CHAPTER 18

Libraries and Interoperating with Other Languages

Programming in different languages is like composing pieces in different keys, particularly if you work at the keyboard. If you have learned or written pieces in many keys, each key will have its own special emotional aura. Also, certain kinds of figurations "lie in the hand" in one key but are awkward in another. So you are channeled by your choice of key. In some ways, even enharmonic keys, such as C-sharp and D-flat, are quite distinct in feeling. This shows how a notational system can play a significant role in shaping the final product.

Gödel, Escher, Bach: an eternal golden braid, Hofstadter, 1980, Chapter X

Software integration and reuse is becoming one of the most relevant activities in software development. This chapter discusses how F# programs can interoperate with the outside world, accessing code available from both .NET and other languages.

Types, memory and interoperability

F# programs need to call libraries, even for the very basic tasks, such as printing or accessing files, and these libraries come in binary form as part of the Common Language Runtime (CLR), the piece of software responsible for running programs after compilation. Libraries can be, and have been, written using different programming languages, leveraging on the fact that the output of the compilation has a common format that the CLR uses for executing their code. Even considering just the core libraries, such as `mscorlib.dll` (containing essential types such as `System.String`, the underlying type for the F# type `string`), most of the code has been written using C#.

You need to understand language interoperability, at least to some extent, to be a proficient F# programmer. There are four levels of interoperability to consider:

- F# library
- Non-F# .NET library
- COM library
- Binary DLL library

If a library has been compiled using the F# compiler and it is meant to be used from F#, everything feels like F#—types, modules, and functions are available as in F# source form, except for scope restrictions introduced by visibility clauses, such as `private`. The F# compiler, unless told otherwise, generates additional information into the binary files meant to be used when compiling other F# source files that depend on them.

■ **Note:** For more information about designing F# libraries to be used from other .NET languages, see the F# component design guidelines, available at `http://tinyurl.com/fs-component-design-guidelines`

In all the other cases, the problem is essentially the same: how to represent types defined in a library within F# so that their values can be safely accessed from the program. Sometimes a value can be accessed as is, accessing the single in-memory copy from F# generated code; otherwise, some form of *marshalling* is needed to convert a value from the library representation to one accessible from F#. Although simple in theory, sharing types and their values among different runtimes of programming languages results in many subtle issues in practice, with increasing complexity depending how close their runtime support is. Even a relatively simple type, such as the .NET string, has a memory representation very different from C.

.NET libraries compiled with languages other than F# and designed to be consumed from many .NET languages (i.e., designed according to the Common Language Specification, http://tinyurl.com/dotnetCLS, to define a subset of the .NET type system that can be safely shared among different programming languages) are easy accessible from F# programs in the form of a set of classes that can be instantiated and whose methods can be invoked. F# specific types, such as tuples or discriminated unions, are usually unavailable as type of method arguments; thus, you need some form of marshalling to convert F# types into simpler .NET types. Apart from these small issues, using these libraries is seamless, because the runtime is shared and the .NET binary format contains meta-information, such as the type structure, that can be used by compilers to spare many details related to interoperability.

COM components have been constituents of the Windows operating system and are still widely used for integrating binary components. Notorious ActiveX controls are, in fact, a kind of COM component capable of displaying a UI inside a host application. The CLR itself is exposed as a COM component, and it tightly couples with the COM infrastructure. The COM type system and memory management are less rich and safe than .NET, but type libraries and `IDispatch` interface approximate .NET reflection, and the CLR usually can wrap COM components as .NET types. COM is still underlying in some form to the newest WinRT API released by Microsoft for programming Windows 8-style applications.

C has been de facto standard for language interoperability for many years. CLR is capable of interoperating with C binary interfaces at the cost of manually annotating .NET types and, sometime, taking care explicitly of marshalling and memory handling.

This chapter explores the various levels of interoperability, giving an overview of the .NET libraries in the first place. An overview of the COM components is also given, with the goal of enabling F# programmers to consume these software units. Finally, the platform invoke API is introduced, showing how to link binary libraries and interacting with C code; this interface is part of the .NET ECMA standard and it is available on different CLR implementations, including Mono, the open-source implementation running on MacOS X and Linux.

Libraries: A High-Level Overview

One way to get a quick overview of the .NET Framework and the F# library is to look at the primary DLLs and namespaces contained in them. Recall from Chapters 2 and 7 that DLLs correspond to the *physical* organization of libraries, and that namespaces and types give the *logical* organization of a naming hierarchy. Let's look at the physical organization first. The types and functions covered in this chapter are drawn from the DLLs in Table 18-1.

Table 18-1. *DLLs containing the library constructs referred to in this chapter*

DLL Name	Notes
`mscorlib.dll`	Minimal system constructs, including the types in the `System` namespace.
`System.dll`	Additional commonly used constructs in namespaces, such as `System` and `System.Text`.
`System.Xml.dll`	See the corresponding namespace in Table 18-2.
`System.Data.dll`	See the corresponding namespace in Table 18-2.
`System.Drawing.dll`	See the corresponding namespace in Table 18-2.
`System.Web.dll`	See the corresponding namespace in Table 18-2.
`System.Windows.Forms.dll`	See the corresponding namespace in Table 18-2.
`System.Core.dll`	The foundation types for LINQ and some other useful types. From .NET 3.5 onward.
DLL Name	Notes
`WindowsBase.dll`	Core functionality for Windows Presentation Foundation.
`PresentationCore.dll`	Core functionality for Windows Presentation Foundation.
`PresentationFramework.dll`	Core functionality for Windows Presentation Foundation.
`FSharp.Core.dll`	Minimal constructs for F# assemblies.

To reference additional DLLs, you can embed a reference directly into your source code in F# scripting files that use `.fsx` extension. For example:

```
#I @"C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.5";;
#r "System.Core.dll";;
```

The first line specifies an include path, the equivalent of the `-I` command-line option for the F# compiler. The second line specifies a DLL reference, the equivalent of the `-r` command-line option. Chapter 7 described these. If you're using Visual Studio, you can adjust the project property settings for your project.

■ **Note**: Hundreds of high-quality frameworks and libraries are available for .NET, and more are appearing all the time. For space reasons, this chapter covers only the .NET libraries and frameworks listed in Table 18-1. "Some Other .NET Libraries" lists some libraries you may find interesting.

Namespaces from the .NET Framework

Table 18-2 shows the primary namespaces in .NET Framework DLLs from Table 18-1. In some cases, parts of these libraries are covered elsewhere in this book; the table notes these cases. For example, Chapter 4 introduced portions of the .NET I/O library from the `System.IO` namespace.

Table 18-2. Namespaces in the DLLs from Table 18-1, with MSDN descriptions

Namespace	Description
`System`	Types and methods that define commonly used value and reference data types, events and event handlers, interfaces, attributes, and processing exceptions, supporting data-type conversions, mathematics, application environment management, and runtime supervision of managed and unmanaged applications. See Chapter 3 for many of the basic types in this namespace.
`System.CodeDom`	Types that can be used to represent the elements and structure of a source-code document. Not covered in this book.
Namespace	Description
`System.Collections`	Types that define various nongeneric collections of objects, such as lists, queues, and bit arrays. Partially covered in "Using Further F# and .NET Data Structures" later in this chapter.
`System.Collections.Generic`	Types that define generic collections. See Chapter 4 and "Using Further F# and .NET Data Structures" later in this chapter.
`System.ComponentModel`	Types that are used to implement the runtime and design-time behavior of components and controls. See Chapter 16.
`System.Configuration`	Types that provide the programming model for handling configuration data.
`System.Data`	Types that represent the ADO.NET database access architecture. See Chapter 13.
`System.Diagnostics`	Types that allow you to interact with system processes, event logs, and performance counters. See Chapter 19.
`System.Drawing`	Types that allow access to GDI+ basic graphics functionality. More advanced functionality is provided in the `System.Drawing.Drawing2D`, `System.Drawing.Imaging`, and `System.Drawing.Text` namespaces. See Chapter 16.
`System.Globalization`	Types that define culture-related information, including the language, the country/region, the calendars in use, the format patterns for dates, the currency, the numbers, and the sort order for strings. Not covered in this book.
`System.IO`	Types that allow reading and writing files and data streams, as well as types that provide basic file and directory support. See Chapter 4 for a basic overview.
`System.Media`	Types for playing and accessing sounds and other media formats. Not covered in this book. .NET 3.0 and later.
`System.Net`	Types to programmatically access many of the protocols used on modern networks. See Chapters 2 and 14 for examples and a basic overview.
`System.Reflection`	Types that retrieve information about assemblies, modules, members, parameters, and other entities in managed code. See Chapter 17 for a brief overview.
`System.Reflection.Emit`	Types for generating .NET code dynamically at runtime.
`System.Resources`	Types that let you create, store, and manage various culture-specific resources used in an application. See Chapter 7 for a brief overview.

Namespace	Description
`System.Security`	Types to interface with the underlying structure of the CLR security system, including base classes for permissions. Not covered in this book.
`System.Text`	Types representing ASCII, Unicode, UTF-8, and other character encodings. Also abstract types for converting blocks of characters to and from blocks of bytes. See Chapter 3 and "Using Regular Expressions and Formatting" later in this chapter.
`System.Threading`	Types for creating and synchronizing threads and, in .NET 4.0, tasks. Also parallel operations and some functionality related to cancellation. See Chapter 11.
`System.Web`	Types that enable Web applications. See Chapter 14.
`System.Windows.Forms`	Types for creating windowed applications. See Chapter 16.
`System.Xml`	Types that implement standards-based support for processing XML. See Chapter 8.
`Microsoft.Win32`	Types that wrap Win32 API common dialog boxes and components. Not covered in this book.

Namespaces from the F# Libraries

Table 18-3 shows the primary namespaces in F# library DLLs from Table 18-1. The following are opened by default in F# code:

```
Microsoft.FSharp.Core
Microsoft.FSharp.Collections
Microsoft.FSharp.Control
Microsoft.FSharp.Text
```

Table 18-3. *Namespaces in the DLLs from Table 18-1*

Namespace	Description
`Microsoft.FSharp.Core`	Provides primitive constructs related to the F# language, such as tuples. See Chapter 3.
`Microsoft.FSharp.Collections`	Provides functional programming collections, such as sets and maps implemented using binary trees. See Chapter 3 and "Using Further F# and .NET Data Structures" later in this chapter.
Namespace	Description
`Microsoft.FSharp.Control`	Provides functional programming control structures, including asynchronous and lazy programming. Chapter 11 covers programming with the `IEvent<'T>` type and the `IEvent` module, as well as the `Async<'T>` type.
`Microsoft.FSharp.Text`	Provides types for structured and `printf`-style textual formatting of data. See Chapter 4 for an introduction to `printf` formatting.
`Microsoft.FSharp.Reflection`	Provides extensions to the `System.Reflection` functionality that deal particularly with F# record and discriminated union values. See Chapter 17 for a brief introduction, and see "Further Libraries for Reflective Techniques" section later in this chapter for more details.
`Microsoft.FSharp.Quotations`	Provides access to F# expressions as abstract syntax trees. See Chapter 17.

Using the System Types

Table 18-4 shows some of the most useful core types from the `System` namespace. These types are particularly useful because of the care and attention taken crafting them and the functionality they provide.

Table 18-4. *Useful core types from the* `System` *Namespace*

Function	Description
`System.DateTime`	A type representing a date and time
`System.DayOfWeek`	An enumeration type representing a day of the week
`System.Decimal`	A numeric type suitable for financial calculations requiring large numbers of significant integral and fractional digits and no round-off errors
`System.Guid`	A type representing a 128-bit globally unique ID
`System.Nullable<'T>`	A type with an underlying value type `'T` but that can be assigned `null` like a reference type
`System.TimeSpan`	A type representing a time interval
`System.Uri`	A type representing a uniform resource identifier (URI), such as an Internet URL

Many .NET types are used to hold static functions, such as those for converting data from one format to another. Types such as `System.Random` play a similar role via objects with a small amount of state. Table 18-5 shows some of the most useful of these types.

Table 18-5. *Useful services from the* `System` *Namespace*

Function	Description
`System.BitConverter`	Contains functions to convert numeric representations to and from bit representations
`System.Convert`	Contains functions to convert between various numeric representations
`System.Math`	Contains constants and static methods for trigonometric, logarithmic, and other common mathematical functions
`System.Random`	Provides objects to act as random-number generators
`System.StringComparer`	Provides objects implementing various types of comparisons on strings (case insensitive, and so on)

Using Further F# and .NET Data Structures

As you saw in Chapter 2, F# comes with a useful implementation of some functional programming data structures. Recall that functional data structures are *persistent*: you can't mutate them, and if you add an element or otherwise modify the collection, you generate a new collection value, perhaps sharing some internal nodes but from the outside appearing to be a new value.

Table 18-6 summarizes the most important persistent functional data structures that are included in `FSharp.Core.dll`. It's likely that additional functional data structures will be added in future F# releases.

***Table 18-6.** The F# Functional Data Structures from `Microsoft.FSharp.Collections`*

Type	Description
`List<'T>`	Immutable lists implemented using linked lists
`Set<'T>`	Immutable sets implemented using trees
`Map<'Key,'Value>`	Immutable maps (dictionaries) implemented using trees
`LazyList<'T>`	Lists generated on demand, with each element computed only once

System.Collections.Generic and Other .NET Collections

Table 18-7 summarizes the imperative collections available in the `System.Collections.Generic` namespace.

***Table 18-7.** The .NET and F# imperative data structures from `System.Collections.Generic`*

Type	Description
`List<'T>`	Mutable, resizable integer-indexed arrays, usually called `ResizeArray<'T>` in F#.
`SortedList<'T>`	Mutable, resizable lists implemented using sorted arrays.
`Dictionary<'Key,'Value>`	Mutable, resizable dictionaries implemented using hash tables.
`SortedDictionary<'Key,'Value>`	Mutable, resizable dictionaries implemented using sorted arrays.
`Queue<'T>`	Mutable, first-in, first-out queues of unbounded size.
Type	Description
`Stack<'T>`	Mutable, first-in, last-out stacks of unbounded size.
`HashSet<'T>`	Mutable, resizable sets implemented using hash tables. New in .NET 3.5. The F# library also defines a `Microsoft.FSharp.Collections.HashSet` type usable in conjunction with earlier versions of .NET.

SOME OTHER COLLECTION LIBRARIES

Two additional libraries of .NET collections deserve particular attention. The first is PowerCollections, currently provided by Wintellect. It provides additional generic types, such as `Bag<'T>`, `MultiDictionary<'Key,'Value>`, `OrderedDictionary<'Key,'Value>`, `OrderedMultiDictionary<'T>`, and `OrderedSet<'T>`. The second is the C5 collection library, provided by ITU in Denmark. It includes implementations of some persistent/functional data structures, such as persistent trees, that may be of particular interest for use from F#.

Supervising and Isolating Execution

The .NET `System` namespace includes a number of useful types that provide functionality related to the execution of running programs in the .NET Common Language Runtime. Table 18-8 summarizes them.

Table 18-8. *Types related to runtime supervision of applications*

Function	Description
`System.Runtime`	Contains advanced types that support compilation and native interoperability
`System.Environment`	Provides information about, and the means to manipulate, the current environment and platform.
`System.GC`	Controls the system garbage collector. Garbage collection is discussed in more detail later in this chapter.
Function	Description
`System.WeakReference`	Represents a weak reference, which references an object while still allowing that object to be reclaimed by garbage collection.
`System.AppDomain`	Represents an application domain, which is a software-isolated environment in which applications execute. Application domains can hold code generated at runtime and can be unloaded.

Further Libraries for Reflective Techniques

As discussed in Chapter 17, .NET and F# programming frequently use reflective techniques to analyze the types of objects, create objects at runtime, and use type information to drive generic functions in general ways. For example, in Chapter 17, you saw an example of a technique called *schema compilation*, which was based on .NET attributes, F# data types, and a compiler to take these and use reflective techniques to generate an efficient text-file reader and translator. The combination of reflective techniques and .NET generics allows programs to operate at the boundary between statically typed code and dynamically typed data.

Using General Types

There are a number of facets to reflective programming with .NET. In one simple kind of reflective programming, a range of data structures are accessed in a general way. For example, .NET includes a type `System.Array` that is a parent type of all array types. The existence of this type allows you to write code that is generic over *all* array types, even one-dimensional and multidimensional arrays. This is occasionally useful, such as when you're writing a generic array printer.

Table 18-9 summarizes the primary general types defined in the .NET Framework.

Table 18-9. *General types in the .NET Framework*

Function	Description
`System.Array`	General type of all array values.
`System.Delegate`	General type of all delegates.
`System.Enum`	General type of all enum values.
`System.Exception`	General type of all exception values.
`System.Collections.IEnumerable`	General type of all sequence values. This is the nongeneric version of the F# type `seq<'T>`, and all sequence and collection values are compatible with this type.
`System.IComparable`	General type of all comparable types.

Function	Description
`System.IDisposable`	General type of all explicitly reclaimable resources.
`System.IFormattable`	General type of all types supporting .NET formatting.
`System.Object`	General type of all values.
`System.Type`	Runtime representation of .NET types.
`System.ValueType`	General type of all value types.

Using Microsoft.FSharp.Reflection

In Chapter 17, the schema compiler used functions from the `Microsoft.FSharp.Reflection` namespace. This namespace is a relatively thin extension of the `System.Reflection` namespace. It offers an interesting set of techniques for creating and analyzing F# values and types in ways that are somewhat simpler than those offered by the `System.Reflection` namespace. These operations are also designed to be used in precompilation phases to amortize costs associated with reflective programming.

Table 18-10 summarizes the two types in this namespace

Table 18-10. *Some operations in the* `Microsoft.FSharp.Reflection` *namespace*

Class and Static Members	Description
`Microsoft.FSharp.Reflection.FSharpType`	Operations to analyze F# types
`Microsoft.FSharp.Reflection.FSharpValue`	Operations to analyze F# values

Some Other .NET Types You May Encounter

When .NET was first designed, the .NET type system didn't include generics or a general notion of a function type as used by F#. Instead of functions, .NET uses delegates, which you can think of as named function types (that is, each kind of function type is given a different name).

This means that you often encounter delegate types when using .NET libraries from F#. Since .NET 2.0, some of these are even generic, giving an approximation of the simple and unified view of function types used by F#. Every .NET delegate type has a corresponding F# function type. For example, the F# function type for the .NET delegate type `System.Windows.Forms.PaintEventHandler` is `obj -> System.Windows.Forms.PaintEventArgs -> unit`. You can figure out this type by looking at the signature for the `Invoke` method of the given delegate type.

.NET also comes with definitions for some generic delegate types. F# tends to use function types instead of these, so you don't see them often in your coding. However, Table 18-11 shows these delegate types just in case you meet them.

Table 18-11. *Delegate types encountered occasionally in F# coding*

Function	F# Function Type	Description
`System.Action<'T>`	`'T -> unit`	Used for imperative actions.
`System.AsyncCallback`	`System.IAsyncResult -> unit`	Used for callbacks when asynchronous actions complete.
`System.Converter<'T,'U>`	`'T -> 'U`	Used to convert between values.
`System.Comparison<'T>`	`'T -> 'T -> int`	Used to compare values.

Function	F# Function Type	Description
`System.EventHandler<'T>`	`obj -> 'T -> unit`	Used as a generic event-handler type.
`System.Func<'T,'U>`	`'T -> 'U`	A .NET 3.5 LINQ function delegate. Further arity-overloaded types exist accepting additional arguments: for example, `System.Func<'T,'U,'V>`, `System.Func<'T,'U,'V,'W>`.
`System.Predicate<'T>`	`'T -> bool`	Used to test a condition.

Under the Hood: Interoperating with C# and other .NET Languages

Libraries and binary components provide a common way to reuse software; even the simplest C program is linked to the standard C runtime to benefit from core functions, such as memory management and I/O. Modern programs depend on a large number of libraries that are shipped in binary form, and only some of them are written in the same language as the program. Libraries can be linked statically during compilation into the executable, or they can be loaded dynamically during program execution. Dynamic linking has become significantly common to help share code (dynamic libraries can be linked by different programs and shared among them) and adapt program behavior while executing.

Interoperability among binaries compiled by different compilers, even of the same language, can be a nightmare. One of the goals of the .NET initiative was to ease this issue by introducing the Common Language Runtime (CLR), which is targeted by different compilers and different languages to help interoperability among software developed in those languages.

The Common Language Runtime

The CLR is a runtime designed to run programs compiled for the .NET platform; in addition to Microsoft. NET, CLR has been implemented by the open source project Mono. The binary format of these programs differs from the traditional one adopted by executables; Microsoft terminology uses *managed* for the first class of programs and *unmanaged* otherwise (see Figure 18-1).

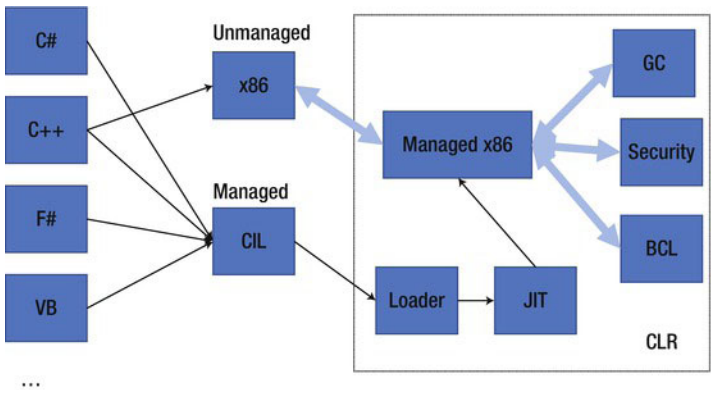


Figure 18-1. *Compilation scheme for managed and unmanaged code*

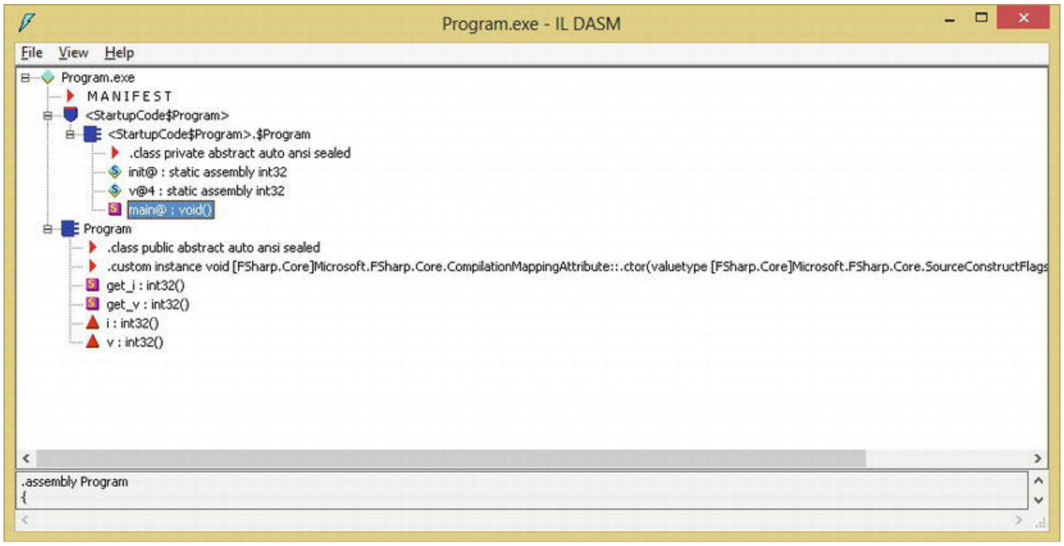
A DEEPER LOOK INSIDE .NET EXECUTABLES

Programs for the .NET platform are distributed in a form that is executed by the CLR. Binaries are expressed in an intermediate language that is compiled incrementally by the Just-In-Time (JIT) compiler during program execution. A .NET assembly, in the form of a `.dll` or an `.exe` file, contains the definition of a set of types and the definition of the method bodies, and the additional data describing the structure of the code in the intermediate language form are known as *metadata*. The intermediate language is used to define method bodies based on a stack-based machine, with operations performed by loading values on a stack of operands and then invoking methods or operators.

Consider the following simple F# program in the `Program.fs` source file:

```
open System

let i = 2g

Console.WriteLine("Input a number:")

let v = Int32.Parse(Console.ReadLine())

Console.WriteLine(i * v)
```

The F# compiler generates an executable that can be disassembled using the `ildasm.exe` tool distributed with the .NET Framework. The following screenshot shows the structure of the generated assembly. Because everything in the CLR is defined in terms of types, the F# compiler must introduce the class `$Program$Main` in the `<StartupCode$`*`applicationname`*`>` namespace. In this class, the definition of the `main@` static method is the entry point for the execution of the program. This method contains the intermediate language corresponding to the example F# program. The F# compiler generates several elements that aren't defined in the program, whose goal is to preserve the semantics of the F# program in the intermediate language.



```
Program.exe
    MANIFEST
    <StartupCode$Program>
        <StartupCode$Program>.$Program
            .class private abstract auto ansi sealed
            init@ : static assembly int32
            v@4 : static assembly int32
            main@ : void()
    Program
        .class public abstract auto ansi sealed
        .custom instance void [FSharp.Core]Microsoft.FSharp.Core.CompilationMappingAttribute::.ctor(valuetype [FSharp.Core]Microsoft.FSharp.Core.SourceConstructFlags
        get_i : int32()
        get_v : int32()
        i : int32()
        v : int32()

.assembly Program
{
```

If you open the `main@` method, you find the following code, which is annotated here with the corresponding F# statements:

```
.method public static void  main@() cil managed
{
  .entrypoint
  // Code size       38 (0x26)
  .maxstack  4

   // Console.WriteLine("Input a number:")
  IL_0000:  ldstr      "Input a number:"
  IL_0005:  call       void [mscorlib]System.Console::WriteLine(string)

  // let v = Int32.Parse(Console.ReadLine())
  IL_000a:  call       string [mscorlib]System.Console::ReadLine()
  IL_000f:  call       int32 [mscorlib]System.Int32::Parse(string)
  IL_0014:  stsfld     int32 '<StartupCode$ConsoleApplication1>'.$Program::v@4

  // Console.WriteLine(i * v) // Note that i is constant and its value has been inlined
  IL_0019:  ldc.i4.2
  IL_001a:  call       int32 Program::get_v()
  IL_001f:  mul
  IL_0020:  call       void [mscorlib]System.Console::WriteLine(int32)

// Exits
  IL_0025:  ret
} // end of method $Program$Main::main@
```

The `ldxxx` instructions are used to load values onto the operand stack of the abstract machine, and the `stxxx` instructions store values from that stack in locations (locals, arguments, or class fields). In this example, a static field is used for `v`, and the value of `i` is inlined using the `ldc` instruction. For method invocations, arguments are loaded on the stack, and a `call` operation is used to invoke the method.

The JIT compiler is responsible for generating the binary code that runs on the actual processor. The code generated by the JIT interacts with all the elements of the runtime, including external code loaded dynamically in the form of DLLs or COM components.

Because the F# compiler targets the CLR, its output is managed code, allowing compiled programs to interact directly with other programming languages targeting the .NET platform. Chapter 16 showed how to exploit this form of interoperability, demonstrating how to develop a graphic control in F# and use it in a C# application.

Memory Management at Runtime

Interoperability of F# programs with unmanaged code requires an understanding of the structure of the most important elements of a programming language's runtime. In particular, consider how program memory is organized at runtime. Memory used by a program is generally classified in three classes depending on the way it's handled:

- *Static memory*: Allocated for the entire lifetime of the program
- *Automatic memory:* Allocated and freed automatically when functions or methods are executed

- *Dynamic memory:* Explicitly allocated by the program, and freed explicitly or by an automatic program called the *garbage collector*

As a rule of thumb, top-level variables and static fields belong to the first class, function arguments and local variables belong to the second class, and memory explicitly allocated using the `new` operator belongs to the last class. The code generated by the JIT compiler uses different data structures to manage memory and automatically interacts with the operating system to request and release memory during program execution.

Each execution thread has a stack where local variables and arguments are allocated when a function or method is invoked (see Figure 18-2). A stack is used, because it naturally follows the execution flow of method and function calls. The topmost record contains data about the currently executing function; below that is the record of the caller of the function, which sits on top of another record of its caller, and so on. These *activation records* are memory blocks used to hold the memory required during the execution of the function and are naturally freed at the end of its execution by popping the record off the stack. The stack data structure is used to implement the automatic memory of the program, and because different threads execute different functions at the same time, a separate stack is assigned to each of them.

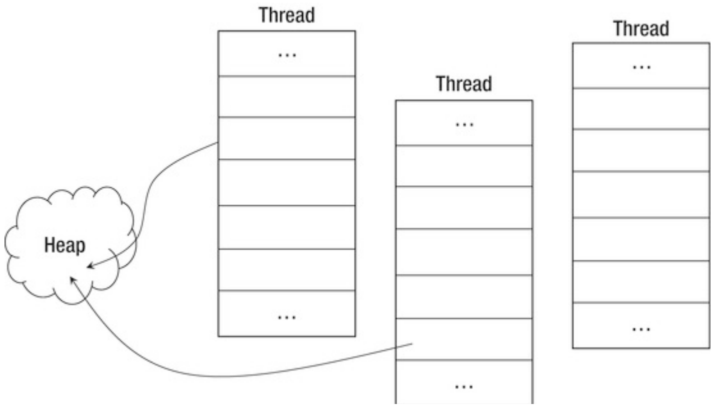


Figure 18-2. *Memory organization of a running CLR program*

Dynamic memory is allocated in the *heap*, which is a data structure where data resides for an amount of time not directly related to the events of program execution. The memory is explicitly allocated by the program, and it's deallocated either explicitly or automatically, depending on the strategy adopted by the runtime to manage the heap. In the CLR, the heap is managed by a *garbage collector*, which is a program that tracks memory usage and reclaims memory that is no longer used by the program. Data in the heap is always referenced from the stack—or other known areas, such as static memory—either directly or indirectly. The garbage collector can deduce the memory potentially reachable by program execution in the future, and the remaining memory can be collected. During garbage collection, the running program may be suspended, because the collector may need to manipulate objects needed by its execution. In particular, a garbage collector may adopt a strategy called *copy collection* that can move objects in memory; during this process, the references may be inconsistent. To avoid dangling references, the memory model adopted by the CLR requires that methods access the heap through object references stored on the stack; objects in the heap are forbidden to reference data on the stack.

Data structures are the essential tool provided by programming languages to group values. A data structure is rendered as a contiguous area of memory in which the constituents are available at a given offset from the beginning of the memory. The actual layout of an object is determined by the compiler (or by the interpreter for interpreted languages) and is usually irrelevant to the program because the programming language provides operators to access fields without having to explicitly access the memory. System programming, however, often requires explicit manipulation of memory, and programming languages such as C let you control the in-memory layout of data structures. The C specification, for instance, defines that fields of a structure are laid out sequentially, although the compiler is allowed to insert extra space between them. Padding is used to align fields at word boundaries of the particular architecture in order to optimize the access to the fields of the structure. Thus, the same data structure in a program may lead to different memory layouts depending on the strategy of the compiler or the runtime, even in a language such as C , in which field ordering is well defined. By default, the CLR lays out structures in memory without any constraint, which gives you the freedom of optimizing memory usage on a particular architecture, although it may introduce interoperability issues if a portion of memory must be shared among the runtimes of different languages.[1]

Interoperability among different programming languages revolves mostly around memory layouts, because the execution of a compiled routine is a jump to a memory address. But routines access memory explicitly, expecting that data are organized in a certain way. The rest of this chapter discusses the mechanisms used by the CLR to interoperate with external code in the form of DLLs or COM components.

COM Interoperability

Component Object Model (COM) is a technology that Microsoft introduced in the 1990s to support interoperability among different programs that were possibly developed by different vendors. The Object Linking and Embedding (OLE) technology that lets you embed arbitrary content in a Microsoft Word document, for instance, relies on this infrastructure. COM is a binary standard that allows code written in different languages to interoperate, assuming that the programming language supports this infrastructure. Most of the Windows operating system and its applications are based on COM components.

The CLR was initially conceived of as an essential tool to develop COM components, because COM was the key technology at the end of 1990s. It's no surprise that the Microsoft implementation of CLR interoperates easily and efficiently with the COM infrastructure.

This section briefly reviews how COM components can be consumed from F# (and vice versa) and how F# components can be exposed as COM components.

Calling COM Components from F#

COM components can be easily consumed from F# programs, and the opposite is also possible, by exposing .NET objects as COM components. The following example is based on the Windows Scripting Host and uses F# and `fsi.exe`:

```
> open System;;
> let o = Activator.CreateInstance(Type.GetTypeFromProgID("Word.Application"));;
val o : obj

> let t = o.GetType();;
val t : Type = Microsoft.Office.Interop.Word.ApplicationClass

> t.GetProperty("Visible").SetValue(o, (true :> Object), null);;
val it : unit = ()
```

[1]Languages targeting .NET aren't affected by these interoperability issues because they share the same CLR runtime.

```
> let m = t.GetMethod("Quit");;
val m : Reflection.MethodInfo =
  Void Quit(System.Object ByRef, System.Object ByRef, System.Object ByRef)

> m.GetParameters().Length;;
val it : int = 3

> m.GetParameters();;
val it : Reflection.ParameterInfo [] =
  [|System.Object& SaveChanges
      {Attributes = In, Optional, HasFieldMarshal;
       CustomAttributes = seq
                           [[System.Runtime.InteropServices.InAttribute()];
                            [System.Runtime.InteropServices.OptionalAttribute()];
                            [System.Runtime.InteropServices.MarshalAsAttribute((System.
Runtime.InteropServices.UnmanagedType)27, ArraySubType = 0, SizeParamIndex = 0, SizeConst = 0,
IidParameterIndex = 0, SafeArraySubType = 0)]];
       DefaultValue = System.Reflection.Missing;
       HasDefaultValue = false;
       IsIn = true;
       IsLcid = false;
       IsOptional = true;
       IsOut = false;
       IsRetval = false;
       Member = Void Quit(System.Object ByRef, System.Object ByRef, System.Object ByRef);
       MetadataToken = 134223584;
       Name = "SaveChanges";
       ParameterType = System.Object&;
       Position = 0;
       RawDefaultValue = System.Reflection.Missing;};
    ... more ... |]

> m.Invoke(o, [|null; null; null|]);;
```

val it : obj = null Because F# imposes type inference, you can't use the simple syntax provided by an interpreter. The compiler should know in advance the number and type of arguments of a method and the methods exposed by an object. Remember that even though fsi.exe allows you to interactively execute F# statements, it's still subject to the constraints of a compiled language. Because you're creating an instance of a COM component dynamically in this example, the compiler doesn't know anything about this component, so it can be typed as System.Object. To obtain the same behavior as an interpreted language, you must resort to .NET runtime's reflection support. Using the GetType method, you can obtain an object describing the type of the object o. Then, you can obtain a PropertyInfo object describing the Visible property, and you can invoke the SetValue method on it to show the Word main window. The SetValue method is generic; therefore, you have to cast the Boolean value to System.Object to comply with the method signature.

In a similar way, you can obtain an instance of the MethodInfo class describing the Quit method. Because a method has a signature, you ask for the parameters; there are three of them, and they're optional. You can invoke the Quit method by calling the Invoke method and passing the object target of the invocation and an array of arguments that you set to null, because arguments are optional.

■ **Note**: Although COM technology is still widely used for obtaining so-called automation, .NET is quietly entering the picture, and several COM components are implemented using the CLR. Whenever a reference to `mscoree.dll` appears in the `InprocServer32` registry key, the .NET runtime is used to deliver the COM services using the specified assembly. Through COM interfaces, native and .NET components can be composed seamlessly, leading to very complex interactions between managed and unmanaged worlds. Microsoft Word 2010, for instance, returns a .NET object instead of a COM wrapper, which provides access to Word services without the need for explicit use of reflection.

How can the runtime interact with COM components? The basic approach is based on the *COM callable wrapper* (CCW) and the *runtime callable wrapper* (RCW), as shown in Figure 18-3. The former is a chunk of memory dynamically generated with a layout compatible with the one expected from COM components, so that external programs—even legacy Visual Basic 6 applications—can access services implemented as managed components. The latter is more common and creates a .NET type dealing with the COM component, taking care of all the interoperability issues. It's worth noting that although the CCW can always be generated, because the .NET runtime has full knowledge about assemblies, the opposite isn't always possible. Without `IDispatch` or type libraries, there is no description of a COM component at runtime. Moreover, if a component uses custom marshalling, it can't be wrapped by an RCW. Fortunately, for the majority of COM components, it's possible to generate an RCW.

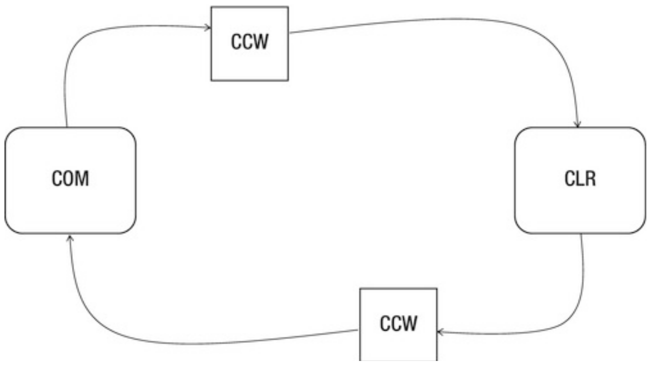


Figure 18-3. *The wrappers generated by the CLR to interact with COM components*

Programming patterns based on event-driven programming are widely adopted, and COM components have a programming pattern to implement callbacks based on the notion of a *sink*. The programming pattern is based on the delegate event model, and the sink is where a listener can register a COM interface that should be invoked by a component to notify an event. The Internet Explorer Web Browser COM component (implemented by `shdocvw.dll`), for instance, provides a number of events to notify its host about various events, such as loading a page or clicking a hyperlink. The RCW generated by the runtime exposes these events in the form of delegates and takes care of handling all the details required to perform the communication between managed and unmanaged code.

Although COM components can be accessed dynamically using .NET reflection, explicitly relying on the ability of the CLR to generate CCW and RCW, it's desirable to use a less verbose approach to COM interoperability. The .NET runtime ships with tools that allow you to generate RCW and CCW wrappers offline, which lets you use COM components as .NET classes and vice versa. These tools are:

- `tlbimp.exe`: This is a tool for generating an RCW of a COM component given its type library.
- `aximp.exe`: This is similar to `tlbimp.exe` and supports the generation of "ActiveX" COM components[2] that have graphical interfaces and that can be integrated with Windows Forms.
- `tlbexp.exe`: This generates a COM type library describing a .NET assembly. The CLR is loaded as a COM component and generates the appropriate CCW to make .NET types accessible as COM components.
- `regasm.exe`: This is similar to `tlbexp.exe`. It also performs the registration of the assembly as a COM component.

To better understand how COM components can be accessed from your F# programs and vice versa, let's consider two examples. In the first, you wrap the widely used Flash Player into a form interactively; in the second, you see how an F# object type can be consumed as if it were a COM component.

The Flash Player you're accustomed to using in everyday browsing is a COM control that is loaded by Internet Explorer using an `OBJECT` element in the HTML page (it's also a plug-in for other browsers, but here you're interested in the COM component). By using a search engine, you can easily find that an HTML element similar to the following is used to embed the player in Internet Explorer:

```
<OBJECT
    classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
    codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab"
    title="My movie" width="640" height="480">
    <param name="movie" value="MyMovie.swf" />
    <param name="quality" value="high" />
</OBJECT>
```

From this tag, you know that the CLSID of the Flash Player COM component is the one specified with the `classid` attribute of the `OBJECT` element. You can now look in the Windows registry under `HKEY_CLASSES_ROOT\CLSID` for the subkey corresponding to the CLSID of the Flash COM control. If you look at the subkeys, you notice that the `ProgID` of the component is `ShockwaveFlash.ShockwaveFlash`, and `InprocServer32` indicates that its location is `C:\Windows\system32\Macromed\Flash\Flash10d.ocx`. You can also find the GUID relative to the component type library—which, when investigated, shows that the type library is contained in the same OCX file.

■ **Note**: With a 64-bit version of Windows and the 32-bit version of the Flash Player, you should look for the key `CLSID` under `HKEY_CLASSES_ROOT\Wow6432Node`, which is where the 32-bit component's information is stored. In general, all references to the 32-bit code are stored separately from the 64-bit information. The loader tricks old 32-bit code into seeing different portions of the registry. In addition, executable files are stored under `%WinDir%\SysWow64` instead of `%WinDir%\system32`. Moreover, to wrap 32- or 64-bit components, you need the corresponding version of the .NET tool.

---

[2]ActiveX components are COM components implementing a well-defined set of interfaces. They have a graphical interface. Internet Explorer is well known for loading these components, but ActiveX can be loaded by any application using the COM infrastructure.

Because Flash Player is a COM control with a GUI, you can rely on aximp.exe rather than just tlbimp.exe to generate the RCW for the COM component (for 64-bit systems, use c:\Windows\SysWOW64 instead of c:\Windows\System32 and link the .ocx file contained in the Flash directory):

```
C:\> aximp c:\Windows\System32\Macromed\Flash\Flash32_11_3_370_17810d.ocx
Generated Assembly: C:\ShockwaveFlashObjects.dll
Generated Assembly: C:\AxShockwaveFlashObjects.dll
```

If you use ildasm.exe to analyze the structure of the generated assemblies, notice that the wrapper of the COM component is contained in ShockwaveFlashObjects.dll and is generated by the tlbimp.exe tool. The second assembly contains a Windows Forms host for COMM components and is configured to host the COM component, exposing the GUI features in terms of the elements of the Windows Forms framework.

You can test the Flash Player embedded in an interactive F# session:

```
#I @"c:\";;
--> Added 'c:\ ' to library include path

#r "AxShockwaveFlashObjects.dll";;
--> Referenced 'c:\AxShockwaveFlashObjects.dll'

> open AxShockwaveFlashObjects;;
> open System.Windows.Forms;;

> let f = new Form();;
val f : Form = System.Windows.Forms.Form, Text:

> let flash = new AxShockwaveFlash();;
val flash : AxShockwaveFlash = AxShockwaveFlashObjects.AxShockwaveFlash

> f.Show();;
> flash.Dock <- DockStyle.Fill;;
> f.Controls.Add(flash);;
> flash.LoadMovie(0, "http://laptop.org/img/meshDemo18.swf");;
```

First add to the include path of the fsi.exe directory containing the assemblies generated by aximp.exe using the #I directive, and then reference the AxShockwaveFlashObjects.dll assembly using the #r directive. The namespace AxShockwaveFlashObjects containing the AxShockwaveFlash class is opened; this is the managed class wrapping the COM control. You create an instance of the Flash Player that is now exposed as a Windows Forms control; then set the Dock property to DockStyle.Fill to let the control occupy the entire area of the form. Finally, add the control to the form.

When you're typing the commands into F# Interactive, it's possible to test the content of the form. When it first appears, a right-click on the client area is ignored. After the COM control is added to the form, the right-click displays the context menu of the Flash Player. You can now programmatically control the player by setting the properties and invoking its methods; the generated wrapper takes care of all the communications with the COM component.

The Running Object Table

Sometimes you need to obtain a reference to an out-of-process COM object that is already running. This is useful when you want to automate some task of an already-started application or reuse an object model without needing to start a process more than once. The easiest way to achieve this is through the `GetActiveObject` method featured by the `Marshal` class:

```
#r "EnvDTE80"
open System.Runtime.InteropServices

let appObj = Marshal.GetActiveObject("VisualStudio.DTE.11.0") :?> EnvDTE80.DTE2
printfn "%s" appObj.ActiveDocument.FullName.
```

In this example, you obtain a reference to one of the most important interfaces of Visual Studio's COM automation model. An interesting experiment is printing the name of the active document open in the editor and trying to run different instances of Visual Studio, opening different documents. The COM infrastructure connects to one instance of the COM server without being able to specify a particular one.

You can find a specific instance by accessing a system-wide data structure called the Running Object Table (ROT), which provides a list of running COM servers. Because the name of a running server must be unique within the ROT, many servers mangle the PID with the COM ProgID, so it's possible to connect to a given instance. This is the case for Visual Studio. The following F# function connects to a specific Visual Studio instance:

```
#r "EnvDTE"

open System.Runtime.InteropServices
open System.Runtime.InteropServices.ComTypes

[<DllImport("ole32.dll")>]
extern int internal GetRunningObjectTable(uint32 reserved, IRunningObjectTable& pprot)

[<DllImport("ole32.dll")>]
extern int internal CreateBindCtx(uint32 reserved, IBindCtx& pctx)

let FetchVSDTE (pid : int) =
    let mutable prot : IRunningObjectTable = null
    let mutable pmonkenum : IEnumMoniker = null
    let (monikers : IMoniker[]) =  Array.create 1 null
    let pfeteched = System.IntPtr.Zero
    let mutable (ret  :obj) = null
    let endpid = sprintf ":%d" pid

    try
        if (GetRunningObjectTable(0u, &prot) <> 0) || (prot = null) then
            failwith "Error opening the ROT"
        prot.EnumRunning(&pmonkenum)
        pmonkenum.Reset()
        while pmonkenum.Next(1, monikers, pfeteched) = 0 do
            let mutable (insname : string) = null
            let mutable (pctx : IBindCtx) = null
            CreateBindCtx(0u, &pctx) |> ignore
            (monikers.[0]).GetDisplayName(pctx, null, &insname);
            Marshal.ReleaseComObject(pctx) |> ignore
            if insname.StartsWith("!VisualStudio.DTE") && insname.EndsWith(endpid) then
                prot.GetObject(monikers.[0], &ret) |> ignore
    finally
        if prot <> null then Marshal.ReleaseComObject(prot) |> ignore
        if pmonkenum <> null then Marshal.ReleaseComObject(pmonkenum) |> ignore
    (ret :?> EnvDTE.DTE)
```

You use two `PInvoke` declarations to import functions from the `ole.dll` COM library in which the ROT is defined. After you get a reference to the table, you perform an enumeration of its elements, retrieving the display name and looking for any Visual Studio DTE with a specific PID at its end. The `GetObject` method is used to finally connect to the desired interface.

This example shows the flexible control .NET and F# can provide over COM infrastructure. The ability to access specific COM object instances is widely used on the server side to implement services made available through Web pages.

Interoperating with C and C++ with PInvoke

Through the CLI, F# implements a standard mechanism for interoperating with C and C++ code that is called "Platform Invoke", normally known as "PInvoke". This is a core feature of the standard available on all CLI implementations, including Mono.

The basic model underlying PInvoke is based on loading DLLs into the program, which allows managed code to invoke functions exported from C and C++. DLLs don't provide information other than the entry point location of a function; this isn't enough to perform the invocation unless additional information is made available to the runtime.

The invocation of a function requires:

- The address of the code in memory
- The calling convention, which is how parameters, return values, and other information are passed through the stack to the function
- Marshalling of values and pointers so that the different runtime support can operate consistently on the same values

You obtain the address of the entry point using a system call that returns the pointer to the function given a string. You must provide the remaining information to instruct the CLR about how the function pointer should be used.

CALLING CONVENTIONS

Function and method calls (a method call is similar to a function call but with an additional pointer referring to the object passed to the method) are performed by using a shared stack between the caller and the callee. An activation record is pushed onto the stack when the function is called, and memory is allocated for arguments, the return value, and local variables. Additional information—such as information about exception handling and the return address when the execution of the function terminates— is also stored in the activation record,

The physical structure of the activation record is established by the compiler (or by the JIT in the case of the CLR), and this knowledge must be shared between the caller and the called function. When the binary code is generated by a compiler, this isn't an issue, but when code generated by different compilers must interact, it may become a significant issue. Although each compiler may adopt a different convention, the need to perform system calls requires that the calling convention adopted by the operating system is implemented, and it's often used to interact with different runtimes. Another popular approach is to support the calling convention adopted by C compilers, because it's widely used and has become a fairly universal language for interoperability. Note that although many operating systems are implemented in C, the libraries providing system calls may adopt different calling conventions. This is the case with Microsoft Windows: the operating system adopts the *stdcall* calling convention rather than *cdecl*, which is the C calling convention.

A significant dimension in the arena of possible calling conventions is the responsibility for removing the activation record from the thread stack. At first glance, it may seem obvious that before returning, the called function resets the stack pointer to the previous state. This isn't the case for programming languages such as C that allow functions with a variable number of arguments, such as `printf`. When variable arguments are allowed, the caller knows the exact size of the activation record; therefore, it's the caller's responsibility to free the stack at the end of the function call. Apart from being consistent with the chosen convention, there may seem to be little difference between the two choices, but if the caller is responsible for cleaning the stack, each function invocation requires more instructions, which leads to larger executables. For this reason, Windows uses the stdcall calling convention instead of the C calling convention. It's important to notice that the CLR uses an array of objects to pass a variable number of arguments, which is very different from the variable arguments of C: the method receives a single pointer to the array that resides in the heap.

It's important to note that if the memory layout of the activation record is compatible, as it is in Windows, using the cdecl convention instead of the stdcall convention leads to a subtle memory leak. If the runtime assumes the stdcall convention (which is the default), and the callee assumes the cdecl convention, the arguments pushed on the stack aren't freed, and at each invocation, the height of the stack grows until a stack overflow happens.

The CLR supports a number of calling conventions. The two most important are stdcall and cdecl. Other implementations of the runtime may provide additional conventions to the user. In the PInvoke design, nothing restricts the supported conventions to these two (and in fact the runtime uses the fcall convention to invoke services provided by the runtime from managed code).

The additional information required to perform the function call is provided by custom attributes that are used to decorate a function prototype and inform the runtime about the signature of the exported function.

Getting Started with PInvoke

This section starts with a simple example of a DLL developed using C++, to which you add code during your experiments using PInvoke. The `CInteropDLL.h` header file declares a macro defining the decorations associated with each exported function:

```
#define CINTEROPDLL_API __declspec(dllexport)
extern "C" {
void CINTEROPDLL_API HelloWorld();
}
```

The `__declspec` directive is specific to the Microsoft Visual C++ compiler. Other compilers may provide different ways to indicate the functions that must be exported when compiling a DLL.

The first function is `HelloWorld`; its definition is as expected:

```
void CINTEROPDLL_API HelloWorld()
{
    printf("Hello C world invoked by F#!\n");
}
```

Say you now want to invoke the HelloWorld function from an F# program. You have to define the prototype of the function and inform the runtime how to access the DLL and the other information needed to perform the invocation. The program performing the invocation is:

```
open System.Runtime.InteropServices

module CInterop =
    [<DllImport("CInteropDLL", CallingConvention = CallingConvention.Cdecl)>]
    extern void HelloWorld()

CInterop.HelloWorld()
```

The extern keyword informs the compiler that the function definition is external to the program and must be accessed through the PInvoke interface. A C-style prototype definition follows the keyword, and the whole declaration is annotated with a custom attribute defined in the System.Runtime.InteropServices namespace. The F# compiler adopts C-style syntax for extern prototypes, including argument types (as you see later), because C headers and prototypes are widely used; this choice helps in the PInvoke definition. The DllImport custom attribute provides the information needed to perform the invocation. The first argument is the name of the DLL containing the function; the remaining option specifies the calling convention chosen to make the call. Because you don't specify otherwise, the runtime assumes that the name of the F# function is the same as the name of the entry point in the DLL. You can override this behavior using the EntryPoint parameter in the DllImport attribute.

It's important to note the declarative approach of the PInvoke interface. No code is involved in accessing external functions. The runtime interprets metadata in order to automatically interoperate with native code contained in a DLL. This is a different approach from the one adopted by different virtual machines, such as the Java virtual machine. The Java Native Interface (JNI) requires that you define a layer of code using types of the virtual machine and invoke the native code.

PInvoke requires high privileges in order to execute native code, because the activation record of the native function is allocated on the same stack that contains the activation records of managed functions and methods. Moreover, as discussed shortly, it's also possible to have the native code invoking a delegate marshalled as a function pointer, allowing stacks with native and managed activation records to be interleaved.

The HelloWorld function is a simple case, because the function doesn't have input arguments and doesn't return any value. Consider this function with input arguments and a return value:

```
int CINTEROPDLL_API Sum(int i, int j)
{
    return i + j;
}
```

Invoking the Sum function requires integer values to be marshalled to the native code and the value returned to managed code. Simple types, such as integers, are easy to marshal, because they're usually passed by value and use types of the underlying architecture. The F# program using the Sum function is:

```
module CInterop =
    [<DllImport("CInteropDLL", CallingConvention = CallingConvention.Cdecl)>]
    extern int Sum(int i, int j)

printf "Sum(1, 1) = %d\n" (CInterop.Sum(1, 1));
```

Parameter passing assumes the same semantics of the CLR, and parameters are passed by value for value types and by the value of the reference for reference types. Again, you use the custom attribute to specify the calling convention for the invocation.

Mapping C Data Structures to F# Code

Let's first cover what happens when structured data are marshalled by the CLR in the case of nontrivial argument types. Here, you see the SumC function responsible for adding two complex numbers defined by the Complex C data structure:

```
typedef struct _Complex {
    double re;
    double im;
} Complex;

Complex CINTEROPDLL_API SumC(Complex c1, Complex c2)
{
    Complex ret;
    ret.re = c1.re + c2.re;
    ret.im = c1.im + c2.im;
    return ret;
}
```

To invoke this function from F#, you must define a data structure in F# corresponding to the Complex C structure. If the memory layout of an instance of the F# structure is the same as that of the corresponding C structure, values can be shared between the two languages. How can you control the memory layout of a managed data structure? Fortunately, the PInvoke specification helps with custom attributes that let you specify the memory layout of data structures. The StructLayout custom attribute indicates the strategy adopted by the runtime to lay out fields of the data structure. By default, the runtime adopts its own strategy in an attempt to optimize the size of the structure, keeping fields aligned to the machine world in order to ensure fast access to the structure's fields. The C standard ensures that fields are laid out in memory sequentially in the order they appear in the structure definition; other languages may use different strategies. Using an appropriate argument, you can indicate that a C-like sequential layout strategy should be adopted. It's also possible to provide an explicit layout for the structure, indicating the offset in memory for each field of the structure. This example uses the sequential layout for the Complex value type:

```
module CInterop =
    [<Struct; StructLayout(LayoutKind.Sequential)>]
    type Complex =
        val mutable re : double
        val mutable im : double

        new(r, i) = {re = r; im = i}

    [<DllImport("CInteropDLL", CallingConvention = CallingConvention.Cdecl)>]
    extern Complex SumC(Complex c1, Complex c2)

let c1 = CInterop.Complex(1.0, 0.0)
let c2 = CInterop.Complex(0.0, 1.0)
let mutable c3 = CInterop.SumC(c1, c2)
printf "c3 = SumC(c1, c2) = %f + %fi\n" c3.re c3.im
```

The `SumC` prototype refers to the F# `Complex` value type. But because the layout of the structure in memory is the same as the corresponding C structure, the runtime passes the bits that are consistent with those expected by the C code.

Marshalling Parameters to and from C

A critical aspect of dealing with PInvoke is ensuring that values are marshalled correctly between managed and native code, and vice versa. A structure's memory layout doesn't depend only on the order of the fields. Compilers often introduce padding to align fields to memory addresses so that access to fields requires fewer memory operations, because CPUs load data into registers with the same strategy. Padding may speed up access to the data structure, but it introduces inefficiencies in memory usage: there may be gaps in the structures, leading to allocated but unused memory.

Consider, for instance, the C structure:

```
struct Foo {
    int i;
    char c;
    short s;
};
```

Depending on the compiler decision, it may occupy from 8 to 12 bytes on a 32-bit architecture. The most compact version of the structure uses the first 4 bytes for `i`, a single byte for `c`, and 2 more bytes for `s`. If the compiler aligns fields to addresses that are multiples of 4, then the integer `i` occupies the first slot, 4 more bytes are allocated for `c` (although only one is used), and the same happens for `s`.

Padding is a common practice in C programs; because it may affect performance and memory usage, directives instruct the compiler about padding. It's possible to have data structures with different padding strategies running within the same program.

The first step you face when using PInvoke to access native code is finding the definition of data structures, including information about padding. Then, you can annotate F# structures to have the same layout as the native ones, and the CLR can automate the marshalling of data. You can pass parameters by reference; thus, the C code may access the memory managed by the runtime, and errors in memory layout may result in corrupted memory. For this reason, keep PInvoke code to a minimum and verify it accurately to ensure that the execution state of the virtual machine is preserved. The declarative nature of the interface is a great help in this respect, because you must check declarations and not interop code.

Not all the values are marshalled as is to native code; some may require additional work from the runtime. Strings, for instance, have different memory representations between native and managed code. C strings are arrays of bytes that are null terminated, whereas runtime strings are .NET objects with a different layout. Also, function pointers are mediated by the runtime: the calling convention adopted by the CLR isn't compatible with external conventions, so code stubs are generated that can be called by native code from managed code, and vice versa.

In the `SumC` example, arguments are passed by value, but native code often requires data structures to be passed by reference to avoid the cost of copying the entire structure and passing only a pointer to the native data. The `ZeroC` function resets a complex number whose pointer is passed as an argument:

```
void CINTEROPDLL_API ZeroC(Complex* c)
{
    c->re = 0;
    c->im = 0;
}
```

The F# declaration for the function is the same as the C prototype:

```
[<DllImport("CInteropDLL", CallingConvention = CallingConvention.Cdecl)>]
extern void ZeroC(Complex* c)
```

Now you need a way to obtain a pointer given a value of type `Complex` in F#. You can use the `&&` operator to indicate a pass by reference; this results in passing the pointer to the structure expected by the C function:

```
let mutable c4 = CInterop.SumC(c1, c2)
printf "c4 = SumC(c1, c2) = %f + %fi\n" c4.re c4.im

CInterop.ZeroC(&&c4)
printf "c4 = %f + %fi\n" c4.re c4.im
```

In C and C++, the notion of objects (or struct instances) and the classes of memory are orthogonal: an object can be allocated on the stack or on the heap and share the same declaration. In .NET, this isn't the case; objects are instances of classes and are allocated on the heap, and value types are stored in the stack or wrapped into objects in the heap.

Can you pass objects to native functions through PInvoke? The main issue with objects is that the heap is managed by the garbage collector, and one possible strategy for garbage collection is *copy collection* (a technique that moves objects in the heap when a collection occurs). Thus, the base address in memory of an object may change over time, which can be a serious problem if the pointer to the object has been marshalled to a native function through a PInvoke invocation. The CLR provides an operation called *pinning* that prevents an object from moving during garbage collection. Pinned pointers are assigned to local variables, and pinning is released when the function performing the pinning exits. It's important to understand the scope of pinning: if the native code stores the pointer somewhere before returning, the pointer may become invalid but still usable from native code.

Now, let's define an object type for `Complex` and marshal F# objects to a C function. The goal is to marshal the F# object to the `ZeroC` function. In this case, you can't use the pass-by-reference operator, and you must define everything so that the type checker is happy. You can define another function that refers to `ZeroC` but with a different signature involving `ObjComplex`, which is an object type similar to the `Complex` value type. The `EntryPoint` parameter maps the F# function onto the same `ZeroC` C function, although in this case, the argument is of type `ObjComplex` rather than `Complex`:

```
module CInterop =
    [<StructLayout(LayoutKind.Sequential)>]
    type ObjComplex =
        val mutable re : double
        val mutable im : double

        new() = {re = 0.0; im = 0.0}
        new(r : double, i : double) = {re = r; im = i}

     [<DllImport("CInteropDLL", EntryPoint = "ZeroC",
                CallingConvention = CallingConvention.Cdecl)>]
    extern void ObjZeroC(ObjComplex c)

let oc = CInterop.ObjComplex(2.0, 1.0)
printf "oc = %f + %fi\n" oc.re oc.im
CInterop.ObjZeroC(oc)
printf "oc = %f + %fi\n" oc.re oc.im
```

In this case, the object reference is marshalled as a pointer to the C code, and you don't need the `&&` operator in order to call the function. The object is pinned to ensure that it doesn't move during the function call.

Marshalling Strings to and from C

PInvoke defines the default behavior for mapping common types used by the Win32 API. Table 18-12 shows the default conversions. Most of the mappings are natural, but note that there are several entries for strings. This is because strings are represented in different ways in programming language runtimes.

Table 18-12. *Default mapping for types of the Win32 API listed in* `Wtypes.h`

Unmanaged Type in `Wtypes.h`	Unmanaged C Type	Managed Class	Description
`HANDLE`	`void*`	`System.IntPtr`	32 bits on 32-bit Windows operating systems, 64 bits on 64-bit Windows operating systems
`BYTE`	`unsigned char`	`System.Byte`	8 bits
`SHORT`	`short`	`System.Int16`	16 bits
`WORD`	`unsigned short`	`System.UInt16`	16 bits
`INT`	`int`	`System.Int32`	32 bits
`UINT`	`unsigned int`	`System.UInt32`	32 bits
`LONG`	`long`	`System.Int32`	32 bits
`BOOL`	`long`	`System.Int32`	32 bits
`DWORD`	`unsigned long`	`System.UInt32`	32 bits
`ULONG`	`unsigned long`	`System.UInt32`	32 bits
`CHAR`	`char`	`System.Char`	Decorate with ANSI
`LPSTR`	`char*`	`System.String` or `System.Text.StringBuilder`	Decorate with ANSI
`LPCSTR`	`const char*`	`System.String` or `System.Text.StringBuilder`	Decorate with ANSI
`LPWSTR`	`wchar_t*`	`System.String` or `System.Text.StringBuilder`	Decorate with Unicode
`LPCWSTR`	`const wchar_t*`	`System.String` or `System.Text.StringBuilder`	Decorate with Unicode
`FLOAT`	`Float`	`System.Single`	32 bits
`DOUBLE`	`Double`	`System.Double`	64 bits

To see how strings are marshalled, start with a simple C function that echoes a string on the console:

```
void CINTEROPDLL_API echo(char* str)
{
    puts(str);
}
```

The corresponding F# PInvoke prototype is:

```
[<DllImport("CInteropDLL", CallingConvention = CallingConvention.Cdecl)>]
extern void echo(string s)
```

What happens when the F# function echo is invoked? The managed string is represented by an array of Unicode characters described by an object in the heap; the C function expects a pointer to an array of single-byte ANSI characters that are null terminated. The runtime is responsible for performing the conversion between the two formats, and it's performed by default when mapping a .NET string to an ANSI C string.

It's common to pass strings that are modified by C functions, yet .NET strings are immutable. For this reason, it's also possible to use a `System.Text.StringBuilder` object instead of a string. Instances of this class represent mutable strings and have an associated buffer containing the characters of the string. You can write a C function in the DLL that fills a string buffer given the size of the buffer:

```
void CINTEROPDLL_API sayhello(char* str, int sz)
{
    static char* data = "Hello from C code!";
    int len = min(sz, strlen(data));
    strncpy(str, data, len);
    str[len] = 0;
}
```

Because the function writes into the string buffer passed as an argument, use a `StringBuilder` rather than a string to ensure that the buffer has the appropriate room for the function to write. You can use the F# PInvoke prototype:

```
open System.Text
[<DllImport("CInteropDLL", CallingConvention = CallingConvention.Cdecl)>]
extern void sayhello(StringBuilder sb, int sz)
```

Because you have to indicate the size of the buffer, you can use a constructor of the `StringBuilder` class that allows you to do so:

```
let sb = new StringBuilder(50)
sayhello(sb, 50)
printf "%s\n" (sb.ToString())
```

You've used ANSI C strings so far, but this isn't the only type of string. Wide-character strings are becoming widely adopted and use 2 bytes to represent a single character; following the C tradition, the string is terminated by a null character. Consider a wide-character version of the `sayhello` function:

```
void CINTEROPDLL_API sayhellow(wchar_t* str, int sz)
{
    static wchar_t* data = L"Hello from C code Wide!";
    int len = min(sz, wcslen(data));
    wcsncpy(str, data, len);
    str[len] = 0;
}
```

How can you instruct the runtime that the `StringBuilder` should be marshalled as a wide-character string rather than an ANSI string? The declarative nature of PInvoke helps by providing a custom attribute to annotate function parameters of the prototype and to inform the CLR about the marshalling strategy to be adopted. The `sayhello` function is declared in F# as:

```
open System.Text
[<DllImport("CInteropDLL", CallingConvention = CallingConvention.Cdecl)>]
```

```
extern void sayhellow([<MarshalAs(UnmanagedType.LPWStr)>]StringBuilder sb, int sz)
```

In this case, the `MarshalAs` attribute indicates that the string should be marshalled as `LPWSTR` rather than `LPSTR`.

Passing Function Pointers to C

Another important data type that often should be passed to native code is a function pointer. Function pointers, which are widely used to implement callbacks, provide a simple form of functional programming; think, for instance, of a sort function that receives as input the pointer to the comparison function. Graphical toolkits have widely used this data type to implement event-driven programming, and they often have to pass a function that is invoked by another one.

PInvoke can marshal delegates as function pointers; again, the runtime is responsible for generating a suitable function pointer callable from native code. When the marshalled function pointer is invoked, a stub is called, and the activation record on the stack is rearranged to be compatible with the calling convention of the runtime. Then, the delegate function is invoked.

Although in principle the generated stub is responsible for implementing the calling convention adopted by the native code receiving the function pointer, the CLR supports only the stdcall calling convention for marshalling function pointers. Thus, the native code should adopt this calling convention when invoking the pointer. This restriction may cause problems, but in general, on the Windows platform, the stdcall calling convention is widely used.

The following C function uses a function pointer to apply a function to an array of integers:

```
typedef int (_stdcall *TRANSFORM_CALLBACK)(int);

void CINTEROPDLL_API transformArray(int* data, int count, TRANSFORM_CALLBACK fn)
{
    int i;
    for (i = 0; i < count; i++)
        data[i] = fn(data[i]);
}
```

The `TRANSFORM_CALLBACK` type definition defines the prototype of the function pointer you're interested in here: a function takes an integer as the input argument and returns an integer as a result. The `CALLBACK` macro is specific to the Microsoft Visual C++ compiler and expands to `__stdcall` in order to indicate that the function pointer, when invoked, should adopt the stdcall calling convention instead of the cdecl calling convention.

The `transformArray` function takes as input an array of integers with its length and the function to apply to its elements. You now have to define the F# prototype for this function by introducing a delegate type with the same signature as `TRANSFORM_CALLBACK`:

```
type Callback = delegate of int -> int

[<DllImport("CInteropDLL", CallingConvention = CallingConvention.Cdecl)>]
extern void transformArray(int[] data, int count, Callback transform)
```

Now you can increment all the elements of an array by using the C function:

```
let anyToString any = sprintf "%A" any
let data = [|1; 2; 3|]
printf "%s\n" (String.Join("; ", (Array.map anyToString data)))
```

```
transformArray(data, data.Length, new Callback(fun x -> x + 1))
printf "%s\n" (String.Join("; ", (Array.map anyToString data)))
```

PInvoke declarations are concise, but for data types such as function pointers, parameter passing can be expensive. In general, libraries assume that crossing the language boundary causes a loss of efficiency and callbacks are invoked at a price different from ordinary functions. In this respect, the example represents a situation in which the overhead of PInvoke is significant: a single call to `transformArray` causes a number of callbacks without performing any real computation into the native code.

PInvoke Memory Mapping

As a more complicated example of PInvoke usage, this section shows you how to benefit from memory mapping into F# programs. Memory mapping is a popular technique that allows a program to see a file (or a portion of a file) as if it were in memory. This provides an efficient way to access files, because the operating system uses the machinery of virtual memory to access files and significantly speed up data access on files. After proper initialization, which is covered in a moment, the program obtains a pointer into the memory, and access to that portion of memory appears the same as accessing data stored into the file.

You can use memory mapping to both read and write files. Every access performed to memory is reflected in the corresponding position in the file.

This is a typical sequence of system calls in order to map a file into memory:

1. A call to the `CreateFile` system call to open the file and obtain a handle to the file.
2. A call to the `CreateFileMapping` system call to create a mapped file object.
3. One or more calls to `MapViewOfFile` and `UnmapViewOfFile` to map and release portions of a file into memory. In a typical usage, the whole file is mapped at once in memory.
4. A call to `CloseHandle` to release the file.

The PInvoke interface to the required functions involves simple type mappings, as is usual for Win32 API functions. All the functions are in `kernel32.dll`, and the signature can be found in the Windows SDK. Listing 18-1 contains the definition of the F# wrapper for memory mapping.

The `SetLastError` parameter informs the runtime that the called function uses the Windows mechanism for error reporting and that the `GetLastError` function can be read in case of error; otherwise, the CLR ignores such a value. The `CharSet` parameter indicates the character set assumed, and it's used to distinguish between ANSI and Unicode characters; with `Auto`, you delegate the runtime to decide the appropriate version.

You can define the generic class `MemMap` that uses the functions to map a given file into memory. The goal of the class is to provide access to memory mapping in a system in which memory isn't directly accessible, because the runtime is responsible for its management. A natural programming abstraction to expose the memory to F# code is to provide an array-like interface in which the memory is seen as a homogeneous array of values.

Listing 18-1. *Exposing memory mapping in F#*

```
module MMap =
    open System
    open System.IO
    open System.Runtime.InteropServices
open Microsoft.FSharp.NativeInterop
open Printf

type HANDLE = nativeint
type ADDR = nativeint

[<DllImport("kernel32", SetLastError = true)>]
extern bool CloseHandle(HANDLE handler)

[<DllImport("kernel32", SetLastError = true, CharSet = CharSet.Auto)>]
extern HANDLE CreateFile(string lpFileName,
                         int dwDesiredAccess,
                         int dwShareMode,
                         HANDLE lpSecurityAttributes,
                         int dwCreationDisposition,
                         int dwFlagsAndAttributes,
                         HANDLE hTemplateFile)

[<DllImport("kernel32", SetLastError = true, CharSet = CharSet.Auto)>]
extern HANDLE CreateFileMapping(HANDLE hFile,
                                HANDLE lpAttributes,
                                int flProtect,
                                int dwMaximumSizeLow,
                                int dwMaximumSizeHigh,
                                string lpName)

[<DllImport("kernel32", SetLastError = true)>]
extern ADDR MapViewOfFile(HANDLE hFileMappingObject,
                          int dwDesiredAccess,
                          int dwFileOffsetHigh,
                          int dwFileOffsetLow,
                          int dwNumBytesToMap)

[<DllImport("kernel32", SetLastError = true, CharSet = CharSet.Auto)>]
extern HANDLE OpenFileMapping(int dwDesiredAccess,
                              bool bInheritHandle,
                              string lpName)

[<DllImport("kernel32", SetLastError = true)>]
extern bool UnmapViewOfFile(ADDR lpBaseAddress)

let INVALID_HANDLE = new IntPtr(-1)
let MAP_READ = 0x0004
let GENERIC_READ = 0x80000000
let NULL_HANDLE = IntPtr.Zero
let FILE_SHARE_NONE = 0x0000
let FILE_SHARE_READ = 0x0001
let FILE_SHARE_WRITE = 0x0002
let FILE_SHARE_READ_WRITE = 0x0003
let CREATE_ALWAYS = 0x0002
let OPEN_EXISTING = 0x0003
```

```
    let OPEN_ALWAYS = 0x0004
    let READONLY = 0x00000002

    type MemMap<'T when 'T : unmanaged> (fileName) =

        let ok =
            match typeof<'T> with
            | ty when ty = typeof<int> -> true
            | ty when ty = typeof<int32> -> true
            | ty when ty = typeof<byte> -> true
            | ty when ty = typeof<sbyte> -> true
            | ty when ty = typeof<int16> -> true
            | ty when ty = typeof<uint16> -> true
            | ty when ty = typeof<int64> -> true
            | ty when ty = typeof<uint64> -> true
            | _ -> false

        do if not ok then failwithf "the type %s is not a basic blittable type" ((typeof<'T>).
ToString())
        let hFile =
            CreateFile (fileName,
                        GENERIC_READ,
                        FILE_SHARE_READ_WRITE,
                        IntPtr.Zero, OPEN_EXISTING, 0, IntPtr.Zero  )
        do if (hFile.Equals(INVALID_HANDLE)) then
            Marshal.ThrowExceptionForHR(Marshal.GetHRForLastWin32Error());
        let hMap = CreateFileMapping (hFile, IntPtr.Zero, READONLY, 0, 0, null)
        do CloseHandle(hFile) |> ignore
        do if hMap.Equals(NULL_HANDLE) then
            Marshal.ThrowExceptionForHR(Marshal.GetHRForLastWin32Error());

        let start = MapViewOfFile (hMap, MAP_READ, 0, 0 ,0)

        do if (start.Equals(IntPtr.Zero)) then
            Marshal.ThrowExceptionForHR(Marshal.GetHRForLastWin32Error())

        member m.AddressOf(i : int) : 'T nativeptr =
            NativePtr.ofNativeInt(start + (nativeint i))

        member m.GetBaseAddress (i : int) : int -> 'T =
            NativePtr.get (m.AddressOf(i))

        member m.Item with get(i : int) : 'T = m.GetBaseAddress 0 i

        member m.Close() =
           UnmapViewOfFile(start) |> ignore;
           CloseHandle(hMap) |> ignore

        interface IDisposable with
            member m.Dispose() = m.Close()
```

The class exposes two properties: Item and Element. The former returns a function that allows access to data in the mapped file at a given offset using a function; the latter allows access to the mapped file at a given offset from the origin.

This example uses the MemMap class to read the first byte of a file:

```
let mm = new MMap.MemMap<byte>("somefile.txt")

printf "%A\n" (mm.[0])

mm.Close()
```

Memory mapping provides good examples of how easy it can be to expose native functionalities into the .NET runtime and how F# can be effective in this task. It's also a good example of the right way to use PInvoke to avoid calling PInvoked functions directly and build wrappers that encapsulate them. Verifiable code is one of the greatest benefits provided by virtual machines, and PInvoke signatures often lead to nonverifiable code that requires high execution privileges and risks corrupting the runtime's memory.

A good approach to reducing the amount of potentially unsafe code is to define assemblies that are responsible for accessing native code with PInvoke and that expose functionalities in a .NET verifiable approach. This way, the code that should be trusted by the user is smaller, and programs can have all the benefits provided by verified code.

Wrapper Generation and Limits of PInvoke

PInvoke is a flexible and customizable interface, and it's expressive enough to define prototypes for most libraries available. In some situations, however, it can be difficult to map directly the native interface into the corresponding signature. A significant example is function pointers embedded into structures, which are typical C programming patterns that approximate object-oriented programming. Here, the structure contains a number of pointers to functions that can be used as methods; but you must take care to pass the pointer to the structure as the first argument to simulate the this parameter. Oracle's Berkeley Database (BDB) is a popular database library that adopts this programming pattern. The core structure describing an open database is:

```
struct __db {
        /* ... */
        DB_ENV *dbenv;              /* Backing environment. */
        DBTYPE type;               /* DB access method type. */
        /* ... */
        int  (*close) __P((DB *, u_int32_t));
        int  (*cursor) __P((DB *, DB_TXN *, DBC **, u_int32_t));
        int  (*del) __P((DB *, DB_TXN *, DBT *, u_int32_t));
        // ...
}
```

The System.Runtime.InteropServices.Marshal class features the GetFunctionPointerForDelegate for obtaining a pointer to a function that invokes a given delegate. The caller of the function must guarantee that the delegate object will remain alive for the lifetime of the structure, because stubs generated by the runtime aren't moved by the garbage collector but can still be collected. Furthermore, callbacks must adopt the stdcall calling convention: if this isn't the case, the PInvoke interface can't interact with the library.

When PInvoke's expressivity isn't enough for wrapping a function call, you can still write an adapter library in a native language such as C. This is the approach followed by the BDB# library, in which an intermediate layer of code has been developed to make the interface to the library compatible with

PInvoke. The trick has been, in this case, to define a function for each database function, taking as input the pointer to the structure and performing the appropriate call:

```
DB *db;
// BDB call
db->close(db, 0);
// Wrapper call
db_close(db, 0);
```

The problem with wrappers is that they must be maintained manually when the signatures of the original library change. The intermediate adapter makes it more difficult to maintain the code's overall interoperability.

Many libraries have a linear interface that can be easily wrapped using PInvoke, and, of course, wrapper generators have been developed. At the moment, there are no wrapper generators for F#, but the C-like syntax for PInvoke declarations makes it easy to translate C# wrappers into F# code. An example of such a tool is SWIG, which is a multilanguage wrapper generator that reads C header files and generates interop code for a large number of programming languages, such as C#.

Summary

In this chapter, you saw how F# can interoperate with native code in the form of COM components and the standard Platform Invoke interface defined by the ECMA and ISO standards. Neither mechanism is dependent on F#, but the language exposes the appropriate abstractions built into the runtime. You studied how to consume COM components from F# programs and vice versa, and how to access DLLs through PInvoke.