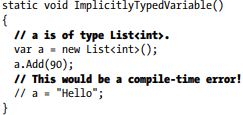
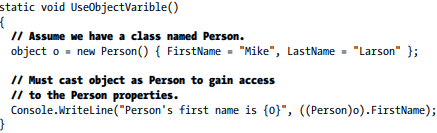
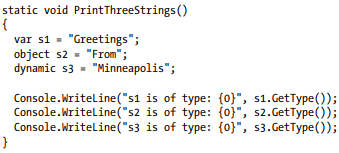
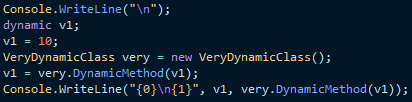
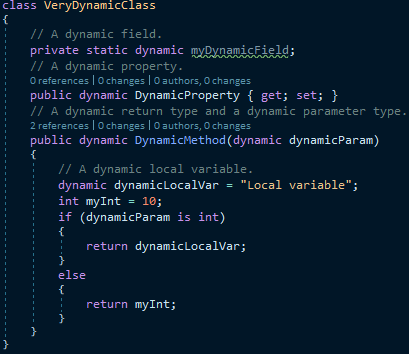
**Dynamic Types and the Dynamic Language Runtime**

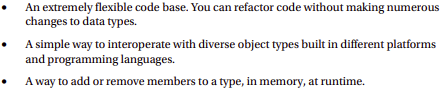
NET 4.0 introduced a new keyword to the C# language, specifically, dynamic. This keyword allows you to incorporate scripting-like behaviors into the strongly typed world of type safety, semicolons, and curly brackets. Using this loose typing, you can greatly simplify some complex coding tasks and also 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.

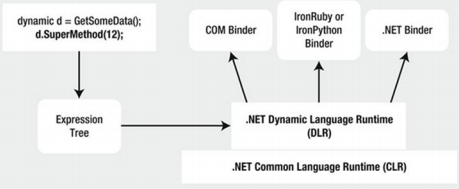
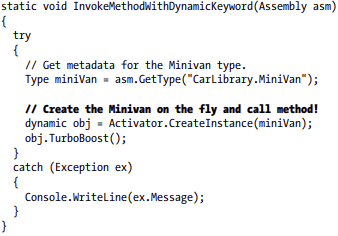
**The Role of the C# dynamic Keyword** Back in Chapter 3, you learned about the var keyword, which allows you to define local variables in such a way that the underlying date type is determined at compile time, based on the initial assignment (recall that this is termed implicit typing). Once this initial 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).