

An orange square with a black border containing the white text '1b'.

Introduction to C# Applications

OBJECTIVES

In this lecture you will learn:

- To write simple C# applications using code rather than visual programming.
- To write statements that input and output data to the screen.
- To declare and use data of various types.
- To store and retrieve data from memory.
- To use arithmetic operators.
- To determine the order in which operators are applied.
- To write decision-making statements.
- To use relational and equality operators.
- To use message dialogs to display messages.

- 3.1 Introduction**
- 3.2 A Simple C# Application: Displaying a Line of Text**
- 3.3 Creating a Simple Application in Visual C# Express**
- 3.4 Modifying Your Simple C# Application**
- 3.5 Formatting Text with `console.write` and `console.WriteLine`**
- 3.6 Another C# Application: Adding Integers**
- 3.7 Memory Concepts**
- 3.8 Arithmetic**
- 3.9 Decision Making: Equality and Relational Operators**

3.1 Introduction

- **Console applications** input and output text in a **console window**, which in Windows XP and Windows Vista is known as the **Command Prompt**.

- Figure 3.1 shows a simple application that displays a line of text.

welcome1.cs

```
1 // Fig. 3.1: 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
```

Comments improve code readability.

Class declaration for class welcome1.

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

Fig. 3.1 | Text-displaying application.



3.2 A Simple C# Application: Displaying a Line of Text (Cont.)

- Programmers insert **comments** to document applications.
- Comments improve code readability.
- The C# compiler ignores comments, so they do not cause the computer to perform any action when the application is run.
- 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.
- **Delimited comments** begin with the delimiter `/*` and end with the delimiter `*/`. All text between the delimiters is ignored by the compiler.



3.2 A Simple C# Application: Displaying a Line of Text (Cont.)

Common Programming Error 3.1

Forgetting one of the delimiters of a delimited comment is a syntax error. The **syntax** of a programming language specifies the rules for creating a proper application in that language. A **syntax error** occurs when the compiler encounters code that violates C#'s language rules.

3.2 A Simple C# Application: Displaying a Line of Text (Cont.)

- A **using directive** tells the compiler where to look for a predefined class that is used in an application.
- Predefined classes are organized under **namespaces**—named collections of related classes. Collectively, .NET's namespaces are referred to as the **.NET Framework Class Library**.
- The **System** namespace contains the predefined **Console** class and many other useful classes.

3.2 A Simple C# Application: Displaying a Line of Text (Cont.)

Error-Prevention Tip 3.1

Forgetting to include a `using` directive for a namespace that contains a class used in your application typically results in a compilation error, containing a message such as “The name 'Console' does not exist in the current context.” When this occurs, check that you provided the proper `using` directives and that the names in the `using` directives are spelled correctly, including proper use of Uppercase and lowercase letters.



3.2 A Simple C# Application: Displaying a Line of Text (Cont.)

- Programmers use blank lines and space characters to make applications easier to read.
- Together, blank lines, space characters and tab characters are known as **whitespace**. Whitespace is ignored by the compiler.

3.2 A Simple C# Application: Displaying a Line of Text (Cont.)

- **Keywords** (sometimes called **reserved words**) are reserved for use by C# and are always spelled with all lowercase letters.
- Every application consists of at least one **class declaration** that is defined by the programmer. These are known as **user-defined classes**.
- The **class** keyword introduces a class declaration and is immediately followed by the **class name**.

3.2 A Simple C# Application: Displaying a Line of Text (Cont.)

Good Programming Practice 3.1

By convention, always begin a class name's identifier with a capital letter and start each subsequent word in the identifier with a capital letter.

3.2 A Simple C# Application: Displaying a Line of Text (Cont.)

- A class name is an **identifier**:
 - Series of characters consisting of letters, digits and underscores (_).
 - Cannot begin with a digit and does not contain spaces.
- The complete list of C# keywords is shown in Fig. 3.2.

C# Keywords and contextual keywords				
abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out

Fig. 3.2 | C# keywords and contextual keywords. (Part 1 of 2.)



3.2 A Simple C# Application: Displaying a Line of Text (Cont.)

C# Keywords and contextual keywords				
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
volatile	while			
<i>Contextual Keywords</i>				
add	alias	ascending	by	descending
equals	from	get	global	group
into	join	let	on	orderby
partial	remove	select	set	value
var	where	yield		

Fig. 3.2 | C# keywords and contextual keywords. (Part 2 of 2.)

- The contextual keywords in Fig. 3.2 can be used as identifiers outside the contexts in which they are keywords, but for clarity this is not recommended.

3.2 A Simple C# Application: Displaying a Line of Text (Cont.)

- C# is **case sensitive**—that is, uppercase and lowercase letters are distinct, so `a1` and `A1` are different (but both valid) identifiers.

Common Programming Error 3.2

C# is case sensitive. Not using the proper uppercase and lowercase letters for an identifier normally causes a compilation error.

- Identifiers may also be preceded by the `@` character. This indicates that a word should be interpreted as an identifier, even if it is a keyword (e.g. `@int`).

3.2 A Simple C# Application: Displaying a Line of Text (Cont.)

Good Programming Practice 3.2

By convention, a file that contains a single `public` class should have a name that is identical to the class name (plus the `.CS` extension) in both spelling and capitalization. Naming your files in this way makes it easier for other programmers (and you) to determine where the classes of an application are located.

3.2 A Simple C# Application: Displaying a Line of Text (Cont.)

- A **left brace**, {, begins the **body** of every class declaration. A corresponding **right brace**, }, must end each class declaration.

Error-Prevention Tip 3.2

Whenever you type an opening left brace, {, in your application, immediately type the closing right brace, }, then reposition the cursor between the braces and indent to begin typing the body. This practice helps prevent errors due to missing braces.

Good Programming Practice 3.3

Indent the entire body of each class declaration one “level” of indentation between the left and right braces that delimit the body of the class. This format emphasizes the class declaration’s structure and makes it easier to read. You can let the IDE format your code by selecting **Edit > Advanced > Format Document**.



3.2 A Simple C# Application: Displaying a Line of Text (Cont.)

Good Programming Practice 3.4

Set a convention for the indent size you prefer, then uniformly apply that convention. The **Tab key may be used to create indents, but tab stops vary among text editors. We recommend using three spaces to form each level of indentation. We show how to do this in Section 3.3.**

Common Programming Error 3.3

It is a syntax error if braces do not occur in matching pairs.

3.2 A Simple C# Application: Displaying a Line of Text (Cont.)

- **Parentheses** after an identifier indicate that it is an application building block called a method. Class declarations normally contain one or more methods.
- Method names usually follow the same casing capitalization conventions used for class names.
- For each application, one of the methods in a class must be called `Main`; otherwise, the application will not execute.
- Methods are able to perform tasks and return information when they complete their tasks. Keyword **void** indicates that this method will not return any information after it completes its task.

3.2 A Simple C# Application: Displaying a Line of Text (Cont.)

- The **body** of a method declaration begins with a left brace and ends with a corresponding right brace.

Good Programming Practice 3.5

As with class declarations, indent the entire body of each method declaration one “level” of indentation between the left and right Braces that define the method body. This format makes the structure of the method stand out and makes the method declaration easier to read.

3.2 A Simple C# Application: Displaying a Line of Text (Cont.)

- Characters between double quotation marks are **strings**.
- Whitespace characters in strings are *not* ignored by the compiler.
- The **Console.WriteLine method** displays a line of text in the console window.
- The string in parentheses is the **argument** to the **Console.WriteLine method**.
- Method **Console.WriteLine** performs its task by displaying (also called outputting) its argument in the console window.

3.2 A Simple C# Application: Displaying a Line of Text (Cont.)

- A method is typically composed of one or more **statements** that perform the method's task.
- Most statements end with a semicolon.

Common Programming Error 3.4

Omitting the semicolon at the end of a statement is a syntax error.

3.2 A Simple C# Application: Displaying a Line of Text (Cont.)

Error-Prevention Tip 3.3

When the compiler reports a syntax error, the error may not be in the line indicated by the error message. First, check the line for which the error was reported. If that line does not contain syntax errors, check several preceding lines.

Good Programming Practice 3.6

Following the closing right brace of a method body or class declaration with a comment indicating the method or class declaration to which the brace belongs improves application readability.

3.3 Creating a Simple Application in Visual C# Express

Creating the Console Application

- Select **File > New Project...** to display the **New Project** dialog (Fig. 3.3).
- Select the **Console Application** template.
- In the dialog's **Name** field, type `Welcome1`, and click **OK** to create the project.

3.3 Creating a Simple Application in Visual C# Express (Cont.)

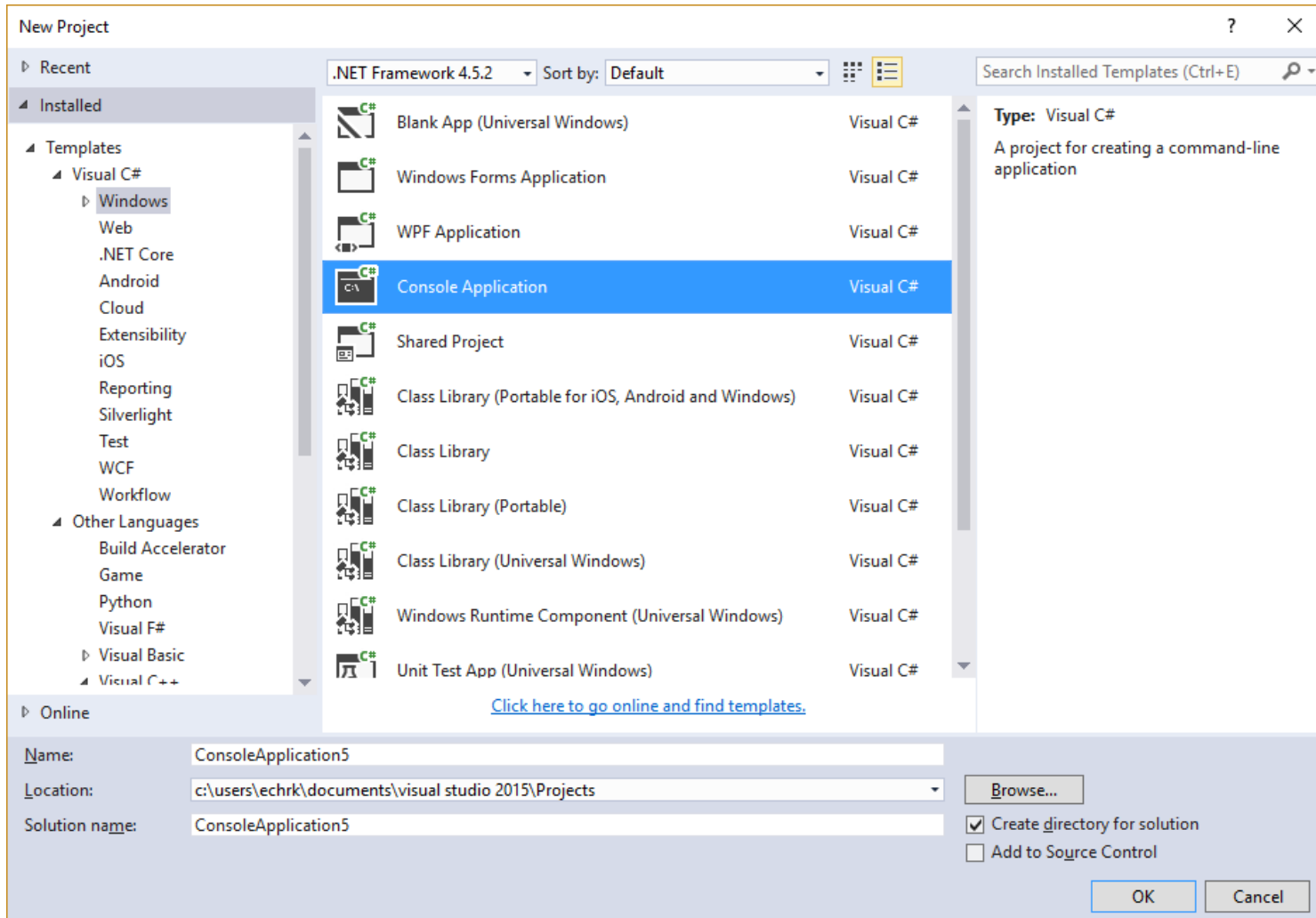


Fig. 3.3 | Creating a **Console Application** with the **New Project** dialog.



3.3 Creating a Simple Application in Visual C# Express (Cont.)

- The IDE now contains the open console application.
- The code coloring scheme used by the IDE is called **syntax-color shading** and helps you visually differentiate application elements.

3.3 Creating a Simple Application in Visual Studio(Cont.)

Editor window

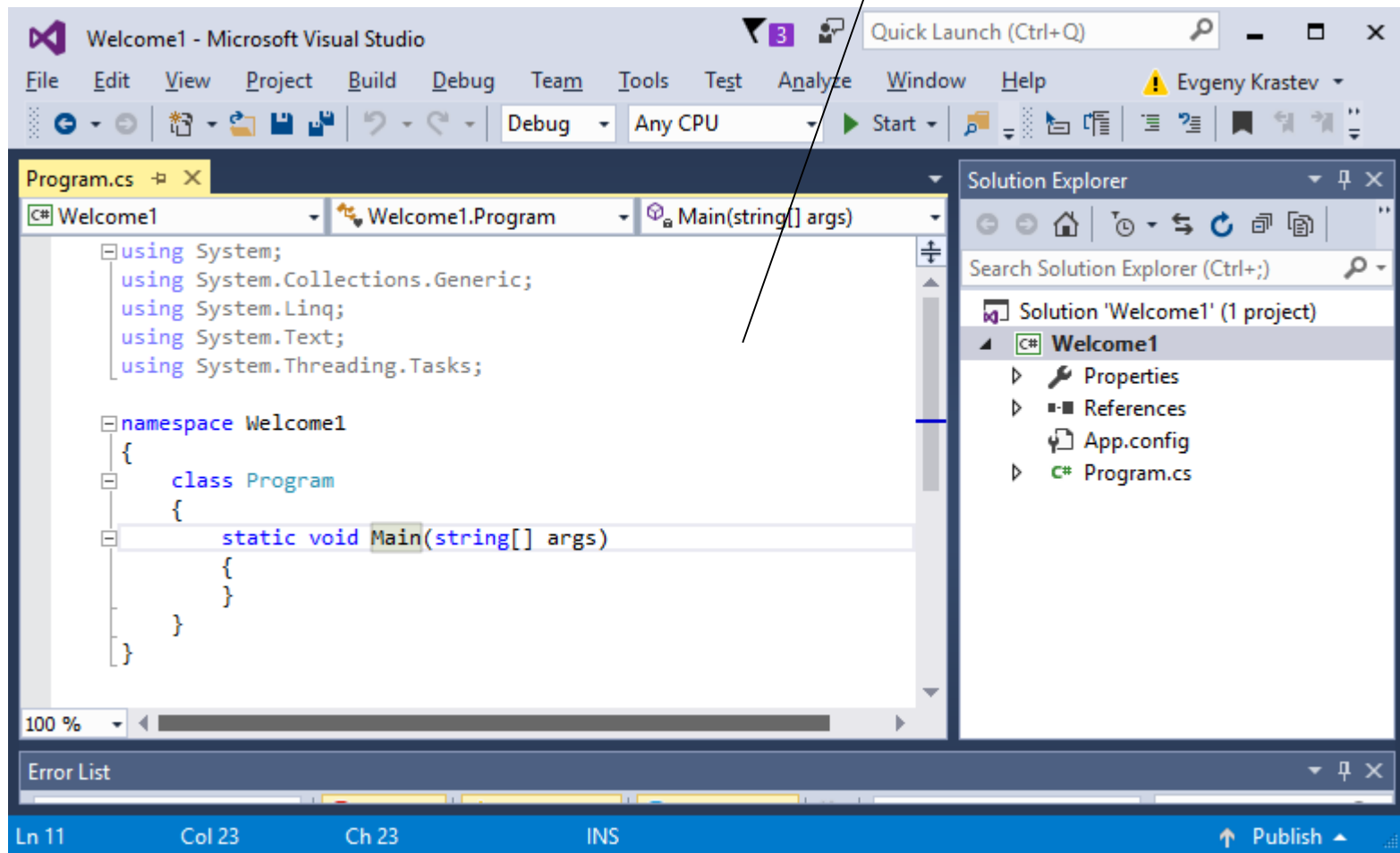


Fig. 3.4 | IDE with an open console application.

3.3 Creating a Simple Application in Visual C# Express (Cont.)

- To have the IDE display line numbers, select **Tools > Options...**
 - In the dialog that appears (Fig. 3.5), click the **Show all settings** checkbox on the lower left of the dialog.
 - Expand the **Text Editor** node in the left pane and select **All Languages**. On the right, check the **Line numbers** checkbox.

3.3 Creating a Simple Application in Visual C# Express (Cont.)

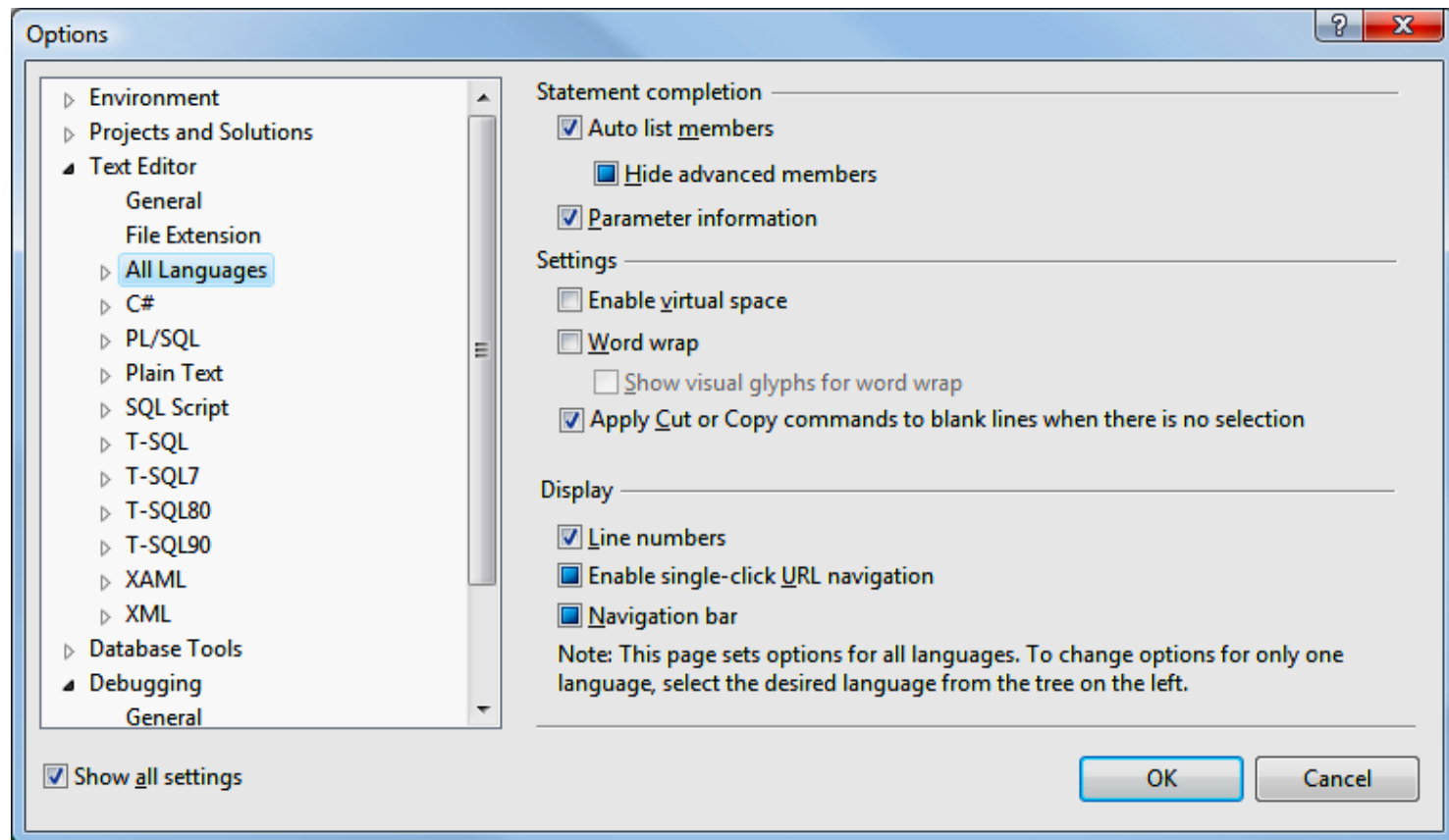


Fig. 3.5 | Modifying the IDE settings.

3.3 Creating a Simple Application in Visual C# Express (Cont.)

- To set code indentation to three spaces per indent:
 - In the **Options** dialog that you opened in the previous step, expand the C# node in the left pane and select **Tabs**.
 - Make sure that the option **Insert spaces** is selected. Enter **3** for both the **Tab size** and **Indent size** fields.
 - Click **OK** to save your settings, close the dialog and return to the editor window.

3.3 Creating a Simple Application in Visual C# Express (Cont.)

- To rename the application file, click `Program.cs` in the **Solution Explorer** window to display its properties in the **Properties** window (Fig. 3.6).
- Change the **File Name property** to `Welcome1.cs`.

3.3 Creating a Simple Application in Visual C# Express (Cont.)

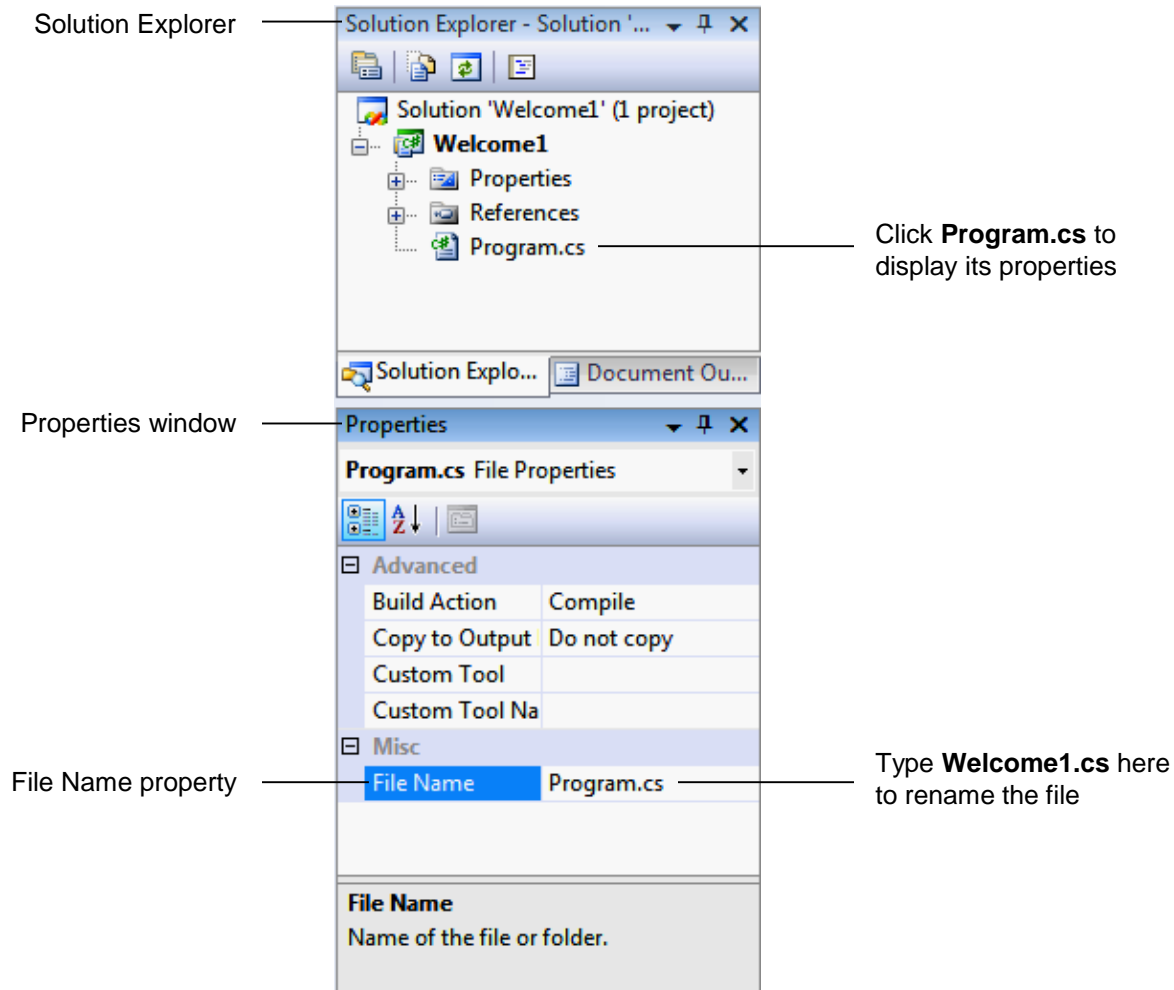


Fig. 3.6 | Renaming the program file in the **Properties** window.



3.3 Creating a Simple Application in Visual C# Express (Cont.)

- *IntelliSense* lists a class's **members**, which include method names.
- As you type characters, Visual C# Express highlights the first member that matches all the characters typed, then displays a tool tip containing a description of that member.
- You can either type the complete member name, double click the member name in the member list or press the *Tab* key to complete the name.
- While the *IntelliSense* window is displayed pressing the *Ctrl* key makes the window transparent so you can see the code behind the window.



3.3 Creating a Simple Application in Visual C# Express (Cont.)

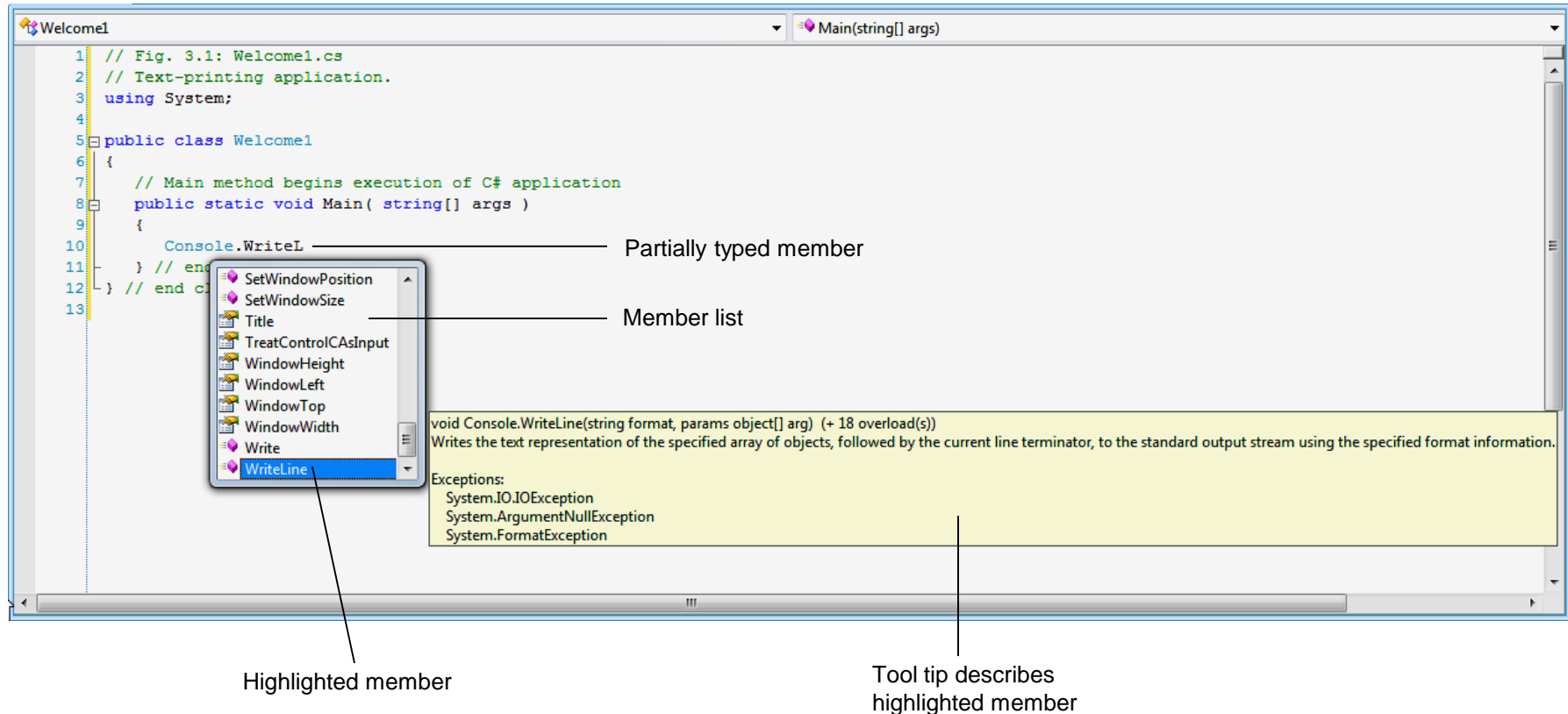


Fig. 3.7 | *IntelliSense* feature of Visual C# Express.

3.3 Creating a Simple Application in Visual C# Express (Cont.)

- When you type the open parenthesis character, (, after a method name, the *Parameter Info* window is displayed (Fig. 3.8).
- This window contains information about the method's parameters.

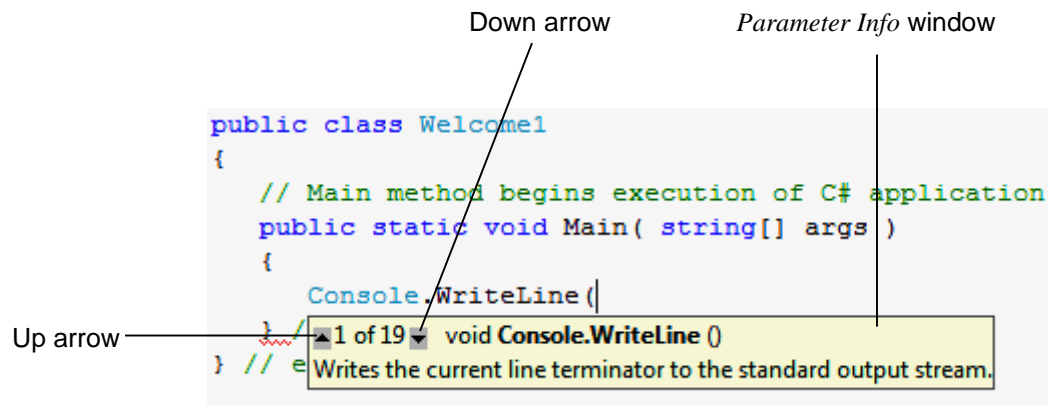


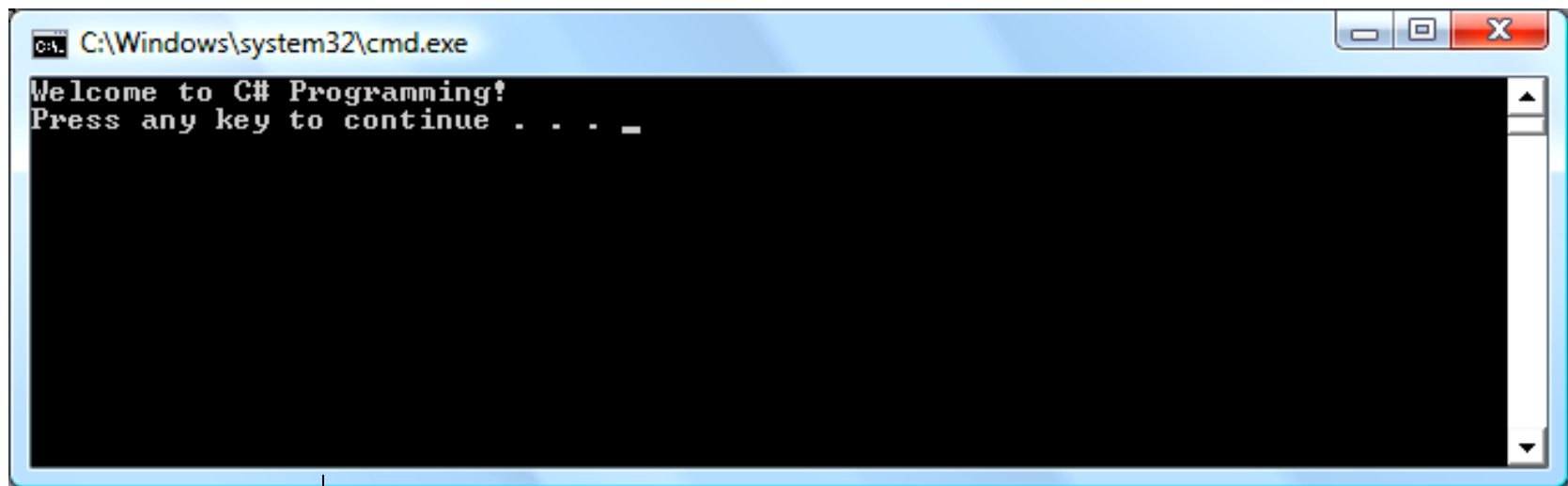
Fig. 3.8 | *Parameter Info* window.

- Up and down arrows allow you to scroll through overloaded versions of the method.

3.3 Creating a Simple Application in Visual C# Express (Cont.)

- To compile an application, select **Build > Build Solution**.
- To execute it, select **Debug > Start Without Debugging** (or type *Ctrl + F5*).
 - This invokes the `Main` method.
- Figure 3.10 shows the results of executing this application, displayed in a console (**Command Prompt**) window.

3.3 Creating a Simple Application in Visual C# Express (Cont.)



Console window

Fig. 3.10 | Executing the application shown in Fig. 3.1.

3.3 Creating a Simple Application in Visual C# Express (Cont.)

Error-Prevention Tip 3.4

When learning how to program, sometimes it is helpful to “break” a working application so you can familiarize yourself with the compiler’s syntax-error messages. Try removing a semicolon or brace from the code of Fig. 3.1, then recompiling the application to see the error messages generated by the omission.

3.3 Creating a Simple Application in Visual C# Express (Cont.)

*Running the Application from the **Command Prompt***

- To open the **Command Prompt**, select **Start > All Programs > Accessories > Command Prompt**.

Default prompt displays when
Command Prompt is opened



User enters the next command here

Fig. 3.11 | **Command Prompt** window when it is initially opened.

3.3 Creating a Simple Application in Visual C# Express (Cont.)

- To Enter the command `cd` (which stands for “change directory”), followed by the directory where the application’s `.exe` file is located
- Run the compiled application by entering the name of the `.exe` file.

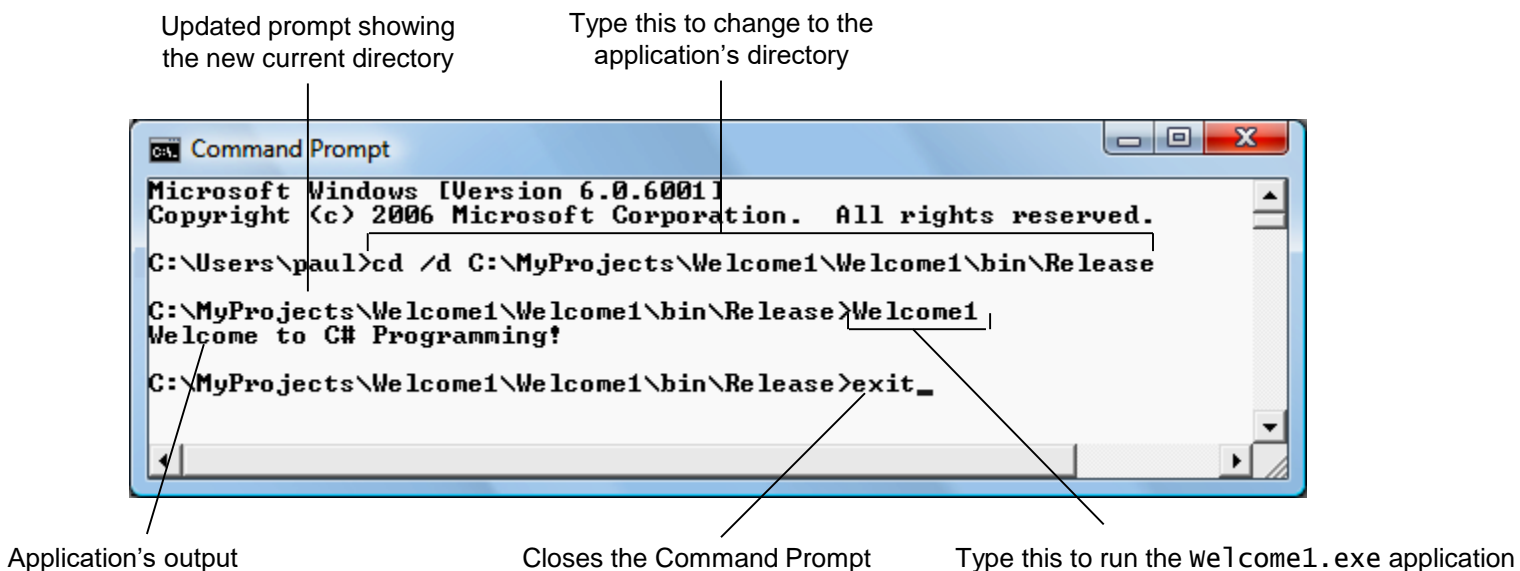


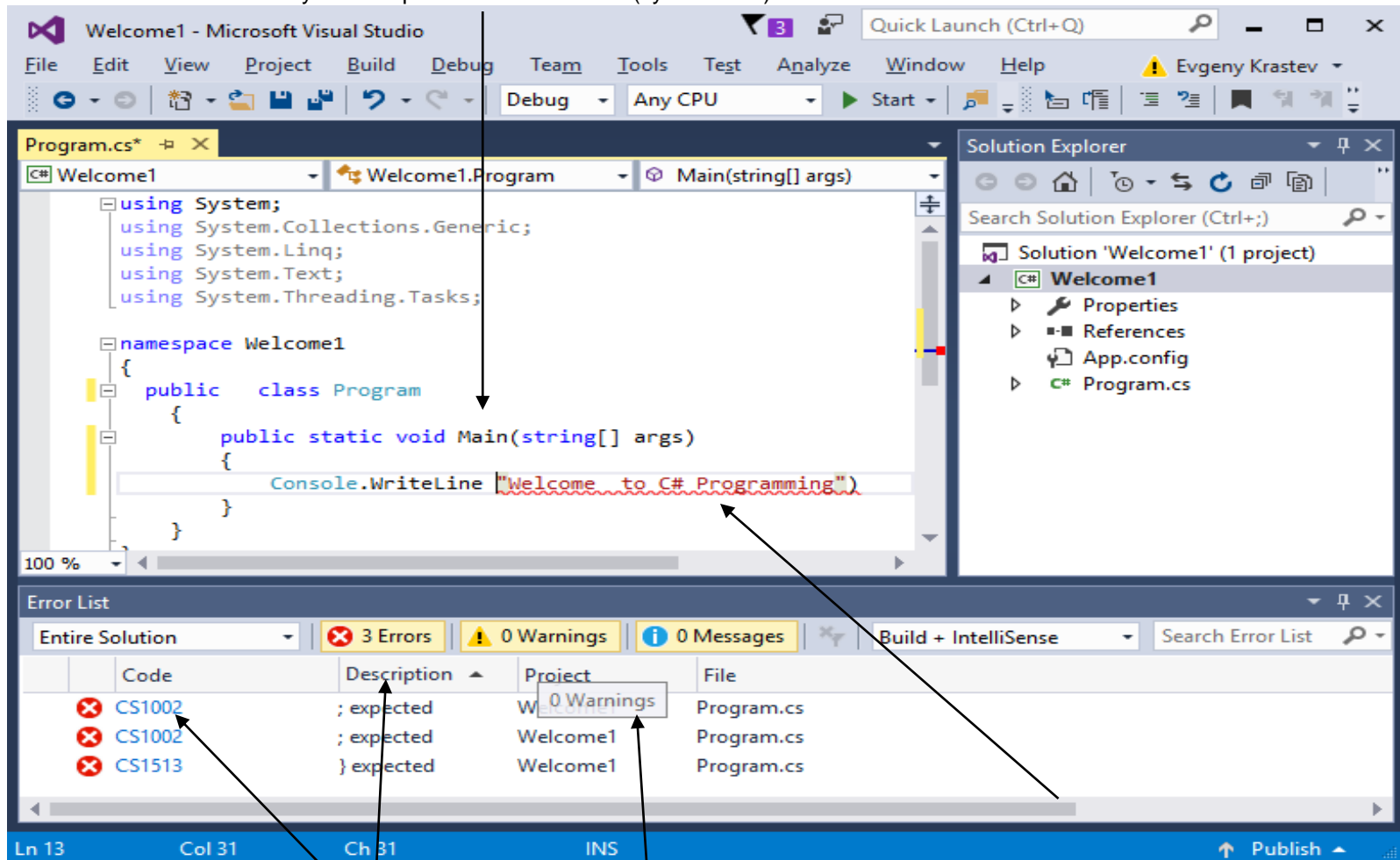
Fig. 3.12 | Executing the application shown in Fig. 3.1 from a **Command Prompt** window.

3.3 Creating a Simple Application in Visual C# Express (Cont.)

- As you type code, the IDE responds either by applying syntax-color highlighting or by generating a **syntax error**.
- A syntax error indicates a violation of Visual C#'s rules for creating correct applications.
- When a syntax error occurs, the IDE underlines the error in red and provides a description of it in the **Error List window** (Fig. 3.13).

3.3 Creating a Simple Application in Visual C# Express (Cont.)

Intentionally omitted parenthesis character (syntax error)



3.3 Creating a Simple Application in Visual C# Express (Cont.)

Error-Prevention Tip 3.5

One syntax error can lead to multiple entries in the **Error List** window. Each error that you address could eliminate several subsequent error messages when you recompile your application. So when you see an error you know how to fix, correct it and recompile—this may make several other errors disappear.

3.4 Modifying Your Simple C# Application

- Class `Welcome2`, shown in Fig. 3.14, uses two statements to produce the same output as that shown in the previous example.
- Unlike `WriteLine`, the `Console` class's `Write` method does not position the screen cursor at the beginning of the next line in the console window.

3.4 Modifying Your Simple C# Application (Cont.)

welcome2.cs

```
1 // Fig. 3.14: 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
```

The `Write` method does not move the cursor to a new line after displaying its argument.

welcome to C# Programming!

Fig. 3.14 | Displaying one line of text with multiple statements.



- A single statement can display multiple lines by using newline characters.
- Like space characters and tab characters, newline characters are whitespace characters.
- The application of Fig. 3.15 outputs four lines of text, using newline characters to indicate when to begin each new line.

welcome3.cs

```
1 // Fig. 3.15: welcome3.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. 3.15 | Displaying multiple lines with a single statement.



3.4 Modifying Your Simple C# Application (Cont.)

- The **backslash** (\) is called an **escape character**, and is used as the first character in an **escape sequence**.
- The escape sequence `\n` represents the **newline character**.
- Figure 3.16 lists several common escape sequences and describes how they affect the display of characters in the console window.

3.4 Modifying Your Simple C# Application (Cont.)

Escape sequence	Description
<code>\n</code>	Newline. Positions the screen cursor at the beginning of the next line.
<code>\t</code>	Horizontal tab. Moves the screen cursor to the next tab stop.
<code>\r</code>	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.
<code>\\</code>	Backslash. Used to place a backslash character in a string.
<code>\"</code>	Double quote. Used to place a double-quote character (") in a string—e.g., <code>Console.Write("\"in quotes\"");</code> displays <code>"in quotes"</code>

Fig. 3.16 | Some common escape sequences.

3.5 Formatting Text with Console.WriteLine

welcome4.cs

- Console methods Write and WriteLine also have the capability to display formatted data.
- Figure 3.17 shows another way to use the WriteLine method.

```
1 // Fig. 3.17: 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
```

Method WriteLine's first argument is a **format string** that may consist of **fixed text** and **format items**.

```
welcome to
C# Programming!
```

Fig. 3.17 | Displaying multiple lines of text with string formatting.



3.5 Formatting Text with `Console.WriteLine` and `Console.WriteLine` (Cont.)

- When a method requires multiple arguments, the arguments are separated with **commas**.

Good Programming Practice 3.7

Place a space after each comma (,) in an argument list to make applications more readable.

- **Large statements can be split over many lines, but there are some restrictions.**

Common Programming Error 3.5

Splitting a statement in the middle of an identifier or a string is a syntax error.

3.5 Formatting Text with `Console.WriteLine` and `Console.WriteLine` (Cont.)

- Method `WriteLine`'s first argument is a **format string** that may consist of **fixed text** and **format items**.
- Each format item is a placeholder for a value, corresponding to an additional argument to `WriteLine`.
 - `{0}` is a placeholder for the first additional argument.
 - `{1}` is a placeholder for the second, and so on.
- Format items also may include optional formatting information.

3.6 Another C# Application: Adding Integers

- Applications remember numbers and other data in the computer's memory and access that data through application elements called **variables**.
- A **variable** is a location in the computer's memory where a value can be stored for use later in an application.
- A **variable declaration statement** (also called a **declaration**) specifies the name and type of a variable.
 - A variable's name enables the application to access the value of the variable in memory—the name can be any valid identifier.
 - A variable's type specifies what kind of information is stored at that location in memory.

Outline

- Three variables declared as type `int`.

Addition.cs

(1 of 2)

```
1 // Fig. 3.18: 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     }
```

Three variables declared as type `int`.

The user is prompted for information.

`Console.ReadLine()` reads the data entered by the user, and `Convert.ToInt32` converts the value into an integer.

Fig. 3.18 | Displaying the sum of two numbers input from the keyboard. (Part 1 of 2).



Addition.cs

(2 of 2)

```
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. 3.18 | Displaying the sum of two numbers input from the keyboard. (Part 2 of 2).



3.6 Another C# Application: Adding Integers (Cont.)

- Variables of type **int** store **integer** values (whole numbers such as 7, -11, 0 and 31914).
- Types **float**, **double** and **decimal** specify real numbers (numbers with decimal points).
- Type **char** represents a single character.
- These types are called **simple types**. Simple-type names are keywords and must appear in all lowercase letters.

3.6 Another C# Application: Adding Integers (Cont.)

- 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.

Good Programming Practice 3.8

Declare each variable on a separate line. This format allows a comment to be easily inserted next to each declaration.

3.6 Another C# Application: Adding Integers (Cont.)

Good Programming Practice 3.9

Declare each variable on a separate line. This format allows a comment to be easily inserted next to each declaration.

Good Programming Practice 3.10

By convention, variable-name identifiers begin with a lowercase letter, and every word in the name after the first word begins with a capital letter. This naming convention is known as **camel casing.**

3.6 Another C# Application: Adding Integers (Cont.)

- The `Console`'s `ReadLine` method waits for the user to type a string of characters at the keyboard and press the *Enter* key.
- `ReadLine` returns the text the user entered.
- The `Convert` class's `ToInt32` method converts this sequence of characters into data of type `int`.
- `ToInt32` returns the `int` representation of the user's input.

3.6 Another C# Application: Adding Integers (Cont.)

- A value can be stored in a variable using the **assignment operator**, **=**.
- Operator **=** is called a **binary operator**, because it works on two pieces of information, or **operands**.
- An **assignment statement** assigns a value to a variable.
- Everything to the right of the assignment operator, **=**, is always evaluated before the assignment is performed.

Good Programming Practice 3.11

Place spaces on either side of a binary operator to make it stand out and make the code more readable.



3.6 Another C# Application: Adding Integers (Cont.)

- An **expression** is any portion of a statement that has a value associated with it.
 - The value of the expression `number1 + number2` is the sum of the numbers.
 - The value of the expression `Console.ReadLine()` is the string of characters typed by the user.
- Calculations can also be performed inside output statements.

3.7 Memory Concepts

- Variable names actually correspond to **locations** in the computer's memory.
- Every variable has a **name**, a **type**, a **size** and a **value**.
- In Fig. 3.19, the computer has placed the value 45 in the memory location corresponding to **number1**.



Fig. 3.19 | Memory location showing the name and value of variable **number1**.

3.7 Memory Concepts (Cont.)

- In Fig. 3.20, 72 has been placed in location `number2`.

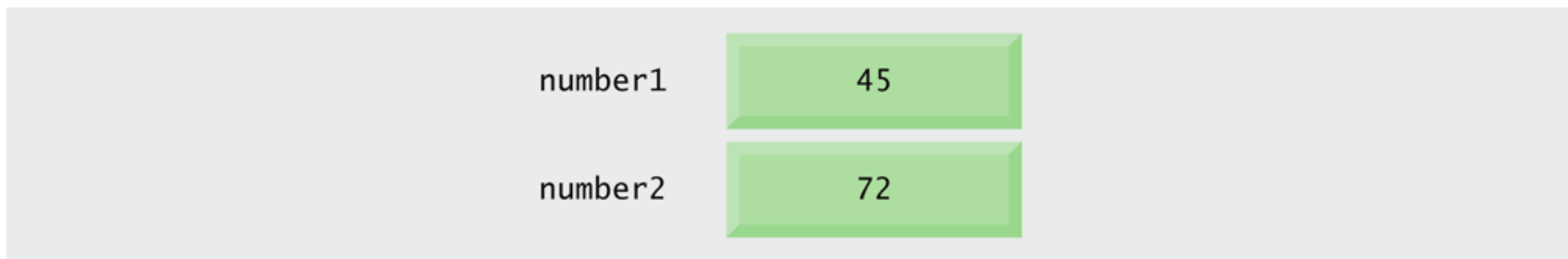


Fig. 3.20 | Memory locations after storing values for `number1` and `number2`.

3.7 Memory Concepts (Cont.)

- After `sum` has been calculated, memory appears as shown in Fig. 3.21.

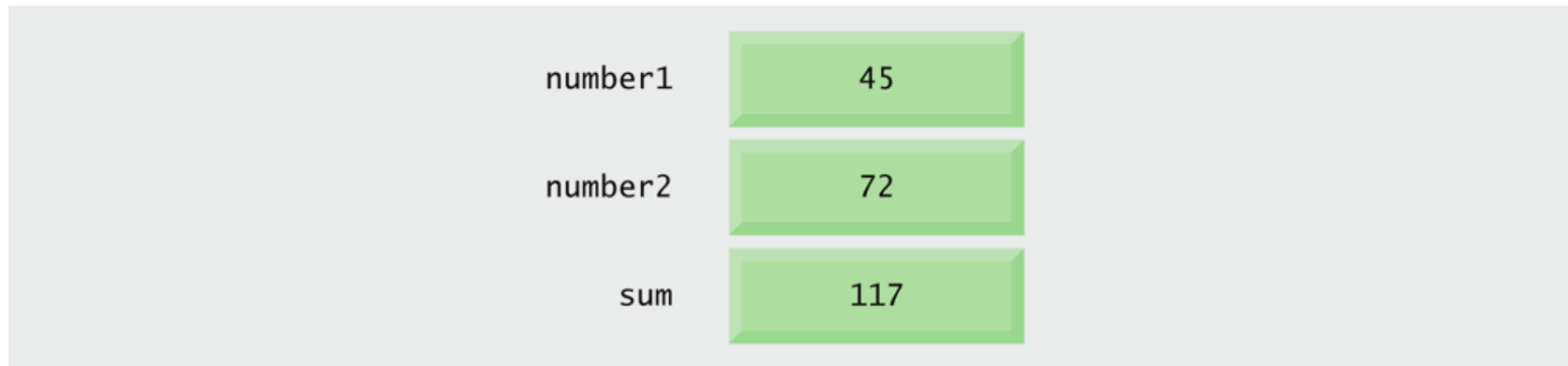


Fig. 3.21 | Memory locations after calculating and storing the sum of `number1` and `number2`.

3.7 Memory Concepts (Cont.)

- Whenever a value is placed in a memory location, the value replaces the previous value in that location, and the previous value is lost.
- When a value is read from a memory location, the process is nondestructive.

3.8 Arithmetic

- The **arithmetic operators** are summarized in Fig. 3.22.

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

Fig. 3.22 | Arithmetic operators.

- The arithmetic operators in Fig. 3.22 are binary operators.

3.8 Arithmetic (Cont.)

- **Integer division** yields an integer quotient—any fractional part in integer division is simply discarded without rounding.
- C# provides the remainder operator, %, which yields the remainder after division.
- The remainder operator is most commonly used with integer operands but can also be used with **floats**, **doubles**, and **decimals**.
- Parentheses are used to group terms in C# expressions in the same manner as in algebraic expressions.
- If an expression contains **nested parentheses**, the expression in the innermost set of parentheses is evaluated first.

3.8 Arithmetic (Cont.)

- Arithmetic operators are evaluated according to the **rules of operator precedence**, which are generally the same as those followed in algebra (Fig. 3.23).

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

Fig. 3.23 | Precedence of arithmetic operators.

3.8 Arithmetic (Cont.)

- The circled numbers under the statement indicate the order in which C# applies the operators.

Algebra: $z = pr \% q + w/x - y$

C#: `z = p * r % q + w / x - y;`




6 1 2 4 3 5

The multiplication, remainder and division operations are evaluated first in left-to-right order (i.e., they associate from left to right), because they have higher precedence than addition and subtraction. The addition and subtraction operations are evaluated next. These operations are also applied from left to right.

3.8 Arithmetic (Cont.)

- To develop a better understanding of the rules of operator precedence, consider the evaluation of a second-degree polynomial ($y = ax^2 + bx + c$):

$$y = a * x * x + b * x + c;$$


6 1 2 4 3 5

3.8 Arithmetic (Cont.)

- As in algebra, it is acceptable to place unnecessary (**redundant**) parentheses in an expression to make the expression clearer.
- The preceding assignment statement might be parenthesized to highlight its terms as follows:

$y = (a * x * x) + (b * x) + c ;$

3.8 Arithmetic (Cont.)

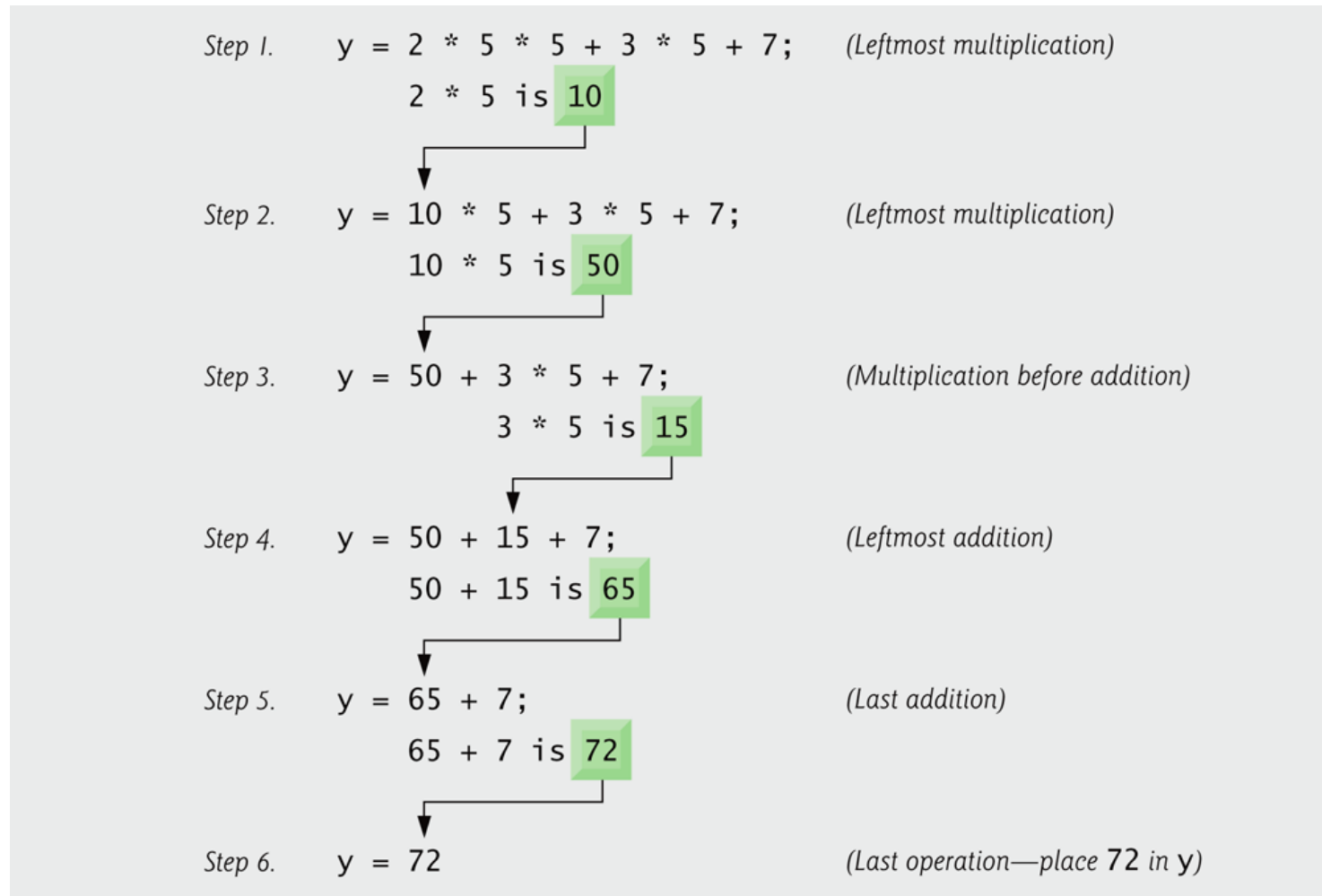


Fig. 3.24 | Order in which a second-degree polynomial is evaluated.

3.9 Decision Making: Equality and Relational Operators

- A **condition** is an expression that can be either **true** or **false**.
- Conditions can be formed using the **equality operators** (**==** and **!=**) and **relational operators** (**>**, **<**, **>=** and **<=**) summarized in Fig. 3.25.

Standard algebraic equality and relational operators	C# equality or relational operator	Sample C# condition	Meaning of C# condition
<i>Equality operators</i>			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y

Fig. 3.25 | Equality and relational operators. (Part 1 of 2.)

3.9 Decision Making: Equality and Relational Operators (Cont.)

Standard algebraic equality and relational operators	C# equality or relational operator	Sample C# condition	Meaning of C# condition
<i>Relational operators</i>			
>	>	<code>x > y</code>	x is greater than y
<	<	<code>x < y</code>	x is less than y
≥	>=	<code>x >= y</code>	x is greater than or equal to y
≤	<=	<code>x <= y</code>	x is less than or equal to y

Fig. 3.25 | Equality and relational operators. (Part 2 of 2.)

3.9 Decision Making: Equality and Relational Operators (Cont.)

Common Programming Error 3.6

Confusing the equality operator, `==`, with the assignment operator, `=`, can cause a logic error or a syntax error. The equality operator should be read as “is equal to,” and the assignment operator should be read as “gets” or “gets the value of.” To avoid confusion, some people read the equality operator as “double equals” or “equals equals.”

- Figure 3.26 uses six **if statements** to compare two integers entered by the user.

Comparison.cs

```
1 // Fig. 3.26: Comparison.cs
2 // Comparing integers using if statements, equality operators,
3 // and relational operators.
4 using System;
5
6 public class Comparison
7 {
8     // Main method begins execution of C# application
9     public static void Main( string[] args )
10    {
11        int number1; // declare first number to compare
12        int number2; // declare second number to compare
13
14        // prompt user and read first number
15        Console.Write( "Enter first integer: " );
16        number1 = Convert.ToInt32( Console.ReadLine() );
17
```

(1 of 3)

Fig. 3.26 | Comparing integers using if statements, equality operators and relational operators. (Part 1 of 3).



Comparison.cs

(2 of 3)

```
18 // prompt user and read second number
19 Console.Write( "Enter second integer: " );
20 number2 = Convert.ToInt32( Console.ReadLine() );
21
22 if ( number1 == number2 )
23     Console.WriteLine( "{0} == {1}", number1, number2 );
24
25 if ( number1 != number2 )
26     Console.WriteLine( "{0} != {1}", number1, number2 );
27
28 if ( number1 < number2 )
29     Console.WriteLine( "{0} < {1}", number1, number2 );
30
31 if ( number1 > number2 )
32     Console.WriteLine( "{0} > {1}", number1, number2 );
33
34 if ( number1 <= number2 )
35     Console.WriteLine( "{0} <= {1}", number1, number2 );
```

Compare number1 and
number2 for equality.

Fig. 3.26 | Comparing integers using if statements, equality operators and relational operators. (Part 2 of 3).



```
36
37     if ( number1 >= number2 )
38         Console.WriteLine( "{0} >= {1}", number1, number2 );
39     } // end Main
40 } // end class Comparison
```

Comparison.cs

(3 of 3)

```
Enter first integer: 42
Enter second integer: 42
42 == 42
42 <= 42
42 >= 42
```

```
Enter first integer: 1000
Enter second integer: 2000
1000 != 2000
1000 < 2000
1000 <= 2000
```

```
Enter first integer: 2000
Enter second integer: 1000
2000 != 1000
2000 > 1000
2000 >= 1000
```

Fig. 3.26 | Comparing integers using if statements, equality operators and relational operators. (Part 3 of 3).



3.9 Decision Making: Equality and Relational Operators (Cont.)

- If the condition in an `if` statement is true, the statement associated with that `if` statement executes.
- An `if` statement always begins with keyword `if`, followed by a condition in parentheses.
- An `if` statement expects one statement in its body.

Common Programming Error 3.7

Forgetting the left and/or right parentheses for the condition in an `if` statement is a syntax error—the parentheses are required.

3.9 Decision Making: Equality and Relational Operators (Cont.)

Common Programming Error 3.8

Reversing the operators `!=`, `>=` and `<=`, as in `=!`, `=>` and `=<`, can result in syntax or logic errors.

Common Programming Error 3.9

It is a syntax error if the operators `==`, `!=`, `>=` and `<=` contain spaces between their symbols, as in `= =`, `! =`, `> =` and `< =`, respectively.

Good Programming Practice 3.12

Indent an `if` statement's body to make it stand out and to enhance application readability.

3.9 Decision Making: Equality and Relational Operators (Cont.)

Common Programming Error 3.10

Placing a semicolon immediately after the right parenthesis of the condition in an `if` statement is normally a logic error.

Good Programming Practice 3.13

Place no more than one statement per line in an application. This format enhances readability.

Good Programming Practice 3.14

A lengthy statement can be spread over several lines. If a single statement must be split across lines, choose breaking points that make sense, such as after a comma in a comma-separated list, or after an operator in a lengthy expression. If a statement is split across two or more lines, indent all subsequent lines until the end of the statement.



3.9 Decision Making: Equality and Relational Operators (Cont.)

- Figure 3.27 shows the precedence of the operators introduced in this chapter from top to bottom in decreasing order of precedence.

Operators				Associativity	Type
*	/	%		left to right	multiplicative
+	-			left to right	additive
<	<=	>	>=	left to right	relational
==	!=			left to right	equality
=				right to left	assignment

Fig. 3.27 | Precedence and associativity of operations discussed.

3.9 Decision Making: Equality and Relational Operators (Cont.)

Good Programming Practice 3.15

Refer to the operator precedence chart (the complete chart is in Appendix A) when writing expressions containing many operators. Confirm that the operations in the expression are performed in the order you expect. If you're uncertain about the order of evaluation in a complex expression, use parentheses to force the order, as you would do in algebraic expressions. Observe that some operators, such as assignment, =, associate from right to left rather than from left to right.