Accelerated C# 2005

Trey Nash

Apress®

Accelerated C# 2005

ISBN-13 (pbk): 978-1-59059-717-0

ISBN-10 (pbk): 1-59059-717-6

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: James Huddleston
Technical Reviewer: David Weller
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Richard Dal Porto
Copy Edit Manager: Nicole LeClerc
Copy Editor: Nicole Abramowitz
Assistant Production Director: Kari Brooks-Copony
Production Editor: Ellie Fountain
Compositor: Kinetic Publishing Services, LLC
Proofreader: April Eddy
Indexer: Brenda Miller
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 3

C# Syntax Overview

This chapter introduces you to the syntax of the C# language. It's assumed that you have a reasonable amount of experience with C++ or Java, since C# shares the same syntax. This is no accident. The designers of C# clearly meant to leverage the knowledge of those who have already developed with C++ and Java, arguably the most dominant languages in software development, particularly object-oriented (OO) development.

I've noted nuances and differences that are specific to the C# language. But if you're familiar with either C++ or Java, you'll feel right at home with C# syntax.

C# Is a Strongly Typed Language

Like C++ and Java, C# is a strongly typed language, which means that every variable and object instance in the system is of a well-defined type. This enables the compiler to check that the operations you try to perform on variables and object instances are valid. For example, suppose you have a method that computes the average of two integers and returns the result. You could declare the C# method as follows:

```
double ComputeAvg( int param1, int param2 )
{
   return (param1 + param2) / 2.0;
}
```

This tells the compiler that this method accepts two integers and returns a double. Therefore, if the compiler attempts to compile code where you inadvertently pass an instance of type `Apple`, then the compiler will complain and stop. Suppose that you wrote the method slightly differently:

```
object ComputeAvg( object param1, object param2 )
{
   return ((int) param1 + (int) param2) / 2.0;
}
```

The second version of `ComputeAvg()` is still valid, but you have forcefully stripped away the type information. Every object and value in C# implicitly derives from `System.Object`. The `object` keyword in C# is an alias for the class `System.Object`. So, it is perfectly valid to declare these parameters as type `object`. However, `object` is not a numeric type. In order to perform the calculation, you must first *cast* the objects into integers. After you're done, you return the result as an instance of type `object`. Although this version of the method can seem more flexible, it's a disaster waiting to happen. What if some code in the application attempts to pass an instance of type `Apple` into `ComputeAvg()`? The compiler won't complain because `Apple` derives from `System.Object`, as every other class does. However, you'll get a nasty surprise at run time when your application throws an exception declaring that it cannot convert an instance of `Apple` to an integer. The method will fail, and unless you have

an exception handler in place, it could terminate your application. That's not something you want to happen in your code that is running on a production server somewhere.

It is *always* best to find bugs at compile time rather than run time. That is the moral of this story. If you were to use the first version of `ComputeAvg()`, then the compiler would have told you how ridiculous it was that you were passing an instance of `Apple`. This is much better than hearing it from an angry customer whose ecommerce server just took a dirt nap. The compiler is your friend, so let it be a good one and provide it with as much type information as possible to strictly enforce your intentions.

Expressions

Expressions in C# are practically identical to expressions in C++ and Java. The important thing to keep in mind when building expressions is operator precedence. C# expressions are built using operands, usually variables or types within your application, and operators. Many of the operators can be overloaded as well. Operator overloading is covered in Table 3-1, which lists the precedence of the operator groups. Entries at the top of the table have higher precedence, and operators that exist in the same category have equal precedence.

Table 3-1. *C# Operator Precedence*

Operator Group	Operators Included	Description
Primary	`x.m`	Member access
	`X(...)`	Method invocation
	`X[...]`	Array access
	`X++`, `x--`	Postincrement and postdecrement
	`new T(...)`, `new T[...]`	Object and array creation
	`typeof(T)`	Gets `System.Type` object for `T`
	`checked(x)`, `unchecked(x)`	Evaluates expression in checked and unchecked environments
Unary	`+x`, `-x`	Identity and negation
	`!x`	Logical negation
	`~x`	Bitwise negation
	`++x`, `--x`	Preincrement and predecrement
	`(T) x`	Casting operation
Multiplicative	`x*y`, `x/y`, `x%y`	Multiplication, division, and remainder
Additive	`x+y`, `x-y`	Addition and subtraction
Shift	`x<<y`, `x>>y`	Left and right shift
Relational and type testing	`x<y`, `x>y`, `x<=y`, `x>=y`	Less than, greater than, less than or equal to, greater than or equal to
	`x is T`	True if `x` is convertible to `T`; false otherwise
	`x as T`	Returns `x` converted to `T`, or null if conversion is not possible
Equality	`x == y`, `x != y`	Equal and not equal
Logical AND	`x & y`	Integer bitwise AND, Boolean logical AND

Operator Group	Operators Included	Description
Logical XOR	`x ^ y`	Integer bitwise XOR, Boolean logical XOR
Logical OR	`x | y`	Integer bitwise OR, Boolean logical OR
Conditional AND	`x && y`	Evaluates y only if x is true
Conditional OR	`x || y`	Evaluates y only if x is false
Null coalescing	`x ?? y`	If x is non-null, evaluates to x; otherwise, y
Conditional	`x ? y : z`	Evaluates y if x is true; otherwise, evaluates z
Assignment	`x = y`	Simple assignment
	`x op= y`	Compound assignment; could be any of *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, or |=

■ **Note** These operators can have different meanings in different contexts. Regardless, their precedence never changes. For example, the + operator can mean string concatenation if you're using it with string operands. By using operator overloading when defining your own types, you can make some of these operators perform whatever semantic meaning makes sense for the type. But again, you may never alter the precedence of these operators except by using parentheses to change the grouping of operations.

Statements

Statements in C# are identical in form to those of C++ and Java. A semicolon terminates one-line expressions. However, unlike C++, code blocks, such as those in braces in an `if` or `while` statement, do not need to be terminated with a semicolon. Adding the semicolon is optional at the end of a block.

Most of the statements that are available in C++ and Java are available in C#, including variable declaration statements, conditional statements, control flow statements, `try/catch` statements, and so on. However, C# (like Java) has some statements that are not available in C++. For example, C# provides a `try/finally` statement, which I discuss in detail in Chapter 7. In Chapter 12, I'll show you the `lock` statement, which synchronizes access to code blocks by using the sync block of an object. C# also overloads the `using` keyword, so you can use it either as a directive or a statement. You can use a `using` statement in concert with the Disposable pattern I describe in Chapters 4 and 13. The `foreach` statement, which makes iterating through collections easier, also deserves mention. You'll see more of this statement in Chapter 9, when I discuss arrays.

Types and Variables

Every entity in a C# program is an object that lives on either the stack or the managed heap. Every method is defined in a `class` or `struct` declaration. There are no such things as free functions, defined outside the scope of `class` or `struct` declarations, as there are in C++. Even the built-in value types, such as `int`, `long`, `double`, and so on, have methods associated with them implicitly. So, in C#, it's perfectly valid to write a statement such as the following:

```
System.Console.WriteLine( 42.ToString() );
```

A statement like this will feel unfamiliar if you're used to C++ or Java. But, it emphasizes how everything in C# is an object, even down to the most basic types. In fact, the keywords in C# are actually mapped directly into types in the `System` namespace that represent them. You can even

elect not to use the C# built-in types and explicitly use the types in the `System` namespace that they map to (but this practice is discouraged as a matter of style). Table 3-2 describes the built-in types, showing their size and what type they map to in the `System` namespace.

Table 3-2. *C# Built-In Types*

C# Type	Size in Bits	System Type	CLS-Compliant
`sbyte`	8	`System.SByte`	No
`short`	16	`System.Int16`	Yes
`int`	32	`System.Int32`	Yes
`long`	64	`System.Int64`	Yes
`byte`	8	`System.Byte`	Yes
`ushort`	16	`System.Uint16`	No
`uint`	32	`System.Uint32`	No
`ulong`	64	`System.UInt64`	No
`char`	16	`System.Char`	Yes
`bool`	8	`System.Boolean`	Yes
`float`	32	`System.Single`	Yes
`double`	64	`System.Double`	Yes
`decimal`	128	`System.Decimal`	Yes
`string`	N/A	`System.String`	Yes
`object`	N/A	`System.Object`	Yes

For each entry in the table, the last column indicates whether the type is compliant with the Common Language Specification (CLS). The CLS is defined as part of the CLI standard to facilitate multilanguage interoperability. The CLS is a subset of the Common Type System (CTS). Even though the CLR supports a rich set of built-in types, not all languages that compile into managed code support all of them. However, all managed languages are required to support the types in the CLS. For example, Visual Basic hasn't supported unsigned types traditionally. So the designers of the CLI defined the CLS to standardize types in order to facilitate interoperability between the languages. If your application will be entirely C#-based and won't create any components consumed from another language, then you don't have to worry about adhering to the strict guidelines of the CLS. But if you work on a project that builds components using various languages, then conforming to the CLS will be much more important to you.

In the managed world of the CLR, there are two kinds of types:

Value types: Defined in C# using the `struct` keyword. Instances of value types are the only kind of instances that can live on the stack. They live on the heap if they're members of reference types or if they're boxed, which I discuss later. They are similar to structures in C++ in the sense that they are copied by value by default when passed as parameters to methods or assigned to other variables. Although C#'s built-in value types represent the same kinds of values as Java's primitive types, there is no Java counterpart.

Reference types: Defined in C# using the `class` keyword. They are called reference types because the variables you use to manipulate them are actually references to objects on the managed heap. In fact, in the CLR reference-type variables are like value types that reference an object on the heap. In this way, C# and Java are identical. C++ programmers can think of them as pointers that you don't have to dereference to access objects. Some C++ programmers like to think of these as smart pointers.

Value Types

Value types can live on either the stack or the managed heap. You use them commonly when you need to represent some immutable data that is generally small in its memory footprint. You can define user-defined value types in C# by using the `struct` keyword.

■ **Note** Even though C++ has a `struct` keyword, the meaning in C# is different in that it's the only way to create value types in C#.

User-defined value types behave in the same way that the built-in value types do. When you create a value during the flow of execution, the value is created on the stack, as shown in this code snippet:

```
int theAnswer = 42;
System.Console.WriteLine( theAnswer.ToString() );
```

Not only is the `theAnswer` instance created on the stack, but if it gets passed to a method, the method will receive a copy of it. This behavior is in line with what a C++ developer would expect. Value types are typically used in managed applications to represent lightweight pieces or collections of data, similar to the way built-in types and structs are sometimes used in C++, and primitive types are used in Java.

Values can also live on the managed heap, but not by themselves. The only way this can happen is if a reference type has a field that is a value type. Even though a value type inside of an object lives on the managed heap, it still behaves the same as a value type on the stack when it comes to passing it into a method; that is, the method will receive a copy by default. Any changes made to the value instance are only local changes to the copy unless the value was passed by reference. The following code illustrates these concepts:

```
public struct Coordinate  //this is a value type
{
    public int x;
    public int y;
}

public class EntryPoint //this is a reference type
{
    public static void AttemptToModifyCoord( Coordinate coord ) {
        coord.x = 1;
        coord.y = 3;
    }

    public static void ModifyCoord( ref Coordinate coord ) {
        coord.x = 10;
        coord.y = 10;
    }

    static void Main() {
        Coordinate location;
        location.x = 50;
        location.y = 50;

        AttemptToModifyCoord( location );
        System.Console.WriteLine( "( {0}, {1} )", location.x, location.y );
```

```
        ModifyCoord( ref location );
        System.Console.WriteLine( "( {0}, {1} )", location.x, location.y );
    }
}
```

In the `Main` method, the call to `AttemptToModifyCoord()` actually does nothing to the `location` value. This is because `AttemptToModifyCoord()` modifies a local copy of the value that was made when the method was called. On the contrary, the `location` value is passed by reference into the `ModifyCoord` method. Thus, any changes made in the `ModifyCoord` method are actually made on the `location` value in the calling scope. It's similar to passing a value by a pointer in C++. The output from the example is as follows:

```
( 50, 50 )
( 10, 10 )
```

Enumerations

Enumerations (`enums`) in C# are similar to enumerations in C++, and the defining syntax is almost identical. At the point of use, however, enumerations are slightly different than in C++, since you must fully qualify the values within an enumeration using the enumeration type name. All enumerations are based upon an underlying integral type, which if not specified, defaults to `int`.

■ **Note** The underlying type of the `enum` must be an integral type that is one of the following: `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, or `ulong`.

Each constant that is defined in the enumeration must be defined with a value within the range of the underlying type. If you don't specify a value for an enumeration constant, the value takes the default value of 0 (if it is the first constant in the enumeration) or 1 plus the value of the previous constant. This example is an enumeration based upon a `long`:

```
public enum Color : long
{
   Red,
   Green = 50,
   Blue
}
```

In this example, if I had left off the colon and the `long` keyword after the `Color` type identifier, the enumeration would have been of `int` type. Notice that the value for `Red` is 0, the value for `Green` is 50, and the value for `Blue` is 51.

To use this enumeration, write code such as the following:

```
static void Main() {
   Color color = Color.Red;
   System.Console.WriteLine( "Color is {0}", color.ToString() );
}
```

If you compile and run this code, you'll see that the output actually uses the name of the enumeration rather than the ordinal value 0. The `System.Enum` type's implementation of the `ToString()` method performs this magic.

Many times, you may use enumeration constants to represent bit flags. You can attach an attribute in the `System` namespace called `System.FlagsAttribute` to the enumeration to make this explicit. The attribute gets stored in the metadata, and you can reference it at design time to determine if members of an enumeration are intended to be used as bit flags. Also, you can reference this

attribute in some places at run time—for example, when an enumeration value is converted to a string. Note that `System.FlagsAttribute` doesn't modify the behavior of the values defined by the enumeration. At run time, however, certain components can use the metadata generated by the attribute to process the value differently. This is a great example of how you can use metadata effectively in an aspect-oriented programming (AOP) manner.

■ **Note** AOP, or aspect-oriented software development (AOSD), is a concept originally developed by Gregor Kiczales and his team at Xerox PARC. Object-oriented methodologies generally do a great job of partitioning the functionality, or concerns, of a design into cohesive units. However, some concerns, called *crosscutting concerns*, cannot be modeled easily with standard OO designs. For example, suppose you need to log entry into and exit from various methods. It would be a horrible pain to modify the code for each and every method that you need to log. It would be much easier if you could simply attach a property—or in this case, an attribute—to the method itself so that the runtime would log the method's call when it happens. This keeps you from having to modify the method, and the requirement is applied out-of-band from the method's implementation. Microsoft Transaction Server (MTS) was one of the first widely known examples of AOP.

Using metadata, and the fact that you can attach arbitrary, custom attributes to types, methods, properties, and so on, you can use AOP in your own designs.

Here is an example of a bit flag enumeration:

```
[Flags]
public enum AccessFlags

{
   NoAccess = 0x0,
   ReadAccess = 0x1,
   WriteAccess = 0x2,
   ExecuteAccess = 0x4
}
```

Here is an example of using the `AccessFlags` enumeration:

```
static void Main() {
   AccessFlags access = AccessFlags.ReadAccess |
      AccessFlags.WriteAccess;

   System.Console.WriteLine( "Access is {0}", access );
}
```

If you compile and execute this example, you'll see that the `Enum.ToString` method implicitly invoked by `WriteLine()` does, in fact, output a comma-separated list of all of bits that are set in this value.

Reference Types

The garbage collector (GC) inside the CLR manages everything regarding the placement of objects. It can move objects at any time. When it moves them, the CLR makes sure that the variables that reference them are updated. Normally, you're not concerned with the exact location of the object within the heap, and you don't have to care if it gets moved around or not. There are rare cases, such as when interfacing with native DLLs, when you may need to obtain a direct memory pointer to an object on the heap. It is possible to do that using *unsafe* (or *unmanaged*) code techniques, but that is outside the scope of this book.

■ **Note** Conventionally, the term *object* refers to an instance of a reference type, whereas the term *value* refers to an instance of a value type, but all instances of any type are also derived from type `object`.

Variables of reference type are either initialized by using the `new` operator to create an object on the managed heap, or they are initialized by assignment from another variable of a compatible type. The following code snippet points two variables at the same object:

```
string idTag = "423-XYZ";
string theTag = idTag;
```

Like the Java runtime, the CLR manages all references to objects on the heap. In C++, you have to explicitly delete heap-based objects at some carefully chosen point. But in the managed environment of the CLR, the GC takes care of this for you. This frees you from having to worry about deleting objects from memory and minimizes memory leaks. The GC can determine, at any point in time, how many references exist to a particular object on the heap. If it determines there are none, it is free to start the process of destroying the object on the heap. The previous code snippet includes two references to the same object. You initialize the first one, `idTag`, by creating a string object. You initialize the second one, `theTag`, from `idTag`. The GC won't collect the string object on the heap until both of these references are outside any usable scope. Had the method that this code lives in returned a copy of the reference to whatever called it, then the GC would still have a reference to track even when the method was no longer in scope.

■ **Note** The fundamental way in which objects are treated in the C++ world is reversed in the C# world. In C++, objects are allocated on the stack unless you create them explicitly with the `new` operator, which then returns a pointer to the object on the native heap. In C#, you cannot create objects of a reference type on the stack. They can only live on the heap. So, it's almost as if you were writing C++ code, where you create every object on the heap without having to worry about deleting them explicitly to clean up.

Default Variable Initialization

By default, the C# compiler produces what is called *safe code*. One of the safety concerns is making sure the program doesn't use uninitialized memory. The compiler wants every variable to be set to a value before you use it, so it is useful to know how various variables are initialized.

The default value for references to objects is null. At the point of declaration, you can optionally assign references from the result of a call to the new operator; otherwise, they will be set to null. When you create an object, the runtime initializes its internal fields. Fields that are references to objects are initialized to null, of course. Fields that are value types are initialized by setting all bits of the value type to zero. Basically, you can imagine that all the runtime is doing is setting the bits of the underlying storage to 0. For references to objects, that equates to a null reference, and for value types, that equates to a value of zero.

For value types that you declare on the stack, the compiler does not zero-initialize the memory automatically. However, it does make sure that you initialize the memory before the value is used.

■ **Note** Since enumerations are actually value types, you should always declare an enumeration member that equates to zero, even if the name of the member is `InvalidValue` or `None` and is otherwise meaningless. If an enumeration is declared as a field of a class, instances of that class will have the field set to zero upon default initialization. Declaring a member that equates to zero allows users of your enumeration to deal with this case easily.

Type Conversion

Many times, it's necessary to convert instances of one type to another. In some cases, the compiler does this conversion implicitly whenever a value of one type is assigned from another type that, when converted to the assigned type, will not lose any precision or magnitude. In cases where precision could be lost, an explicit conversion is required. For reference types, the conversion rules are analogous to the pointer conversion rules in C++.

The semantics of type conversion are similar to both C++ and Java. Explicit conversion is represented using the familiar casting syntax that all of them inherited from C—that is, the type to convert to is placed in parentheses before whatever needs to be converted:

```
int defaultValue = 12345678;
long value = defaultValue;
int  smallerValue = (int) value;
```

In this code, the (int) cast is required to be explicit since the underlying size of an int is smaller than a long. Thus, it's possible that the value in the long may not fit into the space available to the int. The assignment from the defaultValue variable to the value variable requires no cast, since a long has more storage space than an int. If the conversion will lose magnitude, it's possible that the conversion may throw an exception at run time. The general rule is that implicit conversions are guaranteed never to throw an exception, whereas explicit conversions may throw exceptions.

The C# language provides the facilities to define custom implicit and explicit conversion operators to various types for your user-defined value types, or structs. Chapter 6 covers these in more detail. The exception requirements for built-in conversion operators apply to user-defined conversion operators. Namely, implicit conversion operators are guaranteed never to throw.

Conversion to and from reference types models that of Java and of conversions between pointers in C++. For example, a reference to type DerivedType can be implicitly converted to a reference to type BaseType if DerivedType is derived from BaseType. However, you must explicitly cast a conversion in the opposite direction. Also, an explicit cast may throw a System.InvalidCastException if the CLR cannot perform the conversion at run time.

One kind of implicit cast is available in C# that is not easily available in C++, mainly because of the default value semantics of C++. It is possible to implicitly convert from an array of one reference type to an array of another reference type, given that the target reference type is able to be implicitly converted from the source reference type and the arrays are of the same dimension. For example, the following conversion is valid:

```
public class EntryPoint
{
    static void Main() {
        string[] names = new string[4];
        object[] objects = names;  //implicit conversion statement

        string[] originalNames =
            (string[]) objects;  // explicit conversion statement
    }
}
```

Because System.String derives from System.Object and, therefore, is implicitly convertible to System.Object, this implicit conversion of the string array names into the object array objects variable is valid. However, to go in the other direction, as shown, requires an explicit cast that may throw an exception at run time if the conversion fails.

Keep in mind that implicit conversions of arguments may occur during method invocation if arguments must be converted to match the types of parameters. If you cannot make the conversions implicitly, you must cast the arguments explicitly.

Finally, another type of common conversion is a *boxing conversion*. Boxing conversions are required when you must pass a value type to a method or assign it to a variable as a reference type. What happens is that an object is allocated dynamically on the heap that contains a field of the value's type. The value is then copied into this field. I cover boxing in C# extensively in Chapter 4. The following code demonstrates boxing:

```
public class EntryPoint
{
    static void Main() {
        int employeeID = 303;
        object boxedID = employeeID;

        employeeID = 404;
        int unboxedID = (int) boxedID;

        System.Console.WriteLine( employeeID.ToString() );
        System.Console.WriteLine( unboxedID.ToString() );
    }
}
```

At the point where the `object` variable `boxedID` is assigned from the `int` variable `employeeID`, boxing occurs. A heap-based object is created and the value of `employeeID` is copied into it. This bridges the gap between the value type and the reference type worlds within the CLR. The `boxedID` object actually contains a copy of the `employeeID` value. I demonstrate this point by changing the original `employeeID` value after the boxing operation. Before printing out the values, I unbox the value and copy the value contained in the object on the heap back into another `int` on the stack. Unboxing requires an explicit cast in C#, because it can fail with a bad cast exception.

as and is Operators

Given the fact that explicit conversion can fail by throwing exceptions, times arise when you may want to test the type of a variable without performing a cast and seeing if it fails or not. Testing this way is tedious and inefficient, and exceptions are generally expensive at run time. For this reason, C# has two operators that come to the rescue, and they are guaranteed not to throw an exception:

- `as`
- `is`

The `is` operator results in a Boolean that determines if you can convert a given expression to the given type as either a reference conversion or a boxing or unboxing operation. For example, consider the following code:

```
using System;

public class EntryPoint
{
    static void Main() {
        String derivedObj = "Dummy";
        Object baseObj1 = new Object();
        Object baseObj2 = derivedObj;

        Console.WriteLine( "baseObj2 {0} String",
                           baseObj2 is String ? "is" : "isnot" );
        Console.WriteLine( "baseObj1 {0} String",
                           baseObj1 is String ? "is" : "isnot" );
        Console.WriteLine( "derivedObj {0} Object",
                           derivedObj is Object ? "is" : "isnot" );
```

```
        int j = 123;
        object boxed = j;
        object obj = new Object();

        Console.WriteLine( "boxed {0} int",
                           boxed is int ? "is" : "isnot" );
        Console.WriteLine( "obj {0} int",
                           obj is int ? "is" : "isnot" );
        Console.WriteLine( "boxed {0} System.ValueType",
                           boxed is ValueType ? "is" : "isnot" );

    }
}
```

The output from this code is as follows:

```
baseObj2 is String
baseObj1 isnot String
derivedObj is Object
boxed is int
obj isnot int
boxed is System.ValueType
```

As mentioned previously, the `is` operator only considers reference conversions. This means that it won't consider any user-defined conversions that are defined on the types.

The `as` operator is similar to the `is` operator, except it returns a reference to the target type. Because it is guaranteed never to throw an exception, it simply returns a null reference if the conversion cannot be made. Similar to the `is` operator, the `as` operator only considers reference conversions or boxing conversions. Consider the following code:

```
using System;

public class BaseType {}

public class DerivedType : BaseType {}

public class EntryPoint {
    static void Main() {
        DerivedType derivedObj = new DerivedType();
        BaseType baseObj1 = new BaseType();
        BaseType baseObj2 = derivedObj;

        DerivedType derivedObj2 = baseObj2 as DerivedType;
        if( derivedObj2 != null ) {
            Console.WriteLine( "Conversion Succeeded" );
        } else {
            Console.WriteLine( "Conversion Failed" );
        }

        derivedObj2 = baseObj1 as DerivedType;
        if( derivedObj2 != null ) {
            Console.WriteLine( "Conversion Succeeded" );
        } else {
            Console.WriteLine( "Conversion Failed" );
        }
```

```
        BaseType baseObj3 = derivedObj as BaseType;
        if( baseObj3 != null ) {
            Console.WriteLine( "Conversion Succeeded" );
        } else {
            Console.WriteLine( "Conversion Failed" );
        }
    }
}
```

The output from this code is as follows:

```
Conversion Succeeded
Conversion Failed
Conversion Succeeded
```

Sometimes you need to test whether a variable is of a certain type and, if it is, then do some sort of operation on the desired type. You could test the variable for the desired type using the `is` operator and then, if true, cast the variable to the desired type. However, doing it that way is inefficient. The better approach is to follow the idiom of applying the `as` operator to obtain a reference of the variable with the desired type, and then test whether the resultant reference is null to test if the conversion succeeded. That way, you perform only one type lookup operation instead of two.

Generics

Support for generics is one of the most exciting new additions to the C# language. Using the generic syntax, you can define a type that depends upon another type that is not specified at the point of definition, but rather at the point of usage of the generic type. For example, imagine a collection type. Collection types typically embody things such as lists, queues, and stacks. The collection types that have been around since the .NET 1.0 days are adequate for containing any type in the CLR, since they contain `Object` instances and everything derives from `Object`. However, all of the type information for the contained type is thrown away, and the compiler's power to catch type errors is rendered useless. You must cast every type reference that you obtain from these collections into the type you think it should be, and that could fail at run time. Also, the original collection types can contain a heterogeneous blend of types rather than force the user to insert only instances of a certain type. You could go about fixing this problem by writing types such as `ListOfIntegers` and `ListOfStrings` for each type you want to contain in a list. However, you will quickly find out that most of the management code of these lists is similar, or generic, across all of the custom list types. The key word is *generic*. Using generic types, you can declare an open (or generic) type and only have to write the common code once. The users of your type can then specify which type the collection will contain at the point they decide to use it.

Additionally, using generics involves efficiency gains. The concept of generics is so large that I've devoted Chapter 10 entirely to their declaration and usage. However, I believe it's important to give you a taste of how to use a generic type now, since I mention them several times prior to Chapter 10.

■ **Note** The generic syntax will look familiar to those who use C++ templates. However, it's important to note that there are significant behavioral differences, which I'll explain in Chapter 10.

The most common use of generics is during the declaration of collection types. For example, take a look at the following code:

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;

class EntryPoint
{
    static void Main() {
        Collection<int> numbers =
            new Collection<int>();
        numbers.Add( 42 );
        numbers.Add( 409 );

        Collection<string> strings =
            new Collection<string>();
        strings.Add( "Joe" );
        strings.Add( "Bob" );

        Collection< Collection<int> > colNumbers
            = new Collection<Collection<int>>();
        colNumbers.Add( numbers );

        IList<int> theNumbers = numbers;
        foreach( int i in theNumbers ) {
            Console.WriteLine( i );
        }
    }
}
```

This example shows usage of the generic `Collection` type. The telltale sign of generic type usage is the angle brackets surrounding a type name. In this case, I have declared a collection of integers, a collection of strings, and, to show an even more complex generic usage, a collection of collections of integers. Also, notice that I've also shown the declaration for a generic interface, namely, `IList<>`.

When you specify the type arguments for a generic type by listing them within the angle brackets, as in `Collection<int>`, you are declaring a closed type. In this case, `Collection<int>` only takes one type parameter, but had it taken more, then a comma would have separated the type arguments. When the CLR first encounters this type declaration, it will generate the closed type based on the generic type and the provided type arguments. Using closed types formed from generics is no different than using any other type, except that the type declaration uses the special angle bracket syntax to form the closed type.

Now that you've seen a glimpse of what generics look like, you should be prepared for the casual generic references mentioned prior to Chapter 10.

Namespaces

C#, like C++ and analogous to Java packages, supports namespaces for grouping components logically. Namespaces help you avoid naming collisions between your identifiers.

Using namespaces, you can define all of your types such that their identifiers are qualified by the namespace that they belong to. You have already seen namespaces in action in many of the examples so far. For instance, in the "Hello World!" example from Chapter 1, you saw the use of the `Console` class, which lives in the .NET Framework Class Library's `System` namespace and whose fully qualified name is `System.Console`. You can create your own namespaces to organize components. The general recommendation is that you use some sort of identifier, such as your organization's name, as the top-level namespace, and then more specific library identifiers as nested namespaces.

Namespaces provide an excellent mechanism with which you can make your types more discoverable, especially if you're designing libraries meant for consumption by others. For example, you can create a general namespace such as `MyCompany.Widgets`, where you put the most commonly used types of widgets. Then you can create a `MyCompany.Widgets.Advanced` namespace where you place all of the less commonly used, advanced types. Sure, you could place them all in one namespace. However, users could become confused when browsing the types and see all of the types they least commonly use mixed in with the ones they use the most.

■ **Note** When picking a name for your namespace, the general guideline suggests that you name it using the formula `<CompanyName>.<Technology>.*`, such that the first two dot-separated portions of the namespace name start with your company name followed by your company's technology. Then you can further subclassify the namespace. You can see examples of this in the .NET Framework—for example, the `Microsoft.Win32` namespace.

Defining Namespaces

The syntax for declaring a namespace is simple. The following code shows how to declare a namespace named `Acme`:

```
namespace Acme
{
   class Utility {}
}
```

Namespaces don't have to live in only one compilation unit (i.e., the C# source code file). In other words, the same namespace declaration can exist in multiple C# files. When everything is compiled, the set of identifiers included in the namespace is a union of all of the identifiers in each of the namespace declarations. In fact, this union spans across assemblies. If multiple assemblies contain types defined in the same namespace, the total namespace consists of all of the identifiers across all the assemblies that define the types.

It's possible to nest namespace declarations. You can do this in one of two ways. The first way is obvious:

```
namespace Acme
{
   namespace Utilities
   {
      class SomeUtility {}
   }
}
```

Given this example, to access the `SomeUtility` class using its fully qualified name, you must identify it as `Acme.Utilities.SomeUtility`. The following example demonstrates an alternate way of defining nested namespaces:

```
namespace Acme
{
}

namespace Acme.Utilities
{
class SomeUtility {}
}
```

The effect of this code is the exact same as the previous code. In fact, you may omit the first empty namespace declaration for the `Acme` namespace. I left it there only for demonstration purposes to point out that the `Utilities` namespace declaration is not physically nested within the `Acme` namespace declaration.

Any types that you declare outside of a namespace become part of the global namespace.

■ **Note** You should always avoid defining types in the global namespace. Such practice is known as polluting the global namespace and is widely considered poor programming practice. This should be obvious since there would be no way to protect types defined by multiple entities from potential naming collisions.

Using Namespaces

In the "Hello World!" example of Chapter 1, I quickly touched on the options available for using namespaces. Let's examine some code that uses the `SomeUtility` class that I defined in the previous section:

```
public class EntryPoint
{
   static void Main() {
      Acme.Utilities.SomeUtility util =
         new Acme.Utilities.SomeUtility();
   }
}
```

This usage of always qualifying names fully is rather verbose and might eventually lead to a bad case of carpal tunnel syndrome. The `using` namespace directive avoids this. It tells the compiler that you're using an entire namespace in a compilation unit or another namespace. What the `using` keyword does is effectively import all of the names in the given namespace into the enclosing namespace, which could be the global namespace of the compilation unit. The following example demonstrates this:

```
using Acme.Utilities;

public class EntryPoint
{
   static void Main() {
      SomeUtility util = new SomeUtility();
   }
}
```

The code is now much easier to deal with and somewhat easier to read. The `using` directive, because it is at the global namespace level, imports the type names from `Acme.Utilities` into the global namespace. Sometimes when you import the names from multiple namespaces, you may still have naming conflicts if both imported namespaces contain types with the same name. In this case, you can import individual types from a namespace, creating a naming alias. This technique is available via namespace aliasing in C#. Let's modify the usage of the `SomeUtility` class so that you alias only the `SomeUtility` class rather than everything inside the `Acme.Utilities` namespace:

```
namespace Acme.Utilities
{
   class AnotherUtility() {}
}

using SomeUtility = Acme.Utilities.SomeUtility;
```

```
public class EntryPoint
{
   static void Main() {
      SomeUtility util = new SomeUtility();
      Acme.Utilities.AnotherUtility =
            new Acme.Utilities.AnotherUtility();
   }
}
```

In this code, the identifier `SomeUtility` is aliased as `Acme.Utilities.SomeUtility`. To prove the point, I augmented the `Acme.Utilities` namespace and added a new class named `AnotherUtility`. This class must be fully qualified in order for you to reference it, since no alias is declared for it. Incidentally, it's perfectly valid to give the previous alias a different name than `SomeUtility`. Although giving the alias a different name may be useful when trying to resolve a naming conflict, it's generally better to alias it using the same name as the original class name in order to avoid maintenance confusion in the future.

■ **Note** If you follow good partitioning principles when defining your namespaces, you shouldn't have to deal with this problem. It is bad design practice to create namespaces that contain a grab bag of various types covering different groups of functionality. Instead, you should create your namespaces with intuitively cohesive types. In fact, many times in the .NET Framework, you'll see a namespace with some general types for the namespace included in it, with more advanced types contained in a subnamespace named `Advanced`. In many respects, when creating libraries, these guidelines mirror the principle of discoverability applied when creating intuitive user interfaces. In other words, name and group your types intuitively and make them easily discoverable.

Control Flow

Like C, C++, and Java, the C# language contains all the usual suspects when it comes to control flow structure. C# even implements the dastardly `goto` statement.

if-else, while, do-while, and for

The `if-else` construct within C# is identical to those in C++ and Java. As a stylistic recommendation, I'm a proponent of always using blocks in `if` statements, or any other control flow statement as described in the following sections, even when they contain only one statement, as in the following example:

```
if( <test condition> ) {
   Console.WriteLine( "You are here." );
}
```

The `while`, `do`, and `for` statements are identical to those in C++ and Java.

switch

The syntax of the C# `switch` statement is very similar to the C++ and Java `switch` syntax. The main difference is that the C# `switch` doesn't allow falling through to the next section. It requires a `break` (or other transfer of control) statement to end each section. I believe this is a great thing. Countless hard-to-spot bugs exist in C++ and Java due to developers forgetting a `break` statement or rearranging the order of sections within a `switch` when one of them falls through to another. In C#, the compiler will immediately complain with an error if it finds a section that falls through to the next. The one exception to this rule is that you can have multiple switch labels (using the `case` keyword) per switch

section, as shown in the following code snippet. You can also simulate falling through sections with the `goto` statement:

```
switch( k ) {
   case 0:
      Console.WriteLine( "case 0" );
      goto case 1;
   case 1:
   case 2:
      Console.WriteLine( "case 1 or 2" );
      break;
}
```

Notice that each one of the cases has a form of *jump* statement that terminates it. Even the last case must have one. Many C++ and Java developers would omit the `break` statement in the final section because it would just fall through to the end of the `switch` anyway. Again, the beauty of the "no fall-through" constraint is that even if a developer maintaining the code at a later date whimsically decides to switch the ordering of the labels, no bugs can be introduced, unlike in C++ and Java. Typically, you use a `break` statement to terminate `switch` sections, and you can use any statement that exits the section. These include `throw` and `return`, as well as `continue` if the `switch` is embedded within a loop where the `continue` statement makes sense.

foreach

The `foreach` statement allows you to iterate over a collection of objects in a syntactically natural way. Note that you can implement the same functionality using a `while` loop. However, this can be ugly, and iterating over the elements of a collection is such a common task that `foreach` syntax is a welcome addition to the language. If you had an array (or any other type of collection) of strings, for example, you could iterate over each string using the following code:

```
static void Main() {
   string[] strings = new string[5];
   strings[0] = "Bob";
   strings[1] = "Joe";
   foreach( string item in strings ) {
      Console.WriteLine( "{0}", item );
   }
}
```

Within the parentheses of the `foreach` loop, you declare the type of your iterator variable. In this example, it is a string. Following the declaration of the iterator type is the identifier for the collection to iterate over. You may use any object that implements the Collection pattern. Chapter 9 covers collections in greater detail, including what sorts of things a type must implement in order to be considered a collection. Naturally, the elements within the collection used in a `foreach` statement must be convertible, using an explicit conversion, to the iterator type. If they're not, the `foreach` statement will throw an `InvalidCastException` at run time. If you'd like to experience this inconvenience yourself, try this modification to the previous example:

```
static void Main() {
   object[] strings = new object[5];
   strings[0] = 1;
   strings[1] = 2;
   foreach( string item in strings ) {
      Console.WriteLine( "{0}", item );
   }
}
```

Note, however, that it's invalid for the code embedded in a `foreach` statement to modify the iterator variable at all. It should be treated as read-only. That means you cannot pass the iterator variable as an `out` or `ref` parameter to a method, either. If you try to do any of these things, the compiler will quickly alert you to the error of your ways.

break, continue, goto, return, and throw

C# includes a set of familiar statements that unconditionally transfer control to another location. These include `break`, `continue`, `goto`, `return`, and `throw`. Their syntax should be familiar to any C++ or Java developer (though Java has no `goto`). Their usage is essentially identical in all three languages.

Summary

This chapter introduced the C# syntax while pointing out that C#, like other similar object-oriented languages, is a strongly typed language. For these languages, you want to utilize the type-checking engine of the compiler to find as many errors as possible at compile time rather than find them later at run time. In the CLR, types are classified as either value types or reference types, and each category comes with its own stipulations, which I'll continue to dig into throughout this book. I also introduced namespaces and showed how they help keep the global namespace from getting polluted with too many types whose names could conflict. And finally, I showed how control statements work in C#, which is similar to how they work in C++ and Java.

In the next chapter, I'll dig deeper into the structure of classes and structs, while highlighting the behavioral differences of instances thereof.