Introduction to C# for VFP Developers

Doug Hennig
Stonefield Software Inc.
2323 Broad Street
Regina, SK Canada S4P 1Y9
Email: dhennig@stonefield.com
Web site: www.stonefieldquery.com
Blog: DougHennig.BlogSpot.com
Twitter: DougHennig

Even if you're planning to continue development in VFP, learning another language like C# can be very useful. There are some things that are a lot easier and faster to do in .Net than in VFP (the reverse is also true). This document introduces the C# language, comparing it to constructs and syntax in VFP to shorten the learning curve.

Introduction

Many VFP developers, myself included, have resisted learning .Net languages like C# because we thought it meant replacing the platform we've known and loved for years. Certainly, some of the well-known VFP developers who adopted .Net early have moved permanently to that platform, which reinforced that view. However, I've come to realize that, just as with a human language, learning another computer language doesn't necessarily mean you're abandoning the one you use daily. Some VFP gurus, such as Rick Strahl, Paul Mrozowski, and Craig Boyd, work in both environments. While I've been learning and using C# over the past couple of years, I still use VFP daily.

Also, again as with a human language, learning another computer language makes you more versatile, employable, and able to solve problems your original language may not be able to solve.

This document is intended to give you, a VFP developer, an introduction to the C# language. We'll compare language features between VFP and C# and explain concepts like string data typing, case sensitivity, enumerations, and other things that have no VFP equivalent.

Rick Strahl likes to say that .Net makes the hard things easy and the easy things hard. You'll find that accessing data in .Net is certainly different and more work than we're used to in VFP but other things are considerably easier and in many cases, much faster in .Net.

One of the things .Net does better than VFP is object creation and destruction. For example, suppose you want to create a collection of objects from the records in a table. Code like this takes care of that:

```
use PEOPLE
loPeople = createobject('Collection')
lnStart = seconds()
scan
   loPerson = createobject('Person')
   scatter name loPerson additive
   loPeople.Add(loPerson)
endscan
lnEnd = seconds()
messagebox(transform(lnEnd - lnStart) + ' to create ' + transform(loPeople.Count) + ;
   ' objects')

define class Person as Custom
   FirstName = ''
   LastName = ''
   Phone = ''
   City = ''
   Region = ''
   PostalCode = ''
   Email = ''
enddefine
```

This code does something similar in C#:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics;
using System.Data;

namespace TestPerformance
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create a collection of Person objects.
            List<Person> people = new List<Person>();

            // Read people.xml into a DataTable.
            DataSet peopleData = new DataSet();
            peopleData.ReadXml("people.xml");
            DataTable peopleTable = peopleData.Tables["people"];

            // Create a person object for every person in the XML and add them to the
            // collection.
            Person person;
            Diagnostics diagnostics = new Diagnostics();
            diagnostics.Start();
            foreach (DataRow row in peopleTable.Rows)
            {
                person = new Person();
                person.FirstName = (String)row["FirstName"];
                person.LastName = (String)row["LastName"];
                person.Phone = (String)row["Phone"];
                person.City = (String)row["City"];
                person.Region = (String)row["Region"];
                person.PostalCode = (String)row["PostalCode"];
                person.Email = (String)row["Email"];
                people.Add(person);
            }

            // Stop the watch and show the results.
            diagnostics.End();
            TimeSpan actualTime = diagnostics.ElapsedTime;
            System.Windows.Forms.MessageBox.Show(String.Format(
                "{0} to create {1} objects", actualTime.ToString(),
                people.Count.ToString()));
        }
    }

    public class Person
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Phone { get; set; }
        public string City { get; set; }
        public string Region { get; set; }
```

```
        public string PostalCode { get; set; }
        public string Email { get; set; }
    }
}
```

There's a lot more C# code because it doesn't have native data access built into the language. However, the performance difference is astounding: on my system, 0.433 seconds to create 64,000 objects in C# versus 44.164 seconds in VFP. That's more than 100 times faster.

In case you think this is a contrived example, this is exactly the kind of thing I've wanted to do in Stonefield Query for years: expose a data dictionary as a collection of meta data objects. Because the performance of creating a lot of objects is so slow in VFP once you've gone past a certain threshold, I had to write a convoluted mechanism that used a collection façade to a cursor. Even that workaround can't load a 64,000-item data dictionary in half a second.

Note that I don't go over user interface, such as forms or Web pages, in this document. It's strictly a language discussion.

Why C#

If you're interested in working with a .Net language, you have several to choose from besides C#, including the popular Visual Basic (VB). The language you choose doesn't matter that much in the grand scheme of things, as every language uses the .Net framework and its thousands of classes. The reason I chose C# are:

- It seems like a "clean" language. Although VB is closer to VFP in syntax, it also seems quite wordy to me.
- There appear to be more examples available in C# than VB.
- C# is very similar in syntax to JavaScript, so learning one gives a big head-start to learning the other. Since JavaScript is the lingua franca of web applications, this is no small benefit.
- VB developers I know claim to be the "red-haired stepchildren" of the .Net world. Although that certainly a place we VFP developers know well, I thought it might be fun to hang out with the cool kids for a change <g>.
- The developers in my shop were already familiar with C#, so it seemed better for just one of us to learn a new language than all three.

VFP's language based on hundreds of commands and functions whereas C# has very few of the former and none of the latter. Instead, it has the basic language constructs and uses the .Net framework for everything else. The .Net framework consists of thousands of built-in classes. In fact, the hardest part of learning any .Net language is learning the classes (although you certainly don't have to learn them all). That means that switching from one .Net language to another is relatively easy since most differences are minor syntactical ones.

The language you choose is a personal decision and you probably won't go wrong with either one. However, I assume you're interested in C# or you wouldn't be reading this.

Visual Studio

Although you can write programs in Notepad and use the command-line C# compiler to compile the code, why would you want to? Microsoft Visual Studio (VS) is a complete interactive development environment (IDE), similar to but considerably more powerful that VFP's. Some of its features are:

- Solution Explorer: similar to the VFP Project Manager but able to manage multiple projects at a time.
- Code editor: syntax coloring, IntelliSense, breakpoints, and all of the features you'd expect in a modern editor.
- Debugging tools.
- Automated testing tools.
- Advanced search capabilities.

Depending on the version, VS can be quite expensive. Fortunately, Microsoft provides several free versions, called "Express." For example, Visual Studio C# Express is a free download that allows you to create C# applications. If you don't already have VS, download one of the Express versions to learn C# and then consider using a full, paid version when you need the more powerful features for professional development. See http://www.microsoft.com/visualstudio for details.

Before we get started, let's discuss a few things, starting with solutions and projects. Like VFP, VS has projects. Each project creates a single EXE or DLL. Solutions allow you to make your application more modular: different components can go into separate projects and therefore separate DLLs. This makes these projects more reusable and easier to update a client site: just install one small DLL rather than a large EXE.

To create a new project, choose New, Project from the File menu. The New Project dialog shown in **Figure 1** appears. A project starts from a template, which defines the type of application being built and determines what settings the project should default to. You can chose the language and template for the project from the dialog, specify the project name, the folder, and whether or not to create a new solution or add the project to the existing solution if there is one. For our purposes, we'll create a Console Application, which simply outputs text to a console window.

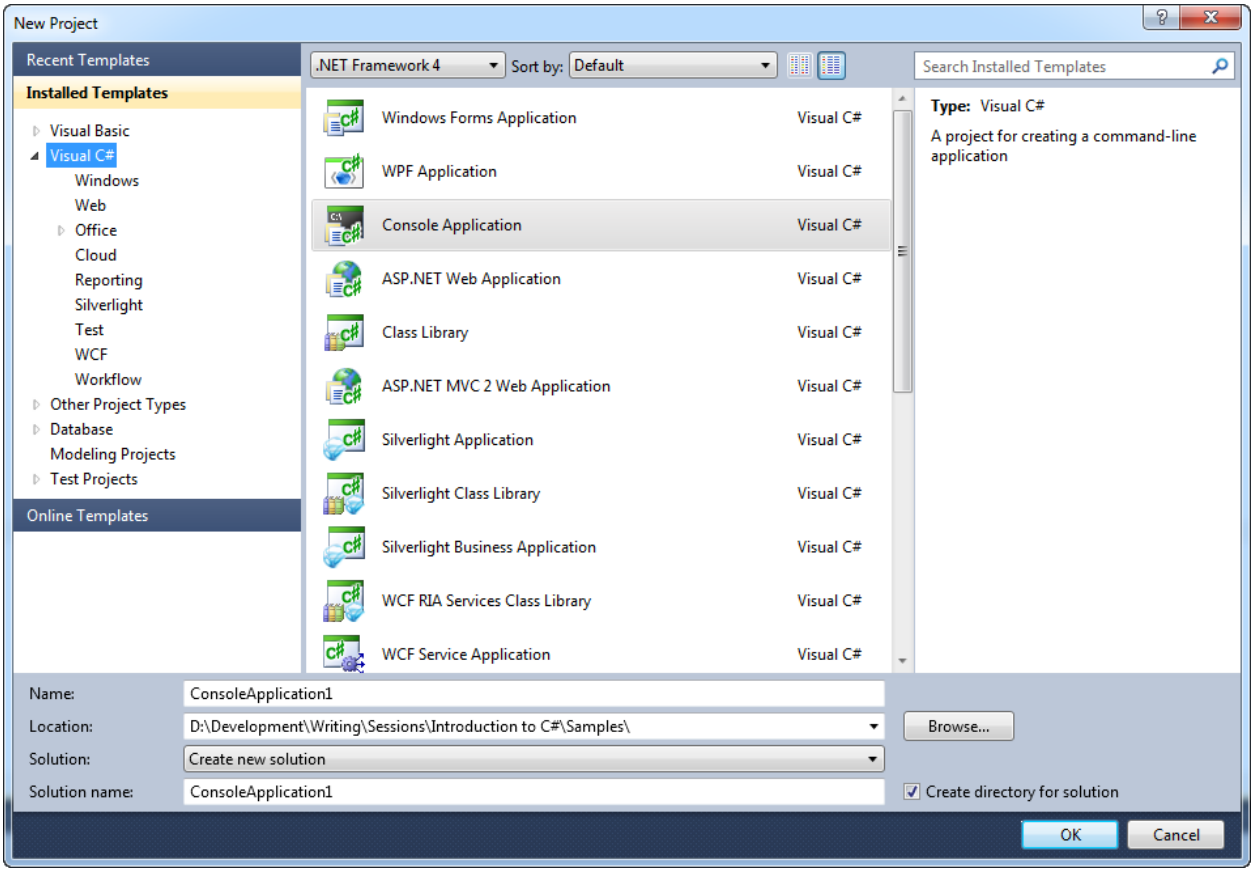

Figure 1. The New Project dialog allows you to choose a project template.

C# code goes into files with a CS extension. They're the C# equivalent of PRG files in VFP. A CS file can contain one class (the preferred way) or many classes. A common and sensible convention is that the CS file is named the same as the class defined in the file.

When you create a new project, VS automatically creates a starting CS file named Program.CS. You can rename the CS file if you wish; doing so automatically renames the stub class VS generated in the file as well.

Our first program

The traditional first program for a new language is one that simply outputs "hello world," so let's not break tradition. To the code that VS automatically generated, add the `Console.WriteLine` statement shown below:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace FirstApp
{
   class Program
   {
```

```
        static void Main(string[] args)
        {
            // Say hello to the world.
            Console.WriteLine("Hello World");
        }
    }
}
```

Run the program by pressing F5 or by choosing Start Debugging from the Debug menu. A console window flashes and disappears. That's because, like a VFP program with modeless forms and no READ EVENTS statement, we didn't tell the program to stop to give us a chance to see the window. Add the following below the `Console.WriteLine` statement:

```
Console.ReadKey();
```

Run it again. This time you should a console window displaying "Hello World," as shown in **Figure 2**.

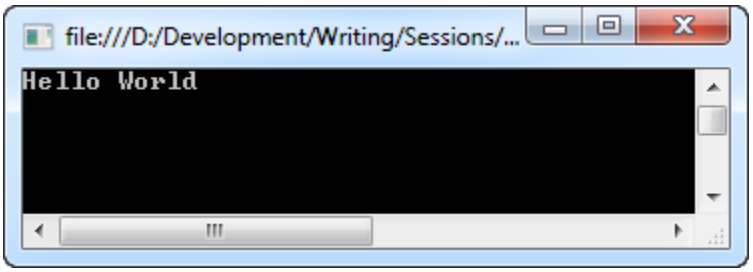

Figure 2. Running our first application.

Breaking it down

Let's break down our program to see what it tells us about C#.

The first four statements aren't executable code but rather tell the C# compiler that we may be using classes in the System, System.Collections.Generic, System.Linq, and System.Text namespaces. Before we talk about what the `using` statement does, let's talk about namespaces.

Namespaces

VFP doesn't have the concept of namespaces. A namespace is simply a way to logically combine classes. Note that a namespace is not a physical mechanism like a VCX file; classes in different files or even different projects can be in the same namespace.

One of the purposes of namespaces is to avoid class name conflicts. As a tool developer, this is something I've run into in VFP. For example, one of the VFPX tools I've contributed to, PEM Editor, has a class library named BaseControls.VCX that contains subclasses of the VFP base classes, with names like BaseCheckbox and BaseTextBox. Unfortunately, some developers also use classes with those names, and run into problems when using PEM Editor with them.

Namespaces in .Net avoid this problem because the fully qualified name for each class uses its namespace as a prefix. For example, the full name for our class is FirstApp.Program because it's in the FirstApp namespace. As you can see in our program, creating your own namespace is simply done using the `namespace` keyword.

Namespace names can be multiple levels. Notice in the `using` statements, one namespace is System and another is System.Text. This doesn't mean there's containership or physical hierarchy or anything; it's just a naming convention Microsoft uses. Although you can use any naming system you want, one frequently-used convention is CompanyName.ProjectName.ModuleName. For example, classes in the accounts receivable module in the Pharmacy Accounting application created by ABC Software Inc. might be in the ABCSoftware.PharmacyAccounting.AccountsReceivable namespace.

Like VFP, .Net has certain rules about what's acceptable for names. The first character must be a letter or underscore and the rest can be letters, digits, and underscores. (Actually, the rules are more complicated than that because Unicode characters are allowed; see http://tinyurl.com/cy8k64n for the full specification.) This doesn't just apply to namespace names, but all names in .Net.

The project templates automatically add a namespace declaration to each new program file, using the default namespace for the project, which is the name of the project. You can, of course, edit the `namespace` statement to use a different name. You can also change the default namespace for the project by right-clicking the project in Solution Explorer, choosing Properties, and editing the *Default namespace* setting (**Figure 3**).

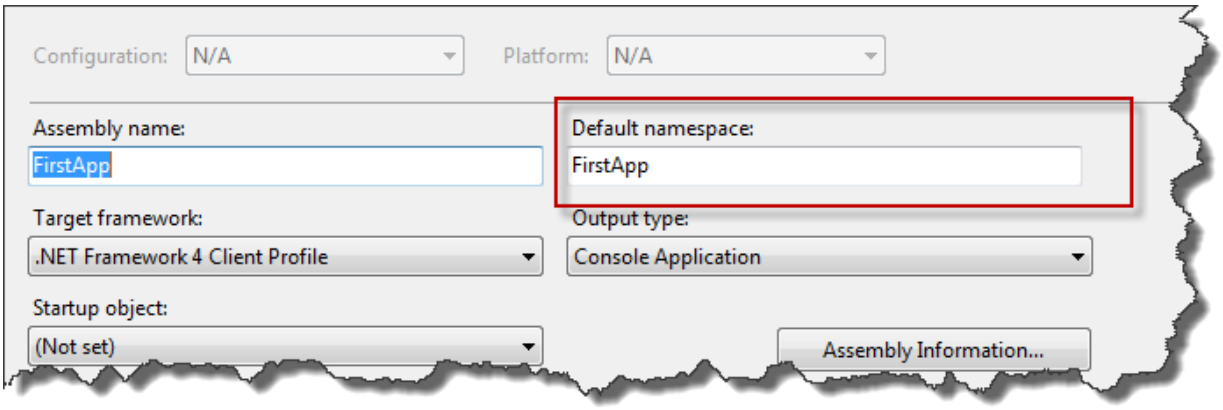

Figure 3. You can change the default namespace for the project.

Using

As I mentioned earlier, the full name for a class includes its namespace. So, unless you tell the compiler otherwise, you need to specify, for example, System.Console rather than just Console. However, it would kind of be a pain to have to type the full name every time you want to use a class. That's where the `using` statement comes in.

`using` allows you to reference the classes in the specified namespace without having to include the namespace with the name. In that regard, it's similar to VFP's `SET CLASSLIB TO` statement, which allows you to instantiate a class without specifying the class library name.

To see the effect of `using`, comment out the first `using` statement by adding "//" at the start of the line. Notice that the two instances of "Console" now have a red squiggly line under them and if you hover your mouse over them, you see the message "The name 'Console' does not exist in the current context." That's because the full name for Console is System.Console and without the `using` statement, the compiler can't find a class named Console. You can fix the error by changing "Console" to "System.Console."

Different class templates generate different `using` statements. The template for a console application main program generates the four `using` statements we see in our code, but a Windows Presentation Foundation (WPF) form has a lot more. You can, of course, add your own `using` statements as necessary.

Related to `using` is the concept of project references. In VFP, if you want to use a class in a VCX in an application, you have to include that VCX in the project used to create the application. .Net is similar: if your code uses any classes not defined in the project itself, you must add references to the assemblies (DLLs) containing those classes. To do that, right-click the References node in the Solution Explorer, choose Add Reference, and select the desired assembly in the dialog that appears.

Curly braces

If you've worked in C++, Java, JavaScript, or similar languages, you've used curly braces before. If not, they may seem a little foreign. Essentially, curly braces are used to delimit logical blocks. A block could be the content of a namespace, a class, a method, a loop structure, and so on. In our sample program, we see a curly brace after the namespace statement and the matching one at the end of the code. Everything between them belongs to the namespace. In this case, there's only one class, but there could be multiple classes. Similarly, there are curly braces to contain the members of the Program class and the code of the Main method.

Matching curly braces should be at the same indentation level and everything inside them should be one indent position to the right to make it obvious what goes inside each block. VS automatically handles indentation for you, although you can override it.

One important thing to know about code blocks is that variables declared within them are scoped to that block and do not exist outside of it. For example, this code gives a compiler error on the last statement because the variable message doesn't exist outside the code block that defines it:

```
{
    string message = "This is a message";
}
Console.WriteLine(message);
```

Curly braces seem like kind of a pain at first, but you'll get used to them.

Class definition

Defining a class is very straightforward: use the `class` keyword, followed by the class name. There are actually more options for the `class` statement, including visibility and parent class specifications, but we'll see those later. Notice that since we didn't specify a parent class, this class automatically inherits from System.Object, the base class for all objects in .Net.

Method definition

Like a class definition, a method definition is straightforward: the return data type, the method name, and the list of parameters in parentheses. If there's no `return` statement in a method, the return data type must be specified as `void`, which is the case of our Main method. Method definitions can also have other options, such as visibility and, in this case, the `static` keyword. We'll discuss some of those later.

Parameters are specified as the data type followed by the parameter name, and like VFP are separated by commas. In our Main method, the single parameter is an array of strings (the square brackets specify an array) named "args;" the Main method of a console application is automatically passed any command line arguments as this array.

The code

The first line in the Main method is a comment. Comments can either be single line (starting with "//") or multi-line: everything between "/*" and "*/" is a comment.

There's only one executable line of code in our program, the `Console.WriteLine` statement. `Console.WriteLine` is like the `?` command in VFP: it outputs the specified text to the console window.

Case sensitivity

There are a few things to note about this simple statement:

- Unlike VFP, which accepts single quotes, double quotes, and square brackets as string delimiters, C# uses only double quotes.
- Unlike VFP, code statements in C# can span more than one line without specifying a line continuation character. As a result, code must have a statement termination character, which is the semi-colon. Non-executable code, such as the class definition, doesn't use them. As with curly braces, semi-colons seem like a pain but you'll quickly get used to them.
- Although it isn't apparent in this code, C# is case-sensitive. That means the exact case of a class name, keyword, namespace, and so forth must be used or you'll get a compile error. For example, try changing the "L" in WriteLine to lower-case. You'll see a red squiggly line under "Writeline" and a "'System.Console' does not contain a definition for 'Writeline'" error in the Error List window (if you have it open). While

this seems very awkward compared to case-insensitive languages like VFP, in practice it isn't much of a problem because you'll normally type the first few letters of a class name or keyword and let IntelliSense fill in the rest for you.

- Unlike VFP, C# doesn't have any functions. Instead, all code has to go into methods of classes. So, notice something interesting about this line of code: we didn't instantiate an object, and yet we're calling the method of a class (the WriteLine method of System.Console). How can that be? To understand that, we need to discuss the difference between a class and an object.

Classes vs. objects

Christof Wollenhaupt has mentioned in some of his presentations that .Net is more class-oriented than VFP, which is more object-oriented. It took me a while to grasp exactly what this means, so I'll try to explain it.

In VFP, we tend to think of classes and objects as the same, with the class definition simply being a template for objects instantiated from it. The first time you instantiate a class, VFP loads the class template into memory, then creates an object from that template. Subsequent instantiations reuse the loaded template. However, we have no access to the class at runtime other than being able to create objects from it. With one exception, nothing really belongs to the class, just to the instantiated objects. The one exception is any property set to an expression in the class definition rather than in code. For example, consider this class:

```
define class MyClass as Custom
   tUpdated = {/:}
   tCreated = datetime()

   function Init
      This.tUpdated = datetime()
   endfunc
enddefine
```

If you instantiate two instances of the class some seconds apart, you would expect that the values of tUpdated for those two instances are different. However, you might be surprised to see that the values for tCreated are the same and equal to the time the first object was instantiated. That's because when VFP loads the class template, it evaluates expressions for properties at that time and makes the resulting values part of the template. So, instantiating an object from the template results in the object having the date and time the template was loaded rather than when the object was created.

In .Net, a class is more than just a template; it can have members that belong to it rather than objects instantiated from it. You can even create a class that can't be instantiated at all, but is used directly anyway.

This is done using the `static` keyword, either at the class level (meaning the class can't be instantiated) or the member level. From what we've seen, we can conclude that the WriteLine method of System.Console must be a static method. That means it belongs to the

class, so rather than instantiating the class into an object, we call the WriteLine method of System.Console directly. In this regard, the static members of a class act like VFP functions, since they're called without instantiating an object as well. The Main method of our Program class is also defined as static, so it can be called as Program.Main rather than SomeObjectInstantiatedFromProgram.Main.

Having a non-static class with some static and some non-static members is interesting. It means the class can be instantiated, but the static members can't be referenced as members of the object and the non-static members can't be referenced as members of the class. For example, we can instantiate the Program class into the variable X but typing "X." in VS won't show Main as a member in IntelliSense.

Building the project

As you saw earlier, you can press F5 to run the application. You can also choose Start Without Debugging from the Debug menu or press Ctrl-F5. When you start the application, VS first compiles it into an EXE. If the code has any errors, the compilation stops and VS displays a message that there were build errors and the Error List window shows the error. If the build was successful, VS then runs the EXE.

You can also build the project without running it by choosing Build Solution or Build *ProjectName* from the Build menu.

There are two types of builds: debug and release. The differences between the two are that in debug mode, the compiler doesn't optimize the binary it produces whereas it does in release mode, and while both builds generate a PDB file which contains data to make debugging easier, the debug version contains more information. You can change the type of build from the build combobox in the toolbar or in the Build page of the Properties for the project.

The binary files are generated in either the Debug or Release subdirectory, depending on the build type, of the bin subdirectory of the project folder. In addition to the EXE (or DLL, depending on the type of project) and PDB files, VS also generates *projname*.vshost.exe and *projname*.vshost.exe.manifest files, where *projname* is the name of the project. These are VS hosting process files which assist in the debugging process. You don't distribute those or the PDB file to your users.

Notice the small size of the EXE: 5K on my system. Compare that to the smallest VFP version (a single PRG with MESSAGEBOX('Hello world') and Debug Info turned off in the Project Information dialog), which is 24K. The C# compiler generates IL (intermediate language) code, which is converted to machine language when the application is run. Thanks to the optimized code and the .Net framework, EXEs are small.

Note that unlike VFP, you can't run just a "program" (that is, one cs file). Instead, you have to run the application. However, you can perform unit tests on a class. Unit testing is beyond the scope of this document.

Using another method

Rather than making our main method do all the work, let's move the code to another method, then call that method from Main. You could do this manually by creating a new method, cutting and pasting the code from Main to the new method, and then adding a call to the new method in Main, but VS has a built-in mechanism to automate all of that.

Select the two Console statements, right-click, choose Refactor, then choose Extract Method. In the Extract Method dialog, type the name of the new method (in this case, use Hello), and click OK. The resulting code should be (omitting the `using` and `namespace` statements):

```
class Program
{
    static void Main(string[] args)
    {
        HelloWorld();
    }

    private static void HelloWorld()
    {
        Console.WriteLine("Hello");
        Console.ReadKey();
    }
}
```

Let's see what's new here:

- Although C# has a `this` keyword, unlike VFP, calling a method of the same class doesn't require it as a prefix to the method name. Since C# doesn't have any functions, a method name by itself, like HelloWorld() here, must be a method of the class.
- You might think you can add "this." in front of the call to HelloWorld, and in a different example that would work, but doing that here causes a "Keyword 'this' is not valid in a static property, static method, or static field initializer" error. That's because `this` means a reference to the instantiated object. Since static methods belong to the class and not the object, `this` doesn't apply here.
- The code generated for HelloWorld has a visibility scope specified: the `private` keyword. That means this method can only be called from other methods in this class, similar to how the `protected` keyword in VFP works. **Table 1** shows the four visibility keywords in C#. One difference with VFP is that in VFP, the default visibility is `public` unless otherwise specified, whereas in C#, the default is `private`.

Table 1. The visibility keywords in C#.

Keyword	Description
public	The class or member can be accessed by anything. This is similar to the VFP `public` keyword.
private	The member can only be accessed by methods in the same class (not even

	subclasses can access it). This is similar to the VFP `hidden` keyword.
protected	The member can only be accessed by methods in the same class or a subclass. This is similar to the VFP `protected` keyword.
internal	The class or member can only be accessed by other classes in the same project. There is no VFP equivalent.

Using another class

Let's create a new class and use it from our Program class. Right-click the project in Solution Explorer, choose Add, then Class, enter the desired name of the program file, and click OK. Edit the code so it's like this (replacing the class name with the name you specified):

```
class AnotherClass
{
    public string Hello(string name)
    {
        string returnValue = "Hello, " + name;
        return returnValue;
    }
}
```

Here's what's new:

- Hello has the public keyword or it can't be used from other classes.
- It returns a string, so the return data type in the method definition is "string."
- It accepts a single string parameter called name.
- It creates a new variable of type string and assigns a value to it, then returns the value.

Let's use this class in our Main method. Add the following code below the call to HelloWorld:

```
AnotherClass other = new AnotherClass();
string result = other.Hello("Doug");
Console.WriteLine(result);
Console.ReadKey();
```

The new concept here is, pardon the pun, the `new` keyword. This is the equivalent of VFP's `CREATEOBJECT()` function: it instantiates a class and returns the reference to the instantiated object. The syntax looks a little odd—specifying AnotherClass twice—but that's because of strong data typing, discussed in the next section: the object reference has to be stored in a variable of the correct data type. You can actually use a shortcut if you wish: the `var` keyword. `var` doesn't mean the variable is a dynamic type; it just means that the compiler should infer the data type so you don't have to. Here's what the code would look like with `var`:

```
var other = new AnotherClass();
```

Press F5 to run the application, and you should see “Hello World”. Press Enter and “Hello, Doug” appears. Press Enter again to close the console window.

Strong typing

One thing we’ve seen is the specification of data types for variables, parameters, and return values. In VFP, this is optional via the “as” clause in a LOCAL statement, but that’s for IntelliSense rather than enforcement because VFP is a weakly-typed (sometimes called “dynamic”) language. For example, the following code is perfectly legal:

```
local lcValue as String
lcValue = 'Test'
lcValue = 5
```

However, C# is a strongly-typed language, meaning you must specify data types and stick by them. To see the effect of this, try the following:

- Change the return type of Hello to int.
- Change the data type of the name parameter to int.
- After changing the data type for name back to string, change Program.cs so the parameter passed is “5.”

In each case, you’ll get an error because the data type doesn’t match the expected. This is actually a good thing: you wouldn’t catch errors like this in VFP until runtime whereas it’s the C# compiler that catches them (that is, design time).

There are pros and cons to both approaches:

- Weak typing is very flexible. You can create code in VFP that works with data types that aren’t known at design time but figured out at runtime. It’s very difficult to do that in C#.
- Weak typing is a very common source of errors in VFP. Using a variable containing a numeric value in an expression that expects it to be a string (for example, the VAL function) causes a runtime error. As a result, there’s a heavier burden on testing all possible cases in such code to avoid the user running into them. As mentioned earlier, in C#, this causes a compiler error so the code won’t even run until fixed.

Data types

There are two types of data types in .Net: value and reference. Value types store the actual value and reference types store a reference to a value stored elsewhere. (See http://tinyurl.com/7cq6pkq for a discussion of how .Net internally stored values types and reference types.) This has several consequences:

- Assigning one value type variable to another copies the value, so changing the value in the second variable doesn’t affect the first one. On the other hand, assigning one reference type variable to another copies the reference, so both variables point to

the same object. This is similar to the idea of passing values by reference or value in VFP.

- You cannot subclass a value type but you can subclass a reference type.
- Value types can't be null, although you can use a special feature called nullable types to support this. A reference type that's null simply means the reference is null, so it doesn't point to anything.

The data types built into C# are discussed in the following sections. All are value types except string.

bool

bool is the equivalent of the VFP Logical data type, containing the values true and false (in all lower-case).

char

char is a single 16-bit Unicode character. Literal values can be assigned in one of these ways:

```
char char1 = 'Z';        // Character literal; uses single quotes
char char2 = '\x0058';   // Hexadecimal
char char3 = (char)88;   // Cast from integral type
char char4 = '\u0058';   // Unicode
```

char is similar to a single VFP character, although it's 16-bit rather than VFP's 8-bit. char isn't used nearly as often as string.

string

string is a reference type storing zero or more Unicode (16-bit) characters. Strings in .Net are immutable, meaning they can't be changed. This means changes you make to a string actually create a new object rather than updating the existing one. For example, consider the following code:

```
string message = "This is a message";
message += " with more text";
```

In the first line, the message variable points to a string object. A second string is concatenated to that string object (the += operator is an abbreviation for Variable = Variable + SomeValue) and the result stored in a third object, which the message variable now points to. The first string originally pointed to by message is still is memory but is no longer pointed to by anything, and will be removed from memory the next time the .Net garbage collector does its thing. (Garbage collection is an important subject but beyond the scope of this document.)

Because of the immutable nature of strings, concatenating a lot of strings together, such as in a loop, can create many string objects in memory:

```
string message = "";
for (int i = 0; i < 255; i++)
{
    message += (char)i;
}
```

(We'll discuss the syntax of the `for` loop later, but for now, this is the equivalent of FOR I = 0 TO 255.)

A better solution is to use a StringBuilder object to build up the string, then copy the result to a string variable, because this avoids a large number of small strings in memory:

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 255; i++)
{
    sb.Append((char)i);
}
string message = sb.ToString();
```

string acts like an array of char, so you can use code like the following to access individual characters (remember that C# is zero-based):

```
string message = "This is a message";
char letter = message[0]; // "T"
```

String literals are delimited by double-quotes and can contain escape sequences. An escape sequence is a special character that can't easily be entered from the keyboard, such as "\t" for tab, "\n" for a new line, or "\\" for a backslash (see http://tinyurl.com/7co2zn4 for a complete list of the escape sequences). Here's an example that has a newline character in the middle:

```
string message = "This is the first line\nThis is the second one";
```

If you need to use a backslash character as part of the string, prefix the literal with "@," which tells .Net to not evaluate escape sequences in the string. This is commonly used when specifying a path, such as:

```
string path = @"C:\MyFolder\";
```

To include a double-quote in a string, use "@" and double the quote mark:

```
string message = @"""Help,"" he said";  // "Help," he said
```

This looks a little confusing, but here's the explanation: the first quote starts the string, the next two provide a quote, the next two provide another quote, and the last one ends the string.

decimal

decimal is a 128-bit data type. It has more precision (28-29 significant digits) and a smaller range ($\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$) than float or double, so it's frequently used for financial

calculations. It's the equivalent of the VFP Currency data type, although Currency is only 64-bits.

To assign a literal numeric value to a decimal, specify "M" or "m" as a suffix; otherwise, the compiler assumes the value is double (see the next section) and gives an error because you're trying to assign it to a decimal variable. Here's an example:

```
decimal amount = 100.5M;
```

double

double is a 64-bit floating point value. It has 15-16 digits of precision and a range of ±5.0 × 10^{-324} to ±1.7 × 10^{308}. It's the .Net equivalent of the VFP Double data type.

An expression containing a double and some other numeric data type evaluates to double. A literal numeric value is automatically assumed to be double, but to assign an integer literal to a double, use "D" as a suffix. For example:

```
double d1 = 100.5;
double d2 = 10D;
```

float

float is similar to double, but is only 32-bits, so its precision (7 digits) and range (±1.5 × 10^{-45} to ±3.4 × 10^{38}) are smaller. It's the equivalent of the VFP Numeric or Float data types, although Numeric and Float have a smaller range even though they're stored internally as 64-bits.

A numeric expression that does not contain a double evaluates to float. To assign a literal numeric value to a float, specify "F" or "f" as a suffix; otherwise, the compiler assumes the value is double (see the previous section) and gives an error because you're trying to assign it to a float variable. Here's an example:

```
float amount = 100.5F;
```

int

int is a signed 32-bit integer with a range of -2,147,483,648 to 2,147,483,647. It's the equivalent of the VFP Integer data type.

long

long is a signed 64-bit integer with a range of –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. You can assign a literal value to a long by either specifying a value outside the range of int or specifying "L" as a suffix:

```
long amount = 150L;
```

byte, sbyte, short, uint, ulong, and ushort

These integer data types aren't used often in native C# code but more commonly with COM or Windows API functions that need these data types. **Table 2** has a list of these data types and their attributes.

Table 2. Less-commonly used integer data types in C#.

Data type	Size (bits)	Signed	Range
byte	8	No	0 to 255
sbyte	8	Yes	-128 to 127
ushort	16	No	0 to 65,535
short	16	Yes	-32,768 to 32,767
uint	32	No	0 to 4,294,967,295
ulong	64	No	0 to 18,446,744,073,709,551,615

Arrays

An array isn't a specific data type, but sort of fits in here because it's a collection of values of a particular data type. To create an array, specify a data type followed by square brackets. Note that you can't create an array of more than one data type.

For example, this code creates an array of int values:

```
int[] values = new int[3];
values[0] = 10;
values[1] = 7;
values[2] = 12;
```

values is defined as a three-element array of int and the elements are filled with literal values. Note that C# is zero-based, so the first array element is 0 and the last one is the number of elements minus one.

C# allows you to initialize the contents of an array in the same statement that defines it by specifying a comma-delimited list of values between curly braces. The following statement replaces the four earlier ones:

```
int[] values = new int[3] { 10, 7, 12 };
```

Arrays have several built-in members you can use, including:

- Length returns the number of elements in the array, like the VFP ALEN() function.
- Rank returns the number of dimensions in the array.
- Min() and Max() return the minimum and maximum values stored in the array.
- Sum() returns the sum of the elements of the array if they're numeric.

Data type conversion

There are times when you need to convert from one data type to another. C# provides both implicit and explicit conversion.

Implicit conversion means C# automatically converts a value from one type to another. Implicit conversions are allowed when the compiler can guarantee they will always succeed and no data is lost. For example, you can store the value of an int variable into a float variable because the range of values for float encompasses that of int. However, you can't go the other way because it's possible a double value is outside the bounds of int, and certainly any places after the decimal would be lost.

Explicit conversion means you specify how to convert a value from one type to another. There are numerous ways to do this:

- You can cast the value by prefixing it with the target data type in parentheses, such as:

```
int i = 10;
string s = (string)i;
```

- You can use the ToString method that all objects have to convert the value to a string:

```
int i = 10;
string s = i.ToString();
```

- You can use the Parse or TryParse methods of a type (the difference is that Parse throws an exception if it fails):

```
string s = "10";
int i = int.Parse(s);
int j = int.TryParse(s);
```

- You can use one of the methods of the Convert class:

```
string s = "10.5";
decimal d = Convert.ToDecimal(s);
```

Fields and properties

One of the things that makes learning .Net in general and C# in particular a little tougher for VFP developers is different terminology for the same thing. For example, you'll often hear a C# developer refer to a "type" when what they really mean is a class. Fields and properties pose a similar issue.

In VFP, a property is a variable that belongs to a class. For example, the VFP MyClass class we saw in the "Classes vs. objects" section has two properties: tUpdated and tCreated. C# has something similar but they're called "fields," which is extra confusing because a "field" to a VFP developer is a column in a table. Here's an example of a C# class with six fields:

```
class Person
```

```
{
    public string FirstName;
    public string LastName;
    public string Phone;
    public string Address;
    public string City;
    public DateTime Created;
    public decimal Balance;
}
```

Like pretty much everything else in C#, fields must have a defined data type. In this example, all of the fields except Created and Balance are strings.

Setting the values of fields is just like setting the values of properties in VFP:

```
// Create a person and set its fields.
Person person = new Person();
person.FirstName = "Doug";
person.LastName = "Hennig";
person.Address = "123 Main Street";
person.City = "Anytown";
person.Phone = "999-999-9999";
person.Balance = 100;
person.Created = DateTime.Now;
```

Note how Created is set to the current date/time using the static Now property of the System.DateTime class.

It turns out that in addition to fields, C# also has properties. If a C# field is like a VFP property, what's a C# property? It's like a VFP property with access and/or assign methods. In C#, the access method uses the `get` keyword and the assign method uses `set`. Another difference is that a C# property doesn't have a storage location associated with it, so typically you'll use a private field to store the value for a public property (often referred to as the "backing" field).

Here's the Person class rewritten to use properties instead of fields:

```
class Person
{
    private string _firstName;
    public string FirstName
    {
        get { return _firstName; }
        set { _firstName = value; }
    }

    public string _lastName;
    public string LastName
    {
        get { return _lastName; }
        set { _lastName = value; }
    }
```

```
    private string _phone;
    public string Phone
    {
        get { return _phone; }
        set { _phone = value; }
    }

    private string _address;
    public string Address
    {
        get { return _address; }
        set { _address = value; }
    }

    private string _city;
    public string City
    {
        get { return _city; }
        set { _city = value; }
    }

    private decimal _balance;
    public decimal Balance
    {
        get { return _balance; }
        set { _balance = value; }
    }

    private DateTime _created;
    public DateTime Created
    {
        get { return _created; }
        set { _created = value; }
    }
}
```

Here are the things to note about this code:

- There's a lot more code! Every property takes four lines (not counting curly braces on their own lines) and two methods.
- For every property, there's an underlying private field with a similar name, starting with an underscore and lower-case letter. This is a common naming convention. Also note that the data type for the private field must match that of the property.
- The "getter" for a property is straightforward—it returns the value of the underlying private field—but the "setter" is interesting: it sets the private field to something called "value." That's a built-in keyword in set methods that refers to the value specified in the assignment statement for the property. The VFP equivalent for the setter for FirstName, for example, is:

  ```
  function FirstName_Assign
     lparameters tuValue
  ```

```
    FirstName = tuValue
endfunc
```

The difference is that VFP needs a specific LPARAMETERS statement to name the incoming value whereas C# automatically puts the incoming value into the `value` variable.

- The code setting the values for the properties doesn't change; it still sets FirstName, LastName, and other properties as before. There is a difference in internal execution, however; rather than writing to a field, the setter of a property fires, which stores the specified value to an underlying private field.

Since there's a lot more code involved, why bother using properties instead of fields? The common wisdom is that fields should only be used for private data and that properties should be used for public data (actually, to provide public access to private data). That's because properties provide control over how the data is accessed, how it's changed, or whether it even can be changed outside the class. Fields provide no control at all.

Read-only properties and fields

The value of the Created property probably should be set automatically and not rely on client code setting it. That also means that we want to prevent client code from changing it.

The first task can be done by assignment code added to the definition. Here's the code for the field-based class:

```
public DateTime Created = DateTime.Now;
```

For the property-based class, we'll also set the value of the field:

```
private DateTime _created = DateTime.Now;
```

How we do the second task, preventing client code from changing the value, depends on whether we're using fields or properties. For a field, we'll add the `readonly` keyword:

```
public readonly DateTime Created = DateTime.Now;
```

For a property, remove the setter code; this automatically makes the property read-only:

```
private DateTime _created = DateTime.Now;
public DateTime Created
{
    get { return _created; }
}
```

Note that a read-only field or property can't be written to by anything, not even the class itself. (The exception is that a read-only field can be written to in the constructor of a class; we'll talk about constructors later.) If you need to change the value of a property under some conditions, but only in the class, include the setter code but make it private:

```
private DateTime _created = DateTime.Now;
public DateTime Created
{
    get { return _created; }
    private set { _created = value; }
}
```

Automatic properties

To reduce the amount of code you need to type, C# allows you to define automatic properties: properties that automatically have a field which stores the actual value. Internally, the compiler generates a field and wires up the getter and setter for the property to it, but from our perspective, there's a lot less typing needed.

To define an automatic property, specify empty getter and setters. Here's the Person class, rewritten so most properties are automatic:

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Phone { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public decimal Balance { get; set; }

    private DateTime _created = DateTime.Now;
    public DateTime Created
    {
        get { return _created; }
    }
}
```

Note that we didn't make Created an automatic property. The reason is that we want to assign a default value to Created. There are two ways to do that: don't use an automatic property and set the default value for the backing field (as in this case) or set the value of the automatic property in the constructor of the class (constructors are discussed in the next section). Which you choose is up to you. I prefer to use a non-automatic property in this case. Another case where you can't use an automatic property is when you want behavior in the getter or setter, as we'll see in the "Branching code" section.

Setting properties at instantiation

C# has an interesting feature in that you can set the values of properties when a class is instantiated. To do this, use curly braces rather than parentheses in the `new` statement, and between the curly braces put comma-delimited "property = value" pairs. For example:

```
Person person = new Person {
    FirstName = "Doug",
    LastName = "Hennig",
    Address = "123 Main Street",
    City = "Anytown",
```

```
    Phone = "999-999-9999",
    Balance = 100 };
```

Constructor and destructor methods

VFP classes have an Init method that automatically fires when the class is instantiated into an object. C# has something similar, called the constructor method (sometimes abbreviated to "ctor"). Unlike VFP, the method doesn't have a constant name like "Init;" rather, it's named the same as the class.

Like VFP, ctors are used when you want to initialize the object when it's created. Ctors can accept parameters like other methods. For example, suppose you want client code using the Person class to specify the first and last name when the object is instantiated.

```
public Person(string firstName, string lastName)
{
    FirstName = firstName;
    LastName = lastName;
}
```

There are a few things to note about this code:

- Ctors don't return a value, so no data type, not even `void`, is specified. This also means that C# doesn't support the sometimes-used VFP feature of returning false from Init to prevent instantiation of the object.
- A ctor doesn't have to be public; it could be, for example, internal. That means that only classes in the same project can instantiate the class.
- A common naming convention is that parameters use Camel case with a starting lower-case letter.
- Because the ctor accepts two strings, you must specify values for those strings in the `new` statement instantiating the object or the code won't compile:

  ```
  Person person = new Person("Doug", "Hennig");
  ```

While ctors are very common, destructors are much less so, even less than the Destroy method in VFP. That's because .Net has a built-in garbage collector that takes care of releasing memory and resources that are no longer needed. Destructors are really only needed if your class deals with "unmanaged" resources such as network connections or Windows file handles.

A destructor is named the same as the class with a "~" prefix. For example:

```
~Person()
{
    // Code goes here
}
```

There are several limitations of destructors:

- A class can only have one destructor.
- Destructors cannot be inherited or overloaded.
- Destructors cannot be called; they are invoked automatically.
- A destructor does not take modifiers (such as visibility) or have parameters.

Overloaded methods

Suppose you want to allow, but not force, client code to specify the first and last name when the object is instantiated. VFP allows this because method parameters are optional. For example, it's common to write code like this:

```
function Init(tcFirstName, tcLastName)
   if vartype(tcFirstName) = 'C'
      This.FirstName = tcFirstName
   endif vartype(tcFirstName) = 'C'
   if vartype(tcLastName) = 'C'
      This.LastName = tcLastName
   endif vartype(tcLastName) = 'C'
endfunc
```

If client code specifies the first and/or last name, they're used. If not, no harm done.

In C#, that's not the case: if a method accepts parameters, you have to pass them or the code won't compile. However, C# has a way around this: overloaded methods. An overloaded method is one that has several instances, each with a different signature. Here's an example of an overloaded ctor:

```
public Person(string firstName, string lastName)
{
    FirstName = firstName;
    LastName = lastName;
}

public Person()
{
}
```

(Note that if a class has no ctor, the compiler automatically creates a default one that's essentially the same as the second one in this example. However, as soon as you create any ctor, the compiler no longer generates the default one. That's why we had to add a default ctor here that does nothing.)

With this code, either of these two instantiations work:

```
Person person = new Person("Doug", "Hennig");
Person person = new Person();
```

Because the first line matches the ctor accepting two parameters, the compiler automatically calls that ctor, and similarly the second line calls the ctor with no parameters. This statement, on the other hand, fails to compile because it doesn't match any ctor:

```
Person person = new Person("Doug");
```

If you want one ctor to call another, use ": this" in the constructor statement. For example, we could make the ctor that accepts no parameters pass nulls to the one that does, and put all other behavior in the latter. That way, you don't have to duplicate code.

```
public Person(string firstName, string lastName)
{
   FirstName = firstName;
   LastName = lastName;
   // Other behavior goes here
}

public Person()
   : this(null, null)
{
}
```

It isn't just ctors that can be overloaded; any method can be. In order to have overloaded methods, each one must have a different signature: different return data type, different number of parameters, different data types for the parameters, or whatever change is needed so the compiler can distinguish one method from another. Parameter names aren't enough to distinguish methods; for example, this code won't compile because both methods accept two strings, regardless of the names:

```
public Person(string firstName, string lastName)
{
    FirstName = firstName;
    LastName = lastName;
}

public Person(string lastName, string firstName)
{
    FirstName = firstName;
    LastName = lastName;
}
```

Inheritance

Like VFP, C# supports subclassing. To do that, specify the parent class in the class definition line. For example, here's the Employee class, which is a subclass of Person:

```
public class Employee : Person
{
   public DateTime HireDate { get; set; }
   public double Salary { get; set; }
}
```

As expected, Employee has all of the attributes of Person plus a couple of additional properties. One interesting thing about subclasses in C#: ctors aren't inherited so the following code fails to compile:

```
Employee emp = new Employee("Doug", "Hennig");
```

If you want Employee to have the same behavior as Person—being able to specify the first and last name to the ctor—you have to add a ctor to support that.

Another interesting thing is that you can't override a method of a parent class unless that method was designed to be overridden. For example, support Person has this simple method:

```
public int GetValue()
{
   return 5;
}
```

You would think that if you want different behavior in Employee, you'd simply add this code, which is how you'd do it in VFP:

```
public int GetValue()
{
   return 5;
}
```

However, this gives a compiler warning (not error): "'MyFirstApp.Employee.GetValue()' hides inherited member 'MyFirstApp.Person.GetValue()'. Use the new keyword if hiding was intended." We won't discuss what the new keyword would do for you, but here's the way you define a method that can be overridden:

```
public virtual int GetValue()
{
   return 5;
}
```

The virtual keyword means this method can be overridden. The overridden code looks like this:

```
public override int GetValue()
{
   return 7;
}
```

If you want to call the parent behavior in the subclass (like the VFP DODEFAULT() function), call it using the base keyword:

```
public override int GetValue()
{
   base.GetValue();
   return 7;
```

```
}
```

Polymorphism

You can upcast a class to a parent. (The reason it's called "upcast" is that parent classes are considered to be at the top of a class drawing and subclasses at the bottom). For example, in the following code, Employee is instantiated into a variable of type Person:

```
Person emp = new Employee();
```

Note that when you do this, you don't have access to the additional attributes of Employee. For example, using emp.HireDate in code results in a "'Person' does not contain a definition for 'HireDate'" compile error, which makes sense since emp is of type Person. However, this is still useful because it allows you to not worry about an exact type under some conditions.

Suppose you have several subclasses of Person: Employee, Customer, and Supplier. You want to create an array of instances of them, but you can't create an array of more than one data type. However, if you upcast them all to Person, you can create an array of various types:

```
Person[] people = new Person[5];
people[0] = new Person();
people[1] = new Employee();
people[2] = new Supplier();
people[3] = new Person();
people[4] = new Customer();
```

We can now set the values of common properties—those defined in Person—for these objects.

Upcasting is implicit (meaning you don't have to use a casting statement) and safe, because a subclass is always an instance of its parent. Downcasting an object back to its original class, on the other hand, is more complex: you have to use a casting statement and be careful that the object can actually be casted to the desired type.

There are two types of casting statements:

- Specifying the desired class in parentheses:

  ```
  Employee emp = (Employee)people[1];
  ```

- Using the `as` operator:

  ```
  Supplier supp = people[2] as Supplier;
  ```

The benefit of the second approach is that if the object being upcast isn't an instance of the specified type, null is returned, whereas the first approach throws an InvalidCastException runtime exception. For example, the first statement errors when run but the second simply sets supp to null:

```
Employee emp = (Employee)people[2];
Supplier supp = people[3] as Supplier;
```

If you're not sure whether an object is an instance of a particular class, you can use the `is` operator, which returns true if the specific object is an instance of the specified class. For example, suppose you want to set the class-specific properties of an object in the people array. This code allows you to handle each possibility:

```
if (people[1] is Employee)
{
    // handle Employee
}
else if (people[1] is Supplier)
{
    // handle Supplier
}
else if (people[1] is Customer)
{
    // handle Customer
}
```

For more information about upcasting and downcasting, see http://tinyurl.com/7w2vjyu.

Interfaces

One of the guidelines in object-oriented programming is that to reduce dependencies, you should "program to interface, not implementation." This means that you rely on only the public properties and methods of a class and not its internal details. Another way of stating this is that you depend on what a class does and not how it does it. In fact, you should be able to replace one class with another and not have to change the behavior of the client. In the words of OOP guru Erich Gamma:

"Once you depend on interfaces only, you're decoupled from the implementation. That means the implementation can vary, and that's a healthy dependency relationship. For example, for testing purposes you can replace a heavy database implementation with a lighter-weight mock implementation."—http://tinyurl.com/684sj7

VFP doesn't have the concept of interfaces, but .Net does. An interface defines the signatures of methods, properties, and events. It has no executable code at all. Instead, the code, or implementation of the interface, is done in classes. Interfaces are often used when developing an API so that you don't have to change the method signatures that others will be programming against.

By convention, interface names start with a capital "I." Here's our Person class, modified to implement the IPerson interface:

```
public class Person : IPerson
{
   // rest of the code stays the same
}
```

Here's the IPerson interface:

```
interface IPerson
{
    string Address { get; set; }
    string City { get; set; }
    DateTime Created { get; }
    string FirstName { get; set; }
    string LastName { get; set; }
    string Phone { get; set; }
}
```

(Note that I didn't create this interface manually. Instead, I right-clicked the Person class definition and chose Refactor, Extract Interface from the shortcut menu.)

IPerson defines what classes that implement it look like: what properties and methods they have, the data types, whether properties are read-only, and so on.

To tell the truth, although I understood the concept of interfaces, I couldn't think of a practical reason to create them until I started creating unit tests for my classes. When you have a class with dependencies on other classes, testing becomes more complicated: to test the class, you have to create instances of the other classes it depends upon. If some of those other classes are complex, such as opening databases or accessing resources that may not be available in a test environment, it's a lot more work to set up the test properly.

For example, suppose Person holds a reference to another class (let's call it DataAccessClass) that loads the properties of the person from a database record. Here's what the LoadPerson method of Person might look like:

```
public DataAccessClass DataAccess;
public void LoadPerson(string ID)
{
    DataAccess.FindRecord(ID);
    FirstName = DataAccess.GetField("FirstName");
    LastName = DataAccess.GetField("LastName");
    // code that gets the rest of the properties
}
```

(Ok, this is a bad design. I just wanted to demonstrate something simple, so let's just go with the flow <g>.)

Obviously, we'll need to have tests for DataAccessClass that test the FindRecord and GetField methods to ensure they work properly. However, tests for LoadPerson don't really have anything to do with what FindRecord and GetField do; what they should test is that the properties of Person are set properly. The problem is that DataAccess is hard-coded to be of type DataAccessClass, which carries all kinds of baggage with it.

Instead, create an interface for DataAccessClass and use that in Person instead:

```
public interface IDataAccess
```

```
{
    void FindRecord(string ID);
    string GetField(string fieldName);
}

public class DataAccessClass : IDataAccess
{
    public void FindRecord(string ID)
    {
        // code goes here
    }
    public string GetField(string fieldName)
    {
        // code goes here
    }
}

public class Person
{
    public IDataAccess DataAccess;
    // rest of code goes here
}
```

For testing purposes, we don't need to use the heavyweight DataAccessClass: we can use a lightweight "mock" class that implements IDataAccess and set Person.DataAccess to an instance of it:

```
public class MockDataAccessClass : IDataAccess
{
    public void FindRecord(string ID)
    {
        // This code does nothing
    }
    public string GetField(string fieldName)
    {
        // Returns a dummy value
        return "XXX";
    }
}
```

Now when we test LoadPerson, we're using a simple mock class that does nothing but is very lightweight and doesn't require a complex setup to work.

Enumerations

An enumeration (often shorted to "enum") is a mechanism that allows you to provide names for related constants. VFP sort of supports enumerations. For example, in the Properties window, you can set the BorderStyle property of a form to 0 – No Border, 1 – Fixed Single, 2 – Fixed Dialog, or 3 – Sizable. The names associated with those numeric values are easier to understand than the values themselves. FoxPro.H also defines some constants that can be thought of as enumerations, such as the values representing MESSAGEBOX() parameter values: MB_OK, MB_OKCANCEL, MB_YESNO, and so on.

However, VFP doesn't allow us to define our own, real, enums. C# does, and they're used extensively, both in the .Net framework and in user code.

An enum definition is similar to a class, but uses the keyword `enum` instead. Between the curly braces is a comma-delimited list of the possible values of the enum. Here's an example of an enum that defines employee types:

```
public enum EmployeeType
{
    Executive,
    Management,
    FrontLine,
    BackOffice
}
```

By default, the underlying data type for each item in the enum is int, although other types such as byte, short, and long are also supported. The values of the items start at zero and increment by one for each item in the order listed. So, in the EmployeeType enum, Executive is 0, Management is 1, and so on. You can specify a different starting value using syntax like this:

```
public enum EmployeeType
{
    Executive = 1,
    Management,
    FrontLine,
    BackOffice
}
```

Let's add a property to the Employee class that uses this enum:

```
public EmployeeType Type { get; get; }
```

Here's an example that sets Type to a specific value:

```
Employee emp = new Employee();
emp.Type = EmployeeType.Executive;
```

From this code, you can see the advantage of enums: rather than setting emp.Type to 0, which is meaningless, you set it to "Executive," which is obvious. Also, IntelliSense makes it easy to see the range of possible values, as you can see in **Figure 4**.

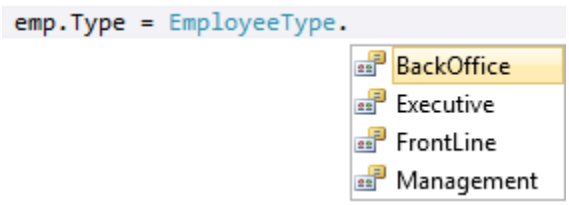

Figure 4. Visual Studio IntelliSense shows the range of possible values for enums.

Branching code

Like VFP, C# has an `if` statement that allows conditional branches. Here's the syntax:

```
if (some condition)
   statement(s)
```

`statement(s)` can be a single statement or a code block; the latter has to be surrounded by curly braces (some coding conventions state you should always use curly braces even if there's only a single statement):

```
if (some condition)
{
   statement
   statement
}
```

The condition must be in parentheses and be an expression that returns a Boolean value. You can combine expressions using `&&` as the "and" operator, `||` as the "or" operator, and `==` as the comparison operator. For example, suppose the Employee class has a new property, Title, and if Title is set to "President" or "Vice President," you want Type automatically set to the enum value Executive. The setter of Title takes care of that:

```
private string _title;
public string Title
{
    get { return _title; }
    set
    {
        _title = value;
        if (value == "President" || value == "Vice President")
            Type = EmployeeType.Executive;
    }
}
```

`else` is an optional part of `if`:

```
if (some condition)
   statement(s)
else
   statement(s)
```

You can add an `if` statement to `else`:

```
if (some condition)
   statement(s)
else if (some other condition)
   statement(s)
else
   statement(s)
```

You can, of course, have as many `else if` statements as desired.

The C# `switch` statement is similar to the VFP DO CASE structure, but not as flexible: rather than each CASE statement being a complete Boolean expression, including function and method calls, `switch` only allows you to test the various possible values of a single expression (although it doesn't have to be and is rarely Boolean). Here's the syntax:

```
switch (expression)
{
   case value1:
      statement(s)
      break;
   case value2:
      statement(s)
      break;
   default:
      statement(s)
      break;
}
```

Here are some comments about this:

- Like `if`, the expression must be in parentheses and `statement(s)` can be a single statement or a code block.
- If you don't add `break` after the code block for a case, execution will fall into the next case, which would be a problem. In fact, this causes a compiler error.
- `default` is the C# equivalent of VFP's `otherwise`. It's not required but is a good idea.

Here's an example: this code replaces the `if` structure in the setter for Title to handle additional cases:

```
switch (value)
{
    case "President":
        Type = EmployeeType.Executive;
        break;
    case "Vice President":
        Type = EmployeeType.Executive;
        break;
    case "Manager":
        Type = EmployeeType.Management;
        break;
    case "Clerk":
        Type = EmployeeType.FrontLine;
        break;
}
```

Notice that the code for "President" and "Vice President" is the same, so it can be simplified to:

```
switch (value)
{
    case "President":
```

```
    case "Vice President":
        Type = EmployeeType.Executive;
        break;
   // rest of code
}
```

Loops

Like VFP (and most languages), C# has keywords for looping constructs.

In VFP, if you need to break out of a loop early, you use the `exit` command. In C#, use `break` instead. The C# equivalent of VFP's `loop` command, which skips to the next iteration, is `continue`.

for

Although the syntax is different, C#'s `for` loop works very similar to VFP's. The syntax is:

```
for (variable initial value; end condition; how to change the variable)
   statement(s)
```

As with `if`, `statement(s)` can be a single statement or a code block surrounded by curly braces. We saw a typical example in the "string" section earlier in this document:

```
string message = "";
for (int i = 0; i < 255; i++)
{
    message += (char)i;
}
```

In this code, the variable being used for the loop is i and it's initialized to 0. The loop runs as long as i is less than 255. On each iteration, i is incremented by 1 (the ++ operator is a shortcut for `variable = variable + 1`). For a loop that runs backwards, use i--. You can, of course, step by values other than one by specifying an expression that increments or decrements the variable by the necessary value.

VS has a shortcut that generates a `for` loop for you: type "for" and press Tab to generate the following code:

```
for (int i = 0; i < length; i++)
{

}
```

"i" is automatically highlighted so you can type the variable name (which is automatically substituted into the other two instances of "i"), then hit Tab to highlight "length" and type the ending value for the loop.

foreach

C#'s `foreach` loop is very similar to VFP's `FOR EACH`: it processes each item in a collection of some type, such as an array. `foreach` is much more convenient to use with collections than

the equivalent `for` loop because you don't have to specify the length of the collection nor retrieve the current item from the collection on each iteration; the loop does both of those for you automatically. Here's the syntax:

```
foreach (variable in collection)
   statement(s)
```

Here's an example that outputs each character in a string to the console:

```
string message = "This is a message";
foreach (char letter in message)
{
    Console.WriteLine(letter);
}
```

Here's one that sums up the values in an array of the first eight numbers in the Fibonacci series:

```
int[] fibarray = new int[] { 0, 1, 1, 2, 3, 5, 8, 13 };
int fibsum = 0;
foreach (int value in fibarray)
{
    fibsum += value;
}
```

(Of course, we could also use the Sum() method of the array.)

VS has a shortcut that generates a `foreach` loop for you: type "foreach" and press Tab to generate the following code:

```
foreach (var item in collection)
{

}
```

"var" is automatically highlighted so you can type the data type, then hit Tab to highlight "item" and type the variable name, and then hit Tab to highlight "collection" and type the name of the collection object.

while

C#'s `while` loop is similar to VFP's DO … WHILE loop. Here's the syntax:

```
while (condition)
   statement(s)
```

Like VFP, if the condition evaluates to false on the first iteration, the loop may not execute at all.

VS has a shortcut that generates a `while` loop for you: type "while" and press Tab to generate the following code:

```
while (true)
{

}
```

"true" is automatically highlighted so you can type the condition.

do

A `do` loop is similar to a while loop, but since the condition comes at the end of the loop rather than the start, the loop executes at least once. The syntax is:

```
do
   statement(s)
while (condition)
```

VS has a shortcut that generates a `do` loop for you: type "do" and press Tab to generate the following code:

```
do
{

} while (true);
```

"true" is automatically highlighted so you can type the condition.

Lists and collections

We saw earlier how to create an array in C#. While arrays are useful, they suffer the same limitations as they do in VFP, such as being slow to search. For that reason, lists and collections are more popular. Also, lists and collection dynamically resize, so they're much easier to work with when adding elements.

A very popular class for maintaining a list is the generic List<T> class. The "<T>" notation means that the type is specified when the class is instantiated, and the list can only contain objects of that type. Here's an example that uses List<T> instead of an array to hold the Fibonacci series we saw earlier:

```
List<int> fibList = new List<int>();
fibList.Add(0);
fibList.Add(1);
fibList.Add(1);
fibList.Add(2);
fibList.Add(3);
fibList.Add(5);
fibList.Add(8);
fibList.Add(13);
```

List<T> is in the System.Collections.Generic namespace, so be sure there's a `using` statement for that namespace in the code.

In addition to Add(), List also has an AddRange() method that adds the elements of an array or another list to the list:

```
fibList.AddRange(new int[]{0, 1, 1, 2, 3, 5, 8, 13});
```

List are often processed in loops, so `foreach` is perfect for this use.

Some other useful properties and methods of List<T> are:

- Count returns the number of elements in the list.
- Average(), Max(), Min(), and Sum() return the values you'd expect from the method names.
- Reverse() reverses the order of the elements in the list.
- Sort() sorts the list.
- Remove() and RemoveAt() remove certain items from the list while Clear() removes all items.
- BinarySearch() searches the list for the specified item.
- Contains() returns true if the list contains the specified item.
- ToArray() copies the elements to a new array.

While List<T> is great for lots of things, it isn't very fast when it comes to searching. If you need to search a lot, Dictionary<TKey, TValue> is a better choice.

Dictionary<TKey, TValue> provides a collection of items indexed by key. You specify the type of both the key and the value when you instantiate the class. For example, suppose you want to maintain a list of Canadian province names and their abbreviations. You could use code like this:

```
Dictionary<string, string> Provinces = new Dictionary<string, string>();
Provinces.Add("BC", "British Columbia");
Provinces.Add("AB", "Alberta");
Provinces.Add("SK", "Saskatchewan");
Provinces.Add("MB", "Manitoba");
Provinces.Add("ON", "Ontario");
Provinces.Add("PQ", "Quebec");
Provinces.Add("NB", "New Brunswick");
Provinces.Add("NS", "Nova Scotia");
Provinces.Add("PE", "Prince Edward Island");
Provinces.Add("NF", "Newfoundland");
```

Once you have the collection loaded, you can treat it like an array, specifying a key to obtain the value associated with that key:

```
Console.WriteLine(Provinces["MB"]); // Displays Manitoba
```

Note that doing this throws an exception if the key doesn't exist, so you should either wrap the code in a `try` structure (see the "Exception handling" section later in this document) or use the TryGetValue() method. This method has unusual syntax: rather than returning the value associated with the specified key, it returns a Boolean value indicating whether it succeeded or not and if so, puts the value into a variable passed by reference using the `out` keyword:

```
string provName;
if (Provinces.TryGetValue("MB", out provName))
    Console.WriteLine(provName);
```

As with List<T>, it's common to use `foreach` to process the entire collection. In that case, the variable in the loop is a KeyValuePair<TKey, TValue> type, which has Key and Value properties. It's obviously more convenient to use the `var` keyword rather than KeyValuePair<TKey, TValue> in this case:

```
foreach (var province in Provinces)
{
    Console.WriteLine(province.Value);
}
```

You could also iterate through the Keys or Values members of the dictionary:

```
foreach (var provinceName in Provinces.Values)
{
    Console.WriteLine(provinceName);
}
```

Like List<T>, you can use Count to determine the number of items in the collection, Remove() to remove an item from the collection, and Clear() to remove all items.

.Net has numerous other collection classes available, including:

- SortedList<T>: like List<T> but values are stored in a sorted order. SortedList<T> is slower than List<T>, so only use it when you need the list sorted for display purposes.
- SortedDictionary<TKey, TValue>: like Dictionary<TKey, TValue> but values are stored in a sorted order. Like SortedList<T>, SortedDictionary<TKey, TValue> is much slower than Dictionary<TKey, TValue>.
- Collection<T>: this class isn't really meant to be used by itself but as a base class for custom collections you build. This is in the System.Collection.ObjectModel namespace.
- KeyedCollection<T>: this is typically used when T is a class that has a key associated with it, usually as a member of the class. For example, this could be used for a collection of our Person objects, where the key is the LastName property (or more likely, a FullName property). This is also in the System.Collection.ObjectModel namespace.

- ObservableCollection<T>: this class provides notifications when items are added, removed, or when the whole collection is refreshed. This is useful when you want other objects automatically notified via events when the collection changes. This is also in the System.Collection.ObjectModel namespace.

All of these classes implement the .Net IEnumerable interface (among several others), which means they can be used in `foreach` loops and other places IEnumerable is expected.

LINQ

LINQ, which stands for Language Integrated Query, was originally designed to add querying to .Net languages, similar to the native database access commands, including SQL, built into VFP. (In fact, some members of the VFP team worked on LINQ.) However, it evolved into the ability to query just about anything, not just databases.

LINQ is a complex subject; entire books have been written on it. This is merely an introduction into this topic.

Here are some things to know about LINQ:

- Out of the box, LINQ works on objects that implement IEnumerable, which means we can query arrays, lists, collections, and other things.
- LINQ has operators similar to SQL: Select to do a projection, Where to filter, OrderBy and OrderByDescending to sort, GroupBy to group, and so on.
- LINQ has two different syntaxes: fluent and query. Fluent syntax looks like normal methods calls, although they can be chained, such as SomeObject.Where(Some Expression).Select(SomeProperty). Query syntax looks like a SQL statement with the syntax rearranged, such as:

  ```
  from n in names where n.Contains("Doug") select n;
  ```

  Some people prefer to use fluent, others prefer to use query, and others use both, depending on the circumstances. I prefer to use fluent, so the examples shown in this document use that syntax.

Let's start by creating a list of people. Note the compact syntax here, initializing both the collection and each person in the ctor:

```
List<Person> people = new List<Person>{
    new Person {
        FirstName = "Doug",
        LastName = "Hennig",
        Address = "123 Main Street",
        City = "Anytown",
        Phone = "999-999-9999",
        Balance = 100 },
    new Person {
        FirstName = "Rick",
        LastName = "Schummer",
```

```
        Address = "456 North Avenue",
        City = "Sterling Heights" },
    new Person {
        FirstName = "Tamar",
        LastName = "Granor",
        Address = "789 Smith Road",
        City = "Philadelphia",
        Balance = 50 }
};
```

Let's do a simple query retrieving those who have a balance of greater than zero:

```
var customers = people.Where(p => p.Balance > 0);
foreach (var person in customers)
{
    Console.WriteLine("{0} {1}", person.FirstName, person.LastName);
}
```

The LINQ statement looks complicated, so let's break it down:

- `var` is frequently used because sometimes the exact type returned is a little complex, so it's easier to let the compiler deal with it. In this case, the statement returns IEnumerable<Person>.
- Where isn't actually a method of List<T>. Instead, it's what's known as an extension method, a method that appears to be dynamically added to the class at compile time. Although I won't discuss extension methods in this document, it's a very cool subject because it allows you to add methods to any class without modifying the class. The extension methods that comprise LINQ are found in the System.Linq namespace, so be sure to have a `using` statement for it.
- The odd syntax passed to Where is known as a lambda expression. Lambda expressions are discussed in more detail below. Essentially, the parameter passed to Where is a function that's executed for each item in the collection. Those for which the function returns true are included in the final result, and the rest are rejected.
- The parameters passed to Console.WriteLine specify a formatted string. The first parameter is the string to display, with digits between curly braces being placeholders replaced from other parameters ("{0}" is replaced with the first parameter after the format string, "{1}" with the second, and so on). In this case, we're simply outputting two strings separated by a space, so we could have used this instead:

  ```
  Console.WriteLine(person.FirstName + " " + person.LastName);
  ```

A lambda expression is essentially an anonymous function; that is, a nameless function declared inline rather than in its own code block. This lambda expression:

```
p => p.Balance > 0
```

is similar to this pseudo-code:

```
private bool NoName(Person p)
return p.Balance > 0;
```

The variable name preceding the "=>" symbol is passed to the function as a parameter. The value of the variable comes from the item in the collection currently being processed.

So, because Where acts on every item in the collection, similar to `foreach`, the following pseudo-code is similar to the LINQ statement:

```
IEnumerable<Person> customers;
foreach(Person p in people)
{
   if (NoName(p))
      customers.Add(p);
}
```

In fact, before LINQ, this was very much how code was written to conditionally select a subset of a collection (and is still used by developers who haven't taken the time to learn LINQ). Obviously, LINQ is much more compact and, once you get past the lambda expression syntax, it's easier to understand the intent of the code.

Because many LINQ methods both operate on and return IEnumerable, you can chain some of them together, with the result of one acting as the input for the next one. For example, let's sort the list of people with a balance of more than 0 by last name:

```
var customers = people.Where(p => p.Balance > 0).OrderBy(p => p.LastName);
```

In this statement, the IEnumerable returned by Where is operated on by OrderBy, the lambda expression of which specifies the thing to sort by (in this case, the LastName property). OrderBy returns an IEnumerable, which is placed into the customers variable.

We don't have to return Person objects; like the Select clause of a SQL statement, the Select method allows you to determine what the resulting collection consists of. For example, since we just need the first and last name of the people who match our criteria, we could use:

```
var customers = people.Where(p => p.Balance > 0)
    .OrderBy(p => p.LastName)
    .Select(p => p.FirstName + " " + p.LastName);
foreach (var person in customers)
{
    Console.WriteLine(person);
}
```

In this case, customer is IEnumerable<string> because that's what the lambda expression in Select returns.

As I mentioned, LINQ is a big topic, but it's well worth the time exploring it in detail, as it can significantly reduce the amount of code you write to process collections. A tool you

might find useful while learning LINQ, and even after you've become familiar with it, is LINQPad, available free from http://www.linqpad.net.

Exception handling

VFP has three mechanisms for catching errors:

- The oldest is ON ERROR: when an error occurs, the specified command executes.
- The next oldest is the Error method of objects, which is automatically called if an error occurs in one of the object's methods.
- The newest is the TRY structure, which provides local error handling.

C# has only one mechanism, the try structure (in fact, VFP's TRY came from .Net thanks to cross-pollination of the VFP and .Net teams). The syntax is very similar to VFP's:

```
try
{
   // Execute some code
}
catch
{
   // Deal with the error
}
```

Similar to VFP's CATCH WHEN, C#'s catch statement can include the type of exception expected, and the try structure can include multiple catch statements:

```
try
{
   string x = "abc";
   int i = int.Parse(x);
}
catch (FormatException)
{
   MessageBox.Show("x does not contain an integer value");
}
catch (IndexOutOfRangeException)
{
   // Deal with index out of range exception
}
catch
{
   // Deal with any other type of error
}
```

Like a VFP application, it's very important to trap errors in a C# application to avoid an unhandled exception error like the one shown in **Figure 5**, generated by moving the two statements inside the try block outside the structure.

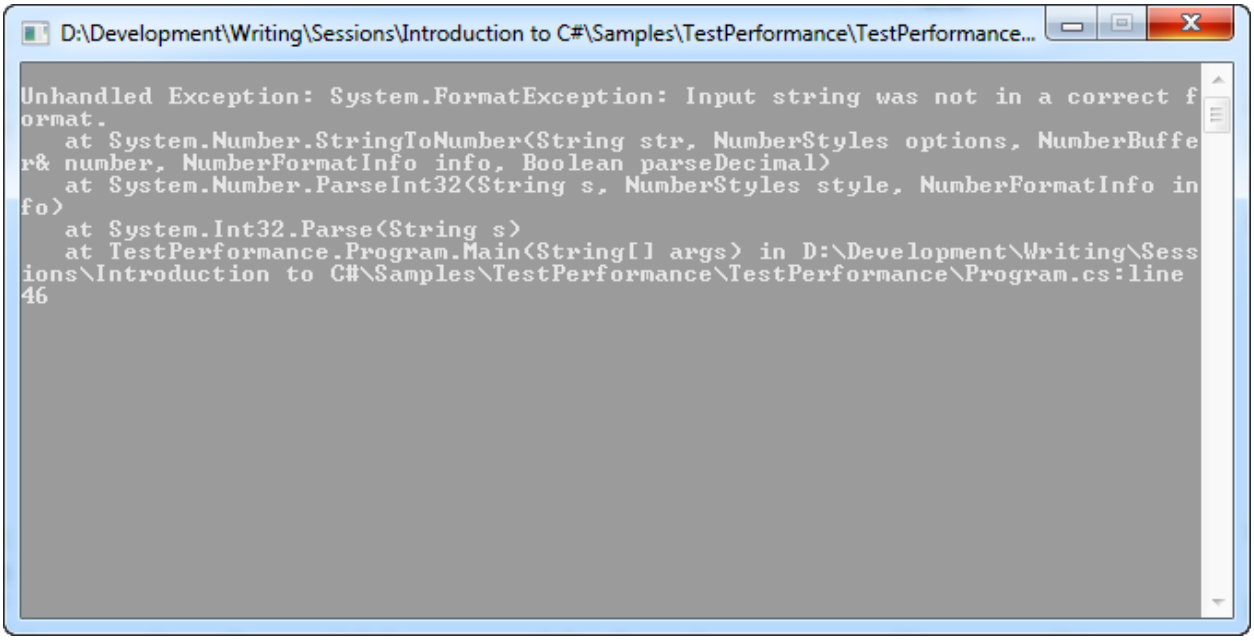

Figure 5. You want to avoid unhandled exceptions at all costs!

Event handling

There are two types of events in VFP: built-in and developer-created. Built-in events like Click and InteractiveChange fire when the user does something in the UI of a form. Developer-created events use the RAISEEVENT function in conjunction with BINDEVENT to notify one object when something happens in another object. Here's an example of the latter:

```
loRaiser = createobject('EventRaiser')
loHandler = createobject('EventHandler')
bindevent(loRaiser, 'SomethingHappened', loHandler, 'OnSomethingHappened')
loRaiser.DoSomething()


define class EventRaiser as Custom
   function DoSomething
      raiseevent(This, 'SomethingHappened')
   endfunc

   function SomethingHappened
      * We don't need code here; it just needs to exist.
   endfunc
enddefine


define class EventHandler as Custom
   function OnSomethingHappened
      messagebox('Something happened')
   endfunc
enddefine
```

While this appears to be entirely different than the built-in events, in fact they're similar; it's just that with developer-created events, you have to wire up the event handling yourself while in built-in events, the wiring is done for you behind the scenes. When you open a code window for the Click event of a button, you think you're putting code into the Click event. However, that's not really the case. Windows raises an event when you click the button, and VFP has an event handler called Click that's called when the event is raised. VFP hides the internals from us so we only see the event handler, not the wiring between the event and its handler. It's almost as if a CommandButton were defined like this and Windows called ClickEvent when you click the button:

```
define class CommandButton as Custom
   hidden function ClickEvent
      raiseevent(This, 'Click')
   endfunc

   function Click
      messagebox('You clicked the button')
   endfunc
enddefine
```

.Net events are like the developer-created ones in VFP. Even UI events, which sort of look like they're handled like VFP's events, are wired up with generated code you can see. By default, VS names a UI event handler *ObjectName_EventName* but you can specify a different name if you wish. The Click event is specifically wired to its handler through generated code like this in the "Component Designer generated code" section of a form's code file, which is executed when the form instantiates:

```
this.myButton.Click += new System.EventHandler(this.myButton_Click);
```

This code adds the myButton_Click method to the list of handlers for the Click event for the myButton object. (In case you're wondering, yes, an event can have more than one handler.) VS also creates the myButton_Click method, ready for you to put in the code.

Raising events

There are three components to raising an event in .Net. First, you have to define a delegate. A delegate is a pointer to a function. It specifies what the signature of the function is: what parameters it accepts and what data type it returns. Here's an example of a delegate definition:

```
public delegate void MyEventHandler(object sender, EventArgs e);
```

There are three rules for the delegate:

- It must have a void return type.
- It must accept two arguments. The first contains a reference to the object that raised the event, and must be of type object. The second contains an object subclassed from EventArgs, which holds extra information about the event.

- Its name must end with “EventHandler.”

If you don’t feel like creating your own delegate, .Net has a built-in one called, conveniently enough, System.EventHandler. It’s a generic class, so it must be specified as EventHandler <TEventArgs>.

The second thing you need is an event. An event is a special type of delegate, so you specify what delegate it’s based on. The following creates an event that uses the delegate we just defined:

```
public event MyEventHandler MyEvent;
```

This one uses the built-in handler:

```
public event EventHandler<EventArgs> MyEvent;
```

You can pass more information about the event to the event handler using a subclass of EventArgs, such as:

```
public class MyEventArgs : EventArgs
{
   public readonly string EventDetails;

   public MyEventArgs(string details)
   {
      EventDetails = details;
   }
}
```

Finally, to raise the event, you call it like you would any method, passing a reference to the object and the event arguments:

```
public void SomethingHappened()
{
   MyEventArgs e = new MyEventArgs("this is the detail");
   MyEvent(this, e);
}
```

However, the usual design pattern for raising events is to create a non-public “On” method that actually raises the event, and call that method from the method where something happened:

```
public void SomethingHappened()
{
   MyEventArgs e = new MyEventArgs("this is the detail");
   OnSomethingHappened(e);
}

protected virtual void OnSomethingHappened(MyEventArgs e)
{
   if (MyEvent != null)
```

```
    {
        MyEvent(this, e);
    }
}
```

Note the check for the presence of an event handler (MyEvent != null).

Here's a more complete example. The Person class raises an event when the value of a property changes, similar to the ProgrammaticChange event in VFP. Note that this code doesn't define a delegate since the built-in EventHandler is used.

```
// A class that raises events when properties change.
public class Person
{
    private string _firstName;
    public string FirstName
    {
        get { return _firstName; }
        set
        {
            // If the value is different, store it and call OnChanged to raise an
            // event.
            if (_firstName != value)
            {
                _firstName = value;
                OnChanged(new PropertyChangedEventArgs("FirstName"));
            }
        }
    }

    private string _lastName;
    public string LastName
    {
        get { return _lastName; }
        set
        {
            // If the value is different, store it and call OnChanged to raise an
            // event.
            if (_lastName != value)
            {
                _lastName = value;
                OnChanged(new PropertyChangedEventArgs("LastName"));
            }
        }
    }

    // Define the event.
    public event EventHandler<PropertyChangedEventArgs> PropertyChanged;

    // The actual method that raises the event.
    protected virtual void OnChanged(PropertyChangedEventArgs e)
    {
        if (PropertyChanged != null)
        {
```

```
            PropertyChanged(this, e);
        }
    }
}

// The custom EventArgs class for property changes.
public class PropertyChangedEventArgs : EventArgs
{
    public readonly string PropertyName;

    public PropertyChangedEventArgs(string name)
    {
        PropertyName = name;
    }
}
```

Handling events

To handle an event (whether one raised in code or raised by a Windows event), create a method that should be called when the event is raised and wire up the method to the event. Because the method is called from the delegate, it must have the same signature as the delegate: return void and accept the same two parameters.

Here's an example that uses the Person class above and handles its PropertyChanged event:

```
class Program
{
    static void Main(string[] args)
    {
        TestEvents test = new TestEvents();
        test.CreatePerson();
        Console.ReadLine();
    }
}

class TestEvents
{
    public void CreatePerson()
    {
        Person person = new Person();
        person.PropertyChanged += HandlePropertyChanged;
        person.FirstName = "Doug";
        person.LastName = "Hennig";
    }

    void HandlePropertyChanged(object sender, PropertyChangedEventArgs e)
    {
        Console.WriteLine(e.PropertyName + " fired");
    }
}
```

Reading from and writing to files

VFP has several ways to read from and write to text files, the easiest being the single function FILETOSTR and STRTOFILE functions. .Net also has numerous ways, including some more complicated approaches using streams. We'll just look at the simplest mechanism.

To write to a file in one operation, you can use one of the methods of System.IO.File, such as WriteAllText, which writes a string to a file, WriteAllLines, which writes a string array, and WriteAllBytes, which writes from an array of byte. To append to a file (or create it if it doesn't exist), use AppendAllText or AppendAllLines (there is no AppendAllBytes). You can also use AppendText, which appends UTF-8 encoded text to a file. You can use ReadAllText, ReadAllLines, and ReadAllBytes to read an entire file. ReadLines is like ReadAllLines except it returns a collection of strings (actually, IEnumerable<string>) and you can start working with the collection before it's full populated, which can be more efficient with very large files. Here's an example of some of these methods:

```
using System.IO;

// WriteAllText and ReadAllText
string s = "This is a string to write to a file";
File.WriteAllText("SomeText.txt", s);
string result = File.ReadAllText("SomeText.txt");
MessageBox.Show(result);

// WriteAllLines and ReadAllLines
string[] a = { "One", "Two", "Three" };
File.WriteAllLines("Lines.txt", a);
string[] aresult = File.ReadAllLines("Lines.txt");
MessageBox.Show(aresult.Length.ToString());
```

Data Access

We're spoiled as VFP developers: not only does VFP have a database engine based on DBF files, it includes built-in data-related commands and functions that make accessing data easy and fast. .Net developers aren't so lucky: C# doesn't know anything about data. Instead, you have to use one of the .Net mechanisms (there are several) to access data. We'll briefly look at the simplest one, ADO.NET.

The first concept to understand is a data provider. Like an ODBC driver, a data provider knows how to talk to a specific database engine. There are data providers for Microsoft SQL Server, Oracle, MySql, SQLite, and many other database engines, although not all of them come in the box. You can also use the ODBC and OLE DB providers to talk to any ODBC driver or OLE DB provider, but those have extra overhead.

There are two ways to retrieve data: using a DataReader and using a DataSet. A DataReader provides fast, forward-only, read-only access to data, so it's suitable for things like data processing or reports. A DataSet is essentially an in-memory database.

Using a DataReader

To retrieve data using a DataReader, start by connecting to a database using a connection object. To complicate matters, you have to use the connection class that's specific for your database engine. This makes writing generic code that can switch between different database engines a little tricky. Fortunately, all connection types implement IDbConnection, so this is one place you'll definitely want to use interfaces.

Here's an example of connecting to a SQL Server database (the System.Data.SqlClient namespace has classes specific for SQL Server):

```
using System.Data.SqlClient;
string connString = "server=(local);database=Northwind;trusted_connection=Yes";
// can also specify uid=UserName and pwd=Password
SqlConnection conn = new SqlConnection(connString);
conn.Open();
MessageBox.Show(conn.State.ToString());
conn.Close();
```

Next, you'll create a command object to execute a command. Like connection objects, you have to use the correct class for your database engine; SqlCommand is the class for SQL Server. You can either instantiate the command object or you can call the connection object's CreateCommand method. Command objects have several methods for retrieving data:

- ExecuteReader: executes the command and returns a DataReader object, of the type specific for the database engine; SqlDataReader in the case of SQL Server.
- ExecuteNonQuery: used when no result set is retrieved, such as an INSERT, UPDATE, or DELETE statement.
- ExecuteScalar: used when the result is a single value, such as a count or sum.

Here's an example that uses the conn object from the previous example:

```
string selectCmd = "select CustomerID, CompanyName from Customers";
SqlCommand cmd = conn.CreateCommand();
cmd.CommandText = selectCmd;
// Can also use SqlCommand cmd = new SqlCommand(selectCmd, conn);
```

Use ExecuteReader to create a DataReader, and then process the records in the result set in some way. Here's an example that displays the company names from the Northwind Customer table:

```
SqlDataReader reader = cmd.ExecuteReader();
while (reader.Read())
{
   Console.WriteLine(reader.GetString(1));
}
reader.Close();
conn.Close();
Console.ReadLine();
```

Note a few things about this code:

- Use the DataReader's Read method to read the next record into the DataReader. Read returns false when there are no more records, so it's used in a while loop to process all records.
- To get the value of a field in the current record, use one of the Get methods, such as GetString, GetDouble, or GetInt32. Note that you have to specify the column by its column number in the result set, remembering that C# is zero-based.
- Because the DataReader is connected to the database, you have to close it and the connection object once you're finished.

Using a DataSet

Unlike a DataReader, a DataSet is a connectionless object: a connection is opened, data is retrieved, and the connection is closed. To update the database with changes in the DataSet, the connection is reopened and the updates sent. You might think that constantly opening and closing connections causes a performance hit, but .Net internally manages a pool of connections that new connection requests are handled from.

A DataSet consists of one or more DataTable objects, contained in the DataSet's Table collection. A DataTable has a collection of DataRow objects named Rows, which contain the actual data, and a collection of DataColumn objects named Columns, which describe the columns in the table. See **Figure 6** for an illustration. These classes are in the System.Data namespace and are not database-specific.

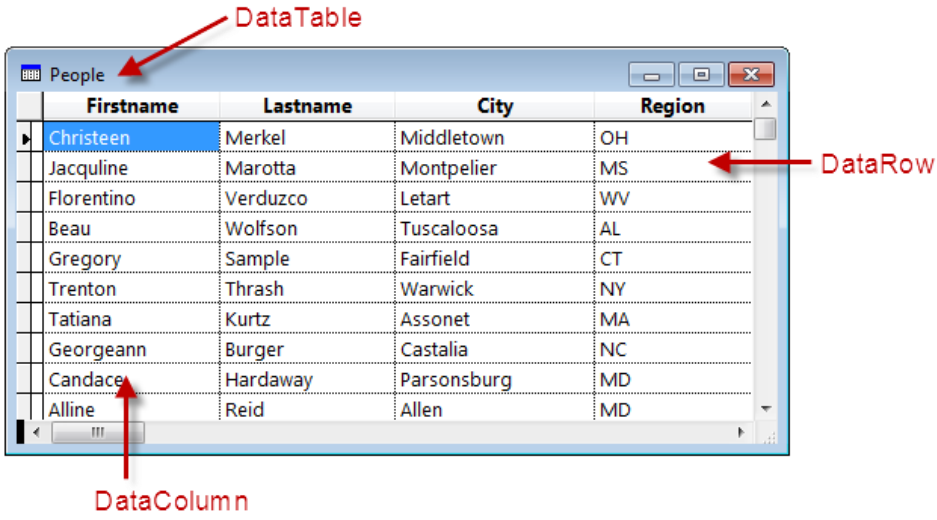

Figure 6. A DataTable consists of DataRows and DataColumns.

A DataAdapter, which is database-specific, is the conduit between the DataSet and the database. Its Fill method connects to the database, fills a DataSet, and closes the connection. Here's an example:

```
string connString = "server=(local);database=Northwind;trusted_connection=Yes";
```

```
string selectCmd = "select CustomerID, CompanyName from Customers";
SqlDataAdapter adapter = new SqlDataAdapter(selectCmd, connString);
DataSet ds = new DataSet();
adapter.Fill(ds);
```

You can then reference the tables, rows, and if necessary, columns in the DataSet. Here's the DataSet equivalent of the DataReader code that outputs company names:

```
DataTable table = ds.Tables[0];
foreach (DataRow row in table.Rows)
{
   Console.WriteLine(row["CompanyName"]);
}
Console.ReadLine();
```

Note a couple of things about this code:

- Although you can reference columns by column number, it's more convenient to reference them by name as in this example. However, it does cause a bit of a performance hit because the name needs to be looked up to find the correct column.
- You could also reference the table by name rather than number, but you'd need to know what the table's name is (in this case, it's just "Table).
- There's no need to close the DataAdapter; that happens automatically. And since DataSet and DataTable are connectionless, there's nothing to close with them either.

Using interfaces

As I mentioned, to create generic code that can work with different data providers, you'll want to use interfaces: IDbConnection, IDbCommand, IDataReader, and IDataAdapter. For example, replace the command instantiating SqlDataAdapter with:

```
IDataAdapter adapter = GetDataAdapter(selectCmd, connString);
```

Here's the GetDataAdapter method:

```
private IDataAdapter GetDataAdapter(string selectCmd, string connString)
{
   // Make a decision about what type of DataAdapter to create, such as reading it
   // from a configuration file. In this demo, we'll just hard-code it as a
   // SqlDataAdapter.
   IDataAdapter adapter = new SqlDataAdapter(selectCmd, connString);
   return adapter;
}
```

GetDataAdapter decides what type of DataAdapter to create and returns a generic IDataAdapter. Client code doesn't care what type of DataAdapter it is, since the code is programmed to interface.

Updating data

Adding and deleting records to a database is straightforward with ADO.NET: use a DataCommand object with an INSERT or DELETE statement. Typically, you'll use a parameterized statement such as:

```
insert into Customers (CustomerID, CompanyName) values (@ID, @Name)
```

(The "@" is the ADO.NET equivalent of the ? parameter operator in VFP.) Unlike VFP, which can substitute parameters with any variable that's in-scope, you need to specify the parameter values in the command object's Parameters collection. Here's some code that adds a record using a parameterized statement:

```
string insertStatement = "insert into Customers (CustomerID, CompanyName) " +
   "values (@ID, @Name)";
SqlCommand insertCmd = new SqlCommand(insertStatement, conn);
IDataParameter idParameter = insertCmd.CreateParameter();
idParameter.ParameterName = "ID";
idParameter.Value = "ABCDE";
insertCmd.Parameters.Add(idParameter);
IDataParameter nameParameter = insertCmd.CreateParameter();
nameParameter.ParameterName = "Name";
nameParameter.Value = "Stonefield Software";
insertCmd.Parameters.Add(nameParameter);
int records = insertCmd.ExecuteNonQuery();
MessageBox.Show(records.ToString());
```

Note: this code uses the IDataParameter interface rather than the SqlParameter class to be more generic.

Here's some similar code that deletes the record just added:

```
string deleteStatement = "delete from Customers where CustomerID=@ID";
SqlCommand deleteCmd = new SqlCommand(deleteStatement, conn);
IDataParameter idParameter = deleteCmd.CreateParameter();
idParameter.ParameterName = "ID";
idParameter.Value = "ABCDE";
deleteCmd.Parameters.Add(idParameter);
int records = deleteCmd.ExecuteNonQuery();
MessageBox.Show(records.ToString());
```

Updating a database from changes to a DataSet (additions, deletions, or changes) can be done by populating the DeleteCommand, UpdateCommand, and InsertCommand members of the DataAdapter (there's also a SelectCommand member, filled automatically when you pass the SELECT statement to the constructor). You can either populate them manually by instantiating Command objects and passing them the correct SQL statements, which could be a lot of typing if there are a lot of fields in a table, or automatically using a CommandBuilder object. When you call the Update method of the DataAdapter, it uses these command objects to make any necessary changes to the database. The following code uses the CommandBuilder approach to update a record (note that the DeleteCommand and

InsertCommand objects aren't actually needed in this case but are shown for completeness). This code also shows an alternate way of specifying a parameter.

```
// Retrieve a specific record.
string selectCmd = "select * from Customers where CustomerID=@ID";
SqlDataAdapter adapter = new SqlDataAdapter(selectCmd, connString);
adapter.SelectCommand.Parameters.Add("ID", SqlDbType.VarChar, 5).Value = "ABCDE";
DataSet ds = new DataSet();
adapter.Fill(ds);

// Change the contact name.
DataTable table = ds.Tables[0];
table.Rows[0]["ContactName"] = "Doug Hennig";

// Create DELETE, INSERT, and UPDATE commands.
SqlCommandBuilder builder = new SqlCommandBuilder(adapter);
adapter.DeleteCommand = builder.GetDeleteCommand();
adapter.UpdateCommand = builder.GetUpdateCommand();
adapter.InsertCommand = builder.GetInsertCommand();

// Update the database.
int rowsUpdated = adapter.Update(table);
MessageBox.Show("updated " + rowsUpdated.ToString());
```

Resources

There are tons of resources online for learning C# and .Net. Here are some that I used:

- *C# 4.0 in a Nutshell*, by Jospeh Albahari and Ben Albahari (ISBN 978-0-596-80095-6)
- Pluralsight online training courses: http://www.pluralsight.com
- *.Net for Visual FoxPro Developers*, by Kevin McNeish (ISBN 1-930919-30-1)
- LINQPad: http://www.linqpad.net

Summary

I've only been using C# for a couple of years but found that it didn't that long to learn the syntax. I've found it's useful for some things that VFP can't do easily or at all. Learning a new language can be fun and can make you more marketable as a developer. Learning C# also makes it easier to learn JavaScript, so you get a double benefit from time spent with it.

Biography

Doug Hennig is a partner with Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT); the award-winning Stonefield Query; the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro; and the My namespace and updated Upsizing Wizard in Sedna.

Doug is co-author of *VFPX: Open Source Treasure for the VFP Developer*, *Making Sense of Sedna and SP2*, the *What's New in Visual FoxPro* series, *Visual FoxPro Best Practices For The Next Ten Years*, and *The Hacker's Guide to Visual FoxPro 7.0*. He was the technical editor of

The Hacker's Guide to Visual FoxPro 6.0 and *The Fundamentals*. All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). He wrote over 100 articles in 10 years for FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe. He currently writes for FoxRockX (http://www.foxrockx.com).

Doug spoke at every Microsoft FoxPro Developers Conference (DevCon) starting in 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the annual Southwest Fox conference (http://www.swfox.net). He is one of the administrators for the VFPX VFP community extensions Web site (http://vfpx.codeplex.com). He was a Microsoft Most Valuable Professional (MVP) from 1996 through 2011. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://tinyurl.com/ygnk73h).