Learn C# Errata

The following pages show errors from the original edition, published in July 2008, corrected in red. Future editions of this book will be printed with these corrections. We apologize for any inconvenience these errors have caused.

page 40

3-6-3 The Assignment Operators

The = assignment operator is used for assigning a value, expression, or variable to a variable. The following table shows the use of assignment operators combined with arithmetic operators.

Table 3-5: The assignment operators

Compound Assignment Operator	Example Statement	Equivalent Statement
*=	x *= y;	x = x * y;
/=	x /= y;	x = x / y;
%=	x %= y;	x = x % y;
+=	x += y;	x = x + y;
–=	x –= y;	x = x – y;

As you can see in the above table, each example in the Example Statement column is equivalent to a statement in the Equivalent Statement column. For example, the statement:

```
x *= y;
```

means multiply x by y and store the result in x.

The statement:

```
x %= y;
```

means divide x by y and store the remainder of the division in x.

page 53

3-9 The Nullable Types

The nullable types were added to value types with C# 2005. This feature enables you to assign the value null to a value-type variable. You need this with databases where a variable can assume any value including null. The nullable variable is declared by using the ? symbol next to the type name, like this:

```
myType? myVariable;
```

where myType is one of the value types including the **struct** type. It is called the underlying type of the nullable type. For example, consider the following statement:

```
int? myInt = null;
```

In this statement, the underlying type is **int**. This means that the variable myInt can accept all the values that can be assigned to **int** in addition to null, which means "not used" or "empty." In the following example,

```
bool? myBool;
```

myBool can assume one of the values true, false, or null. In databases, this feature is important because a database field may contain null to indicate that the variable is not defined. This is the same concept used with reference types where the value null has been used to indicate that a variable is not initialized.

page 68

4-1-2-4 The Bitwise Operators

The logical operators & and | can be used with integral operands, in which case they compute the bitwise AND and the bitwise OR of their operands. The following examples perform bitwise AND and OR operations on two integers (21 and 4) and display the results in several formats:

```
// Hexadecimal:
Console.WriteLine("0x{0:x}",0x15 | 0x4);   // 0x15
Console.WriteLine("0x{0:x}",0x15 & 0x4);   // 0x4
// Decimal:
Console.WriteLine(21 | 4);   // 21
Console.WriteLine(21 & 4);   // 4
```

page 79

Under section 4-6-1-2, in the last code segment:

```
for (i = 0, j = 10; i<=j, i++; j = j-1)
```

should be:

```
for (i = 0, j = 10; i<=j; i++, j = j-1)
```

page 186

The job of the **new** modifier here is to hide the member with the same name in the base class. A method that uses the **new** modifier hides properties, fields, and types with the same name. It also hides the methods with the same signatures. In general, declaring a member in the derived class would hide any members in the base class with the same name. If you declare MyMethod in the above example without the **new** modifier, it still works, but you will get a compiler warning:

'MyDerivedClass.MyMethod()' hides inherited member 'MyBaseClass.MyMethod()'. Use the new keyword if hiding was intended.

page 275

12-1 What Are Generics?

Generics, a new feature that was added to C# 2005, enable the programmer to design classes or function members and postpone the definition of types until the class is instantiated. By adding this feature, C# made a big step in code reuse, especially in the field of collections.

When you use a collection such as **Stack**, you store the items as objects because the type **object** can hold any kind of data. Consider this example where you store some **double** numbers in a **Stack** collection:

```
Stack myStack = new Stack ();
myStack.Push (4.5);
myStack.Push (2.1);
myStack.Push (3.2);
```

page 276

When you create this collection, the compiler automatically boxes the **double** numbers to convert them to the **object** type. When you retrieve the data, you have to cast the numbers to the **double** type. For example:

```
foreach(object obj in myStack)
   Console.WriteLine((double)obj * 3.14);
```

page 331

14-2 **Implicitly Typed Local Variables**

You can declare a variable with the **var** keyword and let the compiler infer the type from the expression used to initialize the variable. For example:

```
var myVariable = 1.25;
var yourVariable = "Hello, World!";
var myArray = new int[] { 1, 2, 4, 8, 64 };
```

The C# compiler can infer that the type of myVariable is **double**, the type of yourVariable is **string**, and that of myArray is **int**. In other words, the three statements above generate the same result as the following statements:

```
double myVariable = 1.25;
string yourVariable = "Hello, World!";
int[] myArray = new int[] { 1, 2, 4, 8, 64 };
```

page 341

Example 14-5

```
// Example 14-5.cs
// Anonymous types

using System;
using System.Collections.Generic;
using System.Text;
using System.Query;
using System.Xml.XLinq;
using System.Data.DLinq;

namespace AnonTypes
{
   class MyClass
   {
      static void Main(string[] args)
      {
         // Declare an anonymous type:
         var obj1 = new { Title = "Organic Babies", ISBN =
                        "5-1234-5678-x" };

         // Display the contents:
         Console.WriteLine("Title: {0}\nISBN: {1}", obj1.Title,
                        obj1.ISBN);
         Console.ReadLine();
      }
```

```
    }
}
```

Output:

```
Title: Organic Babies
ISBN: 5-1234-5678-x
```

In order to see the class tree, run ILDASM on the executable file generated from the compilation (14-5.exe) by using the following command line:

```
ILDASM 14-5.exe
```

page 344

14-6 Implicitly Typed Arrays

The implicitly typed local variables can be extended to declaring arrays. By examining the members of an array in the initialization expression, the compiler can infer the type of the array elements.

The following is the regular C# 2.0 code to declare variables of different types:

```
int[] a = new[] { 1, 2, 4, 8, 16, 32, 64 };
double[] b = new[] { 3, 3.14, 2.7 };
bool[] c = new [] { true, false };
string[] d = new[] { "Hello,", "world!" };
```