

Sharpen Up on C#

Burton Harvey, M.S., MCSD

Introduction

This is an introduction to C#, a new programming language that combines the productivity of Visual Basic, the elegance of Java, and the power of C++.

Because C# is a part of Microsoft's .NET, we'll begin with a look at what .NET is, what it can do, and how it works. Next, we'll compare C# to other .NET languages. After exploring C#'s datatypes and idioms, we'll examine its facilities for creating objects and user interfaces. We'll conclude with a glance into C#'s future.

Burton Harvey is a software development consultant for Oakwood Systems Group, a Microsoft Partner company specializing in Internet solutions. A MCSD with fifteen years' experience using Microsoft development tools, Burt enjoys teaching others how to program and architecting software that elegantly fulfills clients' needs. In 1998, Burt founded the online journal of scientific research, *Scientia*. I added italics here for this publication. Is that correct? His areas of interest include compiler theory, UML, and the object-orientated paradigm. Contact Burton by emailing kbharvey@mindspring.com

C# and .NET

What .NET Is

.NET is Microsoft's new platform for software development. With .NET, a developer can use his programming language-of-choice to quickly and easily implement distributed applications that run on any platform. .NET supplies a rich class hierarchy of re-usable functionality, a drag-and-drop IDE that can host many popular languages, and a set of runtime interpreters.

How .NET Works

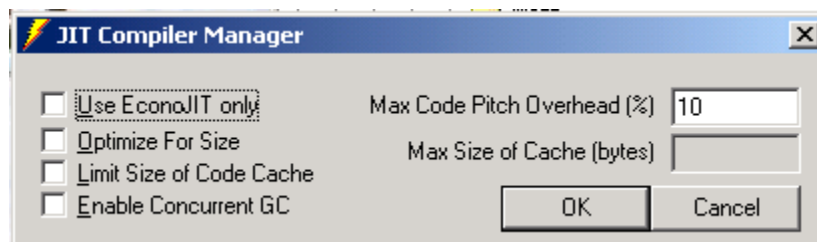
Intermediate Language (IL) is the key to how .NET works. Microsoft publishes the specifications for its IDE and IL, and vendors use these specifications to write compilers that plug into the IDE and generate IL. Because IL is the same regardless of which high-level language created it, developers using different high-level languages can share their compiled code. This is how the .NET class library, written in C#, can be referenced in Visual Basic (VB), C#, and Visual C++.

As its name suggests, IL code is half way between a high-level source code and low-level machine code. A "bytecode", IL consists of a series of eight-bit instructions for performing basic operations and shuttling fixed-length values between memory and a virtual stack. Some IL instructions mirror those in the instruction set of any RISC microprocessor, and others are special instructions designed for manipulating objects; you can think of IL bytecode as "assembly language +." Faster than script but not coupled to any one operating system or architecture, the same snippet of IL will execute on any platform equipped with an IL interpreter and a .NET runtime.

IL interpreters are called Just-In-Time (JIT) compilers. .NET ships with three different JITs, each of which targets a different execution environment.

- The **default JIT** compiles IL to highly optimized machine code, method-by-method, on an "as-needed" basis. The machine code is cached for re-use. The performance of programs using the default JIT will improve with use. This JIT targets execution environments with plenty of memory, and situations in which slower initial executions are acceptable.
- Like the default JIT, the **EconoJIT** also compiles on a method-by-method, as needed basis. The EconoJIT, however, compiles more quickly, producing less efficient machine code. The machine code is not cached. The EconoJIT therefore targets execution environments with less RAM and a greater need for consistent, albeit less-than-optimal, performance.
- The third JIT, **PreJIT**, compiles all IL to machine code before execution begins. This is the compiler to choose when execution speed is of primary importance.

For situations in which a finer degree of runtime control is desired, .NET supplies `JITMAN.EXE`. This program allows the user to turn EconoJIT on and off, and to dictate the amount of memory reserved for the caching of native code.



The .NET Runtime

In addition to compiling IL code, .NET uses a runtime to "manage" it. The management services include array bounds-checking, security against malicious code, and automatic garbage collection. This last service is a technique for thwarting memory leaks. Understanding how automatic garbage collection works can provide developers with an insight on how to use C# more effectively.

The garbage collection algorithm maintains a table of the number of references to each instantiated object. As references to an object are added or removed, the object's reference count is incremented or decremented in this table. In COM, where each object maintained its own reference count internally, objects could de-allocate themselves (*this->destroy;*) as soon as they were no longer needed. In .NET, the reference table is scanned at intervals, and un-referenced objects are de-allocated *en masse*, from without. Because there may be some time between the zeroing of an object's reference count and the next table scan ("sweep"), .NET developers should use objects' destructor functions to release resources explicitly.

We'll conclude our discussion of the .NET runtime by dispelling a common myth. The terms *unsafe code* and *unmanaged code* are not synonymous. "Unmanaged" describes native code that does not execute in the context of the runtime, such as might be generated in VB6. "Unsafe" code is code prefaced by C#'s `unsafe` keyword. Unsafe code sidesteps C#'s type-checking and thus can perform all sorts of "dangerous" pointer trickery. Because unsafe code can potentially wreak havoc on the system executing it, the runtime requires that it be granted a trust before it is executed. Although tagged as "unsafe," *such code is still managed by the runtime.*

Assemblies

An assembly is a re-usable packet of IL – an IL component. Most assembly files end with the ".dll" extension. Unlike COM DLLs, however, assemblies are not listed in the system registry. Instead, the information that assembly components need to provide clients is embedded within the components themselves: hence the term "assembly." The format of this metadata is an XML document called the *manifest*.

Clients are capable of using classes from any assembly sharing their working folder. Assemblies used by more than one application can be stored in the "assemblies" folder under the Windows folder and shared by all. "mscorlib.dll," an assembly stored in the `ComPlus` folder on the system drive, contains many of the .NET base classes. Because assemblies contain platform-independent IL and require no global registration, they are sometimes call *portable executables*.

.NET provides facilities for interfacing assemblies with legacy COM components. `TYPLIB.EXE` wraps COM components in an IL interface that makes them accessible from assembly clients. Conversely, `REGASM.EXE` wraps assemblies in a COM interface that makes them accessible from COM clients. Another program, `AXIMP.EXE`, wraps Active-X controls so that they can be used in .NET WinForm programs.

The IL Disassembler Program (`ILDASM.EXE`), mentioned earlier in this chapter, provides a Windows Explorer-style view of all the classes, interfaces, methods, fields, events, and properties defined in an assembly. Its sister program, `ILASM.EXE`, is provided as a debugging tool for language vendors targeting the Visual Studio IDE. It provides a quick-and-dirty method of packaging IL code and its metadata into an assembly:

```
Class1::Main : int32(class System.String[])
.method public hidebysig static int32 Main(class System.String[] args) il managed
{
    .entrypoint
    // Code size      16 (0x10)
    .maxstack 1
    .locals ([0] int32 V_0)
    IL_0000: ldstr      "Hello World!"
    IL_0005: call       void ['mscorlib']System.Console::WriteLine(class System
    IL_000a: ldc.i4.0
    IL_000b: stloc.0
    IL_000c: br.s      IL_000e
    IL_000e: ldloc.0
    IL_000f: ret
} // end of method 'Class1::Main'
```

The Efficiency of .NET

There is a tendency among veteran programmers to sneer at interpreted languages. While it is true that interpreted programs tend to execute more slowly than their machine code counterparts, today's high speed processors and gargantuan RAM stores blunt the discrepancies, and as processors become even faster, the validity of this criticism continues to diminish. In most real world situations, moderately fast code that ships on time is preferable to lightning fast code that ships 18 months late (or not at all).

How C# Relates to .NET

C# is the premier language for .NET development. When you use C#, you are using the language in which the Next Generation Windows Services base classes were coded before they were compiled to IL.

When you use C#, you're using a modern language that includes the best features of the old languages and eliminates their worst ones.

When you use C#, you're not using a language that was adapted to the .NET platform, but one designed for it.

Conclusion

At this point, you should understand .NET well enough to start learning C#. Let's begin by comparing C# to other, more familiar languages.

C# and Other .NET Languages

Visual Basic

Visual Basic makes it easy to program user interfaces. To do so, you follow three steps:

- Drag controls from the toolbox onto a form.
- Set the values of the controls' properties.
- Write functions to handle events raised by the controls.

C# takes the same approach to interface design, both for traditional Windows applications that must be installed on the client, and for web applications that run in the client's web browser. Furthermore, C# extends the drag-and-drop paradigm to new tasks, such as the construction of data access objects.

Despite sharing this approach to interface design, VB and C# have many grammatical differences. The following list, although not conclusive, might help VB veterans with C# aspirations over a few hurdles.

- C# delimits lines with the semicolon (";") rather than a carriage return/line feed.
- C# has distinct assignment ("=") and equality ("==") operators.
- C# has short-circuiting conditionals; VB does not. In other words, in C#, expression B in the compound expression "(A && B)" will not be evaluated if expression A evaluates to `False`.
- You can't use a string variable in a C# `switch...case` block the way that you can use a string in a VB `select...case` block.

VB veterans may be a bit perplexed when they first encounter VB7 code; it is quite different to VB6 code. In fact, you could argue that VB7 code resembles C# code more closely than it resembles VB6 code.

I prefer C# to VB because I think it is more elegant and consistently object-oriented, but one of the great things about .NET is that it lets you develop with whichever language(s) you want.

C++

Visual C++ programmers will want to adopt C# for enterprise development because it is simpler and safer than C++. C# discourages such bug-prone idioms as pointers, fall-through `switch...case` blocks, the equivalency of Boolean and integer types, preprocessor macros, and the segregation of class interfaces into separate header files. A smaller language, C# eliminates redundant idioms, retaining only the most obvious mechanism where formerly there were more to choose from, and potentially be confused by...

Some of C#'s other improvements include:

- Automatic garbage collection (goodbye, memory leaks!).
- Subsuming the functionality of the "->" operator within the "." operator.
- Providing a dedicated, more intuitive syntax for arrays.
- Eliminating the need for tons of boilerplate when developing reusable components.
- Restricting inheritance (multiple interface inheritance permitted, single implementation inheritance permitted).
- Performing stronger type checking.

You can think of C# as a subset of C++. You can still use C++ features like pointers inside C# code if you tuck them into code blocks prefaced with the `unsafe` keyword.

Java

Java isn't a .NET language, but I'll put the comparison here anyway.

As critics have pointed out, C# code looks a lot like Java code. However, there are at least three reasons to choose C# instead of Java.

First, C# retains more of C++'s power than Java does. For example, Java dispenses with type-safe enumerations; C# retains them. Java lacks operator overloading; C# provides simple facilities for using operator overloading effectively.

Second, one of the most significant new technologies, XML, is integrated into the very foundation of C#. Using XML with Java requires patching together disparate technologies into a "homegrown" solution.

Third, and most significantly, C# code can interact with code written in any other .NET language. You don't have to migrate your whole organization to C# in order to use it effectively.

Datatypes in C#

We've examined .NET, the languages in it, and how they relate to C#. Now it's time to delve into C# properly.

Datatypes are a good place to start investigating a language. C#'s datatypes are merely aliases for classes shared throughout .NET. These classes are organized into *namespaces*.

Namespaces

Just as assemblies are physical containers for IL classes, namespaces are logical ones. They allow the developer to use two classes with the same name in the same program. This capability is invaluable in modern projects involving multiple teams, parallel efforts, and third-party components.

A namespace consists of several words delineated by periods. Typically, organizations "brand" components developed in-house by using the organization's name as the first word in the components' namespace. Subsequent words indicate various areas of functionality, with specialty increasing from left to right. "Harvey.WroxDemo.Automata.UI" is one example.

Since typing long namespaces can be a chore, most IL-ready languages provide facilities for abbreviating them. The `using` statements appearing at the top of C# code modules do not provide information to the linker, but instead allow the developer to reference classes with relative, rather than absolute, names. Namespaces can be thought of as hierarchical trees *but a code module's reference of one namespace does not automatically include references to all of the child namespaces below it.*

Because .NET makes the information from classes available over the web, a logical namespacing scheme benefits not only the organization that enforces it, but all the other organizations to which that organization provides data. To encourage specificity on the Internet, Microsoft provides namespacing guidelines in the MSDN library.

The base classes provided with the .NET framework are organized into hierarchical namespaces. Let's take a look at those.

The System Namespace

System is the root of the .NET base class namespace. In it, a set of datatypes recognized by all .NET-compliant languages is defined. This set includes a dedicated Boolean type, a string type, a decimal type for financial computations, and a group of integer types ranging from 8 to 64 bits in length. One type, the object type, is the type from which user-defined classes are derived.

In the parlance of .NET, the term "object" only refers to classes derived from the object type. However, all the .NET types are "objects" in the academic sense that an object is data combined with the operations that may be performed on it. Instances of all types, even integers, have conversion and formatting methods. There are no "primitives."

Data Type	Description	Value Type	Reference Type
bool	Dedicated Boolean type (true or false)	X	
byte	8-bit unsigned integer value	X	
char	Unicode (2-byte) character	X	
decimal	Monetary value	X	
double	Floating-point value	X	
float	Floating-point value		
int	32-bit signed integer value	X	
long	64-bit signed integer value	X	
object	Reference to an object		X
sbyte	8-bit signed integer value	X	
short	16-bit signed integer value	X	
string	Unicode (2-byte) b-string	X	
struct	Struct	X	
uint	32-bit unsigned integer value	X	
ulong	64-bit signed integer value	X	
ushort	16-bit signed integer value	X	

There are, however, two distinct categories of types. *Value types* are those types for which an assignment results in a new copy of the assigned value. *Reference types* are those types for which an assignment involves not a copy, but a new reference to a value already existing in memory. The following demonstrates the difference between value and reference types.

```
public static int Main(string[] args)
{
    /*** VALUE TYPES***/
    int a, b; //Two value variables.
    a=10;     //The value "10" is copied into a's space.
    b=a;     //a's value("10") is copied into b's space.
    b=20;    //The value("20") is copied into b's space.
            //Because a and b are value types with their
            //own spaces, the value of a remains "10".

    /***OBJECT TYPES***/
    object c, d; //Two reference variables.
    c=10;        //The value "10" is copied into c's space.
    d=c;         //c is set to refer to d.
    d=20;        //The value("20") is into the space referred to
                //by both c and d.
                //Because c and d refer to the same space,
                //the value of c has become "20":
}
```

Reference types are stored on the heap. Value types are stored on the stack. Occasionally, it may be desirable to cast a value type to a reference type so that it persists after the function that created it ends. Such casting from a value type to a reference type is called *boxing*. Conversely, casting from a reference type to a value type is called *unboxing*. Only certain types can be boxed and unboxed.

C and C++ programmers should note that C#'s `struct` is a value type.

The datatypes in the `System` namespace are aliased with words that are more familiar to developers. The `System.Integer32` type, for instance, is aliased as `long`. VB7 and C# use the `System` datatypes as their core types. Projects created in with those languages automatically reference the `System` namespace. Creating .NET applications with Visual C++ requires linking in some header files.

In addition to .NET datatypes, the `System` namespace defines several other classes with useful, general functionality.

Other Useful Namespaces

`System` is not the only .NET namespace that provides useful functionality.

`System.Math` provides useful mathematical functions. They are class methods; you don't need to create an instance of an object in order to use them.

Method	Returns
<code>Abs</code>	A number's absolute value
<code>Round</code>	A number, rounded
<code>Floor</code>	The largest integer less than a number
<code>Ceil</code>	The smallest integer greater than a number
<code>Max</code>	The largest of a sequence of numbers
<code>Min</code>	The smallest of a sequence of numbers
<code>Sin</code>	The sine of a number (in radians)
<code>Cos</code>	The cosine of a number (in radians)
<code>Tan</code>	The tangent of a number (in radians)
<code>Asin</code>	The arc-sine of a number (in radians)
<code>Acos</code>	The arc-cosine of a number (in radians)
<code>Atan</code>	The arc-tangent of a number (in radians)
<code>Sqrt</code>	The square root of a number
<code>Pow</code>	The value of a number taken to a power
<code>Log</code>	The natural logarithm of a number
<code>Log10</code>	The base-10 logarithm of a number

`System.Random` creates a sequence of pseudo-random numbers. To use this class, you create a `Random` object, set its `seed` property, and collect numbers from repeated calls to the object's `Next()` method.

`System.Exception` defines several exception classes, all of which derive from the `Exception` base class. `ArgumentException`, `ArgumentOutOfRangeException`, and `ArgumentNullException` could be used by solid, self-checking code that examines the value of input arguments to its functions. `NullReferenceException` will probably be encountered often in the object-oriented world that is .NET. `SystemException` represents an internal error in the .NET runtime. `COMException` denotes a non-zero `HRESULT` returned from a call to a legacy component. The fact that `Exception` objects can be passed between IL code written in different

languages is one of the best features of .NET. Developers can derive their own exception classes from the ones included in the `System.Exception` namespace.

Code can interrogate an instance of the `OperatingSystem` class to dynamically determine the execution environment.

Microsoft permits the addition of new types to the `System` namespace, but only if certain conditions are met.

`System.Collection` provides a `Collection` class and a `Dictionary` class for storing instances of the object type. For developers who want to roll their own type-safe container classes, the `ICollection`, `IDictionary`, and `IEnumerator` interfaces are defined.

`System.IO` provides classes for easy, abstracted access to the filesystem.

`System.Reflection` provides interfaces for implementing events.

`System.Threading` defines `Timer`, `Mutex`, and `Thread` objects for multithreading functionality.

`System.Diagnostics` contains functionality for determining such runtime conditions as call stack depth. `Assert` and `Log` classes are aids to debugging.

Last but not least, the `System.Web.UI.WebControls` namespace defines the ASP+ *web controls*. The important role that they play in .NET is a topic addressed later in this presentation.

Basic C# Constructions

An exhaustive discussion C#'s grammar is beyond the scope of this presentation, but this section provides a sufficient overview to get experienced programmers up-and-coding.

Declarations

C# supports the kind of flexible variable declarations that C/C++ users expect and love.

The following lines are all legal:

```
int I;  
int j=10;  
int x=10,y,z=30;
```

Assignments

Assignments of values to variables are made, as would be expected, with the assignment operator ("=").

With an eye to safety, the C# compiler delivers warnings for variables whose values are referenced before they are initialized.

Loops

C# provides three different kinds of loops:

- `for...next` loops are post-test loops useful for looping a pre-determined number of times.
- `do...while` loops are post-test loops useful for looping an indeterminate number of times (until a condition is met).
- `while` loops are the pre-test versions of `do...while` loops.

Variables declared within loops are visible only within the scope of the loop in which they are declared.

Conditionals

For branching, C# provides the `if...else` construction. As in C++, only the first statement following a triggered `if` condition is executed, unless that statement is a compound statement, in which case the entire code block executes. `if...else` statements can be sequenced one after the other to express complex conditional logic. C# supports the ternary operator ("`A?B:C;`") for expressing `if...else` constructions in a shorthand form.

C# supports `switch...case` statements, but does not allow execution to "fall-through" from one case clause to the next. In cases where such behavior is desirable, labels can be used to achieve an approximation.

C/C++ veterans should take note that C# does not permit implicit conversions between the `int` and `bool` types.

Comments

C# supports both C-style comments ("`/*comment here*/`") and C++ style comments ("`//comment here`").

The IDE can automatically generate XML documentation from comments in a component's source code. C# reserves a special comment sequence ("`///`") to indicate comments that should be included in the generated XML.

Object-Oriented Case Study: "Life"

The code samples in this presentation are excerpts from a C# implementation of John Conway's "Life" game. I chose Life because it is fun and familiar, and because an object-oriented implementation of it allows a thorough demonstration of C#'s features, including: inheritance, abstract and sealed classes, abstract and virtual methods, indexers, Win32 API invocation, enumerations, exceptions, and events. Because the program was written with a pre-Beta, "Tech Preview" release of C#, it has a lot of room for improvement, but I hope that it will serve readers as an introduction now and as a reference later.

Life

As you may know, Life is a *cellular automaton*, a matrix of cells that "evolves" from one generation to the next. By applying simple rules that determine each cell's birth, death, or survival, a cellular automaton can develop from ragged chaos into entrancing (and strangely beautiful) designs.

The rules of Life are as follows:

Live cells with less than two neighbors die of loneliness.

Live cells with more than four neighbors die of overcrowding.

Dead cells with exactly three neighbors come to life.

Despite its popularity, Life is not the only cellular automaton. Other automata apply different survival rules and exhibit different behavior. In this presentation, I leverage the object-oriented techniques of abstraction, encapsulation, polymorphism, and data hiding to build a program flexible enough to display any kind of cellular automaton.

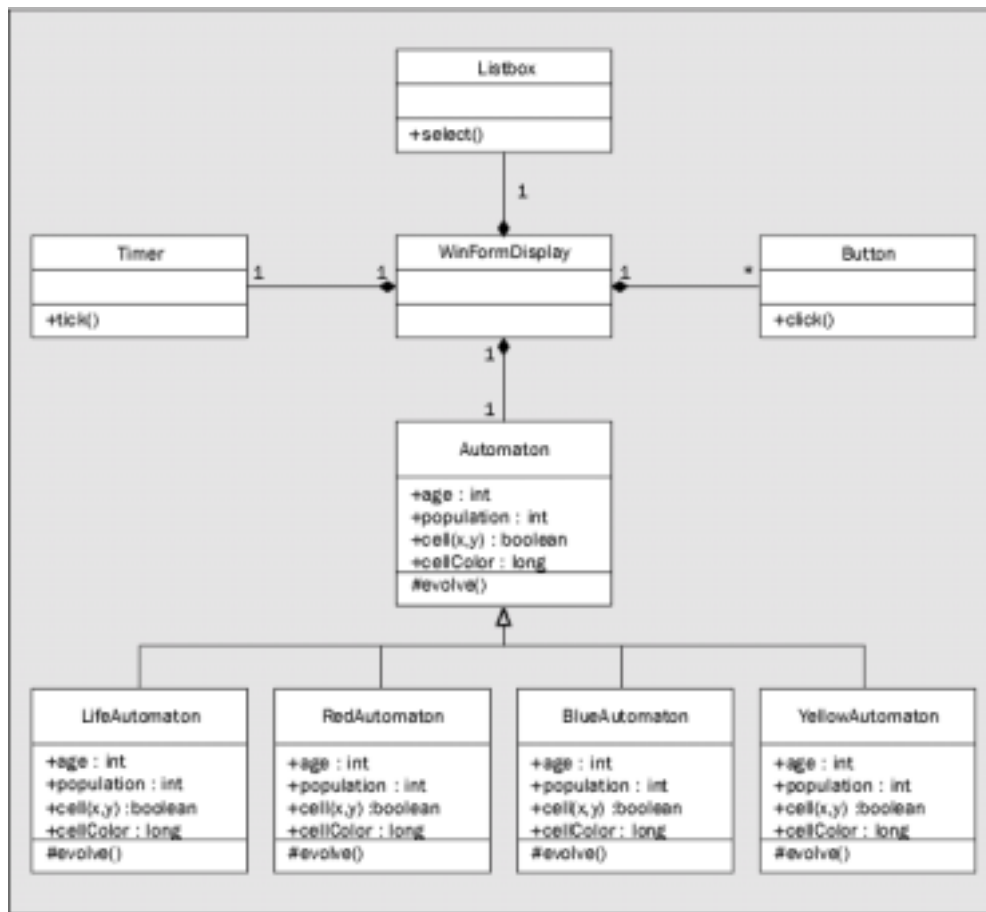
Now that we understand C#'s basic types and idioms, let's use our Life project to explore C#'s rich object-oriented facilities. While doing so, we'll use Unified Modeling Language (UML) to document our design ideas for the project.

Requirements

We want a program that can monitor any type of cellular automaton, regardless of the automaton's particular evolutionary rules. We should be able to add a new automaton to the program simply by adding a new class; the user interface shouldn't have to change very much. Because most automata will implement certain tasks (the creation of a random starting configuration, for instance) in the same way, we ought to be able to share common functionality between automaton classes. This re-use shouldn't be mandatory, however; an unusual automaton should be able to implement these tasks in unusual ways.

Designing the Classes for Life

An architecture for a cellular automata program meeting our requirements is depicted in the following diagram.



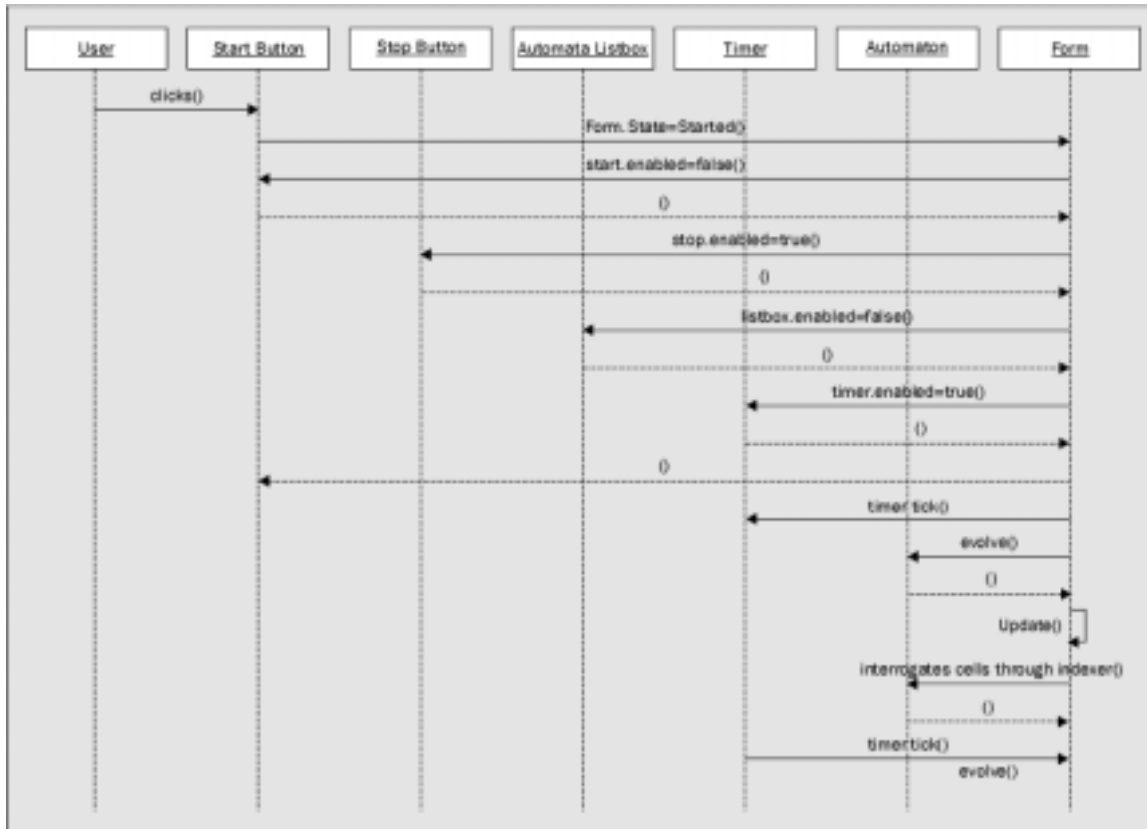
The core of this architecture is the Automaton base class. Its features – `age()`, `population()`, `cell[x,y]`, `cellColor()`, and `evolve()` – are the intersection of all features that any automaton may conceivably have. After instantiating an object with an Automaton interface, a client will be able to interrogate the object's `age()` property, `cellColor()` property or `population()` property. The `cell[x,y]` property allows the client to get or set the value of any cell in the automaton. The `evolve()` method allows the client to effectively age the automaton by one generation while remaining oblivious to the actual rules governing how this is done.

`LifeAutomaton`, `RedAutomaton`, `BlueAutomaton`, and `YellowAutomaton` all inherit their interfaces from the Automaton base class. The display class, `WinFormDisplay`, maintains an association with an object supporting the Automaton interface. Because all of the child Automaton classes support the Automaton interface, they can be "switched out" with each other dynamically at runtime, leaving the `WinFormDisplay` class blissfully ignorant of the substitution, calmly calling the `evolve()` property on its object reference to move the automaton forward, and querying that reference's `cell[x,y]` property from within a nested loop that draws the cells on the screen.

I toyed with the idea of giving each Automaton class the responsibility of being able to draw itself. In such a scheme, the client might pass an Automaton object a reference to a device context through a `Draw()` method, and wait while the Automaton object drew a line, circle, or some other geometric shape for each of its cells. I dispensed with this idea because, although initially attractive, it couples the Automata classes to the convention of a user interface, and I could foresee remote possibilities in which these classes might be used without one. Although it sounds far-fetched, it might be interesting to instantiate these objects on a web server, apply a genetic algorithm to them to discover an initial configuration that will evolve for a very long

time without dying out, and supply those configurations to other Life hobbyists across the Web using XML.

I decided that an instance of the `Timer` control in the `System.WinForms` namespace could serve as the "timing belt" of the program. Every time the timer ticked, it would raise an event to the `WinFormDisplay` object. In its `Timer_Tick` event handler, the `WinFormDisplay` could tell its `Automaton` object to `evolve()`, and would then redraw itself - see the following diagram:



Implementing the Abstract Automaton Class

When implementing the `Automaton` class in C#, I wanted to make sure that clients would be unable to create actual `Automaton` objects. The purpose of `Automaton` is to provide a user interface and default functionality that creatable child classes can inherit. It wouldn't make much sense for a client to create a generic `Automaton` object. It would be like asking someone, "What kind of car do you have?" and the person replying, "Car."

C# offers three conventions for creating what are known as *abstract classes*.

First, you can use an old C++ trick and provide a default constructor, modified with the `protected` access modifier. This prevents clients from being able to invoke the object's constructor because they can't see it.

Second, you can use the `interface` keyword to define an interface rather than a class. An interface is just a list of function signatures; the functions themselves cannot be implemented.

Third, you can put the `abstract` keyword before the class definition.

I chose to use the third method to make the `Automaton` class abstract. The first approach is little more than a hack. The second approach would not allow me to define some default functionality that inheriting classes could use out-of-the-box.

Incidentally, just as C# provides a keyword for indicating that a class can only be inherited from and not instantiated, it provides another keyword for indicating that a class can only be instantiated and not inherited from! This keyword is `sealed`. Classes prefaced with the `sealed` keyword cannot serve as the parent for any other class. In tribute to John H. Conway, the inventor of Life, I protected the `LifeAutomaton` class from further specialization by making it `sealed`.

Overriding Functions and Properties

I wanted to implement the `Automaton` class in such a way that child classes inheriting its default functionality would be forced to do so correctly. Specifically, I wanted child classes to be forced to provide functionality where they ought to, to be able to override functionality where they might want to, and to be prevented from overriding functionality when doing so would be unsafe or would comprise the conceptual integrity of the architecture.

Child classes should be forced to implement an `evolve()` method that encapsulates their idiosyncratic evolutionary rules. To enforce this constraint, I prefixed the C# keyword `abstract` to the `evolve()` function signature in the `Automaton` base class. In this context, the `abstract` keyword allows the parent class to forego a body for the function. This absence forces inheriting classes to provide their own method matching this signature. Child classes providing functionality for a parent class' abstract method must prefix the function they supply with `overrides`.

I wanted child classes to have the option of whether or not to implement the `cellColor()` function. They could override the function to make it return a color unique to them, or they could forego that override and use the parent's functionality for returning the color white. To accomplish this, I put the keyword `virtual` before the `cellColor()` function definition in the `Automaton` base class. Again, child classes opting to re-implement this default functionality would have to supply the `override` keyword with their definition.

Since several of the `Automaton` properties, including `population()` and `age()`, were likely to be implemented the same way in every child class, I chose to bar child classes from doing so. Because the correspondence between the properties and simple data members within the class interior was one-to-one, I decided that no flexibility would be destroyed, and that I could prevent tedious bugs of the kind that might result from mistyping. No special keywords were needed to enforce this constraint. Because the property definitions were not prefaced with the `virtual` or `abstract`, child classes have to take the functionality as is.

Controlling Property Access

To further ensure the integrity of the child classes, I made the `population()` and `age()` properties read-only. I accomplished this by providing only a `get{}` clause and leaving out the `set{}` clause from those property definitions.

Exposing Arrays via Indexers

I knew that clients of `Automaton` classes would tend to think of those server objects as kinds of magical matrices, two-dimensional arrays that could be told to change state and then interrogated. Instead of using a hokey "get" method that added unnecessary complexity to the client's view of the `Automaton` family of classes, I decided to use an *indexer* as the means by which clients could get and set values of the cells in the `Automata` objects.

A C# indexer allows the client to reference the class containing the array as if it were referencing the array itself. In the case at hand, the `WinFormDisplay` class could call `"m_objAutomaton[0,0]=true;"` to set the value of `cell(0,0)` in the array maintained hidden inside of `m_objAutomaton`.

You might wonder if it would make more sense to simply expose the internal array of the `Automaton` class as a public data member. Doing so would result in simpler, but much less safe code. One of the benefits of using indexers is that they allow you to validate the input arguments to the indexer before you return the value that they reference.

Events and the Win32 API

Automata objects have a lot of intelligence inside them. They know how to update themselves, how old they are, how big their populations are, etc. I thought that it would be good if there were some way for Automata objects to alert clients when special events – such as a catastrophic population growth spurt – occurred. To implement this ability, I used a C# event.

Events allow a client to asynchronously notify none, one, or many clients that some condition (usually a failure) has transpired. Clients add pointers to functions inside themselves to a list maintained inside the server. When the server needs to "raise" an error, it iterates through the list of function pointers, calling the functions, giving the recipient classes opportunity to "handle" the event.

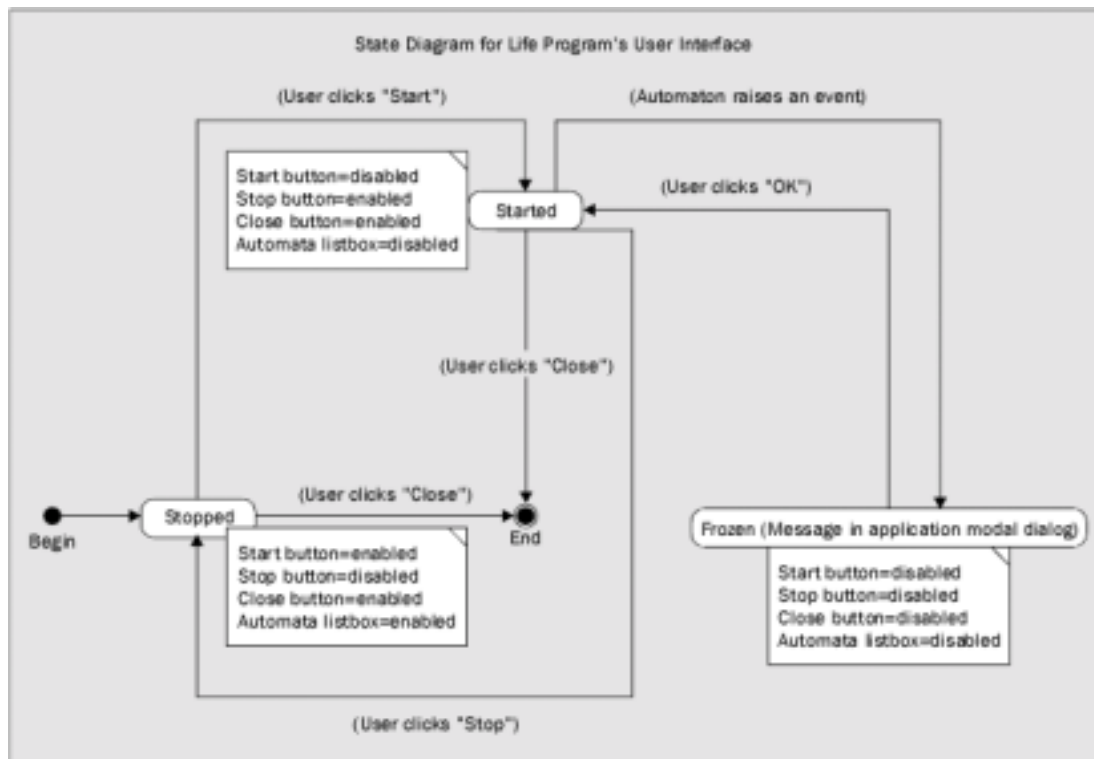
These are the steps to implementing an event in C#:

- In the module containing the server class, forward-declare a `delegate` function with the `extern` keyword.
- In the server class itself, declare an instance of the `delegate` function modified with the `event` keyword. We'll call this the "connection point."
- Also in the server class, put in the code that "raises" the event by calling the connection point at appropriate times.
- In the client class, define event handler functions that match the signature of the `delegate` function declared in the server module.
- In the constructor function of the client class, cast references to your handler functions to the connection point type defined in the server module, and add those references to the server object's connection point.

In my event handler in the `WinFormDisplay` class, I used the `MessageBoxA` function from the Win32 API. To use this function, I simply provided a C# wrapper function definition modified by a `DllImport` attribute with a value of `"user32.dll"`. When I needed the function, I called it as if it were any other C# function.

Designing and Implementing the User Interface

Although the user interface would consist of only three controls – a start/stop button, a close button, and a listbox for selecting an automaton – the states of these controls would have to be managed intelligently in order for the user interface as a whole to stay in a consistent state. For instance, when the start button is clicked to begin the evolutionary process of the automaton, it should change to a stop button, and the listbox for selecting an automaton should become disabled. Conversely, when the stop button is clicked, it should turn back into a start button and the listbox for selecting automata should become enabled. Finally, an event raised by the Automaton object should freeze all the controls until the user recognized it by clicking "OK" on the event's message box. To make sure I got the state right, I drew a state diagram for the user interface.



This diagram forced me to designate definite states that the user interface could be in: "Started," "Stopped," and "Frozen." Once I had determined these states, representing them in C# code was a breeze.

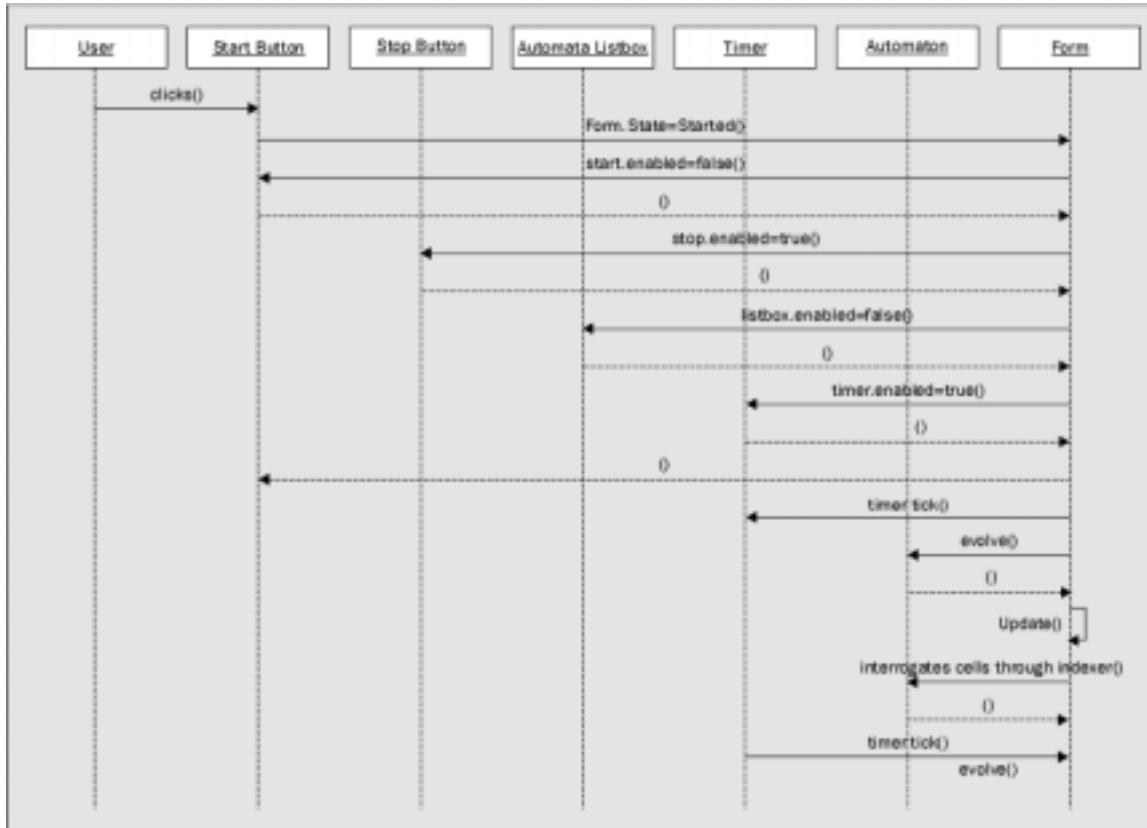
To do so, I used C#'s `enum` statement to define an enumeration called `FormStateEnum`, with one value corresponding to each of the determined states. I gave the `WinFormDisplay` class a private data member of type `FormStateEnum` for persisting this state, and then wrapped access to this variable in a property method. When the `WinFormDisplay` `FormState` property is set, the property method uses code in a `switch...case` statement to make sure that the buttons and listboxes correctly reflect the newly assigned state of the interface. See the following three diagrams.

The following code demonstrates how I ensured consistent state in the interface.

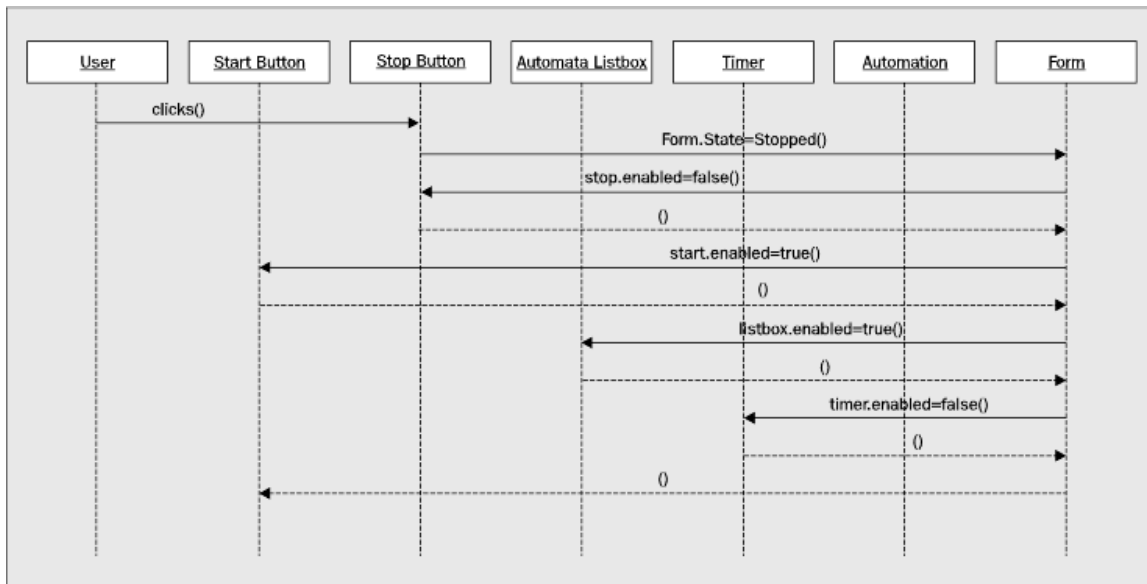
```
//Use of a enumed property to control and track a user interface's state
improves readability
//in the code and consistency in the subcontrol states.
private enum FormStateEnum { StoppedState=0, StartedState=1,FrozenState=2
}
private FormStateEnum m_enumFormState;
private FormStateEnum FormState{
    set {
        m_enumFormState=value;
        switch (m_enumFormState)
        {
        case FormStateEnum.StoppedState:
            timer1.Enabled=false;
            this.btnEvolve.Text="Start";
            this.btnClose.Enabled=true;
            this.btnEvolve.Enabled=true;
            this.lstAutomata.Enabled=true;
            break;
        case FormStateEnum.StartedState:
            timer1.Enabled=true;
            this.btnEvolve.Text="Stop";
            this.btnClose.Enabled=true;
            this.btnEvolve.Enabled=true;
            this.lstAutomata.Enabled=false;
            break;
        case FormStateEnum.FrozenState:
            timer1.Enabled=false;
            this.btnClose.Enabled=false;
            this.btnEvolve.Enabled=false;
            this.lstAutomata.Enabled=false;
            break;
        default:
            throw new Exception("Form is in an unanticipated state.");
        }
    }
    get {
        return m_enumFormState;
    }
}

//This changes the state of the form in response to a user's input
protected void btnEvolve_Click(object sender, System.EventArgs e)
{
    if (this.FormState==FormStateEnum.StartedState)
        this.FormState=FormStateEnum.StoppedState;
    else if (this.FormState==FormStateEnum.StoppedState)
        this.FormState=FormStateEnum.StartedState;
    else
        throw new Exception("Form is in unanticipated state.");
}
```

The next figure shows the changing interface state at "Start":



Finally, the third figure shows the changing interface state at "Stop":



Conclusion: The Future of C#

There are rumors of two new technical advances for C#.

First, generic programming will come to C#. Developers will be able to write sections of code that work with objects of more than one datatype. Although languages such as C++ currently implement generic programming with templates, C#'s generic programming mechanism will be different. Specifically, the facilities for generating programming will be pressed down into the JIT and the .NET runtime, so that genericized code can be re-used between different .NET languages.

Second, word is out that a new JIT, OptJIT, will allow C# developers to use attributes to make suggestions to the JIT compiler on how to target specific platforms.

No matter what technical embellishments are added to C#, it is destined to be around for a long time. Its elegant design represents decades of lessons learned from earlier programming language attempts.

Microsoft has stated that they will submit a proposal to an international standards committee for the ratification of a C# standard, allowing other vendors to implement C# flavors that target their platforms.

I hope this presentation has provided you with a good introduction to C#. At least the accompanying code might prove a useful reference to the syntax of certain C# idioms.

Further Resources

MSDN

<http://msdn.microsoft.com/vstudio/nextgen/technology/csharpintro.asp>

<http://msdn.microsoft.com/msdnmag/issues/0900/csharp/csharp.asp>

<http://msdn.microsoft.com/library/default.asp?URL=/library/welcome/dsmsdn/deep07202000.htm>

<http://msdn.microsoft.com/voices/deep07202000.asp>

Other

<http://www.csharptimes.com>

<http://www.csharpindex.com/rwc/>

<http://www.aspheute.com/artikel/20000713.htm> (German)

http://www.mcp.com/sams/detail_sams.cfm?item=0672320371

<http://commnet.pdc.mscomptech.com/slides/5-313.ppt>

http://java.oreilly.com/news/farley_0800.html

http://www.genamics.com/visualj++/csharp_comparative.htm

http://windows.oreilly.com/news/hejlsberg_0800.html