

## **Who Am I?**

My name is Eric. I work for Eastern Maine Healthcare Systems as a Programmer / Analyst.

I teach part-time (junior-high level robotics / computer class). If you have questions, please raise your hand, and I'll call on you. :) I run a small business called RLI Solutions. I do website and user-interface reviews, write custom tutorials and help files, and I automate office tasks.

I can be reached at: [eric@rlisolutions.com](mailto:eric@rlisolutions.com) or <http://rlisolutions.com/>

### **What is VB Classic?**

VB Classic is a collective term to refer to anything in the Visual Basic family that is not VB.Net. This could be VB3, VB4, VB5, VB6, VB Script, Office VBA or even Classic ASP. For today's discussion, we are going to be targeting VB6, although much of what we will talk about can be applied to most of the VB family.

### **What is VB.Net?**

VB.Net is the next version of Visual Basic after VB6. While it does have some backwards compatibility with VB 6, the upgrade path is not always clear.

### **New Language Features**

VB 6 was a compiled language. You wrote code, and that code was then compiled to machine language and executed on a computer.

.NET uses a JIT (Just In Time) compiler. You write code, that code is compiled to an intermediary language (called MSIL - Microsoft Intermediary Language) and then the MSIL code is compiled to the native code by the JIT compiler on the target computer. This is referred to as Managed Code. Some of the .NET languages let you write Unmanaged Code. But VB.Net doesn't allow unmanaged code at this time. For those of you who seem shocked that Microsoft would move away from a true compiled program.... It's the same method that Java has been using for years. Java just calls their intermediate language ByteCode.

Having your language use a JIT compiler means you can run the same program on any computer that has an available JIT compiler, regardless of the operating system. Java will run on Windows, Mac, and Linux - without making changes to the program. Currently, the only Microsoft supported operating system is Windows. There are third-party implementations of a JIT compiler that allow a .NET application to run on a Mac or Linux machine. Mono is the most popular, but there are a few other projects in development.

#### Other new features:

- Threading
- Garbage Collection
- New UI controls – Take advantage of Themes in the OS
- Block level variables
- Variable Changes
  - Integers are now 32 bit.
  - Longs are now 64 bit
  - Shorts are included, these are 16 bit
  - Unsigned Variables
  - Variants are dead. Long live objects!
- Short Circuit Condition
- New Operators:
  - IsNot
  - OrElse
  - AndAlso
- Ability to override Methods and Operators
- Partial Classes
- Continue Statement
  - Skips to the next iteration of a loop

#### IDE Enhancements

- Automatic indentation
- Better syntax checking
- Task List
- Code Snippets
- Macros
- Import/Export your IDE settings

## Missing VB 6 Features

There are some things that may seem like a big deal because they are missing in VB.Net. Once you see the replacements for those features, it's not as scary. It does mean that if you used some of these features, you will have to change how your application behaves to move it to .NET

- There are no controls arrays in VB.Net
- Variants are gone
- No more NULL propagation (appending NULL characters to string)
- DHTML Applications have no equivalent.
- ActiveX Documents have no equivalent.

## Why Upgrade?

There are several reasons to upgrade. And several reasons to NOT upgrade. I can't make that decision for you... you have to make that choice.

### Possible Reasons to Stay in VB 6

- Your application does not significantly benefit from the .NET features, and will not be receiving any new features.
  - Bug fixes are not a reason to upgrade.
- You use DAO, property pages, or VB 5 controls.
  - These are not supported by VB.NET, lots of rewrite
- You wrote a game in VB6.
  - Graphics have changed significantly.

### Possible Reasons to Upgrade to .NET

- You are going to do a major overhaul anyhow.
- You need a specific feature(s) of the .NET framework:
  - Threading
  - Increased security
  - Themed UI
- The need for forward-compatibility.
- Console applications
- Many performance increases.
  - According to VB Internals, a similarly written VB.NET app will significantly outperform its VB 6 counterpart.

Since this presentation is about upgrading from VB 6 to VB.Net, I'm going to focus on upgrading your skills, vs. specifics of upgrading your application.

Upgrading your skills involves learning more about .NET than what the Upgrade Wizard tells you to fix. It also means you are more valuable in the long run.

You can then decide which of the three options you want to take when you migrate your applications to VB.Net.

## **What are my upgrade options for my program?**

You have several options when considering an upgrade.

1. Don't upgrade – Keep using VB 6.
2. Try using the Upgrade Wizard and upgrade your project.
3. Rewrite the program using VB.Net

### **Don't Upgrade – Keep using VB 6.**

This is a serious option. Many applications will not benefit much from the upgrade, and the time and resources necessary to migrate to VB.Net may not be worth the benefit. So... before you upgrade, consider leaving the application as a VB Classic program. But... since this option kind of defeats the purpose of tonight's discussion, let's move on.

### **Try using the Upgrade Wizard and upgrade your project.**

The Upgrade Wizard was created by Microsoft to help developers migrate their projects from VB6 to VB.Net. Unfortunately, the migration tool is not that robust, and it only works well on small projects. And by small, I mean less than 500 lines of code.

Visual Basic has always had a very lax syntax. There are multiple ways to do things, some better than others. But this kind of lazy syntax makes writing a parser difficult. That's one of the reasons why the Upgrade Wizard doesn't work as well as it could.

The Upgrade Wizard is only available to Standard and higher editions of Microsoft Visual Studio. I'm using Visual Studio Express 2005, so I can't demonstrate the wizard. I do have some sample programs that were upgraded, so you can see the notes.

### **Rewrite the program using VB.Net**

A lot of people don't like to hear this option. But it is the most common. And usually the best approach if you have to upgrade! *Remember Bullet #1. Upgrading is not mandatory.*

Since the Upgrade Wizard doesn't work as well as it's name implies, you are left with rewriting the program. Frequently, you can copy whole routines from the old VB Classic application into the VB.Net application, and they will run, or they will require minimal changes.

However, there are some significant changes in the way that VB.Net functions, and if you rewrite the routines, you will often get better performance.

## **Option Strict – Finally! HURRAH!**

First, a bit about Option Explicit. This still exists in .NET. This command prevents the compiler from compiling and executing your code, if a variable is not declared. In VB.NET, this is turned ON by default for new projects. In VB 6, this was off by default, and you had to explicitly activate it.

Option Strict is a new compiler directive. Unfortunately, it is turned off by default, but you can easily turn it on in the options.

Option Strict prevents you from doing implicit conversions. For Instance: In VB 6, you can assign a number to a string. <<see code example: Samples – ImplicitConversion>>

With Option Strict On in VB.Net, this will generate a compiler-time error. You would have to explicitly convert the number to a string. <<see code example: Samples – ExplicitConversion>>

This may not sound like a good thing... but it is. Your code is safer now, since you explicitly know what is going on. No more behind the scenes type changes to cause obscure bugs or holes in your application.

This will compile OK with Option Strict Off, but will fail during runtime. <<see code example: Samples– ImplicitConversionFailed>>

Option Strict On will prevent late-bound calls. But you can put all of those late-bound methods into their own module or class (this is a great example of where to use Partial Classes), and turn off Option Strict for just that module.

## **I, Object!**

Everything pretty much is an object in VB.Net.

So, you can get the length of a string, by typing `myString.Length`. You can see if a string variable contains a specific value, by using: `myString.Contains("Hello")`

You can convert an integer to a string, by using: `myInt.ToString`

## **Performance: Developer time and Runtime**

Do you use just the "VB.Net super-cool new methodologies" to do your work, or can you use the VB Classic methods that you are familiar with?

The answer? It Depends!

The Visual Basic Internals document (see link at the end of this guide) compares many of the common methods that can be called multiple ways. It compares the new System namespace calls, to the Visual Basic Runtime calls, and shows you where it is better to use the System calls, and where it doesn't matter. Sometimes, it's even better to use the VB Runtime methods.

### **LEN vs. String.Length**

These both return the length of a string. They both produce almost the same IL code. However, the `LEN()` statement can save you a few steps at design time.

In .NET, a string can contain Nothing. This isn't an empty string, it is a null reference. If you try to check the length of a string containing nothing, `LEN()` will return 0 as a result (the string contains nothing). `myString.Length` will produce an exception. Using `LEN()` saves you an extra check.

### **Replace vs. String.Replace**

Strings in VB.Net are still immutable. <<ask if anyone needs clarification on immutable strings>> They cannot be changed once they are created. Both methods are identical, and both perform poorly on a large string with lots of replacements. Using `StringBuilder.Replace` can produce significant performance increases.

## **Mid, Left, Right**

These all call the `myString.Substring()` function. However, it's often easier for the developer to remember `myString = Right(myString, 3)` than it is to remember: `myString = myString.Substring(myString.Length - 3)` or: `Left(MyString, 3)` vs.

`myString.Substring(0, 3)`

<<see code example: Samples – StringParsing>>

With the use of `Mid` vs. `Substring`, the calls are about the same.

If you want to use `Mid` as a way to assign characters to a string, the best .NET equivalent is to use the `StringBuilder` class.

## **Directory Work:**

`CurDir` and `Dir` should be avoided. Using the new `System.IO` methods is generally faster, and the preferred approach. Same with the `FileSystemObject`. Use the `System.IO` of the `FileIO` methods instead.

<<see code example: Samples – FilesFolders>>

## **Shell**

Use the `Process` class. Much MUCH more powerful.

## **MsgBox vs. MessageBox.Show, InputBox**

`MsgBox` just wraps the `MessageBox.show` command. So it's your choice. I prefer `MessageBox.Show` for no reason that I can really defend. It's personal choice on this one.

`InputBox` just wraps a custom form behind the scenes. It actually creates an instance of a form with a textbox and some labels and buttons. If you have to use the `InputBox`, then go ahead. Although it's usually a poor user interface design to use it. But that's outside the scope of tonight's meeting.

## **Rnd vs. System.Random**

`Rnd` still works, but `System.Random` has more options and better performance if called repetitively.

## **Working with Dates:**

Not enough time to cover all of the neat features of Dates, but definitely use the new Date types. You can easily add/subtract dates, find the differences between two dates, etc. <<see code example: Samples – DateTesting>>

The IsDate function is a VB Runtime method. It's a fast and efficient way to see if a date is in a valid format. Use it if needed.

## **Conversions**

CInt, CLng, CType(), DirectCast(), System.Convert – they all do the same basic thing. There really isn't much of a performance gain in most cases. The CInt, CStr, etc methods are from VB6 and work just as well in most situations. DirectCast() can give you a slight performance increase in certain situations. Research it if you want more information.

## **UBound vs. Array.GetUpperBound**

UBound just wraps the Array.GetUpperBound methods, and they compile to the same IL code. Use the one you want.

## Control Arrays

Visual Basic .NET does not support control arrays in the Visual Basic 6 style. The compatibility library provides a set of container classes that mimic control arrays by associating indices with controls. Controls are added to a container, such as an instance of `VB6.ButtonArray`. The container handles the events raised by its contained controls and then raises its own corresponding events. So when a user clicks on a `Button`, the `Button.Click` event is handled by the container control, which then raises its own `Click` event.

The control array classes add an extra layer of indirection for event handling that is not discernible in a typical GUI application. The bulk of the work for a control array class happens during initialization. The extra cost of adding a control to a control array container is measurable with a large number of controls but certainly not noticeable for a user. For example, adding 500 `Button` controls to a control array takes about 20 percent longer than just adding 500 `Button` controls to a form, but the difference is a mere fraction of a second.

In Visual Basic 6, you were required to use a control array if you wanted a single method to handle events from multiple objects. With Visual Basic .NET, you can point events from many objects (even different types of objects) at the same method with the **Handles** keyword:

<<see code examples: Samples>

```
Private Sub ColorChoice_CheckedChanged(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles optRed.CheckedChanged, _  
    optGreen.CheckedChanged, optBlue.CheckedChanged  
  
End Sub
```

## **Block Scope for Variables**

Variables in VB 6 had a rather loose scope. You could declare a variable anywhere in a procedure (including at the end) and it had a lifespan of that procedure.

In VB.Net, you now have block scope for variables. Where you declare your variables is important. If you declare them inside an If/Then statement, then that variable is only available to code inside that If/Then statement. This applies to just about any block level syntax... such as Try/Catch, If/Then, loops, with, using, etc.

## **Exceptions Vs. Error Handling**

New syntax. <<see code examples: Samples>>  
Exceptions bubble up the stack the same way they did in VB 6.

## **Strings, StringBuilders and Performance, oh my!**

As I mentioned before... strings are immutable. Multiple string concatenations can be very costly. .NET introduces a new class called StringBuilder. This class is used for building strings (especially useful in loops). It optimizes the way that strings are created and managed, so you get less throw-away variables, which means less garbage to collect.

## **Garbage Collection**

New garbage collection. For most developers, there is really nothing you need to be aware of, other than the garbage collection is being done behind the scenes.

## **API Calls**

Most of the things that API calls were necessary for in VB Classic, is now part of the .NET System namespace. For the items that aren't, you can use what is referred to as a "P/Invoke". The P/Invoke website is a great place to find samples of functionality. Just bear in mind... P/Invokes are a step outside of managed code, and are not cross-platform compatible.

## **ClickOnce Deployment**

.NET has a new Deployment method called ClickOnce. It allows you to deploy a program from a website, network share or media (USB drive, CD-Rom, etc).

The ClickOnce application can automatically update the application if a new version is available. It can also control some security settings, and control whether or not an application can be run offline or not.

I do not have a lot of experience using ClickOnce, so all I can say is... search online and play with it.