1 Language Selection

The chapters in this part of the book cover activities that occur before programming begins. They describe development and design approaches, and some useful design techniques. Many programmers skimp on these phases, and later pay a huge price when their initial assumptions prove inappropriate for the application.

This chapter describes some of the issues that you should consider when picking a development language. The book assumes you will use Visual Basic, but this chapter describes some of the language's shortcomings. While you may stick with Visual Basic, it's worth understanding some of the language's weak spots and what you can do about them. If you know where trouble may arise, you can more easily cope with it. In extreme cases, you may be able to move some of the more troublesome modules into a library written in another language such as C# or C++, and call the library from your Visual Basic code. For example, C++ and C# handle pointers more transparently than Visual Basic does. If you need to use a library to control special-purpose hardware, it may be easier to call that library's routines from C++ or C#, particularly if the library was written in one of those languages.

One of the other times I've found this sort of thing useful is with "unsafe" array manipulation for graphics processing. A program can obtain a pointer to the memory containing an image's pixel data. A C# program can treat the pointer as a reference to an array, whereas a Visual Basic program must use API functions to copy the pointer's memory into a new array and, after the program has finished manipulating the array, it must copy the result back to the pointer's location. If you place the graphics-processing code in a C# module and call it from Visual Basic, you can avoid these two memory copies and save some time.

Language Addiction

If you surf the Web looking for reviews of languages, you'll find a very strong correlation between the programming language a person uses and the language he or she thinks is best. You can find lots of C# and C++ programmers who can tell you all about the bad points of Visual Basic, and plenty of Visual Basic developers who will tell you at length what's wrong with C# and C++. When you use a language extensively, that's the one you find most natural, and anything else seems strange.

If you speak to developers who have used several languages extensively, you'll generally get a very different response. These developers tend to say that most languages do pretty much the same thing, just in different ways, and the choice isn't that big a deal. In particular, I know a lot of programmers who have used both Visual Basic and C# for several years and they agree that the languages are practically identical. Many have a personal preference, but very few people with a lot of experience claim that one language is clearly superior to all others in all situations.

During my career, I've used quite a few languages, including Pascal, assembly, FORTRAN, C, C++, Delphi, PostScript (many developers don't even know it's a programming language), some graphics programming languages, various flavors of BASIC, and an assortment of Visual Basic versions such as VBA, VBScript, and Visual Basic versions 3 through 2005. Each of these languages has its strengths and weaknesses.

Disadvantages

The following section describes some of the most common objections raised against Visual Basic and explains whether these are real, significant objections; a problem that is easy to work around; or merely a matter of perception.

Interpreted, Not Compiled

Visual Basic does not produce a compiled executable. Instead it produces Intermediate Language (IL) code that is interpreted at run-time by the Command Language Runtime (CLR). This has a couple of important consequences.

First, interpreting code reduces execution speed. Visual Basic uses a just-in-time (JIT) compiler so after a program has executed a piece of code, it should be compiled and run at faster speeds later. However, running the code the first time will still be a bit slower than it will be in a truly compiled language.

Second, a computer must have the CLR and the .NET Framework installed to interpret the Visual Basic program's IL code. These not only increase the installation complexity, but they are also quite large and getting larger with each new release. Whereas the Visual Basic 6 run-time libraries took about 1MB compressed, the .NET Framework 2.0 needed to run Visual Basic 2005 programs weighs in at 22.4MB compressed, and expands to 280MB of disk space when installed (610MB if you're running 64-bit Windows).

Third, IL code is relatively easy to read, so even distributed as an "executable," a Visual Basic or C# application is relatively easy to reverse-engineer. Code obfuscators that rearrange code, rename variables so they are harder to understand, and so forth are available, but the result is still easier to read than a compiled executable.

Lutz Roeder's excellent "Reflector for .NET" program allows you to easily view IL code for .NET assemblies written in Visual Basic, C#, C++, IL, Delphi, and Chrome. It will even display code in those *languages that could have generated the IL code. For example, you can load a compiled Visual Basic application and Reflector will show you C# code that is roughly equivalent. This tool is also very useful for comparing the IL generated by similar pieces of Visual Basic and C# code — the results are not always the same. You can download Reflector and some other useful tools at* `www.aisto.com/roeder/dotnet`.

How important are these objections? Programmers often focus on speed but, as I'll argue later in this book, usability and correctness are more important than speed for most applications. If an application spends most of its time waiting for user input and calling a database engine, pure CPU speed isn't an issue. With today's fast computers and JIT compilation, speed is usually not a problem.

C# uses IL just as Visual Basic does, so there's no reason to think using C# will improve performance. Some languages such as Delphi may give better performance because they use better compiler optimization techniques. Assembly can also give better performance, but only if you write good assembly code, and the difficultly is rarely worth it.

Installing huge run-time libraries can be a serious issue. If you're developing software for use within your company, you may be able to require users to download a 22.4MB file that uses 280MB of disk space when installed. However, if you want to allow customers to download your program from the Internet, this may be a problem. For example, a 56K modem would take almost an hour to download the file. Once customers have the run-time libraries installed, they will not need to download them for future Visual Basic applications, but downloading them the first time could be problematic. Even with Visual Basic's new Click-Once Deployment, it's much easier for the user to run a self-installing Java application on whatever browser he or she prefers.

Finally, IL security can be a big issue for some applications. Many companies are protective of their software, and the chance of reverse-engineering has made many of them reluctant to adopt Visual Basic .NET. For many applications, you can easily deduce the code from the user interface. Any programmer can figure out how to make menus, dialogs, and reports. Often, there is only a small part of the application that is proprietary. If security is a real concern, you may be able to move that small piece of code into a library written in a truly compiled language such as C++.

When considering whether the reduced security provided by IL is a show-stopping issue, ask yourself these questions:

- How likely is it that someone will try to reverse-engineer the code?
- Does the program contain real proprietary information that you need to protect? Or, is most of the code obvious from the user interface?
- Can you pull the sensitive pieces out and compile them into a compiled library?

You can also use obfusticators to deter casual hackers. Some of the latest obfusticators do a pretty good job of making the code so confusing that it will be easier for a hacker to rewrite the application from scratch, rather than to sift through the scrambled code. If you do use an obfusticator, you should also use Lutz Roeder's Reflector program to see what the resulting IL looks like. If you're not confused by the result, a hacker won't be either.

Language Features

Visual Basic has an intuitive, self-documenting syntax but it also has a few annoying quirks. In an attempt to make every customer happy, Microsoft has left in some features and options that it probably should have removed. These are generally syntactic annoyances and don't cause much trouble if you are aware of them:

- *Strict Type Checking* — Type checking in Visual Basic .NET is stricter than it was in Visual Basic 6, but it's still looser than in some other languages. For example, by default, Visual Basic 2005 doesn't let you assign an Integer to a String (`s = 12`), but it will allow you to assign an Integer to a Long (`long_var = integer_var`). Sometimes that's what you want to do, but sometimes it's an indication that there's something wrong with your design. It also costs some execution time as the value is converted from one data type to another.
- *Option Explicit* — This option shouldn't be optional, it should always be on. I've never seen an example where turning this option off was necessary.
- *Option Strict* — This option should also always be on. Unfortunately, there are very rare cases when it's convenient to bypass type checking and turn Option Strict off (for example, when coercing parameters in and out of library function calls). If you must turn Option Strict off, do so in the smallest module you can build to get the job done. Use partial classes, if necessary, so that Option Strict is on in the rest of the application.
- *IIF* — Generally, an IIF statement is harder to read than a corresponding multi-line `If Then Else` statement, although I have seen cases where IIF can make a series of very simple statements easier to compare. IIF is also less efficient than `If Then Else` because it always evaluates both results even though only one is used. Usually you can avoid IIF.
- *Pointers* — Visual Basic uses references to refer to objects rather than actual pointers. That makes working with true pointers somewhat awkward. For example, if an API routine returns a pointer to a chunk of memory that contains one of several different data structures, it's more difficult to bully the result into the right kind of Structure type in Visual Basic than it is in languages such as C++ that use pointers explicitly. However, explicit use of pointers and manual memory allocation leads to some of the most difficult kinds of bugs to find in C++. The times when I've missed pointers have been pretty rare, so on the whole, Visual Basic is better off with references. The Marshal class provides methods that coerce pointers from one data type to another, so usually a Visual Basic program can produce the correct results with a little extra work.
- *Strings* — In Visual Basic, a String is more than a pointer to a series of characters in memory as it is in C and C++. If your application performs extremely complex string manipulation, those languages may give better performance. On the other hand, you can convert Strings in Visual Basic into an array of characters, and then manipulate the entries in the array. You couldn't use C-style character pointer arithmetic, but you can use array indices.
- *Garbage Collection* — The garbage collection techniques used by .NET take control of memory management away from the developer. The garbage collector can run at any time, and you have almost no control over when or how it does its job. You also cannot tell when an object will finally be destroyed and its resources freed (this is called *non-deterministic finalization*). That forces you to implement other methods such as the IDispose interface to allow the program to free resources before an object is garbage-collected. The garbage collector makes some of these issues more complicated than necessary, but they should only be a real showstopper in real-time applications.

A famous story from the MIT artificial intelligence laboratory illustrates how garbage collection can hurt real-time systems. A robotics application was supposed to track a thrown ball and catch it with a robot arm. In one trial, the machine vision system found the ball, tracked its trajectory, calculated its future trajectory, and then stopped for garbage collection. By the time garbage collection had finished and the computer moved the robot arm, the ball had hit the floor and bounced away.

- ❑ `AndAlso` *and* `OrElse` — The `AndAlso` and `OrElse` operators perform logical evaluation with short-circuiting. For example, consider the statement `A AndAlso B`. If `A` is `False`, then there's no reason to evaluate `B`, so Visual Basic doesn't bother. In the statement `A And B`, Visual Basic evaluates both `A` and `B`, even if `A` is `False`. If `A` and `B` are simple Boolean values or variables, this doesn't matter much. If they represent time-consuming function calls, `AndAlso` and `OrElse` can be much faster than `And` and `Or`. It would probably be better if `And` and `Or` used short-circuit evaluation, because that is generally more efficient, but Microsoft left the definitions of `And` and `Or` the way they were in Visual Basic 6 for backward compatibility.

The only time when you would want to always evaluate `A` *and* `B` *is if* `A` *and* `B` *were routines that have side effects such as updating global variables or opening files. Writing functions with side effects such as these is a bad programming practice, so this shouldn't be an issue anyway.*

- ❑ *Routine Libraries* — Visual Basic doesn't allow you to build a library that contains nothing but functions and subroutines. Instead, you must create a class library. The user of the library must make an instance of the class, and then use the object's methods to take action. Though this can sometimes be an annoyance, it's not too hard to live with.
- ❑ *Inline Code* — Visual Basic does not allow you to include inline code written in other languages. For example, you cannot include a small section of assembly or C# code to handle a particularly tricky or CPU-intensive task. You can, however, place that code in a separate library written in another language and call it from Visual Basic, although there will be a slight overhead in making the library call.
- ❑ *Macro Expansion* — Visual Basic does not allow inline macro expansion as provided by C++. This kind of macro expansion replaces a macro reference with a chunk of code at compile time. You can get much the same effect by placing the code in a subroutine instead of a macro definition. This may make the code slightly less efficient, but it makes issues such as setting breakpoints in the code much simpler.

Multiple Inheritance

Some languages support *multiple inheritance* (where a class can inherit from more than one parent class). For example, if you have a `Vehicle` class and a `Domicile` class, you could make the `MotorHome` class inherit from both to gain the properties and methods of both `Vehicle` and `Domicile`.

Though it's not too hard to come up with believable examples where multiple inheritance might make sense (`MotorHome`, `HouseBoat`, `EmtFirefighter`), I have yet to see a real application where that sort of inheritance was absolutely crucial. You can always implement classes such as these by using the facade design pattern. For example, the `EmtFirefighter` class would contain objects from the `Emt` (for "emergency medical technician") and `Firefighter` classes. It would then delegate calls to its properties and methods to those objects. Although this would be less automatic, it would be easier to understand and would also encourage a simpler, flatter object hierarchy.

Platform Dependencies

After you compile a Visual Basic application into IL, you use the CLR to execute the IL code. Microsoft provides support to use those libraries on Windows platforms, but, in theory, you could run on any platform that provides run-time support for IL.

The Mono project (`www.mono-project.com`) is an Open Source effort to support .NET applications in various operating systems. Currently, Mono lets you build and run .NET applications on Windows, Unix, Linux, Solaris, and Mac OS X. It also supports Java, Python, and other languages, in addition to C# and Visual Basic. See the Mono Web site `www.mono-project.com` for more information.

Between .NET's native support on Windows and Mono, .NET applications should run on the vast majority of PCs in use. An application written in Java or another Web-based language can run on those operating systems, and also on any other system that supports modern Web browsers, so such an application would have slightly greater portability.

Often, desktop applications built with Visual Basic provide a better user experience than those built with Java. Desktop applications have fewer restrictions than Java applications, and some things that are trivial in Visual Basic are much more difficult in Java, particularly if the Java application needs to run in a browser that doesn't support the same features available on the desktop.

However, there seems to be a growing feeling that "good enough is good enough" — that the restrictions imposed by Java are worth the ability to run by simply pointing a browser at the appropriate URL.

I prefer the performance and flexibility you usually get from desktop applications rather than Java, but there's much to recommend the "good enough is good enough" philosophy. When Visual Basic 3 was released, I had spent just over a month working on a C++ application. When I gave Visual Basic 3 a try, I was able to reproduce everything I had built within that month in only four days. There were some things that Visual Basic 3 could not do that were possible in C++, but they meant extra work and were things I was willing to live without for the sake of a 75 percent reduction in development time. I moved to Visual Basic because it was good enough, so I can understand the idea that Java is good enough for some applications.

Upgrading from Visual Basic 6

Visual Basic .NET includes an upgrade wizard that converts Visual Basic 6 code into Visual Basic .NET. Unfortunately, the wizard isn't very good. Microsoft claims the wizard can handle as much as 85 percent of Visual Basic 6 code without intervention, but in practice, the results aren't very good and a typical application requires a lot of manual rewriting.

However, this is still better than nothing. Even if the wizard converts only a small fraction of the code correctly, it's a head start that you wouldn't get if you wanted to port a Visual Basic 6 application to C++ or Java.

GUI Building

Believe it or not, the fact is that building graphical user interfaces (GUIs) in Visual Basic is so easy that it can actually cause problems. It's extremely easy for an inexperienced developer to slap together an awkward interface that's difficult for users to understand.

This is a problem with the developer rather than Visual Basic. The development environment gives you the tools you need to build a good or bad interface. It's up to the developer to learn how to use the tools wisely. The fact that it's more difficult to build GUIs in some other languages may force developers to spend more time on design, but that's time you should spend no matter what language you are using.

Verbosity

Many C++ and C# developers think Visual Basic is too verbose and that code is easier to read and write if it doesn't include as many characters. Though Visual Basic does require more typing, it is also more self-documenting. If the goal were to write programs as tersely as possible, all developers would use assembly code. Only a tiny part of application development is spent actually typing code. Far more time is spent reading code and looking for bugs.

In any case, Visual Studio's excellent IntelliSense makes it unnecessary for programmers to type out most class, variable, property, and method names in their entirety. Usually, you only need to type a few characters. If Visual Studio can figure out what choices might match (as when you're accessing an object's properties), it provides a list. If you're starting a new line, you can press Ctrl+spacebar to get a list of possible matches. Then you can quickly select one without additional typing.

I teach introductory Visual Basic courses where I always stress IntelliSense and, after a few weeks with the students, Ctrl+spacebar. It's particularly effective if you name controls and variables consistently. For a text box, type ***txt*** *and press Ctrl+spacebar to get a short list of the available choices. After a few weeks, the students become experts at using Ctrl+spacebar.*

Power and Flexibility

Probably the silliest disadvantage to Visual Basic (but perhaps the one that has the most impact on developers) is the perception that Visual Basic is less powerful than Java, C++, and C#. Although those languages are more difficult to learn, they are not significantly more powerful. They all run on the same hardware and can all do more or less whatever the hardware supports. In fact, C# and Visual Basic are practically identical, aside from their syntactic differences.

The fact that many pre-.NET Visual Basic developers were self-taught and that many came to Visual Basic via Visual Basic for Applications (VBA) makes Visual Basic seem more ordinary to many managers, and lends an extra mystique to Java, C++, and C#.

Salaries for Java, C++, and C# programmers also tend to be higher than those of Visual Basic developers. This is probably partly because of the perception that those languages are more powerful, the fact that they are more difficult, and the fact that there are far more Visual Basic developers available in the job marketplace.

All of these facts sometimes lead management to prefer those languages to Visual Basic, although there really isn't much justification on the basis of power or flexibility.

Advantages

Visual Basic does have some legitimate disadvantages, but it has many advantages as well. The following sections describe some of the most important advantages Visual Basic has over other common programming languages.

Self-Documenting

Visual Basic's syntax makes it much more self-documenting than other languages. Its keywords are spelled out and easy to read. Statements that close blocks such as `End Do` and `Next employee` explicitly tell you what block is ending — in this case a `Do` loop and a loop involving a variable named `employee`. This makes the code much easier to read than code that ends blocks with braces.

The `Next` statement that ends a `For` loop allows you to explicitly give the variable controlling the loop, specifying exactly which loop is ending. Although you are not required to include this variable, you should because it makes the code more self-documenting. The following code shows a small example:

```
For i As Integer = 0 To 100
    ...
Next i
```

Unfortunately, you cannot similarly attach the name of the variable controlling an `If Then` block, `Do` loop, or other structure. That makes some sense, because these statements are often controlled by Boolean expressions rather than simple variables. You can, however, include the controlling expressions in a comment after the closing statement. The following code shows how you might add comments to a `Do` loop and an `If Then Else` block:

```
Do Until all_done
    ...
Loop ' Until all_done

If num_employees <= 100 Then
    ...
ElseIf num_employees <= 500 Then ' If num_employees <= 100 Then...
    ...
Else ' If num_employees <= 100 Then ... ElseIf <= 500 ...
    ...
End If ' If num_employees <= 100 Then ... ElseIf <= 500 ... Else ...
```

As is shown in the last two comments, you don't need to include every detail of the block; you just need to include enough so that someone remembers the block to which the statement belongs.

The .NET Framework also uses nice, long, descriptive class, property, and method names, making it self-documenting as well. That makes using the Framework easier, and provides a good example to encourage Visual Basic developers to use similarly descriptive names.

Prototyping and Simple Applications

It is extremely easy to build user interface prototypes with Visual Basic. For many applications, you can put together a simple application in a few hours and get feedback from the customer before starting any really difficult development. Providing a prototype quickly is far more effective than making users visualize what an application will look like through drawings or textual descriptions.

Visual Basic is also extremely effective for building very simple applications. In a matter of minutes, you can build a basic database application that lets you add, query, update, and delete records in a database. Though you would need to spend some extra effort to make such an application bulletproof, adding validations, and so forth, you can quickly build prototypes or throwaway programs for your own use.

IDE

Visual Studio is an excellent integrated development environment (IDE) for programming and debugging code. IntelliSense makes it easier to correctly enter complex parameter lists. Breakpoints and watches make stepping through code easy. The call stack browser lets you jump through the code to see how pieces call each other.

Other languages provide their own IDEs and, though some such as Delphi's are quite good, Visual Studio is an outstanding IDE.

Language Relatives

Visual Basic is a close relative to several other languages, including the following:

- *VBA* (*Visual Basic for Applications*) — Used as a macro language by the Microsoft Office products and can be used for scripting in other applications.
- *VBScript* — Used in ASP applications to build interactive Web applications. VBScript is very similar to the Visual Basic language. In contrast, JavaScript is more like VBScript than Java.
- *Visual Basic "Classic"* — Although Visual Basic .NET is different in many ways from Visual Basic 6 and earlier versions, the languages have much in common. As the "Y2K problem" showed, software often lives far beyond its expected lifetime. While Visual Basic .NET is Microsoft's path of the future, legacy Visual Basic "Classic" applications will continue to need support and even development for many years to come.

This makes translating code between these languages relatively easy, so applications that you write in one may be useful in others. By using Visual Basic, you acquire skills and a code base that can be helpful in building desktop, Web, and scripted applications.

While the syntax details of Visual Basic and C# differ, the languages have a common enough structure that it's not too hard for a Visual Basic developer to read C# code. Though translating C# code line-by-line into Visual Basic is not as straightforward as translating VBA code, it is relatively easy to get the general idea of C# code and then rewrite it. That gives Visual Basic developers another source for ideas and solutions to common problems.

Garbage Collection

Although the garbage-collection scheme used by Visual Basic .NET has its drawbacks, it also has some advantages. Memory allocation is often the most difficult part of C and C++ applications to debug and maintain. If you forget to free allocated memory, the application has "memory leaks" and uses more and more of the system's memory over time. If you free the same memory twice, the memory system can crash. If you forget that you have freed a piece of memory and then later try to use it, the program may crash.

The nature of these bugs often means problems are obvious only long after the program executes the incorrect code. For example, suppose a program frees the memory holding an object. It may be much later that the program tries to access that object, causing a crash. Because the effects of these errors may occur long after their causes, these bugs can be extremely difficult to locate.

Visual Basic's garbage collection scheme avoids all of these problems by removing your memory management responsibility. When the garbage collector runs, it frees memory that cannot be accessed by the application. If a piece of memory such as an object reference is still usable by the program, the garbage collector leaves it alone. This prevents all of these memory-related bugs.

Large Talent Pool

It's hard to find reliable data about the number of developers currently using Visual Basic. At one time, Microsoft claimed there were more than 3.2 million Visual Basic 6 users (`msdn.microsoft.com/isv/technology/vba/overview/default.aspx`). They have also claimed more than 3.5 million Visual Studio users (`msdn.microsoft.com/vstudio/extend/vsta/default.aspx`), although they don't specifically mention which version of Visual Studio, and 1 million Visual Basic .NET users (`msdn.microsoft.com/netframework/technologyinfo/Overview/default.aspx`).

Whatever the numbers, it's clear that there are a *lot* of developers who use Visual Basic of one form or another. This in itself doesn't mean Visual Basic is the best language, but it does mean that there is a large pool of Visual Basic talent available. It means there are active newsgroups and forums where you can ask questions if you need help, and that there are plenty of developers that you can hire if you need extra hands on a project.

Summary

Although Visual Basic is an excellent programming language, the choice of language is not always trivial. Whether Visual Basic is right for you depends on your application.

Arguments that Visual Basic is less powerful or slower than languages such as C# are just plain wrong. All high-level languages have about the same capabilities and any missing features are usually easy to supply with a few well-chosen libraries. Still, Visual Basic does have a few legitimate drawbacks.

If you only need your application to run on Windows operating systems, then Visual Basic is fine. If you need to run on other systems, you will need support for running IL code. Mono may be an acceptable solution for Unix, Linux, Solaris, and Mac OS X, and there have been rumors that Microsoft is bringing support for C# to the Mac OS, but support will probably be better on Windows platforms.

Visual Basic does not provide multiple inheritance, so if your application architecture requires multiple inheritance, you may want to use another language. I have never seen a real-world application that truly *required* multiple inheritance, so if you only use it in a few places, you may be able to redesign the application slightly to avoid it. If all else fails, you can mimic multiple inheritance in limited cases by using a facade design pattern.

Though Visual Basic .NET is as fast as C# and is much faster than earlier versions of Visual Basic, it is still an interpreted language, and a truly compiled language may give better performance for extremely CPU-intensive applications. Most applications are limited by factors other than CPU use, however. Many applications spend most of their time waiting for user input, database requests, network access, or file access, and making the code itself faster may not help much. If you are writing a high-performance real-time application where every clock cycle counts, you may need to move to a truly compiled language. However, for most applications, you can move selected CPU-intensive pieces of code into compiled libraries and call them from Visual Basic when necessary.

Another consequence of the IL is that a Visual Basic .NET application requires a 280MB run-time library. Even in today's environment of cheap hard drives, that's a lot of disk space. More important, however, is that downloading such a large library takes a long time. That will prevent many users with only casual interest from downloading your application from the Internet, particularly users with slower network connections. Once customers have installed one .NET application, the run-time library will already be installed so they won't need to download it again. However, the first installation can be painful.

Visual Basic's interpreted nature also lets developers read IL code relatively easily. That makes it much simpler to reverse-engineer Visual Basic or C# applications to either steal useful code or exploit weaknesses in the code. Obfusticators can make code much more difficult to read, but not as hard as truly compiled code.

Finally, many C, C++, and C# developers complain about Visual Basic's verbosity. You can reasonably argue that this is a matter of personal preference, but I claim it's something more. Extra verbosity makes the language more readable, and that's absolutely crucial for debugging and maintaining any complex application. You can add extra comments to C++ or C# code to make it more maintainable, but Visual Basic has extra help built in. That leads to the most important concept in this book:

> *Programs are written for people, not computers.*

If you were writing only for the computer, you would use 0s and 1s, or at least some form of assembly language rather than a high-level language. The computer doesn't care what language you use. It's all 0s and 1s when the CPU executes it. The reason you use a high-level language is to make programming, understanding, debugging, and maintaining the code easier *for people*. Any feature that makes the code easier for a human to read and understand gives you an advantage that you should not lightly throw away in the name of conciseness.

Taking all of these factors into account, you need to decide whether Visual Basic is the right language for your application. Performance and capability are critical issues far less often than the ability of programmers to write, debug, and maintain the code, so Visual Basic makes an excellent choice.

Once you've picked a programming language, whether it is Visual Basic or something else, you're still not ready to start writing code. There are plenty of other tasks that you should perform before starting to build any non-trivial application. Chapter 2, "Lifecycle Methodologies," discusses different patterns for bringing a project through to produce a finished application. The chapters after that one describe other pre-coding design and development activities that you should study before you get down to the serious business of programming. If you stint on these phases, you may start coding sooner, but you may later pay a huge price when you need to debug and maintain the application.