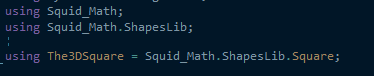
**Building and Configuring Class Libraries**

During the first four parts of this book, you have created a number of “stand-alone” executable applications, in which all the programming logic was packaged within a single executable file (\*.exe). These executable assemblies were using little more than the primary .NET class library, mscorlib.dll. While some simple .NET programs may be constructed using nothing more than the .NET base class libraries, chances are it will be commonplace for you (or your teammates) to isolate reusable programming logic into custom class libraries (\*.dll files) that can be shared among applications. In this chapter, you will learn about various ways to package your types into custom libraries of code. To begin, you’ll learn the details of partitioning types into .NET namespaces. After this, you will examine the class library project templates of Visual Studio and learn the distinction between private and shared assemblies. Next, you’ll explore exactly how the .NET runtime resolves the location of an assembly, and you’ll come to understand the global assembly cache, XML application configuration files (\*.config files), publisher policy assemblies, and the System.Configuration namespace.

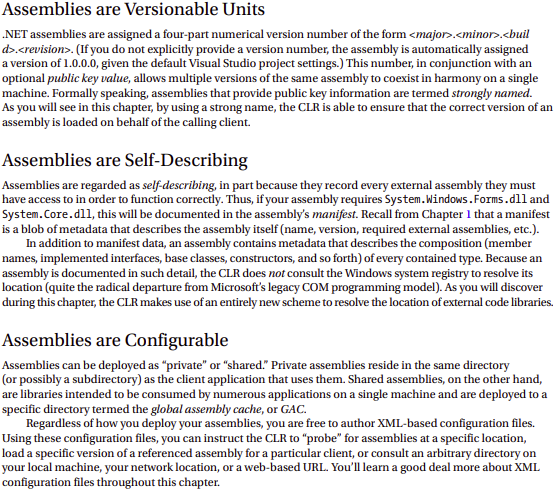
**Defining Custom Namespaces** Before diving into the aspects of library deployment and configuration, the first task is to learn the details of packaging your custom types into .NET namespaces. Up to this point in the text, you’ve been building small test programs that leverage existing namespaces in the .NET universe (System, in particular). However, when you build larger applications with many types, it can be helpful to group your related types into custom namespaces. In C#, this is accomplished using the namespace keyword. Explicitly defining custom namespaces is even more important when creating .NET \*.dll assemblies, as other developers will need to reference the library and import your custom namespaces to use your types.

**Resolving Name Clashes with Aliases** The C# using keyword also lets you create an alias for a type’s fully qualified name. When you do so, you define a token that is substituted for the type’s full name at compile time.

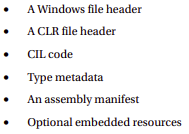
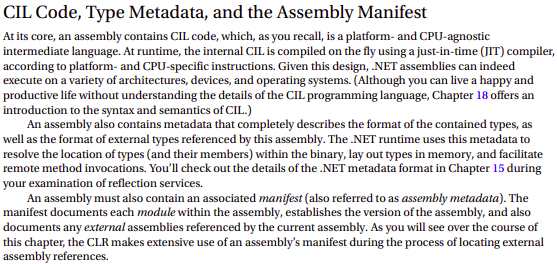
**Creating Nested Namespaces** When organizing your types, you are free to define namespaces within other namespaces. The .NET base class libraries do so in numerous places to provide deeper levels of type organization. F

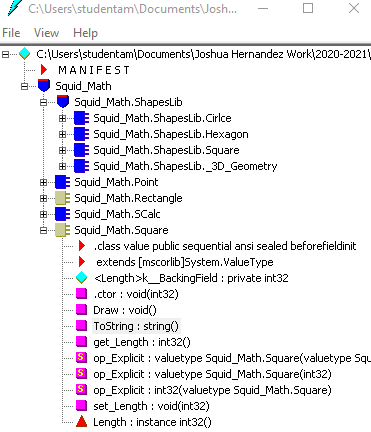
**The Role of .NET Assemblies** .NET applications are constructed by piecing together any number of assemblies. Simply put, an assembly is a versioned, self-describing binary file hosted by the CLR. Now, despite that .NET assemblies have the same file extensions (\*.exe or \*.dll) as previous Windows binaries, they have little in common with those files under the hood. Thus, to set the stage for the information to come, let’s consider some of the benefits provided by the assembly format.

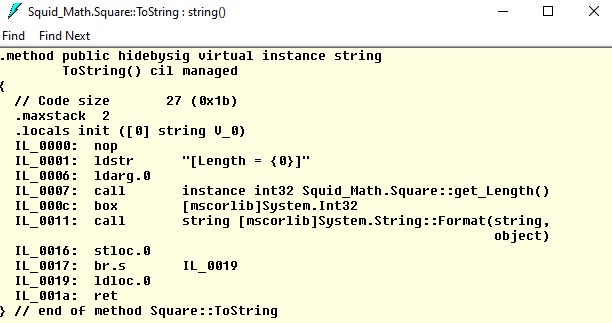
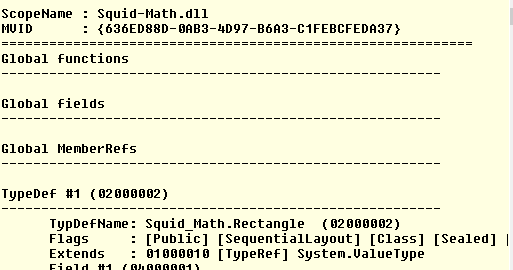
**Assemblies Promote Code Reuse** As you have built your Console Application projects over the previous chapters, it might have seemed that all the applications’ functionality was contained within the executable assembly you were constructing. In reality, your applications were leveraging numerous types contained within the always-accessible .NET code library, mscorlib.dll (recall that the C# compiler references mscorlib.dll automatically), and in the case of some examples, System.Core.dll.

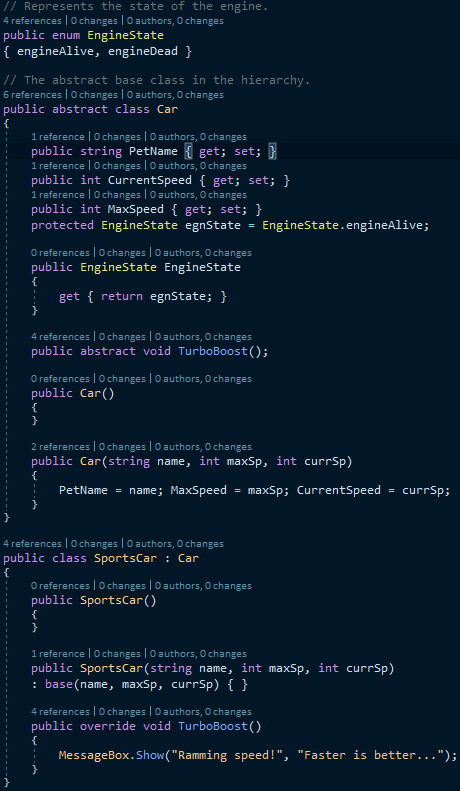
Regardless of how a code library is packaged, the .NET platform allows you to reuse types in a language-independent manner. For example, you could create a code library in C# and reuse that library in any other .NET programming language. It is possible not only to allocate types across languages but also to derive from them. A base class defined in C# could be extended by a class authored in Visual Basic. Interfaces defined in F# can be implemented by structures defined in C#, and so forth. The point is that when you begin to break apart a single monolithic executable into numerous .NET assemblies, you achieve a language-neutral form of code reuse.

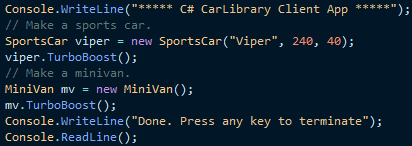
**Assemblies Establish a Type Boundary** Recall that a type’s fully qualified name is composed by prefixing the type’s namespace (e.g., System) to its name (e.g., Console). Strictly speaking, however, the assembly in which a type resides further establishes a type’s identity. For example, if you have two uniquely named assemblies (say, MyCars.dll and YourCars.dll) that both define a namespace (CarLibrary) containing a class named SportsCar, they are considered unique types in the .NET universe.

**The CLR File Header** The CLR header is a block of data that all .NET assemblies must support (and do support, courtesy of the C# compiler) to be hosted by the CLR. In a nutshell, this header defines numerous flags that enable the runtime to understand the layout of the managed file.

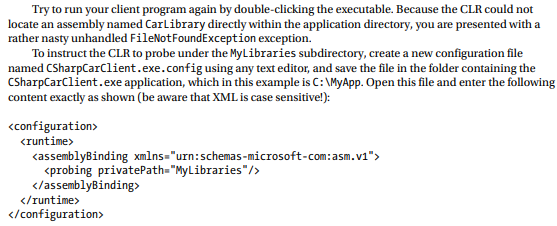


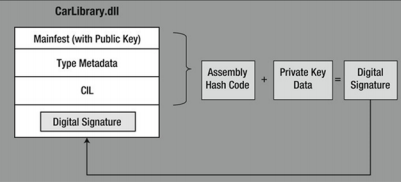
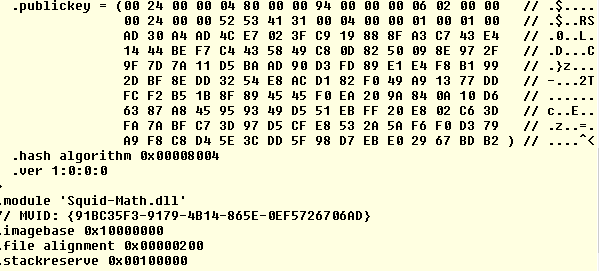




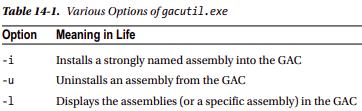
**Building a C# Client Application** Because each of the CarLibrary types has been declared using the public keyword, other .NET applications are able to use them as well. Recall that you may also define types using the C# internal keyword (in fact, this is the default C# access mode). Internal types can be used only by the assembly in which they are defined. External clients can neither see nor create types marked with the internal keyword.

**Understanding Private Assemblies** Technically speaking, the class libraries you’ve created thus far in this chapter have been deployed as private assemblies. Private assemblies must be located within the same directory as the client application that’s using them (the application directory) or a subdirectory thereof. Recall that when you add a reference to CarLibrary.dll while building the CSharpCarClient.exe and VisualBasicCarClient.exe applications, Visual Studio responded by placing a copy of CarLibrary.dll within the client’s application directory (at least, after the first compilation).

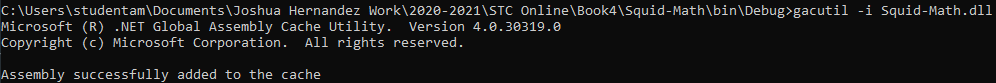


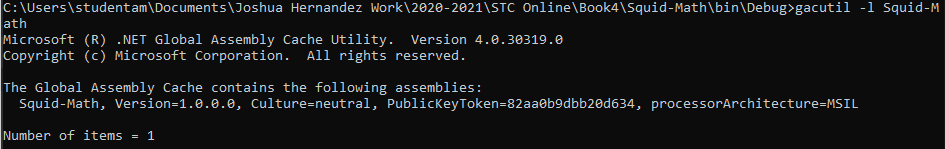
**The Global Assembly Cache** As suggested in the previous paragraph, a shared assembly is not deployed within the same directory as the application that uses it. Rather, shared assemblies are installed into the GAC. However, the exact location of the GAC will depend on which versions of the .NET platform you installed on the target computer

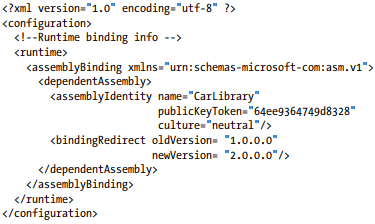


Command

**Installing Strongly Named Assemblies** **to the GAC** The final step is to install the (now strongly named) CarLibrary.dll into the GAC. While the preferred way to deploy assemblies to the GAC in a production setting is to create an installer package (using a commercial installer program such as InstallShield), the .NET Framework SDK ships with a command-line tool named gacutil.exe, which can be useful for quick tests





**Understanding Publisher Policy Assemblies** The next configuration issue you’ll examine is the role of publisher policy assemblies. As you’ve just seen, \*.config files can be constructed to bind to a specific version of a shared assembly, thereby bypassing the version recorded in the client manifest. While this is all well and good, imagine you’re an administrator who now needs to reconfigure all client applications on a given machine to rebind to version 2.0.0.0 of the CarLibrary.dll assembly. Given the strict naming convention of a configuration file, you would need to duplicate the same XML content in numerous locations (assuming you are, in fact, aware of the locations of the executables using CarLibrary!). Clearly this would be a maintenance nightmare.

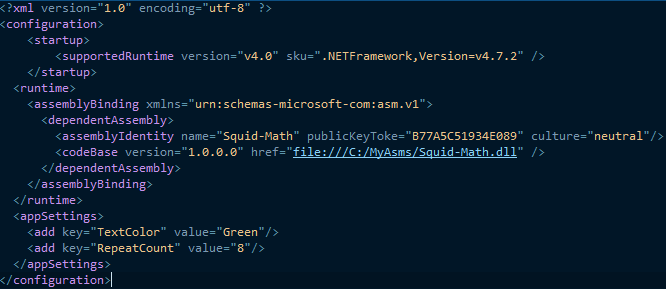
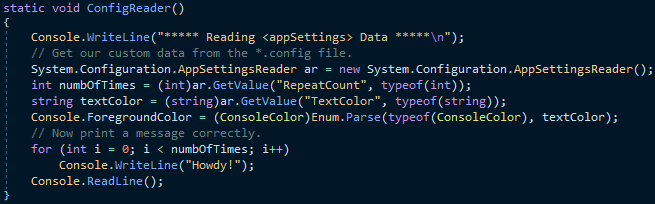
**Disabling Publisher Policy** Now, assume you (as a system administrator) have deployed a publisher policy assembly (and the latest version of the related assembly) to the GAC of a client machine. As luck would have it, nine of the ten affected applications rebind to version 2.0.0.0 without error. However, the remaining client application (for whatever reason) blows up when accessing CarLibrary.dll 2.0.0.0. (As we all know, it is next to impossible to build backward-compatible software that works 100 percent of the time.)

**Understanding the <codebase> Element** Application configuration files can also specify code bases. The element can be used to instruct the CLR to probe for dependent assemblies located at arbitrary locations (such as network end points or an arbitrary machine path outside a client’s application directory).

If the value assigned to a element is located on a remote machine, the assembly will be downloaded on demand to a specific directory in the GAC termed the download cache. Given what you have learned about deploying assemblies to the GAC, it should make sense that assemblies loaded from a element will need to be assigned a strong name (after all, how else could the CLR install remote assemblies to the GAC?).







**The Configuration File Schema Documentation** In this chapter, you were introduced to the role of XML configuration files. Here, you focused on a few settings you can add to the element that control how the CLR will attempt to locate externally required libraries. As you work on upcoming chapters of this book (and as you move beyond this book and begin to build larger-scale software), you will quickly notice that use of XML configuration files is commonplace.

**Summary** This chapter examined the role of .NET class libraries (aka .NET \*.dlls). As you have seen, class libraries are .NET binaries that contain logic intended to be reused across a variety of projects. Recall that libraries can be deployed in two primary ways, specifically privately or shared. Private assemblies are deployed to the client folder or a subdirectory thereof, provided you have a proper XML configuration file. Shared assemblies are libraries that can be used by any application on the machine and can also be influenced by the settings in a client-side configuration file.