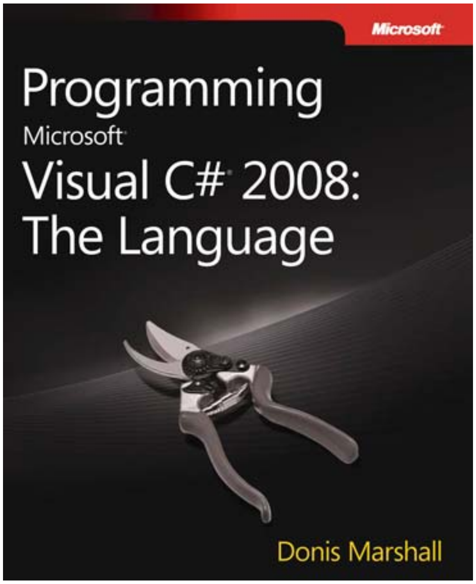


Programming Microsoft® Visual C#® 2008: The Language

Donis Marshall

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/12283.aspx

9780735625402

Microsoft Press

Table of Contents

Part I Core Language

Part II Core Skills

Part III More C# Language

Part IV Debugging

16 Advanced Debugging 593

Part V Advanced Features

Chapter 1

Introduction to Microsoft Visual C# Programming

Microsoft Visual Studio 2008, known during development as Orcas, is the successor to Microsoft Visual Studio 2005. The launch of the latest version of Visual Studio coincides with the release of Visual C# 2008, .NET Framework 3.5, and ASP.NET 3.5. Microsoft continues to shift Visual Studio from simply a comprehensive developer tool to a solution for the software lifecycle. This includes an Integrated Development Environment (IDE), components (testing tools, code analysis, and more), and tools for software design, development, testing, quality assurance, and debugging.

Anders Hejlsberg, technical fellow and chief architect of the C# language at Microsoft, has discussed Visual C# 2008 on numerous occasions. He highlights Language Integrated Query (LINQ) and related enhancements as the primary new features in Visual C# 2008. LINQ is a unified query model that is object-oriented and integrated into the C# language. New features, such as lambda expressions, extension methods, expression trees, implicitly typed local variables and objects, and other new features are useful in isolation, but they also extend the language to support LINQ. These and other changes are sure to secure the stature of C# as the preeminent development language for .NET.

With LINQ, Visual C# 2008 changes the relationship between developers and data. LINQ is an elegant solution for accessing specific data using a query language that is independent of the data source. LINQ is also object-oriented and extensible. LINQ moves C# a measure closer to functional programming, it refocuses developers from managing the nuts and bolts of data (state) to behavior of information (objects), and it provides a unified model for querying data that no longer depends on the vagaries of a specific language or technology. With LINQ, you can access with the same unified query model a Structured Query Language (SQL) database, an Extensible Markup Language (XML) file, or even an array.

LINQ removes the distinction between data and objects. Traditional queries are strings that are not entirely type-safe and return a vector of information, which defines the disconnect between data and objects (objects being type-safe, supporting IntelliSense, and not necessarily rectangular). LINQ forms an abstraction layer that allows developers to treat data as objects and focus on solutions in a unified manner without sacrificing specificity.

Visual Studio 2008 epitomizes the term *unified*. For example, Visual Studio 2008 provides a unified environment for developing to different .NET targets. In addition, the Visual Studio Designer provides a unified canvas where Microsoft Windows Forms, Extensible Application Markup Language (XAML), and Windows XP and Windows Vista visually themed applications can be designed and implemented. Visual Studio C# 2008 has additional capabilities for creating enterprise, distributed, or Web-based applications—most notably with Microsoft Silverlight. Silverlight is a plug-in that provides a unified environment for building Web applications with cross-browser support and that promotes enhanced rich interactive applications (RIA).

Visual Studio 2008, with ASP.NET 3.5, further separates design and implementation responsibilities, allowing for clearer separation of designer and developer roles.

The trend towards collaborative and team development continues. No developer can be completely self-reliant, and Visual Studio 2008 has additional features that benefit a wide variety of teams ranging from small to large in size. Visual Studio 2008 also recognizes the important role of everyone—not just developers but others on the software team. For example, the new database project separates language developer and database developer/architect roles.

As mentioned, Visual Studio 2008 encompasses the entire life cycle of a software application. The product boasts new revisions of tools for testing and quality assurance. Many of these tools were introduced in Visual Studio 2005. Furthermore, testing tools have been extended to the Professional revision of the product, making them available to more developers and not reserved for Microsoft Visual Studio Team editions. Both developers and non-developers, such as quality assurance staff, can use the testing tools that have been incorporated into Visual Studio 2008. The testing tools are easy to use but comprehensive, and performance also has been improved. Finally, the testing tools are integrated better, including seamless access to unit testing in the IDE. Chapter 17, “Testing,” reviews Visual Studio 2008 testing tools.

.NET Framework 3.5 and the Common Language Runtime (CLR) platform continues to provide important services for managed applications, such as memory management, security services, type-safety, and structured exception handling. The .NET Framework now supports LINQ and other new features. Some favorite new elements of the .NET Framework include (but are not limited to) Active Directory APIs (System.DirectoryServices.AccountManagement.dll), ASP.NET AJAX (System.Web.Extensions), Peer-To-Peer (P2P) support (System.NET.dll), STL to CLR for Managed C++ developers (System.VisualC.STLCLR.dll), Window Presentation Foundation (WPF; System.Windows.Presentation.dll), and Windows Communication Foundation (System.WorkflowServices.dll). .NET Framework 3.5 offers numerous other enhancements, including improved network layers and better performing sockets.

Visual C# 2008 is a modern, object-oriented, and type-safe programming language. C# has its roots in the C family of languages and will be immediately comfortable to C, C++, and Java programmers. The ECMA-334 standard and ISO/IEC 23270 standard apply to the C# language. Microsoft’s C# compiler for the .NET Framework is a conforming implementation of both of these standards.

A Demonstration of Visual C# 2008

To introduce programming in Visual C# 2008, the following code examples are presented with explanations. Some of the programming concepts given in this section are explained in depth throughout the remainder of the book.

Sample C# Program

In deference to Martin Richards, the creator of the Basic Combined Programming Language (BCPL) and author of the first “Hello, World!” program, I present a “Hello, World!” program. Actually, I offer an enhanced version that displays “Hello, World!” in English, Italian, or Spanish. The program is a console application.

Here is my version of the “Hello, World!” program, which is stored in a file named hello.cs:

```
using System;

namespace HelloNamespace {

    class Greetings{
        public static void DisplayEnglish() {
            Console.WriteLine("Hello, world!");
        }
        public static void DisplayItalian() {
            Console.WriteLine("Ciao, mondo!");
        }
        public static void DisplaySpanish() {
            Console.WriteLine("Hola, imundo!");
        }
    }

    delegate void delGreeting();

    class HelloWorld {
        static void Main(string [] args) {
            try {
                int iChoice = int.Parse(args[0]);
                delGreeting [] arrayofGreetings={
                    new delGreeting(Greetings.DisplayEnglish),
                    new delGreeting(Greetings.DisplayItalian),
                    new delGreeting(Greetings.DisplaySpanish)};

                arrayofGreetings[iChoice - 1]();
            }
            catch(Exception ex) {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

Csc.exe is the C# compiler. Enter the following **csc** command at the Visual Studio command prompt to compile the hello.cs source file and create the executable hello.exe:

```
csc hello.cs
```

The hello.exe file is a .NET single-file assembly. The assembly contains metadata and Microsoft Intermediate Language (MSIL) code but not native binary. Mixed assemblies (both native and managed) may contain binary.

Run the Hello application from the command line. Enter the program name and the language (**1** for English, **2** for Italian, or **3** for Spanish). For example, the following command line displays “Ciao, mondo!” (“Hello, world!” in Italian).

```
Hello 2
```

The source code of the Hello application highlights the common elements of most .NET applications: using statement directive, a namespace, types, access modifiers, methods, exception handling, and data.

> **Note** C# is case-sensitive.

The *HelloNamespace* namespace contains the *Greetings* and *HelloWorld* types. The *Greetings* class has three static methods, and each method displays “Hello, World!” in a different natural language. Static methods are invoked on the type (*classname.member*), not an instance of that type. The static methods of the *Greetings* type are also public and therefore visible inside and outside the class.

Delegates define a type of function pointer. The *delGreeting* delegate is a container for function pointers. A *delGreeting* delegate points to functions that return *void* and have no parameters. This is (not so coincidentally) the function signature of the methods in the *Greetings* type.

The entry point of this and any other C# executable is a *Main* method. Command-line parameters are passed as the *args* parameter, which is a string array. In the HelloWorld program, the first element of the *args* array is used as a number indicating the language of choice, as input by the user. The Hello application converts that element to an integer. Next, the program defines an array of function pointers, which is initialized to point to each of the methods of the *Greetings* class. The following statement invokes a function pointer to display the selected HelloWorld message:

```
arrayofGreetings[iChoice - 1]();
```

The value *iChoice - 1* is an index into the delegate array. Since arrays are zero-based in C#, *iChoice* is offset by -1.

Most of the code in the *Main* method is contained in a *try* block. The code in the *try* block is a guarded body. A guarded body is protected against exceptions defined in the corresponding catch filter. When an exception is raised that meets the criteria of the *catch* filter, execution is transferred to the *catch* block, which displays the exception message. If no exception is raised, the *catch* block is not executed. In our HelloWorld application, omitting the choice or entering a non-numeric command-line parameter when running the program will cause an exception, which is caught, and the *catch* block displays the appropriate message.

There are several statement blocks in the sample code. A statement block contains zero or more statements bracketed with curly braces {}. A single statement can be used separately or within a statement block. The following two code examples are equivalent:

```
// Example 1
if (bValue)
    Console.WriteLine ("information");

// Example 2
if (bValue)
{
    Console.WriteLine ("information");
}
```

Sample LINQ Program

Because LINQ plays such an important role in Visual C# 2008, here are two sample applications to demonstrate LINQ. Actually, the LINQ example comprises two examples: a traditional version and a LINQ version. Both examples list the names of people that are 30 years old or older. The traditional example filters the names with an *if* statement. The LINQ example uses a LINQ query to ascertain the correct names. The result of both applications is identical.

Here is the traditional example, stored in a file named people.cs:

```
using System;

namespace Example {

    class Person {
        public Person(string _name, int _age) {
            name = _name;
            age = _age;
        }
        public string name = "";
        public int age = 0;
    }
    class Startup {
        static void Main() {
            Person [] people = {new Person("John", 35),
                                new Person("Jill", 37),
                                new Person("Jack", 25),
                                new Person("Mary", 28)};
            foreach (Person p in people) {
                if (p.age >= 30) {
                    Console.WriteLine(p.name);
                }
            }
        }
    }
}
```

The code has a single namespace, which is *Example*. *Example* contains the *Person* and *Startup* types. The *Person* class has a public two-argument constructor. A constructor is a method with the same name as its class. This constructor is used to initialize instances of the *Person* class. The *Person* type contains two fields—name and age.

Main, the entry point function, is in the *Startup* class. In *Main,* an array of four employees is defined. The *foreach* statement iterates through the elements in the *people* array. Next, the *if* statement filters the employees and displays the names of people 30 years old or older in the console window.

The program is compiled with the C# compiler as follows:

```
csc people.cs
```

Here is the LINQ version of this example:

```
using System;
using System.Linq;

namespace Example {

    class Person {
        public Person(string _name, int _age) {
            name = _name;
            age = _age;
        }
        public string name = "";
        public int age = 0;
    }

    class Startup {
        static void Main() {
            Person [] people={new Person("John", 35),
                              new Person("Jill", 37),
                              new Person("Jack", 25),
                              new Person("Mary", 28)};
            var ageQuery = from p in people
                           where p.age >= 30
                           select p;
            foreach (var p in ageQuery) {
                Console.WriteLine(p.name);
            }

        }
    }
}
```

The LINQ version filters people using a LINQ query instead of an *if* statement. This highlights the seminal difference between a traditional query and LINQ. The *if* statement in the traditional version filters data, whereas the LINQ query in the LINQ example filters *Person* objects. In this way, LINQ removes the disconnect between objects and data.

The *var* keyword in the *ageQuery* variable declaration is for type inference. Type inference is not typeless. A specific type is inferred at compile time from the result of the LINQ query expression. This keeps the code type-safe, which is an important tenet of .NET. In our example, the query expression evaluates to an array of *Person* types. Therefore, at compile time, "*var query*" implies "*Person [] query*".

Common Elements in Visual C# 2008

The remainder of the chapter discusses the common elements of Visual C# 2008 programs.

Namespaces

Namespaces provide hierarchical clarity of classes within and across related assemblies. The .NET Framework Class Library (FCL) is an example of the effective use of namespaces. The FCL would sacrifice clarity if it were designed as a single namespace with a flat hierarchy. Instead, the FCL is organized using a main namespace (*System*) and several nested namespaces. *System*, which is the root namespace of the FCL, contains the classes ubiquitous to .NET, such as *Console*. Types related to LINQ are grouped in the *System.Linq* namespace. Other .NET services are similarly nested in .NET namespaces. For example, data services are found in the *System.Data* namespace and are further delineated in the *System.Data.SqlClient* namespace, which contains classes specific to Microsoft SQL.

A nested namespace is considered a member of the containing namespace. Use the dot punctuator (.) to access members of the namespace, including nested namespaces.

A namespace at file scope, not nested within another namespace, is considered part of the compilation unit and included in the global declaration space. (A compilation unit is a source code file. A program partitioned into several source files has multiple compilation units—one compilation unit for each source file.) Any namespace can span multiple compilation units.

For example, all namespaces defined at file scope are included in a single global declaration space that also spans separate source files.

The following code has two compilation units and three namespaces. *ClassB* is defined in the global declaration space of both compilation units, which is a conflict. *ClassC* is defined twice in *NamespaceZ,* which is another conflict. For these reasons, the following program will not compile.

The global declaration space has four members. *NamespaceY* and *NamespaceZ* are members. The classes *ClassA* and *ClassB* are also members of the global namespaces. The members span the File1.cs and File2.cs compilation units, which both contribute to the global namespace:

```
// file1.cs

public class ClassA {
}

public class ClassB {
}

namespace NamespaceZ {
   public class ClassC {
   }
}

// file2.cs

public class ClassB {
}

namespace NamespaceY {
    public class ClassA {
    }
}

namespace NamespaceZ {

    public class ClassC {
    }

    public class ClassD {
    }
}
```

Attempt to compile the above code into a library from the command line with this statement. You will receive compile errors because of the conflicts:

```
csc /t:library file1.cs file2.cs
```

The relationship between compilation units, the global namespace, and nonglobal namespaces are illustrated in Figure 1-1.

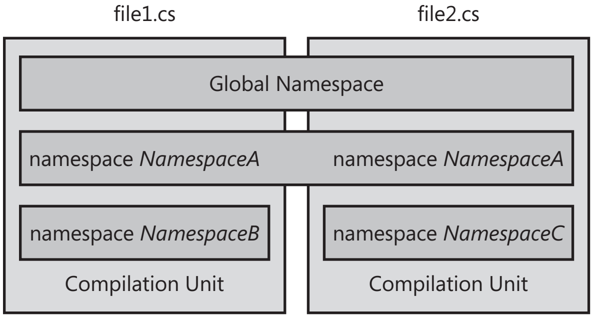


FIGURE 1-1 Global declaration space vs. namespaces

The *using* directive makes a namespace implicit. You then can access members of the named namespace directly without their fully qualified names. Do you refer to members of your family by their "fully qualified names" or just their first names? Unless your wife is the queen of England, you probably refer to her directly, simply using her first name. The *using* directive means that you can treat members of a namespace like family members.

The *using* directive must precede the first member within a namespace in a compilation unit. The following code defines the namespace member *ClassA*. The fully qualified name is *NamespaceZ.NamespaceY.ClassA*. Imagine having to type that several times in a program!

```
using System;

namespace NamespaceZ {
    namespace NamespaceY {
        class ClassA {
            public static void FunctionM() {
                Console.WriteLine("FunctionM");
            }
        }
    }
}

namespace Application {
    class Starter {
        public static void Main() {
            NamespaceZ.NamespaceY.ClassA.FunctionM();
        }
    }
}
```

The *using* directive in the following code makes *NamespaceZ.NamespaceY* implicit. Now you can directly access *ClassA* without further qualification:

```
namespace Application {
    using NamespaceZ.NamespaceY;
    class Starter {
        public static void Main() {
            ClassA.FunctionM();
        }
    }
}
```

Ambiguities can occur when separate namespaces with identically named members are made implicit. When this occurs, the affected members can be assessed only with their fully qualified names.

The *using* directive also can define an alias for a namespace or type. Aliases are typically created to resolve ambiguity or simply as a convenience. The scope of the alias is the space where it is declared. The alias must be unique within that space. In this source code, an alias is created for the fully qualified name of *ClassA*:

```
namespace Application {
    using A=NamespaceZ.NamespaceY.ClassA;
    class Starter {
        public static void Main() {
            A.FunctionM();
        }
    }
}
```

In this code, *A* is the alias and a nickname for *NamespaceZ.NamespaceY.ClassA* and can be used synonymously.

Using directive statements are not cumulative and are evaluated independently. Take the following example:

```
using System.Text;
```

The previous statement makes *System.Text* implicit but not the *System* namespace. The following code makes both namespaces implicit:

```
using System;

using System.Text;
```

The *extern alias* directive is an alias to another assembly. The resulting alias can be combined with a namespace to make an explicit reference to a namespace in a different assembly. Separate the alias and referenced assembly with two colons as follows:

```
extern alias::namespace
```

Here is sample code for a library that contains two namespaces, stored in a file named mylib.cs:

```
using System;

namespace ANamespace{
    namespace BNamespace {
        public class XClass {
            public static void MethodA() {
                Console.WriteLine("MyLib::XClass.MethodA");
            }
        }
    }
}
```

The following command will compile the source file into a Dynamic Link Library (DLL) assembly:

```
csc /t:library mylib.cs
```

Here is sample code, stored in a file named program.cs, for an executable that uses the DLL assembly. The *extern alias* statement resolves the ambiguity between the library and the executable:

```
extern alias MyLib;
using System;

namespace ANamespace{
    namespace BNamespace {
        class XClass {
            public static void MethodA() {
                Console.WriteLine("Program::XClass.MethodA");
            }
        }
    }
}

class Startup{
    public static void Main() {
        MyLib::ANamespace.BNamespace.XClass.MethodA();
    }
}
```

The following command will compile the program and define *MyLib* as an alias for mylib.dll:

```
csc program.cs /r:MyLib=mylib.dll
```

In the preceding code, the call to *XClass.MethodA* is not ambiguous because of the *extern* alias. Because of the alias, the call to *XClass.MethodA* executes the version in the library rather than the version in the current compilation unit.

Main Entry Point

Main is the entry point method for a C# application and a member function of a class or struct (the entry point method is where the C# application starts executing). There are four valid signatures for *Main* when being used as the entry point method:

```
static void Main() {
    // main block
}

static int Main() {
    // main block
}

static void Main(string [] args) {
    // main block
}

static int Main(string [] args) {
    //
}
```

A class or struct can contain only one entry point method. *Main* must be static and should be private, although that is not required. Naturally, a public *Main* method is accessible as an entry point method.

Application arguments are passed into the program as a string array parameter of the *Main* function. Arrays in .NET are instances of the *System.Array* class. You can use the properties and methods of *System.Array* to examine the application arguments, including the *Length* field to determine the number of arguments passed into *Main*. The command arguments start at element zero of the string array. When no arguments are passed, the *arg* parameter is non-null but the array length is zero.

The return value of an entry point method is cached internally for interprocess communication. If the application is part of a system of applications and spawned to complete a specific task, the return value could represent a status code or the result of that task. The default exit code of an application is zero. The exit code of a process is stored in the Process Environment Block (PEB) and is accessible through the *GetExitCodeProcess* application programming interface (API).

What if the entry point is ambiguous? Look at this code, stored in main.cs:

```
using System;

namespace Application{
    class StarterA{
        static void Main() {
        }
    }
    class StarterB{
        static void Main() {

        }
    }
}
```

This code has two valid entry points, which is inherently ambiguous. The compiler option *main* is available to designate the class name where the desired entry point method is found. The following command successfully compiles the previous program:

```
csc /main:Application.StarterB main.cs.
```

Local Variables

Local variables are local to a statement block. Local variables can be declared anywhere in the block, but they must be defined before use. Local variables can refer to either value or reference types. A value type is allocated storage on the stack, whereas reference types have memory allocated on the managed heap. Actually, the reference itself is on the stack, while the object being referenced is on the managed heap. Value types are types such as primitives, structures, and enumerations. The memory storage for value types is released deterministically when the variable is no longer in scope. Reference types are types such as user defined types, interfaces, strings, arrays, and pointers. They are always initialized with the new keyword and removed nondeterministically by the Garbage Collector, which is a component of the CLR. Value types can be initialized with a simple assignment and declared in an individual declaration or in a daisy-chain:

```
int variablea = 5, variableb, variablec = 10;
```

The scope and visibility of a local variable is the statement block, where it is declared, and any subsequent nested code blocks in the current statement block. This is called the *variable declaration space,* in which local variables must be uniquely declared.

In the following code, several local variables are defined. The storage for *variablea, variableb, variablec*, and *variabled* is released at the end of the function block when the variables are no longer within scope. However, the lifetime of *variablee*, a local variable and reference type, is managed by the Garbage Collector. It is generally good policy to set reference types to *null* when they are no longer needed*:*

```
void Function() {
    int variablea = 0;
    int variableb = 1,variablec, variabled = 4;
    const double PI = 3.1415;
    UserDefined variablee = new UserDefined();
    // function code

    variablee = null;
}
```

Nullable Types

In the previous code, a reference type is set to *null*. Assigning *null* to an object indicates that it is unused. This is consistent for all reference types. Can you similarly flag an integer as unused? How can you stipulate that a value type contains nothing? Nulls are not assignable to primitive value types like an integer or *char* (a compilation error would occur).

```
int variablea = null;  // compiler error
```

Setting an integer to –1 is a possible solution, assuming that this value is outside the range of expected values. However, this solution is non-portable, requires explicit documentation, and is not very extensible. Nullable types provide a consistent solution for setting a value type to *null*. This is especially important when manipulating data between C# and a database source, where primitives often contain *null* values.

Declare a nullable type by adding the *?* type modifier in the value type declaration. Here is an example:

```
double? variable1 = null;
```

The object *variable1* is a nullable type and the underlying type is double. A nullable type extends the interface of the underlying type. The *HasValue* and *Value* properties are added. Both properties are public and read-only. *HasValue* is a Boolean property, whereas the type of *Value* is the same as the underlying type. If the nullable type is assigned a non-null value, *HasValue* is true and the *Value* property is accessible. Otherwise, *HasValue* is false, and an exception is raised if the *Value* property is accessed. The acceptable range of values for a nullable type includes the *null* value and the limits of the underlying type.

The *null* coalescing operator (??) evaluates the value of a nullable type. The syntax is as follows:

```
variable ?? r_value
```

If the nullable type contains a value, the expression evaluates to that value. If the nullable type is empty (that is, it contains *null*), the expression evaluates to the *r_value* of the *null* coalescing operator. The stated *r-value* of the *null* coalescing operator must be the same type as the underlying type. The following code sets *variable2* to the value of *variable1* if *variable1* is not null and to zero otherwise:

```
double variable2 = variable1 ?? 0;
```

Here is another example of nullable types:

```
static void Main() {
    int? variablea = null;
    Console.WriteLine(variablea.HasValue); // false
    int variableb = variablea ?? 5;
    Console.WriteLine(variableb);  // 5
}
```

Expressions

Expressions resolve to a value. An expression commonly contains one or more operators. However, an expression also can be a single value or constant. Operators are unary, binary, or ternary.

With the exception of the assignment and ternary operators, expressions are evaluated from left to right. Expressions can contain multiple operators; operators are evaluated in order of precedence. Use parentheses to change the precedence or to clarify the desired precedence.

Table 1-1 lists the order of precedence.

TABLE 1-1 Order of precedence for expressions

Precedence	Operator
1	array '[ ]', *checked*, function '( )', member operator '.', *new*, postfix decrement, postfix increment, *typeof*, *default*, anonymous method, *delegate*, and *unchecked* operators
2	unary addition '+', casting '( )', one's complement '~', logical not '!', prefix decrement, prefix increment, and negation '-' operators
3	division '/', modulus '%', and multiplication '*' operators
4	binary addition '+' and binary subtraction '–' operators
5	left-shift '<<' and right-shift '>>' operators
6	*as*, *is*, less than '<', less than or equal to '<=', greater than '>', and greater than or equal to '>=' operators
7	equals '==' and not equal '!=' operators
8	Logical And '&' operator
9	Logical XOR '^' operator
10	Logical Or '|' operator
11	Conditional And '&&' operator
12	Conditional Or '||' operator
13	Null coalescing '??'operator
14	Conditional '?:' operator
15	Assignment '=', compound '*=, /=, %=, +=, –=, <<=, >>=, &=, ^=, and |=', and lambda operator '=>'

Selection Statements

A selection statement evaluates an expression to determine what code branch is executed next. Selection statements include *if* statements, *while* loops, *for* loops, and *goto* and *switch* statements.

An *if* statement evaluates a Boolean expression. If the expression is *true,* control is transferred to the next *true_statement*. If the expression is *false,* execution is transferred to the first statement after the *true_statement*.

Here is the syntax of the *if* statement:

```
if (Boolean_expression) true_statement
```

In the preceding code, the *true_statement* is executed when *Boolean_expression* is *true*. When combined with an *else* condition, the *if* statement has *true_statement* and *false_statement*. The *false_statement* immediately follows the *else* statement. When the *Boolean_expression* is true, you are transferred to the *true_statement*. If it is false, control is transferred to the *false_statement*. If nested, the *else* statement belongs to the nearest *if* statement.

Here is the syntax:

```
if (Boolean_expression)
     true_statement;
else
     false_statement;
```

An alternative to nested *if* and *else* statements is the *else if* clause, which is particularly useful in evaluating choices. The *else if* statement can be used along with an *else* statement.

The syntax appears here:

```
if (Boolean_expression_1)
     true_statement_1;
else if (Boolean_expression_2)
     true_statement_2;

...
else if (Boolean_expression_n)
     true_statement_n;
else
     false_statement;
```

This is an example of various *if* statements:

```
static void Main() {
    Console.WriteLine("Enter command:");
    string menuChoice=(Console.ReadLine()).ToLower();
    if (menuChoice == "a")
        Console.WriteLine("Doing Task A");
    else if (menuChoice == "b")
        Console.WriteLine("Doing Task B");
    else if (menuChoice == "c")
        Console.WriteLine("Doing Task C");
    else
        Console.WriteLine("Bad choice");
}
```

A *switch* statement is sometimes a better solution then an *if* statement. Within a *switch* statement, execution jumps to the case label that matches the *switch* expression. The *switch* expression must resolve to an integral*, char, enum,* or *string* type. The case label is a constant or literal and must have the same underlying type as the *switch* expression.

Here is the syntax for the *switch* statement:

```
switch (switch_expression)
{
    case label1:
        switch_statement1;
    case  label2:
        switch_statement2;
    default:
        default_statement;
}
```

A *switch* statement contains a *switch* expression and is followed by a *switch* block, which contains one or more *case* statements. Within the *switch* block, each *case* statement must evaluate to a unique label. After the *switch* expression is evaluated, control is transferred to the matching case label. The matching case has the same value as the *switch* expression. If no case label matches the *switch* expression, control is transferred to the *default* case statement or (if the *default* case statement is not present) to the next statement after the *switch* statement.

Unlike C and C++, cascading between *case* statements is not allowed—that is, you cannot "crash the party" of another *case* statement. Each case block must conclude with a transfer of control, such as *break*, *goto*, *return*, or *throw*. The exception is cases that have no statements, where *fallthrough* is allowed.

This is sample code for a *switch* statement:

```
static void Main() {
    Console.WriteLine("Enter command:");
    string resp = (Console.ReadLine()).ToLower();
    switch (resp) {
        case "a":
            Console.WriteLine("Doing Task A");
            break;
        case "b":
            Console.WriteLine("Doing Task B");
            break;
        case "c":
            Console.WriteLine("Doing Task C");
            break;
        default:
            Console.WriteLine("Bad choice");
            break;
    }
}
```

Any object, value, or reference type that is convertible to an integral, *char, enum,* or *string* type is acceptable as the *switch_expression*, which is demonstrated in the following code. You are allowed a one-step conversion to one of the acceptable types.

```
class Employee {
    public Employee(string f_Emplid) {
        m_Emplid = f_Emplid;
    }

    static public implicit operator string(Employee f_this) {
        return f_this.m_Emplid;
    }

    private string m_Emplid;
}

class Starter {
    static void Main() {
        Employee newempl = new Employee("1234");
        switch (newempl) {
            case "1234":
                Console.WriteLine("Employee 1234");
                return;
            case "5678":
                Console.WriteLine("Employee 5678");
                return;
            default:
                Console.WriteLine("Invalid employee");
                return;
        }
    }
}
```

Iterative Statements

C# has the full repertoire of C-style iterative statements. C# also has a *foreach* statement. Iterative statements repeat a statement until a condition has been satisfied.

The *for* statement is designed for structured iteration. The *while* and *do* statement iterations are more flexible. The *for* statement contains three clauses. First is the *initializer_clause,* where the loop iterators are declared. The scope of an iterator is the *for* statement and *for_statement*. Second is the *Boolean_expression,* which must evaluate to a Boolean type. The expression normally compares the iterator to a *stop* value. Third, the *iterator_expression* is executed at each iteration, which is usually responsible for updating the iterator. Each clause is optional and delimited with a semicolon. The *for_statement* is repeated until the *Boolean_expression* is false.

The *for_statement* is repeated zero or more times. If the *Boolean_expression* is initially false, the *for_statement* is executed zero times. The syntax of the *for* statement is as follows:

```
for (initializer_clause; Boolean_expression; iterator_expression) for_statement
```

The following is a rather mundane *for* loop:

```
static void Main() {
    for (int iCounter = 0; iCounter < 10; ++iCounter) {
        Console.Write(iCounter);
    }
}
```

Both the *initializer_clause* and *iterator_expression* can contain multiple statements delimited by commas, not semicolons. This allows additional flexibility and complexity. Here is an example:

```
static void Main() {
    for (int iBottom = 1, iTop = 10; iBottom < iTop; ++iBottom, --iTop) {
        Console.WriteLine(“{0}x{1} {2}”, iBottom, iTop, iBottom * iTop);
    }
}
```

The *while* statement, which is an iterative statement, is more free-form than the *for* statement. The body of the *while* statement is executed zero or more times; it is executed when the *Boolean_expression* is true. If the *Boolean_expression* is initially false, the body is executed zero times.

Typically, the *while* statement or expression is responsible for altering an iterator or other factors, eventually causing the *Boolean_expression* to evaluate to *false,* which ends the loop. Care should be taken to avoid unintended infinite loops.

The syntax for the *while* statement is as follows:

```
while (Boolean_expression) body_statement
```

This is source code for selecting a choice rewritten with a *while* statement:

```
static void Main() {
    string resp;
    Console.WriteLine("Enter command ('x' to end):");
    while ((resp=(Console.ReadLine()).ToLower()) != "x") {
        switch (resp) {
            case "a":
                Console.WriteLine("Doing Task A");
                break;
            case "b":
                Console.WriteLine("Doing Task B");
                break;
            default:
                Console.WriteLine("Bad choice");
                break;
        }
    }
}
```

A *do* statement is a loop that evaluates the *Boolean_expression* at the end. This is the reverse of the *while* statement. The impact is that the body of the *do* statement is repeated one or more times. The niche for the *do* statement is when the body must be executed at least once. The iteration of the body continues while the *Boolean_expression* is true.

Here is the syntax of the *do* statement:

```
do body_statement  while (Boolean_expression)
```

Here is sample code of the *do* statement:

```
static void Main() {
    string resp;
    do {
        Console.WriteLine("Menu\n\n1 - Task A");
        Console.WriteLine("2 - Task B");
        Console.WriteLine("E - E(xit)");
        resp = (Console.ReadLine()).ToLower();
    }
    while(resp!="e");
}
```

The *foreach* statement is a convenient mechanism for automatically iterating elements of a collection. The alternative is manually iterating a collection with an enumerator object obtained with the *IEnumerable.GetEnumerator* method. All collections implement the *IEnumerable* interface. The *foreach* statement is unquestionably simpler.

This is the syntax of the *foreach* statement:

```
foreach (type variable in collection) body_statement
```

The *foreach* statement iterates the elements of the *collection*. As each element is enumerated, the *variable* is assigned the current element, and the body of the *foreach* statement is executed. The scope of the *variable* is the *foreach* statement. When the *collection* is fully iterated, the iteration stops.

The *variable* type should be related to the type of objects contained in the *collection*. In addition, the *variable* is read-only. Even using the *variable* in a context that implies change, such as passing the *variable* as a *ref* function parameter, is an error.

This code iterates an array of numbers:

```
static void Main() {
    string [] numbers={ "uno", "dos", "tres",
        "quatro", "cinco"};
    foreach (string number in numbers) {
        Console.WriteLine(number);
    }
 }
```

The *break* statement forces a premature exit of a loop or switch. Control is transferred to the statement after the loop or switch. In a *switch* block, the break prevents fallthrough between *switch_labels*. For an iterative statement, a break stops the iteration unconditionally and exits the loop. If the switch or iterative statement is nested, only the nearest loop is exited.

The *continue* statement transfers control to the end of a loop where execution of the loop is allowed to continue. The *Boolean_expression* of the iterative statement then determines whether the iteration continues.

This is sample code of the *break* statement:

```
static void Main() {
    string resp;
    while(true) {
        Console.WriteLine("Menu\n\n1 - Task A");
        Console.WriteLine("2 - Task B");
        Console.WriteLine("E - E(xit)");
        resp = (Console.ReadLine()).ToLower();
        if (resp == "e") {
             break;
        }
    }
}
```

C# Core Language Features

Now that some basic examples and the framework of Visual C# 2008 code have been presented, we can discuss the fundamental building blocks of any C# application. This section starts by discussing symbols and tokens, the most elemental components of a Visual C# 2008 application.

Symbols and Tokens

Symbols and tokens are the basic constituents of the C# language. C# statements consist of symbols and tokens—indeed, they cannot be assembled without them. Table 1-2 provides a list of the C# symbols and tokens. Each entry in Table 1-2 is explained in the text that follows.

TABLE 1-2 C# Symbols and tokens

Description	Symbols or tokens
White space	Space, Form Feed
Tab	Horizontal_tab, Vertical_tab
Punctuator	. , : ;
Line terminator	Carriage return, line feed, next line character, line separator, paragraph separator, carriage return and line feed together
Comment	// /* */ /// /** */
Preprocessor directive	#
Block	{}
Lambda expression	=>
Generics	< >
Nullable type	?
Character	Unicode_character
Escape character	*\code*
Integer suffix (case-insensitive)	*u l ul lu*
Real suffix (case-insensitive)	*f d m*
Operator	+ - * % / > < ? ?? ( ) [ ] | || ^ ! ~ ++ -- = is as & && -> :: << >>
Compound operator	== != <= >= += -= *= /= %= &= |= ^= <<= >>= =>

White Space

White space is defined as a *space, horizontal tab, vertical tab,* or *form feed* character. White space characters can be combined; where one whitespace character is required, two or more contiguous characters of white space can be substituted.

Tabs

Tabs—horizontal and vertical—are white-space characters, as discussed just previously.

Punctuators

Punctuators separate and delimit elements of the C# language. Punctuators include the semicolon (;), dot (.), colon (:), and comma (,).

Semicolon punctuator In Visual C#, statements are terminated with a semicolon (;). C# is a free-form language in which a statement can span multiple lines of source code and can start in any position. Conversely, multiple statements can be combined on a single source code line. Here are some variations:

```
int variablea =
         variableb +
               variablec;

variableb = variableb + 3; variablec = variablec + 1;

++variableb;
```

Dot punctuator Dot syntax connotes membership. The dot character (.) binds a target to a member, in which the target can be a namespace, type, structure, enumeration, interface, or object. This assumes the member is accessible. Membership is sometimes nested and described with additional dots.

Here is the syntax for the dot punctuator:

```
Target.Member
```

This is an example of the dot punctuator:

```
System.Windows.Forms.MessageBox.Show("A nice day!");
```

System, Windows, and *Forms* are namespaces. *MessageBox* is a class. *Show,* the most nested member, is a static method.

Colon punctuator The colon punctuator primarily delimits a label, indicates inheritance, indicates interface implementation, sets a generic constraint, or is part of a conditional operator.

Labels are tags for locations to which program execution can be transferred. A label is terminated with a colon punctuator (:). The scope of a label is limited to the containing block and any nested block. There are various methods for transferring to a label. For example, you can jump to a label with the *goto* statement. Within a *switch* block, you also can use the *goto* statement to jump to a *case* or *default* statement.

Here is the syntax for the label punctuator:

label_identifier: *statement*

A statement must follow a label, even if it's an empty statement.

Here is an example of a *goto* statement:

```
    public static void Main() {
        goto one;
        // do stuff
one:    Console.WriteLine("one");
    }
```

Comma punctuator The comma punctuator delimits array indexes, function parameters, types in an inheritance list, statement clauses, and other language elements. The comma punctuator separates clauses of a *for* statement in the following code:

```
for (int iBottom = 1, iTop = 10; iBottom < iTop; ++iBottom, --iTop) {
    Console.WriteLine("{0}x{1} {2}", iBottom, iTop, iBottom*iTop);
}
```

A statement clause is a substatement in which multiple statement clauses can be combined into a single statement. Statement clauses are not always available—check documentation related to the language artifact to be sure.

Line Terminators

Line terminators separate lines of source code. Where one line terminator is available, two or more are allowed. Except in string literals, line terminators can be inserted anywhere white space is allowed. The following code is syntactically incorrect:

```
int variableb, variablec;

int variablea = var
            iableb+variablec; // wrong!
```

The *variableb* identifier cannot contain spaces. Therefore, it also cannot contain a line terminator.

Comments

C# supports four styles of comments: single-line, delimited, single-line documentation, and multi-line documentation comments. Although comments are not required, the liberal use of comments is considered good programming style. Be kind to those maintaining your program (present and future) —comment! I highly recommend reading *Code Complete, Second Edition* (Microsoft Press, 2004), by Steve McConnell; this book provides valuable best practices on programming, including how to document source code properly.

Single-line comments: // Single-line comments start at the comment symbol and conclude at the line terminator, as follows:

```
Console.WriteLine(objGreeting.French);  // Display Hello (French)
```

Delimited comments: /* and */ Delimited comments, also called *multi-line* or *block* comments, are bracketed by the */** and **/* symbols. Delimited comments can span multiple lines of source code:

```
/*
        Class Program: Programmer Donis Marshall
*/
class Program {
    static int Main(string[] args) {
        Greeting objGreeting = new Greeting();
        Console.WriteLine(objGreeting.French);  // Display Hello (French)
        return 0;
    }
}
```

Single-line documentation comments: /// Documentation comments apply a consistent format to source code comments and use XML tags to classify comments. With the documentation generator, documentation comments are exportable to an XML file. The resulting file is called the *documentation file,* which is identified in the Visual Studio project options. IntelliSense and the Object Browser use information in this file.

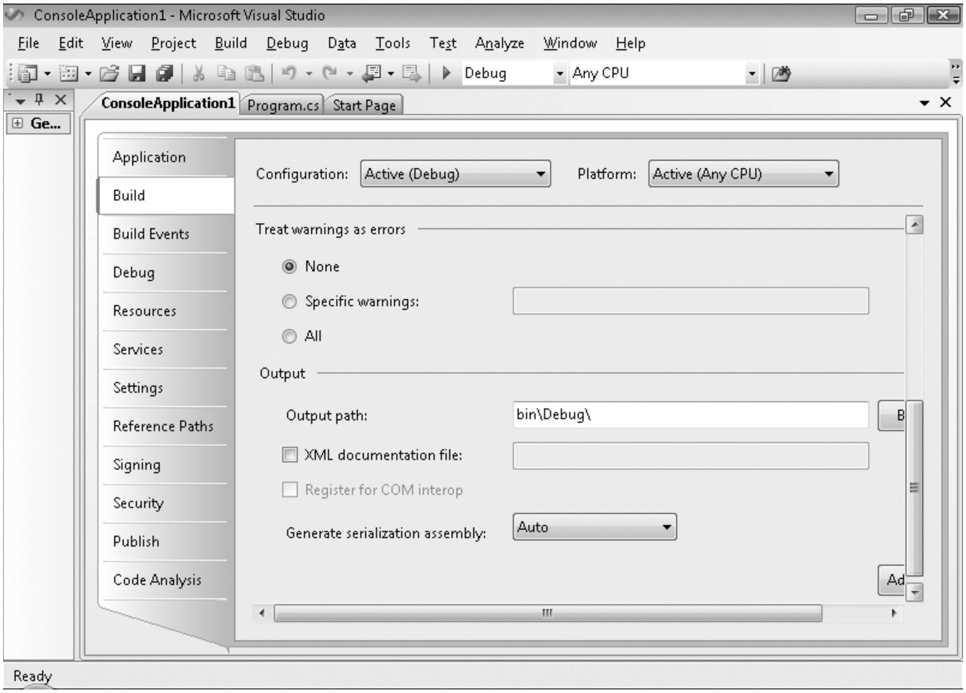
Single-line documentation comments are partially automated in the Visual Studio IDE. The Visual Studio IDE has Smart Comment Editing, which automatically continues or creates a skeleton for a document comment after initially entering the */// * symbol. For example, the following code snippet shows sample code with single-line documentation comments. After entering an initial *///*, Smart Comment Editing completed the remainder of the comment framework, including adding comments and XML tags for the type, methods, method parameter, and return value. You only need to update the comment framework with specific comments and additional comment tags that might be helpful:

```
/// <summary>
///
/// </summary>
class Program {
    /// <summary>
    ///
    /// </summary>
    /// <param name="args"></param>
    /// <returns></returns>
    static int Main(string[] args) {
         Greeting objGreeting = new Greeting();
         Console.WriteLine(objGreeting.French); // Display Hello (French)
         return 0;
    }
}
```

Here are the documentation comments with added details:

```
/// <summary>
/// Starter class for Simple HelloWorld
/// </summary>
class Program {
   /// <summary>
   /// Program Entry Point
   /// </summary>
   /// <param name="args">Command Line Parameters</param>
   /// <returns>zero</returns>
   static int Main(string[] args) {
         Greeting objGreeting = new Greeting();
         Console.WriteLine(objGreeting.French);   // Display Hello (French)
         return 0;
   }
}
```

The C# compiler is a documentation generator. The */doc* compiler option instructs the compiler to generate the documentation file. This can be done using the Visual Studio IDE. Select *Project* Properties from the Project menu. In the Properties window, select the Build tab. Toward the bottom of the Build pane (shown in Figure 1-2), you can specify the name of the XML documentation file.



Configuration: Active (Debug)   Platform: Active (Any CPU)

Treat warnings as errors

● None

○ Specific warnings:

○ All

Output

Output path: bin\Debug\

☐ XML documentation file:

☐ Register for COM interop

Generate serialization assembly: Auto

FIGURE 1-2 The Build pane of the Project Settings window

For the preceding source file, this is the documentation file generated by the C# compiler:

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>ConsoleApplication1</name>
    </assembly>
    <members>
        <member name="T:ConsoleApplication1.Program">
            <summary>
            Starter class for Simple HelloWorld
            </summary>
        </member>
        <member name="M:ConsoleApplication1.Program.Main(System.String[])">
            <summary>
            Program Entry Point
            </summary>
            <param name="args">Command Line Parameters</param>
            <returns>zero</returns>
        </member>
    </members>
</doc>
```

The documentation generator prefixes IDs to element names of the *member* name tag. In the preceding documentation file, *T* is the prefix for a type, whereas *M* is a prefix for a method. Here’s a listing of IDs:

E	Event
F	Field
M	Method
N	Namespace
P	Property
T	Type
!	Error

Multi-line documentation tags Multi-line documentation tags are an alternative to single-line documentation tags. Smart Comment Editing is not available with multi-line documentation tags. You must enter the documentation tags explicitly. However, Intellisense is available. Multi-line documentation comments must adhere to a degree of consistency, which is explained in the article titled “Delimiters for Documentation Tags (C# Programming Guide),” in Visual Studio Help *(ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.en/dv_csref/html/9b2bdd18-4f5c-4c0b-988e-fb992e0d233e.htm)*.

Here is an example of delimited documentation tags:

```
/**
 *<summary>this is an example.</summary>
*/
```

Preprocessor Directives

Preprocessor directives define symbols, undefine symbols, include source code, exclude source code, name sections of source code, and set warning and error conditions. The variety of preprocessor directives is limited compared with C++, and many of the C++ preprocessor directives are not available in C#. There is not a separate preprocessor or compilation stage for preprocessor statements. Preprocessor statements are processed by the normal C# compiler. The term *preprocessor* is used because the preprocessor directives are semantically similar to related commands in C++.

Here is the syntax for a preprocessor directive:

```
#command expression
```

This is a list of preprocessor directives available in C#:

#define *#undef* *#if*
#else *#elif* *#endif*
#line *#error* *#warning*
#region *#endregion* *#pragma*

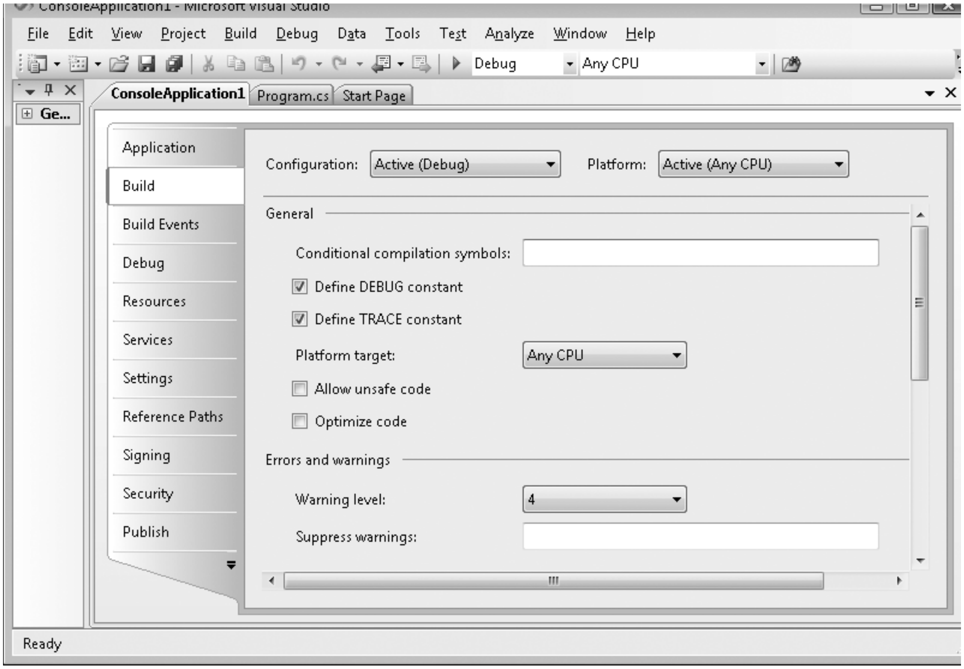
The preprocessor symbol (#) and subsequent directive are optionally separated with white space but must be on the same line. A preprocessor directive can be followed with a single-line comment but not a multi-line comment.

Declarative preprocessor directives The declarative preprocessor directives are *#define* and *#undef*, which define and undefine a preprocessor symbol, respectively. Defined symbols are implicitly true, whereas undefined symbols are false. Declarative symbols must be defined in each compilation unit where the symbol is referenced. Undeclared symbols default to *undefined* and *false*. The *#define* and *#undef* directives must precede any source code. Redundant *#define* and *#undef* directives have no effect.

Preprocessor symbols can also be set as a compiler option. In the Build pane of the *Project* Properties dialog box, you can define one or more symbols in the Conditional Compilation Symbols text box. This is shown in Figure 1-3. Other preprocessor symbols, such as *DEBUG* and *TRACE,* are commonly set implicitly as a compiler option.

Here is the syntax for declarative preprocessor directives:

```
#define identifier
#undef identifier
```



Configuration: Active (Debug)
Platform: Active (Any CPU)
Conditional compilation symbols:
☑ Define DEBUG constant
☑ Define TRACE constant
Platform target: Any CPU
☐ Allow unsafe code
☐ Optimize code
Warning level: 4
Suppress warnings:

FIGURE 1-3 The Conditional Compilation Symbols text box

Conditional preprocessor directives Conditional preprocessor directives are the *#if*, *#else*, *#elif*, and *#endif* directives, which exclude or include source code. A conditional preprocessor directive begins with *#if* and ends with *#endif*. The intervening conditional preprocessing directives, *#else* and *#elif*, are optional.

Here is the syntax for conditional preprocessor directives:

```
#if Boolean_expression
#elif Boolean_expression
#else
#endif
```

The *Boolean_expression* of the *#if* and *#elif* directive is a combination of preprocessor symbols and Boolean operators (*! == != && ||*). If the *Boolean_expression* is true, the source code immediately after the *#if* or *#elif* directive and before the next conditional preprocessor directive is included in the compilation. If the *Boolean_expression* is false, the source code is excluded from source compilation. The *#else* directive can be added to a *#if* or *#elif* combination. If the *Boolean_expression* of *#if* and *#elif* is false, the code following the *#else* is included in the compilation. When true, the source code after the *#else* is not included. Here's sample code with preprocessor symbols and related directives:

```
#define DEBUGGING

using System;

namespace Donis.CSharpBook {
    class Starter{
#if DEBUGGING
        static void OutputLocals() {
              Console.WriteLine("debugging...");
        }
#endif
        static void Main() {
#if DEBUGGING
            OutputLocals();
#endif
        }
    }
}
```

Finally, the *#elif* directive is a combination of an *else* and *if* conditional preprocessor directive. It is matched with the nearest *#if* directive:

```
#if expression
    source_code
#elif expression
    source_code
#else
    source_code
#endif
```

Diagnostic directives Diagnostic directives include the *#error*, *#warning*, and *#pragma* directives. The *#error* and *#warning* directives display error and warning messages, respectively. The diagnostic messages are displayed in the Error List window of the Visual Studio IDE. Similar to standard compilation errors, an *#error* directive prevents the program from compiling successfully; a *#warning* directive does not prevent the program from successfully compiling unless Treat Warnings As Error is set as a compiler option. You can use conditional directives to conditionally apply diagnostic directives.

Here is the syntax for diagnostic directives:

```
#error error_message
#warning error_message
```

The *error_message* is of string type and is optional.

Pragma directives The *#pragma* directive disables or enables compilation warnings. When disabled, the warning or error is suppressed and will not appear in the Error List window. This is useful for suppressing an unwanted warning or error temporarily or permanently.

Here is the syntax for pragma directives:

```
#pragma warning disable warning_list
#pragma warning restore warning_list
```

The *warning_list* contains one or more warnings delimited with commas. A disabled warning remains disabled until it is restored or the compilation unit ends.

The following code demonstrates the *pragma* warning directive. In this example, the *219* warning (Variable Is Assigned But Its Value Is Never Used) is initially disabled and then restored. Therefore a warning is received about *variableb* but not *variablea*.

```
class Starter
{
#pragma warning disable 219
    static void Main()
    {
        int variablea = 10;
    }
#pragma warning restore 219

    static void FuncA()
    {
        int variableb = 20;
    }
}
```

Region directives Region directives mark sections of source code. The *#region* directive starts a region, whereas the *#endregion* directive ends the region. Region directives can be nested. The Visual Studio IDE outlines the source code based on region directives. In Visual Studio, you can collapse or expand regions of source code.

Here is the syntax for region directives:

```
#region identifier
source_code
#endregion
```

Line directives Line directives modify the line number reported in subsequent compiler errors and warnings. There are three versions of the line directive.

Here is the syntax for line directives:

```
#line line_number source_filename
#line default
#line hidden
```

The first *#line* directive renumbers the source code from the location of the directive until the end of the compilation unit is reached or overridden by another *#line* directive. In the following code, the *#line* directive resets the current reporting line to *25*:

```
#line 25
static void Main() {
    Console.WriteLine("#line application");
    int variablea=10;  // 219 warning
}
```

The *#line default* directive undoes any previous *#line* directives. The line number is then reset to the natural line number.

The *#line hidden* directive is only tangentially related to the line number. This directive does not affect the line number; it hides source code from the debugger when stepping. The source code is skipped until the next *#line* directive is encountered. This is helpful in stepping through source code. Hidden code is essentially stepped over. For example, when stepping in the following source, the *for* loop is stepped over. The *#line default* directive then returns normal stepping.

```
    static void Main()
    {
        int variablea = 10;
        variablea++;
#line hidden
        for (int i = 0; i < 5; ++i)
        {
            variablea++;
        }
#line default
        Console.WriteLine(variablea);
    }
```

Blocks

A type, which can be a class, struct, interface, or enum, is defined within a block. Members of the type are contained inside the block.

Here is the syntax for a type block:

```
type typename {  // block
}
```

A block can also be a statement block. Statement blocks contain one or more statements. Each statement of the statement block is delimited by a semicolon. Typically, where a single statement is allowed, a statement block can be substituted. Statement blocks are commonly used as function bodies, conditional statements, and iterative statements. For function bodies, a statement block is required.

The *if* path in the following code consists of a single statement. Therefore, a statement block is not required. The *Console.WriteLine* is the only statement within the context of the *if* statement:

```
static void Main() {
    int variablea = 5, variableb = 10;
    if (((variablea + variableb) % 2) == 0)
        Console.WriteLine("the sum is even");
}
```

In the modified code, the *if* path contains multiple statements and a statement block is needed. Some would suggest, and I agree, that always using statement blocks with conditional or iterative statements is a good practice. This prevents an inadvertent future error when a single statement is expanded to multiple statements as shown below, but the block is forgotten:

```
static void Main() {
    int variablea = 5, variableb = 10;
    if (((variablea + variableb) % 2) == 0) {
        Console.WriteLine(“{0} {1}”, variablea,
            variableb);
        Console.WriteLine("the sum is even");
    }
}
```

Generic types

A generic is an abstraction of a type, which itself is an abstraction of a noun, place, or thing.

The *NodeInt* class is an abstraction of a node within a linked list of integers. The following is a partial implementation of the code (the full implementation is presented later in this book):

```
class NodeInt {
    public NodeInt(int f_Value, NodeInt f_Previous) {
        m_Value = f_Value;
        m_Previous = f_Previous;
    }

    // Remaining methods

    private int m_Value;
    private NodeInt m_Previous;
}
```

The *Node* generic type further abstracts a linked list. Unlike *NodeInt, Node* is not integer-specific but a linked list of any type. In the generic type, integer specifics of the *NodeInt* class have been removed and substituted with placeholders.

```
class Node<T> {
    public Node(T f_Value, Node<T> f_Previous) {
        m_Value = f_Value;
        m_Previous = f_Previous;
    }
    // Remaining methods

    private T m_Value;
    private Node<T> m_Previous;
}
```

In the preceding example, *T* is the generic type parameter, which is then used as a placeholder throughout the class for future type substitution.

There is much more about generics later in Chapter 7, "Generics."

Characters

C# source files contain Unicode characters, which are the most innate of symbols. Every element, keyword, operator, or identifier in the source file is a composite of Unicode characters.

Numeric Suffixes

Numeric suffixes cast a literal value to the underlying or a related type. Literal integer values can have the *l, u, ul,* and *lu* suffixes appended to them; literal real values can have the *f, d,* and *m* suffixes added. The suffixes are case-insensitive. Table 1-3 describes each suffix.

TABLE 1-3 Description of suffixes

Description	Type	Suffix
Unsigned integer or unsigned long	*uint* or *ulong*	*U*
Long or unsigned long	*long* or *ulong*	*L*
Unsigned long	*ulong*	*ul* or *lu*
Float	*float*	*F*
Double	*double*	*D*
Money	*decimal*	*M*

When casting a real type using the *m* suffix (for monetary or currency calculations), rounding might be required. If so, banker's rounding is used.

Here is an example of a numeric suffix:

```
uint variable = 10u;
```

Escape Characters

The escape character provides an alternate means of encoding Unicode characters, which is particularly useful for special characters that are not available on a standard keyboard. Escape sequences can be embedded in identifiers and string literals. Unicode escape sequences must have four hexadecimal digits and are limited to a single character.

A Unicode escape sequence looks like this:

```
\u hexdigit1 hexdigit2 hexdigit3 hexdigit4
```

Hexadecimal escape sequences contain one or more digits as defined by the location of a Unicode character.

A hexadecimal escape sequence looks like this:

```
\x hexdigit1 hexdigit2 ... hexdigitn
```

Table 1-4 shows a list of the predefined escape sequences in C#.

TABLE 1-4 Predefined escape sequences

Simple escape	Sequence
Single quote	\’
Double quote	\”
Backslash	\\
Null	\0
Alert	\a
Backspace	\b
Form feed	\f
New line	\n
Carriage return	\r
Horizontal tab	\t
Unicode character	\u
Vertical tab	\v
Hexadecimal character(s)	\x

This is an unconventional version of the traditional “Hello World!” program:

```
using System;

class HelloWorld {
    static void Main() {
        Console.Write("\u0048\u0065\u006C\u006C\u006F\n");
        Console.Write("\x77\x6F\x72\x6C\x64\x21\b");
    }
}
```

Verbatim Characters

The verbatim character (@) prevents the translation of a string or identifier, where it is treated “as-is.” To create a verbatim string or identifier, prefix it with the verbatim character. This is helpful, for example, when storing directory paths in a string literal.

A verbatim string is a string literal prefixed with the verbatim character. The characters of the verbatim string, including escape sequences, are not translated. The exception is the escape character for quotes, which is translated even in a verbatim string. Unlike a normal string, verbatim strings can even contain physical line feeds.

Here is a sample verbatim string:

```
using System;

class Verbatim{
    static void Main() {
        string fileLocation = @"c:\datafile.txt";
        Console.WriteLine("File is located at {0}",
                          fileLocation);
    }
}
```

A verbatim identifier is an identifier prefixed with the verbatim character that prevents the identifier from being parsed as a keyword. When porting source code from another programming language where allowable keywords and identifiers may be different, this feature could be useful. Otherwise, it is a best practice not to use this technique because verbatim identifiers almost always make your code less readable and harder to maintain.

The following source code is technically correct:

```
public class ExampleClass {
    public static void Function() {
        int @for = 12;
        MessageBox.Show(@for.ToString());
    }
}
```

In the preceding code, the *for* keyword is being used as a variable name. This is confusing at best. The *for* keyword is common in C# and many other programming languages, and therefore most developers would find this code confusing.

Operators

Operators are used in expressions and always return a value. There are three categories of operators: unary, binary, and ternary. Some operators, such as *is, as,* and *default,* also are considered keywords. The following sections describe all the operators in C#.

Unary operators Unary operators have a single operand. Table 1-5 lists the unary operators.

TABLE 1-5 **Unary operators**

Operator	Symbol	Sample	Result
unary plus	+	*variable = +5;*	5
unary minus	–	*variable = –(–10);*	10
Boolean negation	*!*	*variable = !true;*	false
bitwise 1's complement	~	*variable = ~((uint)1);*	4294967294
prefix increment	++	*++variable;*	11
prefix decrement	– –	*– –variable;*	10
postfix increment	++	*variable++;*	11
postfix decrement	– –	*variable – –;*	10
cast	*( )*	*variable = (int)123.45;*	123
function	*( )*	FunctionCall*(*parameter*);*	return value
array index	*[ ]*	arrayname*[*index*];*	nth element
dot	.	container.member	member
global namespace qualifier	*::*	*global::*globalmember	Globalmember

Here are some more details on unary parameters:

- Prefix operators are evaluated before the encompassing expression.
- Postfix operators are evaluated after the encompassing expression.

Binary operators Binary operators have a left and right operand. Table 1-6 details the binary operators.

TABLE 1-6 **Binary operators**

Operator	Symbol	Sample	Result
assignment	=	*variable =10;*	10
binary plus	+	*variable = variable + 5;*	15
binary minus	–	*variable = variable – 10;*	5
multiplication	*	*variable = variable * 5;*	25
division	/	*variable = variable / 5;*	5
modulus	%	*variable = variable % 3;*	2
bitwise AND	&	*variable = 5 & 3;*	1
bitwise OR	|	*variable = 5 | 3;*	7
bitwise XOR	^	*variable = 5 ^ 3;*	6
bitwise shift left	<<	*variable = 5 << 3;*	40
bitwise shift right	>>	*variable = 5 >> 1;*	2
null coalescing	*??*	*variableb = variable??5*	2

Here's more information on binary operators:

- Integer division truncates the floating point portion of the result.
- The operands in a bitwise shift left are *value* << *bitcount*.
- The operands in a bitwise shift right are *value* >> *bitcount*.

Compound operators Compound operators combine an assignment and another operator. If the expanded expression is *variable = variable operator value*, the compound operator is *variable operator*= *value*. For example, assume that we want to code the following:

```
variable = variable + 5;
```

The preceding statement is equivalent to this:

```
variable += 5;
```

Compound operations are a shortcut and are never required in lieu of the expanded statement. Table 1-7 lists the compound operators.

TABLE 1-7 Compound operators

Operator	Symbol	Sample
addition assignment	+=	*variable += 5;*
subtraction assignment	–=	*variable –= 10;*
multiplication assignment	*=	*variable *= 5;*
division assignment	/=	*variable /= 5;*
modulus assignment	%=	*variable %= 3;*
AND assignment	&=	*variable &= 3;*
OR assignment	|=	*variable |= 3;*
XOR assignment	^=	*variable ^= 3;*
left-shift assignment	<<=	*variable <<= 3;*
right-shift assignment	>>=	*variable >>= 1;*

Boolean operators Boolean expressions evaluate to *true* or *false*. Unlike C++, the integer values of nonzero and zero are not equivalent to a Boolean *true* or *false*.

There are two versions of the logical *and* and *or* operators. The *&&* and || operators support short-circuiting, whereas *&* and | do not. What is short-circuiting? If the result of the expression can be determined with the left side, the right side is not evaluated. Without disciplined coding practices, short-circuiting might cause unexpected side effects.

Next is an example of possible short-circuiting. If *FunctionA* evaluates to *false,* the entire Boolean expression is false. Therefore the right-hand expression (*FunctionB*) is not evaluated. If calling *FunctionB* has a required side effect, short-circuiting in this circumstance could cause a bug.

```
if (FunctionA() && FunctionB()) {

}
```

Table 1-8 shows the Boolean operators.

TABLE 1-8 Boolean operators

Operator	Symbol
equals	==
not equal	!=
less than	<
greater than	>
logical AND (allows short-circuiting)	&&
logical OR (allows short-circuiting)	||
logical AND	&
logical OR	|
less than or equal	<=
greater than or equal	>=
logical XOR	^

Ternary operators The conditional operator is the sole ternary operator in C# and is an abbreviated *if else* statement.

Here is the syntax of the conditional operator:

```
Boolean_expression ? true_statement : false_statement
```

This is the conditional operator in source code:

```
char e = (x > 0) ? '>' : '<'
```

Type operators Type operators act on a type. The *as* and *is* operators are binary operators, while the *typeof* operator is unary. Table 1-9 lists the type operators.

TABLE 1-9 Type operators

Operator	Syntax	Description
as	*object* as *type*	Casts object to type if possible. If not, returns null.
is	*object* is *type*	Expression evaluates to true if object is related to type; otherwise, evaluates to false.
typeof	*typeof*(object)	Returns the type of the object.

Pointer operators Pointer operators are available in unsafe mode. This allows C# developers to use C++ style pointers. The *unsafe* compiler option sets unsafe mode. From the Visual Studio IDE, you can choose the Allow Unsafe Mode option in the *Project* Properties dialog box on the Build pane. Table 1-10 lists the pointer operators.

TABLE 1-10 Pointer operators

Operator	Symbol	Description
asterisk operator (postfix)	*	Declare a pointer
asterisk operator (prefix)	*	Dereference a pointer
ampersand operator	&	Obtain an address
arrow operator	->	Dereference a pointer and member access

Here is some sample code using pointers:

```
static void Main(string[] args)
{
   unsafe {
         int variable = 10;
         int* pVariable = &variable;
         Console.WriteLine("Value at address is {0}.",
               *pVariable);
   }
}
```

A more extensive review of pointers is presented later in the book in Chapter 19, "Unsafe Code."

Miscellaneous operators The miscellaneous operators are unary but do not otherwise fit into a clear category. Table 1-11 lists the miscellaneous operators.

TABLE 1-11 Miscellaneous operators

Operator	Syntax	Description
New	*new* type(parameters)	Calls a matching constructor of the type. For a reference type, creates an instance of the object on the managed heap. For a value type, creates an instance of an initialized object on the stack.
checked	*checked*(expression)	Exception is raised if *expression* overflows.
delegate	*delegate* return_type method	Defines a type that holds type-safe function references.
lambda	=>	Separates the input and expression body of a *lambda* expression.
unchecked	*unchecked*(expression)	Overflows are ignored in *expression*.

Identifiers

An identifier is the name of a C# entity, which includes type, method, property, field, and other names. Identifiers can contain Unicode characters, escape character sequences, and underscores. A verbatim identifier is prefixed with the verbatim character (as discussed in the section "Verbatim Characters" earlier in this chapter).

Keywords

One of the strengths of C# is that the language has relatively few keywords. Table 1-12 provides an overview of the C# keywords. Extended explanations of each keyword are provided in context at the appropriate location in this book.

TABLE 1-12 **Overview of C# keywords with explanations**

Keyword	Syntax	Explanation
abstract	*abstract* class identifier *abstract* method	The class cannot be instantiated. The method is implemented in a descendant class. This includes properties, indexers, and events.
base	*base*.member	Accesses a member of the base class.
break	*break*	Exits current loop or switch statement.
case	*case* label	Target of a switch expression.
catch	*catch(*filter*){* handler *}*	The *catch* clause is where an exception is handled. The exception filter determines if the exception is handled in the handler.
checked	*checked {* statement *}*	If an expression within the *statement_block* overflows, throws an exception.
class	*class* identifier	Defines a new class.
const	*const* type identifier	Declares a constant local variable or field. Constants cannot be modified.
continue	*continue*	Continues to the next iteration of the loop, if any.
do	*do {* statement *} while (*expression*)*	The *do* loop is iterated until the expression is false.
else	*else {* statement *}*	The *else* statement is matched to the nearest *if* statement and provides the *false* path.
enum	*enum* identifier	Defines an enumeration type.
event	*event* delegatename identifier	Defines an event of the *delegatename* type.
explicit	*explicit* operator conversiontype	This user-defined conversion requires an explicit cast.
extern	*extern* return_type method	A method implemented externally—outside the current assembly.

TABLE 1-12 **Overview of C# keywords with explanations**

Keyword	Syntax	Explanation
false	*false*	A Boolean value.
finally	*finally {* statement *}*	Associated with the nearest *try* block. The *finally* block has cleanup code for resources defined in the *try* block.
fixed	*fixed (*declaration*)*	Fixes a pointer variable in memory and prevents relocation of that variable by the Garbage Collector.
for	*for (*initializers*;* Boolean_expression*;* iterators*)* statement	The *for* loop iterates the statement until the *Boolean_expression* is false.
foreach	*foreach (*element in enumerable_collection*)*	Iterates elements in a enumerable_collection.
get	*get*	Accessor method of a property member.
goto	*goto* identifier	Transfers control to a label.
	goto case identifier	Transfers control to a label inside a *switch* statement.
	goto default	Transfers control to a default label inside a *switch* statement.
if	*if (*Boolean_expression*)* statement	The statement is executed if the *Boolean_expression* resolves to *true*.
implicit	*implicit* operator conversiontype	This user-defined conversion requires only an implicit cast.
in	*foreach (*element *in* enumerable_collection*)*	Iterate elements in an *enumerable_collection*.
interface	*interface* identifier	Defines an interface.
internal	*internal* identifier	Type or member accessible only within the current assembly.
lock	*lock(*object*) {* statement *}*	Statement blocks locked on the same object are protected by a shared critical section, and access to those blocks is synchronized.
namespace	*namespace* identifier	Defines a namespace.
new	*new* return_type method	The *new* method hides the matching method of the base class.
	new type	The *new* operator declares a new structure or class. The *new* operator is not required for structures.
null	*null*	*null* can be assigned to references and nullable value types.

TABLE 1-12 Overview of C# keywords with explanations

Keyword	Syntax	Explanation
object	*object*	The *object* keyword is an alias for *System.Object*, which is the base class to all .NET objects (value or reference type).
operator	*operator* operator	Define a user-defined operator as a class or struct member.
out	*out* type parameter	The actual parameter is passed by reference into the method and can be modified directly. The parameter can be uninitialized prior to the function call.
override	*override* method	Override a virtual method in a base class. Method includes a member function, property, indexer, or an event.
params	*params* type *[]* identifier	Variable-length parameter list. The *params* parameter must be the last parameter in a parameter list.
private	*private* member	The member is visible only within the containing class or struct.
protected	*protected* member	Protected members are visible to the parent and any descendant classes.
public	*public* member	Public members are visible to everyone. This includes inside and outside the class.
readonly	*readonly* type identifier	Read-only fields can be initialized at declaration or in a constructor but nowhere else.
ref	*ref* type parameter	The parameter is passed by reference into the method and can be modified directly in the called function. The parameter must be initialized prior to the function call. The ref keyword is also required at the call site.
return	*return* expression	Returns the result of an expression from a method. Functions with a void return can have a return statement without a value (i.e., *return;*).
sealed	*sealed* identifier	Class is not inheritable.
set	*set*	Mutator method of property member.
sizeof	*sizeof(*valuename*)*	*Sizeof* returns the size of a value type. *Sizeof* of non-primitive types requires unsafe mode.
stackalloc	*stackalloc* type *[*expression*]*	Allocates an array of a value type on the stack; available only in unsafe mode. Expression determines the size of the array.

TABLE 1-12 **Overview of C# keywords with explanations**

Keyword	Syntax	Explanation
static	*static* method	Method is class-bound and not associated with a specific object instance.
struct	*struct* identifier	Defines a new structure.
switch	*switch(*expression*) {* statement *}*	Control is transferred to either a matching case label or the default label, if present.
this	*this*	The *this* object is a reference to the current object instance.
throw	*throw* object	Throws a user-defined exception. Exception objects should be derived from *System.Exception*.
true	*true*	A Boolean value.
try	*try {* statement *}*	Code in *statement_block* of *try* is guarded. If an exception is raised, control is transferred to the nearest *catch* statement.
unchecked	*unchecked {* statement *}*	Overflows in unchecked statements are ignored.
unsafe	*unsafe* type	Type can contain unsafe code, such as pointers. Also requires the *unsafe code* compiler option.
	unsafe return_type method	The method can contain unsafe code such as pointers. Also requires the *unsafe code* compiler option.
using	*using* identifier	The *using* keyword makes the specified namespace implicit.
	*using (*identifier_constructor_statement*)* statement	*IDisposable.Dispose,* which is the explicit destructor, is called on the named object after the statement is executed.
virtual	*virtual* method	Makes the method overridable in a derived class.
void	*void* method	A void return means that the method does not return a value. The method can omit a return statement or have an empty return.
	void *identifier	*Identifier* is a pointer name. A void pointer is a typeless pointer; supported only in unsafe mode.
volatile	*volatile* fieldname	Access to volatile fields is serialized. This is especially useful in a multi-threaded environment. In addition, volatile fields also are not optimized.
while	*while (*expression*)* statement	The statement is repeated while the expression is true.

Primitives

Primitives are the predefined data types that are intrinsic to C#. Primitives are also keywords. Primitives historically found in C-base languages, including *int, long,* and many others, are included in C#. The intrinsic types are declared as C# keywords but are aliases for types in the .NET FCL. Except for the string type, the primitives are value types and allocated on the stack as structures. The string type is a class and allocated on the managed heap.

The primitives are listed in Table 1-13. Primitives have a published interface. For numeric types, the *min* property, *max* property, and *Parse* methods of the interface are particularly useful. The *min* and *max* property are invaluable for bounds checking, whereas the *Parse* method converts a string to the target primitive.

TABLE 1-13 **Primitives in C#**

Type	Primitive	Description	Range
bool	*System.Boolean*	Boolean	*true* or *false*.
byte	*System.Byte*	8-bit integer	0 to 255.
char	*System.Char*	16-bit Unicode character	/u0000 to /uFFFF.
decimal	*System.Decimal*	128-bit decimal	0 and ±1.0 × 10^{-28} to ±7.9 × 10^{28}, with 28 digits of precision.
double	*System.Double*	64-bit floating point	0 and ±5.0 × 10^{-324} to ±1.7 × 10^{308}, with 15 digits of precision.
float	*System.Single*	32-bit floating point	0 and ±1.5 × 10^{-45} to ±3.4 × 10^{38}, with 7 digits of precision.
int	*System.Int32*	32-bit unsigned integer	–2,147,483,648 to 2,147,483,647.
long	*System.Int64*	64-bit integer	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
sbyte	*System.SByte*	8-bit integer	–128 to 127.
short	*System.Int16*	16-bit integer	–32,768 to 32,767.
string	*System.String*	not applicable	String is an immutable variable length string.
uint	*System.UInt32*	32-bit unsigned integer	0 to 4,294,967,295.
ulong	*System.UInt64*	64-bit unsigned integer	0 to 18,446,744,073,709,551,615.
ushort	*System.UInt16*	16-bit unsigned integer	0 to 65,535.

Types

This chapter is an introduction of Microsoft Visual C# 2008, including LINQ, which is the most important new feature in C#. The remaining chapters of this book provide the underlying details of LINQ and other topics introduced in this chapter, beginning with the next chapter, which pertains to types.

The core ingredient of most programming languages is the type. The term *type* encompasses classes, structures, interfaces, and enumerations. Classes are reference types and are placed on the managed heap, structures are value types and appear on the stack, and an enumeration is a set of flags.

Even a nontrivial C# program has at least one type. Literarily, except for namespaces, every entity in C# is a type or a member of a type. This includes the common primitives, such as *int*, *float*, and *double*. Classes are the nouns of the C# language, and it is certainly difficult to write a great story without any nouns. It is impossible to write a C# program without classes.