CHAPTER 7

Constants and Fields

In this chapter:

In this chapter, I'll show you how to add data members to a type. Specifically, we'll look at constants and fields.

Constants

A *constant* is a symbol that has a never-changing value. When defining a constant symbol, its value must be determinable at compile time. The compiler then saves the constant's value in the assembly's metadata. This means that you can define a constant only for types that your compiler considers primitive types. In C#, the following types are primitives and can be used to define constants: `Boolean`, `Char`, `Byte`, `SByte`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Single`, `Double`, `Decimal`, and `String`. However, C# also allows you to define a constant variable of a non-primitive type if you set the value to `null`.

```
using System;

public sealed class SomeType {
   // SomeType is not a primitive type but C# does allow
   // a constant variable of this type to be set to 'null'.
   public const SomeType Empty = null;
}
```

Because a constant value never changes, constants are always considered to be part of the defining type. In other words, constants are always considered to be static members, not instance members. Defining a constant causes the creation of metadata.

When code refers to a constant symbol, compilers look up the symbol in the metadata of the assembly that defines the constant, extract the constant's value, and embed the value in the emitted Intermediate Language (IL) code. Because a constant's value is embedded directly in code, constants don't require any memory to be allocated for them at run time. In addition, you can't get the address of a constant and you can't pass a constant by reference. These constraints also mean that constants don't have a good cross-assembly versioning story, so you should use them only when you know that

the value of a symbol will never change. (Defining `MaxInt16` as `32767` is a good example.) Let me demonstrate exactly what I mean. First, take the following code and compile it into a DLL assembly.

```
using System;

public sealed class SomeLibraryType {
   // NOTE: C# doesn't allow you to specify static for constants
   // because constants are always implicitly static.
   public const Int32 MaxEntriesInList = 50;
}
```

Then use the following code to build an application assembly.

```
using System;

public sealed class Program {
   public static void Main() {
      Console.WriteLine("Max entries supported in list: "
         + SomeLibraryType.MaxEntriesInList);
   }
}
```

You'll notice that this application code references the `MaxEntriesInList` constant defined in the `SomeLibraryType` class. When the compiler builds the application code, it sees that `MaxEntriesInList` is a constant literal with a value of `50` and embeds the `Int32` value of `50` right inside the application's IL code, as you can see in the following IL code. In fact, after building the application assembly, the DLL assembly isn't even loaded at run time and can be deleted from the disk because the compiler does not even add a reference to the DLL assembly in the application's metadata.

```
.method public hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       25 (0x19)


  .maxstack  8
  IL_0000:  nop
  IL_0001:  ldstr      "Max entries supported in list: "
  IL_0006:  ldc.i4.s   50
  IL_0008:  box        [mscorlib]System.Int32
  IL_000d:  call       string [mscorlib]System.String::Concat(object, object)
  IL_0012:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0017:  nop
  IL_0018:  ret
} // end of method Program::Main
```

This example should make the versioning problem obvious to you. If the developer changes the `MaxEntriesInList` constant to 1000 and only rebuilds the DLL assembly, the application assembly is not affected. For the application to pick up the new value, it will have to be recompiled as well. You can't use constants if you need to have a value in one assembly picked up by another assembly at run time (instead of compile time). Instead, you can use `readonly` fields, which I'll discuss next.

Fields

A *field* is a data member that holds an instance of a value type or a reference to a reference type. Table 7-1 shows the modifiers that can be applied to a field.

TABLE 7-1 Field Modifiers

CLR Term	C# Term	Description
Static	`static`	The field is part of the type's state, as opposed to being part of an object's state.
Instance	(default)	The field is associated with an instance of the type, not the type itself.
InitOnly	`readonly`	The field can be written to only by code contained in a constructor method.
Volatile	`volatile`	Code that accessed the field is not subject to some thread-unsafe optimizations that may be performed by the compiler, the CLR, or by hardware. Only the following types can be marked `volatile`: all reference types, `Single`, `Boolean`, `Byte`, `SByte`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Char`, and all enumerated types with an underlying type of `Byte`, `SByte`, `Int16`, `UInt16`, `Int32`, or `UInt32`. Volatile fields are discussed in Chapter 29, "Primitive Thread Synchronization Constructs."

As Table 7-1 shows, the common language run time (CLR) supports both type (static) and instance (nonstatic) fields. For type fields, the dynamic memory required to hold the field's data is allocated inside the type object, which is created when the type is loaded into an AppDomain (see Chapter 22, "CLR Hosting and AppDomains"), which typically happens the first time any method that references the type is just-in-time (JIT)–compiled. For instance fields, the dynamic memory to hold the field is allocated when an instance of the type is constructed.

Because fields are stored in dynamic memory, their value can be obtained at run time only. Fields also solve the versioning problem that exists with constants. In addition, a field can be of any data type, so you don't have to restrict yourself to your compiler's built-in primitive types (as you do for constants).

The CLR supports `readonly` fields and `read/write` fields. Most fields are `read/write` fields, meaning the field's value might change multiple times as the code executes. However, `readonly` fields can be written to only within a constructor method (which is called only once, when an object is first created). Compilers and verification ensure that `readonly` fields are not written to by any method other than a constructor. Note that reflection can be used to modify a `readonly` field.

Let's take the example from the "Constants" section and fix the versioning problem by using a static `readonly` field. Here's the new version of the DLL assembly's code.

```
using System;

public sealed class SomeLibraryType {
   // The static is required to associate the field with the type.
   public static readonly Int32 MaxEntriesInList = 50;
}
```

This is the only change you have to make; the application code doesn't have to change at all, although you must rebuild it to see the new behavior. Now when the application's `Main` method runs, the CLR will load the DLL assembly (so this assembly is now required at run time) and grab the value of the `MaxEntriesInList` field out of the dynamic memory allocated for it. Of course, the value will be 50.

Let's say that the developer of the DLL assembly changes the 50 to 1000 and rebuilds the assembly. When the application code is re-executed, it will automatically pick up the new value: 1000. In this case, the application code doesn't have to be rebuilt—it just works (although its performance is adversely affected). A caveat: this scenario assumes that the new version of the DLL assembly is not strongly named and the versioning policy of the application is such that the CLR loads this new version.

The following example shows how to define a `readonly` static field that is associated with the type itself, as well as `read/write` static fields and `readonly` and `read/write` instance fields, as shown here.

```
public sealed class SomeType {
   // This is a static read-only field; its value is calculated and
   // stored in memory when this class is initialized at run time.
   public static readonly Random s_random = new Random();

   // This is a static read/write field.
   private static Int32 s_numberOfWrites = 0;

   // This is an instance read-only field.
   public readonly String Pathname = "Untitled";

   // This is an instance read/write field.
   private System.IO.FileStream m_fs;

   public SomeType(String pathname) {
      // This line changes a read-only field.
      // This is OK because the code is in a constructor.
      this.Pathname = pathname;
   }

   public String DoSomething() {
      // This line reads and writes to the static read/write field.
      s_numberOfWrites = s_numberOfWrites + 1;

      // This line reads the read-only instance field.
      return Pathname;
   }
}
```

In this code, many of the fields are initialized inline. C# allows you to use this convenient inline initialization syntax to initialize a class's constants and `read/write` and `readonly` fields. As you'll see in Chapter 8, "Methods," C# treats initializing a field inline as shorthand syntax for initializing the field in a constructor. Also, in C#, there are some performance issues to consider when initializing fields by using inline syntax versus assignment syntax in a constructor. These performance issues are discussed in Chapter 8 as well.

Important When a field is of a reference type and the field is marked as `readonly`, it is the reference that is immutable, not the object that the field refers to. The following code demonstrates.

```
public sealed class AType {
   // InvalidChars must always refer to the same array object
   public static readonly Char[] InvalidChars = new Char[] { 'A', 'B', 'C' };
}

public sealed class AnotherType {
   public static void M() {
      // The lines below are legal, compile, and successfully
      // change the characters in the InvalidChars array
      AType.InvalidChars[0] = 'X';
      AType.InvalidChars[1] = 'Y';
      AType.InvalidChars[2] = 'Z';

      // The line below is illegal and will not compile because
      // what InvalidChars refers to cannot be changed
      AType.InvalidChars = new Char[] { 'X', 'Y', 'Z' };
   }
}
```