### Overview of C#

This chapter takes you on a whirlwind tour of the C# language. OK! OK! We can hear what you’re saying: “But you can’t compress C# into a single chapter!”

We don’t plan to do that. What we’re going to do here is run through the language and introduce to you its features. These features will be covered in greater detail in later chapters.

##### C#

You already know that C# is pronounced “see-sharp” and that C# is an object-oriented, type-safe (this means that you cannot treat a value as a type to which it does not belong; more on this later) language that’s similar to C or C++.

##### C# Basics

Let’s start by looking at the universal “Hello, World!” program in C#:

using System;

class MyClass

{

static void Main()

{

Console.WriteLine(“Hello, World!”);

}

}

The preceding code is the source for the program, stored in text files that will have the extension

.cs (for example, helloworld.cs). A C# program can consist of one or more source files.

The source files are turned into programs using a compiler. We’ll use the command-line compiler here as opposed to looking at something more complicated, such as Visual Studio .NET. To compile this, we use the following command:

csc helloworld.cs

*Note that to compile you will need the .NET Framework installed on the system. This is also required to run the executable.*

Here, csc is the C# compiler that ships with the .NET Framework (more accurately, it is csc.exe, but the extension is not needed), and helloworld.cs is the C# source file passed to the compiler as an argu- ment for compiling.

The result of compiling helloworld.cs is an executable file called by default helloworld.exe. Running the outputted executable will give the following output:

Hello, World!

Examining the C# Source Code

Let’s take a look at the source code.

using System;

Here, the using directive is referencing a namespace called System. This is provided by the class library of the Common Language Infrastructure (CLI — another name for the .NET Framework). It is this namespace that contains the Console class that we’ll be using in a few lines in the source code. By making use of the using directive, we can make unqualified use of the types that are members of the namespace. What does this mean? Well, it means we have to write less code, allowing us to use:

Console.WriteLine

Instead of:

System.Console.WriteLine

OK, the savings here is only seven characters, but over the course of a program, this adds up. Notice that the method called Main is a member of a class called Hello.

A static modifier is used so that the method it is assigned to becomes a method of the class rather than an instance of the class (don’t worry if you don’t know what this means just yet — we’ll be covering static modifiers in a later chapter).

The Main method is the point at which execution begins for the application. This is called the *entry point*.

The “Hello, World” output is handled by a class library that automatically handles all the work neces- sary to display the text onscreen.

##### Types

A type is how a programming language classifies different values and expressions. While a computer stores all the data as zeros and ones, that data needs to have a context or meaning. To preserve this meaning, types are used.

C# supports two basic kinds of type:

* + Value types
  + Reference types

These types are explained briefly in the following sections and will be expanded upon later in this book. For now, all you need to know are the kinds of types and what they represent.

Value Types

Value types are composed of the following:

* + Enum types
  + Struct types
  + Simple types (for example, char, float, and int)

With value type variables, the variable contains the data, which is different from reference variables (as you will see in a moment). Also, with value types, each of the variables will have their own copy of the data and an operation on one copy does not affect any of the others.

Reference Types

Reference types are composed of the following:

* + Array types
  + Class types
  + Delegate types
  + Interface types

The main difference between reference types and value types is that with reference types the variables store references to the object rather than hold the actual data (compare this to value types). Here, if two or more variables point to the same object, an operation carried out on one affects all the other references.

Predefined Types

C# comes complete with a number of predefined types. There are two predefined reference types:

* + object — This is the ultimate base type for all other types.
  + string — This is used to represent Unicode string values.

The following are predefined value types:

* Signed integral types (int, long, sbyte, and short)
* Unsigned integral types (byte, unint, ulong and ushort)
* The types bool, char, and decimal
* Floating-point types (float and double)

The following table offers a complete listing of all the different types in C#, along with an explanation of the data they represent.

**Type Notes**

Bool

Boolean type (values of true and false allowed) Example:

bool x = true;

Byte

Char

bool y = false;

8-bit unsigned integral type Example:

byte x = 13;

A single Unicode character Example:

char x = ‘x’;

Decimal

char y = ‘c’;

A high-precision decimal type, with at least 28 significant digits Example:

decimal x = 1.99M;

Double

decimal y = 9.02M;

Double-precision floating point type Example:

double x = 1.99;

Float

Int

double y = 9.02D;

Single-precision floating point type Example:

float x = 1.99F;

32-bit signed integral type Example:

int x = 7;

int y = 17;

int z = 1237;

**Type Notes**

Long

64-bit signed integral type Example:

long x = 17;

Object

Sbyte

long y = 37L;

Base type for all other types Example:

object x = null;

8-bit signed integral type Example:

sbyte x = 17;

Short

sbyte y = 37;

16-bit signed integral type Example:

short x = 17;

String

short y = 37;

A sequence of Unicode characters Example:

string x = “Hello, World!”;

Uint

string y = “37”;

32-bit unsigned integral type Example:

uint x = 17;

Ulong

uint y = 37U;

64-bit unsigned integral type Example:

ulong w = 17; ulong x = 37U; ulong y = 42L;

Ushort

ulong z = 54UL;

1. bit unsigned integral type Example:

ushort x = 17;

Overloading

Predefined types can make use of operator overloading. A good example of this are the comparison operators == and !=. They have different meanings for different predefined types, as explained below:

* Two expressions of int type are equal if they represent the same integer value. Example:

int x = 2; int y = 2;

x == y would be true

* Two expressions of object type are considered equal if both refer to the same object (or if both are null).

object x = null; object y = null;

x == y would be true

* Two expressions of string type are considered equal if the strings both the characters and whitespace are identical (or if both are null).

string x = “Hello”; string y = “Hello”; x == y would be true string x = “ Hello”; string y = “Hello “;

x == y would be false, since whitespace differences matter.

Conversions

In C# there are two kinds of conversions between types:

* **Implicit conversions.** These are conversions that can be safely performed, and no additional scrutiny is required by the compiler to make sure that the output is accurate.
* **Explicit conversions.** With explicit conversions, there is more attention paid to the conversion and the accuracy and reliability of the output.

Array Types

C# supports both single and multidimensional arrays. As well as regular rectangular arrays, jagged arrays are supported. A *jagged* array is an array of an array. Jagged arrays are easy to spot in code because [] appears in the code more than once:

int[][] a2;

Above you have an array of an array of int. int[][][] a3;

And here is an array of an array of an array of int.

Where do the names *rectangular* and *jagged* come from? Take a look at the following three-dimensional rectangular array:

int[,,] a1 = new int[10, 20, 30];

In this example, the length of a1’s three dimensions are 10, 20, and 30, respectively, and this array con- tains 10 x 20 x 30 elements. This would make up a regular shape if drawn out.

Jagged arrays, on the other hand, do not have this regular pattern.

##### Variables and Parameters

Variables represent storage locations, and every variable has a type that determines what values can be stored in the variable. Local variables are declared in function members (for example, methods, proper- ties, and indexers).

A local variable is defined by specifying the following:

* + A type name
  + A declarator that specifies the variable name and an optional initial value The following code shows three local variable definitions:

int x;

int y = 7; int z = 14;

A local variable declaration can also include multiple declarators. For example:

int x, y = 7, z = 14;

It is absolutely essential that a variable be assigned before its value can be obtained. If not, a compiler error will be generated. As an example, trying to compile the following code would result in a compiler error (because the line highlighted is using a variable that has not yet been assigned a value):

class Test

{

static void Main()

{

int x;

int y = 7;

int z = x + y;

}

}

A *field* is a variable associated with a class or struct or an instance of a class or struct.

A field declared with the static modifier defines a static variable, and a field declared without this modifier defines an instance variable. A static field is associated with a type, and an instance variable is associated with an instance.

using Books.Data; class Titles

{

private static DataSet ds; public string Title; public decimal Price;

}

In the preceding example, there is a class that has a private static variable and two public instance variables.

Formal parameter declarations are also used to define variables. There are four different kinds:

* **Value parameters.** Used for “in” parameter passing, where the value of an argument is passed into a method
* **Reference parameters.** Used for “by reference” parameter passing, where the parameter acts as an alternative name for a caller that provided the argument
* **Output parameters.** Similar to a reference parameter, except that the initial value of the argu- ment provided by the caller is not important
* **Parameter arrays.** Declared with a params modifier. There can be only one parameter array for any method, and it will always be the last parameter specified.

##### Expressions

C# includes a whole raft of operators that can be used in expressions. These are grouped into:

* Unary operators
* Binary operators
* Ternary operator (there is only one)

The following table further subdivides the operators present in C# and lists them in order of precedence, from highest to lowest:

**Category Operator**

Primary

Unary

Multiplicative

Additive Shift

Relational/type-testing

Equality

Logical AND Logical XOR Logical OR Conditional AND Conditional OR Conditional

x.y f(x)

a[x] x++ x-- new

typeof checked unchecked

+

-

!

~

++x

--x

(T)x

\*

/

%

+

-

<<

>>

<

>

<=

>=

is as

==

!=

&

^

| &&

||

?:

*Table continued on following page*

**Category Operator**

Assignment =

\*=

/=

%=

+=

-=

<<=

>>=

&=

^=

|=

When an expression contains multiple operators, the precedence of the operators controls the order in which the individual operators are evaluated.

Precedence can be controlled by using parentheses. For example, the following expressions are processed differently:

* x + y \* z

Here y is multiplied by z and then the result added to x.

* (x + y) \* z

Here x and y are added together and the result multiplied by z.

##### Statements

Here is a listing of the statements present in C#. Many of them will be familiar to anyone who has used C or C++.

* Lists and block statements
* Labeled statements and goto statements
* Local constant declarations
* Local variable declarations
* Expression statements
* if statements
* switch statements
* while statements
* do statements
* for statements
  + foreach statements
  + break statements
  + continue statements
  + return statements
  + yield statements
  + throw statements
  + try statements
  + checked statements
  + unchecked statements
  + lock statements
  + using statements

##### Classes

Class declarations define new reference types. A class can inherit from another class and can also imple- ment interfaces.

All generic class declarations will have one or more type parameters. Class are made up of members and can include the following:

* + Constants
  + Events
  + Fields
  + Finalizers
  + Indexers
  + Instance constructors
  + Methods
  + Nested type declarations
  + Operators
  + Properties
  + Static constructors

Each member will also have an associated accessibility, which is used to control the regions of code that are able to access the member.

There are five possible forms of accessibility:

* public — Access is not limited.
* protected — Access is limited to the containing class or types derived from the containing class.
* internal — Access is limited to the program.
* protected internal — Access is limited to the program or types derived from the containing class.
* private — Access is limited to the containing type.

Constants

A constant is a class member that, as the name suggests, is used to represent a constant value. A constant value can either be declared or can be computed during compilation.

Constants can depend on other constants within the same program as long as there aren’t any circular dependencies in the code (where A depends on B, but then B is defined and depends on A).

Fields

A field is a member used to represent a variable associated with an object or class.

Methods

A method is a member that implements an action that can be performed by an object or class. Methods have:

* A list of formal parameters (which can be empty)
* A return value (unless the return-type is void) Methods can also be either static or nonstatic:
* Static methods are accessed through the class.
* Nonstatic methods are accessed through instances of the class.

*Nonstatic methods are also known as instance methods.*

Properties

A property is a member that provides access to a particular characteristic of an object or a class (for example, the length of a string). Properties are an extension of fields but differ in that they don’t indicate storage locations.

Properties have accessors that specify the statements executed when the values are read or written.

Events

An event is a member that allows an object or class to provide notifications. A class defines an event by providing an event declaration (which is of the delegate type) along with an optional set of event acces- sors.

Operators

An operator is a member used to define the meaning of an expression operator that can be applied to instances of the class.

Three kinds of operators can be defined:

* + Binary
  + Conversion
  + Unary

Indexers

An indexer is a member that allows an object to be indexed and accessed in much the same way as an array.

Instance Constructors

An instance constructor is a member that implements the actions needed to initialize an instance of a class.

Finalizers

A finalizer is a member that implements the actions required to finalize an instance of a class. These actions are carried out when a class is no longer required.

Finalizers cannot make use of the following:

* + Parameters
  + Accessibility modifiers

*Finalizers cannot be called explicitly.*

The finalizer for any instance is called automatically during the garbage collection process by the .NET Framework.

Static Constructors

A static constructor is a member that implements the actions needed to initialize a class. Static constructors cannot make use of any of the following:

* Parameters
* Accessibility modifiers

*Static constructors cannot be called explicitly and are called automatically.*

Inheritance

Classes support single inheritance (that is, they can only inherit from one class, also known as a super- class — this prevents complex code structures). The type object is the base class for all classes.

Methods, properties, and indexers can all be virtual. This means that their implementations can be over- ridden in derived classes.

Static Classes

Static classes are not intended to be instantiated, and they contain only static members. Static classes are all implicitly sealed, and they have no instance constructors.

##### Structs

Structs are quite similar to classes. The two main differences are:

* Structs are value types rather than reference types.
* Structs do not support inheritance.

So why use structs? Well, the main reason is performance: Because values are stored in the stack, they have a performance advantage over classes. Given the limitations of values, however, some program- mers choose to opt for classes.

##### Interfaces

An interface is used to define a *contract*. But what is a contract? An interface contract is a guarantee by an object that it will support all of the elements of its interface. This contract is created using the Interface keyword, which declares a reference type that encapsulates the contract.

A class or struct that implements an interface has to honor the contract, or an error occurs.

Interfaces can contain the following as members:

* + Events
  + Indexers
  + Methods
  + Properties

##### Delegates

Delegates allow programmers to make use of features in C# that other languages leverage using point- ers. There are two main differences between delegates and pointers:

* + Delegates are type-safe.
  + Delegates are object-oriented.

A delegate declaration is used to define a class. This class is derived from the class System.Delegate.

A delegate instance encapsulates one or more methods, and each method will be referred to as a callable entity.

When dealing with instance methods, a callable entity is made up of an instance and a method on that instance.

For static methods, a callable entity is made up of a method on its own.

##### Enums

An enum type declaration is used to define a type name for a related group of symbolic constants. Enums are used in situations where the programmer wants a fixed number of multiple choice options. The final choice is made at runtime from a set of options known at compile-time.

##### Generics

Generics is not a single feature but a group of features that the C# language offers.

Generics is the ability that C# has to parameterize classes, structs, interfaces, and methods based on the types of data stored in them and manipulated.

Many common classes and structs can be parameterized by the types of data being stored and manipu- lated. Parameterized classes are called generic class declarations, while parameterized structs are called generic struct declarations.

In addition, many interfaces will define contracts that can also be parameterized by the types of data they deal with. These are called generic interface declarations.

##### Iterators

In C#, the foreach statement is used to iterate through the elements contained in an enumerable collec- tion. In order to be enumerable, a collection has to make use of the GetEnumerator method, which returns an enumerator.

*Note that GetEnumerator is a parameterless method.*

An iterator is a statement block used to output an ordered sequence of values. Iterators are easy to spot in code because they make use of one or more yield statements. These are:

* yield return — Produces the next value of the iteration
* yield break — Used to indicate that the iteration is complete

##### Nullable Types

C# has support for user-defined nullable types. These nullable types provide support for nullability (that is, no value) across all value types.

Nullable types are built using the type modifier ?. For example, int? is the nullable form of the type

int, bool? is the nullable form of the type bool and, char? is the nullable form of the type char.

*A nullable type’s underlying type must be a non-nullable value type.*

Lifted conversions allow the predefined and user-defined operators that work on the standard value types to work also on the nullable versions of those types.

Both nullable conversions and lifted conversions allow for predefined and user-defined conversions to work on non-nullable value types and with nullable forms of those types.

Lifted operators allow for both predefined and user-defined operators that work for non-nullable value types also to work with nullable forms of those types.

##### Summary

This chapter provided an overview of the C# programming language. You looked at:

* What C# is and where it came from
* C# basics
  + Types in C#, including overloading and conversions
  + Variables
  + Parameters
  + Expressions
  + Statements
  + Classes
  + Structs
  + Interfaces
  + Delegates
  + Enums
  + Generics
  + Iterators

If you’re new to C#, this chapter is recommended reading; otherwise, feel free to dip in as you wish. In Chapter 4, you go on to look at the C# language structure.