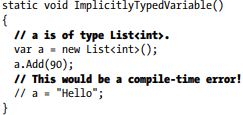
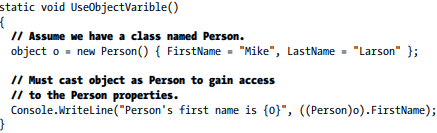
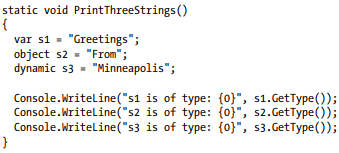
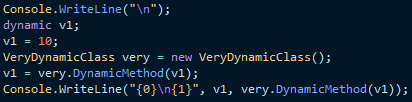
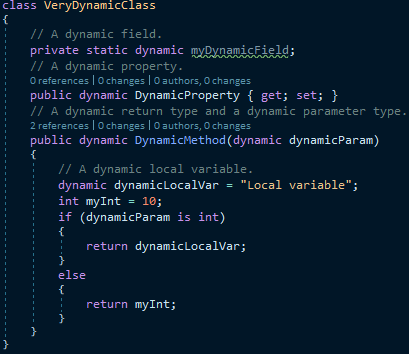
**Dynamic Types and the Dynamic Language Runtime**

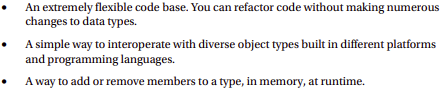
NET 4.0 introduced a new keyword to the C# language, specifically, dynamic. This keyword allows you to incorporate scripting-like behaviors into the strongly typed world of type safety, semicolons, and curly brackets. Using this loose typing, you can greatly simplify some complex coding tasks and also gain the ability to interoperate with a number of dynamic languages (such as IronRuby or IronPython), which are .NET savvy. In this chapter, you will be introduced to the C# dynamic keyword and understand how loosely typed calls are mapped to the correct in-memory object using the Dynamic Language Runtime (DLR). After you understand the services provided by the DLR, you will see examples of using dynamic types to streamline how you can perform late-bound method calls (via reflection services) and to easily communicate with legacy COM libraries.

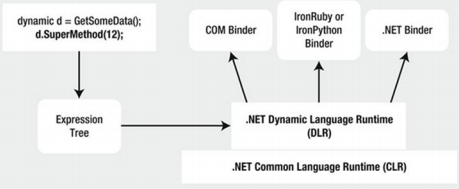
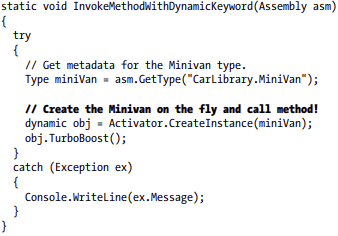
**The Role of the C# dynamic Keyword** Back in Chapter 3, you learned about the var keyword, which allows you to define local variables in such a way that the underlying date type is determined at compile time, based on the initial assignment (recall that this is termed implicit typing). Once this initial assignment has been made, you have a strongly typed variable, and any attempt to assign an incompatible value will result in a compiler error



**Limitations of the dynamic Keyword** While a great many things can be defined using the dynamic keyword, there are some limitations regarding its usage. While they are not showstoppers, do know that a dynamic data item cannot make use of lambda expressions or C# anonymous methods when calling a method

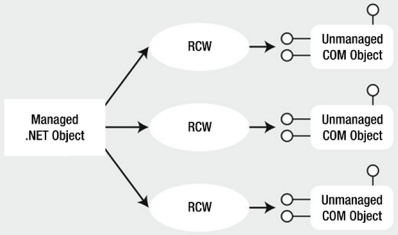


**Practical Uses of the dynamic Keyword** Given that dynamic data is not strongly typed, not checked at compile time, has no ability to trigger IntelliSense, and cannot be the target of a LINQ query, you are absolutely correct to assume that using the dynamic keyword just for the sake of doing so is a poor programming practice. However, in a few circumstances, the dynamic keyword can radically reduce the amount of code you need to author by hand. Specifically, if you are building a .NET application that makes heavy use of late binding (via reflection), the dynamic keyword can save you typing time. As well, if you are building a .NET application that needs to communicate with legacy COM libraries (such as Microsoft Office products), you can greatly simplify your code base via the dynamic keyword. By way of a final example, web sites built using the MVC design pattern frequently use the ViewBag type, which can also be accessed in a simplified manner using the dynamic keyword. Like any “shortcut,” you need to weigh the pros and cons. The use of the dynamic keyword is a tradeoff between brevity of code and type safety. While C# is a strongly typed language at its core, you can opt in (or opt out) of dynamic behaviors on a call-by-call basis. Always remember that you never need to use the dynamic keyword. You could always get to the same end result by authoring alternative code by hand (and typically much more of it).

**Dynamic Runtime Lookup of Expression** Trees As explained, the DLR will pass the expression trees to a target object; however, this dispatching will be influenced by a few factors. If the dynamic data type is pointing in memory to a COM object, the expression tree is sent to a low-level COM interface named IDispatch. As you might know, this interface was COM’s way of incorporating its own set of dynamic services. COM objects, however, can be used in a .NET application without the use of the DLR or C# dynamic keyword. Doing so, however (as you will see), tends to result in much more complex C# coding.

**Simplifying Late-Bound Calls Using Dynamic Types** One instance where you might decide to use the dynamic keyword is when you are working with reflection services, specifically when making late-bound method calls. In Chapter 15, you saw a few examples of when this type of method call can be useful, most commonly when you are building some type of extensible application. At that time, you learned how to use the Activator.CreateInstance() method to create an object, for which you have no compile-time knowledge of (beyond its display name).

**Leveraging the dynamic Keyword to Pass Arguments** The usefulness of the DLR becomes even more obvious when you need to make late-bound calls on methods that take parameters. When you use “longhand” reflection calls, arguments need to be packaged up as an array of objects, which are passed to the Invoke() method of MethodInfo.

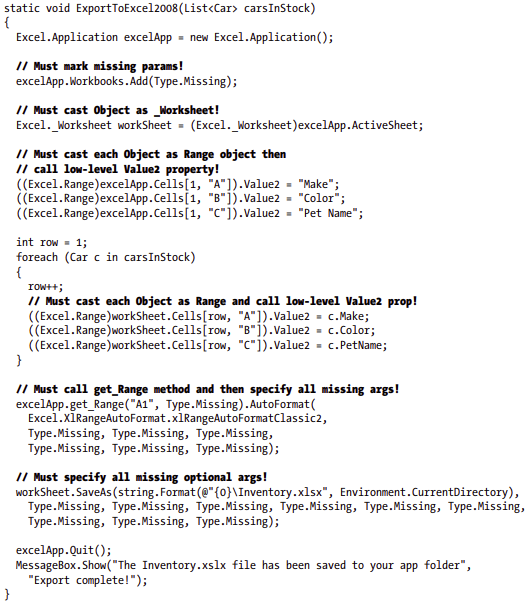
**Simplifying COM Interoperability** Using Dynamic Data Let’s see another useful case for the dynamic keyword within the context of a COM interoperability project. Now, if you don’t have much background in COM development, do be aware for this next example that a compiled COM library contains metadata, just like a .NET library; however, the format is completely different. Because of this, if a .NET program needs to communicate with a COM object, the first order of business is to generate what is known as an interop assembly (described in the following paragraphs). Doing so is quite straightforward. Just activate the Add Reference dialog box, select the COM tab, and find the COM library you want to use (see Figure 16-4).

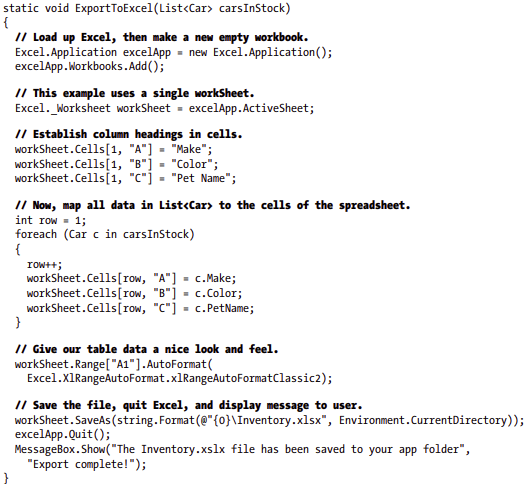
**The Role of Primary Interop** Assemblies Many COM libraries created by COM library vendors (such as the Microsoft COM libraries that allow access to the object model of Microsoft Office products) provide an “official” interoperability assembly termed a primary interop assembly (PIA). PIAs are optimized interop assemblies, which clean up (and possibly extend) the code typically generated when referencing a COM library using the Add Reference dialog box

**Common COM Interop Pain Points** Let’s cover one more preliminary topic before the next example. Before the release of the DLR, when you authored C# code that used a COM library (via the interop assembly), you were sure to face a number of challenges. For example, many COM libraries defined methods that took optional arguments, which were not supported in C# until .NET 3.5

**COM Interop Using C# Dynamic Data** Assume you have a Windows Forms GUI application (named ExportDataToOfficeApp), whose main window hosts a DataGridView control named dataGridCars. This same window has two Button controls, the first of which will bring up a custom dialog box to insert a new row of data to the grid, and the other of which will export the grid’s data to an Excel spreadsheet. Figure 16-8 shows the completed GUI.

Because you imported the COM library using Visual Studio, the PIA has been automatically configured so that the used metadata will be embedded into the .NET application (recall the role of the Embed Interop Types property). Therefore, all COM Variants are realized as dynamic data types. Furthermore, you can use C# optional arguments and named arguments. T

**COM interop Without C# Dynamic Data** Now, if you were to select the Microsoft.Office.Interop.Excel.dll assembly (in Solution Explorer) and set its Embed Interop Type property to False, you would have new compiler errors, as the COM Variant data is no longer realized as dynamic data but as System.Object variables. This will require you to update ExportToExcel() with a number of explicit casting operations.

Although the end result of running this program is identical, this version of the method is much more verbose, as I am sure you agree. That wraps up your look at the C# dynamic keyword and the DLR. I hope you can see how these features can simplify complex programming tasks and (perhaps more importantly) understand the trade-offs. When you opt into dynamic data, you do lose a good amount of type safety, and your code base is prone to many more runtime errors. While there is certainly more to say about the DLR, this chapter has tried to focus on topics that are practical and useful in your day-to-day programming. If you want to learn more about advanced features of the Dynamic Language Runtime, such as integrating with scripting languages, be sure to consult the .NET Framework 4.6 SDK documentation (look up the topic “Dynamic Language Runtime Overview” to get started).

**Summary** The dynamic keyword introduced in C# 4.0 allows you to define data whose true identity is not known until runtime. When processed by the new Dynamic Language Runtime, the automatically created “expression tree” will be passed to the correct dynamic language binder, where the payload will be unpackaged and sent to the correct object member. Using dynamic data and the DLR, a number of complex C# programming tasks can be radically simplified, especially the act of incorporating COM libraries into your .NET applications. As you have also seen in this chapter, .NET 4.0 and higher provides a number of further simplifications to COM interop (which have nothing to do with dynamic data), such as embedding COM interop data into your applications, optional arguments, and named arguments. While these features can certainly simplify your code, always remember that dynamic data makes your C# code much less type safe and open to runtime errors. Be sure you weigh the pros and cons of using dynamic data in your C# projects, and test accordingly!