Chapter 3
The .NET Type System

This chapter provides a tour through the .NET types needed for Compact Framework programming. [Comment 3.1]

Author's Note: In this review chapter C#-Specific Text is highlighted in YELLOW.

.NET programming provides a high level of interoperability between programming languages. Interoperability is achieved by a set of common standards. Among the .NET technologies that Microsoft submitted to ECMA[1], a European standards body, was the *Common Language Infrastructure (CLI)*. The CLI contains basic infrastructure elements of .NET, but excludes the high-level Windows Forms and Web Forms classes. To interoperate with other .NET-compatible languages, a compiler must comply with the CLI. This chapter focuses on one of those standards: the Common Type System (CTS). [Comment 3.3]

The .NET type system provides the foundation for a strongly-typed, object-oriented programming environment. Strong-typing codifies a programmer's intent for how each data item by choosing the most appropriate type. Once the type is chosen, errors can be detected at both build-time (by the compiler) and at runtime (by the runtime system). The .NET execution engine takes this one step further, supporting *verifiable code* which relies on strong typing to insure that the code does not access out-of-bounds memory. [Comment 3.4]

An object-oriented approach organizes the raw material of software – meaning the code that runs and the data that is operated on – as *objects*. The content and behavior of an object is defined, in turn, by its type – that is, by its *class*. Programmers use classes to organize code and data. Using the terminology of .NET, we say that a class is a set of *methods* and *fields*, where a method is the most fundamental type of code, and a field is the most fundamental type of data. [Comment 3.5]

This chapter reviews the elements of the .NET framework type system that Compact Framework programmers are likely to need. Our intent is not to teach C# programming – there are other books better suited for that purpose. Nor is our intent to provide a complete discussion of the intricacies of .NET internals, a fascinating subject which is addressed in other, more specialized, books[2]. [Comment 3.6]

Because this book focuses on C#, we expect readers to have learned C# programming from other sources. For readers with a background in C++ and object-oriented theory, we provide some boxed discussions labeled "For C++ Programmers." For readers with a background in VB6 programming, we provide some boxed discussions labeled "For VB6 Programmers." [Comment 3.7]

Namespaces

To help organize the large number of types that are inevitable in a large, multi-faceted programming environment, C# programmers use *namespaces*. A namespace provides part of an object's identity; it is the family name for a set of types. For example, the `System.Drawing` namespace is the family of types used to create graphical output. Within that namespace are two sibling classes, `Brush` and `Pen`. The fully-qualified names for these types are

[1] ECMA web site: www.ecma-international.org
[2] We recommend *Essential .NET: The Common Language Runtime* by Don Box with Chris Sells, Addison-Wesley, 2003.

`System.Drawing.Brush` and `System.Drawing.Pen`. The fully-qualified name gives these types a unique identity among the vast set of .NET-compatible types. [Comment 3.8]

Every .NET program and library file contains *metadata*, a dictionary of the types defined in a module[3]. At build time, a compiler creates metadata, which the execution engine uses at runtime. Objects are instantiated using type information, and code verified for compatibility with the supported types. Every .NET type is defined in an executable module file as metadata. [Comment 3.9]

Namespaces, Classes, Methods, and Fields

Namespaces are optional, but we like using them to help organize our code. For class libraries, the convention is that the name of a namespace should also be the name of the DLL file. For example, in the Compact Framework, the `System.Windows.Forms` namespace resides inside the file `System.Windows.Forms.dll`. Microsoft suggests combining these elements for namespace names in public libraries: company name, technology name, feature, and design. In the examples for this book, we use the `YaoDurant` namespace for shared libraries, or for source files explicitly built for reuse. [Comment 3.10]

In .NET, everything is a value or an object. As a result, stand-alone data – what C++ programmers and VB programmers call global variables – does not exist. Stand-alone global functions also do not exist. Instead, both code and data must live within a class. [Comment 3.11]

The two fundamental .NET types are *fields* and *methods*. The tiny fragment in listing 3-1 shows a field and a method defined in a class. The class, in turn, lives in a namespace, `YaoDurant.Identity`. [Comment 3.12]

Listing 3-1 Namespaces contain classes, which contain fields and methods [Comment 3.13]

```
namespace YaoDurant.Identity
{
   public class SimpleType
   {
      public int i;

      public int ReturnInteger()
      {
         return i;
      }

   } // class
} // namespace
```

This code contains examples of four key elements of the .NET type system: [Comment 3.14]

- A namespace – `YaoDurant.Identity` [Comment 3.15]
- A class - `SimpleType` [Comment 3.16]
- A field - `i` [Comment 3.17]
- A method - `ReturnInteger` [Comment 3.18]

[3] By *module*, we mean an application (.exe) file or a dynamic link library (.dll) file. Some programming environments use the term to refer to a source file.

The fully-qualified name for the class is `YaoDurant.Identity.SimpleType`, and the fully-qualified name for the method is `YaoDurant.Identity.SimpleType.ReturnInteger`. We discuss the `using` statement in the next section, which lets you reference a namespace to avoid having to use fully-qualified names. [Comment 3.19]

A program source file can have any number of namespaces. The only thing which can reside in a namespace is another namespace, or a class. The purpose of namespaces is to help organize classes, which are the general-purpose container for holding type information. A class is able to contain all other types, including other classes. (A namespace is not a type, and so cannot be contained in a class.) [Comment 3.20]

Accessing Namespaces

A fully-qualified name provides an unambiguous way to identify a class, methods in a class, and static (or shared) fields. That is a good thing. When you look at the code generated for you by the forms designer, you see a lot of fully-qualified names. The forms designer writes the code for you. It later reads the code when you decide to edit a form. In this context, a fully-qualified name helps the forms designer avoid confusion. [Comment 3.21]

The disadvantage of fully-qualified names is that they require a lot of typing. To avoid this extra effort, C# programmers indicate the namespaces they wish to use with a set of `using` statements. For example, if we had a separate source file, module, or assembly which needed to access the types in our namespace, we would eliminate the need for fully-qualified names with a statement like the following:

```
using YaoDurant.Identity;
```

Assembly versus Namespace

Close reading of the CLI reveals that an *assembly*, not a namespace, forms the outermost container of the .NET type system. One reason is that some CLI-compatible languages might not support namespaces. For most purposes, a namespace serves as the outermost container for everything in the .NET type system. There are times, however, when a namespace falls short of providing what we need. [Comment 3.23]

A program sometimes needs to identify itself to the execution engine. There is no namespace class; instead, we use an object created from the `Assembly`[4] class. We use an assembly object to access resources in a module, to load and run programs and shared libraries, and also to query for type information. [Comment 3.24]

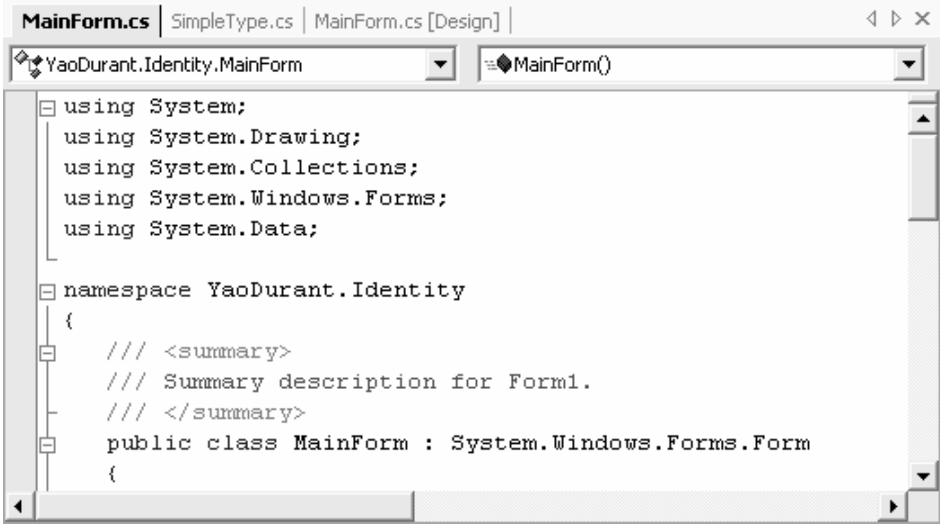
By itself, a `using` statement hints at desired namespaces, but does not indicate what actual module to use. For that, you must add a reference to a Visual Studio .NET project, a subject we discuss in the accompanying text. [Comment 3.25]

On the desktop, an assembly can be created from two or more modules. The Compact Framework does not support multi-module assembly, so every assembly has exactly one module associated with it. [Comment 3.26]

[4] Fully-qualified name: `System.Reflection.Assembly`.

When you create a new Smart Device project in Visual Studio .NET, the Smart Device Application Wizard creates a set of `using` statements for you, as shown in figure 3-1. These are not needed to build the code that has been generated for you, because the forms designer uses fully-qualified names. Instead, these references tell the IDE about the namespaces needed to support IntelliSense.

Figure 3-1 Using declarations created by the Smart Device Application Wizard

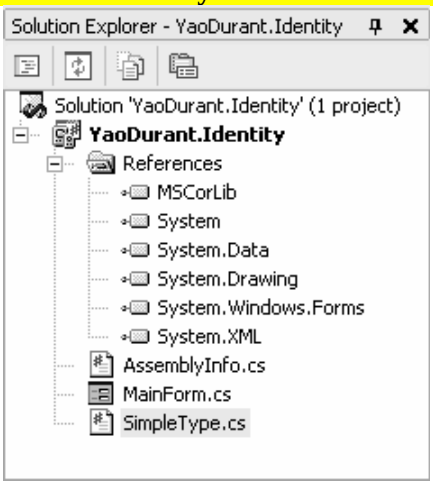


Referencing Assemblies

An assembly reference provides the names of the shared libraries that a .NET program (or shared library) is going to use. Unlike a namespace, which is a logical container for a set of classes, an assembly consists of one or more module files which are the physical container for a set of classes. Any `using` statements provide a hint about what namespaces might be interesting, but those statements are purely optional. The assembly reference, on the other hand, is not optional. [Comment 3.31]

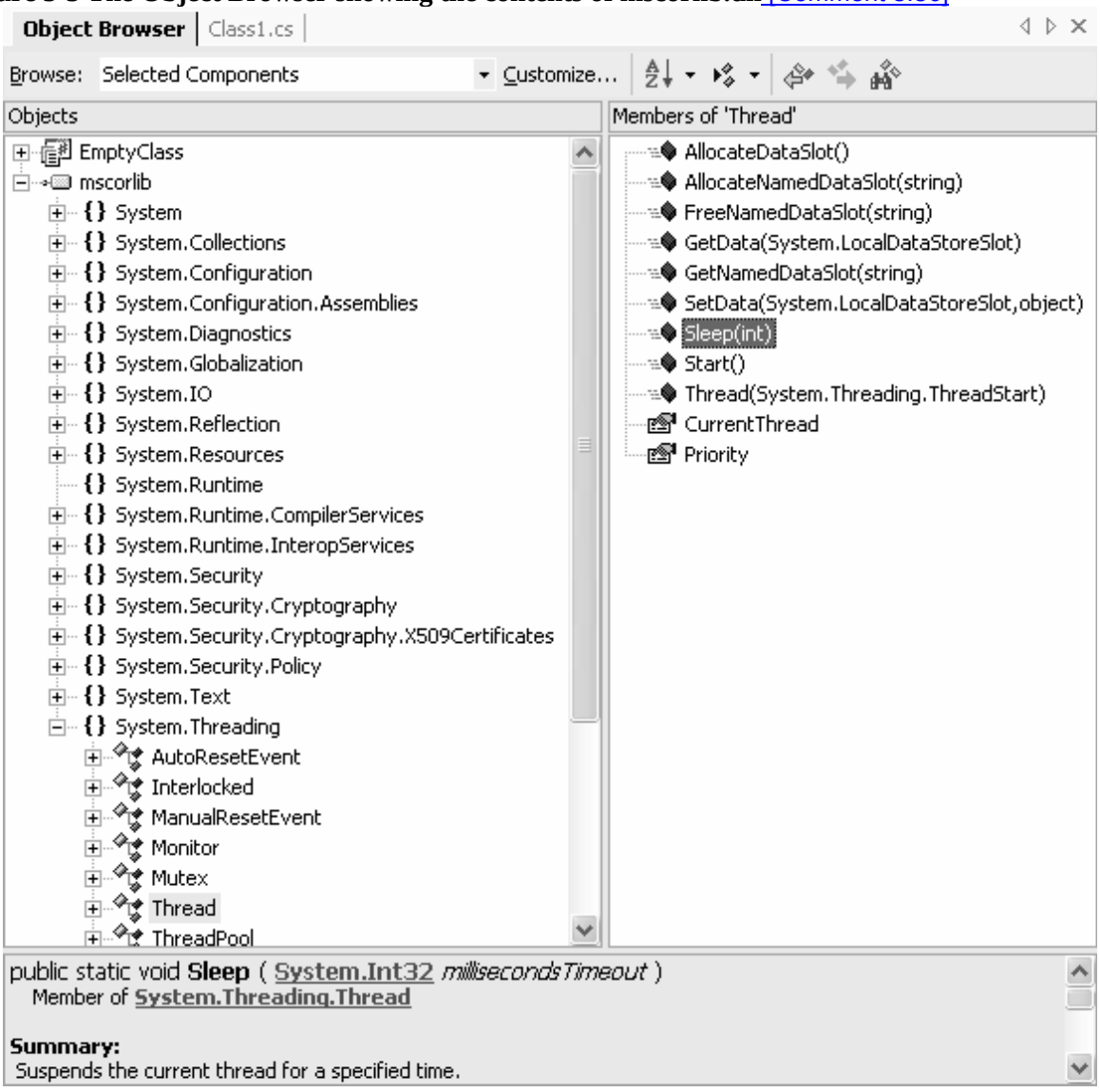
When the Smart Device Application Wizard creates a project for you, it adds a set of assembly references to your project. Figure 3-2 shows an example of the default set that are provided for you in the solution explorer window. You add new assembly references by summoning the ***Add Reference*** dialog box (from the Visual Studio menu, select ***Project -> Add Reference...***; or Right-Click on the Solution Explorer and select ***Add Reference...***). [Comment 3.32]

Figure 3-2 Assembly references created by the Smart Device Application Wizard



The minimum assembly reference required for all .NET programs is `mscorlib.dll`, a short name that stands for "Microsoft Core Library." As the name implies, this library holds the core elements for the .NET runtime system. This sounds like an important library to understand, and so you might wonder how to explore this library. One approach, which we mention in chapter 1, is to use ILDASM – the IL disassembly tool. Another approach is to use the object browser in Visual Studio .NET. In figure 3-3, we show the object browser displaying the contents of `mscorlib.dll`. The left-side shows available namespaces and also classes in the `System.Threading` namespace. The right-side shows details for the public members of the `Thread` class. [Comment 3.35]

Figure 3-3 The Object Browser showing the contents of mscorlib.dll [Comment 3.36]



Standard Types

The portion of Common Language Infrastructure that addresses data types is known as the *Common Type System (CTS)*. The primary motivation for a common type system - and the other elements of the CLI - is to promote interoperability between programming languages. The CTS is sometimes referred to as a unified type system. This term reflects the fact that two groups of types that are often split apart in some object-oriented environments are designed in a way that minimizes the differences. In the CTS, the two groups are called *Value Types* and *Reference Types*. [Comment 3.37]

Value Types

Value types are simple, low-overhead types intended to hold pure data. The CTS defines a core set of values types that all compilers must support to be able to interoperate with other .NET-compatible compilers. Value types are typically supported by language-specific keywords. Integer and floating point numbers, for example, are value types. Also included among value types are data structures. Examples of value types include the following: [Comment 3.38]

- integers
- floating-point numbers
- data structures
- enumerations

Value types are meant to be small and fast, avoiding the inevitable overhead of objects. When used as a local variable, a value type is always allocated on the stack. When nested in an object, the value type is represented as a simple stream of bytes. Value types can also be placed into fixed-length arrays. [Comment 3.39]

Take `Int32`, for example, the type which describes 4-byte signed integers. Such integers can be quickly loaded into 32-bit CPU registers and operated on. In the unified type system of .NET, the 4-byte signed integer is represented by the `System.Int32` class. Like other classes, there are methods for operating on 4-byte integers. We can, for example, call the `ToString()` method to convert an integer to a string, as shown in listing 3-2. [Comment 3.40]

Listing 3-2 Value types have methods just like reference types [Comment 3.41]

```
 int cCount = 10;
string strCount = cCount.ToString();
string strAnother = 10.ToString();
```

A value type can have other elements of objects, including fields. For example, the `Int32` class has two static fields, `MaxValue` and `MinValue`, which define the range of numbers supported by this class. (All numeric value types have similarly named static fields.) [Comment 3.42]

Like all value types, the `System.Int32` class has a base class named `System.ValueType`. And that class, in turn, has a base class named `System.Object`. This is the same base class which all .NET types share in common, yet another aspect of the unified type system. [Comment 3.43]

Value types do have their limitations. While they are defined as .NET classes, they themselves are considered sealed classes. In other words, you cannot use `System.Int32` (or any other value-type class) as the base class for other types. [Comment 3.44]

Integer and Floating Point Value Types

Table 3-1 shows the basic set of built-in numeric value types in C#. Each data type in our table is defined in two ways – as a .NET type, and as a language-specific alias. Which one should you use? It is probably a matter of personal taste, since the language-specific alias gets converted by the compiler to the associated .NET type. We generally prefer language-specific aliases, and that is what you find in the sample code written for this book. For example, we use `int` instead of `System.Int32` or `Int32`, which makes our type definitions more consistent with the rest of our code. The C# compiler converts the language-specific alias into the corresponding .NET type. [Comment 3.45]

Table 3-1 Basic set of built-in value types [Comment 3.46]

.NET Type	*CLI Compat?*	*Size (Bytes)*	*C# Alias*
`System.Boolean`	Yes	1	`bool`
`System.Byte`	Yes	1	`byte`
`System.SByte`	No	1	`sbyte`
`System.Char`	Yes	2	`char`
`System.Int16`	Yes	2	`short`
`System.UInt16`	No	2	`ushort`
`System.Int32`	Yes	4	`int`
`System.UInt32`	No	4	`uint`
`System.IntPtr`	Yes	4	
`System.Single`	Yes	4	`float`
`System.Int64`	Yes	8	`long`
`System.UInt64`	No	8	`ulong`
`System.Double`	Yes	8	`double`
`System.Decimal`	Yes	12	`decimal`

For C++ Programmers:

Experienced Windows programmers who worked in C and C++ are likely to be uncomfortable at first with using native language types like `int` and `char` in Windows code. For many years, Microsoft recommended that Windows programmers use types like `LPSTR` and `LONG` in place of the equivalent C-types `char *` and `long`. The use of the upper-case types (`LPSTR` and `LONG`) played a significant part in making Win16 code extremely portable to the Win32 API when it was introduced in the early 1990s. [Comment 3.47]

The reasons for those recommendations had more to do with limitations on the part of the C language than on anything inherently wrong with using built-in language types. For example, the implementer of a C compiler got to decide whether an `int` was 16, 32, or even 64 bits. (For a system software language like C, that choice arguably should be made on a system-specific basis.) [Comment 3.48]

The language specification for C#, however, does not allow for that kind of flexibility. Instead, `int` is defined as a 32-bit signed integer equivalent to `System.Int32`, and `long` is defined as a 64-bit integer equivalent to `System.Int64`. There is no need for a portability layer with .NET programming languages, because the common type system provides the standard. Changes – like the addition of a 128-bit integer – will come as extensions and not revocations of the existing standard. [Comment 3.49]

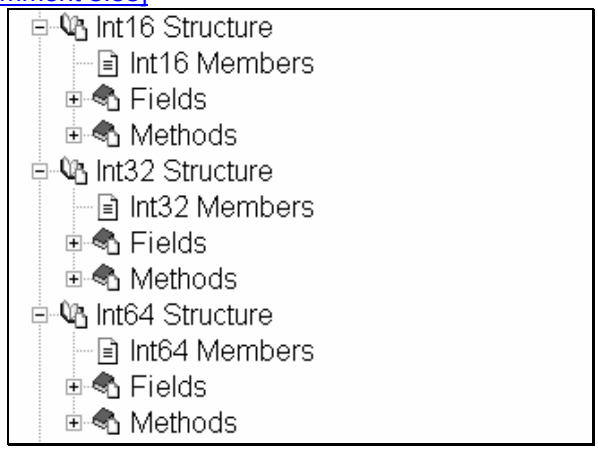
The second column of table 3-1 indicates whether a type is compatible with the CLI or not. If you plan to build assemblies that provide full interoperability with all other .NET languages, then you want to use types with a Yes in this column. For the fourteen types in the table, four are not CLI-compliant: [Comment 3.50]

- `System.SByte` – signed 1-byte integer
- `System.UInt16` – unsigned 2-byte integer
- `System.UInt32` – unsigned 4-byte integer
- `System.UInt64` – unsigned 8-byte integer

By avoiding these types in C# code, we enable our shared assemblies to be called from Visual Basic programs, because Visual Basic does not support these four types. We also gain compatibility with other CLI-compatible languages which may become available in the Compact Framework. [Comment 3.51]

You might notice in the online .NET documentation that built-in value types are referred to as structures. Figure 3-4 shows a portion of the MSDN Library table of contents, in which the `Int32` type is called *Int32 Structure*. You probably do not think of numeric types as data structures. [Comment 3.52]

Figure 3-4 The Table of Contents for MSDN Library refers to value types as Structures [Comment 3.53]



Data Structure Value Types

Within .NET, data structures are value types. The Compact Framework includes support for a large number of data structures which are used for the basic operation of the windowing and graphic systems. Table 3-2 summarizes some of the more commonly used structures in a Compact Framework program. [Comment 3.54]

For VB6 Programmers:

The VB6 keyword, `Type`, for user-defined types, does not exist in VB .NET; VB.NET provides an alternative keyword, the `Structure` keyword, for user-defined types. [Comment 3.55]

For C++ Programmers:

A C++ programmer can think of a C# structure as similar to a C++ structure, with an important difference: while C++ allows a structure to behave like a class and vice-versa, a C# structure is distinctly different from a C# class. The former is a value type, while the latter is a reference type. We discuss that difference later in this chapter. [Comment 3.56]

Table 3-2 Common Compact Framework structures [Comment 3.57]

Namespace	*Class*	*Description*
`System`	`DateTime`	Store date, time, or both date and time.
`System.Drawing`	`Color`	Store color as RGB, a system color, or a named color.
`System.Drawing`	`Point`	An (x,y) location with integer coordinates.
`System.Drawing`	`Rectangle`	An area defined with integer coordinates as a pixel grid, meaning a non-rotated rectangle with a top and bottom, a left and right.
`System.Drawing`	`RectangleF`	An area defined with floating-point coordinates as a pixel grid, meaning a non-rotated rectangle with a top and bottom, a left and right.
`System.Drawing`	`Size`	A width & height pair for the size of a rectangle, with integer coordinates.
`System.Drawing`	`SizeF`	A width & height pair for the size of a rectangle, with floating-point coordinates.
`System`	`Guid`	A globally-unique identifier. Used in COM programming and network programming.
`System`	`IntPtr`	A signed 32-bit wrapper for unmanaged code pointer and handles. (A 64-bit wrapper on 64-bit platforms, but Windows CE – and therefore the Compact Framework – only runs on 32-bit platforms.)
`System`	`TimeSpan`	Stores time duration.

Reference Types

The second major category of CTS types is the set known as *Reference Types*. Objects of reference types are accessed at runtime using a reference, a pointer to the object itself. Examples of reference types include the following: [Comment 3.58]

- Objects created from a class
- Objects encapsulating Win32 system objects (forms, controls, graphic objects, threads, mutexes, files, etc.)
- Arrays
- Strings
- Boxed value types

Just as there are built-in value types, there are also built-in reference types. These types form the foundation for the efficient operation of the type system. Our criteria for built-in reference types are that IL instructions create or require a type, a set which includes: [Comment 3.59]

- `System.Object` – the base type from which all other types are derived.
- `System.String` – immutable string class.
- `System.Array` – Fixed-length array.
- `System.Exception` – Execution errors.
- Pointers – There are three pointer types, but only one is exposed to the type system: `System.IntPtr`. Other pointers are used by the execution engine as references to instantiated objects.

Every object has a standard object header. The size of the object header[5] is 8 bytes, and the smallest possible object (an instance of type `object`) is 12 bytes. This is the smallest possible object, which we mention here simply to underscore that reference types take up a slight bit more space than value types. Object headers contain references to class type information. A program can access this type information using a set of services known as *reflection* and available through classes in the `System.Reflection` namespace. In addition to the header, objects contain instance data: the public and private fields defined for the object's type. [Comment 3.60]

Reference types have complete support for object-oriented programming. The most useful feature is *inheritance*, which lets you define new reference types from existing reference types. When defining a custom type that you expect to use as a base for other new types, the base type must be a class. Custom types that inherit from existing types, in turn, can only inherit from other reference types. [Comment 3.61]

By contrast, inheritance is not supported for value types. Value types are said to be *sealed*, since a value type cannot be used as a base type for other types. (A reference type can also be sealed, using the `sealed` keyword, which prevents that type being used as the base type for new types.) [Comment 3.62]

Objects of reference types are allocated on the runtime heap, and so are subject to garbage collection. Objects without a live reference – either directly in a local variable, or indirectly through another object – are called *unreachable objects*. When the garbage collector reclaims memory, it reclaims the memory occupied by unreachable objects. To make an object unreachable, set the value in the variable to `null`: [Comment 3.63]

```
object obj = new object();
obj = null;
```

Reference Types versus Value Types

When we teach our .NET programming classes, a common discussion topic involves the relative merits of reference types versus value types. Reference types are larger and more complex than value types. In terms of memory and processing time, then, reference types are more costly than value types. Is one better than the other? [Comment 3.64]

This is an interesting question to ask in theory, but in practice both have their natural uses. Value types have less memory overhead, are of a fixed size, and are less extendable. By contrast, reference types are extendable via inheritance, can change size, are inexpensive to pass as parameters, are shareable between threads, have built-in thread synchronization, and enjoy the benefits of automatic garbage collection. [Comment 3.65]

Declaration, Initialization, and Allocation

There are differences between value types and reference types. But a casual look at any .NET code makes those differences hard to detect because a common syntax works for both types of types. Both value type variables and reference type variables can be declared, initialized, and allocated with similar code. There is a real difference in the last step, however, namely in how the two types are allocated. Consider the code in listing 3-3. This code defines two types that are

[5] 8 bytes are for 2 integer fields in 32-bit systems. Object headers in 64-bit systems are 16 bytes.

identical in every way, except that one is a value type and the other is a reference type. [Comment 3.66]

The `MyClick` method declares a variable of each type, `p1` and `p2`. The use of `new` with both types implies that a new object is being created. There is similar code, but a different dynamic comes into play for each variable. Let's start with the value type, `p1`. [Comment 3.67]

Value types do not require the use of `new` when a parameter-less constructor is specified, which is the case in this code. On the other hand, if we had a constructor defined with parameters to pass, we would then use `new`. While optional, a parameter-less constructor for a value type is benign. [Comment 3.68]

The `p1` in our example is allocated on the stack. The compiler notices the use of a value type, and includes space on the stack in the generated IL. At runtime, that space is automatically included as a variable on the stack. [Comment 3.69]

On entry to the function, the contents of `p1` are initialized to zero. This is true for all types, including integers, floating-point, and decimal values. It is also true for reference type variables, like `p2` in our example. All variables start in a known, defined state. [Comment 3.70]

The use of `new` is required for reference types. On entry to the `MyClick` method, the local variable `p2` is initialized to zero. If we omit the `new` and then use that variable to access a member, an exception is generated. Since an unhandled exception causes a program to terminate, we want to avoid using variables for null reference types. [Comment 3.71]

When `new` is encountered for a reference-type variable, the memory allocator gets called to create space in the runtime heap. In addition, one or more constructors are called for the object starting with the constructor for `Object`, and including constructors in all other inherited classes for our reference type. [Comment 3.72]

Listing 3-3 Comparing value types and reference types [Comment 3.73]

```
// Structure -- a value type
struct PointStruct
{
   int X;
   int Y;
}

// Class - a reference type
class PointClass
{
   int X;
   int Y;
}

private void MyClick()
{
   PointStruct p1 = new PointStruct();
   PointClass  p2 = new PointClass();
}
```

Value Types and Reference Types as Parameters

Value types and reference types look the same when you are allocating them and accessing them, so sometimes you can ignore the distinction between them and let the compiler handle

things. Aside from the way the need to use the `new` operator with reference types, the code that you write for dealing with both types is quite similar. [Comment 3.74]

The difference between value types and reference types becomes important when you wish to pass a parameter to a method. Both value and reference types can be passed as parameters, but the declarations are different for each. Value types can be either by-value parameters or by-reference parameters. Reference types can also be by-value parameters or by-reference parameter. The majority of the time, however, both value types and reference types are passed as by-value parameters. [Comment 3.75]

Value Types as Parameters

Value types are allocated as bits local to the owner – on the stack for local variables, and within the object instance data when a class field. Value types can be passed either by-value or by-reference. The value types which are 32-bits and smaller are normally passed by-value. Larger value types and structures containing value types are often passed by-reference. When a value type is passed by-value, a copy of the value is put on the stack. That value makes a one-way trip into the called function, and the local copy of that value is not affected. [Comment 3.76]

When a value type is passed by-reference, an implicit pointer is put on the stack. The called function uses that pointer to read the contents of our variable. The called function can also change the contents of a variable, which is why we sometimes say that a by-reference parameter makes a two-way trip – down to the called function, then back again with whatever changes the function has made. Listing 3-4 shows value types passed as by-value parameters and as by- reference parameters. Passing value parameters by-reference is particularly useful when calling certain native Win32 functions, a topic we explore more fully in chapter 4 (*Platform Invoke*). [Comment 3.77]

Listing 3-4 Value types as parameters [Comment 3.78]

```
   private void ValueParameters()
   {
      // Init to one
      int i1 = 1;
      int i2 = 1;

      // Function adds one to each
      Increment(i1, ref i2);

      // On return:
      //   i1 = 1
      //   i2 = 2
   }

   // i is a value type parameter passed by value
   // pi is a value type parameter passed by reference
   private void Increment(int i, ref int pi)
   {
      i++;
      pi++;
   }
```

Reference Types as Parameters

Reference types are allocated from the process heap. A variable of a reference type contains a pointer that refers to the object. The presence of a pointer in a reference type creates a funny sort

of paradox: you pass a pointer to a reference type by declaring a by-value parameter. Passing a reference type by-reference, then, creates a pointer to a pointer. This is rarely done, but the capability exists for those few situations – such as with strings and in some calls to native code – when it may be necessary. [Comment 3.79]

Strings

Strings get special treatment in the world of .NET programming. Special treatment makes sense because strings are costly in terms of memory consumed, and also because strings are required to communicate with people. The framework provides two main classes to support string operations: `System.String` and `System.Text.StringBuilder`. [Comment 3.80]

To make optimal use of available memory, programmer working with string-intensive applications need to understand how these two classes work. Difference centers on the issue of immutability, a subject we explore later in this chapter. [Comment 3.81]

For programmers writing software to be made available in support of different languages, strings add an additional level of complexity. A program written for Spanish-speaking users and English-speaking users must have two sets of strings, one for each language. The use of string resources allow a programmer to separate strings from code, to simplify the task of preparing software for a multi-lingual end-user population. [Comment 3.82]

Literal Strings

In the "good old days"[6] of Windows programming, we worried about literal strings. To reduce overhead, a programmer put redundant strings into global variables (or in a string table). As the Windows on the desktop got more RAM, fewer programmers worried about redundant strings. [Comment 3.83]

With today's compilers, no one needs to worry about redundant literal strings. The reason is that the C# compiler helps out by folding duplicate strings to reduce storage overhead. Listing 3-5 shows the `SayHello` method, in which `"Hello"` appears four times. The compiler reduces all four references to refer to a single string. In this way, an executable file is smaller than it might otherwise be. [Comment 3.84]

Listing 3-5 The SayHello Method [Comment 3.85]

```
private void SayHello()
{
   string str1 = "Hello";
   StringBuilder sb1 = new StringBuilder("Hello");
   MessageBox.Show("Hello", "Hello");
}
```

At runtime, static strings are allocated from the heap like other types of objects. Allocation of such strings is handled in a just-in-time fashion, in the same way that the IL-to-native translation is handled. The runtime JIT compiler does not convert an entire module, but rather converts

[6] Some of you may remember the days when, knee-deep in snow, we walked uphill (both ways) to school. We allocated memory with a slide rule and drank from punch cards; we used scissors to cut and chewing gum to paste.

individual methods as they are called. Part of the JIT process involves allocating static strings that a specific method is going to need. [Comment 3.86]

The JIT compiler works in a way that allows the garbage collector to remove unreachable strings, and yet eliminates redundant strings. If a static string has already been allocated on the heap during the JITting of another method, the JIT compiler adds a new reference to the string but does not create a second copy of the string. [Comment 3.87]

Literal strings are stored as UTF-16 (2-byte/character) Unicode characters in an executable module. (Unicode is also the character set for strings in memory.) If you have many literal strings, you might consider storing those strings as resources. Strings stored as resources are stored in the smaller, more memory-efficient 1-byte (UTF-8) Unicode strings. We discuss the creation of string resources later in this chapter. [Comment 3.88]

The `System.String` Class

A string is an array of characters. Most of the time, however, we do not want to deal with the fussiness of arrays. Instead, we want an easy way to display a message to a user, or write an entry to a log file that we can later read. We get all these things in the `System.String` class, which provides a lightweight class useful for most string operations. [Comment 3.89]

String Properties, Methods, and Operators

The `Length` property provides instant access to the number of characters in the string. An indexer (`[n]` operator in C#) lets us treat the string as an array of characters when we need the benefits an array provides. A host of other member methods lets us search a string, add or remove padding, or extract a substring. And best of all, the concatenation operator – the plus ('+') – makes it easy to create a new string by joining two existing strings. Each of these is illustrated in listing 3-6. [Comment 3.90]

These are all useful, but we think the most interesting string method is a member of the `Object` class. For that reason, it is a string method that every .NET type supports. We refer here to the `ToString()` method, which converts any object to a string, and provides an easy way to get a string for any type of object. Listing 3-6 shows examples of calling the `ToString()` method for an integer and a character. [Comment 3.91]

Listing 3-6 Common string operations [Comment 3.92]

```
string str = "A literal string";
int cb = str.Length;
string strLength = "Length is " + cb.ToString();
string strFirst  = "First character is " + str[0].ToString();
MessageBox.Show("String is " + str + "\n" +
   strLength + "\n" +
   strFirst);
```

For C Programmers

String operations in C are difficult and painful. C programmers celebrate when they start .NET programming because of the pain they have endured over the years. In particular, the '+' operator concatenates two

strings. The `Length` property replaces the need to call `strlen`[7] to calculate a string length. Life, in short, gets easier. [Comment 3.93]

MFC programmers may yawn at hearing this, but that is because MFC programmers have the `CString` class. VB programmers, for their part, are equally unimpressed. Both come from a world where strings flow freely, are easy to allocate, concatenate, and work with. [Comment 3.94]

Value Type or Reference Type?

It is sometimes said that the `System.String` class is a reference type that gets treated like a value type. All other reference types require `new` for a new instance, but strings created from literal strings do not, as shown in the way `str` gets allocated in listing 3-6. (You use `new` to create strings from arrays.) [Comment 3.95]

Behavior like a value type can also be observed seen in situations where two variables seem to refer to the same string, as in the following code: [Comment 3.96]

```
string str1 = "A";
string str2 = str1;
str2 = "B";
```

If strings were like other reference types, `str1` and `str2` would point to the same string, having a value of `"B"`, when the above code is executed. But strings are not like other reference types. After the above assignments, `str1` references `"A"` and `str2` references a `"B"`. The reason is that strings are immutable, a subject we look at next. [Comment 3.97]

Strings Are Immutable

Another way that strings are treated as value types is often expressed by the phrase "strings are immutable." The word "immutable" means unchangeable. In the system heap, a string is a read-only object. Once created, a string does not change. [Comment 3.98]

Does this mean that you cannot change strings? Of course not! As a programmer it is your job to change things. The string class contains methods like `ToUpper()` and `ToLower()`, which changes the case of a string. But each function leaves the original string unchanged, and what you get back is transformed copy. You can change strings, but any operation on a string leaves the original string untouched. What happens to old strings?[8] They remain on the heap to be used by other references, or to be garbage collected when they are unreachable. [Comment 3.99]

Is there a good reason for making strings work this way? Yes, there are many good reasons, all centered around making strings lightweight and easy to work with. As immutable objects, strings can be passed as parameters and the caller knows that the original string cannot change. [Comment 3.100]

Immutable (read-only) strings make multi-threaded programming easier, because two threads can use a common set of strings, and no extra care need be taken to coordinate the sharing of strings. By contrast, when a reference to a read/write objects get passed as parameters or shared between two threads, there can be trouble. Sharing a read/write object requires strict rules to be followed; sharing read-only objects requires much less care. [Comment 3.101]

[7] Or `wcslen()` or `_tcslen()`.
[8] For fans of American-style football: Old strings, like old quarterbacks, fade back and pass away.

The Other Side of Immutability

Strings behave more like integers than like other objects. There is one difference, though, which is that each operation on a `System.String` object creates a new item in the heap. By contrast, nothing gets created on the heap for integer operations. [Comment 3.102]

The immutability of strings makes them easy to use, but the memory cost makes them a bad fit for extensive string processing. What do we mean by extensive? We mean processing involving any combination of many strings, large strings, or many string operations. For that kind of processing, the second string class is the one to use – the string builder class. [Comment 3.103]

The `System.Text.StringBuilder` Class

The `StringBuilder` class provides a memory and processor efficient way to create mutable strings for intensive string processing. For example, our P/Invoke Wizard tool uses a string builder to read C/C++ function declarations and create equivalent managed code declarations. [Comment 3.104]

The string builder class is not meant as a direct substitute for the regular string class. For one thing, the two classes do not have any kind of inheritance relationship. Instead, you pass regular strings into the various methods of the class, and use the string builder's `ToString()` method to retrieve the final results. [Comment 3.105]

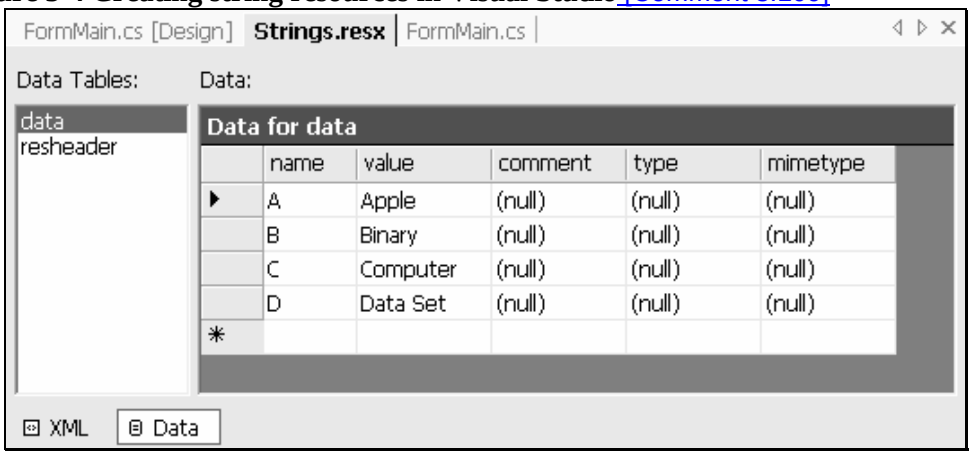
By itself, this class provides a good way to save memory. For optimal savings, however, you should set the size of the buffer – defined with the `Capacity` property – with enough characters for whatever strings you are planning to work with. The default capacity is 16 characters, which is probably too small for all but the simplest operations. The capacity property also defines the allocation granularity, so if you err you probably want to err on the high side. [Comment 3.106]

String Resources

String resources provide an alternative to putting literal strings in your code. Literal strings can make it more difficult to build software for a multi-lingual end-user community. For non-Asian languages, literal strings occupy twice as much space as the same strings stored as resources. String resources are worth a consideration, although their use does take a bit more effort than literal strings. [Comment 3.107]

To create string resources, add an XML resource file to your project with the Visual Studio menu: ***Project | Add New Item...*** and pick ***Assembly Resource File*** from the available types. The extension for XML resources files is `.resx`. Figure 3-4 shows a set of strings we create (the second column) and the associated names (the first column), for which we are using single letters for simplicity. [Comment 3.108]

Figure 3-4 Creating string resources in Visual Studio [Comment 3.109]



Like everything in .NET, XML resources are strongly typed. This is important to keep in mind because we access resources using a fully-qualified name. As with code, a fully-qualified name for a resource involves combining a namespace with a type name. The type name for the resource in figure 3-4 is the file name without the extension ("`Strings`"). The namespace is the default namespace for the project, which we can find in the project properties window. For our sample project, the default namespace is "`StringResources`". [Comment 3.110]

To access the strings in this string table, we create a resource manager object from the `ResourceManager`[9] class. The constructor for that object requires an identifier for the assembly object for our assembly, an object of type `Assembly`[10]. To access individual strings, we call the `GetString` method on the resource manager, specifying the string identifier. In our example, the string identifiers are the four letters for the four strings: "`A`", "`B`", "`C`", and "`D`". Listing 3-7 summarizes each of the steps needed to load the strings and display then when the form receives a `Paint` event. [Comment 3.111]

Listing 3-7 Loading and using string resources. [Comment 3.112]

```
//...
using System.Resources;   // Needed for ResourceManager
using System.Reflection;  // Needed for Assembly
//...
namespace StringResources
{
   //...

      // Resource manager for our set of strings.
      ResourceManager resman;

      // The four strings we load
      string strA;
      string strB;
      string strC;
      string strD;

      private void
      FormMain_Load(object sender, System.EventArgs e)
```

[9] Fully-qualified name: `System.Resources.ResourceManager`.
[10] Fully-qualified name: `System.Reflection.Assembly`.

```
        {
            // Create the resource manager.
            Assembly assembly = this.GetType().Assembly;
            resman = new ResourceManager(
               "StringResources.Strings", assembly);

            // Load the strings.
            strA = resman.GetString("A");
            strB = resman.GetString("B");
            strC = resman.GetString("C");
            strD = resman.GetString("D");
        }

        private void
        FormMain_Paint(object sender, PaintEventArgs e)
        {
            float sinX = 10;
            float sinY = 10;
            SizeF szf;

            Brush brText = new SolidBrush(SystemColors.WindowText);
            szf = e.Graphics.MeasureString(strA, Font);

            e.Graphics.DrawString(strA, Font, brText, sinX, sinY);
            sinY = sinY + szf.Height;
            e.Graphics.DrawString(strB, Font, brText, sinX, sinY);
            sinY = sinY + szf.Height;
            e.Graphics.DrawString(strC, Font, brText, sinX, sinY);
            sinY = sinY + szf.Height;
            e.Graphics.DrawString(strD, Font, brText, sinX, sinY);
            sinY = sinY + szf.Height;
        }

//...
```

Type Conversion

In a strongly-typed environment, type conversions play an important role in adding flexibility to the strict rules of the type system. Strong type checking is meant to help the programmer, but can be a hindrance. For example, integers can be used in arithmetic operations, but must be converted to strings for display to users. Within .NET, a rich set of conversion features helps reduce the friction that sometimes occurs in a strongly-typed system. [Comment 3.113]

In some cases, conversion support is built into the language syntax. C# supports casting, and VB .NET has `CType` and `DirectCast`, among its conversion functions. There are subtle differences between the conversions provided by each language. In converting a floating-point number to an integer, for example, C# truncates the remainder while VB .NET rounds the remainder. [Comment 3.114]

A third set of conversion services is provided by various parts of the framework. In some cases, the framework duplicates existing language services. The `Convert`[11] class, for example,

[11] Fully-qualified name: `System.Convert`.

Thank You

Thank you for taking the time to read this preview chapter. We hope it has provided you insights and tips to help with your Compact Framework programming project. You can help us create a better book by clicking the comment link at the end of each paragraph, and sending your comments and suggestions on our review web site.

Preview Chapter Text

Our public review site provides the complete table of contents for the Compact Framework book at this link: http://www.paulyao.com/cfbook.htm. That table of contents contains links to all the preview chapters. The preview chapter provides the complete outline of topics covered in a chapter, and also the first section or two from each chapter.

Complete Chapter Text

You can get the complete text for each chapter, available to readers who register at our web site. Registration is simply and easy – we only ask for an email address. To register, click on this link: http://www.paulyao.com/ReaderFeedback/Logon.aspx.

When you register, you can download the available chapters from the full-text Table of Contents, available at this link: http://www.paulyao.com/ReaderFeedback/default.aspx. We notify registered readers of new chapters – and chapter updates – as they become available.