2

The Type System

CHAPTER 1 PROVIDED a high-level overview of the issues involved in building distributed systems. It introduced a solution to these issues, the .NET Framework, and used a simple "Hello World" example to highlight the language interoperability offered by the .NET Framework. But, as is so often the case, the devil lies in the details. Chapters 2 through 4 describe in more depth the three CLR subsystems: the type system (described in this chapter) and the metadata and execution systems (described in Chapters 3 and 4, respectively).

As noted in Chapter 1, the facilities provided by the type, metadata, and execution systems are not new. However, the CLR does provide functionality in addition to the services provided by other architectures, such as COM/DCOM, CORBA, and Java. For example:

- The type system supports many programming styles and languages, allowing types defined in one language to be first-class citizens in other languages.
- The metadata system supports an extensibility mechanism, called custom attributes, that allows developers to extend the metadata annotations.
- The execution system ensures security and supports versioning on types in the CLR.

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 20

Using the .NET Framework, developers can both define and share types. Defining and sharing new types in a single language is not particularly challenging; allowing a newly defined type to be used in other languages is much more problematic. This chapter offers a sufficiently detailed understanding of the CLR type system so that developers can appreciate how it achieves type interoperability.

The Relationship Between Programming Languages and Type Systems

The Evolution of Type Systems

Why is a type system necessary at all? Some early programming languages did not provide a type system; they simply saw memory as a sequence of bytes. This perspective required developers to manually craft their own "types" to represent user-defined abstractions. For example, if a developer needed four bytes to represent integer values, then he or she had to write code to allocate four bytes for these integers and then manually check for overflow when adding two integers, byte by byte.

Later programming languages provided type systems, which included a number of built-in abstractions for common programming types. The first type systems were very low level, providing abstractions for fundamental types, such as characters, integers, and floating-point numbers, but little more. These types were commonly supported by specific machine instructions that could manipulate them. As type systems become more expressive and powerful, programming languages emerged that allowed users to define their own types.

Of course, type systems provide more benefits than just abstraction. Types are a specification, which the compiler uses to validate programs through a mechanism such as static type checking. (In recent years, dynamic type checking has become more popular.) Types also serve as documentation, allowing developers to more easily decipher code and understand its intended semantics. Unfortunately, the type systems provided by many programming languages are incompatible, so language integration requires the integration of different types to succeed.

Programming Language-Specific Type Systems

Before attempting to design a type system for use by multiple languages, let's briefly review the type systems used by some of the more popular programming languages.

The C programming language provides a number of primitive built-in types, such as `int` and `float`. These types are said to closely resemble a machine's architecture, as they can often be held in a single register and may have specific machine instructions to process them. The C programmer can also create user-defined types, such as enumerations or structures. Structures are essentially aggregate types that contain members of one or more other types.

The C++ programming language takes the type system of C and extends it with object-oriented and generic programming facilities. C++'s classes (essentially C structures) can inherit from multiple other classes and extend these classes' functionality. C++ does not provide any new built-in types but does offer libraries, such as the Standard Template Library (STL), that greatly enhance the language's functionality.

SmallTalk is an object-oriented language in which all types are classes. SmallTalk's type system provides single-implementation inheritance, and every type usually directly or indirectly inherits from a common base class called `Object`,[1] providing a common root class in the SmallTalk type system. SmallTalk is an example of a dynamically type-checked language.

Like SmallTalk, Java provides an object-oriented type system; unlike SmallTalk, it also supports a limited number of primitive built-in types. Java provides a single-implementation inheritance model with multiple inheritance of interfaces.

The Design Challenge: Development of a Single Type System for Multiple Languages

Given the variety of type systems associated with these programming languages, it should be readily apparent that developing a single type system for multiple languages poses a difficult design challenge. (Most of the lan-

---

[1] In Smalltalk, a class can inherit from `nil` rather than `Object`.

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 22

guages mentioned previously are object-oriented.) Also, it is clear from the list of requirements that not all type systems are compatible. For example, the single-implementation inheritance model of SmallTalk and Java differs from the multiple-implementation inheritance capabilities of C++.

The approach taken when designing the CLR generally accommodated most of the common types and operations supported in modern object-oriented programming languages. In general terms, the CLR's type system can be regarded as the union of the type systems of many object-oriented languages. For example, many languages support primitive built-in types; the CLR's type system follows suit. The CLR's type system also supports more advanced features such as properties and events, two concepts that are found in more modern programming languages. An example of a feature not currently supported in the CLR is multiple-implementation inheritance—an omission that naturally affects languages that do support multiple inheritance, such as C++, Eiffel, and Python.

Although the CLR's type system most closely matches the typical object-oriented type system, nothing in the CLR precludes non–object-oriented languages from using or extending the type system. Note, however, that the mapping from the CLR type system provided by a non–object-oriented language may involve *contortions*. Interested readers should see the appendices at the end of this book for more details on language mapping in non–object-oriented languages.

CLR–Programming Language Interaction: An Overview

Figure 2.1 depicts the relationship between elements of the CLR and programming languages. At the top of the diagram, the source file may hold a definition of a new type written in any of the .NET languages, such as Python. When the Python.NET compiler compiles this file, the resulting executable code is saved in a file with a .DLL or .EXE extension, along with the new type's metadata. The metadata format used is independent of the programming language in which the type was defined.

Once the executable file for this new type exists, other source files—perhaps written in languages such as C#, Managed C++, Eiffel, or Visual Basic (VB)—can then import the file. The type that was originally defined in Python can then be used, for example, within a VB source code file just as if it were a VB type. The process of importing types may be repeated

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 23

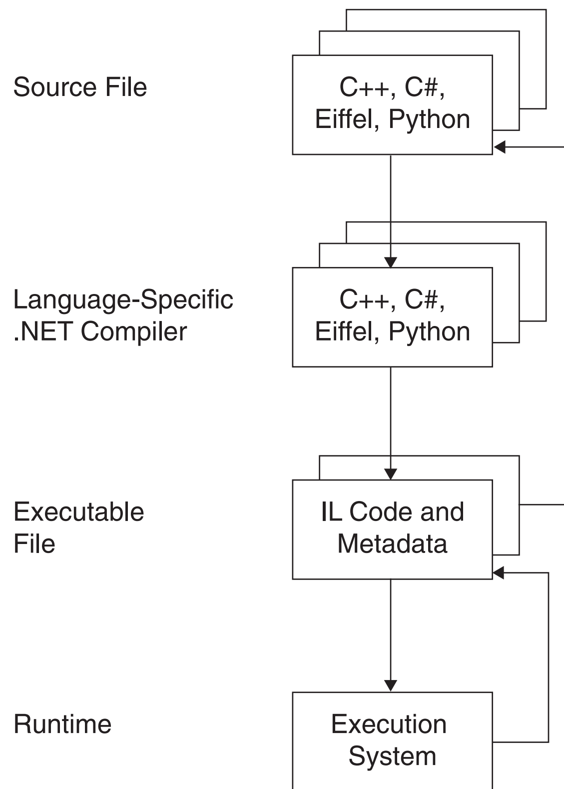

Figure 2.1 *Interaction between languages, compilers, and the CLR*

numerous times between different languages, as represented by the arrow from the executable file returning to another source file in Figure 2.1.

At runtime, the execution system will load and start executing an executable file. References to a type defined in a different executable file will cause that file to be loaded, its metadata will be read, and then values of the new type can be exposed to the runtime environment. This scenario is represented by the line running from the execution system back to the executable files in Figure 2.1.

Elements of the CLR Type System

Figure 2.2 depicts the basic CLR type system. The type system is logically divided into two subsystems, value types and reference types.

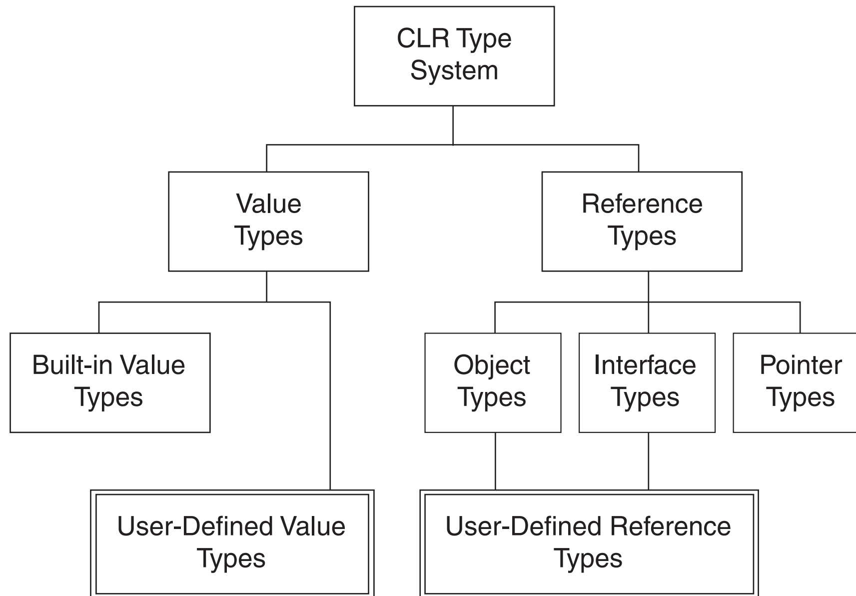

Figure 2.2 *The CLR type system*

A *value type* consists of a sequence of bits in memory, such as a 32-bit integer. Any two 32-bit integers are considered equal if they hold the same number—that is, if the sequence of bits is identical.

Reference types combine the address of a value (known as its identity) and the value's sequence of bits. Reference types can, therefore, be compared using both identity and equality. *Identity* means that two references refer to the same object; *equality* means that two references refer to two different objects that have the same data—that is, the same sequence of bits.

On a more practical level, references types differ from value types in the following ways:

- Value types always directly inherit from `System.ValueType` or `System.Enum`, which itself inherits from `System.ValueType`. The types are always sealed, which means that no other types can inherit from them. Reference types, in contrast, inherit from any class other than `System.ValueType` or `System.Enum`.

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 25

- Reference types are always allocated on the garbage collected heap, whereas value types are normally allocated on the stack. Note, however, that value types may be allocated on the garbage collected heap—for instance, as members of reference types.
- Reference types are accessed via strongly typed references. These references are updated if the garbage collector moves an object.

As mentioned previously, in CLR terminology, an instance of any type (value or reference) is known as a *value*. Every value in the CLR has one exact type, which in turn defines all methods that can be called on that value.

In Figure 2.2, note that the User-Defined Reference Types box does not connect with the Pointer Types box. This fact is sometimes misconstrued as meaning that pointers cannot point to user-defined types. This is not the case, however; rather, the lack of a connection means that developers cannot define pointer types but the CLR will generate pointers to user-defined types as needed. This situation is similar to that observed with arrays of user-defined types: Developers cannot define these arrays but the CLR generates their definitions whenever they are needed.

Value Types

Value types represent types that are known as *simple* or *primitive* types in many languages. They include types such as `int` and `float` in C++ and Java. Value types are often allocated on the stack, which means that they can be local variables, parameters, or return values from functions. By default, they are passed by value. Unlike in some programming languages, CLR value types are not limited to built-in data types; developers may define their own value types if necessary.

Built-in Value Types

Table 2.1 lists the CLR's built-in value types. In the table, the "CIL Name" column gives the type's name as used in Common Intermediate Language (CIL), which could best be described as the assembly language for the CLR. CIL is described in more detail in Chapter 4, which covers the execution

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 26

system. The next column, "Base Framework Name," gives the name for the type in the Base Framework. The Base Framework is often referred to as the Framework Class Library (FCL). As the library contains more than just classes, this name is somewhat inaccurate. Chapter 7 covers the Base Framework in more detail.

TABLE 2.1 CLR Built-in Value Types

CIL Name	Base Framework Name	Description	CLS Support
bool	System.Boolean	Boolean, true or false	Y
char	System.Char	Unicode character	Y
int8	System.SByte	Signed 8-bit integer	N
int16	System.Int16	Signed 16-bit integer	Y
int32	System.Int32	Signed 32-bit integer	Y
int64	System.Int64	Signed 64-bit integer	Y
unsigned int8	System.Byte	Unsigned 8-bit integer	Y
unsigned int16	System.UInt16	Unsigned 16-bit integer	N
unsigned int32	System.UInt32	Unsigned 32-bit integer	N
unsigned int64	System.UInt64	Unsigned 64-bit integer	N
float32	System.Single	IEEE 32-bit floating-point number	Y
float64	System.Double	IEEE 64-bit floating-point number	Y
native int	System.IntPtr	Signed native integer, equivalent to the machine word size (32 bits on a 32-bit machine, 64 bits on a 64-bit machine)	Y
native unsigned int	System.UIntPtr	Unsigned native integer	N

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 27

Note that 8-bit integers appear to be named in an inconsistent manner when compared to the other integral types. Normally, the unsigned integers are known as `System.UIntX`, where `X` is the size of the integer. With 8-bit integers, however, the signed version is known as `System.SByte`, where `S` means signed. This nomenclature is preferred because unsigned bytes are used more frequently than signed bytes are, so the unsigned byte gets the simpler name.

Boolean Values

The `bool` type is used to represent true and false values. Unlike some languages that use an integer for this type (so that a value such as 0 represents false and all other values represent true), the CLR designates a specific type for this purpose. This choice eliminates errors that could potentially arise when integer values are taken to signify Boolean values but that interpretation was not the programmer's intention.

Characters

All characters in the CLR are 16-bit Unicode code points.[2] The UTF-16 character set uses 16 bits to represent characters; by comparison, the ASCII character set normally uses 8 bits for this purpose. This point is important for a component model such as the .NET Framework, for which distributed programming over the Internet was a prime design goal of the architecture. Many newer languages and systems for Internet programming, such as Java, have also decided to support Unicode.

Integers

The CLR supports a range of built-in integer representations. Integers vary in three ways:

- Their size can be 8, 16, 32, or 64 bits. This range covers the size of integers in many common languages and machine architectures.

---

2 Throughout this book, the generic term *character* is used rather than terms from the Unicode standard, such as *abstract character* and *code point.*

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 28

- Integers can be signed or unsigned, designating whether the values they hold are positive only or positive/negative.
- Native integers are used to represent the most natural size integer on the execution architecture. Because the CLR is designed to run on a number of different platforms, it needs a mechanism to inform the execution engine that it is free to choose the most efficient representation on the platform that the code executes on—hence the native integer type.

The last point highlights a recurring theme in the design of the CLR—namely, that many issues are left to the execution system to resolve at runtime. While this flexibility does incur some overhead, the execution system can make decisions about runtime values to ensure more efficient execution. Another example of this facility, which is covered in more detail later, involves the layout of objects in memory. Developers may explicitly specify how objects are laid out or they can defer this decision to the execution engine. The execution engine can take aspects of the machine's architecture, such as word size, into account to ensure that the layout of fields aligns with the machine's word boundaries.

Floating-Point Types

CLR floating-point types vary between 32- and 64-bit representations and adhere to the IEEE floating-point standard.[3] Rather than providing a detailed overview of this standard here, readers are referred to the IEEE documentation. Native floating-point representations are used when values are manipulated in a machine, as they may be the natural size for floating-point arithmetic as supported by the hardware of the underlying platform. These values, however, will be converted to `float32` or `float64` when they are stored as values in the CLR. Providing internal representations for floating-point numbers that match the natural size for floating-point values on the machine on which the code executes allows the runtime environment to operate on a number of different platforms where intermediate results may be larger than these types. A native floating-point

3 IEC 60559:1989, *Binary Floating-Point Arithmetic for Microprocessor Systems.*

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 29

value will be truncated, if necessary, when the value is stored into a 32- or 64-bit location in the CLR.

Special Issues with Built-in Value Types

In Table 2.1, notice which of the built-in value types are CLS compliant. As stated previously, languages must adhere to the CLS subset of the CLR to achieve maximum interoperability between languages. It is not surprising that types such as `int32` are listed in the CLS whereas types such as `native unsigned int` are not. Note, however, that most unsigned integers are not included in the CLS.

Also, note that not all programming languages will expose all of these value types to developers. Types such as `native int`, for instance, may not have a natural mapping into a language's type system. Also, language designers may choose not to expose a type—instead exposing only CLS-compliant types, for example.

A number of value types are defined in the Framework Class Library. Technically speaking, they are really user-defined types; that is, they have been defined by the developers of the Base Framework rather than being integral CLR value types. Developers using the CLR often do not recognize this distinction, however, so these types are mentioned here. Of course, such a blurry distinction is precisely what the designers of the CLR type system were hoping to achieve. Examples of such types include `System.DateTime`, which represents time; `System.Decimal`, which represents decimal values in the approximate range from positive to negative 79,228,162,514,264,337,593,543,950,335; `System.TimeSpan`, which represents time spans; and `System.Guid`, which represents globally unique identifiers (GUIDs).

User-Defined Value Types

In addition to providing the built-in value types described previously, the CLR allows developers to define their own value types. Like the built-in value types, these types will have copy semantics and will normally be allocated on the stack. A default constructor is not defined for a value type. User-defined value types may be enumerations or structures.

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 30

Enumerations

Listing 2.1 gives an example of the declaration and use of a user-defined enumeration.[4] This program defines a simple enumeration representing the months in the year and then prints a string to the console window matching a month name with its number.

Listing 2.1 User-defined value type: `EnumerationSample`

```
using System;

namespace Enumeration
{
  struct EnumerationSample
  {
    enum Month {January = 1, February, March,
                April, May, June,
                July, August, September,
                October, November, December}
    static int Main(string[] args)
    {
      Console.WriteLine("{0} is month {1}",
            Month.September,
           (int) Month.September);
      return 0;
    }
  }
}
```

For the first few examples in the book, the C# code used is described so that you will become familiar with how to read C#. Later, such detailed descriptions of the example code are omitted.

The first line in Listing 2.1 is the `using System` directive, which is included so that types whose names start with "`System.`", such as `System.Console`, can be referenced without fully qualifying those names. While this tactic reduces the amount of typing developers need to do, its overuse can eliminate the advantages gained by using namespaces, so employ this technique judiciously.

4 Although C# is used for most programming examples in this book, this choice was made only because the C-derived syntax is assumed to be familiar to many programmers.

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 31

Next in Listing 2.1 comes the definition of a namespace called `Enumeration`. This name has no programmatic significance; we could have used any name for the namespace or even not used a namespace at all. Nevertheless, because components developed within the CLR are designed to be reused in many scenarios, including being downloaded from the Internet, the use of namespaces is strongly encouraged to avoid collisions between the names of components developed by different developers.

Listing 2.1 continues with the definition of the user-defined value type `EnumerationSample`. This value type will hold the program's entry point, the method with which execution will commence. The C# keyword `enum` is used to define an enumeration. In Listing 2.1, this enumeration is called `Month` and contains constants representing each month of the year. The declaration of the enumeration is reasonably straightforward, except for the fact that the enumeration starts the constants with a value of `1`; the default value would be `0`.

`Main` is the entry point for the program. It prints out a single line of output that informs the user that September is the ninth month (`9`) of the year.

Listing 2.1 produces the following output:

```
September is 9
```

Structures

In this book, user-defined value types are called *structures.* Some languages, such as Managed C++, allow users to use a keyword such as `class` when defining either value or reference types. Other languages, such as C#, use a keyword such as `struct` to indicate a user-defined value type and `class` to indicate a user-defined reference type. This choice is largely a language-specific issue, but readers should be aware of these differences and understand the behavior of the language they are using.

Structures can contain any of the following:

- Methods (both static and instance)
- Fields (both static and instance)
- Properties (both static and instance)
- Events (both static and instance)

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 32

Methods Methods specify a contract that must be honored by both the caller and the callee. Methods have a name, a parameter list (which may be empty), and a return type. Clients that need to call a method must satisfy the contract when calling the method.

Methods on value types can be either static methods or instance methods:

- Static methods are invoked on the type itself and are callable at any time, even if no values of the type exist.
- Instance methods are always invoked on values of a type.

One limitation on value types is that they cannot define a constructor that takes no parameters, known as a *default constructor* in some languages.

Listing 2.2 demonstrates the definition and use of both static and instance methods on a value type in C#. Both methods write a greeting to the console window. The code starts with a `using` directive; it allows types whose names would start with `System`, such as "`System.Console`," to be referenced more simply: `Console`. Next comes the declaration of the user-defined namespace, `ValueTypeMethods`. Note that the CLR does not intrinsically support namespaces; instead, a type `T` declared in namespace `N` is known to the CLR as the type `N.T`. This type, `N.T`, will reside in an *assembly;* the CLR uses such assemblies to uniquely identify types—not namespaces, as in some languages. (Assemblies are covered in Chapter 5.)

Listing 2.2 *Use of static and instance methods with a user-defined value type*

```
using System;

namespace ValueTypeMethods
{
  struct Sample
  {
    public static void SayHelloType()
    {
      Console.WriteLine("Hello world from Type");
    }
    public void SayHelloInstance()
    {
      Console.WriteLine("Hello world from instance");
    }
```

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 33


```
    static void Main(string[] args)
    {
      SayHelloType();
      Sample s = new Sample();
      s.SayHelloInstance();
    }
  }
}
```

The C# keyword `struct` is used to create a value type called `Sample`. This struct has three methods, two of which are static methods: `SayHelloClass` and `Main`. The method `SayHelloInstance` is an instance method and is, therefore, always invoked on values of the type.

The static method `Main` is also the entry point for the program. As far as the CLR is concerned, the entry point for a program need not be called `Main`, although in C# it always has that name. The entry point must be a static method and can be a member of either a value type or a reference type. In Listing 2.2, `Main` calls both the static and instance methods. The entry point function can return a 32-bit value indicating its success or failure; in Listing 2.2, however, `Main` returns `void` (i.e., nothing). (Methods also have visibility and accessibility—topics covered later in this chapter.)

Listing 2.2 produces the following output:

```
Hello world from Type
Hello world from instance
```

Fields A type may contain zero or more fields, each of which has a type. Like methods, fields can be either static or instance. Used to store values, they represent the *state* of a type or value. Every field has a type and a name. For example, a `Point` class may have two fields to represent its *x* and *y* coordinates. These values may exist in every instance of the type, thereby allowing the state to be different within each value. If these fields have private accessibility, which is often the desired situation, then the state of an instance remains hidden and only its other members may access it. (Visibility and accessibility are covered later in this chapter.)

The next section, on properties, gives an example of defining and using fields.

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 34

Properties Languages that target the CLR are provided with support for properties by the CLR; that is, they are not just a naming convention to be followed by developers. This relationship proves particularly useful when the goal is to provide expressive class libraries. The CLR implements properties through the use of some special metadata annotations and methods. To a certain degree, properties are "syntactic sugar": They represent `set` and `get` methods defined on logical fields of a type.

What is a logical field of a type? As an example, a Person type may have properties that represent the Person's Birth Date, Star Sign, and Age. Clearly, storing the actual date of birth is sufficient to allow all the other values to be computed and supplied at runtime. Therefore, Age can be represented as a property—that is, a logical field of a type where an actual member is not used. Properties have a name, a type, and a number of accessor methods. A type, such as the Person type, would be free to implement all of these logical members as properties.

In client code, although they may appear to be accessing public fields when they read and write to these properties, compilers will insert code to call the property's methods. These methods may compute the needed values and provide all the data validation required to ensure the integrity of the member's values. Properties exist in COM and CORBA as well. In CORBA, they are known as *attributes* (the IDL keyword used to describe them).

Listing 2.3 demonstrates the definition and use of a value type with properties and fields in C#. This program first defines a value type called `Point` with properties representing its *x* and *y* coordinates, and then writes and reads values to these properties. The value type is a C# `struct` that has two integers as its data members. Because they are passed by value by default, value types should generally be lightweight; this struct is an example of this requirement.

Listing 2.3 Use of properties and fields with a user-defined value type

```
using System;

namespace ValueType
{
  struct Point
  {
```

```
    private int xPosition, yPosition;
    public int X
    {
      get {return xPosition;}
      set {xPosition = value;}
    }
    public int Y
    {
      get {return yPosition;}
      set {yPosition = value;}
    }

  }
  class EntryPoint
  {
    static void Main(string[] args)
    {
      Point p = new Point();
      p.X = 42;
      p.Y = 42;
      Console.WriteLine("X: {0}", p.X);
      Console.WriteLine("Y: {0}", p.Y);
    }
  }
}
```

The first item in Listing 2.3 is the `using System` directive, which ensures that types whose names start with "`System.`" can be referenced without the need to fully qualify these names. Next comes the definition of a namespace called `ValueType`; this name has no particular significance, as we could have used any name for this namespace or even no namespace at all. The definition of the user-defined value type `Point` follows. This type has two fields, both of type `int`, which represent the *x* and *y* coordinates of a point.

The value type definition is followed by the definition of two more members, both properties. The definition of the properties looks a little awkward initially. The first part of the definition gives the accessibility, type, and name of the property; this information looks identical to the description of any field. The subsequent lines of the definitions provide the `set` and `get` methods for these properties. In fact, using the metadata facilities to look at

this struct (as is done in Chapter 3), it becomes apparent that two methods are generated for each property, both with the words `set_` and `get_` prefixed to the names of the properties—for example, `set_X` and `get_X`.

Note two points relating to Listing 2.3:

- The properties map to fields within the type, although such mapping is not strictly necessary.
- Properties are not limited to types such `int`; they can be of any type.

The class `EntryPoint` provides the entry point for this program. This class could have been given any name, but `EntryPoint` was chosen because it describes the class's purpose (rather than for any syntactical reason). Within `Main`, the first line *appears* to allocate a new instance of the `Point` type on the heap; in reality, this is not the case. For developers familiar with other programming languages, this idea is very counterintuitive; as value types are allocated on the stack, the local variable `p` is, in fact, allocated on the stack. Next, the use of the properties is highlighted. Notice how access to the properties appears similar to access to a public field, but the compiler generated code to call the `get_` and `set_` methods as required. Properties also offer "hints" to the just-in-time (JIT) compiler, which may choose to inline the method calls. (The JIT compiler is discussed in Chapter 4.) For simple properties such as the ones defined in Listing 2.3, little (if any) performance overhead is incurred and many reasons exist to prefer properties over publicly exposing instance fields (e.g., the elimination of versioning and data integrity issues).

Listing 2.3 produces the following output:

```
X: 42
Y: 42
```

Events As with properties, languages that target the CLR are provided with support for events by the CLR; like properties, events are not just a naming convention to be followed by developers. Events are used to expose asynchronous changes in an observed object. At a fundamental level, they are "syntactic sugar" that generates methods and associated metadata.

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 37

An event has both a name and a type. The type specifies the method signature that clients must provide for the event's callback method. When types define an event, methods to add and remove listeners are created automatically, named `add_EventName` and `remove_EventName`. Clients register to listen for events. When an event is raised, a callback method is invoked on the affected clients. When a client is no longer interested in receiving notification of events, it can remove itself from the list of listeners on an event source.

Both COM and CORBA support events, albeit somewhat differently. In COM, an interface can be marked as a `source` interface, which means that the methods in the interface need to be implemented by the client and the component will call back to the client through these methods. CORBA uses a similar method—namely, an interface is passed from a client to a server and callbacks are made through the same interface. The CORBA interface is not specifically marked as a callback interface, however, as it is in COM. The approaches employed in COM and CORBA are similar to events in the CLR, except that the CLR registers individual methods to be called back rather than interfaces (which contain a number of methods). CORBA also provides an Event Service that gives full control over events, providing, for example, both push and pull functionality. Unfortunately, a functional equivalent to CORBA Event Service does exist in the CLR.

Listing 2.4 demonstrates the definition and use of events in a value type in C#. This program defines a value type called `EventClass` that exposes an event, creates a value of this value type, attaches listeners, and then invokes the event. Events are tied to the concept of delegates in the CLR. A delegate is best described as a type-safe function pointer. With events, delegates are used to specify the signature of the method that the event will call when it is raised. For example, the definition of `ADelegate` in Listing 2.4 states that `ADelegate` is a delegate (function pointer) that can point at functions that take no parameters and return nothing. The value type `EventClass` defines an event called `AnEvent` of type `ADelegate`; that is, it can register and call back methods whose signature matches that of `ADelegate`.

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 38

Listing 2.4 Use of events with a user-defined value type

```
using System;

namespace EventSample
{
  public delegate void ADelegate();
  struct EventClass
  {
    public event ADelegate AnEvent;
    public void InvokeEvent()
    {
      if(AnEvent !=null)
        AnEvent();
    }
    static void CallMe()
    {
      Console.WriteLine("I got called!");
    }
    static void Main(string[] args)
    {
      EventClass e = new EventClass();
      e.AnEvent += new ADelegate(CallMe);
      e.AnEvent += new ADelegate(CallMe);
      e.AnEvent += new ADelegate(CallMe);
      e.InvokeEvent();
    }
  }
}
```

Within the class `EventClass`, the event can be raised by calling the event's name, such as `AnEvent()` in the method `InvokeEvent`. When this event is called, all delegates currently listening on the event are called. The sample program attaches three delegates to this event on the instance of the class called `e`. Thus, whenever `e` raises the event, the static method `CallMe` is called three times. Note that the called method does not always have to be a static method as it is in Listing 2.4.

Listing 2.4 produces the following output:

```
I got called!
I got called!
I got called!
```

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 39

Sealed Value Types

As mentioned previously, all value types inherit from specific classes—enumerations from `System.Enum` and structures from `System.ValueType`. It is not possible to build an inheritance hierarchy with value types; that is, a value type cannot inherit from another value type. In CLR terminology, a value type is said to be *sealed*. Sealing explains why instance methods are not declared as virtual in value types, because it is not possible to subtype them and, therefore, the definitions of methods cannot be overridden.

By contrast, reference types in the CLR can optionally be declared as sealed, which prohibits subtyping of these types. An example of a reference type that is sealed in the CLR is the `String` class.

Boxed Types

For every value type, including user-defined value types, there exists a corresponding object type, known as its *boxed type*. The CLR automatically generates boxed types for user-defined value types, which means that values of any value type can be boxed and unboxed:

- *Boxing* a value type copies the data from the value into an object of its boxed type allocated on the garbage collected heap.
- *Unboxing* a value type returns a pointer to the actual data—that is, the sequence of bits—held in a boxed object. (In some programming languages, unboxing not only facilitates obtaining the pointer to the data members of a boxed object but also copies the data from the boxed object into a value of the value type on the stack.)

The fact that all value types can be converted to their corresponding object types allows all values in the type system to be treated as objects if required. This situation has the effect of unifying the two fundamentally different types in the CLR, because everything can be treated as a subtype of `Object`. This approach is somewhat similar to that created by the use of COM's `IUnknown` and CORBA's `Object` interfaces, which also act as the base interface in the IDLs. Because a box type is an object type, it may support *interface types,* thereby providing additional functionality to its

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 40

unboxed representation. Object types, reference types, and interface types are described later in this chapter.

Reference Types

Reference types combine a location and a sequence of bits. The location provides identity by designating an area in memory where values can be stored *and* the type of values that can be stored there. A location is "type safe" in that only assignment-compatible types can be stored in it. (The section "Assignment Compatibility" gives an example of assignment compatibility.)

Because all reference types are allocated on the garbage collected heap and the garbage collector is free to move objects during execution, reference types are always accessed through a strongly typed reference rather than directly. As the garbage collector moves the object, the reference can be updated as part of the relocation process. As shown in Figure 2.2 on page 24, three categories of reference types exist: object types, interface types, and pointer types.

Object Types

Object types represent types that are known as classes in many languages, such as SmallTalk and Java. The built-in object types include `Object` and `String`. The CLR uses the term *object* to refer to values of an object type; the set of all exact types for all objects is known as the *object types*. Because `String` is an object type, all instances of the `String` type are therefore objects. Object types are always allocated on the garbage collected heap. Table 2.2 lists the relevant information for the CLR's built-in object types.

A number of reference types are supplied with the Base Framework. They are not considered to be built-in types because the CLR provides no inherent support for them; instead, these reference types are viewed as being under the user-defined object types that are subtypes of `Object`.

Object

All object types inherit, either directly or indirectly, from the CLR type `System.Object` class. A major facility of the `Object` class is its ability to

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 41

TABLE 2.2 CLR Built-in Reference Types: Object Types

CIL Name	Library Name	Description	CLS Support
Object	System.Object	Base class for all object types	Y
String	System.String	Unicode string	Y

enforce a singular rooted inheritance hierarchy on all types in the CLR. Although you may think that this kind of inheritance does not apply to value types, value types can be treated as a subtype of `Object` through boxing. The libraries make extensive use of the `Object` type as the parameters to, and the return type of, many functions.

The `Object` class provides a number of methods that can be called on all objects:

- `Equals` returns true if the two objects are equal. Subtypes may override this method to provide either identity or equality comparison. `Equals` is available as both a virtual method that takes a single parameter consisting of the other object with which this object is being compared and a static method that, naturally, requires two parameters.
- `Finalize` is invoked by the garbage collector before an object's memory is reclaimed. Because the garbage collector is not guaranteed to run during a program's execution, this method may not be invoked.[5] In C#, if a developer defines a *destructor,* then it is renamed to be the type's `Finalize` method.
- `GetHashCode` returns a hash code for an object. It can be used when inserting objects into containers that require a key to be associated with each such object.
- `GetType` returns the type object for this object. This method gives access to the metadata for the object. A static method on the `Type` class

5 The CLR provides no facility that guarantees the invocation of a `destructor` method. The `IDisposable` interface contains a method called `Dispose`. By convention, clients call this method when they finish with an object.

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 42

can be used for the same purpose; it does not require that an instance of the class be created first.

- `MemberwiseClone` returns a shallow copy of the object. This method has protected accessibility and, therefore, can be accessed only by subtypes. It cannot be overridden. If a deep copy is required, then the developer should implement the `ICloneable` interface.
- `ReferenceEquals` returns true if both object references passed to the method refer to the same object. It also returns true if both references are null.
- `ToString` returns a string that *represents* the object. As defined in `Object`, this method returns the name of the type of the object—that is, its exact type. Subtypes may override this method to have the return string represent the object as the developer sees fit. For example, the `String` class returns the value of the string and not the name of the `String` class.

Most of the methods defined on `Object` are public. `MemberwiseClone` and `Finalize`, however, have protected access; that is, only subtypes can access them. The following program output shows the assembly qualified name and the publicly available methods on the `Object` class.(Assemblies are covered in Chapter 5.)

```
System.Object, mscorlib, Version=1.0.2411.0,
 Culture=neutral, PublicKeyToken=b77a5c561934e089
Int32 GetHashCode()
Boolean Equals(System.Object)
System.String ToString()
Boolean Equals(System.Object, System.Object)
Boolean ReferenceEquals(System.Object, System.Object)
System.Type GetType()
Void .ctor()
```

A simple program using the metadata facilities of the CLR generated this output. Chapter 3 describes how to build this approximately 10-line program and explains the significance of the assembly qualified name

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 43

shown as the first line of output. In brief, the program retrieves the Object class's Type object and then displays its public method's prototypes. The two protected methods are not shown. The preceding output also shows the use of built-in types, such as Boolean, Int32, and String.

Listing 2.5 demonstrates how many classes override the ToString method. The default behavior is to print a string representing the type of the object on which it is invoked—that is the action of the Object class. String and Int32 have both overridden this behavior to provide a more intuitive string representation of the object on which it is invoked.

Listing 2.5 Overriding the ToString *method*

```
using System;

namespace Override
{
  class Sample
  {
    static void Print(params Object[] objects)
    {
      foreach(Object o in objects)
        Console.WriteLine(o.ToString());
    }
    static void Main(string[] args)
    {
      Object o = new Object();
      String s = "Mark";
      int i = 42;
      Print(o, s, i);
    }
  }
}
```

One interesting feature of Listing 2.5 relates to the use of the C# params keyword. You can call the Print method with any number of arguments, and these arguments will be passed to the method in an array that holds Object references. Within the Print method, each member of the array will have its virtual ToString method called, so the correct method for each argument will be invoked. You may wonder how the value i of type int is passed given that it is a value type: It is boxed.

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 44

Listing 2.5 produces the following output:

```
System.Object
Mark
42
```

The constructor for `Object` *should* be called whenever an object is created; it *must* be called if the goal is to produce verifiable code. Chapter 4 discusses verifiable code in more detail, but for now consider "verifiable code" to mean proven type-safe code. As a simplification of the rules, compiler writers must ensure that a call to the constructor for the base class occurs during construction of all derived classes. This call can occur at any time during the construction of derived classes; it need not be the first instruction executed in the constructor of subtype. Developers should be aware that construction of the base class may not have occurred when a user-defined constructor starts running.

String

Like `Object`, `String` is a built-in type in the CLR. This class is sealed, which means that no type can be subtyped from it. The sealed nature of `String` allows for much greater optimization by the execution system if it is known that subtypes cannot be passed where a type, such as `String`, is expected. Strings are also immutable, such that calling any method to modify a string creates a new string. The fact that the `String` class is sealed and immutable allows for an extremely efficient implementation. Developers can also use a `StringBuilder` class if creating temporary strings while doing string manipulations with the `String` class proves too expensive.

The `String` class contains far too many members to describe them all here. A nonexhaustive list of the general functionality of the class would include the following abilities:

- Constructors: No default constructor exists. Constructors take arguments of type `char`, arrays of `char`, and pointers to `char`.
- Compare: These methods take two strings (or parts thereof) and compare them. The comparison can be case sensitive or case insensitive.

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 45

- Concatenate: This method returns a new string that represents the concatenation of a number of individual strings.
- Conversion: This method returns a new string in which the characters within the string have been converted to uppercase or lowercase.
- Format: This method takes a format string and an array of objects and returns a new string. It replaces each placeholder in the format string with the string representation of an object in the array.
- IndexOf: This method returns an integer index that identifies the location of a character or string within another string.
- Insert: This method returns a new string representing the insertion of one string into another string.
- Length: This property represents the length of the string. It supplies only a `get` method—that is, it is a read-only value.
- Pad and Trim Strings: These methods return a new string that represents the original string with padding characters inserted or removed, respectively.

Listing 2.6 is a simple example demonstrating the use of the built-in `String` reference type in C#. This program creates objects of type `string`, converts them between uppercase and lowercase, concatenates them, and then checks for a substring within a string.

Listing 2.6 *Using the* `String` *reference type*

```
using System;

namespace StringSample
{

  class Sample
  {
    static void Main(string[] args)
    {
      String s = "Mark";
      Console.WriteLine("Length is: " + s.Length);
      Console.WriteLine(s.ToLower());
      String[] strings = {"Damien", "Mark", "Brad"};
      String authors = String.Concat(strings);
      if(authors.IndexOf(s) > 0)
```

continues

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 46


```
            Console.WriteLine("{0} is an author", s.ToUpper());
        }
    }
}
```

Main begins by defining an instance of the String class called s with the contents "Mark". Normally, you allocate an instance of a reference type by using a language construct such as new. In the CLR and in many languages, such as C#, a string is a special case because it is a built-in type and, therefore, can be allocated using special instructions or different syntax. The CLR has an instruction for just this purpose. Likewise, as shown in Listing 2.6, C# has a special syntax for allocating a string.

The next line accesses a property of the string object called Length, returns an integer representing the number of characters in a string. You have already seen how a property is defined, but now you may wonder how an integer is "added" to a string. Of course, the answer is that it is not. While an int is a value type, a reference type is always created for all value types, known as the boxed type. In this case, code is inserted to box the integer value and place it on the garbage collected heap.

How, then, are a string and a boxed integer concatenated? Again, the answer is that they are not. One of the overloaded concatenation methods on the String class takes parameters of type Object. The compiler calls this method, and the resulting string is added to the first string; this value is then written to the screen. Developers need to be aware that compilers may silently insert a number of calls to provide conversion and coercion methods, such as boxing. The exact behavior is a function of the particular compiler; some compilers will insert these method calls, whereas others will require the developer to call them explicitly. Of course, Listing 2.6 could be rewritten to avoid boxing altogether; again, this choice requires the developer to know and understand the code generated by the particular compiler.

Finally, the program in Listing 2.6 makes a single string containing all the names of the authors of this book and then searches to see whether "Mark" is one of the authors.

Your attention is also drawn to the generation of temporary string objects in Listing 2.6. Note that calls to methods such as ToLower and

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 47

`ToUpper` generate temporary string objects that may produce performance problems.

Listing 2.6 produces the following output:

```
Length is: 4
mark
MARK is an author
```

Interface Types

Programming with interface types is an extremely powerful concept; in fact, with COM and CORBA, interface-based programming is the predominate paradigm. Object-oriented programmers will already be familiar with the concept of substituting a derived type for a base type. Sometimes, however, two classes are not related by implementation inheritance but do share a common contract. For example, many classes contain methods to save their state to and from persistent storage. For this purpose, classes not related by inheritance may support common interfaces, allowing programmers to code for their shared behavior based on their shared interface type rather than their exact types.

An *interface type* is a partial specification of a type. This contract binds implementers to providing implementations of the members contained in the interface. Object types may support many interface types, and many different object types would normally support an interface type. By definition, an interface type can never be an object type or an exact type. Interface types may extend other interface types; that is, an interface type can inherit from other interface types.

An interface type may define the following:

- Methods (static and instance)
- Fields (static)
- Properties
- Events

Of course, properties and events equate to methods. By definition, all instance methods in an interface are public, abstract, and virtual. This is the opposite of the situation with value types, where user-defined methods on

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 48

the value type, as opposed to the boxed type, are always nonvirtual because no inheritance from value types is possible. The CLR does not include any built-in interface types, although the Base Framework provides a number of interface types.

Listing 2.7 demonstrates the use of the `IEnumerator` interface supported by array objects. The array of `String` objects allows clients to enumerate over the array by requesting an `IEnumerator` interface. (Developers never need to define an array class even for their own types, because the CLR generates one automatically if needed. The automatically defined array type implements the `IEnumerator` interface and provides the `GetLength` method so they can be used in exactly the same manner as arrays for built-in types.) The `IEnumerator` interface defines three methods:

- `Current` returns the current object.
- `MoveNext` moves the enumerator on to the next object.
- `Reset` resets the enumerator to its initial position.

Listing 2.7 Using the `IEnumerator` *interface*

```
using System;
using System.Collections;

namespace StringArray
{
  class EntryPoint
  {
    static void Main(string[] args)
    {
      String [] names = {"Brad", "Damien", "Mark"};
      IEnumerator i = names.GetEnumerator();
      while(i.MoveNext())
        Console.WriteLine(i.Current);
    }
  }
}
```

Of course, you could have written the `while` loop in Listing 2.7 as a `foreach` loop, but the former technique seems slightly more explicit for demonstration purposes. The program produces the following output:

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 49


```
Brad
Damien
Mark
```

Array objects support many other useful methods, such as `Clear`, `GetLength`, and `Sort`. The `Sort` method makes use of another Base Framework interface, `IComparable`, which must be implemented by the type that the array holds. Array objects also provide support for synchronization, an ability that is provided by methods such as `IsSynchronized` and `SyncRoot`.

Pointer Types

Pointer types provide a means of specifying the location of either code or a value. The CLR supports three pointer types:

- Unmanaged function pointers refer to code and are similar to function pointers in C++.[6]
- Managed Pointers are known to the garbage collector and are updated if they refer to an item that is moved on the garbage collected heap.
- Unmanaged pointers are similar to unmanaged function pointers but refer to values. Unmanaged pointers are not CLS compliant, and many languages do not even expose syntax to define or use them. By comparison, managed pointers can point at items located on the garbage collected heap and are CLS compliant.

The semantics for these pointer types vary greatly. For example, pointers to managed objects must be registered with the garbage collector so that as objects are moved on the heap, the pointers can be updated. Pointers to local variables have different lifetime issues. When using unmanaged pointers, objects will often need to be *pinned* in memory so that the garbage collector will not move the object while it is possible to access the object via an unmanaged pointer.

This book will not cover pointer types in detail for two reasons. First, a greater understanding of the .NET platform architecture is needed to

[6] Delegates, discussed later, provide an alternative to unmanaged function pointers.

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 50

understand the semantic issues relating to pointers. Second, most programming languages will abstract the existence of pointers to such a degree that they will be invisible to programmers.

Example: User-Defined Object Type

User-defined object types are the types that most developers will use to expose their services. These types correspond most closely to classes in many object-oriented languages. As discussed earlier, classes can define different types of members. However, at the fundamental level, only two types of members exist: methods and fields. Abstractions layered over methods include the notions of properties and events. Methods are also specialized to produce constructors, which can be either instance or class constructors (`.ctor` and `.cctor`). Once again, on a basic level these constructs are simply methods with added semantics and additional runtime support. This runtime support includes the notion that the class constructor will begin executing before any of its members is used for the first time.[7]

Let's look an example of how to create a user-defined object type. To make the example more complete, Listing 2.8 creates an object type called `Point` that implements two interfaces—one interface with an event, and one interface with properties—as well as the user-defined object type. After creating the `Point` type, the program attaches a listener to the event, writes the properties via interface references that fire the event, and then accesses the properties.

---

7 The language designer chooses which CLR abstractions are exposed to developers and how they are exposed. For example, some languages that target the CLR provide little support for object-oriented programming. Instead, they just provide global functions that communicate by passing data as parameters. These languages may still participate in the CLR, but their global functions will normally be mapped to static functions on a "well-known" class, possibly one with a name like `Global`. Although the way these languages expose their functionality in the CLR is very interesting, the many variations are beyond the scope of this book. The appendices do describe how many languages map their semantics to the CLR.

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 51

Listing 2.8 Creating a user-defined object type

```
using System;

namespace ObjectType
{
  public delegate void ADelegate();
  public interface IChanged
  {
    event ADelegate AnEvent;
  }
  public interface IPoint
  {

    int X {get; set;}
    int Y {get; set;}
  }
  class Point: IPoint, IChanged
  {
    private int xPosition, yPosition;
    public event ADelegate AnEvent;
    public int X
    {
      get {return xPosition;}
      set {xPosition = value; AnEvent();}
    }
    public int Y
    {
      get {return yPosition;}
      set {yPosition = value; AnEvent();}
    }
  }
  class EntryPoint
  {
    static void CallMe()
    {
      Console.WriteLine("I got called!");
    }
    static void Main(string[] args)
    {
      Point p = new Point();
      IChanged ic = p;
      IPoint ip = p;
      ic.AnEvent += new ADelegate(CallMe);
      ip.X = 42;
```

continues

```
      ip.Y = 42;
      Console.WriteLine("X: {0} Y: {1} ", p.X, p.Y);
    }
  }
}
```

The delegate is declared first in the program. It serves as a delegate for functions that accept no arguments and return nothing (`void`). The returning of `void` is common with many delegates.

The first interface, which is called `IChanged`, contains a single event that is of the same type as the delegate. The second interface, which is called `IPoint`, provides two properties that allow access to two properties known as `X` and `Y`. Both properties have `get` and `set` methods.

Next comes the definition of the class `Point`. It would be fair to say that a `Point` could just as easily be a value type as an object type. Here, however, it is used as a simple abstraction. The class `Point` inherits from `Object`, although this relationship is not stated explicitly, and from the two interfaces just defined, `IChanged` and `IPoint`. By inheriting from `IPoint`, for example, the class `Point` agrees to implement the four abstract methods required for the two properties. The `set` methods fire the event whenever they are called.

The class `EntryPoint` provides the entry point for the `Main` program. This class also provides a static function called `CallMe`, which matches the prototype of the delegate. The program registers that this method is to be called whenever the event is fired on the `Point` object named `p`.

A number of fine points can be emphasized about the program in Listing 2.8. First, only one object is ever created through the activity of the `new` operator on the first line of `Main`. The `IChanged` and `IPoint` interface types are merely references used to access this `Point` object. As `Point` is the exact type for the object, all methods available on the `Point` class, including the methods in both interfaces, are available through a reference of type `Point`—in this case, `p`. This relationship is demonstrated by invoking the methods `X` and `Y` in the `WriteLine` method calls. The interface types can call only the methods in their own interface, even though the object has more functionality.

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 53

Listing 2.8 produces the following output:

```
I got called!
I got called!
X: 42 Y: 42
```

Example: Use of Interfaces on Value Types

When considering the use of interfaces on a value type, often the first question asked is, "Can value types support interfaces?" The simple answer is yes; you can call the methods defined in the interface directly on the value type without incurring any overhead. The situation is different, however, if you call the methods through an interface reference. An interface references objects allocated on the garbage collected heap, and value types are normally allocated on the stack—so how does this work? The answer is a familiar one: boxing.

Listing 2.9 demonstrates the use of events and properties in a user-defined interface, `IPoint`. This interface is implemented by a value type, `Point`, and an object type, `EntryPoint`, provides the entry point method. Similar to Listing 2.8, this program creates a value of the value type and then accesses its properties.

Listing 2.9 Using events and properties in a user-defined interface

```
using System;

namespace InterfaceSample
{
  public delegate void Changed();

  interface IPoint
  {
    int X
    { get; set;}
    int Y
    { get; set;}
  }

  struct Point: IPoint
  {
```

continues

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 54


```
      private int xValue, yValue;
      public int X
      {
        get { return xValue; }
        set { xValue = value; }
      }
      public int Y
      {
        get { return yValue; }
        set { yValue = value; }
      }
    }
    public class EntryPoint
    {
      public static int Main()
      {
        Point p = new Point();
        p.X = p.Y = 42;
        IPoint ip = p;
        Console.WriteLine("X: {0}, Y: {1}",
                          ip.X, ip.Y);
        return 0;
      }
    }
}
```

Listing 2.9 produces the following output:

```
X: 42, Y: 42
```

The interesting point regarding this program is the use of the interface `ip` to access the properties of `p` when calling the `WriteLine` method. As Listing 2.9 is written, the interface `ip` can only refer to reference types and `p` is a value type, so `p` is silently boxed and `ip` references the boxed object, which contains a copy of `p`.

Assignment Compatibility

As the concepts of object types and interface types have now been explained, albeit very simply, the question of assignment compatibility can be addressed. A simple definition of assignment compatibility for reference

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 55

types is that a location of type `T`, where `T` may be either an object type or an interface type, can refer to any object:

- With an exact type of `T`, or
- That is a subtype of `T`, or
- That supports the interface `T`.

Listing 2.10 demonstrates aspects of assignment compatibility. The program starts by creating values of two different types and two interface references:

- A value of type `System.Int32` called `i`
- An `Object` reference called `o`
- A `String` reference called `s`
- An `IComparable` reference called `ic`

Listing 2.10 *Assignment compatibility*

```
using System;

namespace AssignmentCompatibility
{
  class Sample
  {
    static void Main(string[] args)
    {
      System.Int32 i = 42;
      Object o;
      String s = "Brad";
      IComparable ic;

      // OK
      o = s;
      ic = s;

      // OK - box and assign
      o = i;
      ic = i;

      // compiler error
      i = s;
```

continues

```
      // runtime error
      s = (String) o;
    }
  }
}
```

The assignments of s to o and of s to ic are all compatible, so they compile and execute without any errors. The next two assignments are also compatible, so they, too, compile and execute without any errors. The surprise may come with the fact that boxing of i is needed, which the C# compiler performs silently in both cases. Other languages might require explicit code to handle this boxing.

The assignment of s to i is not compatible, so it generates a compile-time error. Unfortunately, downcasting (narrowing) is inherently problematic, making the last assignment difficult to test for compatibility until runtime. The JIT compiler will insert code to check the type of the object referenced by o before performing the assignment. In this case, o refers to a boxed integer type at the time of assignment, so the cast fails and an exception is thrown. Of course, if o did refer to a string, then the cast would execute without throwing an exception.

Assignment compatibility is a crucial mechanism that helps ensure type safety in the CLR. The execution system can validate assignment to a reference to verify that the assignment ensures that the type and the reference are compatible.

Nested Types

You can define types inside of other types, known as nested types. Nested types have access to the members of their enclosing types as well as to the members of their enclosing type's base class (subject to the normal accessibility conditions, as will be discussed shortly). Listing 2.11 demonstrates their use. This program defines three different object types: Outer, Middle, and Inner. Inner contains the entry point and accesses the enumerations defined in its enclosing types.

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 57

Listing 2.11 Nested types

```
using System;

namespace NestedTypes
{
  class Outer
  {
    private enum Day {Sunday, Monday, Tuesday,
        Wednesday, Thursday, Friday, Saturday};
    class Middle
    {
      private enum Month {January = 1, February,
              March, April, May, June, July, August,
              September, October, November, December};
      class Inner
      {
        static void Main(string[] args)
        {
          Day d = Day.Sunday;
          Month m = Month.September;
          Console.WriteLine(
             "Father's day is the first "
             + d + " in " + m);
        }
      }
    }
  }
}
```

Running this program generates the following output:[8]

```
Father's day is the first Sunday in September
```

Visibility

Visibility refers to whether a type is available outside its assembly. If a type is exported from its assembly, then it is visible to types in other assemblies. `Object` and `String` are examples of two types that are exported from their

8 Father's Day is the first Sunday in September in Australia.

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 58

assemblies. Visibility does not affect members of types, whereas accessibility (discussed next) can be used to limit access to members.

Accessibility

Members of a type can have different *accessibility* levels. Accessibility levels define which other types can access the members of a type. In many situations, it is desirable to limit the accessibility of a member. If one member implements some functionality for a type, such as data validation of field values, then it may be useful to all other members of that type but may be considered an implementation detail of the type. Developers can limit the accessibility of the member so as to prevent clients from using it. This practice helps to localize the effects of change. For example, if the signature of such a privately scoped member changes, then the change will not affect any of the clients of this type. Thus the effect of the change remains limited to this type's definition. Conversely, accessibility may be restricted to a higher level than private but less than public. This choice could, for example, widen the effect of a change, possibly to the type's assembly, but keep it a much smaller effect than changing a publicly available member.

The CLR supports a number of accessibility levels:

- Public: available to all types
- Assembly: available to all types within this assembly
- Family: available in the type's definition and all of its subtypes
- Family or Assembly: available to types that qualify as either Family or Assembly
- Family and Assembly: available to types that qualify as both Family and Assembly
- Compiler-controlled: available to types within a single translation unit
- Private: available only in the type's definition

All members in an interface are public. Types are permitted to widen accessibility. That is, a subtype may make a member more visible, as sometimes occurs with inherited virtual methods, for example. This approach ensures that the method is at least equally visible in the subtype. Use of this

widening facility, although legal, is considered problematic at best and should be avoided.

Summary

This chapter provided a detailed view of the type system of the Common Language Runtime. The CLR supports an object-oriented programming style, which is reflected in its type system. The CTS contains two basic types: value types and reference types. Value types are similar to primitive types in many programming languages. Reference types are analogous to classes in many object-oriented languages; they are further divided into object, interface, and pointer types.

All object types inherit from `Object`. Object types may introduce new methods, these methods can be either static or instance; instance methods can also be virtually dispatched. Methods may, of course, follow the rules for properties or events, thereby gaining additional support from the CLR or other tools.

Methods may be overloaded. In other words, a class can have many methods with the same name as long as the types of the methods' parameters are different. When a program calls a method with multiple names, the choice of which method is called depends on the argument list and its correspondence with the available methods' signatures.

Virtual methods may also be overridden. In such a case, subtypes supply methods with exactly the same signatures as the base classes. The compiler may issue instructions to do runtime dispatching of the method call. The method selection is then left until runtime, when the exact type of the object on which the method acts is known.

5298 ch02_019-060.qxd 10/10/2002 6:12 PM Page 60