3

Introduction to C# Applications

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.)

- 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.)

- **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.)

- 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			
Contextual Keywords				
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.)

- **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.

3.3 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.)

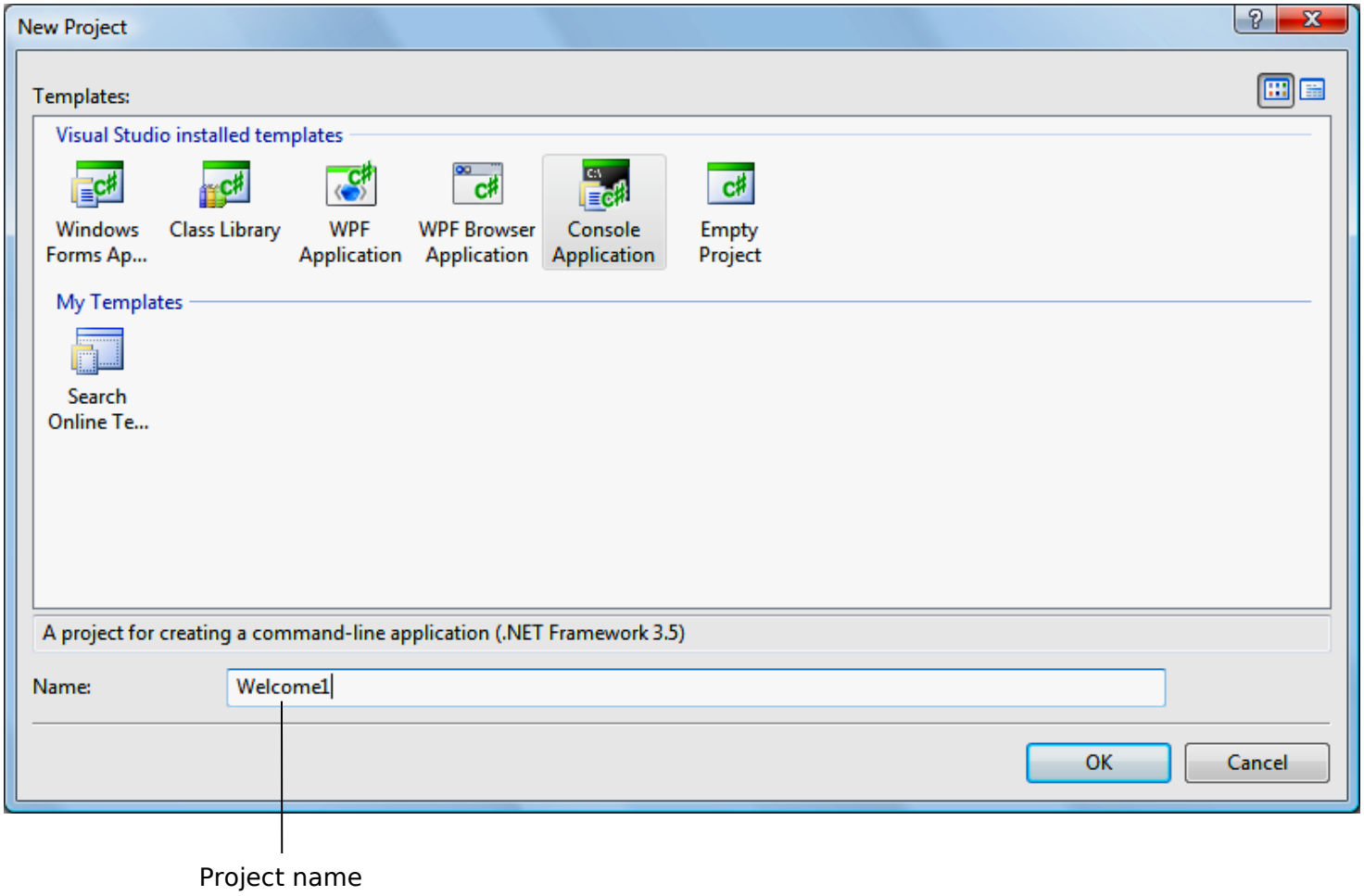

Fig. 3.3 | Creating a **Console Application** with the **New Project** dialog.

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 C# Express (Cont.)

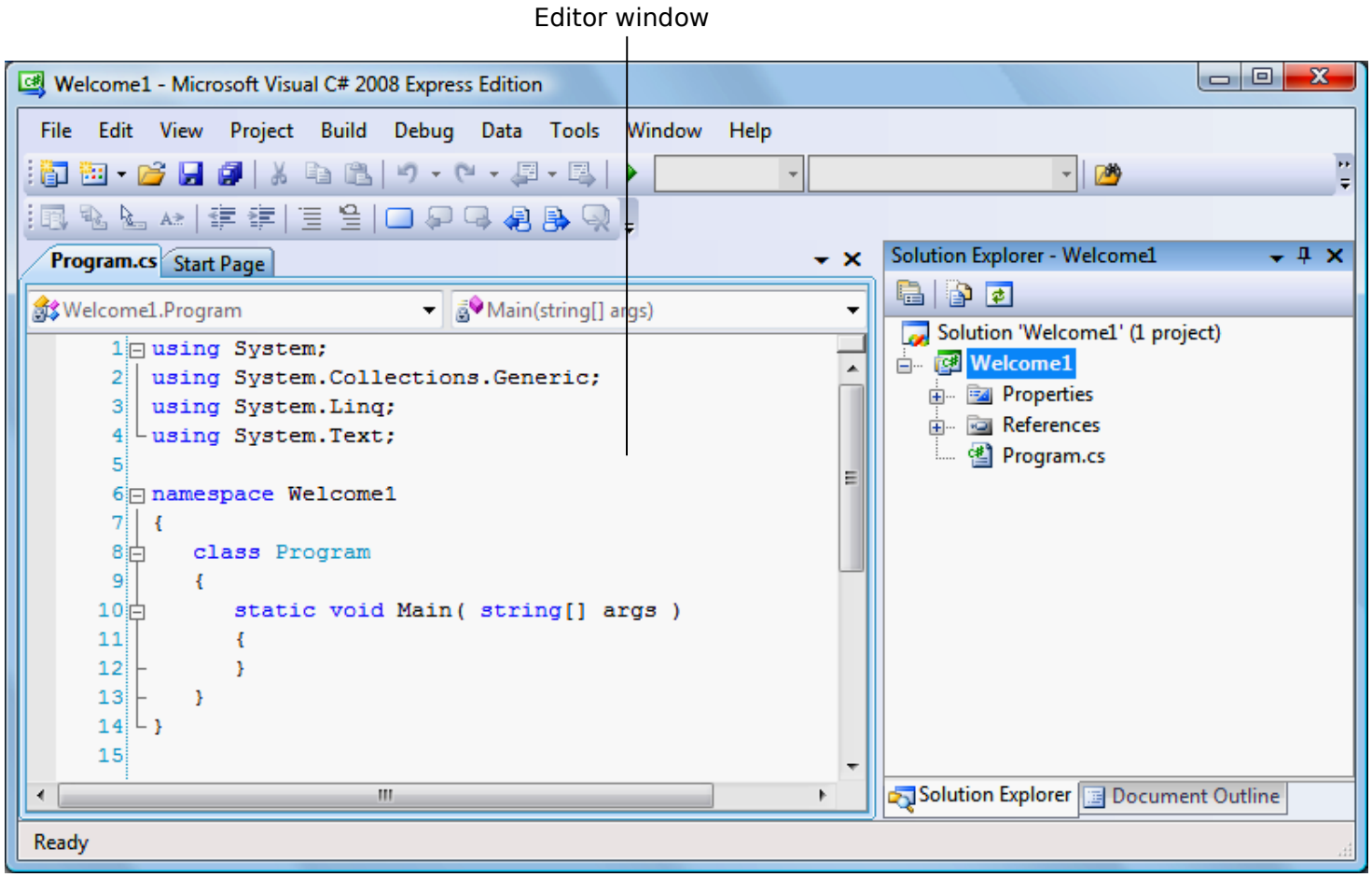

Fig. 3.4 | IDE with an open console application.

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.)

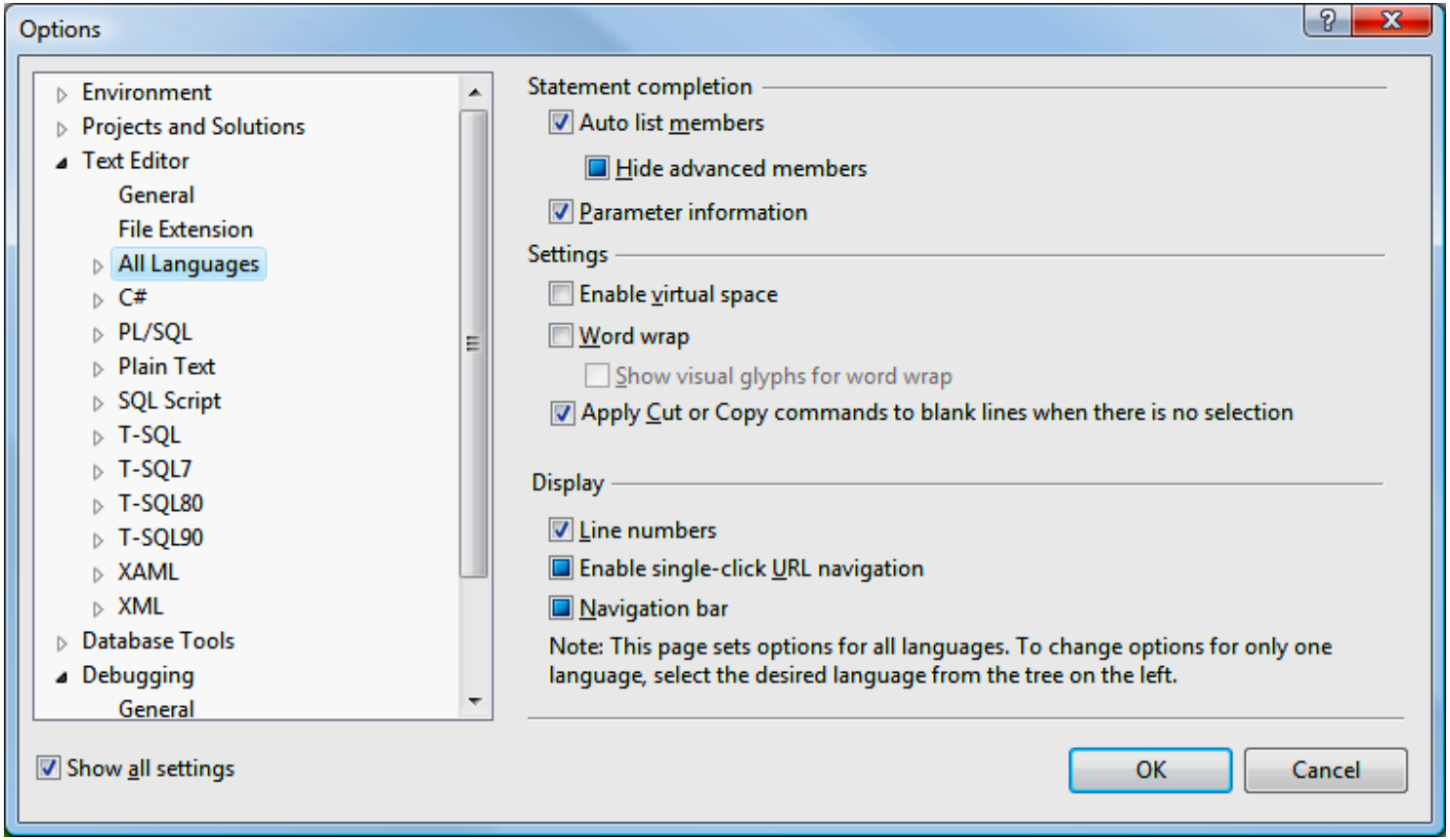

Fig. 3.5 | Modifying the IDE settings.

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.)

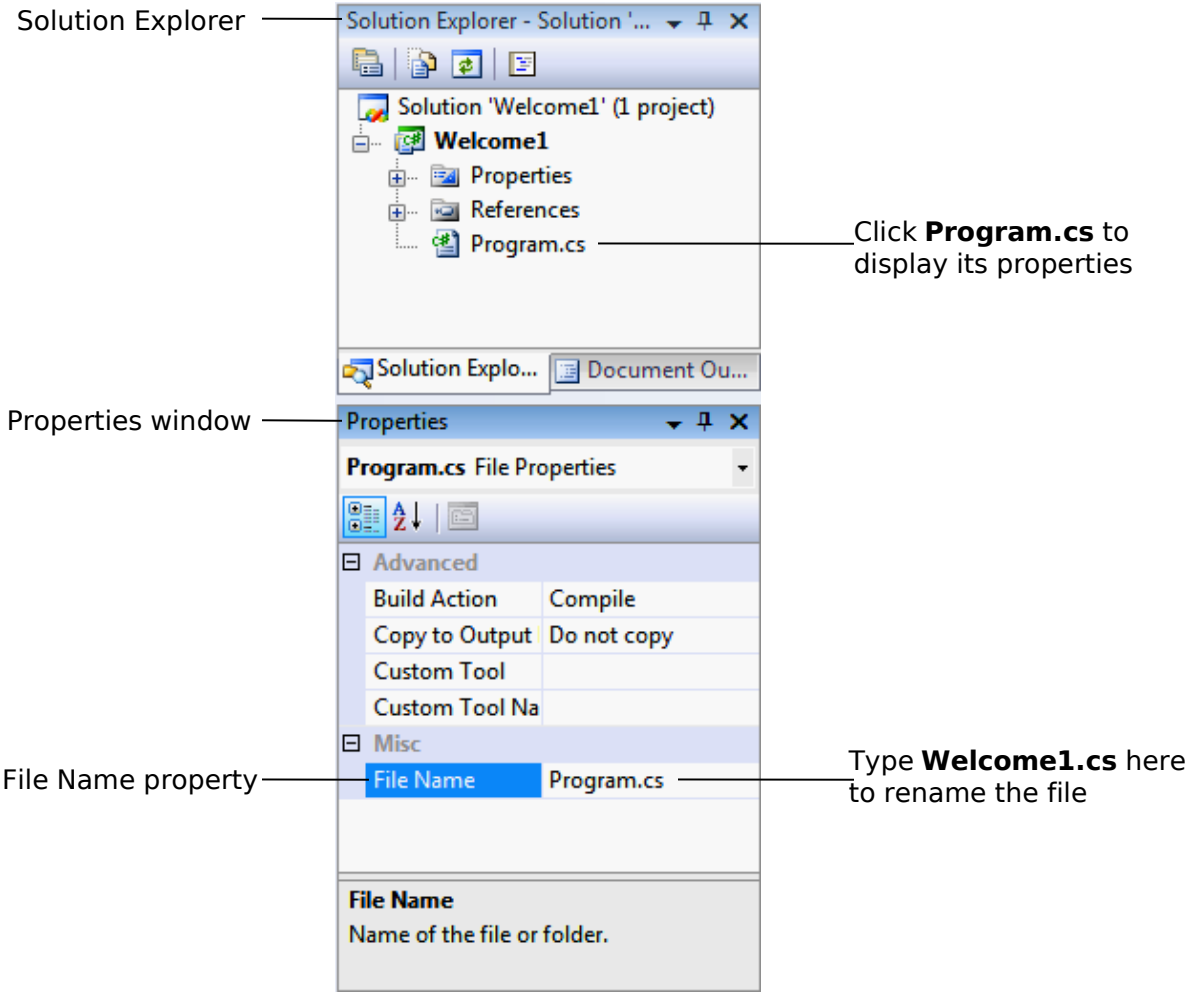

Fig. 3.6 | Renaming the program file in the **Properties** window.

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.)

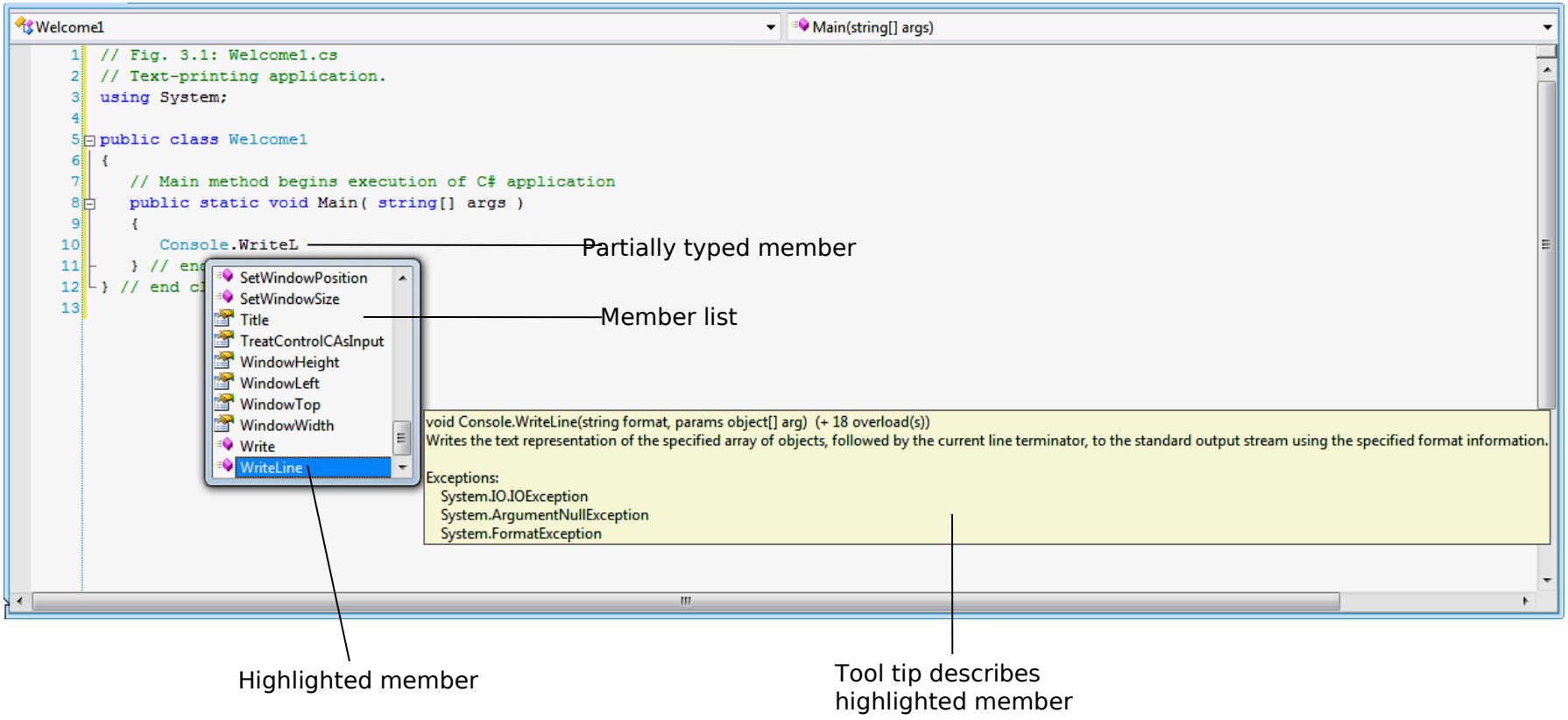

Fig. 3.7 | *IntelliSense* feature of Visual C# Express.

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.

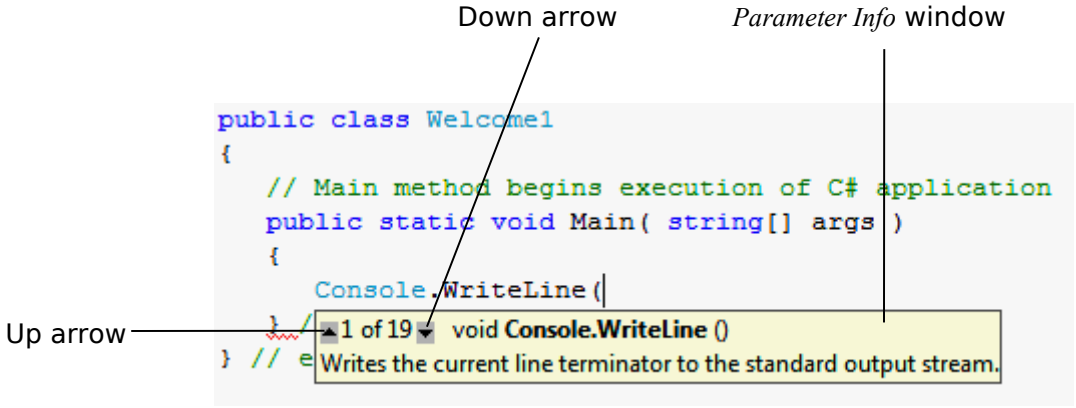

Fig. 3.8 | *Parameter Info* window.

- 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.)

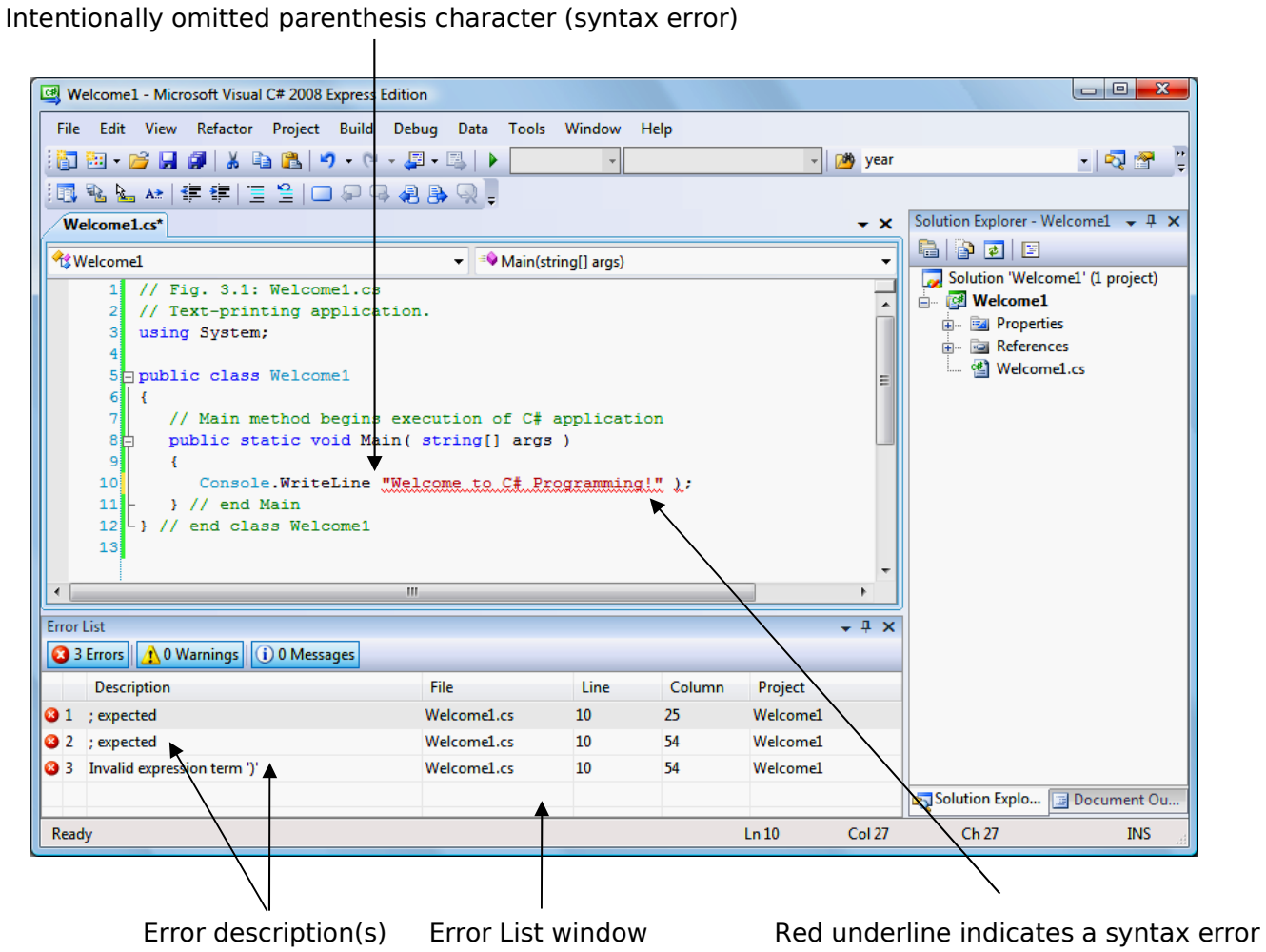

Fig. 3.13 | Syntax errors indicated by the IDE.

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

- To save an application, select **File > Save All** to display the **Save Project** dialog (Fig. 3.9).
- In the **Location** textbox, specify the directory where you want to save this project.
- Select the **Create directory for solution** checkbox and click **Save**.

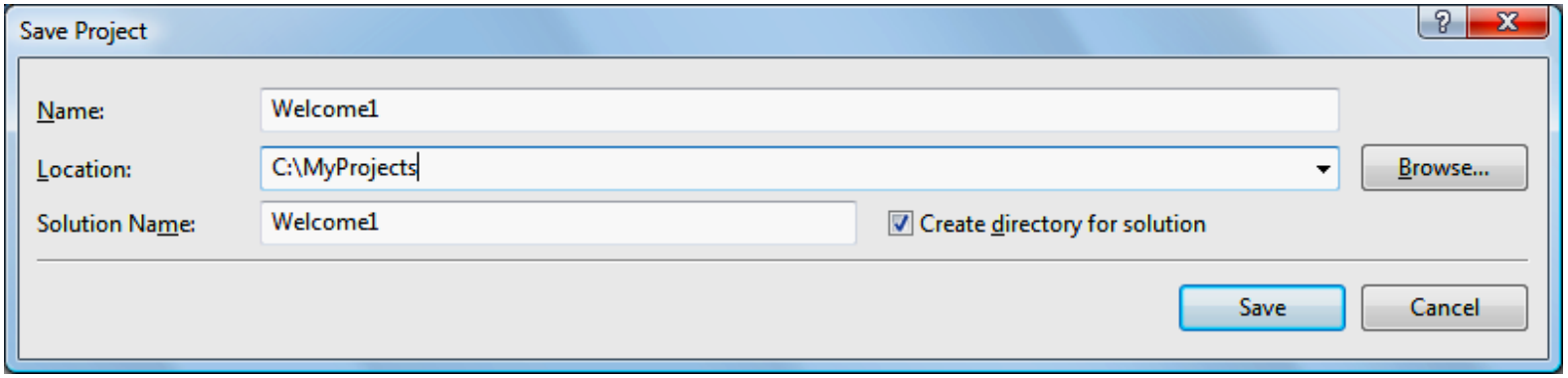

Fig. 3.9 | **Save Project** dialog.

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.)

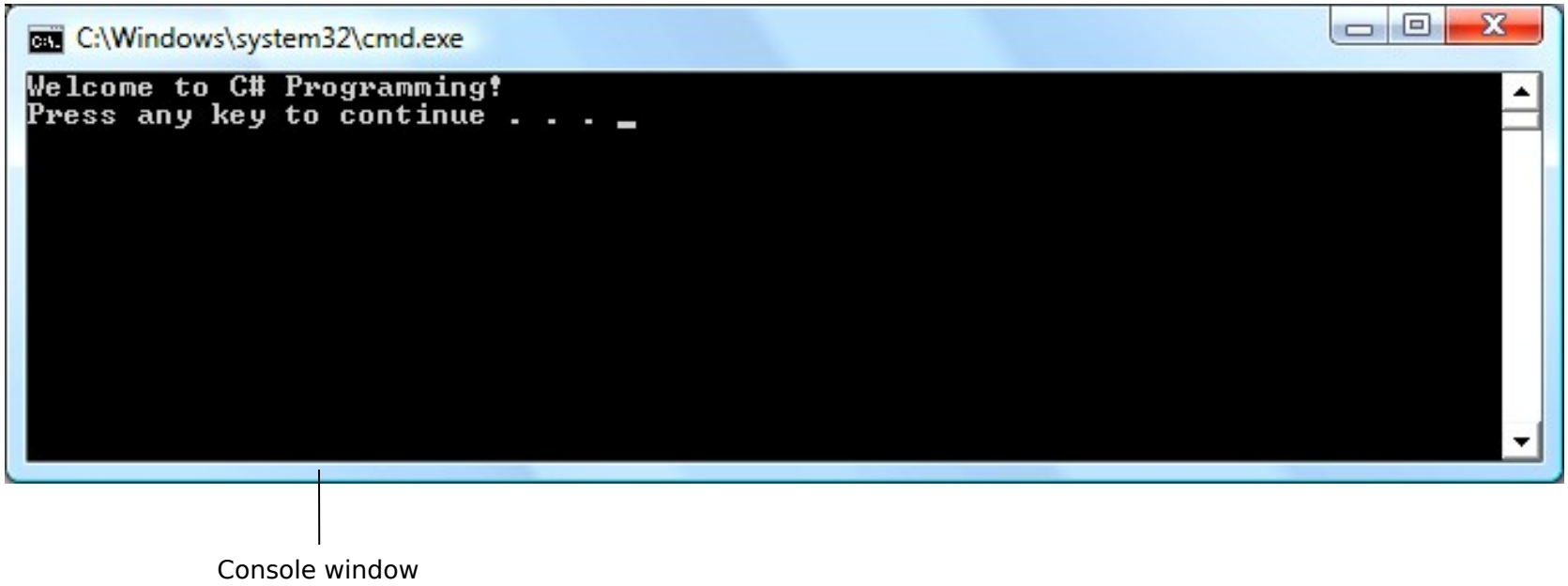

Fig. 3.10 | Executing the application shown in Fig. 3.1.

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.

- 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).

Outline

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.)

- 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.

3.8 Arithmetic

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

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 $\frac{x}{y}$ or $x \div y$	x / y
Remainder	%	$r \bmod s$	v % u

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 `float`s, `double`s, and `decimal`s.
- 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)
Evaluated first		
*	Multiplication	If there are several operators of this type, they are evaluated from left to right.
/	Division	
%	Remainder	
Evaluated next		
+	Addition	If there are several operators of this type, they are evaluated from left to right.
-	Subtraction	

Fig. 3.23 | Precedence of arithmetic operators.

3.9 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
Equality operators			
=	==	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
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. 3.25 | Equality and relational operators. (Part 2 of 2.)

Outline

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

`Comparison.cs`

(1 of 3)

```
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
```

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

Outline

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).

Outline

```
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.)

- 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.

4

Introduction to Classes and Objects

4.3 Declaring a Class with a Method and Instantiating an Object of a Class (Cont.)

- Keyword `public` is an **access modifier**.
  - Access modifiers determine the accessibility of properties and methods.
- The class's body is enclosed in a pair of left and right braces (`{` and `}`).

4.3 Declaring a Class with a Method and Instantiating an Object of a Class (Cont.)

- The method declaration begins with `public` to indicate that the method can be called from outside the class declaration's body.
- Keyword `void`—known as the method's return type—indicates that this method will not return information to its calling method.
- When a method specifies a return type other than `void`, the method returns a result to its calling method.

```
int result = Square( 2 );
```

- The body of a method contains statement(s) that perform the method's task.

4.3 Declaring a Class with a Method and Instantiating an Object of a Class (Cont.)

- To add a class, right click the project name in the **Solution Explorer** and select **Add > New Item...**.
- In the **Add New Item** dialog, select **Code File** and enter the name of your new file.

Outline

- The `GradeBookTest` class declaration (Fig. 4.2) contains the `Main` method that controls our application's execution.

GradeBookTest.cs

```
1  // Fig4.2: GradeBookTest.cs
2  // Create a GradeBook object and call its DisplayMessage method.
3  public class GradeBookTest
4  {
5     // Main method begins program execution
6     public static void Main( string[] args )
7     {
8        // create a GradeBook object and assign it to myGradeBook
9        GradeBook myGradeBook = new GradeBook();
10
11       // call myGradeBook's DisplayMessage method
12       myGradeBook.DisplayMessage();
13    } // end Main
14 } // end class GradeBookTest
```

```
Welcome to the Grade Book!
```

Object creation expression (constructor).

Using the object created in line 9.

Fig. 4.2 | Create a `GradeBook` object and call its `DisplayMessage` method.

4.3 Declaring a Class with a Method and Instantiating an Object of a Class (Cont.)

- Any class that contains a `Main` method can be used to execute an application.
- A `static` method can be called without creating an object of the class.

4.4 Declaring a Method with a Parameter

- A method can specify parameters, additional information required to perform its task.
- A method call supplies values—called arguments—for each of the method's parameters.
- For example, the `Console.WriteLine` method requires an argument that specifies the data to be displayed in a console window.

Outline

- Class `GradeBook` (Fig. 4.4) with a `DisplayMessage` method that displays the course name as part of the welcome message.

GradeBook.cs

```
1  // Fig4.4: GradeBook.cs
2  // Class declaration with a method that has a parameter.
3  using System;
4
5  public class GradeBook
6  {
7     // display a welcome message to the GradeBook user
8     public void DisplayMessage( string courseName )
9     {
10       Console.WriteLine( "Welcome to the grade book for\n{0}!"
11          courseName );
12    } // end method DisplayMessage
13 } // end class GradeBook
```

Indicating that the application uses classes in the `System` namespace.

`DisplayMessage` now requires a parameter that represents the course name.

Fig. 4.4 | Class declaration with a method that has a parameter.

Outline

- The new class is used from the `Main` method of class `GradeBookTest` (Fig. 4.5).

GradeBookTest.cs

(1 of 2)

```
1  // Fig4.5: GradeBookTest.cs
2  // Create a GradeBook object and pass a string to
3  // its DisplayMessage method.
4  using System;
5
6  public class GradeBookTest
7  {
8     // Main method begins program execution
9     public static void Main( string[] args )
10    {
11       // create a GradeBook object and assign it to myGradeBook
12       GradeBook myGradeBook = new GradeBook();
13
14       // prompt for and input course name
15       Console.WriteLine( "Please enter the course name:" );
16       string nameOfCourse = Console.ReadLine(); // read a line of text
17       Console.WriteLine(); // output a blank line
```

Creating an object of class `GradeBook` and assigns it to variable `myGradeBook`.

Prompting the user to enter a course name.

Reading the name from the user.

Fig. 4.5 | Create `GradeBook` object and pass a string to its `DisplayMessage` method. (Part 1 of 2).

Outline

GradeBookTest.cs

(2 of 2)

```
18
19          // call myGradeBook's DisplayMessage method
20          // and pass nameOfCourse as an argument
21          myGradeBook.DisplayMessage( nameOfCourse );
22       } // end Main
23    } // end class GradeBookTest
```

```
Please enter the course name:
CS101 Introduction to C# Programming

Welcome to the grade book for
CS101 Introduction to C# Programming!
```

Calling `myGradeBook`'s `DisplayMessage` method and passing `nameOfCourse` to the method.

Fig. 4.5 | Create `GradeBook` object and pass a string to its `DisplayMessage` method. (Part 2 of 2).

4.4 Declaring a Method with a Parameter (Cont.)

- Classes in the same project are considered to be in the same namespace.
- `using` indicates that the application uses classes in another namespace.
- Without `using`, we would write the **fully qualified class name**:

```
System.Console.WriteLine( "Please enter the course
  name:" );
```

4.5 Instance Variables and Properties

- Variables declared in the body of a method are known as **local variables**.
- When a method terminates, the values of its local variables are lost.
- Attributes are represented as variables in a class declaration.
- When each object of a class maintains its own copy of an attribute, the field is known as an instance variable.

Outline

- Class `GradeBook` (Fig. 4.7) maintains the course name as an instance variable so that it can be used or modified.

`GradeBook.cs`

(1 of 2)

```
1  // Fig4.7: GradeBook.cs
2  // GradeBook class that contains a courseName instance variable,
3  // and a property to get and set its value.
4  using System;
5
6  public class GradeBook
7  {
8     private string courseName; // course name for this GradeBook
9
10    // property to get and set the course name
```

Declaring `courseName` as an instance variable.

Fig. 4.7 | `GradeBook` class that contains a `private` instance variable, `courseName` and a `public` property to `get` and `set` its value. (Part 1 of 2).

```
17    psetic string CourseName
18    {
19       courseName = value;
20    } // end set
21 } // end property CourseName
22
23 // display a welcome message to the GradeBook user
24 public void DisplayMessage()
25 {
26    // use property CourseName to get the
27    // name of the course that this GradeBook represents
28    Console.WriteLine( "Welcome to the grade book for\n{0}!",
29       CourseName ); // display property CourseName
30 } // end method DisplayMessage
31 } //  end class GradeBook
```

GradeBook.cs

(2 of 2)

A `public` property declaration.

Fig. 4.7 | `GradeBook` class that contains a `private` instance variable, `courseName` and a `public` property to `get` and `set` its value. (Part 2 of 2).

4.5 Instance Variables and Properties (Cont.)

- Variables, properties or methods declared with access modifier `private` are accessible only within the class in which they are declared.

- Declaring instance variables with access modifier `private` is known as **information hiding**.

Outline

- Class `GradeBook` (Fig. 4.4) with a `DisplayMessage` method that displays the course name as part of the welcome message.

`GradeBook.cs`

```
1  // Fig4.4: GradeBook.cs
2  // Class declaration with a method that has a parameter.
3  using System;
4
5  public class GradeBook
6  {
7     // display a welcome message to the GradeBook user
8     public void DisplayMessage( string courseName )
9     {
10       Console.WriteLine( "Welcome to the grade book for\n{0}!"
11          courseName );
12    } // end method DisplayMessage
13 } // end class GradeBook
```

Indicating that the application uses classes in the `System` namespace.

`DisplayMessage` now requires a parameter that represents the course name.

Fig. 4.4 | Class declaration with a method that has a parameter.

Outline

- The new class is used from the `Main` method of class `GradeBookTest` (Fig. 4.5).

GradeBookTest.cs

(1 of 2)

```
1  // Fig4.5: GradeBookTest.cs
2  // Create a GradeBook object and pass a string to
3  // its DisplayMessage method.
4  using System;
5
6  public class GradeBookTest
7  {
8     // Main method begins program execution
9     public static void Main( string[] args )
10    {
11       // create a GradeBook object and assign it to myGradeBook
12       GradeBook myGradeBook = new GradeBook();
13
14       // prompt for and input course name
15       Console.WriteLine( "Please enter the course name:" );
16       string nameOfCourse = Console.ReadLine(); // read a line of text
17       Console.WriteLine(); // output a blank line
```

Creating an object of class `GradeBook` and assigns it to variable `myGradeBook`.

Prompting the user to enter a course name.

Reading the name from the user.

Fig. 4.5 | Create `GradeBook` object and pass a string to its `DisplayMessage` method. (Part 1 of 2).

Outline

GradeBookTest.cs

(2 of 2)

```
18
19          // call myGradeBook's DisplayMessage method
20          // and pass nameOfCourse as an argument
21       myGradeBook.DisplayMessage( nameOfCourse );
22       }// end Main
23    } // end class GradeBookTest
```

```
Please enter the course name:
CS101 Introduction to C# Programming

Welcome to the grade book for
CS101 Introduction to C# Programming!
```

Calling `myGradeBook`'s `DisplayMessage` method and passing `nameOfCourse` to the method.

Fig. 4.5 | Create `GradeBook` object and pass a string to its `DisplayMessage` method. (Part 2 of 2).

4.5 Instance Variables and Properties (Cont.)

- The `set` accessor begins with the identifier `set` and is delimited by braces.

```
gradeBook.CourseName = "CS100 Introduction to Computers";
```

- The text `"CS100 Introduction to Computers"` is assigned to the `set` accessor's keyword named `value` and the `set` accessor executes.
- A `set` accessor does not return any data.

Outline

- Class `GradeBookTest` (Fig. 4.8) creates a `GradeBook` object and demonstrates property `CourseName`.

`GradeBookTest.cs`

(1 of 2)

```
1  // Fig4.8: GradeBookTest.cs
2  // Create and manipulate a GradeBook object.
3  using System;
4
5  public class GradeBookTest
6  {
7     // Main method begins program execution
8     public static void Main( string[] args )
9     {
10       // create a GradeBook object and assign it to myGradeBook
11       GradeBook myGradeBook = new GradeBook();
12
13       // display initial value of CourseName
14       Console.WriteLine( "Initial course name is: '{0}'\n"
15          myGradeBook.CourseName );
16
```

Creating a `GradeBook` object and assigning it to local variable `myGradeBook`.

A `public` property declaration.

Fig. 4.8 | Create and manipulate a `GradeBook` object. (Part 1 of 2).

Outline

GradeBookTest.cs

(2 of 2)

```
17         // prompt for and read course name
18         Console.WriteLine( "Please enter the course name:" );
19      myGradeBook.CourseName = Console.ReadLine(); // set CourseName
20      Console.WriteLine(); // output a blank line
21
22         // display welcome message after specifying course name
23      myGradeBook.DisplayMessage();
24   } // end Main
25 } // end class GradeBookTest
```

Assigns the input course name to myGradeBook's CourseName property.

Calling DisplayMessage for a welcome message.

```
Initial course name is: ''

Please enter the course name:
CS101 Introduction to C# Programming

Welcome to the grade book for
CS101 Introduction to C# Programming!
```

Fig. 4.8 | Create and manipulate a `GradeBook` object. (Part 2 of 2).

4.5 Instance Variables and Properties (Cont.)

- Unlike local variables, every instance variable has a **default initial value**.
- The default value for an instance variable of type `string` is `null`.
- When you display a `string` variable that contains the value `null`, no text is displayed.

4.8 Auto-implemented Properties

- Notice- that `CourseName`'s `get` accessor simply returns `courseName`'s value and the `set` accessor simply assigns a value to the instance variable.
- For such cases, C# now provides **automatically implemented properties**.
- If you later decide to implement other logic in the `get` or `set` accessors, you can simply reimplement the property.

Outline

- Figure 4.10 redefines class `GradeBook` with an auto-implemented `CourseName` property.

GradeBook.cs

```
1  // Fig4.10: GradeBook.cs
2  // GradeBook class with an auto-implemented property.
3  using System;
4
5  public class GradeBook
6  {
7     // auto-implemented property CourseName implicitly creates
8     // an instance variable for this GradeBook's course name
9     public string CourseName { get; set; }
10
11    // display a welcome message to the GradeBook user
12    public void DisplayMessage()
13    {
14       // use auto-implemented property CourseName to get the
15       // name of the course that this GradeBook represents
16       Console.WriteLine( "Welcome to the grade book for\n{0}!",
17          CourseName ); // display auto-implemented property CourseName
18    } // end method DisplayMessage
19 } // end class GradeBook
```

Declaring the auto-implemented property.

Implicitly obtaining the property's value.

Fig. 4.10 | GradeBook class with an auto-implemented property.

Outline

- The unchanged test program (Fig. 4.11) shows that the auto-implemented property works identically.

GradeBookTest.cs

(1 of 2)

```
1  // Fig4.11: GradeBookTest.cs
2  // Create and manipulate a GradeBook object.
3  using System;
4
5  public class GradeBookTest
6  {
7     // Main method begins program execution
8    public static void Main( string[] args )
9    {
10       // create a GradeBook object and assign it to myGradeBook
11     GradeBook myGradeBook = new GradeBook();
12
13       // display initial value of CourseName
14     Console.WriteLine( "Initial course name is: '{0}'\n"
15       myGradeBook.CourseName );
16
```

Fig. 4.11 | Create and manipulate a `GradeBook` object. (Part 1 of 2).

Outline

GradeBookTest.cs

(2 of 2)

```
17         // prompt for and read course name
18         Console.WriteLine( "Please enter the course name:" );
19      myGradeBook.CourseName = Console.ReadLine(); // set CourseName
20      Console.WriteLine(); // output a blank line
21
22         // display welcome message after specifying course name
23      myGradeBook.DisplayMessage();
24   } // end Main
25 } // end class GradeBookTest
```

```
Initial course name is: ''

Please enter the course name:
CS101 Introduction to C# Programming

Welcome to the grade book for
CS101 Introduction to C# Programming!
```

Fig. 4.11 | Create and manipulate a `GradeBook` object. (Part 2 of 2).

4.10 Initializing Objects with Constructors

- Each class can provide a **constructor** to initialize an object of a class when the object is created.
- The `new` operator calls the class's constructor to perform the initialization.
- The compiler provides a **`public` default constructor** with no parameters, so *every* class has a constructor.

4.10 Initializing Objects with Constructors (Cont.)

- When you declare a class, you can provide your own constructor to specify custom initialization:

```
GradeBook myGradeBook =
    new GradeBook( "CS101 Introduction to C#
    Programming" );
```

- `"CS101 Introduction to C# Programming"` is passed to the constructor.

Outline

- Figure 4.14 contains a modified `GradeBook` class with a custom constructor.

`GradeBook.cs`

(1 of 2)

```
1  // Fig4.14: GradeBook.cs
2  // GradeBook class with a constructor to initialize the course name.
3  using System;
4
5  public class GradeBook
6  {
7     // auto-implemented property CourseName implicitly created an
8     // instance variable for this GradeBook's course name
9     public string CourseName { get; set; }
10
11    // constructor initializes auto-implemented property
12    // CourseName with string supplied as argument
13    public GradeBook( string name )
14    {
15       CourseName = name; // set CourseName to name
16    } // end constructor
17
```

Declaring the constructor for class `GradeBook`.

Fig. 4.14 | `GradeBook` class with a constructor to initialize the course name. (Part 1 of 2).

Outline

GradeBook.cs

(2 of 2)

```
18    // display a welcome message to the GradeBook user
19    public void DisplayMessage()
20    {
21       // use auto-implemented property CourseName to get the
22       // name of the course that this GradeBook represents
23       Console.WriteLine( "Welcome to the grade book for\n{0}!"
24          CourseName );
25    } // end method DisplayMessage
26 } // end class GradeBook
```

Fig. 4.14 | GradeBook class with a constructor to initialize the course name. (Part 2 of 2).

4.10 Initializing Objects with Constructors (Cont.)

- A constructor must have the same name as its class.
- Like a method, a constructor has a parameter list.

Outline

- Figure 4.15 demonstrates initializing `GradeBook` objects using the constructor.

`GradeBookTest.cs`

(1 of 2)

```
1  // Fig4.15: GradeBookTest.cs
2  // GradeBook constructor used to specify the course name at the
3  // time each GradeBook object is created.
4  using System;
5
6  public class GradeBookTest
7  {
8     // Main method begins program execution
9     public static void Main( string[] args )
10    {
11       // create GradeBook object
12       GradeBook gradeBook1 = new GradeBook( // invokes constructor
13          "CS101 Introduction to C# Programming" );
14       GradeBook gradeBook2 = new GradeBook( // invokes constructor
15          "CS102 Data Structures in C#" );
16
```

Creating and initializing `GradeBook` objects.

Fig. 4.15 | GradeBook constructor used to specify the course name at the time each GradeBook object is created. (Part 1 of 2).

Outline

GradeBookTest.cs

(2 of 2)

```
17         // display initial value of courseName for each GradeBook
18         Console.WriteLine( "gradeBook1 course name is: {0}",
19            gradeBook1.CourseName );
20         Console.WriteLine( "gradeBook2 course name is: {0}",
21            gradeBook2.CourseName );
22      } // end Main
23   } // end class GradeBookTest
```

```
gradeBook1 course name is: CS101 Introduction to C# Programming
gradeBook2 course name is: CS102 Data Structures in C#
```

Fig. 4.15 | GradeBook constructor used to specify the course name at the time each GradeBook object is created. (Part 2 of 2).