1.15: Addition.cs
2  // Displaying the sum of two numbers input from the keyboard.
3  using System;
4
5  public class Addition
6  {
7     // Main method begins execution of C# application
8     public static void Main( string[] args )
9     {
10       int number1; // declare first number to add
11       int number2; // declare second number to add
12       int sum; // declare sum of number1 and number2
13
14       Console.Write( "Enter first integer: " ); // prompt user
15       // read first number from user
16       number1 = Convert.ToInt32( Console.ReadLine() );
17
18       Console.Write( "Enter second integer:" ); // prompt user
19       // read second number from user
20       number2 = Convert.ToInt32( Console.ReadLine() );
21
22       sum = number1 + number2; // add numbers
23
24       Console.WriteLine( "Sum is {0}", sum ); // display sum
25    } // end Main
26 } // end class Addition
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

Fig. 1.15 | Displaying the sum of two numbers input from the keyboard.

The application begins execution with Main (lines 8–25). The left brace (line 9) marks the beginning of Main's body, and the corresponding right brace (line 25) marks the end of Main's body. Method Main is indented one level within the body of class Addition and the code in the body of Main is indented another level for readability. Line 10 is a variable declaration statement (also called a declaration) that specifies the name and type of a variable (number1) used in this application. Variables are typically declared with a name and a type before they're used. The name of a variable can be any valid identifier. (See Section 1.2 for identifier naming requirements.) Declaration state- ments end with a semicolon (;).

The declaration in line 10 specifies that the variable named number1 is of type int— it will hold integer values . The range of values for an int is –2,147,483,648 (int.MinValue) to +2,147,483,647 (int.MaxValue). Fig. 1.16 summarizes the characteristics of the simple types (bool, byte, sbyte, char, short, ushort, int, uint, long, ulong, float, double and decimal).

Name	Meaning	Range
sbyte	8-bit signed integer	-128–127
byte	8-bit unsigned integer	0–255
short	16-bit signed integer	-32,768–32,767
ushort	16-bit unsigned integer	0–65,535
int	32-bit signed integer	-2,147,483,648–2,147,483,647
uint	32-bit unsigned integer	0–4,294,967,295
long	64-bit signed integer	-9,223,372,036,854,775,808–9,223,372,036,854,775,807
ulong	64-bit unsigned integer	0–18,446,744,073,709,551,615
float	Single-precision float	1.5×10-45–3.4×1038
double	Double-precision float	5×10-324–1.7×10308
bool	Boolean	true, false
char	Unicode character	U+0000–U+ffff
decimal	Decimal value with 28-significant-digit precision	±1.0×1028–±7.9×1028

Fig. 1.16 | The Predefined Simple Types

The variable declaration statements at lines 11–12 similarly declare variables number2 and sum to be of type int. Variable declaration statements can be split over several lines, with the variable names separated by commas (i.e., a comma-separated list of variable names). Several variables of the same type may be declared in one declaration or in multiple declarations. For example, lines 10–12 can also be written as follows:

```
int number1, // declare first number to add
    number2, // declare second number to add
    sum; // declare sum of number1 and number2
```

Line 14 uses Console.Write to display the message "Enter first integer: ". This message is a prompt—it directs the user to take a specific action. Line 16 first calls the Console's ReadLine method. This method waits for the user to type a string of characters at the keyboard and press the Enter key. As we mentioned, some methods perform a task, then return the result of that task. In this case, ReadLine returns the text the user entered. Then, the string is used as an argument to class Convert's ToInt32 method, which converts this sequence of characters into data of type int. In this case, method ToInt32 returns the int representation of the user's input.

Technically, the user can type anything as the input value. ReadLine will accept it and pass it off to the ToInt32 method. This method assumes that the string contains a valid integer value. In this application, if the user types a noninteger value, a runtime logic error called an exception will occur and the application will terminate. This is also known as making your application fault tolerant. In line 16, the result of the call to method ToInt32 (an int value) is placed in variable number1 by using the assignment operator, =. The statement is read as "number1 gets the value returned by Convert.ToInt32." Operator = is a binary operator, because it works on two pieces of information. These are known as its operands—in this case, the operands are number1 and the result of the method call Convert.ToInt32. Line 18 prompts the user to enter the second integer. Line 20 reads a second integer and assigns it to the variable number2.Line 22 calculates the sum of number1 and number2 and assigns the result to variable `sum`*. Portions of statements that contain calculations are called expressions.*

For example, the value of the expression number1 + number2 is the sum of the numbers. Similarly, the value of the expression Console.ReadLine() is the string of characters typed by the user. After the calculation has been performed, line 24 uses method Console.WriteLine to display the sum. The format item {0} is a placeholder for the first argument after the format string. Other than the {0} format item, the remaining characters in the format string are all fixed text. So method WriteLine displays "Sum is ", followed by the value of sum (in the position of the {0} format item) and a newline.

Calculations can also be performed inside output statements. We could have combined the statements in lines 22 and 24 into the statement

```
Console.WriteLine( "Sum is {0}", ( number1 + number2 ) );
```

1.7 Arithmetic

The arithmetic operators are summarized in Fig. 1.17. Note the various special symbols not used in algebra. The asterisk (*) indicates multiplication, and the percent sign (%) is the remainder operator (called modulus in some languages), which we'll discuss shortly. The arithmetic operators in Fig. 1.17 are binary operators—for example, the expression f+7 contains the binary operator + and the two operands f and 7.

C# operation	Arithmetic operator	Algebraic expression	C# expression
Addition	+	$f + 7$	f + 7
Subtraction	-	$p - c$	p - c
Multiplication	*	$b \cdot m$	b * m
Division	/	x / y or - or $x \div y$	x / y
Remainder	%	$r \bmod s$	r % s

Fig. 1.17 | Arithmetic operators.

Integer division yields an integer quotient—for example, the expression 7/4 evaluates to 1, and the expression 17/5 evaluates to 3. Any fractional part in integer division is simply discarded (i.e., truncated)—no rounding occurs. C# provides the remainder operator, %, which yields the remainder after division. The expression x% y yields the remainder after x is divided by y.

Thus, 7%4 yields 3, and 17 % 5 yields 2. Arithmetic expressions must be written in straightline form to facilitate entering applications into the computer. Thus, expressions such as "a divided by b" must be written as a/ b, so that all constants, variables and operators appear in a straight line. The following algebraic notation is generally not acceptable to compilers:

$$\frac{a}{b}$$

Parentheses are used to group terms in C# expressions in the same manner as in algebraic expressions. For example, to multiply a times the quantity b+ c, we write

```
a * ( b + c )
```

If an expression contains nested parentheses, such as

```
( ( a + b ) * c )
```

the expression in the innermost set of parentheses (a+b in this case) is evaluated first. C# applies the operators in arithmetic expressions in a precise sequence determined by the following rules of operator precedence, which are generally the same as those followed in algebra (Fig. 1.18).

When we say that operators are applied from left to right, we're referring to their associativity. You'll see that some operators associate from right to left. Figure 1.18 summarizes these rules of operator precedence. We expand this table as additional operators are introduced. Appendix A provides the complete precedence chart.

Operators	Operations	Order of evaluation (associativity)
Evaluated first		
*	Multiplication	If there are several operators of this type, they're evaluated from left to right.
/	Division	
%	Remainder	
Evaluated next		
+	Addition	If there are several operators of this type, they're evaluated from left to right.
-	Subtraction	

Fig. 1.18 | Precedence of arithmetic operators.

1.8 Decision Making: Equality and Relational Operators

This section introduces a simple version of C#'s if statement that allows an application to make a decision based on the value of a condition. For example, the condition "grade is greater than or equal to 60" determines whether a student passed a test. If the condition in an if statement is true, the body of the if statement executes. If the condition is false, the body does not execute. We'll see an example shortly.

Conditions in if statements can be formed by using the equality operators (== and !=) and relational operators (>, <, >= and <=) summarized in Fig. 1.19. The two equality operators (== and !=) each have the same level of precedence, the relational operators (>,<, >= and <=) each have the same level of precedence, and the equality operators have lower precedence than the relational operators. They all associate from left to right.

Standard algebraic equality and relational operators	C# equality or relational operator	Sample C# condition	Meaning of C# condition
Equality operators			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y
Relational operators			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y

Fig. 1.19 | Equality and relational operators.