Chapter 1: Programming Dynamically!

In This Chapter

- **Understanding dynamic typing**
- **Defining variables**
- **Putting dynamic to use**
- **Making static operations dynamic**

For many years, I thought that dynamic programming referred to being really flashy and flamboyant while writing code. So, I started wearing Hawaiian shirts and singing loudly.

Later, I found out this isn't the case.

Dynamic programming is another one of those buzzwords that really doesn't have a clear definition. At its loosest, it means developing something in such a way that the program makes more decisions about the way it runs while running, rather than when you compile it.

Scripting languages are a great example of this. When you write something in VBScript, you don't compile it at all — all of the decisions are made at runtime. Ruby is another good example: Most of the time, an entire program can just be typed into a command prompt and run right from there.

There are examples that are not so good — like VB Classic. Remember the Variant type? You could declare a variable to be Variant and VB wouldn't decide what it was supposed to be for real until the program ran. In the best of cases, this added immense flexibility to the language. In the worst of cases, you got Type Mismatch errors at runtime.

To give a concrete example, when you declare a variable in a dynamically typed language, you don't have to say what type you are making that variable. The compiler will just figure it out for you. In a static language, like C# 3.0, you do have to say what type you are making that variable.

Microsoft originally promised that dynamic types would never be in C#, but later decided that the feature had to be added. Why? Mostly, it's because of the development for Microsoft Office (like the reasoning for much of the rest of Book VIII). Office uses COM, the pre-.NET structure for Microsoft applications.

COM expects that the languages that use it (like VB Classic and C++) will have dynamic types. This made developing for Microsoft Office difficult for C# programmers, which was exactly opposite of what Microsoft wanted to happen. The end result? The dynamic type.

Shifting C# Toward Dynamic Typing

So-called "dynamic languages" are a trend that keeps coming back, like ruffled tux shirts. *Dynamic languages* are languages that allow for loose typing, rather than static. The concept got started in the 1960s with LISP. Dynamic languages came back in the late 1980s for two reasons: network management scripting and the artificial intelligence craze. Thanks to the Web, the buzzword is back yet again.

The World Wide Web, for those of you who aren't old enough to remember, was built on View Source and dynamic languages. Microsoft's original Web development language, Active Server Pages, was built on VBScript — a dynamic language.

The Web is better with a dynamic programming environment, so the trend is probably here to stay this time (until the next big thing, anyway). C# isn't the only language that is adding dynamic language features, and dynamic type isn't the only language feature that has been added to make it more appealing for Web programmers.

Several dynamic languages have been around for a while, like these:

- Perl
- Visual Basic
- Smalltalk
- LISP
- Scheme

While some of these aren't as popular as they once were, they are still out there and have pushed the trend in the newer languages. You can see this trend in all the new or refurbished dynamic languages that have popped up over the last ten years. Many of them have roots in the Web, while others are being newly used for the Web:

- PHP
- Ruby
- JavaScript
- Cold Fusion

- Python
- Cobra (my new favorite)
- Groovy
- Boo
- Newspeak

Programmers who work in dynamic languages — how can I put this gently — feel strongly about their choice of tools. The communities are very strong. Developers who work in dynamic languages use them for practically everything except highly structured team-build kinds of environments, like:

- Scripting infrastructure for system maintenance
- Building tests
- One-use utilities
- Server farm maintenance
- Scripting other applications
- Building Web sites
- File maintenance

Dynamic languages are popular for these kinds of tasks for two reasons. First, they provide instant feedback, because you can try a piece of code outside the constraints of the rest of the program you are writing. Second, you can start building your higher-level pieces of code without building the plumbing that makes it work.

For instance, Ruby has a command line interface that you can simply paste a function into, even out of context, and see how it works. There is even a Web version at `http://tryruby.hobix.com/`. You can type code right in there, even if there are classes referenced that aren't defined, because Ruby will just take a guess at it.

This moves nicely into the next point, that a dynamic language enables you to build a class that refers to a type that you haven't defined elsewhere. For example, you can make a class to schedule an event, without actually having to build the underlying Event type first.

All of this lends itself to a language that is a lot more responsive to change. You can make a logic change in one place and not have to dig through reams of code to fix all the type declarations everywhere. Add this to optional and named parameters (see Chapter 2) and you have a lot less typing to do when you have to change your program.

Other benefits to dynamic languages in general show up as you use them more. For instance, macro languages are usually dynamically typed. If you have tried to build macros in previous versions of Visual Studio, you know what a pain it is to use a static language.

Making C# (and VB.NET, for that matter) more dynamic not only makes it a better language for extending Visual Studio, but it also gives programmers the capability to include the language in the programs they write so that other developers can further extend those applications.

Programming Dynamically

By now, you must be asking, "What exactly are we talking about here?" Fair question. When you define a new variable, you can use the `dynamic` keyword, and C# will let you make assumptions about the members of the variable.

More or less, what I'm talking about it this. If you want to declare a new Course object, you do it like this:

```
Course newCourse = new Course();
newCourse.Schedule();
```

This is, of course, assuming that you have a Course class defined somewhere else in your program, like this:

```
class Course {
   public void Schedule()
   {
       //Something fancy here
   }
}
```

But what if you don't know what class the new object will be? How do you handle that? You could declare it as an Object, because everything derives from Object, right? Here's the code:

```
Object newCourse = new Object();
```

Not so fast, my friend, if you make your next line this:

```
newCourse.Schedule();
```

Note the squiggly line appears almost immediately, and you get the famous "object does not contain a definition for Schedule..." error in the design time Error List.

However, we can do this:

```
dynamic newCourse = SomeFunction();
newCourse.Schedule();
```

All this code needs to have is the stub of a function that returns some value, and we are good to go. What if `SomeFunction()` returns a string? Well, we will get a runtime error. But it will still compile!

About now, if you are anything like me, you have to be thinking: "This is a *good* thing? How!?!" I hear you, trust me. For the time being, you can blame COM.

You see, COM was mostly constructed using C++, which has a variant type. In C++, you could declare a variable to be dynamic, like this:

```
VARIANT newCourse;
```

It worked just like the dynamic type, except C# wasn't invented yet. Anyway, because a lot of the objects in COM used Variant out parameters, it was really tough to handle Interop using .NET.

Because Microsoft Office is mostly made of COM objects, and because it isn't going to change any time soon, and because Microsoft wants us all to be Office developers one day, bam, we have the dynamic type.

Say, for instance, that our `newCourse` is a variant out parameter from a method in a COM class. In order to get the value, we have to declare it an Object, like this:

```
CourseMarshaller cm = new CourseMarshaller(); //a COM object
int courseId = 4;
Object newCourse;
cm.MakeCourse(courseId, newCourse);
//and now we are back to square one
newCourse.Schedule(); //This causes a 'member not found
   exception'
```

Line six will not compile, even if the Schedule method exists, because we can't assume that `newCourse` will always come back as a Course object, because it is declared a variant. We are stuck.

With a dynamic type, though, we are golden once again, with this code:

```
CourseMarshaller cm = new CourseMarshaller(); //a COM object
int courseId = 4;
dynamic newCourse;
cm.MakeCourse(courseId, newCourse);
newCourse.Schedule(); //This now compiles
```

What happens if `newCourse` comes back as something that doesn't have a Schedule method? You get a runtime error. But there are `try/catch` blocks for runtime errors. Nothing will help it compile without the dynamic keyword.

Readers who are long-time Visual Basic programmers, or even newer VB.NET programmers, realize that you can handle this dynamically — and have always been able to — in Visual Basic. For a long time, I have recommended that programmers working with legacy systems use Visual Basic for their new code, and this is exactly why.

In the interest of language parity, now C# can do it, too. In general, this is good, because many organizations are writing legacy code in VB and new code in C# — and it can get pretty messy in the trenches. This change makes the code base slimmer.

Putting Dynamic to Use

When C# encounters a dynamic typed variable, like the variables we created earlier, it changes everything that variable touches into a *dynamic operation.* This dynamic conversion means that when you use a dynamically typed object in an expression, the entire operation is dynamic.

Classic examples

There are six examples of how this works. Say we have the `dynamic` variable `dynaVariable`. Because the dynamic variable will pass through all six examples, they will all be dispatched dynamically by the C# compiler. Here are those examples, with nods to Daniel Ng.

- **`dynamicVariable.someMethod("a", "b", "c");`**: The compiler binds the method `someMethod` at runtime, since `dynaVariable` is `dynamic`. No surprise.
- **`dynamicVariable.someProperty = 42;`**: The compiler binds the property `someProperty` just like it did in the first method.
- **`var newVar = dynamicVariable + 42;`**: The compiler looks for any overloaded operators of "`+`" with a type of `dynamic`. Lacking that, it outputs a `dynamic` type.
- **`int newNumber = dynamicVariable;`**: This is an implicit conversion to `int`. The runtime determines if a conversion to `int` is possible. If not, it throws a type mismatch error.

- **`int newString = (int) dynamicVariable;`**: This is an explicit cast to `int`. The compiler encodes this as a cast — you actually change the type here.
- **`Console.WriteLine(dynamicVariable);`**: Because there is no overload of `WriteLine` that accepts a `dynamic` type explicitly, the entire method call is dispatched dynamically.

Making static operations dynamic

If the compiler chooses to make a static operation dynamic — as it did in item 6 in the preceding section — the compiler rebuilds the code on the fly to have it handle the dynamic variable. What does that mean for you? Glad you asked.

Let's take item 6, `Console.WriteLine(dynamicVariable);`. This piece of code forces the compiler to build intermediary code, which checks for the type of variable at runtime in order to come up with something that is writable to the console. The compiled code first checks if the input is a static type that it knows. Next, it checks for a type present in the program. Then it will just try a few things that might work. It will fail with an error if it finds nothing.

If this must happen, that's fine. But remember that it is slower than all git out. This is why `Variant` got such a bad rap in Visual Basic classic. Dynamic is something you don't use until you need it. It puts a tremendous strain on the machine running the program, especially if all variables are dynamic.

Understanding what's happening under the covers

Let's take an example right out of MSDN. Microsoft points out this simple method:

```
class C
{
    public dynamic MyMethod(dynamic d)
    {
        return d.Foo();
    }
}
```

This is pretty straightforward stuff — a method that accepts a dynamic class and returns the results of the type's `Foo` method. Not a big deal.

Here is the compiled C# code:

```
class C
{
    [return: Dynamic]
    public object MyMethod([Dynamic] object d)
    {
        if (MyMethodo__SiteContainer0.p__Site1 == null)
        {
            MyMethodo__SiteContainer0.p__Site1 =
              CallSite<Func<CallSite, object, object>>
              .Create(new CSharpCallPayload(
                CSharpCallFlags.None, "Foo",
typeof(object), null,
                new CSharpArgumentInfo[] {
              new CSharpArgumentInfo(CSharpArgumentInfoFl
ags.None,
              null) }));
        }
        return MyMethodo__SiteContainer0.p__Site1
          .Target(MyMethodo__SiteContainer0.p__Site1, d);
    }

    [CompilerGenerated]
    private static class MyMethodo__SiteContainer0
    {
        public static CallSite<Func<CallSite, object,
object>> p__Site1;
    }
}
```

Yeah, that's what I said, too. I am not going to even begin to try breaking down this code — and fortunately, we don't have to. That's what we have compilers for, right?

Running with the Dynamic Language Runtime

There is more to dynamic languages than just the dynamic typing. You can do some powerful things. Like all power, you have to be careful not to misuse it.

The Dynamic Language Runtime — shown in Figure 1-1 — is a library added to the .NET Framework specifically to provide for adding dynamic languages (like Ruby) to the Visual Studio fold (like IRONRuby), or to add dynamic language features to existing static languages (like C# 4.0).

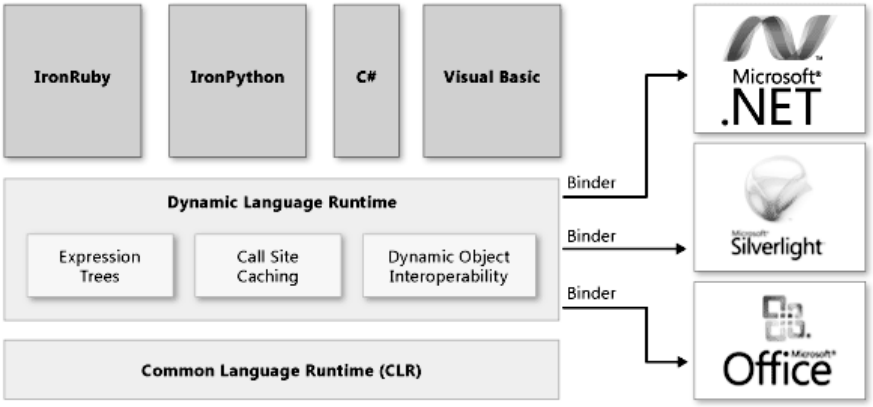

Figure 1-1: The Dynamic Language Runtime.

The runtime helps the compiler to construct code in the compiled assembly that will make a lot of choices dynamically. The code block at the end of the preceding section is an example of the simplest kind.

The DLR assisted in the creation of IRONRuby, which makes it possible to code in Ruby — the current hot dynamic language — right in Visual Studio. Of course, because the DLR enables C# to take on dynamic language features, much that you can do in Ruby you can now do in C#.

Dynamic Ruby

Ruby takes advantage of its dynamic roots in its implementation of the Trabb Pardo-Knuth algorithm. Don't be put off by the name — this is just a straightforward problem that can be solved by computer code.

The program needs to read 11 numbers from an input device — in our case, the console's `ReadLine` method. It stores them in an array. Then, it processes the array backward — starting from the last entered value — with some function. If the value doesn't exceed some arbitrary threshold, it prints the result.

The program looks like this in Ruby:

```
class TPK
  def f( x )
    return Math.sqrt(x.abs) + 5*x **3
  end
```

```
  def main
    Array.new(11) { gets.to_i }.reverse.each do |x|
      y = f(x)
      puts "#{x} #{(y>400) ? 'TOO LARGE' : y}"
    end
  end
end
```

This isn't a Ruby book, and that fact isn't lost on me. Nonetheless, this is the best dynamic language that I can use for an example — bar none.

Two functions are defined: `f` and `main`. `Main` accepts 11 numbers from the console and then moves them to an integer array (that's what `gets.to_i` does). For each value in the array, it sets `y` equal to `f(x)` and then sees if it is higher than our arbitrary value. If so, it prints "TOO LARGE"; otherwise, it prints the number.

Why is being dynamic important for this algorithm? It isn't. You could do it all statically typed. The dynamic bit does have an impact, though.

First, `f(x)` doesn't care what `x` is. The program assumes that whatever comes in gets changed to an integer at `gets.to_i`, but the function itself is case agnostic. This is good and bad, because if we do happen to give it a string or some other type, it will fail.

The array itself isn't typed, either. This can have benefits, because it is possible to drop a differently typed value in there if you know you are just going to write it to the screen.

Dynamic C#

Of course, C# now has similar features, right? We should be able to do the same thing! Yes, in fact, we can. Here's the code:

```
static dynamic f(dynamic x)
{
    return (Math.Sqrt(x) + 5.0 * Math.Pow(x, 3.0));
}
static void Main(string[] args)
{
    dynamic[] array = new Array[11];
    for (int i = 0; i < 11; i++)
    {
        array[i] = Console.ReadLine();
    }
    for (int i = 10; i>=0; i--)
    {
```

```
                dynamic y = f(array[i]);
                if (y > 400.0)
                {
                    Console.WriteLine(string.Format("{0} TOO
LARGE", i));
                }else{
                    Console.WriteLine("{0} : {1}", i,
array[1]);
                }
            }
            Console.ReadKey();
        }
```

Line for line, the application does the same thing as the Ruby code, albeit longer. I kept the names the same so it was easier to follow. Because I had to use `for` loops to handle the integrators, it made the body of the program quite a bit beefier. Figure 1-2 shows what the program looks like when it runs.

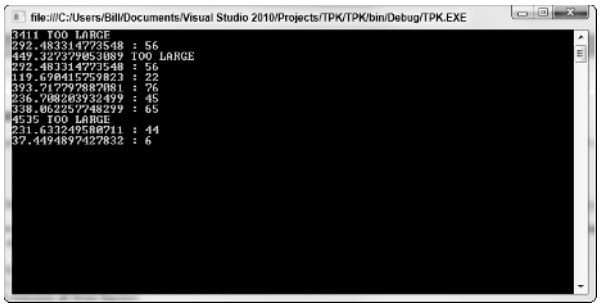

Figure 1-2: The TPK program running.

But why use the dynamic type here? Clearly we could have just used `double` for this. Use of dynamic just made the program easier to create. Try changing the array to an array of `double`, like this:

```
Double[] array = new Double[11];
```

Hey, look at that: Now the `ReadLine` doesn't work. We'll just cast it to a `double`. Nope, can't do that; we have to use `TryParse`. You get the picture. Static types are hard to code with. Dynamic types are easier to code with.

What's the other side of this? Well, obviously, if the user enters a string, she gets a runtime error, and that is bad. If we statically type everything, then we can trap that error much easier, and handle it right on user input.

Add to that the reality that C# is making runtime decisions about every single variable throughout the entire run of the program. That's a whole lot of extra processing that we could have avoided if we had just done that static typing.

The take-home here is that using dynamic types makes your programming job much easier and your troubleshooting job much harder. If you are writing a utility script for your own use, and don't care if it occasionally crashes with a type mismatch, then use dynamic. If you are writing a backup script for a hospital and the lives of thousands are at stake, I advise static types.