Wolfram *Mathematica*® Tutorial Collection

.*NET/LINK*™ USER GUIDE

For use with Wolfram *Mathematica*® 7.0 and later.

For the latest updates and corrections to this manual:
visit reference.wolfram.com

For information on additional copies of this documentation:
visit the Customer Service website at www.wolfram.com/services/customerservice
or email Customer Service at info@wolfram.com

Comments on this manual are welcomed at:
comments@wolfram.com

Content authored by:
Todd Gayley

Printed in the United States of America.

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

Contents

Introduction

.NET/Link

Welcome to *.NET/Link*, a product that integrates *Mathematica* and Microsoft's .NET platform. *.NET/Link* lets you call .NET from *Mathematica* in a completely transparent way, and allows you to use and control the *Mathematica* kernel from a .NET program. For *Mathematica* users, *.NET/Link* makes the entire .NET world an automatic extension to the *Mathematica* environment. For .NET programmers, *.NET/Link* turns *Mathematica* into a scripting shell that lets you experiment with, build, and test .NET classes a line at a time. It also makes .NET an ideal environment for writing programs that use the computational services of *Mathematica*.

.NET/Link's most unique feature is that it lets you load arbitrary .NET types into *Mathematica* and then create .NET objects, call methods, properties, and so on, directly from the *Mathematica* language. Thus, you can use *Mathematica* to "script" the functionality of an arbitrary .NET program—in effect, write a .NET program in *Mathematica*. Essentially anything you can do from .NET, you can now do from *Mathematica* perhaps even more easily because you are working in a true interpreted environment.

.NET/Link also lets you do some very useful things that do not appear to directly involve the .NET runtime. These include calling C-style DLL functions directly from *Mathematica*, and creating and scripting COM objects, much like Visual Basic can do.

.NET/Link is designed for end-users and developers alike. The same features that let *Mathematica* users transparently call any .NET method also let developers create sophisticated commercial add-ons to *Mathematica*. Programmers who want to write custom front ends for *Mathematica* or use *Mathematica* as a computational engine for another program will find using .NET with *.NET/Link* is easier than using the traditional *MathLink* interface from C or C++.

Finally, *.NET/Link* comes with full source code. You can examine the code to supplement the documentation, get tips for your own programs, better understand how to use advanced features, or just see how it works.

Some familiarity with both the .NET Framework and *Mathematica* is assumed in this manual. In Part 2, which covers writing .NET programs that call *Mathematica*, major examples are generally provided in both C# and Visual Basic .NET versions, although overall the documentation is perhaps slightly more C#-centric. Naturally, when writing .NET programs that use *.NET/Link*, you can use any .NET-aware language, not just C# and Visual Basic .NET.

"Calling .NET from *Mathematica*" shows how you use *.NET/Link* to call .NET from *Mathematica* and "Calling *Mathematica* from .NET" shows how to call *Mathematica* from .NET.

What Is .NET?

.NET is a new development platform for Windows programming. It replaces essentially everything that came before it, including an entire alphabet soup of programming technologies such as MFC, COM, ActiveX, ATL, ASP, ADO, and many others. Although Microsoft emphasizes XML Web Services in conjunction with .NET, XML Web Services are only a small part of the .NET platform, and the success of .NET is not dependent on the widespread adoption of XML Web Services.

.NET represents the future of Windows programming, and Microsoft is rapidly shifting more and more of its technology and products to a .NET foundation.

At the core of .NET is a runtime engine, similar to that used by Java, that loads and executes programs compiled into special bytecodes that the runtime understands. This runtime is called the Common Language Runtime (CLR), but we will often refer to it as the .NET runtime. A key feature of this system is that many languages can be compiled into CLR bytecodes and executed by the runtime. This means that .NET is language-neutral, supporting any programming language for which a .NET compiler is available. Microsoft provides compilers for C#, Visual Basic .NET, JScript, Visual J# .NET, and C++ With Managed Extensions. Many other compilers exist, including ones for Fortran, Perl, Python, Eiffel, COBOL. You can even create a class in one .NET language, say Visual Basic .NET, and subclass it in another language.

Although .NET is language-neutral, probably the two most important .NET languages are Visual Basic .NET, a modification of the Visual Basic language, and C#, a new language that is similar in many ways to Java.

What Is *MathLink*?

MathLink is Wolfram Research's protocol for sending data and commands back and forth between *Mathematica* and other programs. *MathLink* is the underlying glue that lets .NET and *Mathematica* talk to each other. When calling .NET from *Mathematica*, *.NET/Link* completely hides the low-level details of the *MathLink* communication, allowing *Mathematica* programmers to load and use .NET classes as if they were part of the *Mathematica* environment itself. When writing .NET programs that call *Mathematica*, *.NET/Link* provides a higher-level layer of functionality than the traditional C *MathLink* programming interface.

How Does *.NET/Link* Compare to *J/Link*?

J/Link is an existing Wolfram Research product that integrates Java and *Mathematica* in almost exactly the same way that *.NET/Link* integrates .NET and *Mathematica.* You can use *J/Link* to do many of the same things you can do with *.NET/Link* and vice versa. Because it is based on Java, *J/Link* has the advantage of being cross-platform. If you want to write programs that run on every *Mathematica* platform, you should use *J/Link*. On the other hand, .NET integrates more tightly with the Windows operating system than Java does, so if you want to do Windows-specific things, or you want a very native Windows look and feel, you should use *.NET/Link*. On Windows, *.NET/Link* also does some things that *J/Link* cannot, such as allowing you to call C-style DLL functions directly from *Mathematica* or controlling COM objects.

.NET/Link and *J/Link* provide a very similar programming model. Familiarity with one will be very helpful when working with the other.

Calling .NET from *Mathematica*

Introduction

.NET/Link provides *Mathematica* users with the ability to interact with arbitrary .NET types directly from *Mathematica*. You can create objects and call methods and properties directly in the *Mathematica* language. You do not need to write any .NET code, or prepare in any way the .NET types you want to use. You also do not need to know anything about *MathLink*. In effect, all of .NET becomes a transparent extension to *Mathematica*, almost as if every existing and future .NET type were written in the *Mathematica* language itself.

We call this facility "installable .NET" because it generalizes the ability that *Mathematica* has always had to plug in extensions written in other languages through the `Install` function. Compared to other languages like C or C++, however, .NET*/Link* makes the intermediate steps go away completely, which is why we say that .NET becomes a *transparent* extension to *Mathematica*.

Although .NET is sometimes referred to as an interpreted environment, this is really a misnomer. To use .NET you must write a complete program in a language like C#, compile it, and then execute it. *Mathematica* users have the luxury of working in a true interpreted, interactive environment that lets them experiment with functions and build and test programs a line at a time. *.NET/Link* brings this same productive environment to .NET programmers. You could say that *Mathematica* becomes a scripting language for .NET.

To *Mathematica* users, then, the "installable .NET" feature of *.NET/Link* opens up the universe of .NET types as an extension to *Mathematica*; for .NET users, it allows the extraordinarily powerful and versatile *Mathematica* environment to be used as a shell for interactively developing, experimenting with, and testing .NET programs.

This guide discusses calling from *Mathematica* into the .NET runtime. You will see how to load .NET assemblies and types into *Mathematica*, create objects of these types, call methods and properties, and so on. You will also learn how to use *.NET/Link* to call COM objects as well as standard C-style DLL functions.

Simple Examples:

ProcessPriority.nb

GUI Examples:

Circumcircle.nb

PackageHelper.nb

SimpleAnimationWindow.nb

RealTimeAlgebra.nb

AsteroidsGame.nb

Calling DLLs:

BZip2Compression.nb

EnumWindows.nb

WindowsAPI.nb

Calling COM Objects:

ExcelPieChart.nb

Loading the NETLink` Package

You must load the *.NET/Link* package before you can use *.NET/Link*.

```
Needs["NETLink`"]
```

Launching the .NET Runtime

The `InstallNET` function is used to launch the .NET runtime.

```
InstallNET[];
```

If you are actively developing .NET classes and other types to use in *Mathematica*, you will need to restart the .NET runtime before you can reload a modified version of a class. Use the `ReinstallNET` function to quit and restart the .NET runtime. Most users will have no need to ever quit or restart .NET and should avoid calling `ReinstallNET` or `UninstallNET`. Remember that the .NET runtime is shared by potentially many programs in your *Mathematica* session. Shutting down or restarting the .NET runtime could have unexpected consequences for those programs.

`InstallNET[]`	launch the .NET runtime and prepare it for use from *Mathematica*
`ReinstallNET[]`	quit and restart the .NET runtime
`NETLink[]`	give the `LinkObject` that is being used to communicate with the .NET runtime

Launching the .NET runtime.

Loading .NET Assemblies and Types

.NET Assemblies

Programs and libraries for .NET are packaged into units called *assemblies*. An assembly can be defined as a versioned, self-describing binary (DLL or EXE) containing a collection of types (classes, interfaces, structs, and so on) and optional resources. An assembly can span multiple files (a *multi-file assembly*), or a single file can contain more than one assembly, but in the typical case an assembly consists of a single DLL or EXE file. Although .NET assemblies can have the DLL extension, internally they are quite different from old C-style DLLs. Conceptually, though, a .NET DLL is similar to a C-style DLL in that they are both libraries of code intended to be loaded and called by other programs. An EXE assembly is an executable program that can be launched directly, but it also can export types like a DLL for use by other programs.

Assemblies can be located anywhere on your system. The .NET Framework maintains a special location on your system where assemblies can be stored so that they can easily be found by all .NET programs on the system. This location is called the Global Assembly Cache (GAC), and is found in the `assembly` subdirectory of your root Windows directory. The assemblies that are part of the .NET Framework itself are located in the GAC, and many .NET programs that you install will put assemblies there. There is no requirement that an assembly be placed into the GAC, and *.NET/Link* can use assemblies located anywhere on your system.

Because all .NET types are packaged into assemblies, to load a type into *Mathematica* you first need to load its assembly. You use the `LoadNETAssembly` function to load an assembly into *.NET/Link*. Assemblies can be loaded by specifying various types of information about the assembly, such as a full path to the assembly file, or by a full or partial name of the assembly. Assembly names will be discussed in more detail later, but for now it suffices to say that assembly names are assigned by their creators, and may bear no resemblance to the name of the actual assembly file. An example of a simple assembly name is `System.XML`, which is the assembly in the .NET Framework that handles XML-related functionality. Assemblies are often named after the most important namespace they define. The actual file name of the `System.XML` assembly is `System.XML.dll`, and it is located somewhere nested deep down inside the GAC.

.NET/Link does not need to be explicitly instructed to load all assemblies. It will automatically load any of the .NET Framework assemblies, meaning all of the assemblies containing types whose names start with `System`. (e.g., `System.Windows.Forms.Form`, `System.Drawing.Rectangle`, `System.Data.DataSet`, and so on). Most of the types that you will use in *.NET/Link* programming are found in the system assemblies. You will have to manually load other assemblies. The `LoadNETAssembly` function, described later in this tutorial, is the *Mathematica* function you use to load assemblies into *.NET/Link* and prepare them to be used from *Mathematica*.

.NET Types

A *type* is the fundamental unit of .NET programming. Every type falls into one of the following categories: classes, interfaces, structs ("value types"), enumerations, and delegates. Every object in .NET is an instance of some type. Although the set of types is broader than just classes, you might find it easier to think of types as being classes.

Types are defined in assemblies. To load and use a type in *.NET/Link*, you must first load the assembly in which it resides and then load the type itself. Often these steps can be combined into a single operation. `LoadNETType`, described later , is the *Mathematica* function that you use to load types into *.NET/Link* so they can be used from *Mathematica*.

LoadNETAssembly

`LoadNETAssembly` is the function you use to load assemblies into *.NET/Link* so that the types they contain can be used from *Mathematica*. `LoadNETAssembly` has a number of different argument sequences. Although all these different possible arguments might seem confusing, the basic principle is to allow virtually any way of specifying enough information about the assembly so that *.NET/Link* can locate it.

`LoadNETAssembly[`*assemblyName*`]`	load the specified assembly based on its name, such as "System.Web"
`LoadNETAssembly[`*path*`]`	load the assembly based on its full file path
`LoadNETAssembly[`*url*`]`	load the assembly pointed to by this URL
`LoadNETAssembly[`*assemblyName*`,`*dir*`]`	load the assembly based on its name and the directory in which it resides
`LoadNETAssembly[`*assemblyName*`,`*context*`` `]`	load the assembly based on its name and the application context in which it resides
`LoadNETAssembly[`*dir*`]`	load all the assemblies in this directory
`LoadNETAssembly[`*context*`` `]`	load all the assemblies in this application context

Loading assemblies.

Here is an example of using `LoadNETAssembly` to load an assembly that is part of the .NET Framework.

```
In[3]:= LoadNETAssembly["System.Web"]
Out[3]= NETAssembly[System.Web, 1]
```

The return value of `LoadNETAssembly` is a `NETAssembly` expression. This is not a .NET object itself, just a special expression that can be used in *.NET/Link* to refer to a loaded assembly in various functions that take an assembly specification as an argument.

The name used in the previous example is the *simple name* of the assembly. The actual *full name*, or *display name*, of the assembly is longer and contains version information, among other things. You can use the full name if you want to force a certain version to be loaded (note that if you execute this on your machine, it will fail unless you have exactly the same version of the .NET Framework installed).

```
In[4]:= LoadNETAssembly["System.Web, Version=1.0.5000.0,
            Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"]
Out[4]= NETAssembly[System.Web, 1]
```

We were able to load this assembly based on only its name, without any location information, because it is located in the GAC.

As another example of `LoadNETAssembly`, say you have obtained an assembly or created one of your own in some .NET language and want to load it into *.NET/Link* to use from *Mathematica*. The assembly is in the file `c:\MyProgram\Bin\Debug\MyAssembly.dll`. Here is how you would load it.

```
LoadNETAssembly["c:\\MyProgram\\Bin\\Debug\\MyAssembly.dll"]
```

Loading an assembly by specifying its path is useful for assemblies that you have created yourself.

`LoadNETAssembly` can also load assemblies from an `assembly` subdirectory in a *Mathematica* application directory. This is intended for developers who are creating applications that use *.NET/Link*. If you have a *Mathematica* application directory called `MyApp`, and it is installed into one of the standard locations for *Mathematica* applications (e.g., `<`*`Mathematica`* `dir>\AddOns\Applications`), you can give the `MyApp` directory a subdirectory named `assembly` and place all the extra assemblies that your application needs into the `assembly` directory. Your application's *Mathematica* code can then load one of these application assemblies by supplying the assembly name and the context that corresponds to your application.

```
LoadNETAssembly["My.Special.Assembly", "MyApp`"]
```

In this way, application developers can bundle private assemblies in their application layout and not require that their users perform any special installation steps such as copying assemblies into the GAC.

LoadNETType

LoadNETType is the *Mathematica* function that loads .NET types so that they can be used from *Mathematica*. It is often not necessary to explicitly call LoadNETType. Whenever a .NET object is returned to *Mathematica*, its type is loaded. This means that if you want to create a new object of a certain type, you can just call NETNew and the type will be loaded when the object is returned to *Mathematica*. The most common reason for calling LoadNETType directly is if you want to use a static method or property from a type. In that case, you are not creating an object with NETNew, so you must manually load the type.

LoadNETType[*typeName*]	load the specified type
LoadNETType[*typeName*,*assemblyName*]	load the type from the specified assembly
LoadNETType[*typeName*,*NETAssembly*]	load the type from the assembly identified by a NETAssembly expression
LoadNETType[*typeName*,*assemblyName*,*context*`]	load the type from the specified assembly residing in the specified application context

Loading types.

Here is a simple example of LoadNETType.

```
In[5]:= LoadNETType["System.Windows.Forms.Form"]
Out[5]= NETType[System.Windows.Forms.Form, 1]
```

The return value of LoadNETType is a NETType expression. This is not a .NET object itself, just a special expression that can be used in *.NET/Link* to refer to a .NET type in various functions that take a type specification as an argument.

Note that you must supply the full type name, including the namespace prefix (System.Windows.Forms in this example). For the load to succeed, the assembly in which the type resides must be already loaded using LoadNETAssembly. As mentioned earlier, assemblies for all System types are automatically loaded as needed by *.NET/Link*, so there was no need to load the System.Windows.Forms assembly manually in the previous example.

Viewing Loaded Assemblies and Types

You can use the utility functions `LoadedNETAssemblies.` and `LoadedNETTypes` to see what assemblies and types have been loaded into the current *Mathematica* session. These are intended mainly for debugging purposes.

`LoadedNETAssemblies[]`	return a list of all assemblies loaded into *Mathematica*
`LoadedNETTypes[]`	return a list of all types loaded into *Mathematica*

Viewing loaded assemblies and types.

Contexts and Visibility of Static Type Members

`LoadNETType` has two options that let you control the naming and visibility of static methods and fields. To understand these options, you need to understand the problems they help to solve. We have to get a bit ahead of ourselves to explain the issues, since we have not yet discussed how to call .NET methods. When a type is loaded, definitions are created in *Mathematica* that allow you to call methods, properties, and fields of objects of that class. Static members are treated quite differently from nonstatic ones. Say you have a class named `MyClass` in the namespace `MyCompany.Utilities`, and this class contains a static method named `Foo`. When you load this class, a definition must be set up for `Foo` so that it can be called by name, something like `Foo[args]`. The question becomes: In what context do you want the symbol `Foo` defined, and do you want this context to be visible (i.e., on `$ContextPath`)?

.NET/Link always creates a definition for `Foo` in a context that mirrors its fully qualified classname: `MyCompany`Utilities`MyClass`Foo`. This is done to avoid conflicting with symbols named `Foo` that might be present in other contexts. However, you might find it clumsy to have to call `Foo` by typing the full context name every time, as in `MyCompany`Utilities`MyClass`Foo[args]`. The option `AllowShortContext -> True` (this is the default setting) causes *.NET/Link* to also make definitions for `Foo` accessible in a shortened context, one that consists of just the class name without the hierarchical namespace prefix. In our example, this means that you could call `Foo` as simply `MyClass`Foo[args]`. If you need to avoid use of the short context because there is already a context of the same name in your *Mathematica* session, you

can use `AllowShortContext -> False`. This forces all names to be put only in the "deep" context. Note that even with `AllowShortContext -> True`, names for statics are also put into the deep context, so you can always use the deep context to refer to a symbol if you desire.

`AllowShortContext`, then, lets you control the context where the symbol names are defined. The other option, `StaticsVisible`, controls whether this context is made visible (put on `$ContextPath`) or not. The default is `StaticsVisible -> False`, so you have to use a context name when referring to a symbol, as in `MyClass`Foo[args]`. With `StaticsVisible -> True`, `MyClass`` will be put on `$ContextPath`, so you could just write `Foo[args]`. Having the default be `True` would be a bit dangerous—every time you load a class a potentially large number of names would suddenly be created and made visible in your *Mathematica* session, opening up the possibility for all sorts of "shadowing" problems if symbols of the same names were already present (see Contexts for a discussion of contexts and shadowing problems).

For these reasons `StaticsVisible -> True` is recommended only for classes that you have written, or ones whose contents you are familiar with. In such cases, it can save you some typing, make your code more readable, and prevent the all-too-easy bug of forgetting to type the classname prefix. A classic example would be implementing the venerable "addtwo" *MathLink* example program. In C#, it might look like this.

```
public class AddTwo {
    public static int AddTwo(int i, int j) {return i + j;}
}
```

With the default `StaticsVisible -> False`, you would have to call addtwo as `AddTwo`AddTwo[3, 4]`. Setting `StaticsVisible -> True` lets you write the more obvious `AddTwo[3, 4]`.

Be reminded that these options are only for *static* methods and fields. As discussed later, non-statics are handled in a way that makes context and visibility issues go away completely.

`StaticsVisible→True`	make static methods and fields accessible by just their names, not in a special context
`AllowShortContext→False`	make static methods and fields accessible only in the context that mirrors the full hierarchical namespace name

Options for `LoadNETType`.

Conversion of Types between .NET and *Mathematica*

Before we encounter the operations of creating .NET objects and calling methods, we should examine the mapping of types between *Mathematica* and .NET. When a .NET method returns a result to *Mathematica*, the result is automatically converted into a *Mathematica* expression. For example, .NET integer types (e.g., `Int32`, `Byte`, and so on), are converted into *Mathematica* integers, and .NET real number types (`Single`, `Double`) are converted into *Mathematica* reals. The table below shows the complete set of conversions. These conversions work both ways—for example, when a *Mathematica* integer is sent to a .NET method that requires a `Byte` value, the integer is automatically converted to a .NET `Byte`.

Note that this table gives type names as they are used in the .NET Framework. Different languages often have their own keywords that map to these underlying types. In C#, for example, the keyword `int` is an alias to the `Int32` type, and in Visual Basic .NET the `Int32` type is called `Integer`.

.NET type	*Mathematica type*
`Byte, SByte, Char, Int16, UInt16, Int32, UInt32, Int64, UInt64`	`Integer`
`Decimal`	`Integer` or `Real`
`Single, Double`	`Real`
`Boolean`	`True` or `False`
`String`	`String`
`Array`	`List`
`controlled by user`	`Complex`
`Object`	`NETObject`
`Expr`	`any expression`
`null`	`Null`

Corresponding types in .NET and *Mathematica*.

.NET arrays are mapped to *Mathematica* lists of the appropriate depth. Thus, when you call a method that takes a `double[]` (in C# notation), you might pass it `{1.0, 2.0, N[Pi], 1.23}`. Similarly, a method that returns a two-deep array of integers (`int[,]` in C# notation) might return to *Mathematica* the expression `{{1, 2, 3}, {5, 3, 1}}`.

Creating Objects

To construct .NET objects, use the NETNew function. The first argument to NETNew is the object's type, specified either as a NETType expression returned from LoadNETType or as a string giving the fully qualified type name (i.e., including the namespace prefix). If you wish to supply any arguments to the object's constructor, they follow as a sequence after the type.

NETNew[*typeName*, *arg1*, ...]	construct a new object of the specified class and return it to *Mathematica*
NETNew[*NETType*, *arg1*, ...]	construct a new object of the specified class and return it to *Mathematica*

Constructing .NET objects.

For example, this will create a new Form.

```
In[6]:= form = NETNew["System.Windows.Forms.Form"]
Out[6]= «NETObject[System.Windows.Forms.Form] »
```

The return value from NETNew is a strange expression that looks like it has the head NETObject, except that it is enclosed in angle brackets. The angle brackets are used to indicate that the form in which the expression is displayed is quite different from its internal representation. These expressions will be referred to as NETObject expressions. NETObject expressions are displayed in a way that shows their type name, but you should consider them opaque, meaning that you cannot pick them apart or peer into their insides. You can only use them in *.NET/Link* functions that take NETObject expressions.

NETNew invokes a .NET constructor appropriate for the types of the arguments being passed in, and then returns to *Mathematica* what is, in effect, a reference to the object. That is how you should think of NETObject expressions—as references to .NET objects very much like object references in a .NET language like C# or Visual Basic .NET. What is returned to *Mathematica* is not large no matter what type of object you are constructing. In particular, the object's data (that is, its fields) are not sent back to *Mathematica*. The actual object remains on the .NET side, and *Mathematica* gets a reference to it.

The previous examples specified the class by giving its name as a string. You can also use a NETType expression, which is a special expression returned by LoadNETType that identifies a class. When you specify the class name as a string, the class is loaded if it has not already been.

```
In[7]:= formType = LoadNETType["System.Windows.Forms.Form"];
        form = NETNew[formType];
```

NETNew is not the only way to get a reference to a .NET object in *Mathematica*. Many methods and properties return objects, and when you call such a method or property, a NETObject expression is created. Such objects can be used in the same way as ones you explicitly construct with NETNew.

Calling Methods, Properties, and Fields

Syntax

The *Mathematica* syntax for calling .NET methods and accessing fields is very similar to the syntax used in C# and Visual Basic .NET. The box below compares the *Mathematica* and C# ways of calling constructors, methods, properties, fields, static methods, static properties, and static fields. You can see that *Mathematica* programs that use .NET are written in almost exactly the same way as C# (or VB .NET) programs, except *Mathematica* uses [] instead of () for arguments and @ instead of the . (dot) as the "member access" operator.

An exception is that for static methods, *Mathematica* uses the context mark ` in place of the dot used by C# and VB. This parallels the usage in those languages also, as their use of the dot in this circumstance is really as a scope resolution operator (like :: in C++). Although *Mathematica* does not use this terminology, its scope resolution operator is the context mark. .NET namespace names map directly to *Mathematica*'s hierarchical contexts.

	Constructors
C#:	MyClass obj = new MyClass(args);
Mathematica:	obj = NETNew["MyClass", args];
	Methods
C#:	obj.MethodName(args);
Mathematica:	obj@MethodName[args]
	Properties and fields
C#:	obj.PropertyOrFieldName = 1; value = obj.PropertyOrFieldName;
Mathematica:	obj@PropertyOrFieldName = 1; value = obj@PropertyOrFieldName;
	Static methods
C#:	MyClass.StaticMethod(args);
Mathematica:	MyClass`StaticMethod[args];
	Static properties and fields
C#:	MyClass.StaticPropertyOrField = 1; value = MyClass.StaticPropertyOrField;
Mathematica:	MyClass`StaticPropertyOrField = 1; value = MyClass`StaticPropertyOrField;

C# and *Mathematica* syntax comparison.

You may already be familiar with @ as a *Mathematica* operator for applying a function to an argument: f @ x is equivalent to the more commonly used $f[x]$. *.NET/Link* does not usurp @ for some special operation—it is really just normal function application slightly disguised. This means that you do not have to use @ at all. The following are equivalent ways of invoking a method.

```
(* These are equivalent *)
obj@Method[args];
obj[Method[args]];
```

The first form preserves the natural mapping of the syntax of most .NET languages into *Mathematica* and will be used exclusively in this manual.

When you call methods, properties, or fields and get results back, *.NET/Link* automatically converts arguments and results to and from their *Mathematica* representations according to the table presented earlier.

In object-oriented languages, method and field names are scoped by the object on which they are called. In other words, when you write `obj.Meth()`, .NET languages know that you are calling the method named `Meth` that resides in `obj`'s class, even though there may be other methods named `Meth` in other classes. *.NET/Link* preserves this scoping for *Mathematica* symbols so that there is never a conflict with existing symbols of the same name. When you write `obj@Meth[]`, there is no conflict with any other symbols named `Meth` in the system—the symbol `Meth` used by *Mathematica* in the evaluation of this call is the one set up by *.NET/Link* for this class. Here is an example using a field. First, we create a `Point` object.

```
In[9]:= pt = NETNew["System.Drawing.Point"]
Out[9]= «NETObject[System.Drawing.Point] »
```

The `Point` class has fields named `X` and `Y`, which hold its coordinates. A user's session might also have symbols named `X` or `Y` in it, however. Let us set up a definition for `X` that will tell us when it is evaluated.

```
In[10]:= X := Print["gotcha"]
```

Now set a value for the field named `X` (this would be written as `pt.X = 42` in C# or VB).

```
In[11]:= pt@X = 42;
```

You will notice that "gotcha" was not printed. There is no conflict between the symbol `X` in the `Global`` context that has the `Print` definition and the symbol `X` that is used during the evaluation of this line of code. *.NET/Link* protects the names of members on the right-hand side of `@` so that they do not conflict with, or rely on, any definitions that might exist for these symbols in visible contexts.

In summary, for nonstatic methods, properties, and fields, you never have to worry about name conflicts and shadowing, no matter what context you are in or what the `$ContextPath` is at the moment. This is not true for static members, however. Static methods and fields are called by their full name, without an object reference, so there is no object out front to scope the name. Here is a simple example of a static method call that invokes the .NET garbage collector. We need to call `LoadNETType` before we call a static method to make sure the class has been loaded.

```
In[12]:= LoadNETType["System.GC"];
         GC`Collect[];
```

The name scoping issue is not usually a problem with static members because they are defined in their own contexts (`GC`` in this example). These contexts are usually not on `$ContextPath`, so you do not have to worry that there is a symbol of the same name in the `Global`` context or in a package that has been read. If there is already a context named `GC`` in your session, and it has its own function `Collect`, you can always avoid a conflict by using the fully-hierarchical context name that corresponds to the full type name for a static member.

```
In[14]:= System`GC`Collect[];
```

Underscores in .NET Names

.NET names can have characters in them that are not legal in *Mathematica* symbols. The only common one is the underscore. *.NET/Link* maps underscores in type, method, property, and field names to 'U'. Note that this mapping is only used where it is necessary—when names are used in symbolic form, not as strings. For example, assume you have a class named `My_Class`. When you refer to this class name as a string, you use the underscore:

```
LoadNETType["My_Class"];
NETNew["My_Class"];
```

But when you call a static method in such a class, the hierarchical context name is symbolic, so you must convert the underscore to U.

```
MyUClass`StaticMethod[];
```

The same rule applies to method and field names. To refer to such names in code, use the U. Here is how to call a property named `Some_Property`.

```
In[1]:= obj@SomeUProperty
```

Getting Information about Types and Objects

NETTypeInfo

It is often convenient to be able to quickly display information about the methods, properties, fields, and so on that exist in a given .NET type. *.NET/Link* provides the `NETTypeInfo` function to obtain this information.

NETTypeInfo[*typeName*]	print information about all the members in the given type
NETTypeInfo[*typeName*,*members*]	print information about just the desired types of members in the given type
NETTypeInfo[*typeName*,*members*,"*pat*"]	print information about the desired types of members whose names match a string pattern
NETTypeInfo[*obj*]	print information about all the members in the object's type
NETTypeInfo[*NETAssembly*]	print information about all the types in the given assembly

Getting information about types and objects.

NETTypeInfo will load the type if it has not already been loaded.

This will display a lot of information about the Process class.

```
NETTypeInfo["System.Diagnostics.Process"]
```

The second argument to NETTypeInfo is an optional list of the members you want to see displayed. The possible values are "Type" (gives general info about the type itself), "Constructors", "Methods", "Properties", "Fields", and "Events". This will show just the properties and methods.

```
NETTypeInfo["System.Diagnostics.Process", {"Methods", "Properties"}]
```

This will show just properties with names that begin with "Peak".

```
NETTypeInfo["System.Diagnostics.Process", "Properties", "Peak*"]
```

The default behavior is to display the members in C# syntax. If you want to see them in Visual Basic .NET syntax, use the LanguageSyntax option.

```
NETTypeInfo["System.Diagnostics.Process", LanguageSyntax → "VisualBasic"]
```

option name	*default value*	
LanguageSyntax	"CSharp"	the language syntax in which output should be formatted must be "CSharp" or "VisualBasic"
Inherited	True	whether to include inherited members
IgnoreCase	False	whether to ignore case in matching names to a string pattern

Options to NETTypeInfo.

NETTypeInfo is also useful for seeing what types are in an assembly. To investigate an assembly, pass a NETAssembly expression as the first argument. The easiest way to get a NETAssembly expression is to call LoadNETAssembly (even if the assembly is already loaded). The following line will show a lot of types.

```
NETTypeInfo[LoadNETAssembly["System.Data"]]
```

When acting on a NETAssembly, the second argument to NETTypeInfo is an optional list of the types you want to see displayed. The possible values are "Classes", "Interfaces", "Structures", "Delegates", and "Enums". This will show just the classes and interfaces with the word "Data" in their names.

```
NETTypeInfo[LoadNETAssembly["System.Data"], {"Classes", "Interfaces"}, "*Data*"]
```

Other Useful Functions

NETObjectQ[*expr*]	return True if *expr* is a valid reference to a .NET object, False otherwise
InstanceOf[*obj*,*type*]	return True if this object is an instance of *type*, False otherwise
GetTypeObject[*NETType*]	return the Type object corresponding to a NETType expression
GetAssemblyObject[*NETAssembly*]	return the Assembly object corresponding to a NETAssembly expression

Utility functions for objects and types.

NETObjectQ is convenient when you need to test whether an expression is a .NET object reference. It is often used as a pattern test in function definitions.

```
f[x_?NETObjectQ] := ...
```

.NET/Link uses special expressions with heads NETType (returned by LoadNETType) and NETAssembly (returned by LoadNETAssembly) to represent .NET types and assemblies in *Mathematica*. As noted earlier, you can pass these expressions to functions that take types or assemblies as arguments (such as NETNew). There are times when you might want not a NETType expression, but an actual .NET Type object reference for a given type, and likewise for an assembly. You can use GetTypeObject to get the Type object corresponding to a NETType expression, and GetAssemblyObject to get the .NET Assembly object.

```
In[1]:= netType = LoadNETType["System.Data.DataRow"]
Out[1]= NETType[System.Data.DataRow, 1]

In[2]:= typeObj = GetTypeObject[netType]
Out[2]= «NETObject[System.RuntimeType] »
```

Notice above that the `NETType` expression is much more informative about what type it represents than the `Type` object. This is one reason why *.NET/Link* uses special `NETType` and `NETAssembly` expressions instead of just `Type` and `Assembly` objects.

Once you have a `Type` object, you can use methods and properties of the `Type` class to learn about it.

```
In[3]:= typeObj@FullName
Out[3]= System.Data.DataRow

In[4]:= typeObj@IsSerializable
Out[4]= True
```

Reference Counts and Memory Management

Object References in Mathematica

In our earlier treatment of `NETObject` expressions we avoided discussing deeper issues such as reference counts and uniqueness. Every time a .NET object reference is returned to *Mathematica*, either as a result of a method or property or an explicit call to `NETNew`, *.NET/Link* looks to see if a reference to this object has been sent previously in this session. If not, it creates a `NETObject` expression in *Mathematica* and sets up a number of definitions for it. This is a comparatively time-consuming process. If this object has already been sent to *Mathematica*, in most cases *.NET/Link* simply creates a `NETObject` expression that is identical to the one created previously, which is a much faster operation.

There are some exceptions to this last rule, meaning that sometimes when an object is returned to *Mathematica* a new and different `NETObject` expression is created for it, even though this same object has previously been sent to *Mathematica*. Specifically, any time an object's hash value (as determined by the object's built-in `GetHashCode()` method) has

changed since the last time it was seen in *Mathematica*, the NETObject expression created will be different. You do not really need to be concerned with the details of this, except to remember that *Mathematica*'s SameQ function is not a valid way to compare NETObject expressions to decide whether they refer to the same object. You must use the SameObjectQ function.

SameObjectQ[*obj1*,*obj2*]	return True if the NETObject expressions *obj1* and *obj2* refer to the same .NET object, return False otherwise

Comparing NETObject expressions.

Here is an example.

```
In[1]:= pt = NETNew["System.Drawing.Point", 1, 1]
Out[1]= «NETObject[System.Drawing.Point] »
```

The variable pt refers to a .NET Point object. Now put it into a container so you can get it back out later.

```
In[2]:= vec = NETNew["System.Collections.ArrayList"];
        vec@Add[pt];
```

Now change the value of one of its coordinates. For a Point object, this changes its hash value.

```
In[4]:= pt@X = 2;
```

Now compare the NETObject expression given by pt and the NETObject expression created when you ask for the first element of the ArrayList to be returned to *Mathematica*. Even though these are both references to the same .NET object, the NETObject expressions are different. Recall that the ArrayList class defines an indexer (in C# terminology), so you can use the [] notation to refer to an element by index.

```
In[5]:= pt === vec[0]
Out[5]= False
```

Because you cannot use SameQ (===) to decide whether two object references in *Mathematica* refer to the same .NET object, *.NET/Link* provides the SameObjectQ function for this purpose.

```
In[6]:= SameObjectQ[pt, vec[0]]
Out[6]= True
```

You may be wondering why the SameObjectQ function is useful. Can't you just call an object's Equals() method? It certainly gives the correct result for this example.

```
In[7]:= pt@Equals[vec[0]]
Out[7]= True
```

The problem with this technique is that `Equals()` does not always compare object references. Any class is free to override `Equals()` to provide any desired behavior for comparing two objects of that class. Some classes make `Equals()` compare the "contents" of the objects, such as the `String` class, which uses it for string comparison. The function that provides the correct test is the static method `ReferenceEquals()`.

```
In[8]:= Point`ReferenceEquals[pt, vec[0]]
Out[8]= True
```

You can think of `SameObjectQ` as a convenience function that does the same thing as explicitly calling `ReferenceEquals()`.

In an unusual case where you need to compare object references for equality a very large number of times, the slowness of `SameObjectQ` compared to `SameQ` could become an issue. The only thing that could cause two `NETObject` expressions that refer to the exact same .NET object to not be `SameQ` is if the hash value of the object changed between the times that the two `NETObject` expressions were created. If you know this has not happened, you can safely use `SameQ` to test whether they refer to the same object.

ReleaseNETObject

The .NET runtime has a built-in facility called "garbage collection" for freeing up memory occupied by objects that are no longer in use by a program. Objects become eligible for garbage collection when no references to them exist anywhere, except perhaps in other objects that are also unreferenced. When an object is returned to *Mathematica*, either as a result of a call to `NETNew` or as the return value of a method or property, the *.NET/Lin*k code holds a special reference to the object on the .NET side to ensure that it cannot be garbage-collected while it is in use by *Mathematica*. If you know that you no longer need to use a given .NET object in your *Mathematica* session, you can explicitly tell *.NET/Link* to release its reference. The function that does this is `ReleaseNETObject`. In addition to releasing the *Mathematica*-specific reference in .NET, `ReleaseNETObject` clears out internal definitions that were created in *Mathematica* for the object. Any subsequent attempt to use this object in *Mathematica* will fail.

```
In[9]:= frm = NETNew["System.Windows.Forms.Form"]
Out[9]= «NETObject[System.Windows.Forms.Form] »
```

Now tell .NET that you no longer need to use this object from *Mathematica*.

```
In[10]:= ReleaseNETObject[frm]
```

It is now an error to refer to *frm* because the object's symbolic representation has been removed from the *Mathematica* session. This is what you see if you try to use the released object.

```
In[11]:= frm@Text
Out[11]= Removed[NETObject$83886081][Text]
```

`ReleaseNETObject[`*obj*`]`	let .NET know that you are done using *obj* in *Mathematica*
`NETBlock[`*expr*`]`	all novel .NET objects returned to *Mathematica* during the evaluation of *expr* will be released when *expr* finishes
`BeginNETBlock[]`	all novel .NET objects returned to *Mathematica* between now and the matching EndNETBlock[] will be released
`EndNETBlock[]`	release all novel objects seen since the matching BeginNETBlock[]
`LoadedNETObjects[]`	return a list of all objects that are in use in *Mathematica*

Memory management functions.

Calling `ReleaseNETObject` will not necessarily cause the object to be garbage-collected. It is quite possible that other references to it exist in .NET. `ReleaseNETObject` does not tell .NET to throw the object away, only that it does not need to be kept around solely for *Mathematica*'s sake.

An important fact about the references *.NET/Link* maintains for objects sent to *Mathematica* is that only one reference is kept for each object, no matter how many times it is returned to *Mathematica*. It is your responsibility to make sure that after you call `ReleaseNETObject`, you never attempt to use that object through any reference that might exist to it in your *Mathematica* session.

```
In[3]:= frm1 = NETNew["System.Windows.Forms.Form"];
        frm2 = frm1;
```

If you call `ReleaseNETObject[`*frm1*`]`, it is not the *Mathematica* symbol *frm1* that is affected but the .NET object that *frm1* refers to. Therefore, using *frm2* is also an error (or any other way to refer to this same `Form` object).

Calling `ReleaseNETObject` is often unnecessary in casual use. If you are not making heavy use of .NET in your session, then you will not usually need to be concerned about keeping track of what objects may or may not be needed anymore—you can just let them pile up. There are times, though, when memory use in .NET will be important, and you may need the extra control that `ReleaseNETObject` provides.

NETBlock

`ReleaseNETObject` is provided mainly for developers who are writing code for others to use. Because it is not possible to predict how code will be used, developers should always be sure that their code cleans up any unnecessary references it creates. Probably the most useful function for this is `NETBlock`.

`NETBlock` automates the process of releasing objects encountered during the evaluation of an expression. Often, a *Mathematica* program will need to create some .NET objects with `NETNew`; operate with them, perhaps causing other objects to be returned to *Mathematica* as the results of method calls; and finally return some result such as a number or string. Every .NET object encountered by *Mathematica* during this operation is needed only during the lifetime of the program, much like the local variables provided in *Mathematica* by `Block` and `Module`, and in C#, C++, Java, and many other languages by block scoping constructs (e.g., {}). `NETBlock` allows you to mark a block of code as having the property that any new objects returned to *Mathematica* during the evaluation are to be treated as temporary and released when `NETBlock` finishes.

It is important to note that the preceding sentence said "new objects." `NETBlock` will not cause every object encountered during the evaluation to be released—only those that are being encountered for the first time. Objects that have already been seen by *Mathematica* will not be affected. This means that you do not have to worry that `NETBlock` will aggressively release an object that is not truly temporary to that evaluation.

It is not enough simply to call `ReleaseNETObject` on every object you create with `NETNew`, because many .NET methods and properties return objects. You might not be interested in these return values. You might never assign them to a named variable because they may be chained together with other calls (as in `obj@ReturnsObject[]@Foo[]`), but you still need to release them. Using `NETBlock` is an easy way to be sure that all novel objects are released when a block of code finishes.

`NETBlock[`*expr*`]` returns whatever *expr* returns.

Many *.NET/Link Mathematica* programs will have the following structure.

```
MyFunc[args__] :=
     NETBlock[
          Module[{locals},
               ...
           ]
      ]
```

It is very common to write a function that creates a number of `NETObject` expressions and then returns one of them, the rest being temporary. To facilitate this, if the return value of a `NETBlock` is a single `NETObject`, it will not be released.

```
MyOtherFunc[args__] :=
    NETBlock[
        Module[{obj},
             ...
            obj = NETNew["System.Windows.Forms.Form"];
             ...
            Return[obj]
   (* OK: obj will not be released when NETBlock finishes. *)
        ]
     ]
```

If you want more control over which objects are allowed to escape from a `NETBlock`, you can use the `KeepNETObject` function. Calling `KeepNETObject` on a single object or sequence of objects means they will not be released when the first enclosing `NETBlock` ends. If there is an outer enclosing `NETBlock`, the objects will be freed when *it* ends, however, so if you want the objects to escape a nested set of `NETBlock`s, you must call `KeepNETObject` at each level. Alternatively, you can call `KeepNETObject[`*obj*`, Manual]`, where the `Manual` argument tells *.NET/Link* that the object should not be released by any enclosing `NETBlock`s. The only way such object will be released is if you manually call `ReleaseNETObject` on it.

`KeepNETObject[`*obj1*`,`*obj2*`,…]`	do not release the given objects when the NETBlock ends
`KeepNETObject[`*obj1*`,Manual]`	do not release the given object when any NETBlock ends

Keeping .NET objects after a `NETBlock` ends.

Here is an example that uses `KeepNETObject` to allow you to return a list of two objects without releasing them.

```
MyOtherFunc[args__] :=
    Module[{obj1, obj2, obj3},
        NETBlock[
            obj1 = NETNew[“System.Windows.Forms.Form“;
            obj2 = NETNew[“System.Windows.Forms.Button“];
            obj3 = NETNew[“SomeTemporaryObject“];
            ...
            KeepNETObject[obj1, obj2];
            {obj1, obj2}
        ]
     ]
```

`BeginNETBlock` and `EndNETBlock` can be used to provide the same functionality as `NETBlock` across more than one evaluation. `EndNETBlock` releases all novel .NET objects returned to *Mathematica* since the previous matching `BeginNETBlock`. These functions are mainly of use during development, when you might want to set a mark in your session, do some work, and then release all novel objects returned to *Mathematica* since that point. `BeginNETBlock` and `EndNETBlock` can be nested. Every `BeginNETBlock` should have a matching `EndNETBlock`, although it is not a serious error to forget to call `EndNETBlock`, even if you have nested levels of them—you will only fail to release some objects.

LoadedNETObjects

`LoadedNETObjects[]` returns a list of all .NET objects that are currently referenced in *Mathematica*. This includes all objects explicitly created with `NETNew` and all those that were returned to *Mathematica* as the result of a .NET method or property. It does not include objects that have been released with `ReleaseNETObject` or through `NETBlock`. `LoadedNETObjects` is intended mainly for debugging. It is very useful to call it before and after some function you are working on. If the list grows, your function leaks references, and you need to examine its use of `NETBlock` and/or `ReleaseNETObject`.

Enums

Enumerations in .NET are a special kind of class, with each member of the enumeration represented as a static constant field in the class. Although the values of enumeration constants are integers, *.NET/Link* does not convert them into integers when they are returned to *Mathematica*. This is because enum values are probably only going to be passed back into some other .NET method—it is not likely that you will want to operate on them as integers in *Mathematica*. In this case, it is more meaningful to have them appear in *Mathematica* as objects of a class instead of just cryptic integer values.

Suppose you have a Button object *btn* that you want to anchor in a form so that it stays in a fixed position with respect to one or more edges as the form is resized. To do this, you set the button's Anchor property to a value from the AnchorStyles enum. First load the AnchorStyles type, as is always necessary when you want to access a static member of the type.

```
In[16]:= LoadNETType["System.Windows.Forms.AnchorStyles"]
Out[16]= NETType[System.Windows.Forms.AnchorStyles, 6]
```

You refer to a member of this enum just like any other static field.

```
In[17]:= AnchorStyles`Top
Out[17]= «NETObject[System.Windows.Forms.AnchorStyles] »
```

The enum is represented in *Mathematica* by a strongly typed object reference that is more meaningful to a programmer than the raw integer value, which happens to be 1. We can use the NETObjectToExpression function to convert the object reference to its integer value.

```
In[18]:= NETObjectToExpression[AnchorStyles`Top]
Out[18]= 1
```

Now apply it to the button.

```
In[19]:= btn@Anchor = AnchorStyles`Top;
```

For any argument that is typed as an enum, you can pass either an instance of the enum class or a raw integer value. This means that the above line could also be written as the following, although it is obviously much less readable.

```
In[21]:= btn@Anchor = 1;
```

Some enums have the [Flags] attribute, which indicates that its values can be combined by a bitwise OR. The AnchorStyles enum has this attribute because you might want to anchor a component to more than one edge of its parent container. Here is an example of what that looks like in C#.

```
// C# code
btn.Anchor = AnchorStyles.Top | AnchorStyles.Left;
```

To do this in *Mathematica*, you need to get the integer values of the enum, so you use NETObjectToExpression. (This is just about the only case where you would want to operate on enum values as integers in *Mathematica*.)

```
In[20]:= btn@Anchor = BitOr[NETObjectToExpression[AnchorStyles`Top],
           NETObjectToExpression[AnchorStyles`Left]];
```

"Out" and "Ref" Parameters

.NET allows parameters to be passed by reference, so that changes to their values can be propagated back to the caller. Such "by-reference" parameters are very rarely used in the .NET Framework classes, but are used more commonly in some third-party libraries. In C# notation, such parameters are marked as `out` or `ref`, the difference being that a `ref` parameter needs an initial value on entry to the method, whereas an `out` parameter does not. In Visual Basic .NET, the keyword `ByRef` is used to indicate a parameter passed by reference. `ByRef` parameters in Visual Basic are like `ref` parameters in C#; there is no notion of an `out`-only parameter in Visual Basic. In IDL notation, `ref` parameters are written as `[in, out]` and `out`-only parameters are written as `[out]`.

Here is an example of a `ref` parameter in a method from the `System.Uri` class.

```
// C#
public static char HexUnescape(string pattern, ref int index);

// Visual Basic
Public Shared Function HexUnescape(ByVal pattern As String, ByRef index As
Integer) As Char
```

This method takes a string (the `pattern` parameter) and a starting `index` and reads the next character in the string, decoding a %xx-format hexadecimal representation if necessary. It also advances the `index` value to just past the end of the decoded character. Like most methods that use `ref` or `out` parameters, this method needs to return more than one piece of information—the decoded character and also the next position in the string. Because the `index` parameter's starting value is used by the method, it must be a `ref` parameter and not merely an `out` parameter.

You call methods with `out` or `ref` parameters from *Mathematica* in exactly the same way they are called from most .NET languages, including C# and Visual Basic .NET. For a `ref` parameter, you call the method with a symbol that has an initial value of the correct type (an integer in this case). When the method returns, the symbol will have a new value assigned to it.

This decodes the %20 character (the familiar encoding for a space character [decimal 32] in a URL).

```
In[50]:= LoadNETType["System.Uri"];
         pos = 3;
         Uri`HexUnescape["abc%20def", pos]
Out[52]= 32
```

Because pos was passed to a `ref` parameter slot, it has been assigned a new value, that of the index of the first character after the %20.

```
In[53]:= pos
Out[53]= 6
```

A common mistake when calling `ref` parameters is to forget to assign an initial value. Here is an example of this error.

```
In[54]:= Clear[pos];
         Uri`HexUnescape["abc%20def", pos]

          NET::methodargs: Improper arguments supplied for method named HexUnescape.

Out[55]= $Failed
```

If a parameter is marked as `out` instead of `ref`, the initial value is ignored, so it doesn't matter what value, if any, the symbol had upon entering the method. As mentioned above, Visual Basic .NET has no notion of an `out`-only parameter, so `ByRef` parameters in methods written in Visual Basic will always need an initial value of the correct type, even if the method does not use the incoming value.

Like Visual Basic .NET (but unlike C#), *.NET/Link* allows you to pass a literal value instead of a symbol to an `out` or `ref` parameter. In such cases, any changes made to the parameter's value are lost. Here is an example.

```
In[56]:= Uri`HexUnescape["abc%20def", 3]
Out[56]= 32
```

Returning Objects "By Value" and "By Reference"

References and Values

.NET/Link provides a mapping between certain *Mathematica* expressions and their .NET counterparts. What this means is that these *Mathematica* expressions are automatically converted to and from their .NET counterparts as they are passed between *Mathematica* and .NET. For example, .NET integer types (`Int32`, `Int16`, `Byte`, and so on) are converted to *Mathematica* integers and .NET real types (`Single` and `Double`) are converted to *Mathematica* real numbers. Another mapping is that .NET objects are converted to `NETObject` expressions in *Mathematica*. These `NETObject` expressions are *references* to .NET objects—they have no meaning in *Mathematica* except as they are manipulated by *.NET/Link*. However, some .NET objects are things that have meaningful values in *Mathematica*, and these objects are by default converted to values. Examples of such objects are strings and arrays.

You could say, then, that .NET objects are by default returned to *Mathematica* "by reference", except for a few special cases. These special cases are numbers, strings, arrays, and booleans. You could say that these exceptional cases are returned "by value". The table under "Conversion of Types Between .NET and *Mathematica*" shows how these special .NET object types are mapped into *Mathematica* values.

In summary, every .NET object that has a meaningful value representation in *Mathematica* is converted into this value, simply because that is the most useful behavior. There are times, however, when you might want to override this default behavior. Probably the most common reason for doing this is to avoid unnecessary traffic of large expressions over *MathLink*.

`ReturnAsNETObject[`*expr*`]`	a .NET object returned by *expr* will be in the form of a reference
`NETObjectToExpression[`*obj*`]`	give the value of the .NET object as a *Mathematica* expression

"By reference" and "by value" control.

ReturnAsNETObject

Consider the case where you have a static method in class `MyClass` called `arrayAbs()` that takes an array of doubles and returns a new array where each element is the absolute value of the corresponding element in the argument array. The declaration of this method in C# syntax thus looks like `double[] ArrayAbs(double[] a)`. This is how you would call such a method from *Mathematica*.

```
In[1]:= LoadJNETType["MyClass", StaticsVisible → True];
        ArrayAbs[{1., -2., 3., 4.}]
Out[2]= {1., 2., 3., 4.}
```

The above example is how you probably want the method to work: you pass a *Mathematica* list and get back a list. Now assume you have another method named `ArraySqrt()` that acts like `ArrayAbs()` except that it performs the `Sqrt()` function instead of `Abs()`.

```
In[3]:= ArraySqrt[ArrayAbs[{1., -2., 3., 4.}]]
Out[3]= {1., 1.41421, 1.73205, 2.}
```

In this computation, the original list is sent over *MathLink* to .NET and a .NET array is created with these values. That array is passed as an argument to `ArrayAbs()`, which itself creates and returns another array. This array is then sent back to *Mathematica* via *MathLink* to create a list, which is then promptly sent back to .NET as the argument for `ArraySqrt()`. You can see that it was a waste of time to send the array data back to *Mathematica*—you had a perfectly good array (the one returned by the `ArrayAbs()` method) living on the .NET side, ready to be passed to `ArraySqrt()`, but instead you sent its contents back to *Mathematica* only to have it immediately come back to .NET again as a new array with the same values! For this example, the cost is negligible, but what if the array had 200,000 elements?

What is needed is a way to let the array data remain in .NET and return only a reference to the array, not the actual data itself. This can be accomplished with the `ReturnAsNETObject` function.

```
In[4]:= ReturnAsNETObject[ArrayAbs[{1., -2., 3., 4.}]]
Out[4]= «NETObject[System.Double[]] »
```

Here is how the computation looks using ReturnAsNETObject.

```
In[5]:= ArraySqrt[ReturnAsNETObject[ArrayAbs[{1., -2., 3., 4.}]]]
Out[5]= {1., 1.41421, 1.73205, 2.}
```

Earlier you saw ArraySqrt() being called with an argument that was a *Mathematica* list of reals. Here it is being called with a reference to a .NET object that is a one-dimensional array of doubles. All arguments can be called from *Mathematica* with either a *Mathematica* value or a reference to a .NET object of the appropriate type.

In summary, the ReturnAsNETObject function causes methods and properties that return objects that would normally be converted into *Mathematica* values to return references instead. It is often used as an optimization to avoid unnecessarily passing large amounts of data between *Mathematica* and .NET, and as such it will be useful primarily for very large arrays and strings. Objects of most .NET types have no meaningful "by value" representation in *Mathematica*, and they are always returned "by reference." ReturnAsNETObject is redundant these cases.

NETObjectToExpression

In the previous section, you saw how the ReturnAsNETObject function can be used to cause objects normally returned to *Mathematica* by value to be returned by reference. It is necessary to have a function that does the reverse—takes a reference and converts it to its value representation. That function is NETObjectToExpression.

Keep in mind that almost always when you are dealing with a .NET object that has a meaningful "value" representation in *Mathematica*, the object will be automatically converted to this value when it is sent to *Mathematica*. There are some exceptions to this rule, and these are where NETObjectToExpression becomes useful. You saw earlier that the ReturnAsNETObject function can be used to force an object to be returned as a reference. Another way to get a reference is to call NETNew or MakeNETObject, as these functions always return an object reference. Here we create a String object explicitly.

```
In[1]:= NETNew["System.String", {65, 66, 67}]
Out[1]= «NETObject[System.String] »
```

This converts the string reference to a *Mathematica* string.

```
In[2]:= NETObjectToExpression[%]
Out[2]= ABC
```

The section on "Overloaded Operators" introduces the `MakeNETObject` function, which is easier than using `NETNew` to construct .NET objects out of *Mathematica* strings, numbers, and arrays.

`NETObjectToExpression` also converts into their value representations some object types that are normally returned by reference: enumerations and collections (objects that implement the `ICollection` interface). Enumeration types are discussed in the "Enums" section. Collections can be usefully operated on in *Mathematica* as lists, but unlike arrays, collections might be expensive to iterate through, so *.NET/Link* leaves them as references and does not automatically convert them to lists. If you want a list, use `NETObjectToExpression`.

This creates a collection object:

```
In[1]:= arrayList = NETNew["System.Collections.ArrayList"]
Out[1]= «NETObject[System.Collections.ArrayList] »
```

Now populate it with values.

```
In[2]:= arrayList[Add[#1]] & /@ Range[10];
```

`NETObjectToExpression` converts the object reference to a list.

```
In[3]:= NETObjectToExpression[arrayList]
Out[3]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Overloaded Operators

Some .NET languages allow you to define overloaded operators such as +, >, and so on for a class. Support for overloading of operators is not required in a .NET language. C# and C++ allow it; Visual Basic .NET does not. An example of a class that defines a number of overloaded operators is `System.TimeSpan`. For instance, it defines a + operator so that you can add two `TimeSpan` objects just like they were numbers. Here is what it looks like in C# code.

```
// C# code
TimeSpan t1 = new TimeSpan(2, 45, 55);  // 2 hrs, 45 mins, 55 secs.
TimeSpan t2 = new TimeSpan(3, 10, 25);  // 3 hrs, 10 mins, 25 secs.
TimeSpan sum = t1 + t2;
```

Because .NET languages are not required to support overloaded operators, any class that defines them should always provide some other means to accomplish the same operation, generally via a method call. The `TimeSpan` class provides an `Add()` method that you can use in languages like Visual Basic .NET that do not allow overloaded operators.

.NET/Link does not support overloaded operators in *Mathematica* syntax, so you should seek out the method that performs the same operation. Here is the `Add()` method to add two `TimeSpan` objects.

```
In[63]:= t1 = NETNew["System.TimeSpan", 2, 4, 55];
         t2 = NETNew["System.TimeSpan", 3, 10, 25];
         sum = t1@Add[t2];
         sum@ToString[]
Out[66]= 05:15:20
```

Even if the class author ignored the .NET guidelines and failed to provide an alternative method for accomplishing the same operation as an overloaded operator, you could still invoke the operation from *Mathematica*. This is because overloaded operators in C# and C++ are achieved internally through special static methods with names like `op_XXX` where `XXX` is the name of the operation. The class author does not write these methods directly—they are created by the compiler. Nonetheless, they can be called directly from *Mathematica* like any other method. Here are all these cryptically-named methods in the `TimeSpan` class.

```
In[60]:= NETTypeInfo["System.TimeSpan", "Methods", "op_*"]
```

● ***Methods*** (matching string pattern op_*)

```
static TimeSpan op_Addition(TimeSpan t1, TimeSpan t2)
static bool op_Equality(TimeSpan t1, TimeSpan t2)
static bool op_GreaterThan(TimeSpan t1, TimeSpan t2)
static bool op_GreaterThanOrEqual(TimeSpan t1, TimeSpan t2)
static bool op_Inequality(TimeSpan t1, TimeSpan t2)
static bool op_LessThan(TimeSpan t1, TimeSpan t2)
static bool op_LessThanOrEqual(TimeSpan t1, TimeSpan t2)
static TimeSpan op_Subtraction(TimeSpan t1, TimeSpan t2)
static TimeSpan op_UnaryNegation(TimeSpan t)
static TimeSpan op_UnaryPlus(TimeSpan t)
```

Even if there was no `Add()` method in the `TimeSpan` class, you could still add them together by invoking the `op_Addition()` method. Note that as always when calling .NET names from *Mathematica*, you map the underscore character to a U because underscore is not a legal character in a *Mathematica* symbol.

```
In[62]:= TimeSpan`opUAddition[t1, t2]@ToString[]
Out[62]= 05:15:20
```

Casting

Introduction

.NET programs often include *casts*, where an object of one type is converted to another type. A typical example is where a programmer has a variable of type `Object`, probably obtained as the result of a method call typed to return `Object`, and wants to cast it to a derived type so that methods from that type can be called on it. This often occurs when dealing with collection classes, as they can hold objects of any type and thus their methods are typed to return nothing more specific than `Object`. Here is the signature of the `IList.Item` property, which extracts from a list the object at a given index.

```
' Visual Basic .NET
Default Property Item(index As Integer) As Object

// C#
object this[int index] {get; set;}
```

In C# the `Item` property is the indexer for the class, so it can be written with the above unusual syntax or as a property named `Item`, like in Visual Basic. If you are putting, say, strings into an `IList` and later extract one using the `Item` property, you would have to cast it to a string to be able to assign it to a string variable.

```
// C#
ArrayList aList = new ArrayList;
aList.Add("abc");
...
string s = (string) aList[0];

' Visual Basic .NET
Dim aList As New ArrayList
aList.Add("abc")
...
Dim s as String
s = CType(aList(0), String)
```

In Visual Basic .NET, a cast is performed using the `CType ()` function, and it is only necessary if `Option Strict` is set.

This type of cast is called a *downcast* because you are casting down the inheritance hierarchy (from a parent type to a derived type). Such downcasting is probably the most common form of casting (aside from the casting used to convert numbers of one type to another, such as `int` to `byte`).

Although you will see downcasting scattered throughout the C# and Visual Basic .NET programs that you might be trying to duplicate in *.NET/Link*, downcasting is virtually *never* relevant in *.NET/Link*. This is because casting between reference types is primarily a compile-time operation. In the above sample code, the programmer is telling the compiler that they know the object they just extracted from the `ArrayList` is a string, and they want the compiler to allow them to treat it as one. But in *.NET/Link* objects are always returned to *Mathematica* as their true runtime types, so when you call the `Item` property, you get back an object of type `String`. It is completely irrelevant that the `Item` property is typed to return `Object`. In effect, downcasting is irrelevant and even impossible in *.NET/Link* because every object has its true runtime type—there is no type further down the inheritance hierarchy to cast an object to!

The point of this preamble is to make it clear that the vast majority of casts you see in .NET programs are irrelevant in *.NET/Link*. They are either conversions between numeric types (which are easily done in *Mathematica* by other means if they are even necessary at all), or downcasts from some general type, like `Object`, to a more specific type (which are pointless because they are done for the sake of the compiler and there is no compilation stage in *.NET/Link*).

There are some places in *.NET/Link*, however, where casting is necessary. One case is when working with COM objects. This is discussed in the "COM" section and will not be dealt with here. All the other cases where casting is necessary in *.NET/Link* are *upcasts*, where you are casting an object to a parent class or interface. The three situations where upcasting is necessary are as follows.

- to call a hidden parent-class implementation of a method
- to call methods written using so-called “explicit interface implementation”
- to call methods on a private class that implements a public interface

Note that these are relatively rare circumstances, and many programmers will never encounter them. Although these are nontrivial aspects of .NET programming, there is really no extra complexity introduced by *.NET/Link*—these are precisely the cases where upcasting is required in C#, Visual Basic .NET, and other .NET languages. These three cases will be discussed individually in the following sections.

The function that casts a .NET object reference is `CastNETObject`. You will see examples in the following sections. `CastNETObject` does not create a new object reference, just an "alias" of an existing object reference. This means that if you call `ReleaseNETObject` on an object, the object and all casted references to it are freed. In other words, an object and its casted versions are really just different ways of viewing the same object, not separate references.

`CastNETObject[`*obj*`,`*"type"*`]`	cast the object *obj* to the given type, specified as a string
`CastNETObject[`*obj*`,`*NETType*`]`	cast the object *obj* to the given type, specified as a NET·Type expression

Casting `NETObject` expressions.

The `CastNETObject` function was introduced in *.NET/Link* 1.1.

Calling Hidden Members from a Parent Class

A child class can hide members of its parent class by declaring members with the same name using the `new` keyword (in C#) or `Shadows` keyword (in Visual Basic .NET). Consider the following classes.

```
// C# code
public class Parent {
    public string Foo() { return "from parent"; }
}

public class Child : Parent {
    public new string Foo() { return "from child"; }
}
```

If you had an instance of the `Child` class, but wanted to call the `Parent` implementation of `Foo()`, you could do this by casting the `Child` instance to the `Parent` class.

```
// C# code
Child c = new Child();
string s1 = c.Foo();             // s1 gets the value "from child"
string s2 = ((Parent) c).Foo();  // s2 gets the value "from parent"
```

The behavior is the same whether the `Foo()` method is declared `virtual` or not in the `Parent` class. Note that the `new` keyword (`Shadows` in Visual Basic .NET) is not strictly required, although the compiler will generate a warning if it is left out.

To call the `Parent` class implementation of `Foo()` using *.NET/Link*, use `CastNETObject` to cast the object to the `Parent` class, exactly as was done in the C# code.

```
In[1]:= child = NETNew["Child"]
Out[1]= «NETObject[Child]»

In[2]:= parentCast = CastNETObject[child, "Parent"]
Out[2]= «NETObject[Parent]»

In[3]:= parentCast@Foo[]
Out[3]= from parent
```

Of course the two references refer to the same object.

```
In[4]:= SameObjectQ[parentCast, child]
Out[4]= True
```

Explicit Interface Implementation

If a class implements two interfaces that each have a method of the same name, it can choose to give each interface member a separate implementation. This is called *explicit interface implementation*. This technique is generally only used when the methods from the two interfaces are so different conceptually that there is no way to provide a single implementation that satisfies the contracts of both interfaces. Here is a simplified example from the .NET SDK documentation. The `Box` class implements `IEnglishDimensions` and `IMetricDimensions`, which both have a `Length()` method, and of course there is no single implementation of `Length()` that can work for both interfaces.

```
// C# code

interface IEnglishDimensions {
   double Length();
}

interface IMetricDimensions {
   double Length();
}

class Box : IEnglishDimensions, IMetricDimensions {

   double lengthInches;

   public Box(double length) {
      lengthInches = length;
   }

   // Explicitly implement for IEnglishDimensions:
   double IEnglishDimensions.Length() {
      return lengthInches;
   }

   // Explicitly implement for IMetricDimensions:
   double IMetricDimensions.Length() {
      return lengthInches * 2.54;
   }
}
```

Here is how you would call these methods. You must cast the `Box` object to either the `IMetricDimensions` or `IEnglishDimensions` interface before you can call the `Length()` method.

```
Box myBox = new Box(30.0);
double englishLength = ((IEnglishDimensions) myBox).Length();
double metricLength = ((IMetricDimensions) myBox).Length();
```

Here is how you would do the same thing in *.NET/Link*.

```
myBox = NETNew["Box"];
englishLength = CastNETObject[myBox, "IEnglishDimensions"]@Length[];
metricLength = CastNETObject[myBox, "IMetricDimensions"]@Length[];
```

Here is a real-world example. The `Array` class (which is the parent class for all arrays in .NET) uses explicit interface implementation for methods from the `IList` interface. If a class uses explicit interface implementation for some of its methods this should be mentioned clearly in the documentation, and this is true for the `Array` class. One method from the `IList` interface is `Contains ()`. You cannot call this directly on an array object, even though `Array` implements `IList`.

```
In[6]:= arr = MakeNETObject[{2, 4, 6, 8}]
Out[6]= «NETObject[System.Int32[]]»

In[9]:= arr@Contains[6]

        NET::nomethod : No public instance method named Contains exists for the .NET type System.Int32[].

Out[9]= $Failed
```

It works if you cast to `IList`.

```
In[10]:= CastNETObject[arr, "System.Collections.IList"]@Contains[6]
Out[10]= True
```

Private Class, Public Interface

A final case where upcasting is necessary in *.NET/Link* can occur when you have a method that is typed to return an interface, and the implementation of the method returns an object of a *non-public* class that implements that interface. This is perfectly legal, but it can cause problems for *.NET/Link*. When you reflect on a non-public class to obtain all its public members (*.NET/Link* calls all methods via reflection), you only see the methods that are implemented by some public parent class. Just because a class implements a public interface does not mean that you can call those methods on an instance of that class. If the class itself is not public, even though its methods are public, you can call them only on an instance of the class that is typed as a public parent class or interface.

This may sound confusing, so consider the following example. Assume an interface called `IFoo` and an internal class that implements `IFoo` (internal classes are visible only to other types within the same assembly).

```
interface IFoo {
    int Foo();
}

internal class InternalIFooImpl : IFoo {
    public int Foo() { return 42; }
}
```

Now assume there is some other class that has a method type to return `IFoo` that returns an instance of the `InternalIFooImpl` class.

```
public class FooFactory {
    public static IFoo CreateIFoo() { return new InternalIFooImpl(); }
}
```

```
In[1]:= LoadNETType["FooFactory"];
        foo = FooFactory`CreateIFoo[]
Out[2]= «NETObject[InternalIFooImpl] »
```

It doesn't work to call the `Foo()` method on the `foo` object because it is typed as a non-public class.

```
In[3]:= foo@Foo[]

        NET::nomethod : No public instance method named Foo exists for the .NET type InternalIFooImpl.

Out[3]= $Failed
```

The error message isn't very accurate above—there *is* a public method named `Foo()` in the `InternalIFooImpl` class, it just cannot be seen via reflection because the class itself is not public. This sort of thing never shows up in C# or Visual Basic .NET because the variable that holds the result of `CreateIFoo()` will be typed as the public interface `IFoo`. The programmer would never have any reason to see or even know about the `InternalIFooImpl` class. The code looks like the following.

```
// C# code
IFoo foo = FooFactory`CreateIFoo();
int result = foo.Foo();
```

In *.NET/Link*, however, objects are seen by default as their true runtime types, so we end up with an instance of an object that is typed as a non-public class. The solution is to do what is done in C# and Visual Basic .NET: upcast the object to the `IFoo` interface.

```
In[3]:= ifoo = CastNETObject[foo, "IFoo"];
        ifoo@Foo[]
Out[4]= 42
```

The "factory" design pattern in the above example is relatively common. A special object-creation method returns objects typed only as some interface. This allows the designer of the library to document an interface only, and hide the implementation details in private classes. Clients of the library write only to the interface and are kept completely isolated from details like the actual names of the implementation classes. This suggests that the need for upcasting a non-public class to an interface type would not be rare for *.NET/Link* programmers. In practice, however, it is often the case that the non-public class inherits implementations of at least some of its methods from a public parent class. If this is the case, these methods will be found and can be invoked on the non-public child class without casting.

Indexers

Some .NET classes define a special member that permits instances of the class to be accessed in the same way as arrays. This special member is called an *indexer* in C# terminology, and a *default parameterized property* in Visual Basic .NET terminology. Here are skeleton definitions of example members in C# and Visual Basic .NET.

```
// C# indexer
public int this(int i) {
    get { ... }
    set { ... }
}

// VB default parameterized property
Default Public Property Item(ByVal i As Integer) As Integer
    Get
        ...
    End Get
    Set(ByVal Value As Integer)
        ...
    End Set
End Property
```

Indexers allow a class to act as if it were an array even though it is not. Most of the .NET collection classes (classes that implement the `ICollection` interface) support an indexer so that elements can be set and retrieved using a simple array-like syntax. If a class had a definition like one of the above, you could access the "i^{th} element" using code like the following.

```
// C#
int firstElement = obj[0];

' VB
firstElement = obj(0)
```

If a class defines an indexer, you can call the indexer in *Mathematica* using function brackets.

```
(* Call indexer *)
firstElement = obj[0];
```

Note that there is no method or property name in the above code—just an "argument" to the object itself, as if it were a function. It could be argued that `Part`-based syntax (i.e., using `obj[[0]]`) would be more suitable for calling indexers in *.NET/Link*, as it is the *Mathematica* equivalent of array access, but that syntax was rejected for various philosophical and technical reasons.

Here is another example of calling an indexer using the `BitArray` class, which is a collection of true/false values that is stored in a very compact way. This class defines an indexer so that it can be treated like an array.

```
In[1]:= bitArray = NETNew["System.Collections.BitArray", {True, False, True}]
Out[1]= «NETObject[System.Collections.BitArray] »
```

This calls the indexer to get the second element (because it is a zero-based index).

```
In[2]:= bitArray[1]
Out[2]= False
```

If you write a class in C# and give it an indexer, the compiler creates a public property named `Item` for you. This is a parameterized property, meaning that it takes an argument like a method call. The indexer syntax is just a shorthand for calling the `Item` property. If you are writing in Visual Basic .NET, the convention is that your default parameterized property should be named `Item`, but this is not a requirement. Whatever the default parameterized property is named, you can skip the indexer-style syntax and call the property directly from *Mathematica* if you wish.

```
In[3]:= bitArray@Item[1]
Out[3]= False
```

Exceptions

How Exceptions Are Handled

.NET/Link handles .NET exceptions automatically. If an uncaught exception is thrown during any call into .NET, you will get a message in *Mathematica*. Here is an example that tries to format a real number as an integer.

```
In[26]:= LoadNETType["System.Int32"];
         Int32`Parse["1234.5"]

   NET::netexcptn :
     A .NET exception occurred: System.FormatException: Input string was not in a correct format.
     at System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)
     at System.Int32.Parse(String s, NumberStyles style, IFormatProvider provider)
     at System.Int32.Parse(String s).

Out[27]= $Failed
```

If an exception is thrown, the result of the call will be `$Failed`.

If the .NET code was compiled with debugging information included, the *Mathematica* message you get as a result of an exception will show the full stack trace to the point where the exception occurred, with the exact line numbers in each file.

GetNETException

You can use the function `GetNETException` to get the `Exception` object for the exception thrown in the last call from *Mathematica* to .NET. It returns `Null` if no exception was thrown. Most programmers will have no use for this function, but you could use it to implement a special exception-handling feature in your programs. Below you see the exception thrown by the previous call to `Int32.Parse()`. You can see that most exceptions that occur will come back wrapped in a special `CallNETException` object that wraps the actual exception thrown using the standard .NET design pattern for "inner exceptions".

```
In[28]:= exc = GetNETException[]
Out[28]= «NETObject[Wolfram.NETLink.Internal.CallNETException] »
```

To get the actual exception thrown, you must examine the `InnerException` property.

```
In[29]:= innerExc = exc@InnerException
Out[29]= «NETObject[System.Reflection.TargetInvocationException] »
```

In this example the .NET reflection system has wrapped the exception in another exception, so you have to dig one level deeper to see the "real" one.

```
In[30]:= innerExc@InnerException
Out[30]= «NETObject[System.FormatException] »
```

Custom Exception Handling

Very advanced programmers might want to implement their own system for exception handling and/or reporting. For example, you might want to use *Mathematica*'s `Throw` and `Catch` functions to handle .NET exceptions in *Mathematica* code, using the same programming style that is used in .NET languages. An even simpler example is the desire to silence exception messages in a certain block of code.

To implement custom handling for .NET exceptions in *Mathematica*, use the symbol `$NETExceptionHandler`. The value of `$NETExceptionHandler` is treated as a function that will be passed three arguments: the symbol associated with the message (this will usually be the symbol `NET`), the message tag (this will typically be the string "netexcptn"), and the descriptive string of text associated with the message.

You will usually set $NETExceptionHandler within a Block so that its effect will be limited to a precisely defined segment of code, as in the following example that silences messages.

```
Block[{$NETExceptionHandler = Null &},
 obj@Method[]
]
```

You can use GetNETException within your handler function to obtain the actual .NET exception object that was thrown. Here is an example.

```
exceptionThrower = Throw[GetNETException[]]&;

Block[{$NETExceptionHandler = exceptionThrower},
   ... code that calls .NET ...
]
```

You should avoid setting $NETExceptionHandler outside of a Block, as you are almost sure to inadvertently create situations where its value does not get cleared, leaving users scratching their heads wondering why their exceptions are not being handled like they expect.

Nested Types

Some .NET types have declarations of other types nested within them. Here is an example in C#.

```
namespace SomeNamespace {

    public class Outer {

        public int Foo() { return 42; }
        public static int StaticFoo() { return 42; }

        public class Inner {
            public int InnerFoo() { return 42; }
            public static int StaticInnerFoo() { return 42; }
        }
    }
}
```

In .NET, the + character is used in a type name to separate the name of an inner class from its outer class. In the above example, the actual type name of the Inner class is SomeNamespace.Outer+Inner. This is the name you must use in LoadNETType.

```
In[1]:= innerClass = LoadNETType["SomeNamespace.Outer+Inner"]
Out[1]= NETType[SomeNamespace.Outer+Inner, 25]
```

You also use type names in NETNew. Here is how you construct an instance of the Inner class.

```
In[2]:= innerObj = NETNew["SomeNamespace.Outer+Inner"]
Out[2]= «NETObject[SomeNamespace.Outer+Inner] »
```

The + character only appears in type *names*. When referring to a nested type in *code*, use the standard scope resolution operator (the period in C# and Visual Basic .NET) to separate the inner class from the outer class.

```
// C# code
Outer.Inner obj = new Outer.Inner();
```

Although most .NET languages allow the above syntax for nested types, keep in mind that the actual type names use the + character to separate the inner type from the outer one. In *.NET/Link*, whenever you enter a type name *as a string*, you must use the + notation.

You call instance methods on objects of nested types in the usual way.

```
In[3]:= innerObj@InnerFoo[]
Out[3]= 42
```

Here is how to call a static member. Notice that as in C# and Visual Basic .NET, the + character disappears and is replaced by the scope resolution operator (` in *Mathematica*).

```
In[4]:= Outer`Inner`StaticInnerFoo[]
Out[4]= 42
```

Another point to note about the above line is that you cannot refer to a static member of a nested class using simply the inner class name, as in Inner`StaticInnerFoo[]. This is to be expected, since you cannot do this in other .NET languages either. You always need to prefix the inner class name with the outer one.

Here is a real-world example. The System.Environment class has a nested enum called SpecialFolder. This enum contains constants that designate special locations within the Windows operating system (it has values that include ProgramFiles, Recent, System, StartMenu, and so on). Here is how to determine the path to the user's Favorites folder. You need to call static members from the System.Environment class and the System.Environment.SpecialFolder enum (members of an enum are static), so first you load these two types. Note the + in the type name.

```
In[5]:= LoadNETType["System.Environment"];
        LoadNETType["System.Environment+SpecialFolder"];
```

The `GetFolderPath()` method takes members of the `SpecialFolder` enumeration and gets the appropriate path as a string. Note how we refer to the `Favorites` member of the `SpecialFolder` enum.

```
In[7]:= Environment`GetFolderPath[Environment`SpecialFolder`Favorites]
Out[7]= C:\Documents and Settings\tgayley\Favorites
```

In summary, the important point to remember about nested types is that when you need to refer to the name of one in a *string*, you use the + character to separate the outer type from the inner type. When you refer to a type in *code*, you go back to using the familiar ` to separate the outer and inner names.

MakeNETObject

The most common way to create a .NET object is to call a constructor via `NETNew`. Sometimes, however, you have a *Mathematica* expression that you want to convert into a .NET object but the class does not have a convenient constructor. A common example is if you want to create an array object out of a *Mathematica* list. You can call an array constructor via `NETNew`, but you cannot initialize the array with values via the constructor. This example creates an array object with `NETNew` and fills it manually.

```
In[1]:= intArray = NETNew["System.Int32[]", 3]
Out[1]= «NETObject[System.Int32[]]»

In[2]:= intArray[SetValue[42, 0]];
        intArray[SetValue[43, 1]];
        intArray[SetValue[44, 2]];
```

You can do this much more easily using `MakeNETObject`.

```
In[5]:= intArray2 = MakeNETObject[{42, 43, 44}]
Out[5]= «NETObject[System.Int32[]]»
```

`MakeNETObject[`*val*`]`	construct an object of the appropriate type to represent the *Mathematica* expression *val* (numbers, strings, list, and so on).
`MakeNETObject[`*val*`,`*type*`]`	construct an object of the specified type

`MakeNETObject`.

Keep in mind that you rarely have to call `MakeNETObject`. When you call a method that takes an array, for example, you can just pass a *Mathematica* list and *.NET/Link* will create the .NET array for you. There are times, however, when you want to explicitly create a .NET object that must be populated with data from *Mathematica* and there is no convenient constructor. An example of a circumstance where `MakeNETObject` is useful is the following method, which reverses a list passed in as an argument. Note that it does not return the reversed list, but rather reverses it in place.

```
public static void ReverseArray(int[] a);
```

You could call this method with a *Mathematica* list, but there would be no way to get back the reversed list. The way around this problem is to create an array object populated with the initial list values, pass the object reference, let its internal data be reversed, and then convert the object reference back to a *Mathematica* list.

```
In[7]:= MyClass`ReverseArray[intArray2]

In[8]:= NETObjectToExpression[intArray2]
Out[8]= {44, 43, 42}
```

Another case where is useful is if you need to give *.NET/Link* a little help in choosing the correct signature of an overloaded method. Consider the following two overloads of a method.

```
public void Foo(byte b);
public void Foo(long l);
```

If you called `Foo()` from *Mathematica* with an integer, the overload with the `long` parameter would be called. This is because *.NET/Link* generally tries to call the method with the widest possible type at each slot (but there is no formal guarantee, especially in complicated cases). If you want to call the `byte` version, you can do this by creating a .NET object of the `Byte` type, because *.NET/Link* always gives preference to a method signature that is an exact match for the incoming argument types.

```
obj@Foo[MakeNETObject[42, "System.Byte"]]
```

Remember that `MakeNETObject` is a rarely used function. You do not need to explicitly construct .NET objects from *Mathematica* strings, arrays, and so on, just to pass them to .NET methods—*.NET/Link* does this automatically for you. There are a few special circumstances outlined above where it is useful.

Complex Numbers

.NET number types (e.g., `byte`, `int`, `double`) are returned to *Mathematica* as integers and reals, and integers and reals are converted to the appropriate types when sent as arguments to .NET. What about complex numbers? It would be nice to have a .NET type representing complex numbers that mapped directly to *Mathematica*'s `Complex` type, so that automatic conversions would occur as they were passed back and forth between *Mathematica* and .NET. .NET does not have a standard type for complex numbers, so *.NET/Link* lets you name the type that you want to participate in this mapping.

`SetComplexType["`*classname*`"]`	set the class to be mapped to complex numbers in *Mathematica*
`GetComplexType[]`	return the class currently used for complex numbers

Setting the type for complex numbers.

You can use any class or struct you like as long as it has the following properties:

1. A public constructor that takes two doubles or two floats (the real and imaginary parts, in that order)
2. Public methods, properties, or fields for the real and imaginary parts, having one of the following signatures:

One of:

```
double Re()
double Real()
float Re()
float Real()
```

And one of:

```
double Im()
double Imag()
double Imaginary()
float Im()
float Imag()
float Imaginary()
```

Or a property or field:

```
double Re
double Real
float Re
float Real
```

And one of:

```
double Im
double Imag
double Imaginary
float Im
float Imag
float Imaginary
```

Here is a trivial complex number class in C#.

```
namespace MyCompany {

    public struct Complex {

        public double Re, Im;

        public Complex(double re, double im) {
            Re = re;
            Im = im;
        }

        public Complex Add(Complex c) {
            return this + c;
        }

        public static Complex operator+(Complex a, Complex b) {
            return new Complex(a.Re + b.Re, a.Im + b.Im);
        }
    }
}
```

Assume that you compiled this class into the assembly `MyCompany.Complex.dll`. Here is an example of using it.

```
In[1]:= LoadNETAssembly["c:\\MyCompany.Complex.dll"];
        SetComplexType["MyCompany.Complex"];

In[3]:= c = NETNew["MyCompany.Complex", 2, 1]

Out[3]= «NETObject[MyCompany.Complex] »
```

Once you have used `SetComplexType`, `NETObjectToExpression` will convert object references of that type to complex numbers in *Mathematica* (objects of type `Complex` will normally be converted to complex numbers when returned to *Mathematica*, but a call to a constructor always returns a `NETObject`).

```
In[4]:= NETObjectToExpression[c]

Out[4]= 46. + 2. i
```

Here are three examples of calling the `Add()` method. This method takes one argument of type `Complex`. Note that you can pass a `Complex` object, a *Mathematica* complex number, or a real number for this argument. *.NET/Link* handles any necessary conversions.

```
In[5]:= c@Add[c]
Out[5]= 4. + 2. i

In[6]:= c@Add[42 + 15 i]
Out[6]= 44. + 16. i

In[7]:= c@Add[42]
Out[7]= 44. + 1. i
```

The `Complex` class has an overloaded + operator, and overloaded operators would be expected in any real implementation of a complex number class. As discussed in the "Overloaded Operators" section, you can call these operators from *.NET/Link* by using the special static method equivalents that always exist in the class. For the + operator, the special method is called `op_Addition()`. Here is how to call it (note the required _ to U conversion).

```
In[8]:= Complex`opUAddition[c, c]
Out[8]= 4. + 2. i
```

Because some .NET languages (such as Visual Basic .NET) do not support overloaded operators, class designers usually provide a documented alternative method, such as the `Add()` method in the `Complex` class.

The .NET Console Window

.NET/Link provides a convenient means to display the .NET "console" window. Any output written to the standard `Console.Out` and `Console.Error` streams will be directed to this window. If you are calling .NET code that writes diagnostic information to the console, then you can see this output while your program runs. Like most *.NET/Link* features, the console window can be used easily from either *Mathematica* or .NET programs (its use from .NET code is described in Calling *Mathematica* from .NET). To use it from *Mathematica*, call the `ShowNETConsole` function.

`ShowNETConsole[]`	display the .NET console window and begin capturing output written to `Console.Out` and `Console.Error`
`ShowNETConsole["stream"]`	display the .NET console window and begin capturing output written to the specified stream, which should be "stdout" for `Console.Out` or "stderr" for `Console.Error`
`ShowNETConsole[None]`	stop all capturing of output

Showing the console window.

```
In[1]:= ShowNETConsole[]
Out[1]= «NETObject[Wolfram.NETLink.UI.ConsoleWindow] »
```

Capturing of output only begins when you call `ShowNETConsole`. When the window first appears, it will not have any content that might have been previously written to `Console.Out` or `Console.Error`. Calling `ShowNETConsole` when the window is already open will cause it to come to the foreground.

The next example writes some output from *Mathematica*. If you executed the `ShowNETConsole[]` above, then you will see "Hello from .NET" printed in the window.

```
In[2]:= LoadNETType["System.Console"];
        Console`Out@WriteLine["Hello from .NET"]
```

Although it is convenient to demonstrate writing to the window using *Mathematica* code like this, this is typically done instead from .NET code that writes diagnostic information to the console.

Distributing Applications That Use *.NET/Link*

This tutorial discusses some issues relevant to *.NET/Link* developers who are creating add-ons for *Mathematica*.

.NET/Link is designed to make it easy for application developers to distribute applications that have parts of their implementation in .NET. If you structure your application directory properly, your users will be able to install it simply by copying it into any standard location for *Mathematica* applications. In particular, *.NET/Link* will be able to find your .NET assemblies without users having to perform any special operations or even restart the .NET runtime.

Mathematica applications are typically deployed as single directories (with subdirectories), installed into one of several standard locations where *Mathematica* expects to find them. These standard locations can be written as `$InstallationDirectory\AddOns\Applications`, `$BaseDirectory\Applications`, and `$UserBaseDirectory\Applications`, where `$InstallationDirectory`, `$BaseDirectory`, and `$UserBaseDirectory` refer to the locations given by these built-in *Mathematica* symbols.

.NET/Link applications might include .NET assemblies or legacy Windows DLLs (which can be called from .NET as described in the "Calling DLLs from *Mathematica*" section. If your *Mathematica* application uses *.NET/Link* and includes its own .NET assemblies, you should create an `assembly` subdirectory in your application directory. You can place any assemblies that your application needs into this `assembly` subdirectory. Legacy Windows DLLs (so-called "unmanaged" DLLs) should be placed into into a `Libraries\Windows` subdirectory of your application directory.

Here is an example directory structure for an application that uses *.NET/Link*.

```
MyApp/
    ... other files and directories used by the application ...
    assembly/
        MyAssembly.dll
    Libraries/
        Windows/
            MyLegacyDLL.dll
```

Remember that even if you use the above directory structure, your application code will still have to load its assemblies explicitly. All assemblies, other than ones that make up the .NET Framework itself, must be manually loaded before they can be used, as described in the "Loading .NET Assemblies and Types" section. Having the assemblies in the proper location merely means that they can be found by `LoadNETAssembly` when only a filename or assembly name is supplied.

```
LoadNETAssembly["MyAssembly.dll"]
    (* or *)
LoadNETAssembly["My.Assembly.Name"]
```

Version Information

.NET/Link provides three symbols that supply version information. These symbols provide the same type of information as their counterparts in *Mathematica* itself, except that they are in the `NETLink`Information`` context, which is not on `$ContextPath`, so you must specify them by their full names.

`NETLink`Information`$Version`	a string giving full version information
`NETLink`Information`$VersionNumber`	a real number giving the current version number
`NETLink`Information`$ReleaseNumber`	an integer giving the release number (the last digit in a full x.x.x version specification
`ShowNETConsole[]`	the console window will show version information for the *.NET/Link* assembly

.NET/Link version information.

```
In[1]:= NETLink`Information`$Version
Out[1]= NET/Link Version 1.2.0 (September, 2004)

In[2]:= NETLink`Information`$VersionNumber
Out[2]= 1.2

In[3]:= NETLink`Information`$ReleaseNumber
Out[3]= 0
```

The `ShowNETConsole[]` function, described in "The .NET Console Window" section, will display the version number of the *.NET/Link* assembly file. This version should match the version of the *.NET/Link Mathematica*-language component.

Creating User Interfaces

Introduction

One application of *.NET/Link* is to write user interfaces for *Mathematica* programs. Examples of such interfaces would be a progress bar monitoring the completion of a computation, a window that displays an image or animation, a dialog box that prompts users for input or helps them compose a proper call of an unfamiliar function, or a mini-application that leads users through the steps of an analysis. These types of user interfaces are distinct from what you might write for a .NET program that uses *Mathematica* in the background in that they "pop up" when the user invokes some *Mathematica* code. These user interfaces do not replace the notebook front end; they just augment it. In this way, they are like an extension of the palettes and other specialty notebook elements that you can create in the front end.

Mathematica with *.NET/Link* is an extremely powerful and productive environment for creating user interfaces. The complexity of user interface code is ideally suited to the interactive line-at-a-time nature of *.NET/Link* development. You can build, modify, and experiment with your user interface *while* it is running.

You can use either *.NET/Link* or *J/Link* to build user interfaces for *Mathematica* programs. *J/Link* has the advantage of being cross-platform, so your interface will run on all *Mathematica* systems. *.NET/Link* integrates more tightly with Windows than *J/Link* does, so if you only need your users interfaces to work on Windows machines, then *.NET/Link* is probably the best choice.

If you have used *J/Link* to build user interfaces in *Mathematica*, please note that there are significant differences between *.NET/Link* and *J/Link* in this area. *.NET/Link* is generally simpler than *J/Link*, not because of any superiority of .NET over Java, but because *.NET/Link* is a second-generation design. The design simplifications in *.NET/Link* will eventually be brought to *J/Link.*

Anyone considering writing user interfaces for *Mathematica* programs should also look at the *GUIKit* add-on, which is bundled with *Mathematica* 5.1 and later, and available for download for users with earlier versions of *Mathematica*. *GUIKit* is built on top of *J/Link*, and provides an extremely high-level means of creating interfaces.

Modal versus Modeless Operation

Writing *Mathematica* programs that display .NET user interface elements, such as windows or buttons, requires some knowledge of special issues that are not present in typical *Mathematica* sessions, where only the notebook front end is being used to communicate with the kernel. To help understand these special issues, it is useful to examine some basic considerations about the kernel's "main loop" in which it acquires input, evaluates it, and sends off any output.

When the *Mathematica* kernel is being used from the front end, it spends most of its life waiting for input to arrive on the *MathLink* that it uses to communicate with the front end. This *MathLink* is given by `$ParentLink`, and it is therefore `$ParentLink` that has the kernel's attention. When input arrives on `$ParentLink`, it is evaluated, any results are sent back on the link, and the kernel goes back to waiting for more input on `$ParentLink`. When *.NET/Link* is being used, the kernel has another *MathLink* open—the one that connects to the .NET runtime. When you execute some code that calls into .NET, the kernel sends something to .NET and then blocks waiting for the return value from .NET. During this period when the kernel is waiting for a return value from .NET, the .NET link has the kernel's attention. It is only during this period of time that the kernel is paying attention to the .NET link. A more general way of saying this is that the kernel is only listening for input arriving from .NET when it has been specifically instructed to do so. The rest of the time it is listening only to `$ParentLink`, which is typically the notebook front end.

Consider what happens when the user clicks a button in your .NET window and that button tries to execute some code that calls into *Mathematica*. The .NET side sends something to *Mathematica* and then waits for the result, but the kernel will never get the request because it is only paying attention to the notebook front end link, not the .NET link. It is necessary to use some means to tell the kernel to look for input arriving on the .NET link. *.NET/Link* provides two main ways to manage the kernel's attention to the .NET link and thereby control its readiness to accept requests for evaluations initiated by the .NET side.

These two ways can be called "modal" and "modeless." In modal interaction, characterized by the use of the `DoNETModal` *Mathematica* function, the kernel is pointed at the .NET link until the .NET side releases it. The kernel is a complete slave to the .NET side, and is unavailable for any other computations. In modeless interaction, characterized by the use of the `DoNETModeless`

Mathematica function, the kernel is kept in a state where it is receptive to evaluation requests arriving from either the notebook front end or .NET, evenly sharing its attention between these two programs.

A common type of user interface element is analogous to a modal dialog: once it is displayed, the *Mathematica* program hangs waiting for the user to dismiss the window. Typically, this is because the window returns a result to *Mathematica*, so it is not meaningful for *Mathematica* to continue until the window is closed. An example of such a window is a simple input window that asks the user for some value, which it returns to *Mathematica* when the OK button is clicked.

It is important to understand the slightly generalized use of the term "modal" to describe these windows. They may not be modal in the traditional sense that they must be dismissed before anything else can be done in the user interface. Rather, they are modal with respect to the *Mathematica* kernel—the kernel cannot do anything else until they are closed. A .NET window that you create might not be modal with respect to other .NET windows on the screen, but it ties up the kernel's attention until it is dismissed.

Another type of user interface element is analogous to a modeless dialog: after it is displayed, the *Mathematica* program that created it will finish, leaving the window visible and usable while the user continues working in the notebook front end. An example would be a window that lets users load packages into *Mathematica* by selecting them from a scrolling list. You write a *.NET/Link* program that creates this window, displays it, and returns. The window is left open and usable until the user clicks its close box. In the meantime, the user is free to continue working in the front end, going back to use this .NET window whenever it is convenient.

Such a window is almost like another type of notebook or palette window in the front end. You can have any number of front end or .NET modeless windows open and active at once, meaning that they can be used to initiate computations in *Mathematica*. They are each their own little interface onto the same kernel. What is different about the .NET window is that it is much more general than a notebook window, and, importantly, it exists in a different application layer than the front end. This last fact makes the .NET window, in effect, a second front end, rather than an extension of the notebook front end. To accommodate such a second front end, the kernel must be kept in a special state that allows it to handle requests for evaluations arriving from either the notebook front end or .NET.

Before presenting examples of how to implement modal and modeless windows, it is necessary to jump ahead a little bit and explain the mechanism by which .NET user interface elements communicate events to *Mathematica*.

Handling Events

User interface elements typically have active components such as buttons, scrollbars, menus, and text fields, that need to trigger certain actions when they are used. In the .NET event model, components fire events in response to user actions, and other components indicate their interest in these events by supplying a delegate that connects an event with its handler. The concept of a delegate is covered in detail in the .NET Framework documentation, but *.NET/Link* users can generally ignore the details of delegates because you use a very simple syntax for assigning a *Mathematica* function to be called when an event fires.

It is useful to compare the *.NET/Link* technique for assigning event handler functions with the C# and Visual Basic .NET techniques. The following shows C# and Visual Basic .NET syntax for adding an event handler to the `KeyPress` event in a `TextBox`.

```
// C#
myTextBox.KeyPress += new KeyEventHandler(MyKeyPressHandlerMethod);

' Visual Basic .NET
AddHandler myTextBox.KeyPress, AddressOf MyKeyPressHandlerMethod
```

After executing either of the above lines, the `MyKeyPressHandlerMethod()` method will be called whenever a key is pressed while the `myTextBox` component has the focus. The C# syntax is a little cryptic, as the += operator is overloaded for adding a delegate to an event.

The preceding code does not show the definition of `MyKeyPressHandlerMethod()`. The signature of this method must be the same as the delegate that corresponds to the `KeyPress` event. As you can see in the C# code, the delegate type is `KeyEventHandler`, and here is the declaration.

```
public delegate void KeyEventHandler(object sender, KeyEventArgs
eventArgs);
```

Because `MyKeyPressHandlerMethod()` must have the same signature, it would look something like this.

```
void MyKeyPressHandlerMethod(object sender, KeyEventArgs eventArgs) {
    // Here you respond to the event in some way. The sender object
    // will be the TextBox, and eventArgs will tell you about the event
    // (such as what key was pressed).
}
```

Here is what it looks like in *Mathematica* to assign a `myKeyPressHandler` *Mathematica* function to the `KeyPress` event.

```
AddEventHandler[myTextBox@KeyPress, myKeyPressHandler]
```

Note that it looks almost exactly like the Visual Basic .NET version. Once you have executed the line above, whenever a key is pressed while `myTextBox` has the input focus, .NET will call back to *Mathematica* and execute the `myKeyPressHandler` function. The *Mathematica* function will be called with the same arguments as the `KeyEventHandler` delegate, and it should return the same type of value (although most event handlers return `void`, so the return value is ignored). Here is what it might look like.

```
myKeyPressHandler[sender_, keyEventArgs_] :=
 Print["The " <> keyEventArgs@KeyCode@ToString[] <> "key was pressed"]
```

`AddEventHandler[` *obj*`@`*eventName*`,` *funcName*`]`	set the *Mathematica* function that will be called when the object *obj* fires the *eventName* event
`RemoveEventHandler[`*delegate*`]`	remove an event handler assigned by a previous call to `AddEventHandler`

Assigning the *Mathematica* function that will be called in response to an event notification.

Wiring up your application's event logic via calls to *Mathematica* functions is vastly more flexible than writing a traditional application in .NET. When you write in a compiled .NET language, or use a drag-and-drop GUI builder, you hard code the event logic. You have to decide at compile time what every click, scroll, and keystroke will do. But when you use *.NET/Link*, you decide how your program is wired together at run time. You can even change the behavior on the fly simply by typing a few lines of code.

You can remove an event handler using the `RemoveEventHandler` function. When you call `AddEventHandler`, it returns a `NETObject`. This object is the delegate created for you by the internals of *.NET/Link*. You can save this object and later pass it into `RemoveEventHandler` to remove the *Mathematica* callback that it represents. This is just about the only use you would have for the return value of `AddEventHandler`.

```
dlg = AddEventHandler[myTextBox@KeyPress, myKeyPressHandler];
...
RemoveEventHandler[dlg];
```

Mathematica event handler functions called in response to events are automatically wrapped in `NETBlock`. This means that the objects sent as arguments to the function, as well as any new

objects you create during the execution of the function, are released after the function returns. You do not have to use NETBlock or ReleaseNETObject manually. If you want an object from your handler function to persist in *Mathematica* after the function returns, you must use KeepNETObject to allow it to escape the unseen NETBlock that wraps the function call. Here is a modified version of myKeyPressHandler that stores the KeyCode objects in a list for later inspection.

```
myKeyPressHandler2[sender_, keyEventArgs_] :=
 Module[{keyCode} ,
  keyCode = keyEventArgs@KeyCode;
  AppendTo[keysPressed, keyCode];
  KeepNETObject[keyCode]
 ]
```

AddEventHandler takes two options that control its behavior. SendDelegateArguments allows you to specify which of the delegate arguments you want to send to your *Mathematica* handler function and in which order. By default, *.NET/Link* sends all the delegate arguments, but as an optimization, you might not want to send them all. Creation of a new NETObject expression in *Mathematica* is comparatively expensive, and the arguments to most event delegates are objects. In the case of the KeyPress event example above, the first argument is the TextBox object, which probably already exists in *Mathematica*, so it is not a significant optimization to avoid sending it. The KeyEventArgs object, however, is definitely new to *Mathematica*, so you might want to avoid sending it if you do not need it. Here is an example of setting up an event handler that sends only the first argument to the *Mathematica* callback function.

```
AddEventHandler[myTextBox@KeyPress,
 myKeyPressHandler3, SendDelegateArguments → {1}]
```

This is what the myKeyPressHandler3 function would look like.

```
myKeyPressHandler3[sender_] := Print["A key was pressed"]
```

The values you can specify for the SendDelegateArguments option are All (the default), None, or a list of integers giving the indices of the arguments you want to send.

option name	*default value*	
SendDelegateArguments	All	which delegate arguments to send to the *Mathematica* event handler function
CallsUnshare	False	whether or not your event handler function calls the advanced function UnshareKernel

Options to AddEventHandler.

The CallsUnshare option is for advanced programmers who are using the ShareKernel and UnshareKernel functions to manually control kernel sharing, instead of using DoNETModeless. The sharing functions are discussed in "Manually Sharing the Kernel and Front End with .NET". If your *Mathematica* callback function calls UnshareKernel, you must set CallsUnshare to True in the call to AddEventHandler.

AddEventHandler is a convenience function that allows you to easily assign a *Mathematica* function that will be called when an event fires. As part of its operation, AddEventHandler creates a .NET delegate object that is assigned to the event, and whose action is to call the specified *Mathematica* function. In some cases, you might want to create such a delegate object manually, but not attach it to an event. *.NET/Link* provides the NETNewDelegate function for this purpose. NETNewDelegate creates a delegate of the specified type whose action is to call the designated *Mathematica* function. The main use for NETNewDelegate is to create a delegate object that will be supplied to an external C function invoked via .NET's PInvoke facilities. It is often used in conjunction with DefineNETDelegate for this purpose. The EnumWindows.nb example file demonstrates using DefineNETDelegate and NETNewDelegate to call a C function that takes a callback function pointer as an argument.

NETNewDelegate[*type*, *funcName*]	create a new instance of the specified delegate type whose action is to call the specified *Mathematica* function
DefineNETDelegate[*name*, *returnType*, {*argType*, ...}]	create a new delegate type, for when there is no existing .NET delegate of the appropriate signature

Creating delegate objects that call *Mathematica*.

Now that you have seen how to specify *Mathematica* event handler callbacks, recall that the kernel must be in a special state to be receptive to calls originating from user events in .NET. The two main ways of doing this are the functions DoNETModal and DoNETModeless, discussed in the next sections.

Modal Windows

The basic concepts of modal and modeless *.NET/Link* interfaces are discussed in the earlier section "Modal versus Modeless Operation". Here is an example of a simple modal window. The window is a simple Form object that changes its background color to a new random color each time it is clicked.

```
In[1]:= frm = NETNew["System.Windows.Forms.Form"]
Out[1]= «NETObject[System.Windows.Forms.Form] »
```

Note that simply creating a new `Form` object does not make it visible. Use the `ShowNETWindow` function to make a window visible and to bring it in front of all other windows. The `DoNETModal` function, used later, will make a form visible, but `ShowNETWindow` is useful while you are just tinkering with an interface, before you want to actually run it modally.

```
In[2]:= ShowNETWindow[frm];
```

At this point, you should see a small frame window centered on the screen. Drag it to the side so that it is not hidden by *Mathematica* windows when you bring the current notebook window to the foreground. A huge advantage of using *.NET/Link* for user interface development, compared to a typical compiled .NET language, is that you can experiment with your interface while it is running. Now change the color of the background.

```
In[3]:= LoadNETType["System.Drawing.Color"];
        frm@BackColor = Color`Red;
```

Now add a *Mathematica* handler for the form's `Click` event.

```
In[5]:= AddEventHandler[frm@Click, onClick];
```

This is the definition of the `onClick` function. It sets the form's `BackColor` property to a random color. This function ignores the event arguments, but you can learn what they are from the signature of the `Click` event.

```
In[6]:= onClick[args___] :=
         frm@BackColor =
          Color`FromArgb[Random[Integer, {0, 255}],
           Random[Integer, {0, 255}], Random[Integer, {0, 255}]]
```

At this point, if you click the form, you will get a beep and nothing will happen to the color. When the `Click` event fires, .NET tries to call *Mathematica* to execute the `onClick` function, but it is not safe to make this call because *Mathematica* is not listening for input on the .NET link. The call to *Mathematica* would hang forever. *.NET/Link* knows the kernel is not ready, so it refuses to make the call and issues a beep warning instead.

What you need is a way to put the kernel into a state where it is continuously reading from the .NET link. This is what makes the window "modal"—the kernel cannot do anything else until the window is closed. The function that implements this modal state is `DoNETModal`. The first argument to `DoNETModal` is a top-level window object (in the .NET Framework, this means a `System.Windows.Forms.Form` or any class that inherits from it).

DoNETModal[*form*]	put the kernel into a state where its attention is solely directed at the .NET link until the specified *form* window is closed
DoNETModal[*form*, *returnValue*]	run the *form* modally and return the result of the *returnValue* computation (this is executed before the window is destroyed)

Running modal windows.

Now that everything is ready, you can enter the modal state and use the window.

```
In[7]:= DoNETModal[form]
```

DoNETModal will not return until the .NET form window is closed. Click the window a few times to see the color change, then click the close box in the title bar to destroy the form and cause DoNETModal to return.

You often want to get some information from a modal dialog box when it is closed, such as the value from a text box, or whether the form was closed by clicking an OK or Cancel button. When DoNETModal returns, the form object has been destroyed, so it is too late to call methods on it. If you need to get some information out of a form before it is destroyed, use the optional second argument DoNETModal. This argument specifies a computation that will be executed just before the form is destroyed (it is held unevaluated until this time). DoNETModal will return the result of this computation. The PackageHelper.nb example file shows how to use the second argument to DoNETModal to determine whether a form was closed by clicking the OK or Cancel button.

Here is how the entire example looks when packaged into a single program.

```
In[8]:= SimpleModal[] :=
         NETBlock[
          Module[{frm, onClick},
           frm = NETNew["System.Windows.Forms.Form"];
           LoadNETType["System.Drawing.Color"];
           frm@BackColor = Color`Red;
           AddEventHandler[frm@Click, onClick];
           onClick[args___] := frm@BackColor = Color`FromArgb[Random[Integer, {0, 255}],
              Random[Integer, {0, 255}], Random[Integer, {0, 255}]];
           DoNETModal[frm]
          ]
         ]
```

The .NET Framework documentation discusses how to implement modal windows using the ShowDialog() method. There are some advantages and disadvantages to using this technique

instead of DoNETModal. One disadvantage is that a .NET window is not guaranteed to come in front of notebook windows if you use ShowDialog(). However, this failure seems to happen only for the very first window displayed in a session. Windows displayed with ShowDialog() are truly modal in the sense that other .NET windows cannot be used until the modal one is closed. The ShowDialog() method makes it easy to determine how the window was closed (whether the OK or Cancel button was clicked), because it returns one of the DialogResult enumeration values. You can get the same result by using the second argument to DoNETModal.

```
(* Returns the DialogResult value. *)
DoNETModal[someFormWindow, someFormWindow@DialogResult]
```

The PackageHelper.nb example file demonstrates a classic modal dialog with OK and Cancel buttons implemented with DoNETModal.

DoNETModal takes one option, FormStartPosition, that specifies the position in which the window will appear on screen. The possible values are Center (the default), Automatic (the form will have the Windows default location), and Manual (the form will appear at a location specified elsewhere, for example, by setting the form's Location property).

option name	*default value*	
FormStartPosition	Center	the position in which the window will be displayed on screen

Options for DoNETModal.

Modeless Windows

The previous section demonstrated how to use the DoNETModal function to display and run modal windows, which cause the kernel to remain busy until the window is closed. Another type of window, which could be called modeless, remains open and usable without completely tying up the kernel. The basic concepts of modal and modeless *.NET/Link* interfaces are discussed in more detail in the earlier section "Modal Versus Modeless Operation".

.NET/Link provides the DoNETModeless function to run a window modelessly. The first argument to DoNETModeless is a top-level window (specifically, a Form or any class that inherits from it). The window is made visible and brought in front of all notebook windows.

DoNETModeless[*form*]	display the specified Form window and return immediately, leaving the window active

Modeless windows.

Here is the SimpleModal example from the previous section implemented as a modeless window.

```
In[1]:= SimpleModeless[] :=
         NETBlock[
          Module[{frm},
           frm = NETNew["System.Windows.Forms.Form"];
           LoadNETType["System.Drawing.Color"];
           frm@BackColor = Color`Red;
           AddEventHandler[frm@Click, onClick];
           DoNETModeless[frm]
          ]
         ]

        onClick[sender_, eventArgs_] :=
          sender@BackColor = Color`FromArgb[Random[Integer, {0, 255}],
            Random[Integer, {0, 255}], Random[Integer, {0, 255}]];
```

Executing SimpleModeless[] will make the window visible and then return immediately. You can click the window to change its background color and also continue to use the kernel for other computations via the notebook front end.

```
In[2]:= SimpleModeless[]
```

There are several important differences in the code of SimpleModeless and SimpleModal, beyond the obvious call to DoNETModeless instead of DoNETModal. These differences revolve around the fact that the form runs after the SimpleModeless function returns. The onClick function must not be local to the Module, because the function's definition would be cleared when the Module ends. A NETBlock is used to automatically release all .NET objects created when SimpleModeless runs, but that means that you cannot refer to frm by name in the code for onClick, because the NETBlock has ended by the time onClick is executed (the symbol frm is also local to the Module so its value is cleared anyway). The first argument to the onClick function is the object that fired the event, which is the Form object, so you can use the first argument to refer to it instead of the symbol frm.

DoNETModeless is very useful during development, even for a window that will be run modally in its final form. Because DoNETModal does not return until the window is closed, you cannot modify your event logic or anything else about your window while it is running. You can use DoNETModeless to make your event callbacks "live" without tying up the kernel, so that you can tinker with your window while it is being displayed. When you are satisfied that it works as desired, you can package a complete program that runs with DoNETModal.

option name	*default value*	
FormStartPosition	Center	the position in which the window will be displayed on screen
ActivateWindow	True	whether to make the window visible
ShareFrontEnd	False	whether the front end in addition to the kernel should be shared with .NET

Options for DoNETModeless.

DoNETModeless takes several options. FormStartPosition specifies the position in which the window will appear on screen. The possible values are Center (the default), Automatic (the form will be in the Windows default location), and Manual (the form will appear at a location specified elsewhere, for example, by setting the form's Location property). ActivateWindow controls whether the window should be made visible and brought to the foreground. Set ActivateWindow to False in the rare cases where you want to call DoNETModeless to enter the modeless state, but only later make the window visible.

The final option, ShareFrontEnd, is used to allow .NET interfaces to interact with the notebook front end. The functions ShareKernel and ShareFrontEnd are discussed in the next section, and programmers using DoNETModeless are generally shielded from those low-level functions. DoNETModeless is essentially a means of encapsulating calls to ShareKernel and UnshareKernel, starting sharing when the window first appears and ending it when the window is closed. If you want your modeless interface to cause actions in the notebook front end (such as printing some text or making graphics appear), you need to force DoNETModeless to turn on front end as well as kernel sharing. This can be done by setting the ShareFrontEnd option to True.

One common circumstance where the ShareFrontEnd option is useful is when you want to debug your program by inserting Print statements in event handler functions. In a modeless interface, Print output and *Mathematica* warning messages triggered by event handler functions are sent to the .NET side and therefore do not appear in the notebook. If you want to see this information, setting ShareFrontEnd to True will cause it to appear in the frontmost notebook window. If you use this option during development, but do not need it in the final version of your program, be sure to remove it, as front end sharing is expensive to turn on and off, and can delay the initial appearance of your window.

Manually Sharing the Kernel and Front End with .NET

Note: In *Mathematica* 5.1 and later, the kernel is always shared with the .NET link. This means that the functions `ShareKernel` and `UnshareKernel` are not necessary and, in fact, do nothing at all. If you are writing programs that only need to run in *Mathematica* 5.1 and later, you never need to call `ShareKernel` or `UnshareKernel`. If your programs need to work on all versions of *Mathematica*, then you will need to use these functions as described below.

The previous sections described the `DoNETModal` and `DoNETModeless` functions, which are ways of putting the kernel into a state where it is receptive to calls from events originating in .NET. Most programmers will only need to use those two functions to display and run .NET windows. Recall that `DoNETModeless` makes the kernel receptive to input either from .NET or from the notebook front end. In effect, the kernel is "shared" between the front end and .NET. In some circumstances you might want to initiate kernel sharing, but find that `DoNETModeless` is not appropriate for your needs. You can take direct control of kernel sharing by calling the `ShareKernel` function.

`ShareKernel` was introduced with *J/Link*, and is defined in the `JLink`` context. Loading `NETLink`` loads `JLink``, so you do not have to worry about the different contexts unless you are using `ShareKernel` or related functions in your own package. When you call `BeginPackage`, *Mathematica* only makes available to the code in your package the contexts explicitly named in the `BeginPackage` statement, not other contexts that are needed by those packages. This means that for every symbol that you want to use within a package, you must always explicitly include that symbol's context in your `BeginPackage` statement. Here is an outline of a package that uses *.NET/Link* and also calls `ShareKernel` directly.

```
BeginPackage[“SomePackageThatUsesSharing`“, {“NETLink`“, “JLink`“}]

... code that calls
ShareKernel/UnshareKernel/ShareFrontEnd/UnshareFrontEnd ...

EndPackage[]
```

The "*J/Link* User Guide" discusses `ShareKernel` and `ShareFrontEnd` in detail, and you should refer to those sections for full information. One thing to remember about using `ShareKernel` or `ShareFrontEnd` with .NET is that you must supply the link to .NET as the argument, otherwise they will use the Java link by default.

```
tok = ShareKernel[NETLink[]];
```

Always save the result from ShareKernel to later pass into UnshareKernel.

```
UnshareKernel[tok];
```

As mentioned earlier, DoNETModeless calls ShareKernel to initiate sharing, and arranges for UnshareKernel to be called when the window is closed. In *J/Link*, programmers have to call ShareKernel and UnshareKernel directly to create a modeless window. Having a special function like DoNETModeless that turns sharing on and off automatically is much simpler.

One example of a program that needs to call ShareKernel directly is presented in "Handling COM Events". That program sets up event handlers in *Mathematica* for COM events fired by the Internet Explorer application. There is no top-level .NET Form window to pass into DoNETModeless, so sharing has to be managed explicitly.

Displaying Mathematica Graphics and Typeset Expressions

.NET/Link includes a special subclass of the standard PictureBox class, called Wolfram.NETLink.UI.MathPictureBox, that makes it easy to display *Mathematica* graphics or typeset expressions in a .NET window. The example file SimpleAnimationWindow.nb demonstrates how to use it. You will find complete documentation for this class in the *.NET/Link* API documentation.

Bringing .NET Windows to the Foreground

If you are creating a .NET window with a *Mathematica* program, you probably want that window to pop up in front of the notebook the user is working in so that its presence becomes apparent. The functions DoNETModal and DoNETModeless automatically make the form visible and bring it to the foreground. You might expect that the Show() or Activate() methods of the Form class would also do this for you, but they do not always work because the .NET windows live in a different application than the notebook front end.

.NET/Link provides a *Mathematica* function, ShowNETWindow, that performs all the necessary steps to make a .NET window visible and appear in front of all other windows. You do not need to call ShowNETWindow if you are using DoNETModal or DoNETModeless, as it is called automatically. Even when using DoNETModeless, however, ShowNETWindow can be useful to bring a window back to the foreground if the user has brought other windows in front of it since it was first displayed.

ShowNETWindow[*form*]	make the specified .NET *form* window visible and bring it in front of all other windows, including notebook windows

Bringing a .NET window to the foreground.

Like DoNETModal and DoNETModeless, ShowNETWindow takes the FormStartPosition option, which specifies the position in which the window will appear on screen. The possible values are Center (the default), Automatic (the form will be in the Windows default location), and Manual (the form will appear at a location specified elsewhere, for example, by setting the form's Location property).

Example Files

The following GUI example programs are included with *.NET/Link.*

Circumcircle.nb

PackageHelper.nb

SimpleAnimationWindow.nb

RealTimeAlgebra.nb

AsteroidsGame.nb

Writing Your Own .NET Types to Use from *Mathematica*

Introduction

This documentation has shown you how to load and use existing .NET types. This gives *Mathematica* programmers immediate access to the entire universe of .NET types. Sometimes, though, existing types are not enough, and you need to write your own.

.NET/Link essentially obliterates the boundary between .NET and *Mathematica*, allowing you to pass expressions of any type back and forth and use .NET objects in *Mathematica* in a meaningful way. This means that when writing your own .NET types to call from *Mathematica*, you

usually do not need to do anything special. You write the code in exactly the same way you would if you wanted to use the type only from .NET, and you can use any .NET language you like.

In some cases, you might want to exert more direct control over the interaction with *Mathematica*. For example, you might want a method to send a result to *Mathematica* that is different from what the method actually returns. Or you might want the method to not only return something, but also trigger a side effect in *Mathematica*—for example, printing something or displaying a message under certain conditions. You can even have an extended "dialog" with *Mathematica* before your method returns, perhaps invoking multiple computations in *Mathematica* and reading their results. You might also want to write code that calls into *Mathematica* as the result of some event triggered in .NET.

If you do not want to do any of these things, then you can happily ignore this tutorial. The whole point of *.NET/Link* is to make concern about the interaction with *Mathematica* through *MathLink* unnecessary. Most programmers who want to write .NET types to be used from *Mathematica* will just write .NET types, period, without thinking about *Mathematica* or *.NET/Link*. For those programmers who want more control or want to know more about the possibilities available with *.NET/Link*, read on.

One point to remember when creating your own types to use with *.NET/Link* is that you must use `LoadNETAssembly` to load the assembly that contains the types. Assemblies must always be loaded before the types they contain can be used in *.NET/Link*, but it is easy to forget this because *.NET/Link* automatically loads assemblies that are part of the .NET Framework as they are needed.

Manually Returning a Result to Mathematica

The default behavior of a .NET method or property called from *Mathematica* is to return to *Mathematica* exactly what the method or property itself returns. There are times, however, when you want to return something else. For example, you might want to return an integer in some circumstances and a symbol in others. Or you might want a method to return one thing when it is being called from .NET but return something different when called from *Mathematica*. In these cases, you will need to manually send a result to *Mathematica* before the method returns.

Say you are writing a file-reading class that you want to call from *Mathematica*. Because you want behavior that is almost identical to the standard class `System.IO.StreamReader`, your class will be a subclass of it. The only changes you want to make are to provide some more *Mathematica*-like behavior. One example is that you want the `Read()` method to return not -1 when it reaches the end of the file, but rather the symbol `EndOfFile`, which is what *Mathematica*'s built-in file-reading functions return.

```
// C# code
using System.IO;
using Wolfram.NETLink;

public class MyFileReader : StreamReader {

    ... constructors, other methods deleted ...

    public override int Read() {

        int i = base.Read();
        if (i == -1) {
            IKernelLink link = StdLink.Link;
            if (link != null) {
                link.BeginManual();
                link.PutSymbol("EndOfFile");
            }
        }
        return i;
    }
}
```

If the file has reached the end, `i` will be -1, and you want to manually return something to *Mathematica*. The first thing you need to do is get an `IKernelLink` object that can be used to communicate with *Mathematica*. This is obtained by calling the static property `StdLink.Link`. If you have written installable *MathLink* programs in C, you will recognize the choice of names here. A C program has a global variable named `stdlink` that holds the link back to *Mathematica*. *.NET/Link* has a `StdLink` class that has a few methods related to this link object.

The next thing you do is check whether `Link` returns `null`. It will never be `null` if the method is being called from *Mathematica*, so you can use this test to determine whether the method is being called from *Mathematica*, or as part of a normal .NET program. In this way, you can have a method that can be used from .NET in the usual way when a *Mathematica* kernel is nowhere in sight.

Once you have verified that a link back to the kernel exists, the first thing you do is inform *.NET/Link* that you will be sending the result back to *Mathematica* yourself, so it should not try to automatically send the method's return value. This is accomplished by calling the `BeginManual()` method on the `IKernelLink` object.

You must call `BeginManual()` before you send any part of a result back to *Mathematica*. If you fail to do this, the link will get out of sync and the next *.NET/Link* call you make from *Mathematica* will probably hang. It is safe to call `BeginManual()` more than once, so you do not have to worry that your method might be called from another method that has already called `BeginManual()`.

Returning to the example program, the next thing after `BeginManual()` is to make the required "put"-type calls to send the result back to *Mathematica* (in this case, just a single `PutSymbol()`). The internal *.NET/Link* code that wraps all method calls will handle the cleanup and recovery from any *MathLink* error that might have occurred calling `PutSymbol()`. You do not need to do anything for `MathLinkException` exceptions that occur while you are putting a result manually—the method call will return `$Failed` to *Mathematica* automatically.

Requesting Evaluations by Mathematica

So far, you have seen only cases where a .NET method has a very simple interaction with *Mathematica*. It is called and returns a result, either automatically or manually. There are many circumstances, however, where you might want to have a more complex interaction with *Mathematica*. You might want a message to appear in *Mathematica*, or some `Print` output, or you might want to have *Mathematica* evaluate something and return the answer to you. This is a completely separate issue from what you want to return to *Mathematica* at the *end* of your method—you can request evaluations during the body of a method whether it returns its final result manually or not.

In some sense, when you perform this type of interaction with *Mathematica* you are turning the tables on *Mathematica*, reversing the "master" and "slave" roles for a moment. When *Mathematica* calls into .NET, the .NET code is acting as the slave, performing a computation and returning control to *Mathematica*. In the middle of a .NET method, however, you can call back into *Mathematica*, temporarily turning it into a computational server for the .NET side. Thus you would expect to encounter essentially all the same issues that are discussed in Calling *Mathematica* from .NET, and you would need to understand the full *.NET/Link* API as seen by .NET programmers.

The full treatment of the `IMathLink` and `IKernelLink` interfaces is presented in Calling *Mathematica* from .NET. Here we will discuss a few special methods in `IKernelLink` interface that are specifically intended for use by "installed" methods. You have already seen one, the `BeginManual()` method. This section will present the `Message()`, `Print()`, and `Evaluate()` methods.

The tasks of issuing a *Mathematica* message from a .NET method or triggering some `Print` output are so commonly done that the `IKernelLink` interface has special methods for these operations. The method `Message()` performs all the steps of issuing a *Mathematica* message.

```
void Message(string symtag, params string[] args);
```

The `Print()` method performs all the steps necessary to invoke *Mathematica*'s `Print` function.

```
public void Print(string s);
```

Here is an example method that uses both. Assume that the following messages are defined in *Mathematica*.

```
Foo::arg = "The `1` argument to foo must be greater than or equal to 0."
```

Here is the C# code.

```
public static double Foo(double x, double y) {

    IKernelLink link = StdLink.Link;
    if (link != null) {
        link.Print("inside foo");
        if (x < 0)
            link.Message("Foo::arg", "first");
        if (y < 0)
            link.Message("Foo::arg", "second");
    }
    return Math.Sqrt(x) * Math.Sqrt(y);
}
```

Note that `Print()` and `Message()` send the required code to *Mathematica* and also read the result from the link (it will always be the symbol `Null`).

Here is what happens when you call `Foo()`.

```
LoadNETAssembly["c:\\path\\to\\FooAssembly"];
LoadNETType["FooClass"];
FooClass`Foo[1.0, -2.0]
```

```
inside foo
```

```
Foo::arg : The second argument to foo must be greater than or equal to 0.
```

```
Indeterminate
```

Note that you automatically get `Indeterminate` returned to *Mathematica* when a floating-point result from .NET is NaN ("Not-a-Number").

The methods `Print()` and `Message()` are convenience functions for two special cases of the more general notion of sending intermediate evaluations to *Mathematica* before your method returns a result. The general means of doing this is to wrap whatever you send to *Mathematica* in `EvaluatePacket`, which is a signal to the kernel that this is not the final result, but rather something that it should evaluate and send the result back to .NET. You can explicitly send the `EvaluatePacket` head, or you can use one of the methods in `IKernelLink` that uses `EvaluatePacket` for you. These methods are `Evaluate()`, `EvaluateToInputForm()`, `EvaluateToOutputForm()`, `EvaluateToImage()`, and `EvaluateToTypeset()`. They are discussed in detail in the *.NET/Link* API documentation.

Here is a simple example.

```
public static double Foo(double x, double y) {

    IKernelLink link = StdLink.Link;
    if (link != null) {
        link.Evaluate(“2+2“);
        // Wait for, and then read, the answer.
        link.WaitForAnswer();
        int sum1 = link.GetInteger();

        // EvaluateToOutputForm makes the result come back as a
        // string formatted in OutputForm, and all in one step
        // (no WaitForAnswer call needed).
        string s = link.EvaluateToOutputForm(“3+3“, 0);
        int sum2 = Int32.Parse(s);

        // If you want, put the whole evaluation piece by piece,
        // including the EvaluatePacket head.
        link.PutFunction(“EvaluatePacket“);
        link.PutFunction(“Plus“, 2);
        link.Put(4);
        link.Put(4);
        link.WaitForAnswer();
        int sum3 = link.GetInteger();
    }
    return Math.Sqrt(x) * Math.Sqrt(y);
}
```

Throwing Exceptions

Any exceptions that your method throws will be handled gracefully by *.NET/Link*, resulting in the printing of a message in *Mathematica* describing the exception. This is discussed in "Exceptions". If you are sending computations to *Mathematica*, as described in the previous section, you need to make sure that an exception does not interrupt your code unexpectedly. In other words, if you start a transaction with *Mathematica*, make sure you complete it or you will leave the link out of sync and future calls to .NET will probably hang.

Making a Method Interruptible

If you are writing a method that may take a while to complete, you should consider making it interruptible from *Mathematica*. In C *MathLink* programs, a global variable named `MLAbort` is provided for this purpose. In *.NET/Link* programs, you call the `WasInterrupted` property in the `IKernelLink` interface.

Here is an example method that performs a long computation, checking every 100 iterations whether the user tried to abort it (using the **Interrupt Evaluation** or **Abort Evaluation** commands in the **Evaluation** menu).

```
public int Foo() {

    IKernelLink link = StdLink.Link;
    for (int i = 0; i < 10000, i++) {
        ... perform one step ...
        if (i % 100 == 0 && link.WasInterrupted)
            return 0;  // Return value will not be seen by Mathematica.
    }
    return 42;
}
```

This method returns 0 if it detects an attempt by the user to abort, but this value will never be seen by *Mathematica*. This is because *.NET/Link* causes a method, property, or constructor call that is aborted to return `Abort[]`, whether or not you detect the abort in your code. Therefore, if you detect an abort and want to honor the user's request, just return some value right away. When *.NET/Link* returns `Abort[]`, the user's entire computation is aborted, just as if the `Abort[]` was embedded in *Mathematica* code. This means that you do not have to be concerned with any details of propagating the abort back to *Mathematica*—all you have to do is return prematurely if you detect an abort request, and the rest is handled for you.

.NET/Link makes no distinction between an interrupt request and an abort request; they each cause `WasInterrupted` to return `true`. Recall that *Mathematica* has separate commands for interrupting and aborting computations. The "Abort" operation (ALT-. on Windows) causes the entire computation to end as soon as possible and return `$Aborted`. The "Interrupt" operation (ALT-, on Windows) brings up a dialog box with further choices. If this Interrupt dialog box is triggered when a .NET method is executing, it has a different set of buttons than when normal

Mathematica code is executing. One of the options is Send Abort to Linked Program and another is Send Interrupt to Linked Program. Both of these choices have the same effect for .NET methods, which is to cause `WasInterrupted` to return `true` and the call to return `Abort[]` when it completes. The third button is Kill Linked Program, which will cause the .NET runtime to quit. If you call a .NET method that is not interruptible, killing the .NET runtime in this way is the only way to make the method call terminate. You can also kill the .NET runtime using the Windows Task Manager.

Sometimes you might want a .NET method to detect an abort and do something other than cause the entire *Mathematica* computation to abort. For example, you might want a loop to stop and return its results up to that point. Note that this is not generally recommended. Users expect a program to abort and return `$Aborted` when they issue an abort request. In some cases, however, especially if the code is not intended for use by a large community, you might find it useful to use an abort as a "message" to communicate some information to your .NET code instead of just having the computation aborted. This idea is similar to *Mathematica*'s `CheckAbort` function, which allows you to detect an abort and absorb it so that it does not propagate further and abort the entire computation. To "absorb" the abort in your .NET code so that *.NET/Link* does not return `Abort[]`, simply reset the `WasInterrupted` property to `false`.

Here is an example.

```
public int Foo() {

    IKernelLink link = StdLink.Link;
    for (int i = 0; i < 10000, i++) {
        ... perform one step ...
        if (i % 100 == 0 && link.WasInterrupted) {
            link.WasInterrupted = false;
            return resultSoFar; // This is the value that will be returned
to Mathematica
        }
    }
    ...
    return 42;
}
```

Writing Your Own Event Handler Code

"Handling Events" introduced the topic of triggering calls into *Mathematica* as a response to events fired in .NET, such as clicking a button. The `AddEventHandler` function provides an easy means of setting up event handlers in *Mathematica*. You are not required to use `AddEventHandler`, of course. You can create your own delegates in any .NET language to handle events and insert calls into *Mathematica* directly into their code. If you choose to do this, there is one very important rule that must be adhered to when writing event handler code that calls into *Mathematica*. You must always call `StdLink.RequestTransaction()` before sending a computation to *Mathematica*. `RequestTransaction()` will throw an exception if *Mathematica* is not in a state where it is receptive to calls originating in .NET. You can catch this exception if you wish, or you can ignore it and *.NET/Link* will catch it for you and issue a warning beep. In other words, `RequestTransaction()` prevents you from interfering with the internals of *.NET/Link* by trying to call the kernel when the kernel and *.NET/Link* are not ready.

To be precise, `RequestTransaction()` will throw an exception unless one of the following conditions holds.

- *Mathematica* 5.1 or later is being used (in 5.1 and later the kernel is always shared with .NET)
- *Mathematica* is executing `DoNETModal`
- *Mathematica* is executing `DoNETModeless`, and the kernel is not busy servicing a computation from the front end
- Kernel sharing has been turned on via `ShareKernel` or `ShareFrontEnd`, and the kernel is not busy with another computation
- *Mathematica* is already in the middle of a call to .NET
- .NET is not being used from *Mathematica* (that is, the code is called in a standalone .NET program that is using *Mathematica* for computations; `InstallNET` has not been called)

The fifth bullet point above warrants further discussion. Recall that whenever *Mathematica* has called into .NET, it is reading from the .NET link and therefore is receptive to calls that arrive from .NET. This means that callbacks to *Mathematica* from code that is itself called from *Mathematica* do not need to call `RequestTransaction()` (although it is harmless to do so). Event handlers, however, are typically called as the result of some user action on the .NET side, and thus they originate in .NET.

The code below shows a prototypical event handler method written in C#.

```
// Add a delegate to the KeyPress event.
myTextBox.KeyPress += new KeyEventHandler(MyKeyPressHandler);
...

// Elsewhere, define MyKeyPressHandler as follows.
public void MyKeyPressHandler(object sender, KeyEventArgs eventArgs) {

    IKernelLink ml = StdLink.Link;
    StdLink.RequestTransaction();
    // Send a single computation to Mathematica to react to the KeyPress
event.
    ml.PutFunction("EvaluatePacket", 1);
    ... code to put rest of expression to evaluate goes here ...
    ml.EndPacket();
    ml.WaitAndDiscardAnswer();
}
```

Debugging Your .NET Classes

You can use your favorite debugger to debug .NET code that is called from *Mathematica*. The only issue is that you typically have to launch a .NET program inside the debugger to do this. The .NET program that you need to launch is `InstallableNET.exe`, which is the program that is normally launched for you when you call `InstallNET`. This program resides in the `NETLink` directory right next to the `Wolfram.NETLink.dll` assembly file.

If you are using Visual Studio .NET and you want to debug a class library project that you are using with *.NET/Link*, the exact steps depend on the language you are using. Select the project, and choose Properties from the Project menu. In the Debugging panel of the Configuration Properties section, set the startup application to be the `InstallableNET.exe` program. Set the command-line arguments to something like `–linkmode listen –linkname foo`. Then start the debugger. The `InstallableNET` program will launch and wait for *Mathematica* to connect. In your *Mathematica* session, execute the following.

```
InstallNET[LinkConnect["foo"]]
```

This works because `InstallNET` can take a `LinkObject` as its argument, in which case it will not try to launch .NET itself. This allows you to manually establish the *MathLink* connection between .NET and *Mathematica*, then feed that link to `InstallNET` and let it do the rest of the work of preparing the *Mathematica* and .NET sides to interact with each other.

Calling DLLs from *Mathematica*

Introduction

This section describes how you can use *.NET/Link* to call DLL functions from *Mathematica*. These are traditional Windows DLLs, typically C-language libraries (although many languages have the capability to create such DLLs). In .NET terminology, these types of DLLs are called "unmanaged" because they do not execute within the .NET runtime. Although the task of calling unmanaged functions does not appear to have anything to do with .NET, *.NET/Link* can leverage existing facilities in .NET for calling such functions. In other words, because .NET can call DLLs, and *Mathematica* can call .NET, you can now easily call DLLs in *Mathematica*.

The capability to easily call external C functions from *Mathematica* means that Windows programmers have virtually no reason to ever write another so-called "template" *MathLink* program that wraps an external function and directly handles *MathLink* communication. Other tutorials have shown how *.NET/Link* eliminates the need for special programming to call .NET code. In this tutorial we see how *.NET/Link* also eliminates those extra steps for calling legacy DLLs.

Many programming languages allow you to call DLL functions by simply "declaring" them with a line of code. Here are examples of such a declaration in several languages. The function is `GetTickCount()`, which is part of the Windows API that is defined in `kernel32.dll`.

```
// Visual Basic 6
Declare Function GetTickCount Lib "kernel32" () As Long

// Visual Basic .NET
<DLLImport("kernel32.dll")>
Shared Function GetTickCount() As Integer

// C#
[DllImport("kernel32.dll")]
static extern int GetTickCount();
```

The *Mathematica* function DefineDLLFunction is analogous to the above declarations. You specify the name of the DLL, the name of the function, and the types of the return value and arguments.

```
In[142]:= getTickCount = DefineDLLFunction["GetTickCount", "kernel32.dll", "int", {}]
Out[142]= Function[Null, If[NETLink`DLL`Private`checkArgCount[GetTickCount, {##1}, 0],
            Wolfram`NETLink`DynamicDLLNamespace`DLLWrapper2`GetTickCount[##1], $Failed], {HoldAll}]
```

Note that DefineDLLFunction returns a function. You should assign it to a symbol and then use that symbol as the name of the function:

```
In[147]:= getTickCount[]
Out[147]= 789 292 652
```

There are four arguments to DefineDLLFunction. The first argument is the name of the function in the DLL. The second argument is the name of the DLL, and you can give either the full pathname to the DLL or just its filename. ("How DLLs are Found" discusses in detail how DLLs are found by *.NET/Link*.) The third argument is the return type, given as a string. The fourth argument is a list of the types of the arguments. If the function takes zero arguments, like GetTickCount(), you specify an empty list. The type specifications are strings, and *.NET/Link* supports a number of ways to specify types. Type specifications are discussed in greater detail in the section "Specifying Arguments and Return Values".

DefineDLLFunction["*funcName*", "*dllName*", *returnType*, {*argType*, ...}]	create a *Mathematica* function suitable for calling the named function from the named DLL
DefineDLLFunction["*declaration*"]	create a Mathematica function from a complete external function delcaration given in C# syntax

Defining DLL functions.

DefineDLLFunction supports several options. The first is CallingConvention, which you need to use if the DLL function uses a calling convention different from the standard convention, which is "stdcall" on versions of Windows other than Windows CE. In rare cases, functions use the "cdecl" calling convention. The "thiscall" convention can be used when calling methods in C++ classes. See the .NET Framework documentation for the System.Runtime.InteropServices.CallingConvention enumeration for more information on these values. For most uses, you leave the CallingConvention option at its default setting.

The MarshalStringsAs option is discussed in the section "Strings". The ReferencedAssemblies option is discussed in the section "Declarations Requiring Special Attributes".

option name	*default value*	
CallingConvention	Automatic	the calling convention expected by the DLL function (possible values are "StdCall", "CDecl", "ThisCall", and Automatic.
MarshalStringsAs	"ANSI"	how string arguments (char*, string, String) should be marshaled to and from the DLL function (possible values are "ANSI", "Unicode", and Automatic)
ReferencedAssemblies	Automatic	a list of the names of assemblies referenced by your declaration

Options for DefineDLLFunction.

How DLLs Are Found

The second argument to DefineDLLFunction is the name of the DLL in which the function resides. You can specify a full pathname to the DLL or just give the filename and rely on *.NET/Link*'s automatic search mechanism to find it. DLLs can be found by just their filename if they are located on your system's PATH or if they are in special subdirectories within *Mathematica* application directories. This automatic search of application directories allows you to distribute *Mathematica* applications that include one or more DLLs without requiring your users to install the DLL files in a separate location outside your application directory.

Mathematica applications are typically deployed as single directories (with subdirectories), installed into one of several standard locations where *Mathematica* expects to find them. These standard locations can be written as $InstallationDirectory\AddOns\Applications, $BaseDirectory\Applications, and $UserBaseDirectory\Applications, where $InstallationDirectory, $BaseDirectory, and $UserBaseDirectory refer to the locations given by these built-in *Mathematica* symbols. If your *Mathematica* application includes DLLs that are intended to be called via *.NET/Link*, your application directory needs to be installed into one of these standard locations, and the DLLs need to be placed into a Libraries\Windows subdirectory of your application directory. "Distributing Applications that use *.NET/Link*" discusses the layout of an application directory in more detail.

When you call DefineDLLFunction, no attempt is made to locate the DLL—that happens only when the function is first called. This means that if *.NET/Link* cannot find the DLL or the named function within it, you will see an error message only when the function is called, not when it is defined.

Specifying Arguments and Return Values

Introduction

To specify the types of the return value and arguments you use strings such as "int", "double", and "void". You can use type names that conform to the syntax of whichever language you are most comfortable with (C, C#, or Visual Basic .NET). You can also use many type names used in the Windows API, such as "WORD", "BOOL", and "LPSTR". In most cases, you will be working from a function prototype in the C language, and it will be most convenient to use the names directly from the prototype. For example, consider the following declaration for the `floor()` function from the `math.h` header file in the Standard C library.

```
double floor(double x);
```

It is not likely that you would want to call this particular mathematical function from *Mathematica*, but it serves as a simple example from a DLL that everyone will have. On Windows, the C runtime library is in the DLL `msvcrt.dll`. Here is one way to use `DefineDLLFunction` to create a *Mathematica* function that calls the `floor()` function.

```
In[158]:= externalFloor = DefineDLLFunction["floor", "msvcrt.dll", "double", {"double"}];

In[161]:= externalFloor[4.2]
Out[161]= 4.
```

`DefineDLLFunction` allows you to use the C type names directly, which is handy when you are looking at a C-language prototype from a header file. The following are equivalent ways of making the same definition.

```
(* Using Visual Basic-style names. *)
DefineDLLFunction["floor", "msvcrt.dll", "Double", {"Double"}];

(* Using Visual Basic-style names with full VB syntax. *)
DefineDLLFunction["floor", "msvcrt.dll", "Double", {"ByVal d As Double"}];

(* Using.NET Framework names. *)
DefineDLLFunction["floor", "msvcrt.dll", "System.Double", {"System.Double"}];
```

Using C# or Visual Basic .NET syntax for type names is convenient if you are copying an external function declaration from some sample code in one of those languages. In fact, the easiest way to use `DefineDLLFunction` is to find an existing declaration for the external function in some C# or Visual Basic .NET sample code, and just copy the type names used in that declaration. Here is what declarations for `floor()` would look like in those languages.

```
// Visual Basic .NET
<DLLImport("msvcrt.dll")>
Shared Function floor(ByVal d As Double) As Double

// C#
[DllImport("msvcrt.dll")]
static extern double floor(double);
```

You can also use a Visual Basic 6 `Declare Function` statement as a guide to the correct type names, but keep in mind some important differences between VB 6 and VB .NET. First, in VB 6 parameters are `ByRef` by default (they are `ByVal` by default in VB .NET), so a type name like `Double` in a VB 6 declaration should be translated to `ByRef Double`. Also, `Integer` in VB 6 is equivalent to `Short` in VB .NET, and `Long` in VB 6 is equivalent to `Integer` in VB .NET. You must use type names that are appropriate for VB .NET.

The following subsections discuss allowed type names in greater detail.

Primitive Types

The following table shows what type names are legal to use for primitive types (i.e., integers, reals, booleans) and what types they map to in *Mathematica*.

Type in External Function Declaration	*Mathematica Type*
C-language names: char, int, short, long (and unsigned versions) *C# names:* byte, sbyte, char, short, int (and unsigned versions) *Visual Basic .NET names:* Short, Integer, Long (these are all ByVal) *.NET Framework names:* Byte, SByte, Char, Int16, UInt16, Int32, UInt32, Int64, UInt64 *Win32 API names:* BOOL, BYTE, SHORT, INT, UINT, LONG, WORD, DWORD, LPARAM, WPARAM	Integer
float, double, Single, Double	Real
bool, Boolean	True or False
void, Void	Null, when used for a return value; for zero-argument functions, use {} as the argument type list

Legal type specifications for primitive types in DefineDLLFunction.

The use of the above type names should be straightforward. Note that "long" means a standard Windows C long (4 bytes), not the C# long, which is 8 bytes. Also, the Windows API BOOL type is mapped to an integer (0 and non-zero) rather than True and False. Pointers to, and arrays of, primitive types are discussed in a later subsection.

Strings

There are some subtleties you need to be aware of when calling DLL functions that take and return strings. These center on how the DLL function expects strings to be represented: either as ANSI-style, single-byte, null-terminated strings or as Unicode, double-byte, null-terminated strings. The process of converting data from one representation to another as it moves across system boundaries is called *marshaling*. The default behavior of .NET when calling unmanaged code is to marshal strings as single-byte, null-terminated strings, because most DLL functions are written to handle this common C string format. If you need different behavior than the default, you can use the `MarshalStringsAs` option to `DefineDLLFunction`.

Here are examples of two DLL functions from the Windows C runtime library that operate on different types of strings. Each converts a string to lowercase; the `_strlwr()` function takes an ANSI string and the `_wcslwr()` function takes a wide-character string.

```
char *_strlwr(char *string);

wchar_t *_wcslwr(wchar_t *string);
```

Here are calls to `DefineDLLFunction` for both of these functions. Because the `_wcslwr()` function takes and returns a wide-character string, you need to override the default marshaling of strings.

```
In[197]:= strlwr = DefineDLLFunction["_strlwr", "msvcrt.dll", "string", {"string"}];
          wcslwr = DefineDLLFunction["_wcslwr", "msvcrt.dll",
             "string", {"string"}, MarshalStringsAs → "Unicode"];
```

Both functions behave identically on a string with characters that fit into a single byte.

```
In[199]:= strlwr["ABC"] == wcslwr["ABC"] == "abc"
Out[199]= True
```

As expected, the `strlwr` function fails on a string with characters that require two bytes. Note that the π character is truncated to a single byte (in this example the truncation occurs as the string is passed from .NET into the DLL, but it would also happen on the way back out of the DLL).

```
In[200]:= strlwr["AπB"]
Out[200]= apb
```

Things work when you call the wide-character version:

```
In[202]:= wcslwr["AπB"]
Out[202]= aπb
```

If you have more than one string as a parameter or return type and the strings must be marshaled differently, then you cannot use the `MarshalStringsAs` option, as this applies to all strings in the function. You can use the special "full declaration" form for `DefineDLLFunction`, as discussed in "Declarations Requiring Special Attributes".

Type in External Function Declaration	*Mathematica Type*
C-language names:	`String`
`char*`	
C# names:	
`string`	
Visual Basic .NET names:	
`String`	
.NET Framework names:	
`String`	
Win32 API names:	
`LPSTR, LPCSTR`	

Legal type specifications for strings in `DefineDLLFunction`. These are all equivalent.

In `DefineDLLFunction` for `_strlwr()` and `_wcslwr()`, we used the type name "string," which is C# syntax, to indicate a string. As the table above shows, the following are completely equivalent declarations.

```
(* C syntax: *)
DefineDLLFunction["_strlwr", "msvcrt.dll", "char*", {"char*"}];

(* C# syntax: *)
DefineDLLFunction["_strlwr", "msvcrt.dll", "string", {"string"}];

(* VB.NET syntax: *)
DefineDLLFunction["_strlwr", "msvcrt.dll", "String", {"String"}];

(* Alternate VB.NET syntax: *)
DefineDLLFunction["_strlwr", "msvcrt.dll", "String", {"ByVal d As String"}];

(* .NET Framework names: *)
DefineDLLFunction["_strlwr", "msvcrt.dll", "System.String", {"System.String"}];
```

So far we have only discussed strings used as "[in]" parameters to functions—that is, where character data is being sent *into* the function. Some functions that are typed to take `char*` use

the string as an "[out]" parameter, meaning that it is actually a buffer that is written into by the function. Functions that use strings as [out] parameters typically require you to pass in an extra argument that gives the length of the string buffer you have allocated, or they have a documented maximum number of characters that they will write into your buffer. An example of a function that writes data into a string is the familiar Standard C library function `sprintf()`.

```
int sprintf(char *buffer, const char *format [,argument] ...);
```

The `buffer` argument is a string into which the function writes. It is an [out] parameter, and although you could pass a string of data into this function to be overwritten (provided it was long enough so that the written data did not overrun the length of the string), you would have no way of getting the modified string back out. The .NET runtime supports a special trick for working with [out] string parameters, which is to use an instance of the `System.Text.StringBuilder` class. A `StringBuilder` is marshaled to an unmanaged function as a character buffer. After the function returns, the `StringBuilder` object will hold the data that was written into the buffer. You can extract the data as a string using the `StringBuffer.ToString()` method.

Let's see how this is done with the `sprintf()` function. This function takes a variable argument count, but `DefineDLLFunction` cannot handle that, so we will define a version specifically for the case of one integer argument after the format string (three total arguments).

```
In[222]:= sprintf = DefineDLLFunction["sprintf", "msvcrt.dll",
              "int", {"System.Text.StringBuilder", "const char*", "int"}];
```

One small point to note in the above call to `DefineDLLFunction` is that you can specify a `const` qualifier on any argument slot. It is ignored by *.NET/Link* because it is not relevant for *.NET/Link*'s purposes, but you can use it if you think it makes your declarations more self-documenting or if you are just blindly copying a C function prototype.

To call `sprintf` you first create a `StringBuilder` object that has a buffer large enough to hold all the data that might be written into it. This example will use a small string, so 20 bytes is more than enough.

```
In[223]:= sb = NETNew["System.Text.StringBuilder", 20];
          sprintf[sb, "xxx%daaa", 42]
Out[224]= 8
```

The return value is the number of characters written into the buffer. To see the string, call `ToString()`.

```
In[225]:= sb@ToString[]
Out[225]= xxx42xxx
```

Arrays and Pointers

When dealing with functions that take or return pointers or arrays, you need to be a little careful and make sure you understand how the parameter is treated by the function you are calling. For example, if you see a parameter of type `int*`, it could be any of the following:

- an array of integers passed in to the function (an [in] array)
- an array of integers that will be written into by the function (an [out] array), that possibly also requires initial values in the array (an [in, out] array)
- the address of an integer variable that will have a value written into it by the function (an [out] int), that possibly also requires an initial value (an [in, out] int)

Each of these possibilities requires a different type specification in `DefineDLLFunction`. As an example, consider the `modf()` function from the Standard C library:

```
double modf(double d, double* pint);
```

This function breaks up the double `d` into an integer plus a fraction. The fraction is the return value, and the integer value is stored in the double pointed to by `pint`. From this description you can see that the `double*` parameter is not an array, but the address of a double that will get written into (an [out] `double`). Try calling `DefineDLLFunction` using the type names directly from the prototype and see what you get.

```
In[251]:= modf = DefineDLLFunction[“modf“, “msvcrt.dll“, “double“, {“double“, “double*“}];
```

When you use a pointer type like `double*` directly in `DefineDLLFunction`, as in the above example, *.NET/Link* assumes that the parameter is an [in, out] `double` (the last item in the bullet list above). In C# notation this type of parameter is called `ref double`, and in Visual Basic .NET notation it is `ByRef As Double`. From “Out” and “Ref” Parameters, you know that what you need to pass to a `ref` parameter slot is a symbol that has a value of the correct type going in, and that this symbol will also be assigned a possibly modified value on the way out. This means that to call `modf` and have it assign the second argument to a symbol called `integerPart`, we need to give `integerPart` a numerical value before the call or *.NET/Link* will complain about bad arguments.

```
In[253]:= integerPart = 0;
          modf[3.5, integerPart]
Out[254]= 0.5

In[255]:= integerPart
Out[255]= 3.
```

This is not ideal because the second argument is conceptually an [out] `double`, not an [in, out] `double`—its value going into the function is not used, so there is no point in having to give it any value before the call. To improve the definition you can use "`out double`" instead of "`double*`" as the type specification.

```
In[256]:= betterModf =
            DefineDLLFunction["modf", "msvcrt.dll", "double", {"double", "out double"}];

          Clear[integerPart];
          betterModf[3.5, integerPart]
          integerPart
Out[258]= 0.5

Out[259]= 3.
```

In summary, if you use a pointer type directly in `DefineDLLFunction`, *.NET/Link* will treat this as a `ref` (C# notation) or `ByRef` (VB notation) parameter. If this does not correctly capture the use of the parameter, you should use a different type specification. I prefer to use C# notation for type names, so that an `int*` parameter that is treated as an address of an integer that will be written into is specified as "`out int`". If the value is both read and written by the function it is a "`ref int`". Visual Basic .NET does not have a syntax for a "pure" [out] parameter, so using C# notation is the best choice for that case.

So far we have not considered the case of an array parameter. In a C function prototype, `int*` could mean an array of `int`s, although this would often be written as `int[]`. If you need to pass an array of data to a DLL function, the parameter type must be written with array brackets, as in "`int[]`", not as "`int*`" (you have already seen how `DefineDLLFunction` treats types declared explicitly as pointers). Consider the following two (fictitious) DLL function prototypes.

```
int SumArray(int[] array, int length);

void ReverseArray(int[] array, int length);
```

The `SumArray()` function takes an array of integers and the length of the array, and it returns the sum. Here is how you would write `DefineDLLFunction` and the call to the function.

```
SumArray = DefineDLLFunction[“SumArray“, “SomeDLL.dll“, “int“, {“int[]“, “int“}];

(* Alternate version using VB .NET syntax:
    SumArray = DefineDLLFunction[“SumArray“,
   “SomeDLL.dll“, “Integer“, {“x() As Integer“, “Integer“}]; *)

result = SumArray[{2, 4, 6, 8, 10}, 5];
```

Note that bracket notation is used to indicate an array passed into the function, and you call it from *Mathematica* with a list of integers.

The `ReverseArray()` function is quite different. It reverses the array of data *in place*, so the array is used as an [in, out] parameter. You can define it as follows:

```
ReverseArray =
  DefineDLLFunction[“ReverseArray“, “SomeDLL.dll“, “void“, {“int[]“, “int“}];
```

But consider what would happen if you called it like this:

```
ReverseArray[{2, 4, 6, 8, 10}, 5];
```

This would succeed in the sense that the DLL function would receive an array of `int`s and reverse it, but there is no way to propagate the modified array back out of the function. You might guess that "`ref int[]`" would be the correct type to use, but that actually translates to `(int[])*` because adding `ref` or `out` to a type is effectively adding a level of indirection. The trick is to create a .NET array object and pass that object into the function. After the function returns, you can get the array data as a *Mathematica* list. This works because anywhere in *.NET/Link* if you have an argument slot typed to take an array you can call it from *Mathematica* with either a list or a reference to a .NET object that is an array of the appropriate type. Here is the proper way to call a DLL function that writes into an array:

```
(*This creates a.NET object of type Int32[] and fills it with
 the data. If you didn‘t care about the initial values in the array,
you could use NETNew[“System.Int32[]“, 5].*)
intArray = MakeNETObject[{2, 4, 6, 8, 10}];

ReverseArray[intArray, 5];

(* This converts the object reference to its value as a Mathematica list. *)
NETObjectToExpression[intArray]
```

But consider what would happen if you called it like this:

```
ReverseArray[{2, 4, 6, 8, 10}, 5];
```

The example files include more pointer-related techniques, including the use of the .NET Framework `IntPtr` type to represent a generic pointer.

Function Pointers

Some DLL functions take a callback function pointer as an argument. .NET maps function pointers to delegates, so you pass a delegate object of the appropriate type for a function pointer argument. "Handling Events" introduces the `NETNewDelegate` function, and its main use is for creating delegate objects for function pointers in DLL calls. It is often the case that there is no existing .NET delegate type with the correct signature for the function pointer. You can use the `DefineNETDelegate` function to create a .NET delegate type with the appropriate signature. The EnumWindows.nb example file demonstrates using `DefineNETDelegate` and `NETNewDelegate` to call a DLL function that takes a callback function pointer as an argument.

Declarations Requiring Special Attributes

The .NET runtime supports a large number of attributes to control precisely how a function is called and how arguments are marshaled. The `CallingConvention` and `MarshalStringsAs` options to `DefineDLLFunction` give you some control over these aspects, but they do not support anywhere near all of the available attributes. Here is an example from the .NET Framework documentation of a complicated C# declaration for the `MoveFile()` function from the Windows API. Although this declaration was made deliberately over-complicated, it demonstrates some of the possible attributes.

```
[DllImport("KERNEL32.DLL", EntryPoint="MoveFileW", SetLastError=true,
CharSet=CharSet.Unicode, ExactSpelling=true,
CallingConvention=CallingConvention.StdCall)]
public static extern bool MoveFile(String src, String dst);
```

When faced with the need to specify attributes beyond what can be done with options to `DefineDLLFunction`, you can use an alternative form where you specify a full declaration as a string in C# syntax.

```
MoveFile =
  DefineDLLFunction["[DllImport(\"KERNEL32.DLL\", EntryPoint=\"MoveFileW\",
     SetLastError=true, CharSet=CharSet.Unicode,
     ExactSpelling=true,\nCallingConvention=CallingConvention.StdCall)]
     public static extern bool MoveFile(String src, String dst);"];
```

In this version of *.NET/Link*, only C# syntax is supported not Visual Basic .NET.

Another example of a function that would need this type of full declaration would be one that had two string arguments that needed separate marshaling conventions.

```
void TwoStrings(char* ansiString, wchar_t* unicodeString);
```

Here is how you would define it in *Mathematica.*

```
TwoStrings =
  DefineDLLFunction["[DllImport(\"SomeDLL.dll\")] public static extern void
     TwoStrings([MarshalAs(UnmanagedType.LPStr)] string ansiString,
     [MarshalAs(UnmanagedType.LPWStr)] string unicodeString);"];
```

If your DLL declaration uses a type that is not in the System assembly, you need to use the `ReferencedAssemblies` option to specify its assembly. This is analogous to adding a reference to an assembly in a Visual Studio project. Here is an example of using this option. The `Rectangle` class is found in the `System.Drawing` assembly, so you will get an error unless you explicitly name it as a referenced assembly.

```
GetWindowRect = DefineDLLFunction["GetWindowRect",
    "user32.dll", "BOOL", {"HWND", "ref System.Drawing.Rectangle"},
    ReferencedAssemblies → {"System.Drawing.dll"}];
```

Example Files

The following example programs, included with *.NET/Link*, demonstrate calling C-style DLLs from *Mathematica.*

BZip2Compression.nb

WindowsAPI.nb

EnumWindows.nb

Calling COM from *Mathematica*

Introduction

The .NET runtime has many features that support interoperability with COM (the Component Object Model, also referred to as ActiveX). Although the arrival of .NET makes COM/ActiveX officially a "legacy" technology, there are still a huge number of COM objects and libraries in use, and COM remains an important part of the Windows programming world. Because COM objects are easily called from .NET, they are easily called from *Mathematica* via *.NET/Link*.

COM programming is a complex subject (one reason, no doubt, that Microsoft replaced it with .NET), and readers are assumed to have some familiarity with the basics of COM. There is considerable discussion of COM and .NET interoperability issues in the .NET SDK documentation.

The central element in .NET-to-COM interoperability is a special proxy object called a Runtime Callable Wrapper. Whenever you create a COM object and want to import it into the .NET environment, the .NET runtime creates an RCW object that represents the COM object in the .NET environment. The RCW mediates calls from .NET into COM, marshaling arguments and return values back and forth between the .NET and COM worlds. You will see examples of RCW objects in the sections that follow.

.NET/Link provides two main ways of calling COM objects from *Mathematica*. The first technique is to use so-called COM Automation (late binding). This is convenient because it requires no preparation, but it is not ideal for various reasons discussed later. The second, preferred, technique is to create or obtain an interop assembly for the COM objects you want to call. An interop assembly is a special .NET assembly that wraps a COM library and makes that library's types and interfaces look like native .NET types. These two methods for calling COM objects are discussed in the next two sections.

Using Automation (Late Binding)

A COM interface is essentially just a table of function pointers. This is ideal for C++ programmers to use, but there needs to be a way for scripting languages, which have no compilation stage and no access to C++ header files, to use COM objects. The solution to this problem is a special COM interface called `IDispatch`. A COM object that implements `IDispatch` allows the user of the object to determine at runtime the methods and properties available, and then invoke them. `IDispatch` is the COM equivalent to the "reflection" capabilities in .NET. Using COM objects via their `IDispatch` interface is often referred to as late binding, Automation, or Dispatch. We will use the term Automation.

Not all COM objects support Automation but many do, including all those that want to be usable from the widest variety of programming languages and environments. Visual Basic 6 is capable of using COM objects via either Automation or early binding (discussed later). Most scripting languages, including VBScript, however, can only use Automation. *.NET/Link* can use either Automation or early binding. Early binding is the preferred technique, and it is discussed in the next section. Here, we focus on Automation. Using COM objects via Automation in *.NET/Link* is almost exactly like using Automation in Visual Basic or VBScript. If you can find sample code in either of those languages that uses the COM objects you are interested in, then you can generally translate that code verbatim into *Mathematica*.

As an example of a COM library, this section will use the Microsoft Speech API. Although Microsoft will eventually move all its APIs to pure .NET implementations, many are still available only as COM objects, and the Speech API is an example. (Note that Microsoft has a .NET-based speech tool called the Speech Application SDK. This is targeted at ASP .NET developers creating telephony applications and should not be confused with the older COM-based Speech API, which will be used in the next examples.) You do not need to have the Speech API installed to understand this section, as you can simply read the inputs and outputs, but if you want to reevaluate the input or play around yourself, you can download the Speech API from http://www.microsoft.com/speech/download/sdk51/. If the line that calls `CreateCOMObject` fails, you do not have the Speech API installed.

The basic function for creating COM objects for control via Automation is `CreateCOMObject`. This function is analogous to the `CreateObject()` function in Visual Basic. The argument to `CreateCOMObject` is a string that provides either the ProgID or CLSID of a COM coclass. A ProgID is a human-readable string, such as "Excel.Application", whereas the CLSID is a sequence of hex digits, such as "{000208d5-0000-0000-c000-000000000046}". You can obtain the CLSID or ProgID of a COM object from its documentation, or, even better, from some sample code in Visual Basic that uses it.

`CreateCOMObject[“ProgID“]`	create a COM object with the given ProgID (e.g., Excel.Application)
`CreateCOMObject[“CLSID“]`	create a COM object with the given CLSID (e.g., {000208d5-0000-0000-c000-000000000046})
`GetActiveCOMObject[“ProgID“]`	acquire a reference to an already-active COM object with the given ProgID (e.g., Excel.Application)
`GetActiveCOMObject[“CLSID“]`	acquire a reference to an already-active COM object with the given CLSID (e.g., {000208d5-0000-0000-c000-000000000046})

Obtaining COM objects.

This creates an instance of the SpVoice COM object.

```
In[1]:= voice = CreateCOMObject[“Sapi.SpVoice“]
Out[1]= «NETObject[COMInterface[SpeechLib.ISpeechVoice]] »
```

The `voice` object is unlike any .NET object you have seen yet. This object is a Runtime Callable Wrapper (RCW), a class of objects that was mentioned in the Introduction. You can think of it

as a proxy object that represents the COM object in the .NET world. For most .NET objects, the string inside the brackets in the `OutputForm` representation of the object gives the name of the object's .NET type. The `voice` object is different—the string shows the name of the default COM interface supported by the object (`SpeechLib.ISpeechVoice`). Here is the actual type name of the object.

```
In[2]:= voice@GetType[]@ToString[]
Out[2]= System.__ComObject
```

As you might have guessed, `System.__ComObject` is the name of the RCW class. Seeing a .NET object represented in *Mathematica* as `<<NETObject[System.__ComObject]>>` would not be very informative, as this could be any COM object and thus tells you nothing about the object. When *.NET/Link* returns an RCW object to *Mathematica*, it tries to determine the name of the default COM interface that the object supports. If this is successful, then *.NET/Link* reports the object as `<<NETObject[COMInterface[Default.COM.Interface]]>>`. Remember that this is the name of a COM interface and has no meaning whatsoever to .NET or *.NET/Link*. It is displayed simply to help you know something about the COM object that this .NET object represents. For *.NET/Link* to be able to determine the default COM interface name, the object must provide sufficiently detailed type information via a COM type library. Most COM objects provide this feature, so you will often see RCW objects formatted with a COM interface name. In some cases, however, the search for a default interface name will fail, and you will see a COM object formatted only as `<<NETObject[System.__ComObject]>>`.

One drawback to using COM objects via Automation is that you cannot get information about COM methods and properties using `NETTypeInfo`.

```
In[3]:= NETTypeInfo[voice]

        NETTypeInfo::com : Type information is not currently available for "raw" COM objects.

Out[3]//TableForm=
```

Although you can call methods and properties on the COM object, you cannot see these methods directly from .NET. For information about methods and properties and their arguments, you will need to turn to the documentation for the COM object. Often you can find sample code in Visual Basic for using a COM object, and using it from *Mathematica* via *.NET/Link* will look almost exactly the same.

The ISpeechVoice COM interface includes a property called Volume.

```
In[4]:= voice@Volume
Out[4]= 100
```

The Speak() method speaks a string of text. Here is the declaration for the Speak() method from the IDL file for the SpeechLib type library.

```
long Speak([in] BSTR Text, [in, optional, defaultvalue(0)]
SpeechVoiceSpeakFlags Flags);
```

Experienced COM programmers will recognize the elements of this declaration. The first argument is a BSTR, which is a string in the COM world. Such arguments are called with a string from .NET, and thus with a string from *Mathematica*. The second argument is marked as optional, with a default value of 0. This means that the Speak() method can be called without the second argument.

```
In[5]:= voice@Speak["This is an example of using the SpVoice object via Automation"];
```

The second argument is listed as being of the type SpeechVoiceSpeakFlags, which is a COM enumeration containing constants that control how the text is spoken. One drawback to using COM objects via Automation is that there is no way to access COM enumerations. To use the second argument, you have to supply an integer value that corresponds to the correct value of the enum. You can get this information from the documentation or from the type library itself using a tool like OLE View, which is bundled with Microsoft Visual Studio. To speak the voice asynchronously, meaning the Speak() method will return before the text finishes speaking, you use the flag SVSFlagsAsync, which has the value 1.

```
In[6]:= voice@
          Speak["This is an example of using the SpVoice object via Automation. This
             text is spoken asynchronously", 1];
```

Drawbacks to Using Automation

You have seen how *.NET/Link* allows you to use COM objects via their IDispatch interface. This has the advantage of requiring no preparation at all, but there are several drawbacks:

- You can only call methods on an object's default interface.
- You cannot use NETTypeInfo to get information about the methods and properties of an object.
- You cannot access COM enumerations or structs.
- You cannot use COM events.

These drawbacks are the same as you would encounter using COM from a pure scripting language like VBScript. The next section discusses a better way to use COM with *.NET/Link*.

Using an Interop Assembly (Early Binding)

The previous section described using COM objects in *.NET/Link* via their `IDispatch` interface, often called late binding or Automation. There are drawbacks to that technique, but luckily .NET supports calling COM objects via a more sophisticated and efficient technique called early binding. This is similar to how COM objects are used in C++, as method dispatch happens via the vtable interface, not `IDispatch`. To use early binding in .NET, you must first create or find a so-called interop assembly. An interop assembly is a special assembly that contains metadata that describes the types and methods in a COM type library. Once you have an interop assembly, it can be loaded and used like any other .NET assembly, and the COM types it describes look like native .NET types to clients.

Interop assemblies can be created with a tool called `tlbimp.exe` ("type library importer"), which is included with the .NET Framework SDK, and also with Visual Studio .NET. You will find ample documentation on how to run `tlbimp`, and you will see an example later. It is also possible to create an interop assembly programmatically, and *.NET/Link* provides the `LoadCOMTypeLibrary` function for this purpose. `LoadCOMTypeLibrary` takes a COM type library, creates an interop assembly from it, and loads this assembly into *.NET/Link*. You can think of it as analogous to `LoadNETAssembly`, except that it takes a path to a COM type library instead.

`LoadCOMTypeLibrary[`*typeLibPath*`]`	create an interop assembly from the given type library and load it

Loading COM type libraries.

Earlier you used the COM-based Microsoft Speech API via Automation. The preferred method is to load the type library to allow early binding.

```
In[7]:= speechAsm = LoadCOMTypeLibrary[
          "C:\\Program Files\\Common Files\\Microsoft Shared\\Speech\\sapi.dll"]
Out[7]= NETAssembly[interop.SpeechLib, 25]
```

Interop assemblies created by `LoadCOMTypeLibrary` are created with default rules for naming of namespaces and types, and it is useful to use `NETTypeInfo` on the assembly to see what types are available. For each coclass in the COM type library, a class will be created in the interop assembly with the name of the coclass with the word "Class" appended. You saw earlier that there was a coclass called `SpVoice`, so you expect to find a .NET class called `SpVoiceClass` in the interop assembly. There are a lot of types in this assembly; this shows just the classes.

```
In[8]:= NETTypeInfo[speechAsm, "Classes"]

Assembly: interop.SpeechLib
Full Name: interop.SpeechLib, Version=5.0.0.0
Location:

• Classes
class SpeechLib._ISpeechRecoContextEvents_SinkHelper
class SpeechLib._ISpeechVoiceEvents_SinkHelper
class SpeechLib.SpAudioFormatClass
class SpeechLib.SpCompressedLexiconClass
class SpeechLib.SpCustomStreamClass
class SpeechLib.SpeechConstants
class SpeechLib.SpeechStringConstants
class SpeechLib.SpFileStreamClass
class SpeechLib.SpInProcRecoContextClass
class SpeechLib.SpInprocRecognizerClass
class SpeechLib.SpLexiconClass
class SpeechLib.SpMemoryStreamClass
class SpeechLib.SpMMAudioEnumClass
class SpeechLib.SpMMAudioInClass
class SpeechLib.SpMMAudioOutClass
class SpeechLib.SpNotifyTranslatorClass
class SpeechLib.SpNullPhoneConverterClass
class SpeechLib.SpObjectTokenCategoryClass
class SpeechLib.SpObjectTokenClass
class SpeechLib.SpPhoneConverterClass
class SpeechLib.SpPhraseInfoBuilderClass
class SpeechLib.SpRecPlayAudioClass
class SpeechLib.SpResourceManagerClass
class SpeechLib.SpSharedRecoContextClass
class SpeechLib.SpSharedRecognizerClass
class SpeechLib.SpStreamClass
class SpeechLib.SpStreamFormatConverterClass
class SpeechLib.SpTextSelectionInformationClass
class SpeechLib.SpUnCompressedLexiconClass
class SpeechLib.SpVoiceClass
class SpeechLib.SpWaveFormatExClass
```

You will find NETTypeInfo very useful when exploring an interop assembly. When creating a COM object earlier using Automation, you used the CreateCOMObject function. Now that you have .NET classes that represent the COM coclasses, you can call NETNew instead.

```
In[9]:= voice = NETNew["SpeechLib.SpVoiceClass"]
Out[9]= «NETObject[SpeechLib.SpVoiceClass] »
```

You use this object just like any other .NET object. Here is the Speak() method. For some reason, the optional nature of the second argument is not preserved in the interop assembly, so you have to call Speak() with two arguments.

```
In[10]:= voice@Speak["Using COM objects is easier with an interop assembly", 1]
Out[10]= 1
```

When using Automation, you saw that the second argument is a COM enumeration called SpeechVoiceSpeakFlags. When using Automation, however, there is no way to access an enumeration, so you had to pass an integer value. But with an interop assembly there is a .NET enumeration you can use to make your code more readable.

```
In[11]:= LoadNETType["SpeechLib.SpeechVoiceSpeakFlags"];
         voice@Speak["Using COM objects is easier with an interop assembly",
            SpeechVoiceSpeakFlags`SVSFlagsAsync];
```

One great advantage of having an interop assembly is that you can use NETTypeInfo to get information about the methods and properties supported by objects. This shows the properties of the voice object.

```
In[12]:= NETTypeInfo[voice, "Properties"]
```

• ***Properties***

```
virtual SpeechLib.SpeechVoiceEvents AlertBoundary
virtual bool AllowAudioOutputFormatChangesOnNextSet
virtual SpeechLib.SpObjectToken AudioOutput
virtual SpeechLib.ISpeechBaseStream AudioOutputStream
virtual SpeechLib.SpeechVoiceEvents EventInterests
virtual SpeechLib.SpeechVoicePriority Priority
virtual int Rate
virtual SpeechLib.ISpeechVoiceStatus Status  [read only]
virtual int SynchronousSpeakTimeout
virtual SpeechLib.SpObjectToken Voice
virtual int Volume
```

Note that once an interop assembly is loaded for a type library, if you call CreateCOMObject with the name of a COM coclass, you get back a native .NET object of the class that corresponds to the COM coclass, not a raw RCW as before. This means that CreateCOMObject and NETNew become equivalent ways of creating an instance of a COM coclass.

```
In[13]:= voice2 = CreateCOMObject["Sapi.SpVoice"]
Out[13]= «NETObject[SpeechLib.SpVoiceClass] »
```

option name	*default value*	
SaveAssemblyAs	None	a full pathname to the assembly file you want created
SafeArrayAsArray	False	whether to marshal SAFEARRAY types as System.Array

Options to LoadCOMTypeLibrary.

LoadCOMTypeLibrary takes two options that control the assembly-creation process. The first is SaveAssemblyAs, which allows you to specify a filename into which you want the created assembly saved. LoadCOMTypeLibrary can take a while to execute, so it is useful to save the assembly in a file and load it using LoadNETAssembly in the future. This makes LoadCOMTypeLibrary the programmatic equivalent to running the tlbimp.exe tool, in that it can write out the generated assembly. The second option to LoadCOMTypeLibrary is SafeArrayAsArray, which specifies whether to import all COM SAFEARRAY's as the System.Array class rather than a typed, single dimensional managed array. The default is False. See the .NET Framework documentation for the System.Runtime.InteropServices.TypeLibImporterFlags enumeration for more details on this advanced option. If you need more control over the generated assembly, use the tlbimp.exe tool as described in the next section.

In the earlier discussion of using COM objects via Automation, the role of the RCW was described. You saw that the class name of a "raw" RCW is System.__ComObject. It is important to remember that all COM objects are represented in .NET as RCWs, even when using an interop assembly. You can see below that the SpVoiceClass derives from __ComObject.

```
In[14]:= NETTypeInfo[voice, "Type"]
```

- ***Type***

```
class SpeechLib.SpVoiceClass
Inheritance:
      System.Object
         System.MarshalByRefObject
            System.__ComObject
               SpeechLib.SpVoiceClass
Interfaces Implemented: SpeechLib.ISpeechVoice, SpeechLib.SpVoice, SpeechLib._ISpeechVoiceEvents_
Assembly-Qualified Name: SpeechLib.SpVoiceClass, interop.SpeechLib, Version=5.0.0.0
Assembly Location: Dynamically generated
```

Here is another way to prove the inheritance relationship.

```
In[15]:= InstanceOf[voice, "System.__ComObject"]
Out[15]= True
```

Using tlbimp.exe to Create an Interop Assembly

As mentioned earlier, the .NET Framework SDK includes a tool called `tlbimp.exe` (type library importer) that creates an interop assembly from a COM type library. You can use this tool to create an interop assembly and load it using `LoadNETAssembly`, instead of using the `LoadCOMTypeLibrary` function. The `tlbimp` program has many options to control how the assembly is generated, so if you need this level of control, you will definitely want to run it manually. The .NET Framework SDK documentation describes how to use `tlbimp` in detail, but here is an example of how it could be used to create an interop assembly for the `SpeechLib` type library.

```
tlbimp "C:\Program Files\Common Files\Microsoft Shared\Speech\sapi.dll"
/out:c:\interop.SpeechLib.dll
```

Once the assembly has been created, load it like any other assembly.

```
LoadNETAssembly["c:\\interop.SpeechLib.dll"];
```

Primary Interop Assemblies

A primary interop assembly (PIA) is a special interop assembly that is signed by the vendor and given a strong name so that it can be placed into the global assembly cache (GAC). The idea behind a PIA is that a vendor of a COM type library will create an "official" interop assembly that represents the ideal interface to their type library. The .NET runtime recognizes PIAs as being special "blessed" assemblies, and it will load one automatically when you call `CreateCOMObject` on a coclass for which the associated PIA has been installed. If you are using a COM library, you should check to see if the vendor has created a PIA for it. If so, you should install it.

A good example of PIAs is the set created by Microsoft to accommodate the Office XP suite, which exposes a rich object model for Automation. Anyone trying to control an Office XP component like Word or Excel from *Mathematica* should obtain the Office PIAs from http://msdn.microsoft.com/library/default.asp?url=/downloads/list/office.asp. These PIAs are installed by default with Office 2003, but you can install them for use with Office XP and perhaps earlier versions of Office as well.

The ExcelPieChart.nb example file shows how to call Excel from *Mathematica* using *.NET/Link*. That example will work whether or not you have the Office PIAs installed, but if the PIAs are present, you have the advantages of working with strongly typed .NET objects instead of raw RCW objects. If the PIAs are installed on your machine, and `CreateCOMObject` is called to start an instance of Excel, it automatically returns a .NET type from the Excel interop assembly, not a raw RCW like `<<NETObject[COMInterface[Excel._Application]]>>`.

```
In[16]:= excel = CreateCOMObject["Excel.Application"]
Out[16]= «NETObject[Microsoft.Office.Interop.Excel.ApplicationClass] »
```

Releasing COM Resources

One of the roles of the RCW object in .NET is to manage the life cycle of the COM object it wraps. The COM object is destroyed when the RCW object is freed by the .NET garbage collector. Often, this level of control over the lifetime of the COM object is fine. Remember, though, that the .NET garbage collector may run infrequently, and generally only when the .NET memory space (the "managed" heap) fills up. RCW objects have a small footprint in the managed heap, but they may hold onto very large objects (such as an instance of Excel) in the unmanaged COM world. In some COM usage scenarios, the .NET garbage collector may not run even though there are a large number of unfreed and unused RCW objects that are keeping alive a large amount of memory and other resources in the COM world. For this reason, it is important to have a function that forces the COM resources held by an object to be released. That function is `ReleaseCOMObject`.

`ReleaseCOMObject[`*obj*`]`	releases the COM resource owned by the given COM object

Releasing COM resources.

Every COM object in .NET is represented by a single unique RCW. If you acquire a reference to the same COM object through two different means, you will get the same RCW each time. This sole RCW keeps a reference count on the COM object. This reference count is internal to COM and should not be confused with the reference count of a .NET object. Calling `ReleaseCOMObject` does not actually force the immediate release of COM resources—it just decrements this internal COM reference count. Often this count will be just one, `ReleaseCOMObject` will cause it to go to zero, and then the resources will be freed. `ReleaseCOMObject` returns the new reference count, so you can see if it has gone to zero yet.

Here is an example of acquiring multiple references to the same COM object. The following line launches a new instance of Excel. It will not become visible, but you can see it in the Task Manager listing of processes.

```
In[17]:= excel1 = CreateCOMObject[“Excel.Application“]
Out[17]= «NETObject[Microsoft.Office.Interop.Excel.ApplicationClass] »
```

Now acquire two more references to that same instance of Excel. The `GetActiveCOMObject` function is like `CreateCOMObject` except that instead of creating a new object, it acquires an already-active one. It is analogous to the `GetActiveObject()` function in the COM API and Visual Basic 6. The `Pause` is necessary here because COM apparently needs to catch its breath a bit between calls to `GetActiveObject()`.

```
In[18]:= excel2 = GetActiveCOMObject[“Excel.Application“];
         Pause[1];
         excel3 = GetActiveCOMObject[“Excel.Application“];
```

If you called `ReleaseCOMObject[excel1]` now, you probably would not want Excel to quit, because there are other outstanding references to Excel via the `excel2` and `excel3` objects. Note how calling `ReleaseCOMObject` decrements the COM reference count on the instance of Excel until it finally goes to zero. Only when it reaches zero will the Excel process quit.

```
In[21]:= {ReleaseCOMObject[excel1], ReleaseCOMObject[excel2], ReleaseCOMObject[excel3]}
Out[21]= {2, 1, 0}
```

You have now seen how to use `ReleaseCOMObject` to force the timely release of COM resources. It is never strictly necessary to do this, as the .NET garbage collector will get around to it eventually, but often it is not timely enough. An alternative to using `ReleaseCOMObject` is simply to make sure that you use `NETBlock` or `ReleaseNETObject` to allow the .NET objects you create to be released. Then you can manually force the .NET garbage collector to run. This is especially convenient if you have code that creates a lot of COM objects. Rather than keeping track of all of them and calling `ReleaseCOMObject` on each one, it is easier to use `NETBlock` to ensure that the objects all have .NET reference counts of zero when you are done with them, and then call the garbage collector. Here is an outline of what that looks like.

```
SomeFunction[args__] :=
    Module[{result},
        NETBlock[
            ... some code here that creates and manipulates COM objects
            result = ...
        ];
        LoadNETType[“System.GC“];
        GC`Collect[];
        result
    ]
```

Casting COM Objects

The CastNETObject function is described in "Casting". This function is rarely used for normal .NET objects, but it has special duties with respect to COM objects. The following example requires that you have the Microsoft Office XP primary interop assemblies installed on your machine. The next line will create a new instance of the Excel application, although it will not be visible.

```
In[22]:= excel = NETNew["Microsoft.Office.Interop.Excel.ApplicationClass"]
Out[22]= «NETObject[Microsoft.Office.Interop.Excel.ApplicationClass] »
```

Now create a new workbook.

```
In[23]:= excel@Workbooks@Add[];
```

The ApplicationClass class has a property called ActiveSheet that will return the worksheet in the workbook you just created.

```
In[24]:= activeSheet = excel@ActiveSheet
Out[24]= «NETObject[COMInterface[Excel._Worksheet]] »
```

Notice that the result from ActiveSheet is a raw RCW object, not a strongly typed .NET object (recall that raw RCW objects are of the class System.__ComObject, and *.NET/Link* tries to format them with the name of the default COM interface that they support). There is a Microsoft.Office.Interop.Excel.WorksheetClass class in the Excel interop assembly that represents worksheets, so why was the result of ActiveSheet not an instance of that class? To see the answer, look at the declaration of the ActiveSheet property.

```
In[25]:= NETTypeInfo[excel, "Properties", "ActiveSheet"]
```

• ***Properties*** (matching string pattern ActiveSheet)
virtual object **ActiveSheet** [read only]

Note that this property is typed to return only object, not WorksheetClass. That is because the active sheet could be a chart or a worksheet, and these are different classes. In an interop assembly, methods or properties that are typed to return object will return a raw RCW. This reminds you that COM objects in .NET are different animals than normal .NET objects. When any object is marshaled from COM into .NET, it always arrives as a raw RCW. With the help of

type information from an interop assembly, the .NET runtime can cast a raw RCW to a specific managed type. For example, if a method is typed to return class `X`, and the method returns a COM object (as an RCW), .NET casts the RCW to the type `X` before returning it from the method. If a method is typed to return only `object`, like the `ActiveSheet` property, then there is no type information that .NET can use to cast the object, so you end up with a raw RCW.

You can use the object returned by `ActiveSheet`, but you will be calling it via late binding because it has no type information. If you want to make a strongly typed object that you can call via early binding, then you do exactly what you would do in C# or Visual Basic .NET—you cast the object to the desired type. Once you have cast to the desired managed type, you once again have all the advantages that come from using an interop assembly instead of using late binding. In this example, you know that the active sheet is a worksheet, so you can cast it to `WorksheetClass`.

```
In[26]:= activeSheet = CastNETObject[excel@ActiveSheet,
           "Microsoft.Office.Interop.Excel.WorksheetClass"]
Out[262]= «NETObject[Microsoft.Office.Interop.Excel.WorksheetClass] »
```

If this sounds confusing, remember that this is exactly what is done in C# or Visual Basic .NET. Here is what it would look like in C#.

```
ApplicationClass excel = new ApplicationClass();
excel.Workbooks.Add();
WorksheetClass activeSheet = (WorksheetClass) excel.ActiveSheet;
```

If the COM object cannot be cast to the specified managed type, `CastNETObject` will issue a message and return `$Failed`.

In "Casting", it was stated that there is never a need to downcast in *.NET/Link*, because objects always have their true runtime type—there is never a type lower in the inheritance hierarchy to downcast to. This rule does not apply for casting COM objects, because in some sense the runtime type of all COM objects is just the raw RCW class `__ComObject`. In the presence of type information, the .NET runtime can downcast automatically to more derived managed types. When a property or method is typed to return only `object`, you can downcast the object yourself, provided you know the correct type.

Handling COM Events

The .NET runtime knows how to map COM events to .NET events, which means that responding to events fired by COM objects is just like responding to events fired by .NET objects. You must use an "interop assembly" to handle COM events in *Mathematica* code—you cannot use late binding.

The following example shows how to handle COM events fired by Internet Explorer. Internet Explorer supports a rich object model for the content of an HTML window. This is called the document object model, and it is exposed to clients via the `mshtml` COM library. Microsoft bundles with the .NET Framework a primary interop assembly for `mshtml`, to make it easier to use from .NET programs. Presumably, there will eventually be a native .NET version of Internet Explorer and `mshtml`, but for now, as with many other COM-based Microsoft technologies, you use them from .NET via an interop assembly. The example developed below will display a web page in an Internet Explorer window, and as the user moves the mouse over elements in the page, the elements will have their font size changed randomly.

The `mshtml` COM component and its associated primary interop assembly only manage the content of an Internet Explorer window. The Internet Explorer application is a separate COM object that must be created in the usual way for raw COM objects.

```
In[27]:= ie = CreateCOMObject["InternetExplorer.Application"]
Out[27]= «NETObject[COMInterface[SHDocVw.IWebBrowser2]] »
```

The fact that this object is formatted with `COMInterface[...]` indicates that it is a raw RCW and you can only interact with it via Automation. This is fine, as only a few simple properties are needed. To get documentation for COM objects, look them up in the MSDN Library, either online or in the Visual Studio help system.

First, navigate to a URL.

```
In[28]:= ie@navigate["www.wolfram.com/solutions/mathlink"]
```

Now make the browser window visible and loop until the page has completely loaded.

```
In[29]:= ie@Visible = True;
         While[ie@Busy, Pause[1]];
```

You want to interact not with the Internet Explorer application but with the document object that it contains, so acquire that object.

```
In[31]:= doc = ie@Document
Out[31]= «NETObject[mshtml.HTMLDocumentClass] »
```

Note that this object is a strongly typed .NET object, not a raw RCW. Because there is a primary interop assembly for the document object model, whenever such a COM object is imported into .NET it can be automatically wrapped in the .NET class that is mapped to the document COM coclass. You are now in the cozy world of using strongly typed objects from an interop assembly.

COM events fired by an object are mapped to .NET events in the interop assembly. You will see a .NET event member for each method in every `[source]` COM interface that a .NET class implements. You can use `NETTypeInfo` to see the events that are fired by the `HTMLDocumentClass` class. There are quite a few for this class, so this just shows the one you need.

```
In[32]:= NETTypeInfo[doc, "Events", "*onmouseover*"]
```

• ***Events*** (matching string pattern *onmouseover*)
virtual event mshtml.HTMLDocumentEvents2_onmouseoverEventHandler **HTMLDocumentEvents2_Event _onmouseover**
virtual event mshtml.HTMLDocumentEvents_onmouseoverEventHandler **HTMLDocumentEvents_Event _onmouseover**

There are two `onmouseover` events inherited from two different interfaces. You want to use the one named `HTMLDocumentEvents2_Event_onmouseover`, which supplies an argument. There is no separate documentation for the `mshtml` primary interop assembly, so you have to use the documentation for the COM version and make the appropriate mental translations, which are usually quite straightforward. You can find full documentation on the `mshtml` component at http://msdn.microsoft.com/library/default.asp?url=/workshop/browser/mshtml/reference/reference.asp.

"Handling Events" shows you how to use the `AddEventHandler` function to assign *Mathematica* functions to be called when events are fired in .NET. You use the same function for COM events, as they are made to look just like .NET events by the interop assembly. Note that as always you must change _ characters to `U` when using .NET names as symbols.

```
In[33]:= AddEventHandler[doc@HTMLDocumentEvents2UEventUonmouseover, onMouseOver];
```

Now define the onMouseOver function. The NETTypeInfo call above told you that the argument to the event is of interface type mshtml.IHTMLEventObj. This is the managed equivalent of the IHTMLEventObj COM interface, and from the documentation for that interface you can cook up the following simple function that wraps every element in a <FONT> element that specifies a random font size.

```
In[34]:= onMouseOver[evt_] :=
          Module[{element},
           element = evt@srcElement;
           element@innerHTML = "<FONT size='" <>
             ToString[Random[Integer, {1, 7}]] <> "'>" <> element@innerHTML <> "</FONT>";
          ]
```

Before you actually try moving the mouse over the web page, there is one more detail that must be handled. "Manually Sharing the Kernel and Front End with .NET" shows how to use the ShareKernel function to put the kernel into a state where it was receptive to calls arriving from .NET. ShareKernel is not often needed in *.NET/Link*, because you can usually enter and leave the sharing state automatically using the DoNETModeless function. DoNETModeless requires a top-level window as its argument, however. Here you have no such window, just an instance of Internet Explorer. Therefore you have to use ShareKernel to manually enter the sharing state. As always, you save the result from ShareKernel to pass into UnshareKernel later.

```
In[35]:= tok = ShareKernel[NETLink[]];
```

Now bring the Internet Explorer window to the foreground and move the mouse over it (without pressing the mouse button). It might take a second or two for the first font effect to occur.

Make sure you clean things up when you are done.

```
(sharing) In[36]:=
          ie@Quit[];
          UnshareKernel[tok];
```

This is a frivolous example, but you can imagine many more useful and sophisticated ways to interact with a browser window using *Mathematica*.

Displaying ActiveX Controls

Many COM objects have a visual representation, such as a toolbar, grid box, or other window element. Although the term "ActiveX control" is really just a synonym for "COM object," visual COM objects are usually referred to as ActiveX controls. If you want to display an ActiveX control in a .NET program, you must create a special type of interop assembly for the control. This assembly is created with the `aximp.exe` tool (ActiveX importer), which is similar to the `tlbimp` tool discussed earlier, except that it is specific to ActiveX controls that must be hosted within a .NET window. If a control is to be hosted within a .NET window, it needs a special wrapper class that inherits from `System.Windows.Forms.Control`. The `aximp` tool creates this wrapper class.

There is full documentation for `aximp` in the .NET Framework SDK, but here is a simple example. Say you want to use the Microsoft Calendar Control in a .NET window (you probably have this control installed on your machine). Assume that the type library for this control is in the file `d:\OfficeXP\Office10\mscal.ocx` on your machine. (One way to get information like this is by looking up the control using the OLE View tool that is bundled with Visual Studio.) The following command line runs `aximp` (of course, `aximp.exe` has to be on your `PATH` for this to work as written).

```
c:\> aximp d:\OfficeXp\Office10\mscal.ocx
```

The above invocation of `aximp` creates two assemblies in the current directory: `MSACAL.dll` and `AxMSACAL.dll` (the "MSACAL" comes from the library `MSACAL` statement in the control's type library definition). The `MSACAL.dll` assembly contains managed types for the `ICalendar` interface and `CalendarClass` class, along with a few others. This is the same interop assembly that would be created by using the `tlbimp` tool. The second assembly, `AxMSACAL.dll`, contains the special wrapper class that makes the Calendar control into a .NET control that can be hosted in a .NET window. This wrapper class is called `AxCalendar`, named according to the convention "Ax" followed by the name of the COM coclass (Calendar). The `ildasm.exe` tool (intermediate language disassembler) is very useful for examining the created assemblies. The `ildasm.exe` program is bundled with the .NET Framework SDK and resides in the same directory as `aximp.exe` and `tlbimp.exe`.

When you load the `AxMSACAL.dll` assembly, the `MSACAL.dll` assembly will be loaded as well, because `AxMSACAL.dll` is dependent on it and it resides in the same directory.

```
In[1]:= LoadNETAssembly["c:\\AxMSACAL.dll"]
Out[1]= NETAssembly[AxMSACAL, 1]
```

Now create an instance of the `AxCalendar` wrapper class. This class has all the methods and properties of the Calendar COM object, and it also inherits from the `System.Windows.Forms.Control` class, so it can be used like any other .NET control.

```
In[2]:= cal = NETNew["AxMSACAL.AxCalendar"]
Out[2]= «NETObject[AxMSACAL.AxCalendar] »
```

Now create a .NET form to host the control and display it. The `DoNETModal` function at the end will display the form and return the calendar's `Value` property when the form is closed. This property will hold the date the user selected.

```
In[3]:= form = NETNew["System.Windows.Forms.Form"];
        cal@Parent = form;
        LoadNETType["System.Windows.Forms.DockStyle"];
        cal@Dock = DockStyle`Fill;
        calDate = DoNETModal[form, cal@Value]
Out[7]= «NETObject[System.DateTime] »
```

The .NET runtime has conveniently mapped the result of the `Value` property to a `DateTime` object. This is easily manipulated to get the selected date as a string.

```
In[8]:= calDate@ToString[]
Out[8]= 1/1/2007 12:00:00 AM
```

Now clean up by releasing the calendar object. Note that you do not call `ReleaseCOMObject` on the `AxCalendar` object, because it is not a COM object—it is a pure .NET class that merely holds a reference to a COM object. The actual COM object is of class `MSACAL.CalendarClass`, but you never directly create an instance of that class, just the `AxCalendar` wrapper.

```
In[9]:= ReleaseNETObject[cal]
```

Example Files

The following example program included with *.NET/Link* demonstrates calling COM components from *Mathematica.*

ExcelPieChart.nb

Calling *Mathematica* from .NET

Introduction

"Calling .NET from *Mathematica*" describes using *.NET/Link* to allow you to call from *Mathematica* into .NET, thereby extending the *Mathematica* environment to include the functionality in all existing and future .NET classes. This tutorial shows you how to use *.NET/Link* in the opposite direction, as a means to write .NET programs that use the *Mathematica* kernel as a computational engine.

.NET/Link uses *MathLink*, Wolfram Research's protocol for sending data and commands between programs. Many of the concepts and techniques in *.NET/Link* programming are the same as those for programming with the *MathLink* C-language API. The *.NET/Link* documentation is not intended to be an encyclopedic compendium of everything you need to know to write .NET programs that use *MathLink*. Programmers may have to rely a little on the general documentation of *MathLink* programming. The Tutorial is divided into two major sections along the same lines as this documentation. You will want to read the second part. Many of the functions *.NET/Link* provides have C-language counterparts that are identical or nearly so.

You should at least skim the tutorial "Calling .NET from *Mathematica*" at some point. Your .NET "front end" can use the same techniques for calling .NET methods from *Mathematica* code and passing .NET objects as arguments that programmers use when running the kernel from the notebook front end. This allows you to have a very high-level interface between .NET and *Mathematica*. When you are writing *MathLink* programs in C, you have to think about passing and returning simple things like strings and integers. With *.NET/Link* you can pass .NET objects back and forth between .NET and *Mathematica*.

The sections below merely provide an overview of topics in *.NET/Link* programming. The main class-by-class reference for *.NET/Link* is the API documentation. You should also look at the example programs.

When you are reading this text, or programming in .NET or *Mathematica*, remember that the entire source code for *.NET/Link* is provided. If you want to see how anything works (or why it doesn't), you can always consult the source code directly.

What Is *MathLink*?

MathLink is a platform-independent protocol for communicating between programs. In more concrete terms, it is a means to send and receive *Mathematica* expressions. *MathLink* is the means by which the notebook front end and kernel communicate with each other. It is also used by a large number of commercial and freeware applications and utilities that link *Mathematica* and other programs or languages. It is implemented as a library of C-language functions. *.NET/Link* brings the capabilities of *MathLink* into .NET in a way that is simpler to use and much more powerful than the raw C-level API.

Overview of the Main *.NET/Link* Interfaces and Classes

Introduction

The *.NET/Link* class library is written in an object-oriented style intended to maximize its extensibility in the future without requiring users' code to change. This requires a clean separation between interface and implementation. This is accomplished by exposing the main link functionality through interfaces, not classes. The names of the concrete classes that implement these interfaces will hardly be mentioned because programmers do not need to know or care what they are. Rather, you will use objects that belong to one of the interface types. You do not need to know what the actual classes are because you will never create an instance directly; instead, you use a "factory method" to create an instance of a link class. This will become clear further on.

This section gives a brief overview of the main interfaces and classes. They will be discussed in more detail later. In addition, there is full documentation for the class library in the *.NET/Link* API documentation. Most of these classes and interfaces are in the `Wolfram.NETLink` namespace; others are in `Wolfram.NETLink.UI`.

IMathLink and IKernelLink

The two most important link interfaces you need to know about are `IMathLink` and `IKernelLink`. The `IMathLink` interface is essentially a port of the *MathLink* C API into .NET. Most of the method names will be familiar to experienced *MathLink* programmers. `IKernelLink` extends `IMathLink` and adds some important high-level convenience methods that are only meaningful if the other side of the link is a *Mathematica* kernel (for example, the method `WaitForAnswer()`, which assumes the other side of the link will respond with a defined series of packets).

The basic idea is that the `IMathLink` interface encompasses all the operations that can be performed on a link without making any assumptions about what program is on the other side of the link. `IKernelLink` adds the assumption that the other side is a *Mathematica* kernel.

`IKernelLink` is the most important interface, as most programmers will work exclusively with `IKernelLink`. Of course, since `IKernelLink` extends `IMathLink`, many of the methods you will use on your `IKernelLink` objects are declared and documented in the `IMathLink` interface.

The most important class that implements `IMathLink` is `NativeLink`, so named because it calls directly into Wolfram Research's *MathLink* library. In the future, other classes could be added that do not rely on native methods—for example, one that uses .NET remoting to communicate across a network. As discussed above, programmers do not need to be concerned about what these classes are, because they will never type a link class name in their code.

MathLinkFactory

`MathLinkFactory` is the class that you use to create link objects. It contains the static methods `CreateMathLink()`, `CreateKernelLink()`, and `CreateLoopbackLink()`, which take various argument sequences. These are the equivalents of calling `MLOpen` in a C program.

MathLinkException

`MathLinkException` is the exception class that is thrown by many of the methods in `IMathLink` and `IKernelLink`. The *.NET/Link* API uses exceptions to indicate errors, rather than function return values like the *MathLink* C API. In C, you write code that checks the return values like this:

```
// C code
if (!MLPutInteger(link, 42)) {
    // was error; print message and clean up.
}
```

In *.NET/Link*, you can wrap *MathLink* calls in a `try` block and catch `MathLinkException`.

Expr

The `Expr` class provides a direct representation of *Mathematica* expressions in .NET. `Expr` has a number of methods that provide information about the structure of the expression and that let you extract components. These methods have names and behaviors that will be familiar to *Mathematica* programmers—for example, `Length()`, `Part()`, `NumberQ()`, `VectorQ()`, `Take()`, `Delete()`, and so on. When reading from a link, instead of using the low-level `IMathLink` interface methods for discovering the structure and properties of the incoming expression, you can just read an entire expression from the link using `GetExpr()`, and then use `Expr` methods to inspect it or decompose it. For writing to a link, `Expr` objects can be used as arguments to some of the most important `IKernelLink` methods.

MathKernel

`MathKernel` is a non-visual component that provides a very high-level interface for interacting with *Mathematica*. It is especially intended for use in visual programming environments, as it is highly configurable via properties. For many types of .NET programs that use *Mathematica* for computations, the `IKernelLink` interface provides ideal functionality. For some types of programs, however, programmers might find the `MathKernel` object even easier to use. This is especially true for programs that want to capture not just the result of a computation, but also messages, `Print` output, or graphics generated as side effects of the computation.

MathPictureBox

The `MathPictureBox` class provides an easy way to display *Mathematica* graphics and typeset expressions. This class is often used from *Mathematica* code, but it is just as useful in .NET programs.

Sample Program

The *.NET/Link* distribution includes a SimpleLink sample program that demonstrates simple techniques for launching *Mathematica* and performing computations. The code is available in both C# and Visual Basic .NET.

Building and Deploying Programs

The .NET/Link Assembly

The *.NET/Link* assembly file is `Wolfram.NETLink.dll`, and it is found in the *<Mathematica* `dir>\SystemFiles\Links\NETLink` directory. When you compile .NET programs that use *.NET/Link*, you will need to add a reference to this file. In previous versions of *.NET/Lin*k, this assembly was placed into the .NET global assembly cache (GAC) by the *Mathematica* installer. Starting with *.NET/Link* 1.2, however, the `Wolfram.NETLink.dll` assembly is no longer strong-named and therefore cannot be placed into the GAC. .NET programs that use *.NET/Link* will therefore need to have a copy of this assembly in their application directory (that is, right next to their `.exe` file). Alternatively, the `Wolfram.NETLink.dll` assembly can be located in a subdirectory of the application's directory, according to the standard rules for how .NET probes for assemblies that are not strong-named.

A consequence of the fact that `Wolfram.NETLink.dll` is not strong-named is that you can replace your application's copy with an updated version, and the updated version will be used without requiring the application to be recompiled. In contrast, strong-named assemblies in .NET are strictly versioned, so that when a program is compiled against a specific version of the assembly, the program can only run with that precise version. Of course, this strict versioning is touted as a benefit of .NET, and it is advantageous in certain circumstances, but for various technical reasons it is not desirable for `Wolfram.NETLink.dll` to be strong-named.

Compiling from the Command Line

The .NET Framework SDK includes command-line compilers for several .NET languages, including C# and Visual Basic .NET. You can use these free tools without purchasing Visual Studio .NET, and you might want to use them to build simple programs even if you do have Visual Studio .NET.

The command-line compilers rely on several DOS environment variables being set correctly, so the .NET Framework SDK comes with a batch file named `sdkvars.bat` that you can run in your DOS session to set these variables. If you own Visual Studio .NET and want to use any command-line tools, use the `Microsoft Visual Studio .NET/Visual Studio .NET Tools/Visual Studio .NET Command Prompt` item on the Windows `Start` menu to launch a DOS session with all the required settings.

It is convenient to place a copy of `Wolfram.NETLink.dll` into your build directory before you compile, as you would otherwise need to include the full path to this assembly on the compiler command line, and you will need a copy of it to be present in the program's directory anyway. The examples below assume that you have copied (not moved!) the `Wolfram.NETLink.dll` file from `<`*`Mathematica`* `dir>\SystemFiles\Links\NETLink` into the directory in which you are performing the build. Here is a sample command for a C# program:

```
csc /target:winexe /reference:Wolfram.NETLink.dll MyProgram.cs
```

Here is a comparable example for Visual Basic .NET:

```
vbc /target:winexe /reference:Wolfram.NETLink.dll MyProgram.vb
```

Either of the above commands will result in the creation of a `MyProgram.exe` file in the current directory. This program will need to have a copy of `Wolfram.NETLink.dll` alongside it to run.

Using Visual Studio .NET

Programs that use *.NET/Link* need to have a reference to the Wolfram.NETLink.dll assembly in their project settings. You add a reference to an assembly by selecting Add Reference from the Project menu. The *Mathematica* installer makes the necessary settings so that the Wolfram.NETLink.dll assembly shows up in the Add Reference dialog box. It will be listed as *.NET/Link* 1.3 in the .NET tab. If for some reason it does not show up there, you can simply use the Browse button to locate the file manually (it will be in the <*Mathematica* dir>\SystemFiles\Links\NETLink directory, or perhaps in another more convenient location if you put a copy elsewhere).

When Visual Studio .NET builds your program it will automatically place a copy of Wolfram.NETLink.dll in the output directory, alongside your .exe file. Your program will require a copy of Wolfram.NETLink.dll alongside it to run, so if you deploy your program to another location or distribute it to other users, you must keep the Wolfram.NETLink.dll file with it.

Deploying Programs

If you build a .NET program that uses *.NET/Link* and want to distribute it to others, you will need to include a copy of the Wolfram.NETLink.dll assembly alongside your application's .exe file. Alternatively, you can put Wolfram.NETLink.dll in a subdirectory of the application's directory, according to the standard rules for how .NET probes for assemblies that are not strong-named.