CHAPTER 23

Assemblies, Application Configuration, and Process Management

So far in this book, you have learned how to write C# programs. As an enterprise programmer, you should know that writing application code is only one of the responsibilities of a programmer. The other responsibilities include assembling, deploying, configuring, and monitoring applications. This chapter introduces packaging and managing applications in C#. We look at assemblies, versioning, and application configuration and management—that is, process management.

In the Java world, applications are packaged not as EXE files but rather as class files that are loaded by the runtime `.java` executable. Many Java applications bundle class files in a `.java` archive file (or a `.jar` file). The `.jar` file is the preferred unit of deployment for applications and libraries in Java. Java products are usually shipped as nothing more than a group of `.jar` files. Several variations of the `.java` archive file (`.jar`) file are now used as units of deployment for the new breed of J2EE applications. J2EE Web applications are deployed as a **Web application archive** (`.war`) file. Similarly, EJB components are bundled in yet another variation of `.jar` files called **enterprise application archive** (`.ear`) files.

Java also provides an API for exploring these archive files and deployed units. The Java environment, however, does not come with a robust set of application monitoring and management tools. These tools tend to be closely tied to the operating system on which applications run, and Java's OS-neutral nature prohibits it from leveraging any OS-specific features in providing tight integration with any one platform. The JDK therefore ships with a minimal set of executables that can help you debug the performance and memory consumption of an application. Also, the API that helps in

monitoring memory allocation and garbage collection is not part of the core Java API. Nor does Java have built-in support for versioning of archive files.

The .NET Framework is a different story. It provides tools and APIs to ease the task of assembling, deploying, and versioning an application. In addition, it has APIs to interact with the Windows platform directly and to control the application's state through OS process management.

23.1 The .NET Assembly

An assembly is a reusable, self-describing, versionable deployment unit for types and resources. Because it is self-describing, it allows the .NET runtime to fully understand the application and enforce dependency and versioning rules. Unlike Java—in which both a stand-alone application and a component library are deployed as a `.jar` file—assemblies come in four types or formats:

1. `exe`. A console executable. The application must contain one entry point defined in the `Main` method.
2. Library. A library (DLL) that can be used by other assemblies.
3. Module. A nonexecutable collection of compiled code for use in other assemblies.
4. `winexe`. A graphical Windows executable. The assembly must contain one entry point.

An assembly consists of the following components:

- A manifest that contains the assembly metadata
- One or more modules
- Resource files, such as icons for Windows applications

Let's explore each of these in detail.

23.1.1 The Assembly Manifest

The assembly manifest consists of metadata describing the assembly and the types inside the assembly. The manifest is automatically created when the compiler creates an assembly.

A manifest consists of the following components:

- Information about the name, version, culture, and strong name, as well as general descriptive information (company name, product name, description, and copyright). The culture information indicates the language supported by the assembly. The version information consists of the major, minor, build, and revision numbers that the runtime uses to enforce versioning policy. The **strong name** is a globally unique name tag given to an assembly so that it can be identified in a shared context. The general descriptive information regarding copyright, product, and company name can easily be changed by the end user (shown later).
- A list of compiled code files that make up the assembly.
- Information about the types exported by the assembly and references to any other assemblies that are required to support the types of this assembly.

Microsoft Intermediate Language (MSIL) code in a portable executable (PE) file will not be executed if it does not have an associated assembly manifest.

23.1.2 Modules

Often, a complex application can consist of components written in different languages. Modules are designed to hold disparate components so that they can be developed in parallel in totally different languages and later integrated into one assembly. Modules contain one or more .NET types. Modules can contain types from more than one namespace, and modules can be compiled from one or more source files.

23.1.3 Resources

Resources let you separate text and images (static content for those working with Web applications) from the application logic, thereby simplifying the maintenance of the application.

23.2 Creating an Assembly

In its simplest form, an assembly is a single file with an `.exe`, `.dll`, or `.module` file extension. This file contains a single module and a manifest.

Let's create an assembly of type `exe`. You will need two source files: `HelloWorld.cs` and `HelloWorldDriver.cs`. The contents of the source files are shown in Listings 23.1 and 23.2.

Listing 23.1 `HelloWorld.cs` (C#)

```
public class HelloWorld {
  public override string ToString() {
    return "Hello World";
  }
}
```

Listing 23.2 `HelloWorldDriver.cs` (C#)

```
using System;
public class HelloWorldDriver {
  public static void Main(string[] args) {
    HelloWorld hw = new HelloWorld();
    Console.WriteLine(hw.ToString());
  }
}
```

Next, we use the following command to create an EXE assembly:

```
csc /target:exe /out:MyApp.exe *.cs
```

We specify the type of assembly using the `/target` option. The `/out` option allows us to name the EXE file.

Note that for the EXE file to be created, the compiled classes should have an entry point defined in a `Main()` method (Listing 23.2). After the EXE assembly is created, you can view its properties by right-clicking on the file name. Some of the properties displayed are the general information section of the manifest discussed earlier. Note that this information can be customized so that when you right-click on the EXE file you can view product, version, and copyright information that is customized to suit your needs.

To customize this information, we edit a special file called `AssemblyInfo.cs` and compile it with the rest of our source files. Here is a sample `AssemblyInfo.cs`:

```
using System.Reflection;
using System.Runtime.CompilerServices;
[assembly: AssemblyTitle("")]
[assembly: AssemblyDescription("This is test")]
[assembly: AssemblyCompany("Acme Inc")]
[assembly: AssemblyProduct("Acme Product")]
[assembly: AssemblyCopyright("Buyer beware")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
[assembly: AssemblyVersion("1.0.0.1")]
```

Next, we run the command again:

```
csc /target:exe /out:MyApp.exe *.cs
```

Upon running MyApp.exe we get the following output:

```
Hello World
```

The `MyApp.exe` file that is created now contains the extra information that was specified in the `AssemblyInfo.cs` file. You can view this information by clicking on the Version tab of the `MyApp.exe` properties box.

As mentioned earlier, assemblies do not have to be of type `.exe`. You can create a `.dll` or a `.module` module by using the appropriate `/target` command line option:

```
csc /target:library /out:MyApp.dll *.cs
csc /target:module /out:MyApp.module *.cs
```

In both cases, the metadata information specified in `AssemblyInfo.cs` is retained in the end product (`MyApp.dll` or `MyApp.module`).

Assemblies can consist of one or more modules. This is typical in complex applications in which the modules may be developed by different teams, perhaps in different languages. To illustrate this, we will need two more source files. Listings 23.3 and 23.4 show the `GoodByeWorld` and `GoodByeWorldDriver` classes.

Listing 23.3 `GoodByeWorld.cs` (C#)

```
public class GoodByeWorld {
  public override string ToString() {
```

```
    return "Good Bye World";
  }
}
```

Listing 23.4 `GoodByeWorldDriver.cs` (C#)

```
using System;
public class GoodByeWorldDriver {
  public static void Main(string[] args) {
    GoodByeWorld hw = new GoodByeWorld ();
    Console.WriteLine(hw.ToString());
  }
}
```

Next, we compile the `Hello*.cs` files into a `Hello.module` and compile the `GoodBye*.cs` files into a `GoodBye.module`:

```
csc /target:module /out:GoodBye.module Good*.cs
csc /target:module /out:Hello.module Hello*.cs
```

Then we combine the two modules into a `Greeting.exe` assembly:

```
al /target:exe /out:Greeting.exe /main:HelloWorldDriver.Main
Hello.module GoodBye.module
```

The assembly linker EXE file (`al.exe`) is used to link the two modules into a `Greeting.exe`. This concept of linking binary code to create an executable is similar to linking `.obj` files to create `.exe` files in traditional C/C++ environments.

Note that both the modules have an entry point defined, and so when we construct the `Greeting.exe` assembly we specify which `Main()` method we want to use as the entry point for that EXE. Running the `Greeting.exe` program now results in the following output:

```
Hello World
```

Now we run the command again, this time changing the value of the `/main` option:

```
al /target:exe /out:Greeting.exe /main:GoodByeWorldDriver.Main
Hello.module GoodBye.module
```

Running the `Greeting.exe` program now results in the following output:

```
Good Bye World
```

Assemblies can therefore be single file assemblies or multiple file assemblies (consisting of modules linked to each other).

The `Greeting.exe` assembly is now made up of two modules: `GoodBye.module` and `Hello.module`. Each module has an entry point defined, and the `Greeting.exe` assembly can adopt the entry point of any one of the modules. Depending on the entry point specified at the time the `Greeting.exe` assembly is created, the output of `Greeting.exe` can be either `Good Bye World` or `Hello World`.

23.3 Programmatic Access to Assemblies

Fortunately, .NET assemblies are not just black boxes. A .NET assembly is represented by the `Assembly` class of the `System.Reflection` namespace. You can use this class to gain programmatic access to the components of an assembly. Listing 23.5 shows how to load the assemblies created earlier and explore them.

Listing 23.5 Loading and Browsing an Assembly in C#

```
using System;
using System.Reflection;

public class AssemblyExplore {

  public static void Main(string[] args) {

    Assembly asm = Assembly.LoadFrom("C:\\BOOK\\
          code\\Chapter 23\\Greeting.exe");
    //Prints the properties of the assembly
    PrintProperties(asm);
```

```
    //Browse the modules of the assembly
    BrowseModules(asm);
  }

  private static void PrintProperties(Assembly asm) {
    Console.WriteLine("CodeBase "+asm.CodeBase);
    Console.WriteLine("Location "+asm.Location);
    Console.WriteLine("Global Assembly Cache
           "+asm.GlobalAssemblyCache);
    Console.WriteLine("EntryPoint "+asm.EntryPoint.ToString());
    Console.WriteLine("FullName "+asm.FullName);
  }

  private static void BrowseModules(Assembly asm) {
    foreach (Module m in asm.GetModules()) {
      BrowseModule (m);
    }
  }

  private static void BrowseModule(Module m) {
    Console.WriteLine("Module "  .Name);
    foreach (Type t in m.GetTypes()) {
      Console.WriteLine("\tType "+t.FullName);
      foreach (MethodInfo method in t.GetMethods()) {
        Console.WriteLine("\t\tMethod signature
          "+method.ToString());
      }
    }
  }
}
```

The output of Listing 23.5 is as follows:

```
CodeBase file:///C:/BOOK/code/Chapter 23/Greeting.exe
Location c:\book\code\chapter 23\greeting.exe
Global Assembly Cache False
EntryPoint Void __EntryPoint(System.String[])
FullName Greeting, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=null
Module greeting.exe
```

```
Module hello.module
Type HelloWorld
Method signature Int32 GetHashCode()
Method signature Boolean Equals(System.Object)
Method signature System.String ToString()
Method signature System.Type GetType()
Type HelloWorldDriver
Method signature Int32 GetHashCode()
Method signature Boolean Equals(System.Object)
Method signature System.String ToString()
Method signature Void Main(System.String[])
Method signature System.Type GetType()
Module goodbye.module
Type GoodByeWorld
Method signature Int32 GetHashCode()
Method signature Boolean Equals(System.Object)
Method signature System.String ToString()
Method signature System.Type GetType()
Type GoodByeWorldDriver
Method signature Int32 GetHashCode()
Method signature Boolean Equals(System.Object)
Method signature System.String ToString()
Method signature Void Main(System.String[])
Method signature System.Type GetType()
```

The output of Listing 23.5 should be self-explanatory based on what we did earlier while constructing the assembly. There are two modules that contain types. The `hello.module` file contains the `HelloWorld` and the `HelloWorldDriver` classes. The `goodbye.module` file contains the `GoodByeWorld` and `GoodByeWorldDriver` classes. The method signatures of the methods of these two classes are listed in the output generated by Listing 23.5.

Notice that `HelloWorld`, `HelloWorldDriver`, `GoodByeWorld`, and `GoodByeWorldDriver` are simple classes; so where did all these methods come from? Remember that all classes extend `System.Object`. What you see in the output are the `System.Object` methods inherited by these classes.

The `Assembly` class provides methods to retrieve information about the assembly. There are no methods provided to modify the assembly. Chapter

22 discusses the reflection emit mechanism, which can be used to create dynamic assemblies (see Listing 22.9).

So far, the assemblies we've created have been private assemblies—that is, specific to the application in question. Many times, applications depend on a shared assembly to function. The notion of sharing an assembly becomes more apparent when the assembly is a type library (a DLL file). Think of multiple EXE files using the same DLL (sound familiar?). A .NET shared assembly is more than just a static DLL that is referenced by the applications.

Consider a Java Web application deployed in a J2EE-compliant Web container. The container has library files that are infrastructure `.jar` files used by all applications deployed in that container. It makes little sense to deploy this common set of `.jar` files every time a new Web application is deployed. What is needed is a mechanism by which the container automatically knows to use the shared `.jar` file when an application makes a reference to it. Java handles this mechanism using a class-loader hierarchy. Because the application references the shared `.jar` file at runtime, it only makes sense that for the application to compile, it would need to reference that `.jar` file at compile time. The `.jar` file required at compile time need not be in the same location as the one required at runtime.

In .NET, a shared assembly is in many ways similar to that J2EE infrastructure `.jar` file. But .NET does not have the concept of class loaders or class paths, so how does the CLR know where to look when an application references a DLL it cannot find in its assembly? The **Global Assembly Cache** (GAC) is a directory (usually `C:\Windows\assembly` or `C:\WINNT\assembly`) where the CLR can locate such shared assemblies. Although it is not necessary to put the shared assembly in the GAC, it makes for easier deployment if shared assemblies are kept there. The .NET Framework provides a tool called the GAC tool (`gacutil.exe`) to install shared assemblies in the GAC.

Sharing an assembly means that the assembly should have a name that is globally unique. Such a name is called the **strong name** of the assembly. It consists of the name, version information, culture information, and a public key for cryptography. The `AssemblyInfo.cs` file can be used to modify the strong name components of an assembly. To create a public key, the .NET Framework provides a strong name tool called `sn.exe`, which creates a key file that can be referenced in the `AssemblyInfo.cs`. The following command creates a `Hello.snk` key file containing the cryptographic public key:

```
sn-k Hello.snk
```

Next, we create the `AssemblyInfo.cs` file with all the strong name components specified, as shown in boldface in Listing 23.6.

Listing 23.6 `AssemblyInfo` with Strong Name Components (C#)

```
using System.Reflection;
using System.Runtime.CompilerServices;

[assembly: AssemblyTitle("MyApp")]
[assembly: AssemblyDescription("A simple app")]
[assembly: AssemblyCompany("MyCompany.com")]
[assembly: AssemblyProduct("MyApp")]
[assembly: AssemblyCopyright("This is the property of
MyCompany.com")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyKeyFile("Hello.snk")]
```

If you run the following command, the `Hello.dll` is created with a strong name:

```
csc /target:library /out:Hello.dll HelloWorld.cs AssemblyInfo.cs
```

Now store it in the GAC using this command:

```
gacutil /i Hello.dll
```

When the DLL is installed in the GAC, an application is free to use it at runtime. But first, the application must use the DLL at compile time to compile itself. To do this, we copy the DLL from the GAC into the directory where you would compile your application. Let's compile `HelloWorld Driver` using this new DLL:

```
csc /target:exe /out:MyApp.exe HelloWorldDriver.cs
/reference:Hello.dll
```

Now remove `Hello.dll` from the local directory. When we run `MyApp` it should print the following as output:

```
Hello World
```

We have just created an EXE (`MyApp.exe`) that references a shared assembly (`Hello.dll`).

23.4 Versioning

In Section 23.3 you learned how to create an application that references a GAC assembly.

Sharing libraries among applications brings home the age-old problem regarding which version of the common library is being used by which application. Can applications that use two different versions of the same DLL coexist? How do you upgrade an application to use the new version of the DLL? Fortunately, .NET answers all these questions.

To illustrate versioning, we will create a new `Hello.dll` consisting of an `AssemblyInfo.cs` (Listing 23.7) with a different strong name and a different `HelloWorld.cs` (Listing 23.8).

Listing 23.7 Modified `AssemblyInfo` with Strong Name Components (C#)

```
using System.Reflection;
using System.Runtime.CompilerServices;

[assembly: AssemblyTitle("MyApp")]
[assembly: AssemblyDescription("A simple app")]
[assembly: AssemblyCompany("MyCompany.com")]
[assembly: AssemblyProduct("MyApp")]
[assembly: AssemblyCopyright("This is the property of
MyCompany.com")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
[assembly: AssemblyVersion("2.0.0.0")]
[assembly: AssemblyKeyFile("Hello.snk")]
```

Note that in Listing 23.7 we need only change the version number in order to change the strong name of an assembly. Next, we create a slightly different `HelloWorld` class.

Listing 23.8 A Slightly Different `HelloWorld` Class (C#)

```
public class HelloWorld {
  public override string ToString() {
  return "A slightly different Hello World";
  }
}
```

Next, we quickly create `Hello.dll`, register it in the GAC, and compile the `HelloWorldDriver` class using this new DLL.

```
csc /target:library /out:Hello.dll HelloWorld.cs AssemblyInfo.cs
gacutil /i Hello.dll
csc /target:exe /out:MyApp.exe HelloWorldDriver.cs
/reference:Hello.dll
```

Now type `MyApp`, and you should see the following output:

```
A slightly different Hello World
```

To list the contents of the GAC, you type `gacutil /l`. This will output all the shared assemblies in your environment. You will see the two versions of `Hello.dll` in the output:

```
Hello, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=cc6fce1c7117f564om=null

Hello, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=cc6fce1c7117f564om=null
```

Note that the version information is different for the two assemblies.

Now that we have two assemblies in the GAC, how do we force an application to use the desired version of the assembly? The .NET Framework supports **assembly policy files**, which are XML files that indicate

how the assembly is to be upgraded. A sample policy file is shown in Listing 23.9.

Listing 23.9 A Sample .NET Policy File (C#)

```
<configuration>
  <runtime>
    <assemblyBinding>
      <dependentAssembly>
        <assemblyIdentity name="Hello"
          publicKeyToken="cc6fce1c7117f564"
                              culture=""/>
          <bindingRedirect oldVersion="1.0.0.0"
                           newVersion="2.0.0.0"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

The new policy-based `Hello.dll` is then created using the following command:

```
al /link:policy.xml /keyfile:Hello.snk /out:policy.1.0.Hello.dll
```

Note that the name format of the new `Hello.dll` includes the word `policy` with a number indicating clearly that the assembly enforces a policy. The next step is to register this policy-based DLL in the GAC:

```
gacutil /i policy.1.0.Hello.dll
```

23.5 Application Configuration

Many applications rely on external static settings that they read before they do something useful. Such settings are never hardcoded in the application itself, because doing so forces a recompilation of the application whenever a simple static setting changes.

A traditional approach is to store these static settings in a static configuration file. Java programmers are probably thinking about `.properties` files. A Java property file is a file containing name/value pairs that can be used at runtime by the application. JDK 1.4 introduced a more structured way of storing these settings in a class called `java.util.Preferences`.

The .NET Framework comes with its own arsenal of classes that allow configuration of the application. Because everything is XML-based in the .NET Framework, there is no special syntax to be learned. The framework provides a rich API for reading these configuration files.

A .NET configuration file ends with the `.config` file extension. For the application `MyApp.exe` the configuration file would be `MyApp.exe.config`. For multiple-file assemblies, the configuration file is named after the file that contains the manifest. The CLR ensures that the configuration files are loaded automatically, and therefore no special effort is needed by the programmer. The configuration files must contain the root XML node `<configuration>`.

In its simplest form, custom application settings are defined in the `<appSettings>` section of the configuration file. The settings are defined as key-value pairs.

```
<configuration>
  <appSettings>
  <add key="Key1" value="Value1"/>
  <add key="Key2" value="Value2"/>
  <add key="Key3" value="Value3"/>
</appSettings>
```

You can also specify **sections** in the configuration file. Sections are used to define complex settings. There are three types of sections depending on the handler that is specified for each section: a single-tag section, a name/value section, and a dictionary section.

23.5.1 Single-Tag Section

A **single-tag section** allows you to define the key-value pairs as an attribute of a single XML tag:

```
<configuration>
  <configSections>
```

```
  <section name="Single"
    type="System.Configuration.SingleTagSectionHandler"/>
  </configSections>
  <Single Key1 = "Value1"
          Key2 = "Value2"
          Key3 = "Value3"></Single>
</configuration>
```

23.5.2 Name/Value Section

A **name/value section** allows you to define the key-value pairs in the same way as the simple settings are defined:

```
<configuration>
  <configSections>
  <section name="NameValue"
    type="System.Configuration.NameValueSectionHandler" />
  </configSections>
  <NameValue>
    <add key="Key1" value="Value1"/>
    <add key="Key2" value="Value2"/>
    <add key="Key3" value="Value3"/>
  </NameValue>
</configuration>
```

23.5.3 Dictionary Section

A **dictionary section** is similar to a name/value section except that a dictionary section uses a different section handler:

```
<configuration>
  <configSections>
    <section name=" Dictionary"
      type="System.Configuration.DictionarySectionHandler" />
  </ configSections >
  <Dictionary>
    <add key="Key1" value="Value1"/>
    <add key="Key2" value="Value2"/>
    <add key="Key3" value="Value3"/>
  </Dictionary>
</configuration>
```

You can even signal the configuration file parser to ignore sections by making the section handler an `IgnoreSectionHandler`.

```
<configuration>
  <configSections>
<section name="Advanced"
type="System.Configuration.IgnoreSectionHandler" />
  </configSections>
<Advanced>
<add key="Key1" value="Value1"/>
<add key="Key2" value="Value2"/>
<add key="Key3" value="Value3"/>
</Advanced>
</configuration>
```

23.5.4 A Sample Config File

Listings 23.10 and 23.11 show how to handle the simple application settings along with the section settings we've discussed. Listing 23.10 shows the complete `MyApp.config` file.

Listing 23.10 A Sample .NET Config File (C#)

```
<configuration>
  <configSections>
    <section name="Single"
    type="System.Configuration.SingleTagSectionHandler" />

    <section name="NameValue"
    type="System.Configuration.NameValueSectionHandler" />

    <section name="Dictionary"
    type="System.Configuration.DictionarySectionHandler" />
  </configSections>

<appSettings>
  <add key="SimpleKey1" value="SimpleValue1"/>
  <add key="SimpleKey2" value="SimpleValue2"/>
  <add key="SimpleKey3" value="SimpleValue3"/>
</appSettings>
```

```
  <Single SingleKey1 = "SingleValue1"
    SingleKey2 = "SingleValue2"
    SingleKey3 = "SingleValue3" />

  <NameValue>
  <add key="NameValueKey1" value="NameValueValue1"/>
  <add key="NameValueKey2" value="NameValueValue2"/>
  <add key="NameValueKey3" value="NameValueValue3"/>
  </NameValue>

  <Dictionary>
    <add key="DictionaryKey1" value="DictionaryValue1"/>
    <add key="DictionaryKey2" value="DictionaryValue2"/>
    <add key="DictionaryKey3" value="DictionaryValue3"/>
  </Dictionary>

</configuration>
```

Note that there can be only one `<configSections>` element of the `<configuration>` root, and it must be the first child of the root node. Listing 23.11 shows how the different settings of the config file are read.

Listing 23.11 A Simple C# Class to Read the Application Settings

```
using System;
using System.Collections;
using System.Collections.Specialized;
using System.Configuration;

public class MyApp {

  public static void Main(string[] args) {
    //Read simple settings
    ReadSimpleSettings();
    //Read single-tag settings
    ReadSingleTagSettings();
    //Read single-tag settings
    ReadNameValueSettings();
    //Read single-tag settings
    ReadDictionarySettings();
  }
```

```
  private static void ReadSimpleSettings() {
    Console.WriteLine(
    ConfigurationSettings.AppSettings["SimpleKey1"]);
  }

  private static void ReadSingleTagSettings() {
    IDictionary dict = (IDictionary)
    ConfigurationSettings.GetConfig("Single");
    Console.WriteLine(dict["SingleKey2"]);
  }

  private static void ReadNameValueSettings() {
    NameValueCollection dict = (NameValueCollection)
    ConfigurationSettings.GetConfig("NameValue");
    Console.WriteLine(dict["NameValueKey3"]);
  }

  private static void ReadDictionarySettings() {
    Hashtable dict = (Hashtable)
    ConfigurationSettings.GetConfig("Dictionary");
    Console.WriteLine(dict["DictionaryKey1"]);
  }
}
```

The output of Listing 23.11 is as follows:

```
SimpleValue1
SingleValue2
NameValueValue3
DictionaryValue1
```

The code in Listing 23.11 is self-explanatory. The `ConfigurationSettings` class of the `System.Configuration` namespace contains all the methods needed to parse a config file. Note that the object returned by the `GetConfig` method is cast to the appropriate collection depending on the section being read.

Note also that the collections returned by the `ConfigurationSettings` class are read-only. You cannot use this class to modify the config file. You should do this manually.

23.6 Process Management

Traditionally, the unit of isolation for applications has been at the process level. A dedicated operating system user process is allocated to every instance of the running application, and stopping the application means killing the OS process. The .NET Framework introduces the concept of application domains, which are new, lighter units of isolation. The CLR allows a process to consist of many application domains, each independent of the others. Application domains thus create a boundary for applications to coexist within the same OS process. Hence, an interapplication method call is made within the same OS process, and that is lighter than making an interprocess method call. Applications can be stopped independently of each other, and a fault in one application does not affect the other applications in the same OS process.

The .NET application domain concept is reminiscent of Web-application isolation provided by Java-based application servers. The J2EE specification mandates that Web applications be independent of each other even though they are created in the same JVM instance. Classes loaded in one Web application are not available to another Web application. This isolation is achieved by having different class loaders for each Web application.

Associated with each application is an application domain. When you start, say, a .NET shell executable from the command line, a piece of software creates the default application domain and your application-specific domain. This piece of software is the **CLR host**. The CLR host is responsible for loading the CLR into the OS process, creating a logical partition within the OS process so that your application can reside in that partition. Running a .NET shell executable automatically invokes the CLR host associated with shell executables. In .NET an application domain is modeled using the `System.AppDomain` class.

23.6.1 Querying the Current Application Domain

Listing 23.12 shows how a shell executable can query its current application domain and get at its properties.

Listing 23.12 Current Application Domain Properties (C#)

```
using System;
using System.Threading;
```

```
public class Test {

  public static void Main(string[] args) {

    AppDomain domain = Thread.GetDomain();
    Console.WriteLine("Friendly name "+domain.FriendlyName);
    Console.WriteLine("Base directory "+domain.BaseDirectory);

    AppDomainSetup setup = domain.SetupInformation;
    Console.WriteLine("ApplicationBase "+setup.
                      ApplicationBase);
    Console.WriteLine("ApplicationName "+setup.
                      ApplicationName);
    Console.WriteLine("Config file "+setup.ConfigurationFile);
  }
}
```

The output of Listing 23.12 is as follows:

```
Friendly name Deployment.exe
Base directory C:\BOOK\code\Chapter 23\Deployment\bin\Debug\
ApplicationBase C:\BOOK\code\Chapter 23\Deployment\bin\Debug\
ApplicationName
Config file C:\BOOK\code\Chapter
23\Deployment\bin\Debug\Deployment.exe.config
```

You can obtain the application domain of the application by calling the `GetDomain` method of the current thread. The `AppDomain` class defines several properties of interest.

23.6.2 Executing an Application in a Remote Application Domain

To execute an application in another application domain, you must first create that application domain. Listing 23.13 shows how.

Listing 23.13 Executing an Assembly inside a Custom Application Domain (C#)

```
using System;

public class Test {
```

```
  public static void Main(string[] args) {

    AppDomainSetup info = new AppDomainSetup();
    info.ApplicationBase = System.Environment.CurrentDirectory;
    AppDomain dom = AppDomain.CreateDomain(
           "RemoteDomain", null,info);
    dom.ExecuteAssembly("c:\\book\\code\\Chapter 23\\MyApp.exe");
    AppDomain.Unload(dom);
  }
}
```

The output of Listing 23.13 is as follows:

```
A slightly different Hello World
```

Listing 23.13 creates a custom application domain called `RemoteDomain` and executes and runs a previously developed assembly (`MyApp.exe`). Note that in this case the application that is run is trivial, but in practice the application can be complex, and it is important to unload the assembly. There is no API for unloading an assembly directly, but you can do so by unloading the application domain in which it runs:

```
AppDomain.Unload(dom);
```

23.6.3 Invoking a Method in a Remote Application Domain

As mentioned earlier, in the absence of application domains, invoking methods of other applications typically means making an interprocess call, which is slower than making an intraprocess call. To demonstrate how to call a method of a class loaded in another application domain, we first create a simple class and create a library type assembly (DLL) out of it. Listing 23.14 shows the `RemoteHello` class that we will use to create the DLL.

Listing 23.14 `RemoteHello.cs` (C#)

```
using System;

public class RemoteHello : MarshalByRefObject {
```

```
  string greeting;
  public RemoteHello(string greeting) {
    this.greeting = greeting;
  }

  public void Greet() {
    Console.WriteLine("Hello "+greeting);
  }
}
```

`MarshalByRefObject` is an abstract class that enables access to objects across application domain boundaries.

Next, we compile `RemoteHello.cs` into `RemoteHello.dll`:

```
csc /target:library /out:RemoteHello.dll RemoteHello.cs
```

Now that we have the DLL, Listing 23.15 shows how to call the `Greet()` method by loading the DLL in a custom application domain.

Listing 23.15 `RemoteHelloDriver.cs` (C#)

```
using System;
using System.Reflection;
using System.Runtime.Remoting;

public class Test {

  public static void Main(string[] args) {

    //Set up information regarding the application domain
    AppDomainSetup info = new AppDomainSetup();
    info.ApplicationBase = System.Environment.CurrentDirectory;

    // Create an application domain with null evidence
    AppDomain dom = AppDomain.CreateDomain(
          "RemoteDomain", null,info);
    BindingFlags flags = (BindingFlags.Public |
      BindingFlags.Instance | BindingFlags.CreateInstance);
    ObjectHandle objh = dom.CreateInstance("RemoteHello",
      "RemoteHello", false, flags, null, new String[]{"Hello
      World!"}, null, null, null);
```

```
        Object obj = objh.Unwrap();
        // Cast to the actual type

        RemoteHello h = (RemoteHello)obj;
        // Invoke the method

        h.Greet();

        // Clean up by unloading the application domain
        AppDomain.Unload(dom);
    }
}
```

Because Listing 23.15 references the `RemoteHello` class, we must compile it using the following command line:

```
csc /r:RemoteHello.dll RemoteHelloDriver.cs
```

If we now run the `RemoteHelloDrive.exe` we will see the following output:

```
Hello Hello World!
```

Being able to execute other applications or even call methods on types defined in other applications without slowing execution is a huge positive of the .NET Framework. It allows enterprise developers to develop and deploy common enterprise components in one assembly and reuse those components in other applications.

We mentioned earlier that multiple applications can be run in a single OS process using the application domain as the unit of isolation and that you can stop one application without affecting the other. To illustrate this, we create `Application1.cs` (Listing 23.16) and `Application2.cs` (Listing 23.17) and then create the assemblies `Application1.exe` and `Application2.exe`, respectively, from the two listings.

Listing 23.16 `Application1.cs` (C#)

```
using System;
using System.Threading;

public class Test {
```

```
  static void Main(string[] args) {
    Thread t = new Thread(new ThreadStart(new Test().Run));
    t.Start();
  }

  public void Run() {
    while (true) {
      Thread.Sleep(400);
      Console.WriteLine("Running App 1");
    }
  }
}
```

Run the following command to create the `Application1.exe` assembly:

```
csc Application1.cs
```

Next, we create `Application2.cs` (Listing 23.17).

Listing 23.17 `Application2.cs` (C#)

```
using System;
using System.Threading;

public class Test{

  static void Main(string[] args) {
    Thread t = new Thread(new ThreadStart(new Test().Run));
    t.Start();
  }

  public void Run() {
    while (true) {
      Thread.Sleep(500);
      Console.WriteLine("Running App 2");
    }
  }
}
```

Run the following command to create the `Application2.exe` assembly:

```
csc Application2.cs
```

Now we create the source file that we will use to create two application domains and run `Application1.exe` and `Application2.exe` in these two domains. We will call this class `Monitor.cs` (Listing 23.18).

Listing 23.18 `Monitor.cs` (C#)

```
using System;
using System.Threading;
using System.Reflection;

public class Test {

  AppDomain dom1, dom2;

  public Test() {

    AppDomainSetup info = new AppDomainSetup();
    info.ApplicationBase = System.Environment.CurrentDirectory;

    dom1 = AppDomain.CreateDomain("RemoteDomain1", null, info);
    dom2 = AppDomain.CreateDomain("RemoteDomain2", null, info);
    dom1.ExecuteAssembly("c:\\book\\code\\Chapter23
          \\Deployment\\Application1.exe");
    dom2.ExecuteAssembly("c:\\book\\code\\Chapter23
          \\Deployment\\Application2.exe");

    public static void Main(string[] args) {
    Thread t = new Thread(new ThreadStart(new Test().Run));
    t.Start();
  }

    public void Run() {
      Thread.Sleep(1000);
      Console.WriteLine("Unloading dom1");
      AppDomain.Unload(dom1);
      Thread.Sleep(1000);
      Console.WriteLine("Unloading dom2");
      AppDomain.Unload(dom2);
        }
}
```

When we run `Monitor.exe` we get the following output:

```
Running App 1
Running App 2
Running App 1
Unloading dom1
Running App 2
Running App 2
Running App 2
Unloading dom2
```

The contents of Listing 23.18 are self-explanatory. The class instantiates a thread, and in its constructor it creates two application domains and executes `Application1.exe` and `Application2.exe` in those domains. Next, the instantiated thread sleeps for 1,000 ms and then unloads the application domain running `Application1.exe`. Note that after this statement, `Application2.exe` is the only assembly running and printing. The main thread then sleeps some more and then unloads the second application domain, thereby stopping all the applications.

`System.Diagnostics`

The `System.Diagnostics` namespace contains many useful classes that help your application code deal with the operating system. Although there are several fun Windows-specific operations that you can do using the classes in this namespace, one of the more important things, perhaps more familiar to Java programmers, is monitoring the operating system process. In Java, this is modeled using the `java.lang.Process` class. This class provides only read-only methods. You cannot create a process in Java using the methods of this class. In .NET the `Process` class of this namespace can be used to query existing process details as well as create new processes.

23.6.4 Querying Processes

Listing 23.19 shows how to query an existing process. The application repeatedly prints the details of the current process in which it runs. Because the application creates a thread, it keeps the process running, and hence you will get an unending stream of output until you kill the process.

Listing 23.19 Getting the Current .NET Process Details (C#)

```
using System;
using System.Threading;
```

```
using System.Diagnostics;

public class OSProcessApp {

  Process p;

  public OSProcessApp() { p = Process.GetCurrentProcess(); }

    static void Main(string[] args) {
      Thread t = new Thread(new ThreadStart(new
      OSProcessApp().Run));
      t.Start();
  }

  public void Run() {
    while (true) {
      Thread.Sleep(400);
      Console.WriteLine(p.BasePriority);
      Console.WriteLine(p.Id);
      Console.WriteLine(p.TotalProcessorTime);
      Console.WriteLine(p.UserProcessorTime);
      Console.WriteLine(p.VirtualMemorySize);
      Console.WriteLine(p.MachineName);
    }
  }
}
```

The output of Listing 23.19 will depend on your machine, but here is a sample excerpt:

```
Priority 8
Id 1380
Total processor time 00:00:00.5007200
User processor time 00:00:00.2303312
Virtual memory size 90882048
Machine name .
Priority 8
Id 1380
Total processor time 00:00:00.5007200
```

```
User processor time 00:00:00.2303312
Virtual memory size 90882048
```

23.6.5 Creating and Killing Processes

Listing 23.20 shows how to create a new process and then kill it. We create and kill the process required to run `Application1.exe`, created earlier in this chapter.

Listing 23.20 Creating and Killing Processes (C#)

```
using System;
using System.Threading;
using System.Diagnostics;

public class Test {

  static void Main(string[] args) {

    try {
      Process p = Process.Start("C:\\BOOK\\code\\Chapter
      23\\Deployment\\Application1.exe");
      Thread.Sleep(5000);
      p.Kill();
    } catch (Exception e) {
      Console.WriteLine(e.StackTrace);
      }

  }
}
```

When you run Listing 23.20 from the command line, a second console window pops up and prints the following:

```
Running App 1
Running App 1
Running App 1
Running App 1
Running App 1
Running App 1
```

After about 5 seconds the second pop-up console window automatically closes itself.

Listing 23.20 uses the `static` method of the `Process` class to create a process. You can use the constructor instead and specify the `ProcessStartInfo` parameters corresponding to this process.

23.6.6 Redirecting Process Output

Sometimes it is necessary to redirect the output generated by a particular process. Java programmers attempting to get the process ID (PID) of a UNIX process are perhaps familiar with this approach of running a UNIX command and then parsing the output from the command. Listing 23.21 shows how to achieve that in C#.

Listing 23.21 Redirecting Output of a Process (C#)

```
using System;
using System.Diagnostics;

public class Test {

  public static void Main() {

    Process p = new Process();
    p.StartInfo.FileName = "cmd.exe";
    p.StartInfo.Arguments = "/c dir *.exe";
    p.StartInfo.UseShellExecute = false;
    p.StartInfo.RedirectStandardOutput = true;
    p.Start();
    string output = p.StandardOutput.ReadToEnd();
    Console.WriteLine("Output of command "+output);
  }

}
```

The output of Listing 23.21 is as follows:

```
Output of command Volume in drive C is Local Disk
Volume Serial Number is 50DD-3A83
Directory of C:\BOOK\code\Chapter 23\Deployment\bin\Debug
01/05/2003 10:13p 5,632 Deployment.exe
```

```
1 File(s)5,632 bytes
0 Dir(s) 14,532,866,048 bytes free
```

23.6.7 Detecting Process Completion

Earlier we created a process and then killed it without caring whether the process was finished executing. In practice, however, you want to know whether the process has completed. The .NET Framework provides an event handler hook that lets you know when the process has exited. Listing 23.22 takes the example in Listing 23.21 and adds event handling.

Listing 23.22 Detecting Process Completion (C#)

```
using System;
using System.Diagnostics;

public class Test {

  public static void Main() {

    Process p = new Process();
    p.StartInfo.FileName = "cmd.exe";
    p.StartInfo.Arguments = "/c dir *.exe";
    p.StartInfo.UseShellExecute = false;
    p.StartInfo.RedirectStandardOutput = true;

    p.EnableRaisingEvents = true;
    p.Exited += new EventHandler(ProcessDone);

    p.Start();

    string output = p.StandardOutput.ReadToEnd();
    Console.WriteLine("Output of command "+output);

    p.WaitForExit();
  }

  private static void ProcessDone(object sender, EventArgs e) {
    Console.WriteLine("Process Exited");
  }
}
```

The output of Listing 23.22 is as follows:

```
Volume Serial Number is 50DD-3A83
Directory of C:\BOOK\code\Chapter 23\Deployment\bin\Debug
01/05/2003 10:13p 5,632 Deployment.exe
1 File(s)5,632 bytes
0 Dir(s) 14,532,866,048 bytes free
```

23.6.8 Exiting a Process

Note that in Listing 23.22 we provide two ways of detecting process completion. One is by registering the `ProcessDone` event handler, and the other is the `WaitForExit` method, which returns when the process is finished.

The `Process` class provides several interesting methods and is richer in its API compared with its Java counterpart. For example, if you have been Web surfing a lot lately and want to quickly kill all your IE browser instances with one click, you can simply create an EXE assembly containing the code displayed in Listing 23.23.

Listing 23.23 Getting Rid of All IE Instances (C#)

```
using System;

using System.Diagnostics;
public class Test {
  public static void Main() {
    Process[] processes = Process.GetProcessesByName
("IEXPLORE");
    foreach (Process p in processes) {
      p.Kill();
    }
  }
}
```

C# also provides an API for querying services and operating system (Windows) log files. Classes for reporting at the level of the operating system are in the `System.Diagnostics` namespace.

23.7 Summary

Both C# and Java provide tools and APIs to package and manage applications.

- Applications in Java are class files or their packaged archive (`.jar`) files. Depending on the application type, Java provides different flavors of the `.jar` file (`.war` and `.ear` files). These archive files are merely a collection of disk files. The unit of deployment in .NET is an assembly. An assembly can be of the type EXE, module, or library (DLL). An assembly can comprise a single file or multiple modules (perhaps written in different languages). Assemblies are represented by the `System.Reflection.Assembly` class. You can use this class to explore assemblies, or you can use reflection to dynamically create assemblies. Assemblies have globally unique strong names. Unlike `.jar`, `.war`, or `.ear` files, assemblies can be versioned. An assembly can have an associated version policy file that can dictate the upgrade path of the assembly. It is possible for two similar assemblies of different versions to coexist.
- Application configuration is built into .NET. Unlike Java property files, the .NET configuration files are XML-based. Configuration files are automatically recognized by the CLR. Configuration files can contain sections of name/value pairs that can be accessed by the `System.Collections` classes.
- In .NET, the unit of application isolation is an application domain. In Java, depending on the application, the unit of application isolation is the JVM instance; or, in the case of modern J2EE-compliant application servers, it can be the Web application–specific class loader, which allows for similar classes to coexist between two Web applications. In .NET, a single operating system process can contain several application domains, each mapped to an application. These applications behave independently of each other. You can stop applications without affecting sibling applications running in the same process. A fault in one application does not affect other applications. Application domains have built-in support for loading and executing remote assemblies. You can call methods on types defined in remote applications without incurring any performance penalty because the method call is intraprocess instead of interprocess.

- The .NET `System.Diagnostics` namespace provides several classes for interacting with the operating system. One such class is the `Process` class, which is similar to the `java.lang.Process` class. However, unlike the Java counterpart, a .NET `Process` class lets you do a lot. You can query, create, and kill processes.