

BY

Yaseen Barkat Ali Kalhoro

Email

[Gh\\_yaseen2006@yahoo.com](mailto:Gh_yaseen2006@yahoo.com)

Contact # no

0321-3411022

used for a Class or a single Method. Only a reference to a class or a call of the method will request the CLR for FileIOPermission.

### **Table 12.2** Continued

#### **Membership Condition Description**

**Security • Chapter 12 565**

As the syntax already suggests, by using <> this code is not treated as ordinary code. In fact, as you compile the code to an assembly, these lines are extracted and placed in the metadata part of the assembly. This metadata is checked at different points, such as during the load of the assembly or when a method in the assembly is called. Using declarative security, you can demand, request, or even override permissions before the code is even executed. This gives you a powerful security tool during the development of the code and assemblies. However, this means that you must be aware of the kind of permissions you need to request and/or demand for your code.

The second method is *imperative security*, which becomes a part of your code and can make permission demands and overrides. It is not possible to request permissions using imperative security because that makes it unclear at what point a specific permission is needed and at what point it is no longer needed. That is why permission requests are related to identifiable units of code. You may want to use imperative security to check if the caller has a permission that is specific for a part of the code. For example, just before a conditional part of the code (this may even be triggered by the role-based security) wants to access a file or a Registry key, you want to check if the caller has this *FileIOPermission* or *RegistryPermission*.

The VB.NET syntax of the imperative security is code looks like this:

```
Dim PermissionObject as New Permission()  
PermissionObject.Demand()
```

Here is an example:

```
Dim CheckPermission as New FileIOPermission()  
CheckPermission.Demand()
```

The permission object is valid only for the scope on which it is declared, and it will be automatically discarded at the time the code returns to a higher scope. During this scope, imperative security demands and overrides overrule the permissions demanded with a declarative security statement.

Having discussed declarative and imperative security, it is time to take a look at how you can use this to request, demand, and override permissions.

## **Requesting Permissions**

Requesting permissions is the best way to create a secure application and prevent possible misuse of your code by malicious code. As mentioned before, based on the evidence an assembly hands over to the CLR, and then a permission set is

### **566 Chapter 12 • Security**

determined, using security policies. These security policies are constructed independently from the permissions an assembly needs. Of course, if you fully trust an assembly, you can grant it all the permissions it needs. An assembly can be granted

more permissions than it actually needs. Requesting permissions is not asking for more permissions than you are granted, based on the security profile, but refraining from granting permissions the code does not need. By now you have probably started to wonder what the use of requesting permissions is if the security policy decides what permissions are available to the assembly. The term *available* implies two issues:

\_ If an assembly requests more permissions than it is granted, based on the security policy, it will not be loaded and/or the code will not be executed. Instead, the CLR will throw an exception

\_ If an assembly requests less permissions, it protects itself from misuse of these additional permissions somewhere up or down the calling chain. Requesting permissions is a characteristic of proper .NET applications and demands from the developer a good understanding of the use of permissions related to the code he writes. Because you can only request permissions by using declarative security, you can first write and test the code and then add the permission requests later. This can make the development process easier, saving you the hassle of constantly having to consider permission requests for unfinished code.

There are three types of permission requests:

\_ **RequestMinimum** Defines the permissions the code absolutely needs to be able to run. If the *RequestMinimum* permission is not part of the granted permission set, the code is not allowed to run.

\_ **RequestOptional** Defines the permissions the code may not necessarily need to be able to run but may need in certain circumstances. If the *RequestOptional* permission is not part of the granted permission set, the code is still allowed to run, however, you need the code to be able to handle the situation in which the permission is needed but not granted, thus handling exceptions.

\_ **RequestRefuse** Defines the permissions the code will never need and which should not be granted to the assembly. By refraining from certain permissions you prevent malicious code or unstable code from misusing these permissions.

#### Security • Chapter 12 567

After the code is completed and you compile assemblies, you should get in the practice of making a minimum, optional, or refuse request for *every* permission (as listed in Table 12.3), based on the permissions needed by the code.

Eventually you can make it more specific to relate it to classes or members.

Besides the fact that you can create secure assemblies, it is also a good way of documenting the permissions related to your code.

**Table 12.3** The Default Permission Classes Derived from the *CodeAccessPermission* Class

#### Permission

##### Permission Class Type Description

*DirectoryServicesPermission* Resource Controls access to the *System.DirectoryServices* classes

*DnsPermission* Resource Controls access to the DNS servers on the network

*EnvironmentPermission* Resource Controls access to the user environment variables  
*EventLogPermission* Resource Controls access to the event log services  
*FileDialogPermission* Resource Controls access to files that are selected through an Open File... dialog  
*FileIOPermission* Resource Controls access to files and directories  
*IsolatedStorageFilePermission* Resource Controls access to a private virtual file system related to the identity of the application or component  
*MessageQueuePermission* Resource Controls access to the MSMQ services  
*OleDbPermission* Resource Controls access to the OLE DB data provider and the data sources associated with it  
*PerformanceCounterPermission* Resource Controls access to the performance counters of Windows 2000 (or NT)  
*PrintingPermission* Resource Controls access to printers  
*ReflectionPermission* Resource Controls access to metadata types

**Continued**

**568 Chapter 12 • Security**

*RegistryPermission* Resource Controls access to the registry  
*SecurityPermission* Resource Controls access to *SecurityPermission* such as Assert, Skip Verification, and Call Unmanaged Code  
*ServiceControllerPermission* Resource Controls access to services on the system  
*SocketPermission* Resource Controls access to socket that are needed to set up or accept a network connection  
*SqlClientPermission* Resource Controls access to SQL server databases  
*UIPermission* Resource Controls access to UI functionality, such as Clipboard  
*WebPermission* Resource Controls access to an Internetrelated resource  
*PublisherIdentityPermission* Identity Permission is granted if the evidence publisher is provided by the caller  
*SiteIdentityPermission* Identity Permission is granted if the evidence site is provided by the caller  
*StrongNameIdentityPermission* Identity Permission is granted if the evidence strong name is

provided by the caller

*UrlIdentityPermission* Identity Permission is granted if the evidence URL is provided by the caller

*ZoneIdentityPermission* Identity Permission is granted if the evidence zone is provided by the caller

Now let's look at some examples of the different types of requests:

```
<assembly: SecurityPermissionAttribute(SecurityAction.RequestMinimum, _  
Flags := SecurityPermissionFlag.ControlPrincipal)>
```

### Table 12.3 Continued

#### Permission

#### Permission Class Type Description

Security • Chapter 12 569

In order for this assembly to run, it needs at least the permission to be able to manipulate the principal object. This is a permission you would give only to an assembly that you trust.

```
<assembly: SecurityPermissionAttribute(SecurityAction.RequestMinimum, _  
ControlEvidence := True)>
```

In order for this assembly to run, it needs at least the permission to be able to provide additional evidence and modify the evidence as provided by the CLR.

This is a powerful permission you would give only to fully trusted assemblies.

```
<FileIOPermissionAttribute(SecurityAction.RequestOptional, _  
Write := "C:\Test\*.cfg")> Public Class ClassAct
```

The *ClassAct* class requests the optional permission to be able to write to files in the C:\Test directory with the extension .cfg. If the security policy permits *FileIOPermission*, this restricted request is given. If the *FileIOPermission* is not granted, then any subsequent write to a CFG file in C:\Test will fail.

```
<assembly: FileIOPermission(SecurityAction.RequestRefuse, Unrestricted  
:= True)>
```

The assembly refuses the *FileIOPermission*, even if the security policy grants this permission. If you used this request in combination with the previous example, and the security policy grants *FileIOPermission*, only *ClassAct* will get this restricted *FileIOPermission*, and the rest of the code in the assembly will not have any *FileIOPermission*.

```
<assembly: FileIOPermission(SecurityAction.RequestRefuse, _  
All := "C:\Winnt\System32\*.*)">
```

The assembly refuses only *FileIOPermission* to the access of files in the C:\Winnt\System32 directory. If the security policy grants this permission, the assembly can access all files, except for the one in the stated directory.

Instead of making requests for every code access permission, you can also request one of the following named permission sets: *Nothing*, *Execution*, *Internet*, *LocalIntranet*, *SkipVerification*, and *FullTrust*. You can do this by issuing the following request:

```
<assembly: PermissionSetAttribute(SecurityAction.RequestMinimum, _  
Name := NamedPermissionSet)>
```

#### 570 Chapter 12 • Security

Another way of requesting more code access permissions in one statement is by using XML-coded permission sets:

```
<assembly: PermissionSetAttribute(SecurityAction.RequestMinimum, File
:= "Filename.xml")>
```

## Demanding Permissions

By demanding permissions, you force the caller to have a specific permission it needs to execute the code. If the caller has this request, it is very likely that he obtained it by requesting it at the CLR. As we discussed before, a permission demand triggers a security stack walk. Even if you do not perform these demands yourself, the .NET Framework classes will. This means that you should never perform permission demands related to these classes, because they will take care of those themselves. If you do perform a demand, it will be a redundant one and only add to the execution overhead. This does not mean that you should ignore it; instead, when writing code, you must be aware of which call will trigger a stack walk and make sure that the code does not encourage a surplus of stack walks. However, when you build your own classes that access protected resources, you need to place the proper permission demands, using the declarative or imperative security syntax.

Using the declarative syntax when making a permission demand is preferable to using the imperative syntax, because the latter may result in more stack walks.

There are, of course, cases that are better suited for imperative permission demands. For example, if a Registry key has to be set under specific conditions, you will perform an imperative *RegistryPermission* demand just before the code actually is called. This also implies that the caller can lack this permission, which will result in an exception that the code needs to handle accordingly. Another reason why you want to use imperative demands is when information is not known at compile time. A simple example is *FileIOPermission* on a set of files whose names are only known during runtime because they are user-related.

Two types of demands are handled differently than previously described. First, the *link demand* can be used only in a declarative way at the class or method level. The link demand is performed only during the JIT compilation phase, in which it is checked if the calling code has sufficient permission to link to your code. A security stack walk is not performed because linking exists only in a direct relation between the caller and code being called. The use of link demands can be helpful to methods that are accessible through reflection. The link demand will not only perform a security check on code that obtains the *MethodInfo* object,

### Security • Chapter 12 571

hence performing the reflection, but the same security check is performed on the code that will make the actual call to the method. The following two examples show a link demand at class and at method level:

```
<SecurityPermissionAttribute(SecurityAction.LinkDemand, _
Unrestricted := True)> Public Class ClassAct
Public Shared Function _
<SecurityPermissiobAttribute(SecurityAction.LinkDemand)> Act1()
As Integer
' body of the function
End Function
```

The second type of demand is *inheritance demand*, which can be used at both the class and method level, through the declarative security. Placing an inheritance demand on a class can protect that class from being inherited by a class that

does not have the specified permission. Although you can use a default permission, it makes sense to create a custom permission that must be assigned to the inheriting class to be able to inherit from the class with the inheritance demand. The same goes for the class that inherits from the inheriting class. For example, let's say that you have created the *ClassAct* class that is inheritable, but also has an inheritance demand set. You have defined your own inherit permission *InheritAct*. Another class called *ClassActing* wants to inherit from your class, but because it is protected with an inheritance demand, it must have the *InheritAct* permission in order to be able to inherit. Let's assume that this is the case. Now there is another class called *ClassReacting* that wants to inherit from the class *ClassActing*. In order for *ClassReacting* to inherit from *ClassActing*, it also needs to have the *InheritAct* permission assigned. The inheritance demand would look like this:

```
<InheritActAttribute(SecurityAction.InheritanceDemand)> Public Class  
ClassAct
```

The inheritance demand at method level can be the following:

```
Public Overridable Function  
<SecurityPermissionAttribute(SecurityAction.InheritanceDemand)>  
Act1() as Integer  
' Body of the function  
End Function
```

## 572 Chapter 12 • Security

### Overriding Security Checks

Because stack walking can introduce serious overhead and thus performance degradation, you need to keep stack walks under control. This is especially true if they do not necessarily contribute to security, such as when a part of the execution can only take place in fully trusted code. On the other hand, your code has permission to access specific protected resources, but you do not want code that you call to gain access to these resources—so you want to have a way of preventing this. In both cases, you want to take control of the permission security checks, hence overriding security checks. You can do this by using the following security actions: *Assert*, *Deny*, and *PermitOnly* (meaning “deny everything but”). After the code sets an override, it can undo this override by calling the corresponding *Revert* method, respectively *RevertAssert*, *RevertDeny* and *RevertPermitOnly*. Get in the practice of first calling the *Revert* method before setting the override because performing a revert on a nonexisting override has no effect.

#### **WARNING**

You can place more than one override of the same type, for example *Deny*, within the same piece of code. However, this is not acceptable to the CLR. If during a stack walk the CLR encounters more than one of the same asserts it throws an exception, because it does not know which of the overrides to trust. If you have more than one place in a piece of code where you set an override, be sure to revert the first one before setting the new one.

#### *Assert Override*

When you set an assert override on a specific permission, you force a stack walk on this permission to stop at your code and not continue to check the callers of your method.

## WARNING

If you use an *assert*, you inadvertently create a security vulnerability, because you prevent the CLR from completing security checks. You must convince yourself that this vulnerability cannot be exploited.

### Security • Chapter 12 573

The use of *Assert* makes sense in the following situations:

\_ You have coded a part of an application that will never be exposed to the outside world. The user of the application has no way of knowing what happens within that part of the application. Your code does need access to protected resources, such as system files and/or Registry keys, but because the callers will never find out that you use these protected resources, it is reasonably safe to set an *Assert* to prevent a full security check from being performed. You do not care if the caller has that permission or not.

\_ Your code needs to make one or more calls to unmanaged code, but because the caller of the code obtains access through your Web site, you are safe in assuming that they will not have permissions to make calls to unmanaged code. On the other hand, the callers cannot influence the calls you make to unmanaged code. Therefore, it is reasonably safe to assert the permission to access unmanaged code.

\_ You know that somewhere in your code you have to perform a search, using a **Do..Loop** structure that at one point has to access a protected resource. You also know that the code that calls the protected resource cannot be called from outside the loop. Therefore, you decide to set an assertion just before the call to the protected resource, to prevent a surplus of stack walks. In case the particular piece of code that does the call to the protected resource can be called by other code, you have to move up the assertion to the code that can only be called from the loop.

Let's take a look at the stack walk that was initially used in Figure 12.1, but now we throw in an assertion and see what happens (see Figure 12.3). The assert is set in *Assembly4* on the *UIPermission*. In the situation with no assert, the stack walk did not succeed because *Assembly1* did not have this permission. Now the stack walk starts at *Assembly6* performing a permission demand on *UIPermission*, and goes on its way as it usually goes. Now the stack walk reaches *Assembly4* and recognizes an assert on the permission it is checking. The stack walk stops there and returns with a positive result. Because the stack walk was short-circuited, the CLR has no way of knowing that *Assembly1* did not have this permission.

An *Assert* can be set using both the declarative and the imperative syntax. In the first example, the declarative syntax is used. An *Assert* is set on the *FileIOPermission*. Write permission for the CFG files in the C:\Test directory:

### 574 Chapter 12 • Security

```
Public Function _
<FileIOPermission(SecurityAction.Assert, Write := "C:\Test\*.cfg")> _
Act1() As Integer
' body of the function
End Function
```

The second example uses the imperative syntax setting the same type of *Assert*:

```

Public Function Act1() As Integer
Dim ActFilePerm As New
FileIOPermission(FileIOPermissionAccess.Write, "C:\Test\*.cfg")
ActFilePerm.Assert
' rest of body
End Function

```

## *Deny Override*

The *Deny* does the opposite of *Assert* in that it lets a stack walk fail for the permission the *Deny* is set on. There are not many situations where a *Deny* override

### **Figure 12.3** A Stack Walk Is Short-Circuited by an *Assert*

```

Calling Chain on the Stack
Assembly1
Method1a Granted:
FileIOPermission
Assembly2
Method2a Granted:
FileIOPermission
UIPermission
Assembly3
Method3a Granted:
FileIOPermission
UIPermission
Assembly4
Method4a Granted:
FileIOPermission
UIPermission
Assembly5
Method5a Granted:
FileIOPermission
UIPermission
Assembly6
Method6a Granted:
FileIOPermission
UIPermission
UIPermission
(SecurityAction.Demand)
Succeeded
Succeeded
Stack Walk Result: SUCCESS
Security Stack Walk demanding the UIPermission
UIPermission(SA.Assert)

```

#### **Security • Chapter 12 575**

makes sense, but here is one: Among the permissions your code has is *RegistryPermission*. Now it has to make a call to a method for which you have no information regarding trust. To prevent that code from taking advantage of the *RegistryPermission*, your code can set a *Deny*. Now you are sure that your code does not hand over a high-trust permission.

Because unnecessary *Deny* overrides can disrupt the normal working of security checks (because they will always fail on a *Deny*), you should revert the *Deny* after the call ends for which you set the *Deny*.

For the sake of the example, we use the same situation as in Figure 12.3, but instead of an *Assert*, there is a *Deny* (see Figure 12.4). Again, the security stack walk is triggered for the *UIPermission* permission in *Assembly6*. When the stack walk reaches *Assembly4*, it recognizes the *Deny* on *UIPermission* and it ends with a fail. In our example, the security check would ultimately have failed in *Assembly1*, but if *Assembly1* had been granted the *UIPermission*, the stack walk would have succeeded, if not for the *Deny*. Effectively this means that *Assembly4* revoked the

*UIPermission* for *Assembly5* and *Assembly6*.

You can set a *Deny* by using both the declarative and the imperative syntax.

In the first example, the declarative syntax is used. A *Deny* is set on the *FileIOPermission* permission for all the files in the C:\Winnt\System32 directory:

### Figure 12.4 A Stack Walk Is Short-Circuited by a *Deny*

```
Calling Chain on the Stack
Assembly1
Method1a Granted:
FileIOPermission
Assembly2
Method2a Granted:
FileIOPermission
UIPermission
Assembly3
Method3a Granted:
FileIOPermission
UIPermission
Assembly4
Method4a Granted:
FileIOPermission
UIPermission
Assembly5
Method5a Granted:
FileIOPermission
UIPermission
Assembly6
Method6a Granted:
FileIOPermission
UIPermission
UIPermission
(SecurityAction.Demand)
Failed
Succeeded
Stack Walk Result: FAIL
Security Stack Walk demanding the UIPermission
UIPermission(SA.Deny)
```

#### 576 Chapter 12 • Security

```
Public Function _
<FileIOPermission(SecurityAction.Deny, All :=
"C:\Winnt\System32\*.*)"> _
Act1() As Integer
' body of the function
End Function
```

The second example uses the imperative syntax setting the same type of *Assert*:

```
Public Function Act1() As Integer
Dim ActFilePerm As New
FileIOPermission(FileIOPermissionAccess.AllAccess, _
"C:\Winnt\System32\*.*)"
ActFilePerm.Deny
' rest of the body
End Function
```

### *PermitOnly* Override

The *PermitOnly* override is more like the negation of the *Deny*, by *Denying* every permission but the one specified. You use the *PermitOnly* for the same reason you use *Deny*, only this one is more rigorous. For example, if you permit only the *UIPermission* permission, every security stack walk will fail but the one that checks on the *UIPermission*. Take Figure 12.4 and substitute *Deny* with *PermitOnly*. If in *Assembly6* the security check for *UIPermission* is triggered, the stack walk will pass *Assembly4* with success, but will ultimately fail in *Assembly1*. If

any other security check is initiated, they will fail in *Assembly*. The end result is that *Assembly5* and *Assembly6* are denied any access to a protected resource that incorporate a *Demand* request, because every security check will fail. As you can see, *PermitOnly* is a very effective way of killing any aspirations of called code in accessing protected resources. The *PermitOnly* is used in the same way as *Deny* and *Assert*.

## Custom Permissions

The .NET Framework enables you to write your own code access permissions, even though the framework comes with a large number of code access permission classes. Because these classes are meant to protect the protected resources and code that are exposed by the framework, it may well be the case that the application you are developing has defined resources that are not protected by the

### Security • Chapter 12 577

framework permissions, or you want to use permissions that are more tuned toward the needs of your application.

You are completely free to replace existing framework permission classes, although this requires a large amount of expertise and experience. In case you are just adding new permission classes to the existing ones, you should be particularly careful not to overlap permissions. If more than one permission protects the same resource or operation, an administrator has to take this into account if he has to modify the rights to these resources.

### NOTE

The subject of overlapping permissions brings up a topic not discussed earlier. Although the whole discussion of code access permission has been from the standpoint of the CLR, or .NET Framework, eventually the CLR has to access resources on behalf of the users/application. Even if the code has been granted a specific permission to access a protected resource, that does not automatically mean that it is allowed to access that system resource. Take the example of a method having the *FileIOPermission* permission to the directory C:\Winnt\System32. If the identity of the Windows principal has not been given access to this part of the file system, accessing a file in that directory will fail anyway. This implies that the administrator not only has to set up the permissions within the security policy, but he also has to configure the Windows 2000 platform to reflect these access permissions.

Building your own permissions does not only imply that certain development issues are raised, but even more so the integrity of the whole security system must be discussed. You have to take into account that you are adding to a rigid security system that relies heavily on trust and permissions. If mistakes occur in the design and/or implementation of a permission, you run the risk of creating security holes that can become the target of attacks or grant an application access to protected resources even if it is not authorized to access these. Discussing the process of designing your own permissions goes beyond the scope of this chapter. However, the following steps give you an understanding of what is involved in creating a custom permission:

1. Design a permission class.

2. Implement the interfaces *IPermission* and *IUnrestrictedPermission*.

#### 578 Chapter 12 • Security

3. In case special data types have to be supported, you must implement the interface *ISerializable*.

4. You must implement XML encoding and decoding.

5. You must implement the support for declarative security.

6. Add Demand calls for the custom permission in your code.

7. Update the security policy so that the custom permission can be added to permission sets.

## Role-Based Security

Role-based security is not new to the .NET Framework. If you already have experience with developing COM+ components, you surely have come across role-based security. The concept of role-based security for COM+ applications is the same as for the .NET Framework. The difference lies in the way it is implemented.

If we talk about role-based security, the same example comes up, over and over again. This is not because we can't create our own example, but because it explains role-based security in a way everybody understands. So here it is. You build a financial application that can handle deposit transactions. The rule in most banks is that the teller is authorized to make transactions up to a certain amount, let say \$5,000. If the transaction goes beyond that amount, the teller's manager has to step in to perform the transaction. However, because the manager is only authorized to do transaction up to \$10,000, the branch manager has to be called to process a deposit transaction that is over this amount.

So, as you can see, role-based security has to do with limiting the tasks a user can perform, based on the role(s) he plays or the identity he has. Within the .NET Framework, this all comes down to the principal that holds the identity and role(s) of the caller. As discussed earlier in this chapter, every thread is provided with a principal object. In order to have the .NET Framework handle the role-based security in the same manner as it does code access security, the permission class *PrincipalPermission* is defined. To avoid any kind of confusion, *PrincipalPermission* is not a derived class of *CodeAccessPermission*. In fact, *PrincipalPermission* holds only three attributes: User, Role, and the Boolean *IsAuthenticated*.

## Principals

Let's get back to where it all starts: the principal. From the moment an application domain is initialized, a default call context is created to which the principal

#### Security • Chapter 12 579

will be bound. If a new thread is activated, the call context and the principal are copied from the parent thread to the new thread. Together with the principal object, the identity object is also copied. If the CLR cannot determine what the principal of a thread is, a default principal and identity object is created so that the thread can run at least with a security context with minimum rights. There are three type of principals: *WindowsPrincipal*, *GenericPrincipal*, and *CustomPrincipal*. The latter goes beyond the scope of this chapter and is not discussed any further.

## *WindowsPrincipal*

Because the *WindowsPrincipal* that references the *WindowsIdentity* is directly related to a Windows user, this type of identity can be regarded as very strong because an independent source authenticated this user.

To be able to perform role-based validations, you have to create a *WindowsPrincipal* object. In the case of the *WindowsPrincipal*, this is reasonably straightforward, and there are actually two ways of implementing it. This depends on whether you have to perform just a single validation of the user and role(s), or you have to do this repeatedly. Let's start with the single validation solution:

1. Initialize an instance of the *WindowsIdentity* object using this code:

```
Dim WinIdent as WindowsIdentity = WindowsIdentity.GetCurrent()
```

2. Create an instance of the *WindowsPrincipal* object and bind the *WindowsIdentity* to it:

```
Dim WinPrinc as New WindowsPrincipal(WinIdent)
```

3. Now you can access the attributes of the *WindowsIdentity* and *WindowsPrincipal* object:

```
Dim PrincName As String = WinPrinc.Identity.Name  
Dim IdentName As String = WinIdent.Name 'this is the same as  
the previous line  
Dim IdentType As String = WinIdent.AuthenticationType
```

If you have to perform role-based validation repeatedly, binding the *WindowsPrincipal* to the thread is more efficient, so that the information is readily available. In the previous example, you did not bind the *WindowsPrincipal* to the thread because it was intended to be used only once. However, it is good practice to always bind the *WindowsPrincipal* to the thread because in case a new thread is created, the principal is also copied to the new thread:

### **580 Chapter 12 • Security**

1. Create a principal policy based on the *WindowsPrincipal* and bind it to the current thread. This initializes an instance of the *WindowsIdentity* object, creates an instance of the *WindowsPrincipal* object, binds the *WindowsIdentity* to it, and then binds the *WindowsPrincipal* to the current thread. This is all done in a single statement:

```
AppDomain.CurrentDomain.SetPrincipalPolicy(PrincipalPolicy.  
WindowsPrincipal)
```

2. Get a copy of the *WindowsPrincipal* object that is bound to the thread:

```
Dim WinPrinc As WindowsPrincipal =  
Ctype(Thread.CurrentPrincipal, WindowsPrincipal)
```

It is possible to bind the *WindowsPrincipal* in the first method of creation to the thread. However, your code must be granted the *SecurityPermission* permission to do so. If that is the case, you bind the principal to the thread by the following:

```
Thread.CurrentPrincipal = WinPrinc
```

## *GenericPrincipal*

In a situation where you do not want to rely on the Windows authentication but want the application to take care of it, you can use the *GenericPrincipal*.

### **NOTE**

Always use an authentication method before letting a user access your application. Authentication, in any shape or form, is the only way to establish an identity. Without it you are not able to implement role-base

security.

Let's assume that your application requested a username and password from the user, checked it against the application's own authentication database, and established the user's identity. You then have to create the *GenericPrincipal* to be able to perform role-based verifications in your application:

1. Create a *GenericIdentity* object for the *User1* you just authenticated:

```
Dim GenIdent As New GenericIdentity("User1")
```

#### Security • Chapter 12 581

2. Create the *GenericPrincipal* object, bind the *GenericIdentity* object to it, and add roles to the *GenericPrincipal*:

```
Dim UserRoles as String() = {"Role1", "Role2", "Role5"}  
Dim GenPrinc As New GenericPrincipal(GenIdent, UserRoles)
```

3. Bind the *GenericPrincipal* to the thread. Again, you need

*SecurityPermission*:

```
Thread.CurrentPrincipal = GenPrinc
```

## Manipulating Identity

You can manipulate the identity that is held by a principal object in two ways.

The first is replacing the principal; the second is by impersonating.

Replacing the principal object on the thread is a typical action you perform in applications that have their own authentication methods. To be able to replace a principal, your code must have been granted the *SecurityPermission*, or more specifically, the *SecurityPermission* attribute *ControlPrincipal*. This will allow your own code to be able to pass on the *PrincipalObject* to other code. This attribute grants you the permission to manipulate the principal, so you are allowed by the CLR to pass on the principal. Replacing the principal object can be done by performing these steps:

1. Create a new identity and principal object and initialize it with the proper values.

2. Bind the new principal to the thread:

```
Thread.CurrentPrincipal = NewPrincipalObject
```

Impersonating is also a way of manipulating the principal, with the intent to take on the identity of another user to perform some actions on their behalf. You can identify two variations:

\_ The code has to impersonate the *WindowsPrincipal* that is attached to the thread. This may seem a little odd, but you have to remember that your code is part of an application domain that runs in a process. A user—whether a system account, a service account, or even an interactive user—starts this process on the Windows platform. Although the principal can be used to perform role-based verification within the code, accessing protected resources is still done with the identity of the process user, unless you actively use the user account of principal through impersonation.

#### 582 Chapter 12 • Security

\_ The code has to impersonate a user that is not attached to the current thread. The first thing you have to do is obtain the Windows token of the user you want to impersonate. This has to be done with the unmanaged code *LogonUser*. The obtained token has to be passed to a new

*WindowsIdentity* object. Now you have to call the *Impersonate* method of *WindowsIdentity*. The old identity, hence token, has to be saved in a new instance of *WindowsImpersonationContext*.

At the end of the impersonation, you have to change back to the original user account by calling the *Undo* method of the *WindowsImpersonationContext*. Remember the principal object is not changed, rather the *WindowsIdentity* token, representing the Windows account, is switched with the current token. At the end of the impersonation, the tokens are switched back again, as shown in the following steps:

1. Call the *LogonUser* method, located in the unmanaged code library *advapi32.dll*. You pass the username, domain, password, logon type, and logon provider to this method that will return you a handle to a token. For the sake of the example, we will call it *hImpToken*.
2. Create a new *WindowsIdentity* object and pass it the token handle:  
`Dim ImpersIdent As New WindowsIdentity(hImpToken)`
3. Create a *WindowsImpersonationContext* object and call the *Impersonate* method of *ImpersIdent*:  
`Dim WinImpersCtxt As WindowsImpersonationContext = ImpersIdent.Impersonate()`
4. At the end of the call, the original Windows token has to be put back in the Identity object:  
`WinImpersCtxt.Undo()`

You could have done Steps 2 and 3 in one statement that looks like this:

```
Dim WinImpersCtxt As WindowsImpersonationContext = _  
WindowsIdentity.Impersonate(hImpToken)
```

Remember that you cannot impersonate when you use a *GenericPrincipal* because it does not reference a Windows identity. For generic principals, you will need to replace the principal with one that has a new identity.

**Security • Chapter 12 583**

## Role-Based Security Checks

Having discussed the creation and manipulation of *PrincipalObject*, it is time to take a look at how they can assist you in performing role-based security checks. Here is where *PrincipalPermission*, already mentioned in the beginning of the section “Role-Base Security,” comes into play. Using *PrincipalPermission*, you can make checks on the active principal object, be it the *WindowsPrincipal* or the *GenericPrincipal*. The active principal object can be one you created to perform a one-time check, or it can be the principal you bound to the thread. Like the code access permissions, the *PrincipalPermission* can be used in both the declarative and the imperative way.

To use *PrincipalPermission* in a declarative manner, you need to use the *PrincipalPermissionAttribute* object in the following way:

```
Public Shared Function  
<PrincipalPermissionAttribute(SecurityAction.Demand, _  
Name := "User1", Role := "Role1")> Act2()  
As Integer  
' body of the function  
End Function  
<assembly: PrincipalPermissionAttribute(SecurityAction.Demand, Role :=  
'Administrator')>
```

To use the imperative manner, you can perform the *PrincipalPermission* check

as shown:

```
Dim PrincPerm As New PrincipalPermission("User1", "Role1")
PrincPerm.Demand()
```

It is also possible to use the imperative to set the *PrincipalPermission* object in two other ways:

```
Dim PrincState As PermissionState = Unrestricted
Dim PrincPerm As New PrincipalPermission(PrincState)
```

The permission state (*PrincState*) can be *None* or *Unrestricted*, where *None* means the principal is not authenticated. So, the user name is *Nothing*, the role is *Nothing*, and *Authenticated* is false. *Unrestricted* matches all other principals.

```
Dim PrincAuthenticated As Boolean = True
Dim PrincPerm As New PrincipalPermission("User1", "Role1",
PrincAuthenticated)
```

### 584 Chapter 12 • Security

The *IsAuthenticated* field (*PrincAuthenticated*) can be true or false. In a situation where you want *PrincipalPermission.Demand()* to allow more than one user/role combination, you can perform a union of two *PrincipalPermission* objects.

However, this is only possible if the objects are of the same type. Thus, if one *PrincipalPermission* object has set a user/role, and the other object uses *PermissionState*, the CLR throws an exception. The union looks like this:

```
Dim PrincPerm1 As New PrincipalPermission("User1", "Role1")
Dim PrincPerm2 As New PrincipalPermission("User2", "Role2")
PrincPerm1.Union(PrincPerm2).Demand()
```

The *Demand* will succeed only if the principal object has the user *User1* in the role *Role1* or *User2* in the role *Role2*. Any other combination fails.

As mentioned before, you can also directly access the principal and identity object, thereby enabling you to perform your own security checks without the use of *PrincipalPermission*. Besides the fact that you can examine a little more information, it also prevents you from handling exceptions that can occur using *PrincipalPermission*. You can query the *WindowsPrincipal* in the same way the *PrincipalPermission* does this:

\_ The name of the user by checking the value of

*WindowsPrincipal.Identity.Name*:

```
If (WinPrinc.Identity.Name = "User1") or _
WinPrinc.Identity.Name.Equals("DOMAIN1\User1") Then
End If
```

\_ An available role by calling the *IsInRole* method:

```
If (WinPrinc.IsInRole("Role1")) Then
End If
```

\_ Determining if the principal is authenticated, by checking the value of

*WindowsPrincipal.Identity.IsAuthenticated*:

```
If (WinPrinc.Identity.IsAuthenticated) Then
End If
```

Additionally for *PrincipalPermission*, you can check the following *WindowsIdentity* properties:

### Security • Chapter 12 585

\_ **AuthenticationType** Determines the type of authentication that is used. Most common values are NTLM and Kerberos.

\_ **IsAnonymous** Determines if the user is identified as an anonymous account by the system.

\_ **IsGuest** Determines if the user is identified as a guest account by the system.

\_ **IsSystem** Determines if the user is identified as the system account of the system.

\_ **Token** Returns the Windows account token of the user.

## Security Policies

This section takes a closer look at the way security policies are constructed and the way you can manage them. To create and modify a security policy, the .NET Framework provides you two tools: a command-line interface (CLI) tool, called **caspol.exe** (see the section “Security Tools”) and a Microsoft Management Console snap-in, “mcsccorcfg.msc” (see Figure 12.5). The latter will be used for demonstration purposes because it is more visual and intuitive.

### Figure 12.5 The .NET Configuration Snap-In

586 Chapter 12 • Security

As you can see in Figure 12.5, the security policy model is made up of the following:

\_ Runtime Security Policy levels:

\_ **Enterprise** Valid for all managed code that is used within the whole organization (enterprise); therefore this will have “by nature” a restrictive policy because it references a large group of code.

\_ **Machine** Valid for all managed code on that specific computer. Because this already limits the amount of code, you can be more specific with handing out permissions.

\_ **User** Valid for all the managed code that runs under that Windows user. This will normally be the account that starts the process in which the CLR and managed code runs. Because the identity of the user is very specific, the granted permissions can also be more specific, thus less restrictive.

\_ A code groups hierarchy that exists for each of the three policy levels. We will look at how you can add code groups to the default structure, which already exists for user and machine.

\_ (Named) Permission Sets. By default the .NET Framework comes with seven named permission sets:

\_ **FullTrust** Unlimited access to all protected resources and operations.

\_ **Everything** Granted all .NET Framework permissions, except the security permission *SkipVerification*.

\_ **LocalIntranet** The default rights given to an application on the local intranet.

\_ **Internet** The default rights given to an application on the Internet.

\_ **Execution** Has only the security permission `EnableAssemblyExecution`.

\_ **SkipVerification** Has only the security permission `SkipVerification`.

\_ **Nothing** Denied all access to all protected resources and operations.

\_ Evidence, which is the attribute that the code hands over to the CLR and on which it determines the effective permission set. Evidence is

used in the construction of code groups.

#### **Security • Chapter 12 587**

`_Policy` assemblies that list the trusted assemblies that hold security objects used during policy evaluation. You should add your assemblies to the list that implements the custom permissions. If you omit this, the assemblies will not be fully trusted and cannot be used during the evaluation of the security policy.

Understand that the evaluation process of the security policy will result in the effective permission set for a specific assembly. For all of the three policy levels, the code groups are evaluated against the evidence presented by the assembly. All the code groups that meet the evidence deliver a permission set. The union of these sets determines the effective permission set for that particular security policy level. After this evaluation is done at all three security levels, the three individual permission sets are intersected, resulting in the effective permission set for an assembly. This means that the code groups within the three security levels cannot be constructed independently, because this may result in a situation where an assembly is given a limited permission set that is too limited to run. When you take a look at the permission set for the *All\_Code* of the enterprise security policy, you will see that it is Full Trust. Doing the same for the *All\_Code* of the user security policy, you will see Nothing. Because the code group tree of the enterprise is empty, it cannot make evidence decisions; therefore it cannot contribute to the determination of the effective permission set of the assembly. By setting it to Full Trust, it is up to the machine and user security policy to determine the effective permission set. Because the user code group already has a limited code group tree, the root does not need to participate in the determination of the permission set. By setting it to Nothing, it is up to the rest of the code groups to decide what the effective permission group for the user security policy is. You can determine the permission set of a code group by performing these steps:

1. Run Microsoft Management Console (MMC) by choosing **Start | Run** and typing **mmc**.
2. Open the .NET Management snap-in, via **Console | Add/Remove Snap-in**.
3. Expand the **Console Root | .NET Configuration | My Computer**.
4. Expand **Runtime Security Policy | Enterprise | Code Groups**.
5. Select the code group **All\_Code**.
6. Right-click **All\_Code** and select **Properties**.

#### **588 Chapter 12 • Security**

7. Select the **Permission Set** tab.
8. The **Permission Set** field lists the current value.

## **Creating a New Permission Set**

Suppose you decide that none of the seven built-in permissions sets satisfy your need for granting permissions. Therefore, you want to make a named permission set that does suit you. You have a few options:

- `_` Create a permission from scratch.

- \_ Create a new permission set based on a existing one.
  - \_ Create a new permission from an XML-coded permission set.
- To get a better understanding of the working of the security policy and to get some hands-on experience with the tool, we discuss the different security policy issues in the following exercises.
- We use the second option and base our new permission set on the permission set *LocalIntranet* for the user security policy level:
1. Expand the **User** runtime security policy and expand **Permission Sets** (see Figure 12.6).

**Figure 12.6** The Users Permission Sets and Code Groups  
**Security • Chapter 12 589**

2. Right-click the permission set **LocalIntranet** and select **Duplicate**; a permission set called **Copy of LocalIntranet** is added to the list.
3. Select the permission set **Copy of LocalIntranet** and rename it to **PrivatePermissions**. Then, right-click it and select **Properties**. Change the **Permission Set Name** to **PrivatePermissions** and, while you're at it, change the corresponding **Permission Set Description**.
4. Change the permissions of the permission set: Right-click the **PrivatePermissions** permission set and select **Change Permissions**.
5. The **Create Permission Set** dialog box appears (see Figure 12.7). You see two permissions lists: on the left, the Available Permissions that are not assigned, and on the right, the list with assigned permissions. Between the two Permissions lists are four buttons. The **Add** and **Remove** buttons let you move individual permissions between the lists. Note that you cannot select more than one at the same time. This is done to prevent you from making mistakes. You will better understand a given permission if you select that permission in the Assigned Permissions list and press the **Properties** button. You can use the fourth button (**Import**) to load an XML-coded permission set. Now, let's make some modifications to the permission set, because that was the reason to duplicate the permission set:

**Figure 12.7** Modify the Permission Set Using the Create Permission Set Dialog Box  
**590 Chapter 12 • Security**

- \_ Add the *FileIOPermission* to the Assigned Permission list.
  - \_ Add the *RegistryPermission* to the Assigned Permission list.
  - \_ Modify the *SecurityPermission* properties.
- To do so:
1. Select **FileIO** in the Available Permissions list. (Notice that if you have selected a permission in the Assigned Permissions list, this permission stays selected.)
  2. Click **Add**. A **Permission Settings** dialog box for the FileIO appears (see Figure 12.8). (You can also double-click the permission to add it to the Assigned Permissions list. But do not double-click an assigned permission by accident—this will remove the permission from the assigned permission list.) On the Permission Settings dialog box, you are given

the option to select between **Grant assemblies access to the following files and directories** and **Grant assemblies unrestricted access to the file system**.

3. Choose the first one, and because it is already selected, we can focus our attention on the empty list window below the option. You may expect an Add button below the list, especially because there is a **Delete Entry** one. However, there is an auto-add list. You fill in a line, and it is automatically added. Add a second line, and a third empty line will appear.

**Figure 12.8** Modify the Settings of FileIO Using the Permission Settings Dialog Box  
**Security • Chapter 12 591**

4. As you saw earlier this chapter, this resembles the way we used *FileIOPermission* and *FileIOPermissionAttribute* to demand and request access to specific files in a specific directory. Go ahead, fill in “C:\Test\\*.cfg”. Surprised you get an error message? The point is that the field demands that you use UNC names. The advantage is that you can reference to files on other servers in the domain. However, the dialog box checks the existence of the path when you click **OK**, so be sure that the UNC path exists.
5. Fill the File Path with a valid UNC of the machine you are working on, and because we want to give full access, you can check all four boxes. (Note that if you do not check any of the boxes, then this is accepted, because you filled in a File Path. However if you check the properties of FileIO as an assigned permission, you will notice that the line has disappeared—hence a beta bug!)
6. Click **OK** and you have added a permission to the assigned permission list. You are now ready for the next permission.
7. Double-click the **Registry** permission and a **Permissions Setting** dialog box appears that looks a lot alike the one you just saw with **FileIO**. Keep the option **Grant assemblies access to the following registry keys**.
8. Fill the **Key** field with a valid HKEY value, such as HKEY\_LOCAL\_MACHINE, and check the **Read** box, so that we can give read permission to the specified Registry tree.
9. Click **OK**, and you have added your second permission to your permission set.
10. The last task is to modify the Security permission. So, select the **Security** permission in the Assigned Permissions list (do not doubleclick, because that will remove the permission from the list) and click **Properties**.
11. A Permission Settings dialog box (see Figure 12.9) appears. You see that the option **Grant assemblies the following security permissions** is selected, together with the properties **Enable assembly execution**, **Assert any permission that has been granted**, and **Enable remoting configuration**.

## 592 Chapter 12 • Security

12. We also want to grant our security policy the security permission properties. Check **Allow calls to unmanaged assemblies** because we want to make calls to unmanaged code. Also check **Allow principal control** because we want to be able to modify principal settings. Click **OK**, and you are done, for now, with modifying your first permission set.

13. Click **Finish**. You will probably get a warning message stating that you changed your security policy and you have to save it. Up until the point you save the policy, an asterisk (\*) will mark the user policy.

14. You can save the policy by right-clicking the **User** runtime security policy and selecting **Save**.

If you want this permission set to also become part of the machine and/or enterprise permission sets, you can simply copy and paste it.

You will also notice two other options: **Reset** and **Restore Policy**. The first one resets the policy back to the default setting of the policy. You can try it, but it will wipe out all the changes you made up until now. The latter makes it possible to go back to the previous save. This is possible because for each of the runtime security policies, the settings are saved in an XML-coded file that becomes the current one. Before this happens, it renames the old one with the extension `.old`. The current one has the extension `.cch`. The default policy has no extension, so to speak. For the user security policy, you have the following files:

### Figure 12.9 Modify the Settings of Security Using the Permission Settings Dialog Box

#### Security • Chapter 12 593

`_security.config` The default security; used by the **Reset** option.

`_security.config.cch` The current/active policy.

`_security.config.old` The last saved policy version; used by the **Restore Policy** option.

The enterprise security uses the name `enterprisesec.config` and the machine uses the name `security.config`. This is possible because the user security policy is saved in the user's directory tree in the following folder:

```
Document and Settings\User_Name\Application Data\Microsoft\CLR Security config\v1.0.xxxx
```

The enterprise and machine security policies are saved in the following directory:

```
WINNT\Microsoft.NET\Framework\v1.0.xxxx\CONFIG
```

This directory is located by the CLR through the HiveKey:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Catalog42\NetFrameworkv1\MachineConfigdirectory
```

Because the configuration files are XML-coded, you can open them with a Web browser and examine them. This will give you additional understanding how the permission sets are set up. This also means that you can modify the default security policies.

## Modifying the Code Group Structure

Now that we have created a security permission set, it makes sense to start using it. We can do so by attaching it to a code group. We are going to modify the code groups structure of the user security policy. By default, the user already has a

basic structure (see Figure 12.10). A few things may strike you at first sight:   
\_ There is a code group called *Wizard\_Machine\_Policy*. The description of this group tells you that a wizard, called the Adjust Security Wizard, copied this group from the computer's policy level and that you should not modify it. This description is not totally true. In fact, if you take a closer look at these code groups, you will see that all groups that end with *\_Zone* have a permission set of Nothing. This means that you, the user, cannot make use of the permission sets of the machine that are

#### 594 Chapter 12 • Security

based on the zone evidence. However, if you are given more permissions based on the zone evidence, this will be toned down by the zone-based permission of the machine policy. The user can have permissions based on zoned evidence that is equal to or less than allowed by the machine. However, you do see zone-based code groups at the same level as the *Wizard\_Machine\_Policy*, because these are the code groups that are copied from the machine policy.

\_ The zone-based code groups contain *NetCodeGroup* and *FileCodeGroup*. As the description states, they are generated by the .NET Configuration Tool, hence the tool we are working with at the moment. The custom code groups are based on XML-code files and can therefore not be edited by the tool. However, you can use the **caspol.exe** tool to do so. Without going into detail regarding what exactly these groups entail, it suffices to state that they are necessary for you to use the .NET Configuration Tool. If you remove or modify them, you may lock yourself out from using this tool.

#### Figure 12.10 The Default Code Group Structure for the User Security Policy

##### Security • Chapter 12 595

Let's create a small code groups structure that is made up of two code groups directly under the *All\_Code* group and apply our own custom-made permission set *PrivatePermissions* to the *LocalIntranet\_Zone* group:

1. If you do not have the MMC with the .NET Management snap-in open, open it now.
2. Expand the tree to **.NET Configuration | My Computer | Runtime Security Policy | User**.
3. Now expand **Code Groups | All\_Code**.
4. Right-click **All\_Code** and select **New**; the Create Code Group dialog box appears.
5. You are given two options: **Create a new code Group** and **Import a code group from a XML File**. Use the first option. (Note: For the *NetCodeGroup* and *FileCodeGroup*, the latter is used).
6. You have to enter at least the **Name** field. For this example, we choose *PrivateGroup\_1*. Now click **Next**.
7. The dialog box shows you a second page called **Choose a condition Type** and has just one field called **Choose the condition type for**

**this code group.** The field has a pull-down menu containing the values you can choose from. All of these, except the first and last one—All Code and (custom)—are evidence-related (see Figure 12.11).

**Figure 12.11** Select a Condition Type for a Code Group

596 Chapter 12 • Security

8. Select **Site** from the drop-down menu. A new field, called **Site Name** appears and is related to the **Site** condition. For the sake of the example, we choose the MSDN Subscribers download site, so we enter the value **msdn.one.microsoft.com** in the site field.
9. Click **Next** and the third page, called **Assign a Permission Set to the Code Group**, appears.
10. You can choose between the options **Use existing permission set** and **Create a new permission set**. Because the site comes from the Internet, that permission set will do.
11. Select the value **Nothing** from the drop-down menu. (Note: The permission set we just made is also part of the list) and click **Next**.
12. Click **Finish**, and you have created your first code group. While we are at it, let's create the second code group, which will be the child of the code group we just created.
13. Right-click the code group *PrivateGroup\_1* and select **New**.
14. Create a new code group named **PrivateGroup\_2** and click **Next**.
15. Select the value **Publisher** from the drop-down menu. Below the field, a new box called **Publisher Certificate Details** appears and has to be filled by importing a certificate. You can do this by reading out of a signed assembly using the **Import from Signed File** button (Note: it should say Import from signed Assembly). Or, you can import a certificate file, using the **Import from Certificate File** button.
16. For the purpose of this example, we use the Certificate from the msdn.one.microsoft.com site. (Note: In case you have forgotten how this is done, you go to a protected site, thus using SSL. You double-click the icon indicating that the site is protected. This opens up the certificate. Go to the **Details** tab and click the **Copy to File** button.) See CD file Chapter 12/MSDN-One.cer.
17. Click the **Import from Certificate File** button, browse to the certificate file (the extension is .cer) and open it. You will see that the field in the certificate box will be filled (see Figure 12.12).
18. Click **Next**.
19. Select the existing permission group **LocalIntranet**. We can give more permissions now we know that the signed assemblies indeed comes from Microsoft MSDN, but also originates from the corresponding Web site.

Security • Chapter 12 597

20. Click **Next** and **Finish**.

Before tackling our last task, let's recap what we have done. We were concerned with creating a permission set for signed assemblies that come from the msdn.one.microsoft.com site. So what if the assembly comes from this Web site but is not

signed? It meets the condition of *PrivateGroup\_1*, so it will get the permission set of this code group. Because this is *Nothing*, this would mean that these assemblies are granted no permission. But because the `msdn.one.microsoft.com` site comes from the *Internet Zone*, it also meets the condition of the code group *Internet\_Zone*, which grants any assembly from this zone the *Internet* permission set. And because a union is taken from all the granted permission sets, these assemblies will still have enough permissions to run.

Why not make the *PrivateGroup\_2* a child of *Internet\_Zone*, because unsigned assemblies from `msdn.one.microsoft.com` are granted the *Internet* permission set any way? The reason is simple: We only want to give signed assemblies from `msdn.one.microsoft.com` additional permission if they also originate from the appropriate Web site. In case such a signed assembly originates from another Web site, we treat it as any other assembly coming from an *Internet Zone*. The reason for giving *PrivateGroup\_1* the *Nothing* permission set is that it is only there to force assemblies to meet both conditions, and *PrivateGroup\_1* is just an intermediate stage to meet all conditions.

### **Figure 12.12** Importing a Certificate for a Publisher Condition in a Code Group

#### **598 Chapter 12 • Security**

What you have to keep in mind is that we only discussed how the actual permission set is determined at the user security policy level. This will be intersected with the actual permission set determined on the machine level. And because at the machine level the assembly will be given only the *Internet* permission set, our signed assembly will wind up with the effective permission set of *Internet*. Normally, the actual permission set of the enterprise is also taken into the intersection, but because that code group tree has only the *All\_Code* code group with full trust, it will play no role in the intersection of this example.

Our last task is replacing a permission set:

1. Right-click the code group *LocalIntranet\_Zone* and select **Properties**. The **LocalIntranet\_Zone Properties** dialog box appears (see Figure 12.13).
2. Select the **Permission Set** tab.
3. Open the pop-up menu with available permission sets and select **PrivatePermissions**. You will see that the list box will reflect the permissions that make up the *PrivatePermissions* permission set.
4. Click **Apply** and go back to the **General** tab.

On this tab, there is a frame called **If the membership condition is met**, which shows two options:

### **Figure 12.13** Setting Attributes in the General Tab of the Code Group Permission Dialog Box

#### **Security • Chapter 12 599**

**\_ This policy level will have only the permissions from the permission set associated with this code group.** This refers to the code group attribute *Exclusive*.

**\_ Policy levels below this level will not be evaluated.** This refers to the code group attribute *LevelFinal*.

Both need some explanation, so let's go back to our msdn.one.microsoft.com example. Suppose you open the properties dialog box of the *Internet\_Zone* code group and check the **Exclusive** option (of course, you have to save it first for it to become active). We received a signed assembly from msdn.one.microsoft.com that also originates from this site. We had established that it would be granted the *LocalIntranet\_Zone* permission at the user policy level. But now the **Exclusive** option comes into play. Because our signed assembly also meets the *Internet\_Zone* condition, the Internet permission set is valid. The exclusive that is set for the *Internet\_Zone* code group forces all other valid permission sets to be ignored by not taking a union of these permission sets. Instead, the permission set with the exclusive attribute becomes the actual permission set for the user policy level. Because it will be intersected with the actual permission sets of the other security levels, it also determines the maximum set of permissions that will be granted to the signed assembly. Use this attribute with care, because from all the code groups an assembly is a member, hence meets the condition, only one can have the exclusive attribute. The CLR determines if this is the case. When the CLR determines that an assembly meets the condition of more than one code group with the Exclusive attribute, it will throw an exception, and it fails to determine the effective permission set and the assembly is not allowed to execute. The way the *LevelFinal* is handled is more straightforward. Understand that by establishing the effective permission set of an assembly, the CLR evaluates the security policies starting at the highest level (enterprise, followed by user and machine). Again take our MSDN example. We set a *LevelFinal* in the *PrivateGroup\_2* code group and removed the Exclusive attribute from *Internet\_Zone*. When the effective permission set for a signed assembly from msdn.one.microsoft.com that originates from that Web site has to be established, the CLR starts with determining the actual permission set of the enterprise policy level. This is for *All\_Code* Full Trust, effectively taking this policy level out of the intersection of actual permission sets. Now the user policy level gets its turn in establishing the actual permission set. As you know by now, this will be equal to the *LocalIntranet\_Zone* permission set. But the CLR has also encountered the *LevelFinal* attribute. It refrains from establishing the actual permission set of

#### 600 Chapter 12 • Security

the machine policy level and intersects the actual permission sets from the enterprise and user policy level. The actual permission set will be equal to *LocalIntranet\_Zone*.

Because the machine policy level is not considered the actual permission set in this case has more permission than in the situation where the *LevelFinal* attribute has not been set.

## Remoting Security

Discussing security between systems always provides a new set of security issues. This is no exception for remoting. Let's start with the communication between systems. If you use an *HttpChannel*, you can make use of the SSL encryption. The *FtpChannel* does not have encryption, but if both servers support IPSec, you are able to create a secured channel, through which the *FtpChannel* can communicate.

The next issue is to what extent you trust the other system. Even with a secure channel in place, how do you know that the other system has not been compromised? You need at least a sturdy authentication mechanism in place and need to avoid the use of anonymous users, although this will not always be possible. At least try to use NTLM or Kerberos for authentication. The latter is a perfect vehicle for handling impersonation between multiple systems. If you need to use anonymous users, you can use IIS as the store-front and let the IIS handle the impersonation. You can also use a proxy to prevent a user from directly accessing your IIS. The messages that are exchanged should always be signed so you are able to verify the sender and/or origin. Even when you are sure that a message is transported over a secured channel, you are never sure if the message that is put in this channel, has been sent out of ill-intent. This chapter has discussed the use of code access and role-base security. The more thoroughly you use this runtime security instrument, the better you can control the remoting security.

## Cryptography

There is no subject about security that does not reference cryptography. Although it is an absolute necessity to create a secure environment, it is not the “Holy Grail” of security. This section highlights the cryptography features that come with the .NET Framework. If you already have worked with Windows 2000 Cryptographic Service Providers (CSPs) and/or used the CryptoAPI, you know nearly everything there is to know about cryptography in the .NET Framework.

### Security • Chapter 12 601

The most important observation is that the ease-of-use of crypto functionalities have improved a lot over the way we had to use the CryptoAPI, which only was available for C/C++. An important addition in the design concept of the cryptography namespace is the use of *CryptoStreams*, which make it possible to chain any cryptographic object that makes use of CryptoStreams together. This means that the output from one cryptographic object can be directly forwarded as the input of another cryptographic object without the need of storing the output result in an intermediate object. This can enhance the performance significantly if large pieces of data have to be encoded or hashed. Another addition is the functionality to sign XML code, although only for use within the .NET Framework security system. To what extend these methods comply with the proposed standard RFC 3075 is unclear. Within the .NET Framework, three namespaces involve cryptography:

*\_System.Security.Cryptography* The most important one; resembles the CryptoAPI functionalities.

*\_System.Security.Cryptography.X509Certificates* Relates only to the X509 v3 certificate used with Authenticode.

*\_System.Security.Cryptography.Xml* For exclusive use within the .NET Framework security system.

The cryptography namespaces support the following CSP classes that will be matched on the Windows 2000 CSPs, by the CLR. If a CSP is available within the .NET Framework, this does not automatically implies that the corresponding

Windows 2000 CSP is available on the system the CLR is running:

*\_DESCryptoServiceProvider* Provides the functionalities of the symmetric key algorithm Data Encryption Standard.

*\_DSACryptoServiceProvider* Provides the functionalities of the asymmetric key algorithm Data Signature Algorithm.

*\_MD5CryptoServiceProvider* Provides the functionalities of the hash algorithm Message Digest 5.

*\_RC2CryptoServiceProvider* Provides the functionalities for the symmetric key algorithm RC 2 (name after the inventor: Rivest's Cipher 2).

*\_RNGCryptoServiceProvider* Provides the functionalities for a Random Number Generator.

#### **602 Chapter 12 • Security**

*\_RSACryptoServiceProvider* Provides the functionalities for the asymmetric algorithm RSA (named after the inventors Rivest, Shamir, and Adleman).

*\_SHA1CryptoServiceProvider* Provides the functionalities for the hash algorithm Secure Hash Algorithm 1.

*\_TripleDESCryptoServiceProvider* Provides the functionalities for the symmetric key algorithm 3DES.

To be complete, a short description of symmetric key algorithm, asymmetric key algorithm, and hash algorithm are given. A *symmetric key algorithm* enables you to encrypt/decrypt data that is sent between you and another party. The same key is used to both encrypt and decrypt the data. That is why it is called a symmetric algorithm. This algorithm forces you to exchange the key with your counter party, but this must be done in a way that no other party can intercept this key. Because symmetric key algorithms are often used for a short exchange of data, it is also referred to as session key algorithm. For the exchange of session keys, the parties involve use an asymmetric key algorithm.

An *asymmetric key algorithm* makes use of a *key pair*. One is private and is kept under lock and key by the owner and the other is public and available for everyone. Because the algorithm uses two related but different keys to encrypt and decrypt, it is called an asymmetric algorithm, but is also referenced as a *public key algorithm*. The public key is wrapped in a certificate that is a "proof of authenticity," and that certificate has to be issued by an organization that is trusted by all involved parties. This organization is called a certificate authority, of which Verisign is the best known. So what about using an asymmetric key algorithm to exchange symmetric keys? The best example is two Windows 2000 servers that need to regularly set up connection between both servers on behalf of their users. Each connection, hence session, has to be secured and needs to use a session key that is unique in relation to the other secured sessions. The servers exchange a session key for every connection. Both have an asymmetric key-pair and have exchanged the public key in a certificate. So if one server wants to send a session key to the other server, it uses the public key of the other server to encrypt the session key before it sends it. The server knows that only the other server can decrypt the session key because that server has the private key that is needed to decrypt the session key.

A *hash algorithm*, also referred to as a one-way hash algorithm, can take a variable piece of data and transform it to a fixed-length piece of data, called a *hash* or *message digest* that is nearly always much shorter, for example 160 bits for SHA-1.

### Security • Chapter 12 603

*One-way* means that you cannot derive the source data by examining only the digest. Another important feature of the hash algorithm is that it generates a hash that is unique for each piece of data, even if just one bit of data is changed. You can see a hash value as the fingerprint of a piece of data. Let's say, for example, you send somebody a plain text e-mail. How do you and the receiver of the email know that the message has not been altered while it was sent? Here is where the message digest comes in. Before you send your e-mail, you apply a hash algorithm on that message, and you send the message and message digest to the receiver. The receiver can perform the same hash on the message, and if both the digest and the message are the same, the message has not been altered. Yes, somebody who alters your message can also generate a new digest and obscure his act. Well, that is where the next trick comes in. When you send the digest, you encrypt it with your own private key, of which you know the receiver has the public part. Because this not only prevents the message from being changed without you and the receiver discovering it, but it also confirms to the receiver that the message came from you and only you. How?

Well, let's assume that somebody intercepts your message and wants to change it. He has your public key, so he can decrypt your message digest. But, because he doesn't have your private key, he is unable to encrypt a newly generated digest. So he cannot go forward with his plan to change the e-mail without anybody finding out. Eventually the e-mail arrives at the receiver's Inbox. He takes the encrypted digest and decrypts it using your public key. If that succeeds, he knows first of all that this message digest must have been sent by you because you are the only one who has access to the private key. He calculates the hash on the message and compares both digests. If they match, he not only knows that the message hasn't been tampered with, but also that the message came from only you because every message has a unique hash. And because he already established that the encrypted hash came from you, the message must also come from you.

## Security Tools

The .NET Framework comes with ten command-line security tools (see Table 12.4) that help you to perform your security tasks. For a more thorough description of these tools, you should consult the .NET Framework documentation.

### 604 Chapter 12 • Security

#### Table 12.4 Command-Line Security Tools

##### Name of

##### Name of Tool Executable Description

Code Access Security Policy Utility	Caspol.exe	This tool can perform any operation in relation to the code access security policy. Because it can do more than the .NET Configuration Tool we have been
-------------------------------------	------------	--

using in this chapter, it is important that you familiarize yourself with it.

**Certificate Chktrust.exe** With this tool, you can check a file that Verification Utility has been signed using Authenticode.

**Certificate Creation Makecert.exe** Creates a X.509 certificate for testing Utility purposes. A option you may consider is to install the Certificates Services on Windows 2000, which makes it a lot easier to create and maintain certificates for development and testing purposes.

**Certificate Manager Certmgr.exe** This utility manages your certificates, Utility certificate trust lists, and so on. Use the Microsoft Management Console with the certificates snap-in, which enables you to maintain not only your own certificates, but also (if you have the rights) the certificates of your computer and service accounts.

**Software Publisher Cert2spc.exe** This tool create a software publishers Certificate Test Utility certificate for one or more X.509 certificates.

**Permissions View Permview.exe** This tool enables you to view the Utility requested permissions of an assembly.

**PE Verify Utility Peverify.exe** This tool enables you to verify the type safety of a portable executable file.

**Secutil Utility Secutil.exe** This tool extracts strong name or public key information from an assembly and converts it so that you can use it directly in your code (for example, for a permission demand).

#### **Continued**

#### **Security • Chapter 12 605**

**File Signing Utility Signcode.exe** This tool enables you to sign a PE file with an Authenticode signature. If this utility is called with no command-line options, a Digital Signature Wizard is started.

**Strong Name Utility Sn.exe** This tool enables you to sign assemblies with strong names.

**Set Registry Utility Setreg.exe** This tools enables you to set Registry keys for use of public key cryptography. If you call this utility without options, it will just list the settings.

**Isolated Storage Storeadm.exe** This tool enables you to manage isolated Utility storage for the current user.

#### **Table 12.4 Continued**

#### **Name of**

#### **Name of Tool Executable Description**

## Summary

Positioning the .NET Framework as a distributed application environment, Microsoft was well aware that they had to pay attention to how an application can be secured, due to the great risks that distributed security incorporate. That is why they introduced a rights- and permission-driven security mechanism, that is flexible as well as rigid. Flexible because you can own your designed and customized permissions and rigid because it is always there, even if the application takes no notice of permissions. To add to that, the CLR will check the code on type safety (it checks whether the code is trying to stick its nose in places it does not belong) during the JIT compilation.

The .NET Common Language Runtime (CLR) will always perform a security check—called code access security—on an assembly if it wants to access a protected resource or operation. To prevent an assembly from obscuring its restricted permissions by calling another assembly, the CLR will perform a security stack walk. It checks every assembly in a calling chain of assemblies to see if every single one has this permission. If this is not the case, the assembly is not given access to this protected resource or operation.

What permissions an assembly is granted and what permission an assembly requests is controlled in two ways. The first one is controlled by code groups that grant permissions to an assembly based on the evidence it presents to the CLR. The assembly itself controls the latter. Secure conscious assemblies request only the permissions it needs, even if the CLR is willing to grant it more permissions. By doing this, the assembly insures itself from being misused by other code that wants to make use of its permission set. A code group hierarchy has to be set up by an administrator, which he can do at different security policy levels: enterprise, user, and machine.

To establish the effective set of permissions, the CLR uses a straightforward and robust method: It determines all valid permission sets based on the evidence an assembly presents per security policy level, and the actual permission set per policy level is the union of the valid permission set. The CLR does this for all the policy levels and intersects the actual permission set to determine the effective permission set of an assembly.

Added to the code access security, the CLR still supports role-based security, although its implementation is slightly different than you were accustomed to with COM. Every executing thread has a security context called principal that reference the identity of the user. The principal is also used for impersonation of the executing user. The principal comes in a few forms: based on Windows user

### **Security • Chapter 12 607**

accounts and the authentication mechanisms that come with it; not based on Windows account, called “Generic” that can be controlled by custom made authentication services and a “Base” form that enables you to custom make your own principal and identity. The code can reference the principal to check if the user has a specific role.

Still, the most important security feature is security policies, which not only allow you to create code groups but to also build your own permission set that

can be enriched with custom permissions. The custom permissions can be added to the .NET Framework without opening up the security system, provided that you make no security mistakes in the coding of the permissions.

As can be expected from every framework that relies on security, the .NET Framework comes with a complete set of cryptography functionalities, equal to what we had with the CryptoAPI, only the ease-of-use has improved a lot and is no longer dependent on C/C++. To control cryptographic functionalities, such as certificates and code signing, the .NET Framework has a set of security utilities that enables you to control and maintain the security of your applications during its development and deployment process.

## Solutions Fast Track

### Security Concepts

À Permissions are used to control the access to protected resources and operations.

À Principal is the security context that is attached to every executing thread in the CLR. It also holds the identity of the user, such as Windows account information, and the roles that user has. It also contributes to the ability of the code to impersonate.

À Authentication and authorization can be controlled by the application itself or rely on external authentication methods, such as NTLM and Kerberos. Once Windows has authorized a user to execute CLR-based code, the code has to control all other authorization that is based on the identity of the user and information that comes with assemblies, called evidence.

À Security policy is what controls the whole CLR security system. A system administrator can build policies that grant assemblies permissions

#### 608 Chapter 12 • Security

access to protected resources and operations. This permission granting is based on evidence that the assemblies hands over to the CLR. If the rules that make up the security policy are well constructed, it enables the CLR to provide a secure runtime environment.

À Type safety is related to the prevention of assembly code to reach into memory/storage of other applications. Type safety is always checked during JIT compilation and therefore before the code is even loaded into the runtime environment. Only code that is granted the Skip Verification permission can bypass type safety checking, unless this is turned off altogether.

### Code Access Security

À Code access security is based on granting assemblies permission and enforcing that it can never gain more permissions. This enforcing is done by what is known as security stack walking. When a call is made to a protected resource or operation, the assembly the CLR demanded from the assembly that has a specific permission. But instead of checking only the assembly that made the call, the CLR checks every assembly that is

part of a calling chain. If all these assemblies have that specific permission, the access to the protected resource/operation is allowed.

À To be able to write secure code, it is possible to refrain from permissions that are granted to the code. This is done by requesting the necessary permissions for the assembly to run, whereby the CLR gives the assembly only these permissions, under the reservation that the requested permissions are part of the permission set the CLR was willing to grant the assembly anyway. By making your assemblies request a limited permission set, you can prevent other code from misusing the extended permission set of your code. However, you can also make optional requests, which allows the code to be executed even if the requested permission is not part of the granted permission set. Only when the code is confronted with a demand of having such a permission, it must be able to handle the exception that is thrown, if it does not have this permission.

À The demanding of a caller to have a specific permission can be done using declarative and imperative syntax. Requesting permissions can only be done in a declarative way. Declarative means that it is not part of the

#### **Security • Chapter 12 609**

actual code but is attached to an assembly, class, or method using a special syntax enclosed with <>. When the code is compiled to the intermediate language (IL) or a portable executable (PE), these demands/request are extracted from the code and placed in the metadata of the assembly. This metadata is read and interpreted by the CLR before the assembly is loaded. The imperative way makes the demands part of the code. This can be sensible if the demands are conditional. Because a demand can always fail and result in an exception being thrown by the CLR, the code has to be equipped in handling these exceptions.

À The code can control the way the security stack walk is performed. By using *Assert*, *Deny*, or *PermitOnly*, which can be set with both the declarative and imperative syntax, the stack walk is finished before it reaches the end of the stack. When CLR comes across an *Assert* during a stack walk, it finishes with a Succeed. If it encounters a *Deny*, it is finished with a Fail. With the *PermitOnly*, it succeeds only if the checked permission is the same or a subset of the permission defined with the *PermitOnly*. Every other demand will fail at the *PermitOnly*.

À Custom permissions can be constructed and added to the runtime system.

## **Role-Based Security**

À Every executing thread in the .NET runtime system has a identity that is part if the security context, called principal.

À Based on the principal, role-based checks can be performed.

À Role-based checks can be performed in a declarative, imperative, and direct way. The direct way is by accessing the principal and/or identity object and querying the values of the fields.

## **Security Policies**

À A security policy is defined on different levels: enterprise, user, machine, and application domain. The latter is not always used.

À A security policy has permission sets attached that are built-in—such as FullTrust or Internet—or custom made. A permission set is a collection of permissions. By grouping permissions, you can easily address them, only using the name of the permission set.

#### 610 Chapter 12 • Security

À The important part of the policy are the security rules, called code groups; these groups are constructed in an hierarchy.

À A code group checks the assembly based on the evidence it presents. If the assembly's evidence meets the condition, the assembly is regarded as a member of this code group and is successively granted the permissions of the permission set related to the code group. After all code groups are checked, the permission sets of all the code groups the assembly is a member of are united to an actual permission set for the assembly at that security level.

À The CLR performs this code group checking on every security level, resulting in three or four actual permission sets. These are intersected to result in the effective permission set of permissions granted to the assembly.

À Remoting limits the extent to which the security policy can be applied. To create a secure environment, you need to secure remoting in such a way that access to your secured CLR environment can be fully controlled.

## Cryptography

À The .NET Framework comes with a cryptography namespace that covers all necessary cryptography functionalities that are at least equal to the CryptoAPI that was used up until now.

À Using the cryptography classes is much easier than using the CryptoAPI.

## Security Tools

À The .NET Framework comes with a set of security tools that enable you to maintain certificates, sign code, create and maintain security policies, and control the security of assemblies.

À Two comparable tools enable you to maintain code access security.

**Caspol.exe** (Code Access Security Policy Utility) has to be operated from the command-line interface. The .NET Configuration Tool comes as a snap-in for the Microsoft Management Console (MMC) and is therefore more intuitive and easier to use than **caspol.exe**.

#### Security • Chapter 12 611

**Q:** I want to prevent an overload of security stack walk, how can I control this?

**A:** This can indeed become a major concern if it turns out that the code accesses a significant number of protected resources and/or operations, especially if they happen in a long calling-chain. The only way to prevent this from happening is to put in a *SecurityAction.Assert* just before a protected resource/operation is called. This implies that you need a thorough understanding

of when a stack walk, hence demand, is triggered and on what permission this stack walk will be performed. By just placing an *Assert*, you create an uncontrolled security hole. What you can do is the following, which can be applied in the situation in which you make a call to a protected resource but do this from within a loop-structure. You can also use it in a situation in which you call a method that makes a number of calls to (different) protected resources/operations that trigger the demand for the same type of permission.

The only way to prevent a number of stack walks is to place an imperative assertion on the permission that will be demanded. Now you know that the stack walk will be stopped in its tracks. To close the security hole you just opened, you place an imperative demand for the permission you asserted in front of the assertion. If the demand succeeds, you know that in the other part of the calling-chain everything is OK in regard to this permission. And because nothing will change if you check a second or third time, you can save yourself from a lot of unnecessary stack walks. Think about a 1,000-fold loop: You just cleared your code from doing redundant 999 stack walks.

**Q:** When should I use the imperative syntax and when should I use the declarative?

**A:** First, make sure that you understand the difference in the effect they take. The imperative syntax makes a demand, or override for that matter, on part of your code. It is executed when the line of code that holds the demand/

## Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to [www.syngress.com/solutions](http://www.syngress.com/solutions) and click on the “Ask the Author” form.

### 612 Chapter 12 • Security

override is encountered during runtime. The declarative syntax brings these demands and overrides right into the metadata of the assembly. During the load phase of the assembly, the metadata is extracted and interpreted, meaning that the CLR already takes action on this information. If a stack walk takes place, the CLR can handle overrides much quicker than if they would occur during execution, thus the imperative way. However, demands should only be made at the point they are really necessary. Most of the time demands are conditional—think about whether the demand is based on a role-based security check. If you would make a demand declarative for a class or method, it will be trigger a stack walk every time this class or method is referenced, even if demands turns out to be not needed. So to recap: Make overrides declarative and place them in the header of the method, unless all methods in the class need the assertion; then, you place it in the class declaration. Remember that an assembly cannot have more than one active override type. If you cannot avoid this, you need to use declarative overrides anyway. Make demands imperative and place them just before you have to access a protected resource/operation.

**Q:** How should I go about building a code group hierarchy?

**A:** You need to remember four important issues in building a code group hierarchy:

\_ An assembly can not be a member of code groups that have conflicting permissions; for example, one with unrestricted *FileIOPermission* and one with a more restricted *FileIOPermission*.

\_ The bigger the code group hierarchy, the harder it is to maintain it.

\_ The larger the number of permission sets; the harder it is to maintain them.

\_ The harder it is to maintain code groups and permissions sets, the more likely it is they contain security holes.

Anyhow the best approach is the largest common denominator. Security demands simplicity with as few exceptions as possible. Before you start creating custom properties sets, convince yourself that this is absolutely necessary.

Nine out of ten times, one of the built-in permission sets suffices. The same goes for code groups—most assemblies will fit nicely in a code group based on their zone identity. If you conclude that this will not do, add only code

#### **Security • Chapter 12 613**

groups that are more specific than the zone identity, like the publisher identity, but still apply to a large group of assemblies. Use more than one level in the code group hierarchy only if it is absolutely necessary to check on more than one membership condition, hence identity attribute. Add a permission set to the lowest level of the hierarchy only and apply the Nothing permission set to the parent code groups.

Take into account that the CLR will check on all policy levels, so check if you have to modify the code group hierarchy of only one policy level, or that this has to be done on more levels. Remember: The CLR will intersect the actual permission sets of all the policy levels.

### **Solutions in this chapter:**

- \_ **Packaging Code**
- \_ **Configuring the .NET Framework**
- \_ **Deploying the Application**
- \_ **Deploying Controls**
- À **Summary**
- À **Solutions Fast Track**
- À **Frequently Asked Questions**

## **Chapter 13**

615

616 Chapter 13 • Application Deployment

# Introduction

The final stage in developing an application is preparing it for deployment. The first thing a user sees of your application is the installation. If problems arise during installation, the customer already has a negative perception of your application. Thankfully, deploying your application in Visual Basic .NET is simpler.

How many times have you heard customers say they installed your application and now something else doesn't work? Windows applications can get complicated with many DLLs needed and so many versions available. The .NET Framework will allow different versions of a component on the same computer. You don't have to worry about registration problems anymore.

Packaging your application can be as simple as copying all of the files into a common directory. Your application will be comprised of one or more assemblies. Because assemblies are self-describing, you don't need to do much. You don't have to worry about all the correct Registry entries being set and whether or not a version of your component(s) already exists on the computer. If you want your application to set up the Start menu or maybe even create an icon in the Quick Launch toolbar, you may want to package your application to be installed by the Windows Installer.

The complexity of configuring your application varies. If you use private assemblies, all you have to do is copy all the files to the same directory. If you want to use public assemblies or use different directories for some assemblies, you will need to create a configuration file. This is just an XML file that contains configuration information. It allows for easy backup and can be created on an application, user, or machine basis. It also allows for easier administration, because you only have to worry about a file, you don't have to concern yourself with the Registry.

Deploying your application can be as simple as copying the files from a CDROM or across the network to a directory on the user's computer. Because the assemblies in your application are self-describing and contain all needed references internally, when the user runs the application, it will search for these references itself. No more runtime errors about components not being registered.

However, this simple installation may not always be suitable for your needs. In this chapter, we cover how to install your Visual Basic .NET applications using the Windows Installer and creating Web downloads. When deploying controls, you need to take some additional factors into account. This chapter shows you how to get your applications ready to deploy.

## Packaging Code

The first step in getting your VB.NET application deployed is getting it packaged (although for the .NET Framework it does not matter in what language the application is written). Depending on the complexity of your application, it will consist of one or more DLL and/or EXE files, also called *portable executables*. You have to package them into one or more *assemblies*. An assembly consists of at least two and at most four parts:

- \_ **Assembly manifest** Mandatory because it contains the metadata that

the CLR needs to execute the code.

**\_Type metadata** Describes the types (class and methods) that are contained in the assembly.

**\_Portable executables** The actual IL code.

**\_Resources** Can be any type of nonexecutable file that needs to be used by code in the assembly.

Let's take a little closer look at the manifest because this is the "passport" of the assembly. By using the Intermediate Language Disassembler (ildasm.exe), you can see what is contained in an assembly. Figure 13.1 shows a part of the manifest of a sample that comes with the .NET Framework SDK. The part under *.assembly graphic* is interesting, not only because it states that the version (.ver) is 0.0.0.0, which is not allowed if you want to distribute your code, but it also lacks a public key, which is mandatory for sharing an assembly. What is actually missing is a strong name—a prerequisite in deploying. Let's set out to create an assembly that has a strong name so that we have a distributable package:

1. Use the strong name utility **sn.exe** to generate a key-pair in a file name

GrphKey.snk:

```
sn -k GrphKey.snk
```

2. Copy this file to a directory where it is easily accessible if you compile the program.

3. Add the necessary declarative statements to the code that take care of the generation of a strong name in the manifest of the assembly. They should be placed after the last **import** line and looks like this:

```
<assembly: System.Reflection.AssemblyVersion("1.0.0.1")>  
<assembly: System.Reflection.AssemblyKeyFile("GrphKey.snk")>
```

**Application Deployment • Chapter 13 617**

**618 Chapter 13 • Application Deployment**

4. Recompile the program and check the manifest, which will look something like Figure 13.2.

5. To be publicly available, it has to be placed in the general assembly cache, by using the General Assembly Cache utility tool (**gacutil.exe**).

Issue the following command:

```
Gacutil.exe -/i Graphic.dll
```

**Figure 13.1** Part of the Manifest from the Private Graphic.exe

**Figure 13.2** Part of the Manifest from the Public Shared Graphic.dll

**Application Deployment • Chapter 13 619**

6. **Gacutil.exe** returns with the message *Assembly successfully added to the cache*. Open Windows Explorer and go to the directory %WinDir%\Assembly. There you will find graphic.dll, ready for you to use (see Figure 13.3).

This does not mean that you cannot distribute an assembly that has no strong name; you can use it only for private use. If you try to add it in the general assembly cache, it will be rejected. After you have added a strong name to all the public shared assemblies and have set up all the private assemblies and other files that are needed for the application, you have to decide how you are going to package it to distribute. These are the two most commonly used methods:

**\_Creating Cabinet files** You can do this with the utility **makecab.exe**.

The advantage is that you compress the files, reducing the amount of data you have to distribute. Cabinet files are often used to download controls over the Internet using a Web browser.

**\_ Creating .msi files** You can use Visual Studio .NET to create MSI files for the deployment of your application.

**Figure 13.3** Listing the Public Assemblies Available in the General Assembly Cache  
620 Chapter 13 • Application Deployment

## Assembly Versioning

Versioning of executables has always been important. How often did you see a dialog box that told you that you needed a DLL with version 1.2.3456 or higher? Whatever you did, you had just one version of that DLL, and a program ran with it or broke. This has changed with the .NET Framework because you can have as many different versions of an assembly on your system as are available. You can have different applications that use an assembly with the same name, but with a different version. The version number has become more of a “compatibility number” and also controls the way the CLR locates the appropriate assembly. A version number must have the following structure:

Major.Minor[.Build[.Revision]]

This means that the **Major** and **Minor** are mandatory, and that **Build** and **Revision** are optional. However, if you want to use **Revision**, you must have a **Build**. The value of all the four parts can range from 0 to 65534 (included). This will be enough, especially with the speed in which Microsoft changes technologies.

As mentioned, the version number can also be regarded as a compatibility number, so a change in version number can reflect the following compatibility phases:

**\_ Compatible** If only the **Revision** number changes. Change of revision is seen as a quick fix engineering (QFE) update.

**\_ Possibly compatible** If the **Build** number has changed.

There is no guarantee for backward compatibility.

**\_ Incompatible** If **Minor** and/or **Major** changes.

Because you no longer need to be concerned with backward compatibility—because any version can be kept available—you need to take care to use proper versioning. The versioning helps the CLR in finding a compatible assembly. Let’s look at that process step-by-step:

1. The CLR reads the application configuration file, the machine configuration file and the publisher policy configuration file to determine what the correct version number is for the assembly that is referenced, and thus needs to be loaded.

**Continued**  
Application Deployment • Chapter 13 621

The publisher policy configuration file is omitted if the application configuration file has put the version resolving in Safe Mode.

2. If the correct version has been established, the CLR checks if this assembly has already been requested in the application domain. If that is the case, the already loaded assembly is used. Note that this check is based on the assembly's full name: name, version, culture and public key token (strong name). You can get in trouble if you have two assemblies with the same name, although one has the .dll and the other the .exe extension. After the .exe version is loaded and another assembly makes a reference to the .dll version, the CLR will conclude that that assembly is already loaded because the CLR makes the distinction based on the full name, and that does not include a file extension.

3. In case the assembly is not loaded yet and is a strong name base assembly (meaning that it can be a shared assembly), the CLR checks the Global Assembly Cache (GAC).

4. If the assembly is not located in the GAC, the CLR goes on a search mission:

- \_ It checks the configuration file if a <codeBase> is provided.

If so, this directory is checked for the presence of the assembly. In case the assembly is not located in the <codeBase> directory, the lookup fails.

- \_ If no <codeBase> is provided, the application base is checked.

- \_ If there is still no success, and the referenced assembly has a culture, the appropriate culture directories are checked.

(By now the CLR is getting pretty desperate).

- \_ It checks the privatePath directories and is satisfied with less than a full name.

Two final remarks on this subject: If you reference an assembly, but it does not supply all the fields of the full name, called a *partial reference*, the CLR will quickly decide that it found the right assembly, even if it turns out not to be the case. In this case, your assembly binds with the wrong assembly (version). With partial references, the CLR goes for a "best effort approach."

**Continued**

**622 Chapter 13 • Application Deployment**

## Configuring the .NET Framework

An important issue for deploying an application is to make sure that the installation process on a computer goes smoothly and that the application executes as intended. You should also remember how and where the CLR finds the right assemblies, chooses which assembly version to use; and how it sets security. The list goes on and on. Before .NET, you mainly needed the Registry to accomplish this. Now, you create the same information in XML-coded configuration files. Perhaps you thought when you first read about the .NET Framework and how it would free you from the hassles of the Registry that you would no longer have to be concerned with configuration issues. If so, think again! There are three types of configuration files—machine, application, and security—so you can finetune the settings according to the needs of administrators and developers.

### Creating Configuration Files

The configuration files, like nearly all other setting files within the .NET Framework, are XML-coded, adhering to a well-formed XML schema. In general, a configuration file consists of the following sections:

- \_ **Startup** Holds settings that are related to the CLR to use.
- \_ **Runtime** Holds settings that are related to the CLR working, especially how and where the CLR can find the proper assemblies.
- \_ **Remoting** Holds settings related to the remoting system.
- \_ **Crypto** Holds the settings related to the cryptography system.
- \_ **Security** Holds the settings of the security policy.
- \_ **Class API** Holds the settings related to the use of API.
- \_ **Configuration** Holds the settings that are used by the application.

Second, if an assembly has no strong name, (remember, this is only mandatory for the GAC), the CLR will not be checking on the correct version, even if you supply a version with the reference. Now the CLR finds itself thrown into a wild goose chase and again goes for the best effort approach.

A good rule of thumb is that you should always supply every assembly with a full name; it costs hardly any effort but makes it possible for the CLR to find the correct assembly and bind it.

#### **Application Deployment • Chapter 13 623**

Technically speaking, you can find or put any of these sections in any configuration file. However, if a section does not apply to the use of the configuration file, it will be ignored.

#### **NOTE**

Configuration files, especially machine and security, have an impact on the workings of all applications that make use of the .NET Framework. Be very careful with making changes to these files before assessing the impact they will have. When you deploy a new application, you should not assume that certain modifications to general configuration files can be made to suit your application needs. These changes may influence the working of other applications.

A second warning about protecting your configuration files: Because they are so readable, making changes to them is easy. Persons with ill intent that have access to these files can do a lot of harm. Be sure that you limit the access to these files and make them at least read-only to also prevent accidental changes.

## **Machine/Administrator Configuration Files**

Every machine that has the .NET runtime system installed has a machine configuration file named machine.config. You can find this file in the directory %CLR\_InstallDir%\config. This file is especially important to reflect the correct assembly binding policy of that machine. The file also holds the settings for remoting channels. The settings in the machine configuration file take precedence over those in any other configuration file and cannot be overridden by any other file. These settings are “etched in stone,” so to speak. The reason is obvious: The machine.config is expected to reflect the machine; any change to it may result in the breaking of the CLR. Nevertheless, you need to find the right balance between putting certain settings in the application configuration file or in the

machine configuration file.

As an example, take a look at an excerpt from the machine.config file regarding remoting:

```
<system.runtime.remoting>
<application>
</application>
```

#### 624 Chapter 13 • Application Deployment

```
<channels>
<channel id="http"
type="System.Runtime.Remoting.Channels.Http.HttpChannel,
System.Runtime.Remoting" />
<channel id="http server"
type="System.Runtime.Remoting.Channels.Http
.HttpServerChannel,
System.Runtime.Remoting" />
<channel id="tcp"
type="System.Runtime.Remoting.Channels.Tcp.TcpChannel,
System.Runtime.Remoting" />
<channel id="tcp server"
type="System.Runtime.Remoting.Channels.Tcp.TcpServerChannel,
System.Runtime.Remoting" />
</channels>
<channelSinkProviders>
<serverProviders>
<formatter id="soap"
type="System.Runtime.Remoting.Channels
.SoapServerFormatterSinkProvider,
System.Runtime.Remoting" />
<formatter id="binary"
type="System.Runtime.Remoting.Channels
.BinaryServerFormatterSinkProvider,
System.Runtime.Remoting" />
<provider id="wsdl"
type="System.Runtime.Remoting.MetadataServices
.SdlChannelSinkProvider,
System.Runtime.Remoting" />
</serverProviders>
</channelSinkProviders>
</system.runtime.remoting>
```

#### Application Deployment • Chapter 13 625

## Application Configuration Files

The application configuration file is located in the installation directory of the application and is named after the application's program executable with *.config* added to the name, thus *program.exe.config*. The CLR checks the application directory for that file. Because an application does not need its own configuration file, it can completely depend on the machine configuration file, but nothing will happen if it is not there. Take notice of this! If you put it somewhere else, or use a different suffix, the CLR will not find it, which may mean that the CLR is not able to load the application. In the case of a browser-based application, the HTML page should use a link element to give the location of the configuration file, which resides in a directory on the Web server.

The application configuration file is especially useful for assembly binding settings that relate to specific assembly versions an application needs and the places the CLR has to look for the application's private assemblies, called *probing*. A possible

configuration file for our earlier example of Graphic.dll may look like this:

```
<configuration>
<runtime>
<assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
<probing privatePath=".\\SubDir1;.\SubDir2"/>
<publisherPolicy apply="no"/>
<dependentAssembly>
<assemblyIdentity name="Graphic"
publicKeyToken="83f879e949c242e1"
culture=""/>
<publisherPolicy apply="no"/>
<bindingRedirect oldVersion="1.0.0.0"
newVersion="1.0.0.1"/>
</dependentAssembly>
</assemblyBinding>
</runtime>
</configuration>
```

This sample configuration shows the use of probing, telling the CLR that it's private assemblies reside in the directories *SubDir1* or *SubDir2*, which are subdirectories of the application directory. It also shows the use of *binding redirection*—

#### 626 Chapter 13 • Application Deployment

applications that want to bind with version 1.0.0.0 of Graphic.dll can bind with version 1.0.0.1 instead, without getting into compatibility problems.

The line **<publisherPolicy apply="no"/>** is worth mentioning. It refers to a publisher policy. A publisher policy file is a special kind of configuration file. It can be issued by the publisher and holds compatibility information regarding a fix or update of an existing component. It is used to let an assembly bind in a proper way with a new version of a component. This publisher policy configuration file resides in the assembly of a shared component. In our example, there can be a publisher policy in the new assembly, version 1.0.0.1. Note that the information in the publisher policy *always* overrides the settings in the application configuration file. The only way to stop this is, as the example shows, to put **<publisherPolicy apply="no"/>** in the application configuration file. This is called *Safe Mode* and is only valid for the assembly it is part of.

## Security Configuration Files

Security configuration files describes the security policy settings. There are at least three security configuration files applicable:

\_ **Enterprise** Resides in the directory %CLR\_InstallDir%\Config and is called Enterprise.config.

\_ **User** Resides in the directory %USERPROFILE%\Application Data\Microsoft\CLR security config\x.x.xxxx (x.x.xxxx is the build number) and is called Security.config.

\_ **Machine** Resides in the directory %CLR\_InstallDir%\Config and is called Security.config.

What these configuration files do and how they are used is described in Chapter 12. For the purpose of the example, take a look at some edited code from the user Security.config file, listing the modifications that were made to it in the exercises in Chapter 12:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
```

```

<mscorlib>
<security>
<policy>
<PolicyLevel version="1">
<NamedPermissionSets>

```

### Application Deployment • Chapter 13 627

```

<PermissionSet class="NamedPermissionSet" version="1"
Name="PrivatePermissions"
Description="My Private Permission Set">
<IPermission class="EnvironmentPermission" version="1"
Read="USERNAME;TEMP;TMP"/>
<IPermission class="FileDialogPermission" version="1"
Unrestricted="true"/>
<IPermission class="FileIOPermission" version="1"/>
<IPermission class="IsolatedStorageFilePermission" version="1"
Allowed="AssemblyIsolationByUser"
UserQuota="9223372036854775807"
Expiry="9223372036854775807"
Permanent="True"/>
<IPermission class="ReflectionPermission" version="1"
Flags="ReflectionEmit"/>
<IPermission class="RegistryPermission" version="1"
Read="HKEY_LOCAL_MACHINE"/>
<IPermission class="SecurityPermission" version="1"
Flags="Assertion, Execution,
RemotingConfiguration"/>
<IPermission class="UIPermission" version="1"
Unrestricted="true"/>
<IPermission class="DnsPermission" version="1"
Unrestricted="true"/>
<IPermission class="PrintingPermission" version="1"
Level="DefaultPrinting"/>
<IPermission class="EventLogPermission" version="1">
<Machine name="." access="Instrument"/>
</IPermission>
<IPermission class="MessageQueuePermission"
version="1" Unrestricted="true"/>
</PermissionSet>
</NamedPermissionSets>

```

### 628 Chapter 13 • Application Deployment

```

<CodeGroup class="UnionCodeGroup" version="1"
PermissionSetName="Nothing"
Name="PrivateGroup_1"
Description="">
<IMembershipCondition class="SiteMembershipCondition"
version="1" Site="msdn.one.microsoft.com"/>
<CodeGroup class="UnionCodeGroup" version="1"
PermissionSetName="LocalIntranet"
Name="PrivateGroup_2"
Description="">
<IMembershipCondition class="PublisherMembershipCondition"
version="1"
X509Certificate="3082025A308201C702101DD1CB6CAEA347000491E0419A84A91E300D
06092A864886F70D0101040500305F310B30090603550406130255533120301E06035504
0A131752534120446174612053656375726974792C20496E632E312E302C060355040B13
25536563757265205365727665722043657274696669636174696F6E20417574686F7269
7479301E170D3031303331353030303030305A170D3032303331353233353935395A3081
80310B3009060355040613025553311330110603550408130A57617368696E67746F6E31
10300E060355040714075265646D6F6E6431123010060355040A14094D6963726F736F66
7431153013060355040B140C456D6572616C6C42043697479311F301D060355040314166D
73646E2E6F6E652E6D6963726F736F66742E636F6D30819F300D06092A864886F70D0101

```

```
01050003818D0030818902818100BFD980FAD50DBC19919C765F2B80EB84B4336C0FE1CB
979B859AD13E9858276BC28F1B3CD82AC24B6205EFEF05F928AAE5DB45724B805BE97ACD
5334EE24F7BD18AC48B648B8FFBD5DCFF3D6362C1E3DB8514247C6D2069EBA5FA7EE09C9
8428D6EED261E250A80E74894BD36D70712F7FC019E8A40F17832659749FAB87F6B90203
010001300D06092A864886F70D0101040500037E007DFCF465F5BB7E171028D8D57C1A39
A9F630DE0F3C6F6924A6F5D50D31A096D26208957168E8F3E81BE6A4DD4B04BDD6DF8F22
63C309BE82D4B880CEAC5927BEB386D1DADA736C3F2432B15C7D3A1849BE564AA1B7F4DF
772FC8EE4A41236E0290130DDDE391E115C2103015CB3D4EB6AC91CC72F7F7F4E234E0C9
FA7B" />
</CodeGroup>
</CodeGroup>
</PolicyLevel>
</policy>
</security>
</mscorlib>
</configuration>
```

#### Application Deployment • Chapter 13 629

### WARNING

You should under no circumstance edit the Security.config and Enterprise.config files directly. It is very easy to compromise the integrity of these files. Always use the Code Access Security Policy utility (**caspol.exe**) or the .NET Configuration tool; these will guard the integrity of the files and will also make a backup copy of the last saved version.

## Deploying the Application

Although preparing the deployment of an application still calls for a lot of attention, things have become far more easy to handle with the .NET environment.

Many people assume that deploying is nothing more than a XCOPY of the application to the destination to get the application up and running. In essence this is true, but it assumes that you have created correct working configuration files, that the CLR is already installed, and that the application is self-contained (does not integrate with other applications). Remember we are still in the Beta phase of a new integrated application environment that gives the Microsoft Windows environment possibilities it never had before. Although the signs are good, we still have to wait for the final verdict until the first full-blown .NET applications are rolled out. To prepare yourself for deploying your first .NET application, we discuss a number of topics that can help you.

## Common Language Runtime

In order to run a VB.NET application on a system, it needs to have the .NET runtime environment. At this time it is very likely that a system does not have it installed. Up to the point where it becomes available, remember that you are still working with the Beta, and you will have to install it yourself. You need the .NET Framework Full version, available on the Visual Studio .NET Windows Component Update CD under the dotNetFramework directory with the name setup.exe and residing under the directory dotNETRedist. Installing this version enables you to run all .NET applications. There is also a limited runtime version available, called Control Version that can be used if you use only a Web browser download and run .NET controls.

### 630 Chapter 13 • Application Deployment

If (and how) Microsoft is going to deal with licensing and distribution of the .NET runtime environment is not known at this point. It is likely that the .NET runtime environment will become a default component of the Windows XP distribution because they put so much emphasis on the .NET Architecture.

## Windows Installer

Using the Windows Installer 2.0 to install a complex .NET application is highly recommended, because this Installer version can recognize assemblies and work with them accordingly. Some of these features are the following:

- \_ Adding and removing, and repairing if necessary, assemblies in the Global Assembly Cache
- \_ Installing and removing, and repairing if necessary, private assemblies in the application's directories
- \_ Rollback of failed assembly operations
- \_ Patching of assemblies

To be able to let the Windows Installer do all the work for you in a controlled way, you need to group all files that directly relate to the assembly. The Windows Installer handles such a group as a single component. If you uninstall the assembly, all files of the component will be uninstalled also. Because assemblies can be used by more than one application, you must prevent the Windows Installer from removing an assembly that is still in use. If you install all assemblies through Installer, this will be no problem because Installer keeps track of the Installer components reference an assembly. It will only remove an assembly if all the components that reference that component are previously removed.

Here is where you should start paying extra attention. Suppose you added a few assemblies to the cache using **gacutil.exe** and one of them references an assembly, let's call it Assembly X, installed by Windows Installer. Times goes by, and Assembly X is uninstalled, but because other assemblies installed by Windows Installer reference X, it was not removed. A bit more times goes by and the last assembly referencing Assembly X is uninstalled. This is noticed by Windows Installer, and it removes assembly X from the cache. The next time the assembly (which you manually installed) runs, it will make a futile attempt to bind to assembly X and fail.

If you are ever confronted with a .NET assembly that breaks, it most likely tried to bind to an assembly that is not available. You can use the Fusion Log

### Application Deployment • Chapter 13 631

Viewer (**fuslogvb.exe**) to examine where in the loading process things go wrong. By the way, the loading and binding ID is performed by a program called Fusion, hence the name of the viewer.

## CAB Files

You can create Cabinet files in a few ways. You can use **makecab.exe** or the deployment tool of Visual Studio .NET. The most important reason to use CAB files is that the compression can decrease the size of the file significantly, thus cutting down on the download time. When you create a CAB file you have to take notice of the following:

\_ A Cabinet file can contain only one assembly.

\_ The Cabinet file must have the same name as the file in the assembly holding the manifest. Take our first example, where we had the single file assembly `graphic.dll` that also holds the manifest. The Cabinet file has to be named `graphic.dll`. In a lot of cases, the assembly's name is equal to the name of this file holding the manifest.

After you have created the Cabinet files you deploy them, making them available for remote clients through a Web server. You can do this by referencing them through the following:

\_ A configuration file, using the `<codeBase>` tag

\_ A Web page, using the `<OBJECT>` tag

An example for the `<codeBase>` may look like this:

```
<configuration>
<runtime>
<assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
<dependentAssembly>
<assemblyIdentity name = "graphic"
publicKeyToken="83f879e949c242e1"
culture=""/>
<codeBase version="1.0.0.1"
href="http://www.company.com/CABS/Graphic.cab"/>
</dependentAssembly>
</assemblyBinding>
```

### 632 Chapter 13 • Application Deployment

```
</runtime>
```

```
</configuration>
```

An example for the `<OBJECT>` may look like this:

```
<HTML>
<OBJECT
codebase="CABS/Graphic.CAB#version=1,0,0,1">
</OBJECT>
</HTML>
```

A final remark on the use of CAB files: The first reference to the CAB file will extract the assembly and load it. However, subsequent references to the assembly will fail because the CLR does not automatically expand Cabinet files.

## Internet Explorer 5.5

Internet Explorer 5.5 and above can help you in deploying assemblies. You can reference a managed executable, hence assembly, from a Web page and it will be downloaded and executed. You can embed this in an installation guide that embodies all information that is needed to install the application.

Some issues surrounding this method of deployment are related to security policy and application domain. If you load an assembly from a Web site, the zone of the Web site, which is *Internet*, is used as evidence to determine the permission set. If the administrator did not make changes to be more specific with permissions, for example *site* or *strong name*, then these assemblies are assigned the limited permission set of *Internet*. To establish a broader permission set, you have to create appropriate Code Groups (see Chapter 12).

If you reference a Web site, the runtime creates an application domain for that site, for example `www.company.com`. If an application domain already exists for that site, the assembly referenced by this page will be added to that `AppDomain`.

As long as the Web site holds assemblies that belong to one application, assemblies sharing one application domain are not a problem. However if the site holds more than one application, for example <http://www.company.com/app1> and <http://www.company.com/app2>, this may be an unwanted situation. You can solve this by placing a <LINK> tag on the Web page that points to an application configuration file. Based on that configuration file, an AppDomain is created and all assemblies on that page run in this application domain context. This is also the case if more than one Web page has a <LINK> tag pointing to the same

#### **Application Deployment • Chapter 13 633**

application configuration file. Having mentioned this, a warning is in order: If you forget to supply the <LINK> tag to a page, the assemblies of this page are loaded in the site-related application domain. This will ultimately result in problems with the execution of the application.

In the “CAB Files” section earlier in the chapter, the use of the <OBJECT> tag was discussed. However, there are two other ways of referencing to an assembly that gets downloaded and loaded into a new application domain:

\_ Using a HREF link, for example <A HREF=“Graphic.exe”>

\_ Pointing your browser directly to the assembly, for example by entering <http://www.company.com/EXECS/Graphic.exe>

In the first option, it is assumed that the assembly resides in the same directory that the Web page is pointing at. The CLR will also check this directory for a configuration file. It is also assumed that the application base (AppBase) is set to the directory the Web page is pointing at. In the case of the HREF link, this may be [www.company.com/App1](http://www.company.com/App1). In the example of the second option, this is [www.company.com/EXECS](http://www.company.com/EXECS). The AppBase is seen as the root directory from which the CLR searches for subsequent referenced assemblies.

#### **NOTE**

As you install Visual Studio .NET Beta, the installation process first wants to install a few “enhancements” before it installs the actual Visual Studio .NET. One of these enhancements is the Installation of Internet Explorer 6.0 (Public Beta). The reason is that this version of Internet Explorer has all the features to deal with advanced .NET applications. Take note that you must be very cautious with installing IE 6.0 on your regular system if you use it for tasks other than developing. Experience has shown that some e-commerce Web sites have problems with IE 6.0, hence are so well-tuned on IE 5.5. And because IE 6.0 is also still a Beta, it may contain some incomplete functioning code.

## **Resource Files**

A resource is a nonexecutable data file that is related to an assembly and packaged in a resource format, so it can be automatically deployed together with the assembly. A few examples of resource files would include Help text for the

#### **634 Chapter 13 • Application Deployment**

applications, icons, images, and sound-files. The most important advantages of resource files are as follows:

\_ Resources can contribute to a better localization of your application; by

using the Culture identifier of the assembly, the correct localized resource is picked.

\_ Resource files make access to different types of data files uniform by using resource objects.

\_ Resources can be part of an assembly file or compiled into a satellite assembly.

Localization has always been a hassle in selecting the correct localized files. Now this is handled by the CLR, presuming that you have packaged your resource files in assemblies. If the Culture field is not empty, also called *neutral*, the CLR checks the localization settings of the system and tries to find a referenced assembly with the correct culture. If this is not the case, it will fall back on the neutral version. This implies that when you use localized assemblies, you must always have a neutral assembly available. If no localized assembly exists that matches the localization setting of the system, the CLR always falls back on the neutral version. In case the CLR cannot find the neutral version, it throws an exception, and the loading process will fail. Also, for localized resource files, the principal of satellite assemblies is used. So what about satellite assemblies? They are called *satellite* because they do not contain any executable code. They must be in close contact with an assembly that does contain executable code and use the resources that are contained in the satellite assemblies. This concept is also referenced to as *hub and spoke*, where the executable assembly is the *hub* and the resource assemblies the *spokes*. This solution has the following advantages:

\_ You can very easily add other localized versions to the application by simply copying them to the correct directory.

\_ You have to deploy only the localized versions that will be used.

\_ Satellite assemblies can be replaced without having to change or (partially) recompile the application.

The only drawback is that you must test your application against every localized resource set.

#### Application Deployment • Chapter 13 635

### NOTE

The localized resource assembly has to follow a specific naming convention. It is preferred that you use the Culture field to identify the localization signature, although the CLR can also check the assembly name for the localization signature. The format is *//-CC*, what stands for two lowercase characters for the language, followed by a dash, followed by two uppercase characters for the country. These codes are standardized in the ISO 3166 standard. Be sure to use these codes because there are some exceptions to the rule. A few examples are the following:

\_ **en-UK** English as spoken in the United Kingdom

\_ **en-US** English as spoken in the United States

\_ **du-BE** Dutch as spoken in Belgium

\_ **fr-BE** French as spoken in Belgium

Let's take a look at how you create the resource files from the command line. The following code is a sample that comes with .NET Framework SDK and is located in `<SDK_InstallDir>\FrameworkSDK\Samples\tutorials\resourcesandlocalization\graphic\vb`.

The build.bat file, which is edited for the sake of readability, reads this way:

```
resxgen /i:un.jpg /o:Images.resx /n:flag
cd en
resxgen /i:en.jpg /o:Images.en.resx /n:flag
cd ..\en-us
resxgen /i:en-US.jpg /o:Images.en-US.resx /n:flag
cd..
resgen Images.resx Images.resources
resgen en\images.en.resx en\images.en.resources
resgen en-us\images.en-us.resx en-us\images.en-US.resources
al /out:en\Graphic.resources.dll /c:en
/embed:en\Images.en.resources,Images.en.resources,Private
al /out:en-us\Graphic.resources.dll /c:en-US
/embed:en-us\Images.en-US.resources,Images.
en-US.resources,Private
```

### 636 Chapter 13 • Application Deployment

```
vbc /target:library /optionstrict+ /r:System.DLL
/r:System.Drawing.DLL
/r:System.Windows.Forms.DLL /r:System.Data.DLL
/res:Images.resources,Images.resources graphic.vb
```

You see three commands that you most likely never encountered before:

\_ **Resxgen** A utility that converts a JPG image into a RESX file. The latter is an XML-coded resource file. You need this utility to be able to generate a resource file for the JPG-image.

\_ **Resgen** The command line Resource Generator utility that converts the RESX file to a .resource-file. In fact, it can convert an input file with the .txt, .resource, or .resx extension to an output file with a .txt, .resource, or .resx extension, assuming that the file extension represents the format of the file.

\_ **Al** The Assembly Generation utility, but *al* is short for assembly linker. It is used to generate an assembly (including a manifest) from one or more MSIL files (without a manifest) or resource files. Let's take a look at the parameters in our example:

\_ **/out** Gives the exact name of the output file.

\_ **/c** Gives the culture string that has to be attached to the output file.

By the way, */c* is short for **/culture**.

\_ **/embed** The full syntax is: **/embed[resource]:file[,name [,private]]**, whereby *file* the name of the resource file that has to be embedded in the output file, containing the manifest; *name* is the internal identifier that will be used in the assembly to reference the resource; **private** takes care that the assembly can only be used by the application it is meant for and therefore will not be visible for other assembles. In Figure 13.4 this all comes back in the line **.mresource private'Images.en-US.resources**.

Application Deployment • Chapter 13 637

## Deploying Controls

Up until now, we have discussed the deployment of assemblies, or managed code, in general. When you start deploying .NET controls, you come across additional issues that are similar to the issues you had to solve when you deploy ActiveX

controls, although you no longer have to bother with Registry settings and CLSIDs (as long as your controls do not interact with COM+ components). Let's first list the steps involved in the deployment process and then discuss them in further detail:

1. Obtain a X.509 Authenticode Certificate from a CA, such as Verisign, or use a Software Publisher Certificate (SPC).
2. License your .NET control.
3. Sign the .NET control assembly.
4. Package the .NET control in a CAB file.
5. Make an application configuration file.
6. Create the Web page that makes an <OBJECT> reference to your CAB file and a <LINK> reference to the application configuration file.
7. Test it.

Step 1 speaks for itself. For testing purposes, you can make use of the **makecert.exe** tool. Step 2 involves the always important issue of software licensing. The .NET Framework comes with a License Compiler utility (**lc.exe**), which creates a .licenses file that will be included in the assembly as a .resource file.

**Figure 13.4** The Manifest of the Resource Assembly en-US/Graphic.resources

**638 Chapter 13 • Application Deployment**

Step 3 can be done using the File Signing tool (**signcode.exe**). Although it is a command-line utility that requests a whole series of options to be filled in, you should locate the file with the Windows Explorer and double-click it. The Digital Signature Wizard will be started, which takes you through the whole process in an easy and straightforward way. You can check if the assembly is indeed signed by performing the following steps:

1. Use the Windows Explorer to locate the assembly you just signed.
2. Right-click the file and select **Properties**.
3. Select the **Digital Signature** tab (see Figure 13.5).
4. Select the Certificate in the **Signature List** and click on **Details** to view the Certificate. It should look familiar.

Steps 4, 5, and 6 have already been discussed. Step 7 should speak for itself—testing has always been something that does not get the highest priority. But it cannot be emphasized enough: Testing is a very important step in the development *and* deployment process of an application. It doesn't matter whether you're dealing with a large and complex application or a single control. Never cut back on testing—you earn it back in all the hours you don't have to spend on troubleshooting.

**Figure 13.5** The Proof of a Signed Assembly  
**Application Deployment • Chapter 13 639**

## Summary

With the .NET Framework application deployment has become easy again—no battles with DLL versions and Registry settings. You are able to focus on what is really important: going smoothly through the deployment phases. The first step in

deploying is *versioning*. This is sort of new from what you were used to. But now you can run multiple versions of the same assemblies next to each other. It is important that assemblies have correct version numbers so that assemblies can bind with correct assemblies. To achieve this, it is important that assemblies get full names, consisting of a name, strong name, culture (also known as locale or localization), and a version number. The use of full names is also necessary to share them. After this is taken care of, you can go to the next step, which is the way the assemblies are going to be packaged for deployment. There are three choices: do not package them, just XCOPY them; package every assembly in a Cabinet file; or package the complete application in a Windows Installer file. The packaging method depends through which channels you want the application to distribute. Assemblies attribute also to the organization of data. All programs need different kinds of resources, such as Help files, images, and audio files. By converting these files into resource files, you can package them in assemblies without executable code. These are called satellite assemblies, which are very useful to solve deploying localized applications. By given them a culture identification, the CLR will automatically pick up the resource assemblies with the correct localization. Localized assemblies can be added without having to bring the application down or to restart it.

Although you do not need to get involved in all kinds of registering of DLL files, using Class Identifiers and other Registry settings, a lot of configuring is still going on. Only now, they are different types of XML-coded configuration files. Every application can have its application configuration file; every machine has its machine configuration file (also called an administrator configuration file) and security configuration files. Every possible setting is controlled by these configuration files and enables you to go from a generic configuration to a tailor-made configuration.

Before you can actually deploy an application, you must be sure that the machines that run the application indeed have the .NET Framework runtime installed. When you use Windows Installer, it can take a lot of work out of your hand installing, and also uninstalling, of the application, especially because Windows Installer recognizes assemblies and knows how to handle them, such as installing them in the general assembly cache. CAB files are especially useful if

#### **640 Chapter 13 • Application Deployment**

you want to distribute the assemblies using the Internet Explorer 5.5 (and up).

When you want to deploy .NET Controls, which you can compare with ActiveX Controls, issues such as licensing and signing are involved.

After deployment of an application, you do not have to reboot the system, and you can easily replace assemblies, just by adding a new version of the assembly to the assembly cache. You can modify configuration files, and the next time an assembly is loaded by the CLR, the modified configuration file is used.

## **Solutions Fast Track**

### **Packaging Code**

À An assembly has a Manifest that describes in detail what is wrapped in the assembly and with which other assemblies it will bind.

À An assembly is identified by its full name that consist of a name, string name, version number and culture (localization identifier).

À A single assembly can be packaged in a Cabinet (CAB) file, using the Cabinet Maker (**makecab.exe**).

À A complete .NET application can be packaged in a Windows Installer file.

## Configuring the .NET Framework

À Application dependent settings can be set in the application configuration file. This file applies to all assemblies that are part of an application.

À Machine dependent settings can be set in the machine/administrator file. This file applies to all applications that run on this machine. Settings in the application configuration file can not override the settings in the machine configuration file.

À There are three security configuration files: enterprise, which applies to one or more systems and applications; user, which is related to the user that starts up an application (every user has his own security configuration file); and machine, which applies to a single machine that runs one or more applications (every machine has its own security configuration file).

### Application Deployment • Chapter 13 641

## Deploying the Application

À Before a .NET application can be deployed, the .NET Framework runtime system must be installed on all the machines that will run (a part of) the application.

À With Windows Installer 2.0, a complete .NET application can be installed and, if needed, be uninstalled later on. Windows Installer 2.0 is fully assembly-aware and can take care of placing and updating assemblies in the general assembly cache. Windows Installer 2.0 keeps an independent administration of assemblies it has installed and are referenced by other assemblies installed by Windows Installer 2.0; it will never remove a uninstalled assembly if it is still referenced by other assemblies.

À With CAB files, you not only compress an assembly significantly, but it can also be deployed using Internet Explorer 5.5 (or higher).

À The nonexecutable data files—such as Help files, audio files, and images—that are used by assemblies are converted in resource files (using the Resource Generator utility **resgen.exe**) that can be added to assembly files that hold executable code. They can also be compiled into an assembly without executable code, using the Assembly Generation utility (**al.exe**). This is called a satellite assembly, which are very useful in deploying localized applications.

## Deploying Controls

À You can protect your controls by licensing them, using the License Compiler utility (**lc.exe**).

À You can authenticate your controls using a X.509 Authenticode Certificate, with the File Signing utility (**signcode.exe**), that comes with a Digital Signature Wizard.

## 642 Chapter 13 • Application Deployment

**Q:** If I can XCOPY a .NET application to its destination, why should I use Installer instead?

**A:** You don't need to use Installer; XCOPY will work just fine. But when you need to deploy a more complex application, you may want to do a little more than just an XCOPY. The most obvious is that you need to install shared assemblies in the general assembly cache (GAC). Additionally, if you want to uninstall the application, removing the sub tree will not remove assemblies from the GAC. Besides how do you know if you can safely remove an assembly from the cache without breaking another application? If you install all your application with Windows Installer, Installer will take care of removing an assembly when it is safe to. That is, if you install all the applications using Installer 2.0. It is a good practice to be able to install an application with an as small as possible footprint *and* uninstall applications without leaving a footprint. Packaging your application with Windows Installer takes all that work away from you and the people who are responsible for the installation and maintenance of the application. By the way, it looks far more professional than copying files from a CD.

Another reason to use Windows Installer 2.0 is that you may have to integrate with applications that run outside the .NET Framework or even ship unmanaged components with the application, that still have all the installation issues attached to it. It will help you a lot to keep using just a single method of deployment and Windows Installer 2.0 is not a bad choice to do so.

The XCOPY example is merely used as to imply that .NET applications are no longer plagued with the hassles of checking for the proper DLL version, using RegSvr32 to get your files registered, fixing CLSID problems and all these other annoyances. On the other hand, you're not out of the woods right now! All these problems still apply dealing with the .NET Framework runtime system and when there become more versions of this runtime system available you may have to deal with all the problems all over again, because

## Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to [www.syngress.com/solutions](http://www.syngress.com/solutions) and click on the "Ask the Author" form.

**Application Deployment • Chapter 13 643**

application may need a specific runtime version to run in. If Microsoft is not able to make it possible to run different CLR's next to each other and independent from each other, we may have to deal with all the issues all over again.

**Q:** What is the best way to deploy a localized application?

**A:** The best part of localization and the .NET Framework is that it uses the systems "Regional Options" to determine the "Culture", in which the culture is a combination of language (two lowercase characters) and country/region (two uppercase characters). Compilation of .NET applications is language

independent and no longer poses the same localization problems. Because culture is part of the assembly's full name, assemblies with a culture set always take precedent over assemblies that have no culture, called neutral. The Culture property at least has to hold the language, but you are advised always to be as specific as possible. This may help you to deploy your application in as many localization settings as possible, without having to replace files later on. Although the .NET Framework helps you a lot with easy localization implementation, it is the program's design and development that decides to what extent your application is able to handle localization. For example, address labels have another format in about every country. The .NET Framework will not help you to solve that problem—that's up to the program. But if you create your code so that it can format an address label based on a set of rules, you can save these rules as a resource in a localized assembly. Your code just has to read the rules, because the CLR takes care that the assembly with the proper localization is presented to the application. Besides setting the culture of the assembly, put the culture in the name of the resource (for example, Graphic.en-US.resources) because the CLR will use this to determine the proper localized resource in case the Culture property is not set. The last thing you should do is place all resources of the same culture in a subdirectory that should be located directly under the application's root directory and must be named with the exact culture's name. The CLR will check this directory to locate the proper localized resource. If you adhere to these things, adding a new language to your application is as easy as copying a directory.

#### 644 Chapter 13 • Application Deployment

**Q:** How do I transfer an image file to a resource file? Using **resgen.exe** doesn't work.

**A:** This seems to be an odd problem: You need to use **resgen.exe** to create a resource file, only it accepts just two different formats—TXT files and RESX files. The latter is an XMLized version of a resource. In fact, it is advised to convert every resource into its XML-coded version. The reason is that the whole .NET Framework is optimized in using XML-based input. You can use streams to take care of all the formatting and stuff. That leads to the problem that **resgen** is not able to convert an image to a RESX file. Let's hope that Microsoft comes up with a more versatile **resgen** that can handle more types of input. Until then, you can use a program called ResXGen with the C source included that is able to do the trick. You can find it in the samples section, under tutorials\resourcesandlocalization, of the Framework SDK. You can use this .NET program or let the source help you to create your own conversion program. After you have run your image file through this program, you have the XMLized (RESX) version of your image that can be converted to a resource file using **Resgen.exe**.

**Q:** Is there a method to use version numbers for assemblies?

**A:** There is no standard method of keeping track of version numbers. But because the format of the version number is very clear, you have to go with

that. As long as you go with that format, you are in the clear. Change version numbers only if it is really necessary. The reason is very simple: The CLR considers every version number as a valid assembly and will check it as it tries to find the correct version of an assembly. The more assemblies there are to check, the longer this will take. For example: You have built a reasonably extensive application, and you are now on version 2.0 (the major and minor part of the version number) and want to change it to 2.1. But because you've done a great deal in the development process, only half of the assemblies have changed. If you would change the version number of all assemblies, you double the number of assemblies while half of them are the same. Only by removing the obsolete ones can you control the number of active assemblies. This is no problem if the assemblies are used only by your own application, but if you share the assemblies with other programs, this is not so straightforward. There are ways around it. The best way is to edit the application configuration file and put a <assemblyredirect> in for all the unchanged assemblies and let the old versions point to the new one. Be careful with using publisher

#### **Application Deployment • Chapter 13 645**

policies to take control of these issues. Applications can run in Safe Mode, thereby ignoring the publisher policy. As far as the build and revision goes, change these only on a per-assembly basis.

There is another way of dealing with a lot of assemblies with version changes by going for a select number of large assemblies, instead of with a lot of small ones. This makes good sense if you can group a lot of code based on a common functionality in relation to deploying such functionality. Be sure that it has a limited number of exposed interfaces, or you will run into version trouble again if the assembly is referenced all over the place.

#### **Solutions in this chapter:**

- \_ **Considerations Before Upgrading**
- \_ **Considering Architecture Before Migration**
- \_ **Data Types**
- \_ **Converting VB Forms to Windows Forms**
- \_ **Keyword Changes**
- \_ **Programming Differences**
- \_ **Understanding Error Handling**
- \_ **Data Access Changes in Visual Basic .NET**
- \_ **Upgrading Interfaces**
- \_ **Using the Upgrade Tool**

- À **Summary**
- À **Solutions Fast Track**
- À **Frequently Asked Questions**

# Chapter 14

647

648 Chapter 14 • Upgrading Visual Basic Applications to .NET

## Introduction

Now that we have seen how to develop an application with Visual Basic .NET, what about your existing applications? Do you leave them as is, or should you upgrade them? There are different factors to consider when making this decision. This chapter will focus on when to upgrade your application, and what is involved in an upgrade when you decide to do it.

The .NET architecture is different from previous versions of Visual Basic and some applications will require significant changes. Visual Basic .NET has transitioned to making everything an object, prompting significant programming changes. You will have to modify data types to match the new Common Type System (CTS), which will impact applications using the Variant data type. There are other considerations. Some keywords have been changed or even removed, and your Visual Basic Forms will need to be upgraded to Windows Forms. Error handling will be completely different as well, and you will have to convert all existing error handling to the new exception-based format.

As you have seen, data access has changed drastically and is based on XML. This is a major paradigm shift, and addressing these issues can take considerable time and effort. Interfaces and events have changed from previous versions of Visual Basic. Not all applications use these features, but you need to understand how to convert them if they do. Fortunately, Microsoft has created an upgrade tool. This will attempt to upgrade your application from Visual Basic 6.0 to Visual Basic .NET. Of course, not all aspects of your application can be automatically upgraded. The portions of your application that cannot be upgraded will be commented out and will require manual conversion. Some applications will still necessitate a large development effort, so think long and hard before an upgrade and develop a plan for it.

## Considerations Before Upgrading

Certain issues must be addressed before a legacy application can be upgraded to Visual Basic .NET. Changing your application could mean anything from changing the data type of a variable to rewriting an entire application so it uses a different data access mechanism. The following points are recommendations you should carefully look at before deciding on migration. We will cover these recommendations in detail in the sections that follow:

- \_ Early binding of variables
- \_ Avoiding Null Propagation
  
- \_ Using ADO
- \_ Using the Date data type to store dates

\_ Using constants instead of actual values

## Early Binding of Variables

Visual Basic .NET, like Visual Basic 6.0, supports late-bound objects. *Late binding* an object is the practice of declaring a variable to be of the data type *Object* and assigning it to an instance of a class at runtime. Early binding also refers to the practice of declaring a variable a specific data type other than type *Object*. The advantage of early binding is that compiler errors, like using incorrect properties or calling non-existent methods, can be immediately detected. This is made possible by using type library. Whenever you set a reference to a type library, all the classes that are part of the component are available at design-time itself.

The usage of late binding objects has its disadvantages. The main drawback is the inability to enumerate the type library during design-time. During an upgrade process, late-bound objects can cause problems because of changes in property names and the removal of the default property feature from the controls. This is especially true if your legacy applications use code that late binds Windows controls, like labels and command buttons.

The reason is that some of the property names have changed in Visual Basic .NET, the most important being the name change for the Caption property. It is now called Text. Suppose you had a form with the CommandButton control on it. The following Visual Basic 6.0 code declares a variable of data type *Object*, assigns a command button object to it, and sets a value to its *Caption* property:

```
Dim obj as Object
Set obj = Me.Command1
obj.Caption = "Ok"
```

The upgrade tool normally converts all references to the Caption property to Text property. Since the preceding code uses late binding, the upgrade tool cannot determine what type of object is being referenced here. Therefore, it won't have a clue as to how the properties must be translated. As a result, the upgrade tool marks these statements with an upgrade error. In such cases, you will have to change the code yourself.

In order to ensure a successful migration, make sure all variables are declared a specific type other than Object. The following code will migrate successfully:

**Upgrading Visual Basic Applications to .NET • Chapter 14 649**  
**650 Chapter 14 • Upgrading Visual Basic Applications to .NET**

```
Dim objCmd as CommandButton
Set objCmd = Me.Command1
objCmd.Caption = "Ok"
```

The practice of using late binding also affects Visual Basic 6.0 components that implement classes and interfaces. In Visual Basic 6.0, it is possible to assign an interface reference to a variable of type *Object* to access the properties and methods of the interface. The following code shows this implementation:

```
Dim IMyInterface as Interface1
Dim clsMyClass as Class1 'Assuming Class1 implements Interface1
Dim obj as Object
.....
.....
Set IMyInterface = clsMyClass 'Gets a reference to the IMyInterface
Set obj = IMyInterface 'Assigns the IMyInterface reference to obj
obj.property1 = Value 'Accessing a property
```

Unfortunately, in Visual Basic .NET, you can only late bind to public members

of a class, not to interface members. Therefore, the previous code must be edited so the *object* variable directly references the class, not the interface, as shown next:

```
Set obj = clsMyClass 'Assigns the IMyInterface reference to obj
obj.property1 = Value 'Accessing a property
```

## Avoiding Null Propagation

Null propagation means that if *Null* is used in an expression, the resulting expression is always Null. In previous versions of Visual Basic, the *Null* value disseminated throughout the expression.

Null propagation is commonly used in database applications where you need to check for a Null in a specific field or fields. The following expressions show you how a Null is propagated in an expression:

```
Dim var
var = 100 + Null
var = "hello" & Null
```

### Upgrading Visual Basic Applications to .NET • Chapter 14 651

Null propagation is not supported in Visual Basic .NET. Consequently, the statement `var = 100 + Null` will result in a type mismatch error. Moreover, the **Null** keyword has been replaced with `System.DBNull.Value`. So, for the purpose of successful migration, your code should always test for Null instead of for Null propagation. Visual Basic uses the `IsDBNull()` function to determine if an expression contains a valid value or Null.

## Using ADO

Visual Basic .NET supports DAO, RDO, and ADO code, but with some slight modifications. Visual Basic .NET, however, does *not* support DAO and RDO data binding to controls, data controls, and the RDO user connection. So, if a data access application has DAO or RDO data binding, it is better to upgrade it to ADO before migrating to Visual Basic .NET. This section aims to give you an overview only. More detailed discussion will come later in the chapter.

It is possible to run existing data access applications that utilize ADO by using Visual Basic .NET with very minor modifications. In order to accomplish this, right-click the **Reference** node in the Solution Explorer and choose **Add Reference**. From the **References** window, choose **ADO library** from the supplied list of registered COM components.

What occurs next—behind the scenes—is quite elaborate. There is a tool called `TLBIMP.EXE`, which is shipped with the .NET Framework, that generates an assembly containing regular .NET metadata based on the content of the specified COM-type library. The following command imports the ADO object model into .NET:

```
Tlbimp.exe msado15.dll
```

On executing this command, the tool creates a file called `adodb.dll` in the current folder. The name of the output file can also be specified using the `/out:` option.

Once the library is imported, all the ADO classes are available to the .NET code as native classes. Using Visual Basic .NET, a typical data access application that fetches rows from a table can be coded as follows:

```
String strSQLConn = "Provider=SQLOLEDB;Initial" +
```

```
"Catalog=pubs;Server=localhost;UID=sa;PWD=";  
Dim objCn As New ADODB.Connection()  
Dim objRs As New ADODB.Recordset()
```

#### **652 Chapter 14 • Upgrading Visual Basic Applications to .NET**

```
objCn.ConnectionString = strSQLConn  
objCn.Open()  
objRs.Open("Select au_lname from authors", objCn,  
ADODB.CursorTypeEnum.adOpenForwardOnly,  
ADODB.LockTypeEnum.adLockReadOnly)  
While Not objRs.EOF  
Debug.WriteLine(objRs(0).Value)  
objRs.MoveNext()  
End While  
objRs.Close()  
objCn.Close()
```

## Using Date Data Type

In Visual Basic 6.0, you could use the Double data type to store and manipulate dates. This is not supported in Visual Basic .NET, however, because dates are not stored as doubles. Therefore, the following code is invalid in Visual Basic .NET:

```
Dim dblVal as Double  
Dim dtVal as Date  
dtVal = now  
dblVal = dtVal 'Invalid in Visual Basic .NET
```

The .NET Framework provides two methods that do the conversion between dates and doubles. The functions are `FromOADate` and `ToOADate`. The `ToOADate` function converts a Date type value to a double and the `FromOADate` converts a double value to Date.

During an upgrade operation, it becomes very difficult to determine what the code is trying to do when it uses double data type to store dates. In order to do away with unwanted changes to your code, use the Date data type to store dates.

## Using Constants

It is a good programming practice to use constants rather than the actual values that represent the constants or variables that store these values. In Visual Basic

#### **Upgrading Visual Basic Applications to .NET • Chapter 14 653**

.NET, the value of True has been changed from -1 to 1. The usage of constants ensures that the correct values are replaced when your project is upgraded. However, if an actual value is used, it is quite possible your project will be upgraded properly.

## Considering Architecture Before Migration

This section discusses what changes are to be made to your existing applications before moving to the .NET platform. The migration to the .NET platform has its own advantages. It is a quantum leap from the previous architectures and provides extended support for scaling applications. The key highlights are disconnected data access and resolution of the DLL hell problem by implementing a file-copy-based deployment of components. To take advantage of these benefits, your existing applications must be modified. Microsoft has provided a migration

tool that will make your applications .NET compatible. Not all applications can be readily modified. Certain projects that will remain the same because of lack of support in Visual Studio .NET.

The existing applications can be broadly classified into the following categories, which we will discuss in detail in the sections that follow:

- \_ Internet/Intranet Applications
- \_ Client/Server Applications
- \_ Single-tier Applications
- \_ Data Access Applications

## Intranet/Internet Applications

Visual Basic 6.0 provided the following project types to build Intranet/Internet applications:

- \_ Internet Information Server (IIS) Applications
- \_ DHTML Applications
- \_ ActiveX Documents

Each of these application types was unique in their way and helped developers build solutions best suited to the scenarios for which they were built. But, over a period of time, DHTML applications and ActiveX documents were used

### 654 Chapter 14 • Upgrading Visual Basic Applications to .NET

less and less because none of these application types were extensible. The introduction to Web forms and Web classes in the .NET architecture aims to increase application compatibility and enhance the functionality of Internet or intranet applications. Web forms are used to build feature-rich Web applications. Though the functional aspect of Web forms remains the same, it offers a variety of benefits. For example, the Web forms framework captures and stores information input on a form and makes it available as object properties. Web application services like these make Web forms unique. A separate section is devoted in this chapter to Web forms.

## Internet Information Server (IIS) Applications

A *WebClass* is the building block of an IIS application. It is a Visual Basic component, and as such, resides on a Web server, responding to input from the browser.

WebClasses, however, do not exist in Visual Basic .NET. The migration tool upgrades all WebClass applications to Web forms instead. As a result, migrated applications have to undergo some modifications before they are ready to run. It is also possible to navigate from a Visual Basic .NET Web form to a Visual Basic 6.0 WebClass.

## Web Forms in ASP.NET

Visual Basic .NET introduces an enhanced version of ASP (Active Server Pages) called ASP.NET, ushering in a new programming model to build powerful Web applications.

Web Forms is an ASP.NET technology useful in creating programmable Web pages. Listed next are some of the highlights of Web Forms:

- \_ Web Forms can be programmed using any of the Visual

Studio.NET languages, like C#, Visual Basic, and so on.

\_ Web Forms can run on any browser and render browsercompliant HTML.

\_ Web Forms support user-created and third-party controls.

\_ Web Forms support managed execution environments, type safety, inheritance, and dynamic compilation.

**Upgrading Visual Basic Applications to .NET • Chapter 14 655**

## DHTML Applications

DHTML applications typically house DHTML pages and client-side ActiveX DLLs that contain the business logic. DHTML applications cannot be upgraded to Visual Basic .NET.

## ActiveX Documents

ActiveX documents, like DHTML applications, cannot be upgraded to Visual Basic .NET. The only recommended option is to replace ActiveX documents with user controls. Despite this, both ActiveX documents and DHTML applications can interoperate with Visual Basic .NET Web forms.

### NOTE

Microsoft recommends implementing multi-tier architecture when building applications, so migration is easier. The architecture involves building the user-interface through ASP, and the business logic using Visual Basic 6.0 or Visual C++ 6.0 component. ASP is fully supported in Visual Basic .NET and the business components can either be upgraded to Visual Studio .NET or used as is.

## Client/Server and Multi-Tier Applications

Multi-tier projects typically house Visual Basic Forms, containing embedded user controls, and middle-tier business components. The middle-tier components can either be a Microsoft Transaction Server (MTS) component or a COM+ component. Client/Server applications contain just two layers: the user-interface layer (also called the client), and the database layer (called the server). The business logic is embedded in either the client-side or server-side.

Visual Basic Forms has been replaced with Windows Forms in Visual Basic .NET. The object model of Windows Forms is different from Visual Basic 6.0 Forms. The good news is that the object models are compatible. The Upgrade Wizard converts Visual Basic Forms to Windows Forms during an upgrade operation. User controls are then upgraded to Windows controls, however, custom property tags and accelerator key settings are not upgraded.

Middle-tier components can remain the same, but it's advisable to upgrade them to .NET as well. This begs the question of debugging, however. How can I

**656 Chapter 14 • Upgrading Visual Basic Applications to .NET**

debug a component written in Visual Basic 6.0 while in a Visual Studio.NET environment? Visual Studio.NET provides a single integrated debugger for all Visual Studio languages. The unified debugger goes beyond supporting components written for the .NET Common Language Runtime (CLR). It also supports debugging Win32 native applications and this includes MTS/COM+ components

written in Visual Basic 6.0. The only caveat is that they must be compiled to native code, with symbolic debug information, and should not include any optimizations.

Visual Basic .NET also introduces a new middle-tier component called Web Services. A Web Service is a component that contains business logic and is hosted by ASP.NET. They use HTTP methods as their transport mechanism and pass and return data using XML. The use of XML allows heterogeneous systems to interact with the Web Service, the only limitation being that Web Services does not support distributed transactions.

## Single-Tier Applications

Single-tier applications can be classified as:

\_ Add-ins

\_ Miscellaneous utility programs

Visual Basic .NET is now an integrated part of the Visual Studio .NET Integrated Development Environment (IDE). This has necessitated a change in the extensibility model used in Visual Studio .NET. Applications employing the Visual Basic 6.0 IDE model cannot be migrated to exploit the advantages of the Visual Studio .NET extensibility model. The new IDE extensibility model is generic for all project types supported in Visual Studio .NET. The add-ins that can be created using the new model can be shared by any of the Visual Studio .NET supported languages.

Miscellaneous utility programs like those which perform file operations or manipulate registry functions upgrade without any problems. The migrated applications can then take advantage of many of the new features available in Visual Basic .NET, like structured exception handling, free threading, and so on.

## Data Access Applications

Data Access applications typically use the following methods to perform data manipulation:

### **Upgrading Visual Basic Applications to .NET • Chapter 14 657**

\_ ActiveX Data Objects

\_ Remote Data Objects

\_ Data Access Objects

Visual Basic .NET introduces ADO.NET, an enhanced version of ADO. ADO.NET provides performance enhancements over ADO and aims at disconnected data. A separate section at the end of this chapter is devoted to highlighting the differences between the two.

DAO, RDO, and ADO applications can still be used in Visual Basic .NET after making some minor modifications. Visual Basic .NET doesn't support data binding to DAO or RDO controls. So if any of your applications use data binding, it is best to leave them to Visual Basic 6.0, or port the code to ADO before migrating to Visual Studio .NET. Data binding is still available in VB.NET and is implemented with the help of the *Binding* class.

## Data Types

Visual Basic .NET has not only brought in language enhancements but also

changed the way we work with data types. It is imperative the programmer have adequate information on the changes made to ensure a successful and smooth migration to Visual Basic .NET. This section is devoted to dealing with changes that have been effected in Visual Basic .NET.

## Variants

Variant is a special data type. What makes the Variant data type so unique is that it can be assigned to any primitive data type such as Empty, Nothing, Error, and Null. A primitive data type is one that is supported by the compiler natively. The only limitation with the Variant data type is that it cannot be assigned to fixedlength strings.

Visual Basic .NET uses the *Object* data type that effectively replaces the Visual Basic 6.0 Variant data type. In fact, the functionality of both *Object* and *Variant* data types has been combined into the new *Object* data type. The *Object* data type can be assigned to any primitive data type, *Empty*, *Nothing*, *Error*, *Null* and as a pointer to an object. The default data type in Visual Basic .NET is *Object*.

When a project is migrated to Visual Basic .NET, all variables of type *Variant* are converted to *Object*. It is a better programming practice to declare variables a specific data type before beginning the upgrade process. Not only does this help

### 658 Chapter 14 • Upgrading Visual Basic Applications to .NET

identify the type of data these variables will store, but it will also result in less ambiguous code after the upgrade has been completed.

## Integers

In Visual Basic 6.0, the Long data type is used to represent signed 32-bit numbers, while the Integer data type is used to store 16-bit numbers. This has been changed in Visual Basic .NET. In Visual Basic .NET, the Long data type is used to store signed 64-bit numbers, the Integer to store 32-bit numbers, and the Short data type to store 16-bit numbers.

The Short data type can store numbers between  $-32768$  to  $32767$ . You must use the Short data type in case the possible values for a variable fall between the specified upper and lower limits. It is possible to convert a Short data type to Integer, Long, and Decimal without an overflow.

The Integer data type can store numbers between  $-2,147,483,648$  to  $2,147,483,647$ . In the earlier versions of Visual Basic, the variable needed to be declared as Long in case you wanted to store large numbers. Now, in order to enhance performance of your applications running on a 32-bit processor, it is advised you use the Integer data type.

The Long data type, meanwhile, stores numbers between  $-9,223,372,036,854,775,808$  and  $9,223,372,036,854,775,807$ , each stored as an 8-byte number. Refer to the following Visual Basic 6.0 code, where:

```
Dim Ctr as Integer
```

```
Dim Total as Long
```

is upgraded to:

```
Dim Ctr as Short
```

```
Dim Total as Integer
```

## Dates

Visual Basic 6.0 and earlier versions used the Double data type to store dates. The Double data type uses four bytes to store the value. Visual Basic .NET, however, uses the Datetime data type, which is an 8-byte integer value. Since the representations are quite different in each of the versions, there is no implicit conversion between the Datetime and Double data types in Visual Basic .NET. The `ToOADate` and `FromOADate` can be used to convert between the Double and Visual Basic 6.0 representation of Date value. The upgrade tool inserts the

#### Upgrading Visual Basic Applications to .NET • Chapter 14 659

`ToOADate` or `FromOADate` method where a Double is assigned to a Date, as shown in the following code, where:

```
Dim dblVal as Double
Dim dtVal as Date
DblVal = dtVal
```

is changed to:

```
Dim dblVal as Double
Dim dtVal as Date
DblVal = dtVal.ToOADate
```

You can avoid calls to the `ToOADate` and `FromOADate` functions by using the Date data type to declare variables that store dates instead of using the Double data type. The OA in both the functions stands for OLE Automation compatible date format.

## Boolean

Boolean variables are stored as 32-bit numbers and can hold one of two values: True or False. True evaluates to 1, False to 0. The actual value translation is different that what it is in Visual Basic 6.0. In previous versions of Visual Basic, a Boolean value of True evaluated to -1 and a False evaluated to 0. Before upgrading, check your code to ensure you are not comparing Boolean variables to the hard-coded value -1 or 0, but to True and False.

## Arrays

In Visual Basic 6.0, it is possible to declare arrays with any lower or upper bound numbers. The **Option Base** statement is used to determine the lower bound number if a range was not specified in the declaration. Visual Basic 6.0 also allows use of the **ReDim** statement to reassign a variant to an array.

In order to maintain interoperability with other languages, arrays defined in Visual Basic .NET have a default lower bound of zero. This has made the **Option Base** statement obsolete. In Visual Basic .NET, a **ReDim** statement cannot be used unless the variable has been declared as an array. To illustrate, the following code is valid in Visual Basic 6.0, but invalid in Visual Basic .NET:

```
Dim x
ReDim x(20) 'Cannot use ReDim since x has not been declared as an array
```

#### 660 Chapter 14 • Upgrading Visual Basic Applications to .NET

Consider the following declaration:

```
Dim x(10) as Integer
```

In Visual Basic 6.0, the preceding code declares an array of 11 integers. Since arrays in Visual Studio .NET are zero-based, the previous declaration pronounces an array of 10 integers, from 0 to 9.

During the upgrade process, all **Option Base** statements are removed. All arrays that have their lower bound as zero are left as is while those that are nonzero-based are upgraded to an array wrapper class. For example, the following

Visual Basic 6.0 code

```
Dim arr(1 to 10) as double
```

is converted to:

```
Dim arr as object = new VB6.GetArray(GetType(Double), 1, 10)
```

There are a host of functions available in the wrapper class. This particular class is called *Microsoft.VisualBasic.Compatibility* and needs to be imported.

## Fixed-Length Strings

In Visual Basic 6.0, variables can be declared with fixed-length strings except for public variables in class modules. Fixed-length strings are not supported in Visual Basic .NET. If the application contains fixed-length strings, then the Upgrade Wizard uses a wrapper function to implement the functionality. This is shown in the following Visual Basic 6.0 code, where:

```
Dim fxString as String * 50
```

is converted to:

```
Dim fxString as new VB6.FixedLengthString(50)
```

This lack of support also means that changes have to be made if your applications employ User-Defined Types (UDT). User-defined types are called structures in VB.NET. Structures do not support primitive types. So, if your VB6 application contains the following UDT declaration:

```
Type EmployeeRecord  
EmpID as Integer  
EmpFirstName as String * 20  
EmpMiddleName as String * 10
```

### Upgrading Visual Basic Applications to .NET • Chapter 14 661

```
EmpLastName as String * 20
```

```
EndType
```

The upgrade tool will mark the statements containing fixed-length strings, telling you to initialize each fixed-length string. However, you can modify the fixed-length strings declarations to strings in the following manner:

```
Type EmployeeRecord  
EmpID as Integer  
EmpFirstName as String  
EmpMiddleName as String  
EmpLastName as String  
EndType
```

This declaration will ensure that the UDT is migrated to Visual Basic .NET without any changes. The same holds true for fixed size arrays. So, if you have the following array declaration:

```
Dim arrSample(12) as String
```

you can change it to:

```
Dim arrSample() as string
```

## Windows API Data Types

A majority of the Windows API functions can be used as they are in Visual Basic .NET. The only modification you will have to make is to change the data types accordingly. The upgrade tool does the following to all your API declarations:

\_ Changes all occurrences of Integer to Short. (i.e., Visual Basic 6.0 Integer

data type is now Short.)

\_ Changes all occurrences of Long to Integer. (i.e., Visual Basic 6.0 Long data type is now Integer.)

\_ Upgrades fixed-length string data types to a fixed-length string wrapper class.

The following Visual Basic 6.0 code displays the name of the logged in user on a Windows 2000 Server:

```
Public Declare Function GetUserName Lib "advapi32.dll" Alias  
"GetUserNameA" (ByVal lpBuffer As String, nSize As Long) As Long
```

#### 662 Chapter 14 • Upgrading Visual Basic Applications to .NET

```
Sub DisplayUserName()  
Dim strUserName As String  
strUserName = String(20, " ")  
Module1.GetUserName strUserName, 20  
MsgBox strUserName  
End Sub
```

After the upgrade, the code is transformed into:

```
Public Declare Function GetUserName Lib "advapi32.dll" Alias  
"GetUserNameA"(ByVal lpBuffer As String, ByRef nSize As Integer) As  
Integer  
Sub DisplayUserName()  
Dim strUserName As String  
strUserName = New String(CChar(" "), 20)  
Module1.GetUserName(strUserName, 20)  
MsgBox(strUserName)  
End Sub
```

Note the differences between the code written in Visual Basic 6.0 and that written in Visual Basic .NET:

\_ The nSize parameter in the Visual Basic 6.0 code is Long, whereas it is Integer in Visual Basic .NET.

\_ The data type of the return value has been changed from Long to Integer.

\_ The initialization of strings has been prompted.

## Converting VB Forms to Windows Forms

Visual Basic .NET has a new Forms package called Windows Forms. While Windows Forms is largely compatible with Visual Basic 6.0 Forms, there are some minor changes that need to be done. The following differences exist between Visual Basic 6.0 Forms and Windows Forms:

\_ Windows Forms does not support the *Form.PrintForm* method.

Therefore, you cannot print an image of the Windows Forms on a printer.

#### Upgrading Visual Basic Applications to .NET • Chapter 14 663

\_ Graphics commands like **Circle**, **Pset**, **Line**, **Point**, and **Cls** are not supported in Windows Forms. The Windows Forms package is built on a more feature-rich layer called GDI+.

\_ Windows Forms supports only twips as the unit of measurement in the *ScaleMode* property. If your Visual Basic 6.0 Forms used twips as the measurement, then it is upgraded correctly. There will be sizing issues if pixels were used as a unit of measurement.

\_ Visual Basic 6.0 supported any font type for forms and controls. But Visual Basic .NET supports only TrueType or OpenType fonts. If your existing application uses a non-TrueType font, they are changed to the default Windows Form font. All formatting is lost during this change, however. If you have formatted text, Microsoft recommends you use Arial instead of Visual Basic's default MS Sans Serif.

\_ In Windows Forms, the *MousePointer* property of the *Screen* object can be used only for forms inside the application.

\_ In Visual Basic 6.0, assigning a value of zero to the *Interval* property of the *Timer* object disables the Timer. Visual Basic .NET resets the value to one when a value of zero is assigned. To disable the timer, you should set the *Enabled* property to False. During an upgrade operation, if the tool detects a statement that assigns a value of zero to the *Interval* property, then the statement is commented with an upgrade error.

\_ Windows Forms does not support the *Name* property for forms and controls at runtime. Any Visual Basic 6.0 code that iterates through the Control Collection looking for a *Name* property will not be upgraded correctly.

\_ Windows Forms has two menu controls. They are *MainMenu* and *ContextMenu*. Visual Basic 6.0 has only one menu control called *Menu* that can be opened as a *MainMenu* or a *ContextMenu*. During an upgrade operation, *Menu* controls are upgraded to *MainMenu* controls. The *ContextMenu* controls, however, have to be explicitly re-created.

\_ Windows Forms does not support the OLE Container control. If your application has to use an OLE Container control, you can use the *WebBrowser* control as an alternative. During an upgrade process, an error is added to the upgrade report and an unsupported-control placeholder is inserted into the form.

#### **664 Chapter 14 • Upgrading Visual Basic Applications to .NET**

\_ Image controls are not supported in Visual Basic .NET. As a result, all *Image* and *PictureBox* controls are upgraded solely to *PictureBox* controls.

\_ The *clipboard* object in Visual Basic .NET has more functionality than the Visual Basic 6.0 *clipboard* object. The new *clipboard* object supports more formats than the previous *clipboard* object. As a result, any Visual Basic 6.0 clipboard code cannot be upgraded to Visual Basic .NET. All clipboard statements will be marked with an upgrade error.

\_ Windows Forms has no built-in Dynamic Data Exchange (DDE) support. DDE is a form of interprocess communication that uses a concept called shared memory to exchange data between applications. Since Windows Forms does not support DDE, you cannot use the *LinkMode* property available in Visual Basic 6.0 forms. During an upgrade operation, all DDE properties and methods are commented with an upgrade warning.

## **Control Anchoring**

If you are designing a form that the user might resize at runtime, you might want to make your controls resize and reposition correctly on the form. In order to resize controls correctly with the form, you should use the *Anchor* property. The *Anchor* property defines an anchor position for the controls on the forms. When a control is anchored on the form and the form is resized, the relative positions will be maintained after the resize. So, if a control's anchor position is set to *bottomright*, irrespective of how the form is resized, horizontally or vertically, the control will always be placed in the bottom-right corner. Without the *Anchor* property, the control's position is fixed and it will not retain its original position after it is resized.

The anchor position value can be chosen from one of the *AnchorStyles* enumeration values. The following statement anchors a Button Control to the topleft corner of the form:

```
Button1.Anchor = AnchorStyles.TopLeft
```

Table 14.1 lists various *AnchorStyles* enumeration values.

### Upgrading Visual Basic Applications to .NET • Chapter 14 665

#### Table 14.1 *AnchorStyles* Enumeration Values

Member Name	Description
-------------	-------------

All	Each edge of the control anchors to the corresponding edge of its container.
-----	--

Bottom	The control is anchored to the bottom edge of its container.
--------	--

BottomLeft	The control is anchored to the bottom and left edges of its container.
------------	--

BottomLeftRight	The control is anchored to the bottom, left, and right edges of its container.
-----------------	--

BottomRight	The control is anchored to the bottom and right edges of its container.
-------------	---

Left	The control is anchored to the left edge of its container.
------	--

LeftRight	The control is anchored to the left and right edges of its container.
-----------	---

None	The control is not anchored to any of the edges.
------	--

Right	The control is anchored to the right edge of its container.
-------	---

Top	The control is anchored to the top edge of its container.
-----	---

TopBottom	The control is anchored to the top and bottom edges of its container.
-----------	---

TopBottomLeft	The control is anchored to the top, bottom, and left edges of its container.
---------------	--

TopBottomRight	The control is anchored to the top, bottom, and right edges of its container.
----------------	---

TopLeft	The control is anchored to the top and left edges of its container.
---------	---

TopLeftRight	The control is anchored to the top, left, and right edges of its container.
--------------	---

TopRight	The control is anchored to the top and right edges of its container.
----------	--

## Keyword Changes

The following keywords have either been removed from Visual Basic or replaced

with a Visual Basic .NET specific version.

#### 666 Chapter 14 • Upgrading Visual Basic Applications to .NET

## Goto

The **Goto** statement is present in Visual Basic .NET only for the purposes of error-handling and can be used only with **On Error...Goto**. The **Goto** statement cannot be used to perform multiple-branching. Instead, use the **Select...Case** statement to perform multiple-branching.

## GoSub

The **GoSub** statement calls a procedure within a subprocedure. There is no support for the **GoSub** statement in Visual Basic .NET. Instead, method calls can be made using the **Call**, **Sub**, and **Function** statements.

## Option Base

Arrays in Visual Basic .NET always have a lower bound of zero. This has rendered the **Option Base** obsolete in Visual Basic .NET. The **Option Base** statement was used in Visual Basic 6.0 to determine the lower bound value of those arrays that were not declared with an explicit lower bound value.

## AND/OR

In Visual Basic 6.0, the AND, OR, XOR, and NOT operators perform both logical and bitwise operations depending on the expressions. In Visual Basic .NET, AND, OR, XOR, and NOT operators apply only to type *Boolean*. The AND and OR operator short-circuits evaluation if the value of the first operand is enough to determine the result of the operation.

Visual Basic .NET uses the new bitwise operators. They are `BitOr`, `BitAnd`, and `BitXor`. The new bitwise operators do not short-circuit.

The Upgrade Wizard upgrades an **AND/OR** statement which is non-Boolean or contains functions, methods, or properties that use a compatibility function with the same behavior as that in Visual Basic 6.0. For Boolean statements, it is upgraded to use the native Visual Basic .NET statement.

## Lset

The **Lset** statement can be used in two ways:

\_ It can assign a variable of one user-defined data type to another variable of a different user-defined data type. This is not supported in Visual Basic .NET.

#### Upgrading Visual Basic Applications to .NET • Chapter 14 667

\_ It can pad a string with spaces to make it a specified length. The `PadRight` method of the *string* class can be used to achieve this functionality.

## VarPtr

`VarPtr` is used to acquire the address of a variable or array element. It takes the variable name or an array element as the parameter and returns the address. Since Visual Basic .NET does not support this function, the upgrade tool does not upgrade this statement and therefore marks it with an upgrade error.

## StrPtr

Strings in Visual Basic are stored as BSTRs. If you pass a string variable to the VarPtr function, you will get the address of the BSTR which acts as a pointer to the string. To get the address of the string buffer, you need to use the StrPtr function. This function returns the address of the first character in the string. Because Visual Basic.NET doesn't support StrPtr, the upgrade tool reports an upgrade error when it encounters this statement.

## Def

The purpose of the **DefBool**, **DefByte**, **DefInt**, **DefLng**, **DefCur**, **DefSng**, **DefDbl**, **DefDec**, **DefDate**, **DefStr**, **DefObj** and **DefVar** statements is to set the default data type for those variables, parameters, and procedure variables whose names start with the specified character. To improve the readability and robustness of the code, Visual Basic .NET does not support these statements. Sample Visual Basic 6.0 code is illustrated in the following:

```
DefStr x-z
Sub Test
x = "hello world"
End Sub
```

is upgraded to:

```
Sub Test
Dim x as String
x = "hello world"
End Sub
```

668 Chapter 14 • Upgrading Visual Basic Applications to .NET

## Programming Differences

The programming differences between the previous versions of Visual Basic and Visual Basic .NET are many. They include changes in the way methods and properties are implemented, and the fact that a host of old features, like GoSub, are not supported. This section discusses the following differences:

- \_ Method implementation
- \_ Dealing with unmanaged code
- \_ Properties
- \_ Property name changes for some controls
- \_ Default properties
- \_ Null usage

### Method Implementation

Procedures and functions in Visual Basic .NET have undergone some changes, ranging from treatment of optional arguments to the latest feature of function overloading. Modifications regarding how methods are implemented can be discussed under the following headings:

- \_ Optional Parameters
- \_ Static Modifier
- \_ Return Statement
- \_ Procedure Calls
- \_ External Procedure Declaration
- \_ Passing Parameters

\_ParamArray  
\_Overloading

## Optional Parameters

In Visual Basic 6.0, you can declare a procedure parameter as optional without specifying a default value. If the optional parameter was of type Variant, the IsMissing function can be used to determine whether the optional parameter is

### Upgrading Visual Basic Applications to .NET • Chapter 14 669

present. This is something Visual Basic .NET can not do, since the IsMissing function is not supported there. But, in Visual Basic .NET, an optional procedure parameter must be declared with a default value. This value is then passed to the procedure if the calling program does not supply the optional parameter. The following declaration shows you how to code a procedure that accepts an optional parameter:

```
Function CheckBalance(ByVal strACNUM as string, Optional ByVal  
strACName as String = "")
```

You can easily check if the optional parameter was passed to a function or procedure with the help of the default value. In the procedure, check to see if the optional parameter contains the default value. If it has the same value, then the optional parameter has not been passed. If the optional parameter contains a different value than that of the default, then the parameter was passed. To successfully use this method, you must make sure the default value you assign is unique.

## Static Modifier

You can declare a procedure in Visual Basic 6.0 with the Static modifier. All variables inside the procedure are then treated as static variables. Static variables retain their value even after a function has finished execution. But in Visual Basic .NET, the Static modifier cannot be used when declaring procedures or functions. If you want to declare a variable as a static variable, you need to declare it explicitly.

## Return Statement

The functionality of the **Return** statement has changed in Visual Basic .NET. In Visual Basic 6.0, the **Return** statement can be used only to branch to a particular statement in the calling code. This is typically the next statement following the **GoSub** statement. In Visual Basic .NET, you use the **Return** statement to give back control to the calling program, which is done by coding the **Return** statement in a Function or Sub procedure. (It is important to note that the **GoSub** statement is not supported in Visual Basic .NET.)

The following Visual Basic 6.0 code illustrates how the **Return** statement is used to return control to the line following the **GoSub** statement. First, the **GoSub** statement transfers control to the Add subroutine. The Add subroutine then adds the two integers and transfers control to the line following the **GoSub** statement with the help of the **Return** statement. After the control is transferred, the result is displayed and the **Exit Sub** statement terminates the operation.

### 670 Chapter 14 • Upgrading Visual Basic Applications to .NET

(The **Exit Sub** statement is required to prevent the control from executing the add subroutine.)

```

Function AddInt() as Integer
Dim x As Integer
Dim y As Integer
Dim result As Integer
x = 10
y = 20
MsgBox result
Return
End Sub

```

The following subroutine demonstrates implementation of the **Return** statement in giving back control to the calling program. First, the subroutine checks the value of the denominator. If the value is zero, then the control is returned to the calling program after displaying an error message. This is done with the help of the **Return** statement. If the denominator has a value greater than zero, the resulting division is displayed:

```

Sub Divide(Byval x as Short, ByVal y as short)
If y = 0 then
Msgbox "Cannot Divide by zero"
Return 0
Else
Return x / y
End if
End Sub

```

## Procedure Calls

In Visual Basic 6.0, all function calls require parentheses around the parameter list. If you are invoking a Sub procedure, the parentheses are required if you use the **Call** statement. You cannot use parentheses when you invoke a Sub procedure

### Upgrading Visual Basic Applications to .NET • Chapter 14 671

without a **Call** statement. The following code fragment shows you the variations when a subroutine is called in Visual Basic 6.0:

```

Result = Add(10, 20)
Call FindDetails(aulname, aufname)
FindDetails aulname, aufname

```

In Visual Basic .NET, parentheses are required for any invocation of a Function or Sub procedure that contains parameters. The usage of a **Call** statement is optional. If a Sub or Function procedure does not contain any parameters, you can either choose to use parentheses, or leave them out altogether. The following code fragments illustrate how function calls are made:

```

Result = Add(10, 20)
FindDetails(aulname, aufname)

```

## External Procedure Declaration

In Visual Basic 6.0, you can reference an external procedure using a **Declare** statement. External procedures are typically API calls. In some cases, when a data type for a parameter is unknown, or a return value is unknown, you can use the **Any** keyword. The **Any** keyword allows you to pass any data type to parameters that have been declared of this type. This does not, however, involve type safety. Visual Basic .NET, on the other hand, does not support the **Any** keyword. So, if an external procedure contains a parameter that has been declared as *Any*, the Upgrade Wizard converts it to the *Object* data type. This was done to increase type safety and ensure consistency and interoperability between applications. You will have to declare parameters to be of a specific type before you can use them.

Declaring external procedures in Visual Basic .NET is done in much the same way as Visual Basic 6.0. The following is a Visual Basic 6.0 external procedure declaration:

```
Public Declare Function GetComputerNameW Lib "kernel32" (lpBuffer As Any, nSize As Long) As Long
```

In Visual Basic .NET, it will be changed to:

```
Public Declare Function GetComputerNameW Lib "kernel32" (ByRef lpBuffer As Object, ByRef nSize As Integer) As Integer
```

## 672 Chapter 14 • Upgrading Visual Basic Applications to .NET

### Passing Parameters

The default parameter passing mechanism in Visual Basic 6.0 is `ByRef`. This means that any change made to the parameter in the called program is reflected in the calling program. Of course, such passing mechanisms have their pros and cons. In Visual Basic .NET, on the other hand, the default parameter passing mechanism is `ByVal`. When parameters are passed `ByVal`, any changes made to the parameter values are effective only in the called function. The original values present in the calling function are not affected.

When an argument is passed as `ByVal`, a copy of the variables is passed to the called function. The advantage is that original values are retained. Arguments passed using `ByRef`, however, run the risk of being modified the called function or subroutine.

### ParamArray

In Visual Basic 6.0, you can designate the `ParamArray` parameter to be the last parameter to accept an array of parameters. This can be helpful when you don't know the number of parameters being passed to a procedure. In addition, you cannot explicitly specify the passing mechanism as `ByVal` or `ByRef` for parameters declared to be `ParamArray`. They are always passed as `ByRef`. This, unfortunately, cannot be changed. Likewise, the data type for `ParamArray` parameters can only be `Variant`. In Visual Basic .NET, on the other hand, the `ParamArray` parameters are always passed as `ByVal`, and need to be declared as the *Object* data type.

The following Visual Basic 6.0 code uses the subroutine called `AddToArray`. This subroutine receives the array, as well as the elements added to the array, and appends the list of values to it. Since the number of individual array elements cannot be determined in advance, the `ParamArray` parameter is used to pass multiple parameters:

```
Private Sub Command1_Click()  
    Dim x() As Integer  
    Dim ctr As Integer  
    Call AddToArray(x, 1, 2, 3, 4)  
    For ctr = 0 To UBound(x)  
        Debug.Print x(ctr)  
    Next
```

## Upgrading Visual Basic Applications to .NET • Chapter 14 673

```
End Sub  
Sub AddToArray(x() As Integer, ParamArray Values() As Variant)  
    Dim ctr As Integer  
    For ctr = 0 To UBound(Values)  
        ReDim Preserve x(ctr)  
        x(ctr) = Values(ctr)
```

```
Next
End Sub
```

The following code illustrates the same program written in Visual Basic .NET:

```
Protected Sub Button1_Click(ByVal sender As Object, ByVal e As
System.EventArgs)
Dim x() As Integer
Dim ctr As Integer
Call AddToArray(x, 1, 2, 3, 4)
For ctr = 0 To UBound(x)
system.Diagnostics.Debug.WriteLine(x(ctr))
Next
End Sub
Sub AddToArray(ByRef x() As Integer, ParamArray ByVal Values() As
Object)
Dim ctr As Integer
ctr = 0
ReDim x((ubound(Values)) + 1)
For ctr = 0 To UBound(Values)
x(ctr) = values(ctr)
Next
End Sub
```

#### 674 Chapter 14 • Upgrading Visual Basic Applications to .NET

Note the change to the ParamArray parameter data type from Variant to Object, as well as the change to the **ReDim** statement. A value of one is added to the Ubound function when the array is re-dimensioned. This is necessary because arrays in Visual Basic .NET are zero-based.

The sample program demonstrates the use of ParamArray. The AddToArray function accepts an array as the first argument, as well as a series of values to be inserted into the array. The second parameter is declared to be a ParamArray since the number of values vary depending on the situation, while the subroutine re-dimensions the array to accommodate the actual number of elements that can be stored there. When re-dimensioning the array, it is necessary to add one to the size since arrays in VB.NET are zero-based.

## Overloading

Overloading is a concept that has been in vogue for a number of years now. The C++ language introduced this feature as a means to achieve the same functionality, differing only in terms of the number or types of parameters. A new paradigm called function signature is closely related to function overloading. A function signature is nothing but a template used by a compiler to check when a call is made to the function or procedure. The signature consists of the following:

- \_ Name of the procedure
- \_ Number of parameters
- \_ Order of parameters
- \_ Data types of parameters

The following are *not* a part of a function signature:

- \_ Procedure modifiers like Private, Public, and so on
- \_ Parameter names
- \_ Parameter modifiers like ByRef and ByVal
- \_ Return values and types

Overloading a function involves changing at least one of the elements that form part of the signature. Overload procedures have the following features:

\_ Overloaded functions must differ only in their signatures, not on anything else.

#### **Upgrading Visual Basic Applications to .NET • Chapter 14 675**

\_ Functions differentiated by virtue of their signatures can have any procedure modifier or return type.

\_ It is possible to overload a Function procedure with a Sub procedure or vice-versa, provided their parameters are different. It is also possible to overload a property or method in a class.

Earlier versions of Visual Basic do not support the concept of overloading. As a result, you cannot have two or more procedures or functions with the same name that differ only in their arguments. Each procedure, even if they offer the same functionality, must be given a unique name. Thus, if you were writing a procedure to divide two values, and you want to offer variations in the arguments passed to these procedures, you have to write the procedures with the idea that each procedure should be different from the other, not only in name but in the arguments passed to it. The following code fragment illustrates this:

```
Function DivideInteger(i1 As Integer, i2 As Integer) As Integer
DivideInteger = i1 / i2
End Function
Function DivideLong(l1 As Long, l2 As Long) As Long
DivideLong = l1 / l2
End Function
```

The two functions, DivideInteger and DivideLong, offer variations on dividing two integers or two longs. This puts added pressure on the programmer to remember the correct name of the procedure when making a call to one or both. Visual Basic .NET, however, overcomes this problem with the help of function overloading. The same code fragment rewritten in Visual Basic .NET is shown next:

```
Overloads Function Divide(i1 As Integer, i2 As Integer) As Integer
If i2 = 0 Then
Return 0
Else
Return i1 / i2
End If
End Function
Overloads Function Divide(l1 As Long, l2 As Long) As Long
```

#### **676 Chapter 14 • Upgrading Visual Basic Applications to .NET**

```
If i2 = 0 Then
Return 0
Else
Return i1 / i2
End If
End Function
```

Now, both functions share the same name, differing only in the data type passed as arguments. The runtime chooses the appropriate method depending on the parameters used when the call is made.

Since overloading is not supported in Visual Basic 6.0, the functions written in Visual Basic 6.0 undergo the same modifications that relate to data type changes, keyword changes, and so on, used in the procedure. The following points summarize changes affected by Visual Basic .NET:

- \_ Optional parameters must be declared with a default value.
- \_ Visual Basic .NET does not support the IsMissing function.
- \_ Static modifiers cannot be used in Visual Basic .NET when declaring procedures or functions.
- \_ The **Return** statement is now used to transfer control to the calling program.
- \_ Visual Basic .NET requires parentheses to invoke any procedure or function containing parameters.
- \_ Use of the **Any** keyword is not supported in Visual Basic .NET. The **Any** keyword is used in external procedure declarations to indicate a particular parameter can contain data of any type.
- \_ The default parameter passing mechanism in Visual Basic .NET is ByVal.
- \_ ParamArray parameters are always passed as ByVal and the data type for the parameters must be of type *Object*.
- \_ Implementation of overloading—the concept of using the same procedure name with varying function signatures.

**Upgrading Visual Basic Applications to .NET • Chapter 14 677**

## References to Unmanaged Libraries

The introduction of the .NET Framework has certainly given programming a new dimension. Likely the biggest worry you have, concerns all those COM components you created over the years. The good news is that all of them can still be used with applications from the .NET Framework. The .NET Framework provides you with various techniques to help leverage the functionality of existing components.

The following example illustrates a very simple implementation of a .NET component, and the unmanaged client accessing that component. The code for the .NET component is shown next:

```
Namespace SimpleComponent
Public Class Simple1
Public Sub New()
End Sub
Public Function SimpleMethod() As String
Return "From SimpleComponent in .NET"
End Function
End Class
End Namespace
```

The following is the code for the unmanaged client:

```
Sub AccessNETComponent()
Dim obj as Object
Set obj = CreateObject("SimpleComponent.Simple1")
Msgbox obj.SimpleMethod
End Sub
```

Though the method of coding has not changed much, the background work necessary to allow the unmanaged code to access the .NET component is quite substantial. The following steps detail the process:

### **678 Chapter 14 • Upgrading Visual Basic Applications to .NET**

1. Once the .NET component is coded, it must be compiled. This is done using the Visual Basic .NET compiler, which can be invoked from the

command-line by typing: **vbc.exe**. The .NET component then has to be compiled into a DLL, the syntax for which is:

```
Vbc.exe /target:library <sourcefile>
```

The /target:library option instructs the compiler to produce a DLL as the output. In this example, the code is contained in

SimpleComponent.vb, making the command:

```
Vbc.exe /target:library SimpleComponent.vb
```

2. Once the .NET component is compiled, the next step is to register with COM. Dynamic Link Libraries compiled in the .NET environment are different from COM DLLs. Since the old regsvr32.exe cannot be used to register the DLLs, the Register Assembly tool (regasm.exe) should be employed instead. Available as part of the .NET SDK, this tool reads the component's metadata and makes appropriate entries in the registry. These entries include the programmatic identifier (ProgID) and the class ID (CLSID) for the co-classes, which register the appropriate subkeys. You can also use this tool to register the type library when you register the component. So, in this example, you can issue the following command to register the .NET component:

```
Regasm.exe /tlb:simplecomponent.tlb simplecomponent.dll
```

3. Once the component is registered, the unmanaged client application can be copied to the same folder as the .NET component, and then run.

The Common Language Runtime (CLR), which is the core of the .NET Framework, manages all code that runs inside the .NET Framework. This code is called the managed code, whereas any code that runs outside the confines of the .NET Framework is called unmanaged code. All COM, COM+ components, ActiveX controls, and Win32 API functions fall under this category. The managed and unmanaged object models vary regarding data types, error handling, and so on. The main function of the CLR is to simplify the interoperation between these components. This is achieved in various ways. This section introduces some of the concepts that aid in this transition:

## **Upgrading Visual Basic Applications to .NET • Chapter 14 679**

\_ Metadata

\_ Runtime Callable Wrapper

\_ COM Callable Wrapper

### **Metadata**

The word Metadata means data about data. Metadata can be defined as a collection of binary code that describes a .NET component. It is either stored in memory or in a .NET Framework portable executable file. It helps interoperate

### **Attaching a Debugger**

Visual Studio .NET introduces a feature called Debugger that allows you to attach to a process running outside the context of Visual Studio. This feature of attaching to a process allows you to:

\_ Debug a program that runs in a different process on the same machine, or on a different machine. Debugging a process in a different machine is called remote debugging. It is important to note that remote debugging is not supported in

this beta version.

- \_ Debug multiple programs at the same time.
- \_ Invoke the debugger automatically whenever the debugged process crashes. This is also referred to as Just-In-Time (JIT) debugging.

Attaching a debugger to a process is very helpful when attempting to debug an application that interoperates with several other components. You can also debug unmanaged code from within managed code. This can be done by selecting Processes from the Debug menu. The list of currently running processes is listed in the available processes pane. The next step is to select the process you want to debug and click the **Attach** button. This brings up a dialog box displaying the list of available program types. Select the program type related to the application you wish to debug. Click the **OK** button to close this window, followed by the **Close** button on the Processes window.

#### 680 Chapter 14 • Upgrading Visual Basic Applications to .NET

with pre-existing COM components, so if your .NET component wants to work with a COM component, the COM component needs to provide information about itself so the .NET component can identify the methods and properties contained in the *COM* object. The runtime uses the metadata definition to bind the component during compile time and generate the relevant wrapper. The metadata about the COM component is available just like any other managed namespace the CLR provides.

Various tools can be used to generate metadata of a COM component.

They are:

- \_ Type Library Importer
- \_ TypeLibConverter Class

The Type Library Importer, `tlbimp.exe`, is a command-line utility that converts classes and interfaces contained in the COM type library to .NET metadata.

The metadata is then used by the .NET clients to instantiate a *COM* object.

Unfortunately, the Type Library Importer utility converts the entire COM type library, not just a portion of it. It also cannot convert an in-memory type library to metadata. The syntax of the `tlbimp.exe` tool is:

```
tlbimp.exe <TypeLibrary file> [/out: outputfilename]
```

For example, the following command will allow you to convert the ADO type library to .NET metadata:

```
Tlbimp.exe msado25.tlb /out:adonet.dll
```

You can use the Intermediate Language Disassembler (ILDASM) tool to view the contents of the file generated by the `tlbimp.exe` tool.

The `TypeLibConverter` class, meanwhile, is part of the *System.Runtime*

*.InteropServices* namespace and can convert the classes and interfaces contained in the COM type library to .NET metadata. The class contains two methods that aid in this conversion. They are:

- \_ `ConvertAssemblyToTypeLib`
- \_ `ConvertTypeLibToAssembly`

Both methods output the same metadata.

## Runtime Callable Wrapper

The Microsoft Component Object Model (COM) differs from the .NET Framework in a number of ways. There is also another CLR component, apart from using metadata, that helps .NET clients talk to COM components. It's called the Runtime Callable Wrapper (RCW). The RCW is a proxy created by the CLR when a .NET client generates an instance of the *COM* object. From the client's point of view, the RCW is seen as an instance of the managed object. The primary function of a RCW is to marshal calls between a .NET client and a *COM* object.

The CLR creates one RCW for each *COM* object. Even though a client may hold multiple references to the *COM* object, only one RCW will be created. The runtime is responsible for creating both the *COM* object and that for the Runtime Callable Wrapper. The *Runtime Callable Wrapper* object contains a repository, which holds the interface pointers to the *COM* object. It releases the reference to the *COM* object when the number of references is zero.

The RCW marshals data between managed and unmanaged code. It also reconciles the data representation differences that exist between the client and the *COM* object in terms of arguments passed to methods and return values.

*Marshalling* is a mechanism that refers to the method by which a client in one process makes a call to functions in another process running on the same machine or on a remote machine. This is achieved in two steps. First, the client must be aware of the existence of the server process. This is done by taking a reference to the interface and passing it over to the client process. The second step is to pass the parameters from the client to the server. The underlying architecture creates a server proxy in the client process and a stub in the server process. The client sees the proxy as the server and makes method calls as if it is the actual server. Once a method call is made, the proxy accepts the call and passes on the stub located in the server process. The actual transportation is handled by some form of remote process communication like shared memory, named pipes, or others. After the stub receives the request, it parses the call and passes it onto the server. Marshalling is needed whenever the client and server are loaded in different processes either on the same or a different machine.

Garbage Collection is another issue to be contended with if you are working with objects. The .NET Framework does an excellent job of Garbage Collection and programmers can now concentrate more on building applications rather than worry about releasing objects or leaking memory. Visual Basic .NET controls the way objects are created and destroyed. The **New** keyword is used to create an

### 682 Chapter 14 • Upgrading Visual Basic Applications to .NET

object in Visual Basic .NET. When you set an object to Nothing, the object is destroyed and the memory referenced by the object is freed. But there is more to this than meets the eye. The *Sub New* procedure is a constructor that is called whenever you create an object, and can contain code that does common initialization tasks. It replaces the *Class\_Initialize* method in Visual Basic 6.0. The *Sub New* constructor cannot be explicitly invoked from anywhere in the program except from another overloaded constructor in the same class or in a derived

class. Just as a constructor is called when an object is created, the *Sub Finalize* method performs the role of a *destructor*, effectively replacing the *Class\_Terminate* method and performing all cleanup activities.

The .NET Framework automatically calls the destructor when it determines that objects are not being used anymore. But it is important to note that the call to the destructor is not immediate. The .NET Framework does not invoke the destructor as soon as the object goes out of scope or is destroyed explicitly by setting it to Nothing. The framework instead calls the destructor sometime after the object has been destroyed. The main advantage with Garbage Collection in Visual Basic .NET is that it is automatic. Objects are released and memory is freed without any additional changes from the application. The disadvantage is that some objects might stay in memory longer than needed, causing the unnecessary locking of memory locations. Another disadvantage is that an application cannot directly make a call to the destructor.

You can also implement an additional destructor called *Dispose* if you want to take control of management of resources. The *Dispose* method can contain code to implement all cleanup activities just like the *Finalize* method. The *Dispose* method is not automatically invoked, so your application must summon it to perform finalization tasks.

## COM Callable Wrapper

The COM Callable Wrapper does the same thing as the Runtime Callable Wrapper but from the COM client's point of view. When a COM client creates an instance of the managed class, the runtime creates a COM Callable Wrapper for the managed object. The runtime creates only one wrapper for the managed object irrespective of the number of COM clients requesting the reference to the managed object. The primary responsibility of a COM Callable Wrapper is to marshal calls between the managed object and the COM client. The following points summarize how references to unmanaged libraries are handled in Visual Basic .NET:

### Upgrading Visual Basic Applications to .NET • Chapter 14 683

\_ Metadata is binary data that describes a .NET component.

\_ The Runtime Callable Wrapper (RCW) is a proxy that helps .NET clients talk to COM components.

\_ The COM Callable Wrapper (CCW) is a proxy that helps COM clients talk to .NET components.

## Tracing Code

The *Debug* class present in the *System.Diagnostics* namespace provides you with various methods that allow you to trace code. Code tracing is important during development because it aids you in identifying a problem or in analyzing performance. The *Write* and the *WriteLine* methods allow you to print messages in the Output window. This way, you can place temporary messages to track the application flow. This is a very important factor to consider if you are building a client and server application and want to track the code-paths in both applications.

The .NET Framework also contains the *Trace* class which helps you

trace the flow of the application. To embed tracing in your application, you should compile your application with a set of trace switches. These switches also allow you to specify where the trace information should be displayed and to what extent tracing should be done.

Since the *Trace* and *Debug* classes allow you to monitor an application's performance, as well as provide information about application flow, you may want to include code, when developing an application, that use the methods of the *Trace* and *Debug* class. The *Debug* class is normally used to display diagnostic or non-tracing information about your application. After the application has been developed and is ready to be deployed, you can compile the application by turning off Debug switches and turning on Trace switches.

To enable or disable Trace or Debug switches, open Solution explorer, right-click **Solution**, and choose **Properties**. In the **Property Page** dialog box, choose **Configuration Properties** from the left pane and select **Build**. In the right pane, select the **Define Debug Constant** and/or the **Define Trace Constant** checkboxes under conditional compilation constants, depending on whether you want debug and/or trace.

#### 684 Chapter 14 • Upgrading Visual Basic Applications to .NET

## Properties

Property Procedures are implemented differently in Visual Basic .NET. With the **Set** statement no longer supported, both variable assignments and object assignments are treated the same. A property procedure consists of a set of Visual Basic statements that allow you to work with properties that are user-defined. These properties are defined in a class or a module. Visual Basic .NET provides two types of property procedures to work with properties. They are:

*\_Get*: The Get procedure is used to return the property's value.

*\_Set*: The Set procedure is used to assign a value to the property.

## Working with Property Procedures

In Visual Basic 6.0, a property procedure is declared in the following manner:

```
Property Let CustName(strCustName as string)
m_CustName = strCustName
End Property
Property Get CustName() as String
CustName = m_CustName
End Property
```

In VB.NET, however, they are declared differently. Property procedure statements are contained within the **Property** and **End Property** statements. The Get and Set procedures are coded within this block. A property can be declared as a default property by prefixing the property procedure with the **Default** keyword, or you can define the scope of the property procedure using the **Public**, **Protected**, **Friend**, or **Private** keywords. Properties are public by default, unless otherwise specified. The following code shows you the implementation of a property procedure:

```
Public Property CustName() as String
Get
Return m_CustName
End Get
Set
```

```
m_CustName = Value
End Set
End Property
```

### Upgrading Visual Basic Applications to .NET • Chapter 14 685

If you look closely at the procedure declaration, you will see that a variable with the name *Value* is being used. Visual Basic .NET uses this variable name as the default variable if you did not declare the Set procedure as receiving any arguments.

In a Get procedure, the return value is the value of the property returned to the calling expression. In a Set procedure, the new property value is passed in as the argument of the **Set** statement. If an argument is declared, then it must be of the same data type as the property. If an argument is not specified, then the implicit argument named *Value* is used to represent the new value.

The following code fragment shows you how to implement a property procedure with arguments:

```
Public Class Class1
Private intSamp As Integer
Property Sample(ByVal x As Integer)
Get
Sample = intSamp
End Get
Set
intSamp = Value
End Set
End Property
End Class
```

The method of declaring arguments for property procedures is the same as declaring arguments for a Function or a Sub procedure. The only difference in the declarations is that all parameters are passed as *ByVal*. You can also declare optional arguments to property procedures. Arguments declared as optional must have a default value assigned to them. The new syntax is vastly different from the earlier versions of Visual Basic.

## Control Property Name Changes

Visual Basic .NET has replaced many property names with new names. Besides this, all data binding properties have been implemented differently in VB.NET. The following section provides a summary of changes effected for property names.

### 686 Chapter 14 • Upgrading Visual Basic Applications to .NET

#### *Label Control*

The Label control has undergone the following changes in Visual Basic .NET:

- \_ The *Align* property has been changed to *TextAlign*.
- \_ The *Appearance* property has no equivalent and has been combined with the *BorderStyle* property.
- \_ The *Caption* property has been replaced with the new *Text* property.
- \_ A new property called *Modifiers* has been introduced to fix the scope of the control. The possible values are Private, Public, and Protected.
- \_ All data binding and OLE properties have been removed.

#### *Button Control*

The Visual Basic 6.0 `CommandButton` control has been renamed `Button` Control. Besides the change in name, the `CommandButton` control has also undergone the following changes:

- \_ The *Caption* property has been changed to *Text* property.
- \_ The `Button` Control can now have a `ContextMenu` associated with it through the *ContextMenu* property.
- \_ A new property called the *DialogResult* property has been introduced. This property has the following valid values: `Abort`, `Cancel`, `Ignore`, `No`, `None`, `OK`, `Retry`, and `Yes`. If the value of this property is set to anything other than `None`, and if the parent form was displayed through the *ShowDialog* method, clicking the button closes the parent form without having to code for any events. The form's *DialogResult* property is then set to the same value as the *DialogResult* property of the *Button* object.
- \_ The *Default* and *Cancel* properties have been removed.

### *Textbox Control*

The `Textbox` control has undergone the following changes in Visual Basic .NET:

- \_ Two new properties have been introduced to facilitate formatting the contents of the `Textbox` control. The properties are *AcceptsTab* and *AcceptsReturn*.

#### **Upgrading Visual Basic Applications to .NET • Chapter 14 687**

- \_ A new property called *CharacterCase* has been introduced to set the case of text entered in the `Textbox` control.
  - \_ A new property called *Lines* has been introduced, allowing a user to enter multiple lines during design time.
- In general, all controls have undergone changes with respect to the following properties:
- \_ The *Index* property is no longer supported in any of the controls.
  - \_ The *MousePointer* property has been changed to *Cursor* property.
  - \_ A new property called *Modifiers* has been added. This can be used in selecting the access specifier for the control.
  - \_ The *Caption* property has been changed to *Text* property.
  - \_ A new property called *ImageAlign* has been added. This property can be used to set the alignment of the control in the form.
  - \_ A new property called *Dock* can be used to dock the controls to a specific location.

During an upgrade, the older properties supported will be automatically mapped to newer properties. This includes properties that have been retained as is or properties that have had their names changed. If your control used properties that are unsupported in Visual Basic .NET, then they are marked as `UPGRADE_ISSUE` with an appropriate description of the issue.

### **Default Property**

A default property is a property that can be accessed by referencing the object directly. In reality, it is more of a programming shortcut. For example, the *Label* object has the *Caption* property as its default property. So, if you had a label named `label1`, instead of writing the following line of code to set the caption on

the label:

```
label1.Caption = "Enter Name"
```

you can write:

```
label1 = "Enter Name"
```

The default property is resolved when the code is compiled. It is also possible to use late-bound objects with default properties. When using late-bound objects, the property is resolved at runtime, as shown in the following:

#### 688 Chapter 14 • Upgrading Visual Basic Applications to .NET

```
Dim objLbl as Object  
Set objLbl = Form1.label1  
ObjLbl = "Enter Name"
```

There are a lot of disadvantages in using default properties as implemented in Visual Basic 6.0:

\_ Default properties assume that the programmer knows what default property is associated with each object. This leads to uncertainty when debugging programs. In the preceding code fragment, it is difficult to determine whether the string value "Enter Name" is assigned to a variable called *label1* or whether the string value is assigned to the default property of the object called *label1*.

\_ It is not easy to determine if an object has a default property and if so, what property that should be.

\_ Default properties necessitate the usage of the **Set** statement. This is because we need to differentiate between working with an object and working with a default property of the object. With the **Set** statement becoming obsolete in Visual Basic .NET, the need for using parameterless default properties is also done away with. The following example illustrates the need for using the **Set** statement when assigning an object reference:

```
Dim Text1 as Textbox  
Dim Text2 as Textbox  
Text1 = "Some Text" 'Assigning a value to the text property  
Set Text2 = Text1 'Assigning the Text1's object reference  
'to Text2  
Text2 = Text1 'Assigning the text property of Text1 to  
'text property of Text2
```

Visual Basic .NET does not implement the concept of parameterless default properties. So, during an upgrade process, the Upgrade Wizard resolves the default properties to the appropriate property. But if you are using late-bound objects, then the Upgrade Wizard does not have much information about the type of object this object will be bound to. The preceding example can be rewritten in Visual Basic .NET as:

```
Dim Text1 as Textbox  
Dim Text2 as Textbox
```

#### Upgrading Visual Basic Applications to .NET • Chapter 14 689

```
Text1.text = "Some Text" 'Assigning a value to the text property  
Text2 = Text1 'Assigning the Text1's object reference  
'to Text2  
Text2.text = Text1.text 'Assigning the text property of Text1 to  
'text property of Text2
```

However, Visual Basic .NET does support default properties with parameters. The nomenclature of the two terms can be a little confusing. The following code

aims to clear this up, however. The *System.Collections* namespace implements a *Dictionary* class that stores various key-value pairs. Consider the following code which adds two words to the dictionary collection and displays them:

```
Dim objCol As New System.Collections.Dictionary()  
objCol.Add(1, "Amrita")  
objCol.Add(2, "Aarthi")  
Msgbox(objCol.Item(1).ToString) 'Explicitly referencing the  
'Item property  
Msgbox(objCol(2).ToString) 'Using the default property  
'with parameter
```

The *Item* property is the default property for the *Collection* object. Since this property can be accessed by specifying an index as a parameter, you can reference this as a default parameter so the statement becomes a valid reference to the default parameter (see the code fragment that follows):

```
Msgbox(objCol(2).ToString)
```

The following points summarize the changes that have been made to the process of working with properties in Visual Basic .NET:

- \_ The syntax of property procedures has been changed.
- \_ There is no longer a Let procedure.
- \_ Property names for commonly-used controls have undergone changes in name. Also, some properties have been removed.
- \_ A property can be considered a default property only if it can be parameterized.

## 690 Chapter 14 • Upgrading Visual Basic Applications to .NET

### Null Usage

The **Null** keyword and **Empty** keywords are not supported in Visual Basic .NET. The **Null** keyword was used in previous versions of Visual Basic to indicate that a variable does not contain any valid data. The **Empty** keyword, when assigned to a variable, indicates that the variable is uninitialized. Visual Basic .NET introduces the **Nothing** keyword, effectively replacing the **Null** and **Empty** keywords. Alternatively, you can use the *DBNull* class in the *System* namespace to represent a Null value. Note that the **Null** keyword is no longer in use.

The word Null is a reserved keyword in Visual Basic .NET, but does not have any implementation so far. With the **Empty** keyword phased out, the *IsEmpty* function is no longer supported in Visual Basic .NET. The *IsNull* function has been replaced by the *IsDBNull* function. The following code illustrates this. A variable of type *Object* is assigned a *DBNull* value and the *IsDBNull* function is used to check for the same:

```
Dim objSample As Object  
objSample = System.DBNull.Value  
If IsDBNull(objSample) Then  
Msgbox("Sample is null")  
Else  
Msgbox("Sample is not null")  
End If
```

It is important to note that some expressions which you might expect to evaluate to True under certain circumstances (such as `If Var = DBNull` and `If Var <> DBNull`) are always False. This is because any expression containing a *DBNull* is itself *DBNull* and, therefore, False.

# Understanding Error Handling

Applications written using Visual Basic 6.0 used the **On Error** statement to handle errors. The *Err* object provided diagnostic information about the error. The *Number* and *Description* properties provided the error code and a description of the error. The main drawback of this kind of error handling is the inability to trap errors raised by Windows DLLs. System errors that arise during calls to Windows DLLs do not raise exceptions and cannot be trapped by this style of error handling.

## Upgrading Visual Basic Applications to .NET • Chapter 14 691

Visual Basic .NET uses structured exception handling to handle errors and exceptions. Most of the object-oriented (OO) languages use this mechanism to handle errors. Structured exception handling enables you to create more robust and comprehensive error handlers. The Common Language Runtime (CLR) uses structured exception handling based on exception objects and protected blocks of code. When an exception or an error occurs, an object is created to represent the exception. The exception objects created are objects of exception classes derived from *System.Exception*. It is also possible to create custom exception classes. All languages that use the CLR handle exceptions in a similar manner. Structured exception handling consists of using the *Try...Catch...Finally* syntax. The *Try* block normally contains both the corresponding *Catch* and *Finally* blocks. The code that throws the exception is surrounded in a *Try* block. The *Catch* block consists of a series of statements beginning with the keyword **Catch**, followed by an optional filter that specifies the exception type. It is also possible to code multiple *Catch* blocks for a *Try* block. A *Catch* block that contains a filter for a specific exception type is invoked when that exception is thrown. The *Catch* block that contains no parameters, also called a general exception handler, is invoked for all other exceptions. The *Finally* block follows the *Catch* block, and contains the cleanup code.

It is important to note the order of the **Catch** statements if you are coding a general exception handler as well as specific exception handlers. The general handler should be the last if you are coding *Catch* blocks to handle specific exceptions. If the general handler is coded first followed by other specific handlers, the runtime invokes the general handler by default since the general handler handles all exceptions. The rule of thumb is to go from specific exception handlers to general exception handlers. See the code that follows:

```
Try
<Statements to be executed>
'Indicates the beginning of the exception handler
'Contains code that might throw exceptions
Catch [<variable> As <ExceptionType>]
<Exception processing statements>
'This will be executed if the statements in the Try
'block fails and the exception thrown matches the
'exception specified as a parameter
[Additional Catch Blocks]
```

## 692 Chapter 14 • Upgrading Visual Basic Applications to .NET

```
Finally
<Cleanup Code>
```

```
'Contains cleanup code
End Try
```

Consider the following code fragment that demonstrates how to implement a **Try...Catch...Finally** in a Visual Basic program.

```
Try
result = intVar1 / intVar2
Catch ex as System.OverflowException
Msgbox (ex.Message)
Finally
End Try
```

The statement that divides two integers is included in the *Try* block because there might be a situation when the denominator could be zero. The *Catch* block is defined with an object of the specific exception type. This is the exception we expect the code to throw. Once the exception is caught, you can display a custom error message to the user. In this case, the message displayed is the default message for this exception as defined in the CLR. Normally, the *Finally* block contains cleanup code. In this case, however, the *Finally* block is left empty. It is also possible to extend exception handling by implementing custom interfaces. Custom exception handlers are implemented by creating a new exception that inherits from a *System.Exception* class or a class derived from the *System.Exception* class. Then you need to determine the situations under which this exception will be thrown, and finally, write appropriate code to throw that exception. The following exercise walks you through these steps.

## Exercise 14.1: Using Error Handling

1. Add a class to your project. The first step is to inherit from the *System.Exception* class or a class derived from the *System.Exception* class.

The following code shows you the implementation:

```
Public Class CreditDebitException
Inherits System.Exception
'Call the base class constructor from the constructor
'of this class.
```

### Upgrading Visual Basic Applications to .NET • Chapter 14 693

```
Public Sub New(ByVal strMessage as String)
MyBase.New(strMessage)
End Sub
End Class
```

2. The second step is to decide how and when to use this exception. This can be done by examining the code and determining where this exception could fit in.

3. After finalizing the usage, you can use the exception in the application. Use the **Throw** statement to explicitly raise an exception. This is equivalent to calling the *Raise* method of the *Err* object in Visual Basic 6.0. In this example, a middle-tier component throws the *CreditDebitException* when the *Credit* and *Debit* amounts are not equal. Once the exception is thrown by the component, the client receives the exception. The client would typically code the *CheckDebitCredit* method in the *Try* block and the *Catch* block can be coded to receive the

*CreditDebitException*, as shown next:

```
Public Sub CheckDebitCredit(ByVal intDebitVal as Integer, ByVal _
intCreditVal as Integer)
```

```
If (intDebitVal != intCreditVal) Then
Throw New CreditDebitException("Credit and Debit must
be equal")
End If
End Sub
```

## Data Access Changes in Visual Basic .NET

ADO.NET is a vast improvement over its predecessor ADO. ADO.NET offers a variety of features that include disconnected data access, performance optimization, and better and richer data type support. There are a lot of differences between ADO and ADO.NET. This section attempts to cover most of them. ADO.NET introduces the *DataSet* object which can represent multiple tables, store relationship information, and provide disconnected access to data.

694 Chapter 14 • Upgrading Visual Basic Applications to .NET

### DataSet and Recordset

ADO uses the *Recordset* object to represent the entire set of records from a single base table. Even though you cannot store multiple tables in a recordset, it is possible to store data from multiple tables using a JOIN clause in the SELECT query that builds the recordset. The ADO.NET *DataSet* object is a collection of one or more tables, and the tables contained in the dataset are called data tables. These can be accessed using the *DataTable* objects. The *TablesCollection* object contains all the *DataTable* objects in a *DataSet*, each *DataTable* object corresponding to a table in the actual database.

The columns in a *DataSet* are represented using a *DataColumn* object. It is also possible to relate the tables in the dataset using the *DataRelation* objects. The *DataRelation* object uses the *DataColumn* object to relate two or more tables by employing the concept of a foreign key. This feature allows the user to implement more complex operations than were hitherto possible. The *DataSet*, with the ability to store multiple tables and the relationships between those tables, offers a feature-rich implementation.

From the user's perspective, a dataset is a representation of an actual database residing on the client's machine. After an upgrade, you can still continue to use your existing applications but will not be able to leverage the benefits of ADO.NET. The Microsoft ActiveX Data Objects type library is automatically upgraded and the code is modified to reflect the syntax of VB.NET during the upgrade.

### Application Interoperability

Marshalling ADO disconnected recordsets was achieved through Component Object Model (COM). The disadvantage with COM marshalling is the restricted availability of data types. The data types that are available are what COM provides. By comparison, ADO.NET transmits datasets as XML streams. Since XML has no restriction on data types, the component consuming the datasets is free to use whatever appropriate data types it normally uses.

Transmitting a large disconnected ADO recordset over the network places enormous stress on the network resources. The increase in stress is directly proportional

to the size of the recordset being transmitted. Though ADO does offer its own performance optimization, it still suffers because of its dependency on COM. The data type conversions are required for COM marshalling between components. ADO.NET, on the other hand, does not need to enforce any data conversions and data is marshaled as XML.

#### **Upgrading Visual Basic Applications to .NET • Chapter 14 695**

Disconnected ADO recordsets that are marshaled across intranets or the Internet suffer from restrictions imposed by firewalls. A firewall allows only HTML text to pass, preventing any operations that access system resources. Here again, ADO.NET scores over ADO in that ADO.NET passes data around as XML streams. The advantage of XML streams is that they are just text data. This allows the data to be transmitted using the HTTP protocol which most firewalls allow. However, if you have to pass an ADO recordset, you also need to package and send interface methods and parameters from the client to the server or vice-versa.

## **Cursor Location**

The ADO *Recordset* object can be created by the application in two places: within the application as a client-side cursor, or within the data store as a server-side cursor. Client-side cursors are supported in ADO.NET by the *DataSet* object, while server-side cursors are implemented using the *DataReader* object.

When you upgrade an ADO application from Visual Basic 6.0 to Visual Basic .NET, the Upgrade Wizard modifies the Microsoft ActiveX Data Objects library as well, prompting your existing code to be altered in order to suit the .NET Framework. Any reference to the *CursorLocation* constants is upgraded to reflect the change. So, the values of *adUseClient* and *adUseServer* are upgraded to *ADODB.CursorLocationEnum.adUseClient* and *ADODB.CursorLocation.adUseServer*, respectively.

## **Disconnected Access**

While ADO is primarily designed for connected access to the database, ADO.NET provides disconnected access to data. Whereas ADO communicates with the database by making calls to the OLEDB provider, or through any of the APIs provided by the DBMS, ADO.NET communicates through the *data adapter* object. The *DataSetCommand* object in turn makes calls to the OLEDB provider. The main difference between disconnected access in ADO and ADO.NET is that the *DataSetCommand* object optimizes performance, performs validity checks, and controls the way the changes are written to the database.

## **Data Navigation**

The data in an ADO recordset can be accessed by calling any one of the move methods supported by the *Recordset* object. In ADO.NET, rows are represented as collections. This allows the programmer to work with them just like objects. New rows can be added through the *Add* method, rows can be deleted using the

#### **696 Chapter 14 • Upgrading Visual Basic Applications to .NET**

*Remove* method, and rows can be accessed through an ordinal or a primary key, such as an index. *DataRelation* objects allow the programmer to maintain a master-detail relationship between the tables in the database. This means that as

you move from one record to another in a master table, the corresponding records in the transaction table are made available.

The object models for ADO and ADO.NET are radically different. ADO does not support any of the objects present in the new object model. The choice here is to either retain the application as it is, or re-write your application to take advantage of the new features.

## Lock Implementation

ADO holds up database locks and database connections for long durations that result in performance bottlenecks and resource contentions. The disconnected data access implemented by ADO.NET ensures that database locks or database connections are not held for longer periods of time. When you upgrade an existing Visual Basic 6.0 ADO application, the Upgrade Wizard converts the existing ADO type library to .NET metadata. The existing classes and their associated methods and properties can be used as they are, without any modifications.

## Upgrading Interfaces

Earlier versions of Visual Basic used interfaces, but could not create them directly. Visual Basic .NET introduces this feature with the **Interface** statement, which allows you to define true interfaces. The interfaces you define are distinctly different from classes, and it is now possible to actually implement them using the enhanced **Implements** keyword.

Interfaces define the properties, methods, and events that classes can implement. The basic purpose of defining an interface is to logically group properties, methods, and events that represent a logical entity. Besides, it is also possible to extend interfaces by creating new interfaces from the old, and adding more functionality without breaking existing clients.

The main purpose of interfaces in any language is to allow the objects and their interfaces (methods, parameters, properties, and so on) to be designed for all the objects in a system. This allows developers to work concurrently on different objects without having to wait on one or the other to be implemented since the interfaces between the objects are defined.

An interface, unlike a class, does not provide implementation, it defines a contract between itself and a class. This is a two-way relationship. The class

### Upgrading Visual Basic Applications to .NET • Chapter 14 697

implementing the interface must implement all the methods, and the interface guarantees no change will be made to the existing interface. In order to incorporate functionality changes, you can create a new interface that inherits from the original version.

Visual Basic .NET uses the **Interface** and **End Interface** statements to define an interface. The property, method, and event definitions are then embedded within these statements. The following rules apply to interfaces:

- \_ The **Inherits** statement follows the **Interface** statement if the interface inherits from another interface.
- \_ The **Inherits** statement must be the first statement after the **Interface** statement. Only comments, if any, can precede the **Inherits** statement.
- \_ The interface can only contain **Event**, **Sub**, **Function**, or **Property**

statements. Interfaces cannot contain any implementation code or even **End Sub** or **End Function** statements.

\_ **Interface** statements are Public by default, but they can also be declared as Friend, Protected, or Private.

While an Interface declaration can contain any of the four modifiers just mentioned, it is not possible to declare a Sub, Function, or Property definition with any other modifier than the **OverLoads** or **Default** keywords. Therefore, a function cannot be declared with Public, Private, Friend, Protected, Shared, Overrides, MustOverride, or Overridable. The reasons for this restriction are described in the following bullet points:

\_ The Public modifier indicates the entity has unrestricted access and can be accessed by any object, even those that do not implement the interface containing this entity. This disassociates the entity as a member of the interface.

\_ The Private modifier restricts the entity's access to only its declaration context, thereby rendering the entity totally inaccessible.

\_ The Friend modifier restricts access to only the program that contains the entity declaration.

\_ The Protected modifier restricts the entity's access to only those interfaces that derive from this class. As a result, those classes that implement this interface have no access to the entity.

#### 698 Chapter 14 • Upgrading Visual Basic Applications to .NET

\_ An entity declared with the Shared modifier does not operate on a specific instance of a type, meaning that the entity can be invoked directly from a class rather than from the instance.

\_ The Overrides modifier indicates that the entity will be overridden in the derived class. This goes against the contract between the Interface and the class, being that a class is required to implement all the entities in the interface without any changes.

\_ The MustOverride modifier means that the derived class must override the entity in order to be creatable.

\_ The Overridable modifier indicates that the entity can be overridden.

The **Implements** keyword signifies that a class member implements a specific interface. An **Implements** statement requires a comma-separated list of values, each value representing a single interface member that is implemented in a class. Normally, only a single interface member is specified, but it is also possible to implement multiple members. The interface member is specified in the following format:

```
<InterfaceName.InterfaceMember>
```

The method that implements the entity need not follow the Visual Basic 6.0 convention of *InterfaceName\_MethodName*. The method name can be any legal identifier. The following code fragment illustrates a method that implements an interface:

```
Function Add(ByVal intOper1 as integer, ByVal intOper2 as integer) as_
Integer Implements ICalculator.Add
Add = intOper1 + intOper2
End Function
```

The implementing method's function signature should match the Interface method's function signature. In other words, the data type of the arguments, return value, and so on, must be exactly the same. You can declare the method that implements the interface member with any of the legal modifiers allowed on the instance method declarations. The legal modifiers are Overloads, Overrides, Overridable, Public, Private, Protected, Friend, Protected Friend, MustOverride, Default, and Static.

The **Implements** statement can also be used to declare a single method that implements multiple methods of multiple interfaces. This feature comes in handy

#### Upgrading Visual Basic Applications to .NET • Chapter 14 699

when all the methods exhibit the same functionality. Consider the following code fragment:

```
Sub Method1 Implements InterfaceA.Method1, InterfaceB.Method2, _  
InterfaceC.Method3, InterfaceD.Method4  
'Visual Basic Code  
End Sub
```

## Upgrading Interfaces from Visual Basic 6.0

Visual Basic 6.0 allowed only components consume interfaces. There was no way an interface could actually be created. There might be situations when you decide to upgrade your component to Visual Basic .NET to take advantage of a lot of new features. When you do the upgrade, the Upgrade Wizard will make changes to your existing component to make it compatible with Visual Basic .NET. This section is devoted to demystifying the changes the Upgrade Wizard will make to your existing component.

Consider the following program, written in Visual Basic 6.0, which implements a simple calculator. The *ICalculator* interface implements four simple functions of Add, Subtract, Multiply, and Divide. Each of the functions accepts two integers and returns a third integer as a result. The *clsCalc* class implements all the interface methods. The client code, meanwhile, is implemented in a Form.

The code for the *ICalculator* interface is shown in the following:

```
Function Add(intOper1 As Integer, intOper2 As Integer) As Integer  
End Function  
Function Subtract(intOper1 As Integer, intOper2 As Integer) As Integer  
End Function  
Function Divide(intOper1 As Integer, intOper2 As Integer) As Integer  
End Function  
Function Multiply(intOper1 As Integer, intOper2 As Integer) As Integer  
End Function
```

The code implementing the *ICalculator* interface is shown next.

#### 700 Chapter 14 • Upgrading Visual Basic Applications to .NET

```
Implements _ICalc  
Private Function ICalc_Add(intOper1 As Integer, _  
intOper2 As Integer) As Integer  
ICalc_Add = intOper1 + intOper2  
End Function  
Private Function ICalc_Divide(intOper1 As Integer, _  
intOper2 As Integer) As Integer  
ICalc_Divide = intOper1 \ intOper2  
End Function
```

```

Private Function ICalc_Multiply(intOper1 As Integer, _
intOper2 As Integer) As Integer
ICalc_Multiply = intOper1 * intOper2
End Function
Private Function ICalc_Subtract(intOper1 As Integer, _
intOper2 As Integer) As Integer
ICalc_Subtract = intOper1 - intOper2
End Function

```

The client uses the methods of the ICalculator interface as illustrated in the following:

```

Private Sub TestInterface()
Dim objCalc As ICalc
Set objCalc = New clsCalc
MsgBox objCalc.Add(10, 20)
End Sub

```

The succeeding code segment illustrates the changes made to the ICalculator interface after the project is upgraded to Visual Basic .NET:

```

Namespace Project1
Interface _ICalc
Function Add(ByRef intOper1 As Short, _
ByRef intOper2 As Short) As Short

```

#### **Upgrading Visual Basic Applications to .NET • Chapter 14 701**

```

Function Subtract(ByRef intOper1 As Short, _
ByRef intOper2 As Short) As Short
Function Divide(ByRef intOper1 As Short, _
ByRef intOper2 As Short) As Short
Function Multiply(ByRef intOper1 As Short, _
ByRef intOper2 As Short) As Short
End Interface
Public Class ICalc
Implements _ICalc
Function Add(ByRef intOper1 As Short, ByRef intOper2
As Short) As _
Short Implements _ICalc.Add
End Function
Function Subtract(ByRef intOper1 As Short, ByRef intOper2
As Short) As _
Short Implements _ICalc.Subtract
End Function
Function Divide(ByRef intOper1 As Short, ByRef intOper2 As Short) As _
Short Implements _ICalc.Divide
End Function
Function Multiply(ByRef intOper1 As Short, ByRef intOper2
As Short) As _
Short Implements _ICalc.Multiply
End Function
End Class
End NameSpace

```

#### **702 Chapter 14 • Upgrading Visual Basic Applications to .NET**

The changes that the Upgrade Wizard makes to the existing code are quite noteworthy:

1. The Upgrade Wizard creates a new interface called `_ICalc`. This name is the name of the Visual Basic 6.0 class that holds the interface definitions.

2. The interface definitions are coded within the Interface...End Interface. The End Functions are also removed so that definitions contain only the function names without any implementation code.
3. The integer data type is changed to short.
4. The Upgrade Wizard then creates a new class called *ICalc* that derives from *\_ICalc*. This wrapper class contains interface member definitions with the partial implementation.
5. Then another derived class, which actually contains the full implementation of the four functions, is created. This class is called *clsCalc*, the contents of which are shown in the following:

```

Namespace Project1
Public Class clsCalc
Implements _ICalc
Private Function ICalc_Add(ByRef intOper1 As Short,
ByRef intOper2 As Short) As Short
Implements _ICalc.Add
ICalc_Add = intOper1 + intOper2
End Function
Private Function ICalc_Divide(ByRef intOper1 As Short,
ByRef intOper2 As Short) As Short
Implements _ICalc.Divide
ICalc_Divide = intOper1 \ intOper2
End Function
Private Function ICalc_Multiply(ByRef intOper1 As Short,
ByRef intOper2 As Short) As Short
Implements _ICalc.Multiply
ICalc_Multiply = intOper1 * intOper2

```

#### Upgrading Visual Basic Applications to .NET • Chapter 14 703

```

End Function
Private Function ICalc_Subtract(ByRef intOper1 As Short,
ByRef intOper2 As Short) As Short
Implements _ICalc.Subtract
ICalc_Subtract = intOper1 - intOper2
End Function
End Class
End NameSpace

```

6. The *clsCalc* class implements the *\_ICalc* interface and contains code that implements the four functions.
7. The client instantiates the *\_ICalc* interface and assigns a reference to the *clsCalc* class. Afterward, the *Add* method is called with two short values.

The code for the client is shown in the following:

```

Public Sub TestInterface()
Dim objCalc As _ICalc
objCalc = New clsCalc
MsgBox(CStr(objCalc.Add(10, 20)))
End Sub

```

## Using the Upgrade Tool

Visual Basic .NET is a paradigm shift from the previous versions of Visual Basic, and there are a lot of advantages in upgrading to it. Exercise 14.2 walks you through this process.

### Exercise 14.2 Using the Upgrade Wizard

1. The upgrade tool is launched as soon as you open a Visual Basic 6.0 project in Visual Basic .NET. Figure 14.1 shows you the initial screen of the Upgrade Wizard. The first step in the wizard summarizes the actions that will be done throughout the Upgrade Wizard. It then creates a new Visual Basic .NET project in a separate folder you specify, leaving your

**704 Chapter 14 • Upgrading Visual Basic Applications to .NET**

existing project unchanged. (It is important to note that a Visual Basic .NET project cannot be opened in Visual Basic 6.0.)

2. After the initial screen is displayed, the next step in the upgrade process is to choose what kind of project the existing project should be upgraded to, as well as configuring certain other options. The Upgrade Wizard determines the project type of the existing Visual Basic 6.0 project and selects the appropriate option. It also displays the existing project type in the first line. For internationalization projects, you can select the code page that translates the project. This step also allows you to configure other options. You can decide if you want to generate default interfaces for public classes, update ActiveX references to Windows Forms, and make arrays zero-based. Refer to Figure 14.2.

**Figure 14.1** Step 1 of 5 of the Upgrade Wizard

**Figure 14.2** Step 2 of 5 of the Upgrade Wizard

**Upgrading Visual Basic Applications to .NET • Chapter 14 705**

3. Step 3 in the upgrade process is to specify where the new upgraded project should be created. Note that your existing project will be left as is, and that the upgraded project won't be able to be opened in previous versions of Visual Basic. Figure 14.3 displays this process.

4. Figure 14.4 shows the screen informing the user that the project is ready to be upgraded. If you are set to proceed, click **Next**. When the project is upgraded, the language is modified for any syntax changes and Visual Basic Forms is converted to Windows Forms. More often than not, you will have to make changes to the code after it is upgraded. This is necessary because certain objects either have no equivalents, or the properties of some objects have been either erased or renamed.

**Figure 14.3** Step 3 of 5 of the Upgrade Wizard

**Figure 14.4** Step 4 of 5 of the Upgrade Wizard

**706 Chapter 14 • Upgrading Visual Basic Applications to .NET**

5. The last screen (Figure 14.5) shows the current status of the upgrade process. After the project is upgraded, the Upgrade Wizard creates an upgrade report that itemizes problems and inserts comments in the code, informing the programmer of what changes should be made. It is not difficult to find the parts of the code that need updating, because the Upgrade Wizard marks that code which needs changing, even including comments with the designation. The comments begin with the text **TODO**, and the IDE picks up these statements and lists them in the

TaskList window. Navigating to the appropriate line is as easy as doubleclicking the item in the TaskList window. Each item in the upgrade report is even associated with a related online help topic, which not only explains the need to change the code, but how to do it.

After the upgrade is completed, the Upgrade Wizard attaches various comments to the upgraded code. These can be categorized into the following four types, based on their severity:

**\_UPGRADE\_ISSUE** errors are items that will generate build errors and prevent the application from compiling. As a result, they are marked as compiler errors. It is necessary to correct them before running the project. These errors are logged in the upgrade report, as well as the Task List.

**\_UPGRADE\_TODO** errors are items that will not hinder the compilation process, but that do result in runtime errors. It is necessary to correct them before the solution will run successfully. These are reported in the upgrade report as both items in the TaskList and comments in the code.

**Figure 14.5** Step 5 of 5 of the Upgrade Wizard  
Upgrading Visual Basic Applications to .NET • Chapter 14 707

**\_UPGRADE\_WARNING** errors are items that will not result in compiler errors but that might still cause an error when referencing the item during runtime. It is not absolutely necessary to rectify them, but doing so will result in a smoother execution of the project. These items are outlined in the upgrade report as both items in the TaskList window and comments in the code.

**\_UPGRADE\_NOTE** indicates serious changes in the code. Upgrade notes are added when there is a major structural change in code. Reported as comments in code, you should read them thoroughly before deciding on a course of action.

After the Visual Basic 6.0 project has been upgraded to Visual Basic .NET, an upgrade report is added to the project. This report contains the following details and is named `_Upgradereport.htm`:

`_Project name`

`_Time of upgrade`

`_Upgrade settings` consist of the following key-value pairs:

`_A Boolean value` indicating whether ADO+ was used.

`_A Boolean value` indicating whether the user requested the Upgrade Wizard to generate public interfaces for classes.

`_The name of the logfile.`

`_The kind of project` this project migrated from.

`_A Boolean value` to indicate if the user preferred to change the arrays to zero-based.

`_The path to the output directory.`

`_The name of the project` that was created.

`_The actual path to the project` that was created.

`_A list of project files` with information regarding the new filename, the old filename, file type, status, errors, warnings, as well as other issues.

## Summary

There are special considerations that must be taken into account when migrating your Visual Basic 6.0 applications to .NET. For instance, you should bind variables because of the changes to property names and data type changes. In addition, you should make sure to avoid null propagation, use ADO for all database applications, use the Date data type to store dates, avoid fixed-length strings, and use constants instead of underlying values. There are changes that have been made to the application architecture in .NET, so it is advisable to port all your applications to ADO before they are moved to .NET.

Data types have also undergone a significant number of changes. Visual Basic .NET, for example uses the Object data type instead of the Variant data type. It also introduces a new data type called Short to store 16-bit numbers. The Integer data type, meanwhile, has been modified to store 32-bit numbers, the Long data type now stores 64-bit numbers, and the underlying value of Boolean True has been changed to -1. Arrays, too, have undergone a change, in that the lowerbound value is now fixed as zero and is impossible to change.

A host of new features have been added to Windows Forms as well, effectively replacing the Visual Basic 6.0 forms. Keywords like **GoSub**, **Option Base**, **Lset**, **VarPtr**, **StrPtr**, **Set**, and **Def** are no longer supported. In addition, the functionality of the **GoTo** statement is restricted only to error-handlers now. Several new modifiers to functions, meanwhile, have been introduced in Visual Basic .NET. The functionality of the **Return** statement has been extended to return a value to the calling function besides returning control. Visual Basic .NET has also introduced the new concept of function overloading, allowing multiple functions to have the same name with each function differing only in their signature. In other developments, the syntax of property procedures has changed, Visual Basic .NET no longer allows parameter-less default properties, and with no support for the **Set** statement, you cannot have a Set property procedure.

Visual Basic .NET allows interoperability with COM components using metadata, Runtime Callable Wrappers, and COM Callable Wrappers. This allows you to leverage the functionality of existing components. Visual Basic .NET also allows you to implement true interfaces, and introduces structured exception handling that uses the *Try...Catch...Finally* block to handle errors instead of just extending support for the **On error...Goto** statement. Database applications can now take advantage of the new data access mechanism called ADO.NET, as well, which is very different from the earlier connection-based ADO model.

### Upgrading Visual Basic Applications to .NET • Chapter 14 709

Finally, the Upgrade Wizard is available to ensure the smooth and easy migration of your existing applications to .NET. Its various upgrade messages clearly outline the changes that must be made before your application is ready to run.

## Solutions Fast Track

### Considerations Before Upgrading

À Early binding of variables is necessary because Visual Basic .NET has

introduced changes to property names and default properties.

À The **Null** keyword is not available in Visual Basic .NET. Instead, you can use the `DBNull.Value` available in the *System* namespace to specify a Null value.

À It is advisable to change the data type of date variables to the `Date` data type in your legacy applications to facilitate an easier migration.

À It is recommended that constants be used instead of the actual underlying value.

## Considering Architecture Before Migrating

À DHTML and ActiveX Document applications cannot be upgraded to Visual Basic .NET.

À Visual Basic 6.0 Forms has been replaced with a new architecture called Windows Forms.

À DAO or RDO data-binding applications must be ported to ADO first before they can be moved to Visual Basic .NET.

## Data Types

À All Variant data types will be converted to the `Object` data type during an upgrade.

À Visual Basic .NET introduces a new data type called `Short`. The Visual Basic 6.0 Integer data type is now represented by the `Short` data type (which stores 16-bit numbers), the Visual Basic 6.0 Long data type is

### 710 Chapter 14 • Upgrading Visual Basic Applications to .NET

now represented by the `Integer` data type (which stores 32-bit numbers), and the `Long` data-type stores 64-bit numbers.

À `ToOADate` and `FromOADate` are used to convert between the `Double` and Visual Basic 6.0 representation of the date value.

À The underlying value of the `True` has been changed from `-1` to `1`.

À Arrays in Visual Basic .NET are zero-based.

À Fixed-length strings are not supported in Visual Basic .NET.

## Converting VB Forms to Windows Forms

À Windows Forms does not support the `OLE Container` control.

À Windows Forms contains two menu controls called `MainMenu` and `ContextMenu`.

À Image controls are not supported in Windows Forms.

## Keyword Changes

À **GoSub**, **Option Base**, **VarPtr**, **StrPtr**, and **Def** keywords are not supported in Visual Basic .NET.

À **GoTo** can be used only for the purposes of error handling.

À Visual Basic .NET introduces three new operators to perform bitwise operations. They are `BitOr`, `BitAnd`, and `BitXor`.

## Programming Differences

À Optional parameters must be supplied with a default value.

À The **Return** statement returns control to the calling program.

- À The default parameter passing mechanism is ByVal.
- À *ParamArray* parameters must be declared as Object.
- À Overloading is implemented with the help of function signature.
- À The Runtime Callable Wrapper (RCW) helps .NET clients talk to COM components.

#### Upgrading Visual Basic Applications to .NET • Chapter 14 711

- À The COM Callable Wrapper (CCW) helps COM clients talk to .NET components.
- À The Set property procedure is not supported in Visual Basic .NET.
- À Visual Basic .NET supports default properties only if the properties can be parameterized.

## Understanding Error Handling

- À Visual Basic .NET introduces structured error handling with the help of the **Try**, **Catch**, and **Finally** statements.
- À It is possible to have multiple **Catch** statements to handle multiple exceptions.
- À You can also create custom exceptions by creating a class that derives from the *System.Exception* class.

## Data Access Changes in Visual Basic .NET

- À Visual Basic .NET introduces a new object model called ADO.NET.
- À The *DataSet* object can hold multiple tables and store relationships between the tables.
- À The *DataReader* object implements the server-side cursor.
- À ADO.NET provides disconnected access to the database.

## Upgrading Interfaces

- À Interfaces are created using the **Interface** keyword.
- À The **Implements** keyword is used to implement an interface.

## Using the Upgrade Tool

- À The Upgrade Wizard is automatically launched when you open a Visual Basic 6.0 project in Visual Basic .NET.
- À The upgraded code is placed in a different folder than that containing the source.

#### 712 Chapter 14 • Upgrading Visual Basic Applications to .NET

- À The Upgrade Wizard lists various upgrade messages indicating what changes must be made to the existing code to ensure a smooth run of the upgraded project.

**Q:** Can I manually invoke the Upgrade Wizard?

**A:** No, you cannot. Visual Basic .NET automatically invokes the wizard when you open an older version of a Visual Basic application.

**Q:** Should I only use structured exception handling to manage errors in my application?

**A:** Structured exception handling is a more comprehensive way to managing errors. However, you can still continue to use the **On Error** statements.

**Q:** Is the object model for ADO.NET frozen? Can I start port my ADO code to ADO.NET?

**A:** At present, it is unclear whether this will be the final draft of ADO.NET. Therefore, it is best to hold on to your existing ADO applications. You can still, however, use the current model to write new non-critical applications for use in your current environment.

**Q:** What is the purpose of the Microsoft Visual Basic 6.0 Compatibility library?

**A:** The Microsoft Visual Basic 6.0 compatibility library contains functions and methods that are a part of Visual Basic 6.0 but not Visual Basic .NET. The contents of this library are used so that your existing applications do not break down solely because the implementation of a concept differed between the two versions.

## Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to [www.syngress.com/solutions](http://www.syngress.com/solutions) and click on the “Ask the Author” form.

713

## Index

/? option, 546

32-bit numbers, 659

### A

A1. *See* Assembly

Generation

About option, 108

Abstract classes, 67–68

AcceptsReturn property,  
686

AcceptsTab property, 686

Access. *See* Disconnected  
access

applications. *See* Data

permissions, limiting, 558

security. *See* Code

model. *See* .NET

Accessibility, 130

options, 141

Action attribute, 471

Active MDI child forms

arranging, 299–300

determination, 299

Active Server Pages (ASP),

7, 461

applications, 14

ASP.NET, 7–8, 37, 83, 443

- server controls, 476–492
- World Wide Web forms, usage, 654
- ASPX files, 7
- contrast. *See* World Wide Web forms
- developers, 466
- pages, 8, 15
- ActiveForm property, 299
- ActiveLinkColor property, 354
- ActiveX
- Control Container, 339
- Control Importer, 270, 338
- usage. *See* Windows forms
- controls, 461, 678
- DLL, 260
- projects, 112
- documents, 653, 655
- usage, 14
- EXE, 112
- Add method, 246, 380, 396
- AddHandler method, 502
- Add-in Manager, 104
- Add-ins, 104–108, 656
- creation, Add-In Wizard (usage), 105–108
- objects, 101
- AddMenu method, 327
- AddRef, 36
- Address text box, 515
- AddressOf (keyword), 235
- AddToArray, 672
- Adjust Security Wizard, 593
- Administrator
- configuration files, 624–625
- policy, applying, 50
- ADO
- ADO.NET (contrast), 414
- applications, 657
- code, 651
- disconnected recordsets, marshalling, 694, 695
- libraries, 417, 418
- usage, 241, 649, 651–652
- versions, 410
- ADO.NET
- architecture, understanding,

412–416, 455  
configuration, 415  
contrast. *See* ADO  
product introduction, 657  
remoting, 415  
usage  
FAQs, 267–268  
introduction, 410  
solutions, 454–456  
ADO.NET.XML, 414  
AdRotator  
control, 487  
server, 488  
Advanced programming  
concepts  
FAQs, 267–268  
introduction, 220–221  
solutions, 265–267  
AfterClosing event, 103  
Alias  
command, 138  
creation, 138–139  
setting. *See* Namespaces  
Align property, 686  
All Code, 563  
All Languages folder, 135  
All\_Code, 587  
code group, 598  
group, 595  
Allocation. *See* Objects  
AllowMargins property, 322  
Alphabetic character, 173  
AlternatingItemStyle property,  
482  
Anchor  
property, 301, 442, 664  
styles, 302  
Anchoring, 300. *See also*  
Controls  
AnchorStyles enumeration  
values, 664  
AND operator, 666  
**714 Index**  
Any (keyword), 671, 676  
API, 9. *See also* Graphics  
Design Interface;  
Metadata  
calls, 265  
data types. *See* Windows  
declarations, 661  
programming, 8

- set, 52
- AppBindingMode element, 48
- AppDomain, 50
- Appearance, 130
  - property, 686
- Application-related information, 306
- Applications. *See* Client; Intranet; Multi-tier applications; Selfdescribing applications; Single-tier applications
  - coding, 573
  - configuration files, 626–627
  - creation. *See* Multiple Document Interface
  - deployment, 616, 629–638, 642
  - FAQs, 643–646
  - solutions, 641–642
- development, 516–518
- domains, 83–84
- host, 106
- interoperability, 694–695
- model, 270–275, 340
- upgrading. *See* Visual Basic Architecture, considerations. *See* Migration
- Archive attribute, 394
- Arrays, 187–192, 215, 659–660. *See also* Dynamic arrays; Multidimensional arrays; Single dimension arrays
  - declaration, 188–189
  - location, 271
  - re-dimensioning, 674
- .ASAX, 467
- ASCII characters, usage, 360
- ASCII files/format, 411
- ASCII-readable characters, 243
- ASP. *See* Active Server Pages
- ASP.NET. *See* Active Server Pages
  - handler, 505
- Assemblies. *See* Resource;

Satellite assemblies  
access, 57  
binding, 46  
cache, 45. *See also* Global  
assembly cache  
code. *See* Unmanaged  
assembly code  
compiling, 45, 567  
creation, 39–51, 86  
dependencies, 41, 55–56  
enabling, 559  
enumeration, 57  
files. *See* Private assembly  
files; Shared assembly  
files  
granting, 559  
identification, metadata  
(usage), 53  
identity, 259  
location, 46–51. *See also*  
Module/assembly  
location  
probing/codebase,  
usage, 49–50  
manifest, 618  
members. *See* Local  
assembly members  
name, 40, 463  
options. *See* .NET  
passport, 618  
references, 260. *See also*  
External assembly references  
versioning, 621–623  
Assembly Generation (A1)  
utility, 637  
AssemblyRef, 46  
Assert method, 540  
Assert Override method,  
572–574  
Assertions, 525, 540–541  
Asymmetric key algorithm,  
602  
At Startup Show, 112  
Attacks, target, 577  
Attributes, 57–58  
Authentication, 557. *See also*  
Windows  
modules, 83  
type, 585  
Authorization, 557–558  
Auto Hide, 123

Automatic resource management,  
34, 62  
reliance, 68–78, 87–88  
AutoSize property, 353

## **B**

B2B. *See* Business-tobusiness  
BackColor property, 396  
BackgroundImage property,  
396  
Backward-compatible class,  
59  
Bandwidth, 427  
Base class, 203  
**Index 715**  
/baseaddress option, 546  
BeforeClosing event, 103  
Behavior, 130  
Beta testing, 547  
Bin directory, 476, 516  
Binary file, 241  
BinaryReader, 241  
BinaryWriter, 241  
Bindable attribute, 495  
BindData procedure, 444  
Binding. *See* Data; Variables  
class, 657  
mode, 48  
BindingPolicy, 48, 49  
BindingRedir element, 48  
Bindings, 130  
Bitmap, 388  
Bitwise operations, 20, 666  
Boolean, 134  
type, 20  
value, 181, 707  
variables, 659  
BooleanSwitch, 540  
Borders  
adjustments, 321  
changing. *See* Forms  
BorderStyle property,  
289–290, 395  
Bound controls, 478  
Boundary. *See* Reference;  
Security; Type; Version  
BoundColumn control,  
483, 484  
Boxes, 415. *See also* Dialog  
boxes; Drop-down  
boxes

creation. *See* Dialog boxes  
displaying. *See* Message  
Branching. *See* Multiple  
branching  
Breakpoints, 525, 528,  
531–532  
BringToFront method, 304  
Browser-compliant HTML,  
654  
Browsers, 464, 508  
independence, 133  
BSTRs, 667  
/bugreport option, 545  
Bugs. *See* Syntax-related  
bugs  
reporting, 538  
types, 524–525  
Build errors, locating, 123  
Build objects, 101  
BuildEvents, 101  
Built-in commands, customization,  
137–139  
Built-in controls, 348–399,  
407  
Built-in customization, 100  
Built-in data types, specification.  
*See* Common  
Language Runtime  
Bulleted paragraphs, 367  
Business logic, 655  
Business-to-business (B2B),  
504  
Button control, 361–363,  
664, 686  
Button1\_Click() method,  
533  
Button-click event, 511  
Buttons property, 332  
ByRef mechanism/modifier,  
672, 674  
ByVal mechanism/modifier,  
672, 674, 685

## **C**

C, 601  
exposure, 60  
usage, 67  
C#, 9, 37, 105, 133, 262,  
654  
class, 237  
usage, 220

C++, 8–9, 19, 37, 262, 601.  
*See also* ISO C/C++  
classes, 172  
client application, 260  
concept, 57  
DLL function declaration,  
270  
environment. *See* Visual  
C++  
exposure, 60  
function pointers, 233  
Java, relationship, 20  
(language), 18  
programmers, 9, 175  
usage, 67, 220  
users, 270  
/c parameter, 637  
Cabinet (CAB) files,  
632–633  
creation, 620  
Caches. *See* Assemblies  
services, 466  
usage. *See* Global assembly  
cache  
Call  
parameters. *See* Functions  
statement, 273, 670  
usage, 671  
Call Stack, 540  
Callable Wrapper. *See*  
Component Object  
Model; Runtime  
Callable Wrapper  
Caller identity, usage, 559  
Calling chain, 560  
Calling code, 669  
class, 57  
CAML, 38  
Capital letters, 489  
Caption property, 353, 363,  
686, 687  
**716 Index**  
Carriage-return delimited,  
243  
CAS. *See* Code access  
security  
Case statement, 183. *See also*  
Select Case statement  
Caspol.exe. *See* Code Access  
Security Policy utility  
Casting, 18–19. *See also*

Type  
Catch  
block, 103, 691, 692  
(keyword), 211  
set. *See* Try/catch  
statement, 103, 104, 691  
wrapper, 60  
Category, 123  
CCW. *See* Component  
Object Model  
Callable Wrapper  
CDBl, 272  
Certificate authority, 602  
CFG file, 569, 573  
CFirst  
class, 402  
instance, 403, 406  
object, 401  
CFirstLib class library, 400  
CharacterCase property, 687  
Check boxes, group, 371  
CheckBox control, 364–365  
CheckBox property, 393  
Checked, 123  
Checked property, 364  
CheckedListBox control,  
374–376  
CheckedState property, 364  
CheckOnClick property,  
375  
Checks, overriding. *See*  
Security  
Child  
code groups, 562  
forms, creation. *See*  
Multiple Document  
Interface  
relationship. *See* Parentchild  
relationship  
windows, 120–123, 297  
Class API, 623  
Class ID (CLSID), 56, 678  
Class (keyword), 198  
Class Library, projects, 116  
creation, 400  
Class Viewer, usage. *See*  
Windows forms  
ClassAct class, 571  
ClassActing class, 571  
Classes, 198–202, 226. *See*  
*also* Abstract classes;

Base class; Images;  
Managed wrapper  
classes; Wrapper  
classes  
contracts, 54  
loader, 79  
module, 196  
organization, namespace  
system (usage), 64–65,  
87  
properties, addition, 402  
Class\_Terminate method,  
542  
CLI. *See* Command-line  
interface  
Click event, 356, 363  
handler, 361  
Clickable headers, 378  
Client  
applications, 464, 653,  
655–656  
platform, 464  
processors, 513  
Client/server applications,  
465, 655  
Client-side cursor, 695  
Clipboard, 349  
object, 664  
Ring, 119–120  
CLng function, 19  
Close() method, 542  
Close/dispose, 276  
CLR. *See* Common  
Language Runtime  
/cls option, 545  
clsCalc class, 699, 702, 703  
CLSID. *See* Class ID  
CMyButton control, 405  
CMyButtonLib class library,  
404  
COBOL/Cobol, 38  
Code. *See* DataBinding  
code; Generated code;  
Malicious code;  
Packaging code;  
Source code;  
Spaghetti code  
access permissions, 555  
addition, 526  
annotating, 126–127  
behind, 473

block, 192, 447  
editor. *See* Visual Studio  
.NET  
customization, 135  
elements. *See* Hide/show  
code elements  
execution, 288  
control, 528  
groups, 559, 562–564,  
587. *See also* Child  
structure, matching,  
593–600  
identity, 559, 561–562  
impersonation, 581, 582  
**Index 717**  
locating, 126  
objects, 101  
optimization, 541–546,  
551  
paths, 683  
protection, 558  
reusing, 492  
sandboxing, 555  
segment, 139  
tracing, 683  
view pane, 531  
window, 276, 297  
Code access security (CAS),  
80, 554, 558–578,  
608–609  
Code Access Security  
Policy utility  
(caspol.exe), 585, 630  
CodeAccessPermission, 578  
CodeBase attribute, 48  
Codebase, usage. *See*  
Assemblies  
CodeBehind file, 448  
COFF. *See* Common  
Object File Format  
Collections, 246–248, 266,  
275. *See also* Garbage  
collection  
objects. *See* Project  
Color property, 314  
ColorDialog control,  
313–315  
Columns property, 316, 380  
COM. *See* Component  
Object Model  
ComboBox, 117

- control, 381–387
- Command
  - addition. *See* Toolbars
  - customization. *See* Builtin
  - commands
    - objects, 101, 421–425
    - window, 534–536
  - CommandBarEvents, 102
  - CommandEvents, 102
  - Command-line interface (CLI), 585
  - Command-line parameters, 116
  - Command-line programs, 60
  - Comment tokens, 126
  - Common Language Runtime (CLR), 2, 8–10, 256, 259–261, 630–631
    - breaking, 624
    - built-in data types, specification, 173
    - checking, 622, 626
    - FAQs, 279–281
    - history, 8–9
    - installation, 630
    - introduction, 35–37, 85–86, 257–258
    - operation, 558
    - probing ability, 50
    - reflection API, 338
    - rules, 83
    - security features, 554
    - sharing, 34
    - solutions, 278–279
    - support, 621
    - updating, 71
    - usage, 29, 656, 678, 692
  - Common Object File Format (COFF), 63
  - Common Type System (CTS), 8, 257, 269–273, 278–279
    - matching, 648
    - usage, 65–68, 87
  - Compatibility
    - phases, 621
    - version, 59
  - CompactView option, 118
  - Compare function, 209

CompareValidator, 488  
Compilation. *See*  
Conditional compilation;  
Dynamic compilation  
Compile errors, locating,  
123  
Compilers, 20–22, 30, 674.  
*See also* Just-in-time  
architecture. *See* Visual  
Basic  
errors, 649  
options, 544–546  
usage. *See* Integrated  
Development  
Environment  
Compile-time errors, 275  
Complete Word window,  
288  
Complex data binding, 333  
Component Object Model  
Callable Wrapper  
(CCW), 264, 270,  
679, 682–683  
Component Object Model  
(COM), 6, 681  
classes, 56  
COM+  
applications, 557  
component, 418, 555,  
638, 655–656, 678  
component, 578, 677  
controls, 118  
developers, 274  
interop. *See* Unmanaged  
COM interop  
library file, 55  
object, 104, 199, 258–259,  
509  
reference counting, 276  
**718 Index**  
Services, 258  
type library, 55, 339, 680  
Components, 119. *See also*  
Windows Forms  
architecture, 257,  
259–261, 278  
coding/compiling. *See*  
.NET  
creation, 403. *See also*  
Custom Windows  
file-copy-based deployment,

653  
registration, 678  
testing, 402–403, 405–406  
usage, 403, 406  
Composite custom control,  
creation, 497–504  
CompositeCustomControl,  
503, 517  
Compound expressions, 181  
Computational processes,  
262  
Concurrent users, 548  
Condition, 184  
Conditional compilation,  
525, 536–537  
constants, 538  
Conditional statement, 185  
Configuration, 466  
files, 47–49, 627. *See also*  
Administrator;  
Applications;  
eXtensible Markup  
Language; Machine;  
Security  
creation, 623–624  
hierarchy, 488  
objects. *See* Window  
section, 623  
Connected layer, 417–427,  
455  
Connection. *See* Database;  
SqlClient connections  
maintenance, 413  
pooling, 420  
strings, 418–419. *See also*  
SQL Server  
creation, 419  
Console Application  
projects, 116  
Console I/O, 62–63  
#Const (keyword), 537  
Constants, 175–176  
usage, 649, 652–653  
Constructors, 201–202  
addition, 401, 405  
ContextMenu property, 686  
Contracts, usage, 54–55  
Control Importer, usage. *See*  
Windows forms  
Control library project,  
creation, 404

ControlEvidence, 561  
Control-menu box, 305  
ControlPrincipal, 581  
Controls. *See* Bound controls;  
Button control;  
Custom controls;  
Data; Images; Intrinsic  
controls; Label control;  
Program flow  
control; Textbox control;  
User; Validation  
controls; Windows  
Forms  
addition. *See* Forms;  
World Wide Web  
forms  
anchoring, 664–665. *See*  
*also* Forms  
creation. *See* Composite  
custom control;  
Custom Web form  
controls; Custom  
Windows; World  
Wide Web forms  
deployment, 638–639, 642  
docking. *See* Forms  
positioning. *See* Forms  
property name, changes,  
685–687  
usage. *See* DataGrid  
visual layering, 304  
Convergence, 9–10  
ConvertAssemblyToTypeLib,  
680  
ConvertTypeLibToAssembly,  
680  
CookieAuthentication  
Module, 466  
CORBA, 6  
Counter, 548  
CPU cycles, optimization,  
439  
CPU/RAM usage  
situation, 64  
systems, 63  
CreditsCorrect function, 25  
Cross-language integration,  
260, 271  
capabilities, 269  
Cross-language support, 65  
Cross-platform development,

35

Crypto section, 623

CryptoAPI, 600, 601

Cryptographic Service  
Providers (CSPs), 600,  
601

Cryptography, 600–603, 610

CsharpProjectEvents, 103

CSng, 272

CSPs. *See* Cryptographic  
Service Providers

CSS, 126

CStr, 272

ctr (variable), 185

CTS. *See* Common Type  
System

**Index 719**

Cursor. *See* Client-side cursor;  
Server-side cursor

location, 414, 695

property, 687

types, 414

Custom controls, 476,  
487–488

Custom parameters, array,  
109

Custom permissions, 556,  
559, 576–578, 595

Custom token, setup, 124

Custom Web form controls,  
creation, 520

Custom Windows  
components, creation,  
399–403, 407

controls, creation,  
403–406, 408

Customer class, 230, 231

CustomerID, 479, 490–492,  
513

orders, 514

retrieval, 502

returning, 498, 518

usage, 481, 485

Custom-made controls, 348

CustomPrincipal, 579

CustomValidator, 488  
control, 491, 492

## **D**

DAO

applications, 657

code, 651  
controls, 657  
Data, 130. *See also* In-memory  
data; Relational  
data  
access, 270  
applications, 651, 653,  
656–657  
changes. *See* Visual Basic  
.NET  
performing, 262  
adapter object, 695  
binary format, 414  
binding, 332–338, 343.  
*See also* Complex data  
binding; Simple data  
binding  
data sources, 333–334  
controls, 117, 440–453,  
456  
conversion, 19  
entry, speeding, 361  
files, 241–243  
navigation, 695–696  
source, 427, 440  
binding, 332  
states, 431–432  
streams, 221  
types, 26–27, 270,  
657–662. *See also* API;  
Decimal data type;  
Double data type;  
Long data type;  
Parameters; Primitive  
data types; Return;  
Short data type; Userdefined  
data type;  
Variant data type;  
Windows  
specification. *See*  
Common Language  
Runtime  
support, 578  
verification, 412  
validation. *See* eXtensible  
Markup Language  
Data Access Components 2,  
7, 93  
Data Encryption Standard  
(DES), 601  
Data Form Wizard, usage,

334–338  
Data Signature Algorithm  
(DSA), 601  
Data Type Definition  
(DTD), 412  
Database  
connection, 422  
connectivity, 538  
counterparts, 428  
layer, 655  
size, 78  
DataBinding code, 452  
Data-bound form, 334, 335  
DataColumn, 434  
object, 694  
Datafield, 483  
DataGrid, 410, 431,  
440–446, 478. *See also*  
WebForm  
customization, 517  
DataGrid control  
customization, 482–487  
tag, 483  
usage, 478–482  
DataList, 410, 446–450  
DataReader, 425–426  
DataRelation, 429–430  
usage, 441–446  
DataRow, 431, 434  
DataSet, 425–427, 433, 440  
creation, 439, 453  
object, 438, 439  
usage, 428–435  
Dataset, 334, 335, 338, 694  
object, 693  
DataSetCommand, usage,  
695. *See also*  
Populating  
DataSetView, 440  
DataTable, 415, 434, 440  
object, 694  
DataTypes, 424  
DataView, 440  
**720 Index**  
DataXmlNavigator object,  
412  
Date data type, usage, 649,  
652  
Dates, 658–659  
DateTimePicker control,  
391–394

- DBMS, 695
- DCOM. *See* Distributed Component Object Model
- DDE. *See* Dynamic Data Exchange
- Deallocation. *See* Objects
- DebitsCorrect function, 25
- Debug
  - class, 683
  - menu, 528–529
- Debugger
  - attachment, 679. *See* External process
  - objects, 101
- Debugging, 103–104, 467.  
*See also* Projects
- concepts, 524–541, 551
- FAQs, 552
- introduction, 524
- services, 262
- solutions, 551

- Decimal data type, 658
- Decimal values, usage, 355
- DecimalPlaces property, 387
- Decision making, 172
- Declarative security, 559, 564–565
  - support, implementation, 578
- Declare statement, 671
- Def statement, 667
- Default (keyword), 697
- Default parameter, 676, 689
- Default properties, 27, 668, 687–689
  - /define option, 546
- Delegates, 226, 232–236, 266. *See also* Multicast delegates; Simple delegates
  - signature, 236
- DELETE statement, 421
- DeleteCommand, 423
- Demands, 556
- Deny override method, 574–576
- Deployment, 465, 467. *See also* Side-by-side deployment
  - unit, 41
- DES. *See* Data Encryption

- Standard
  - Description, 123
  - property, 690
- DESCryptoServiceProvider, 601
- Design, 130, 272
  - surface, 469
- Design-time properties, 129, 348
- DesktopLocation property, 294
- Destructors, addition, 401, 405
- Deterministic finalization, 68, 276. *See also* Nondeterministic finalization
- Development accelerators.  
*See* Integrated Development Environment
- Development Tool Environment (DTE)
- events, 102
- extension, 104
- object, 109
- DHTML. *See* Dynamic HTML
- Dialog boxes, 270–271, 305–323, 342
  - creation, 322–323
- Dialog Editor, 119
- DialogResult property, 686
- Dictionary object, 487
- Digital signature, 40, 639
- Dim
  - (keyword), 174. *See also* Redim
- statement, 176, 188
- Dir() function, 239
- Directory Entry, 119
  - listing, 239–241, 240
- Searcher, 119
- DisabledLinkColor property, 354
- Disconnected access, 695
- Disconnected layer, 427–435, 455–456
- Discovery, 510
- Dispose. *See* Close/dispose method, 682

Distributed applications, 17  
Windows forms, usage,  
513–518, 520  
Distributed architecture,  
440  
Distributed Component  
Object Model  
(DCOM), 2, 6, 504  
Distributed Network  
Architecture (DNA),  
6  
model, 2  
Divide-by-zero exceptions,  
211  
DivideInteger function, 675  
DivideLong function, 675  
**Index 721**  
DLL. *See* Dynamic Link  
Library  
DNA. *See* Distributed  
Network Architecture  
Do Until statement, 184  
Do While (keyword), 184  
Dock property, 303, 687  
Docking, 122, 300  
Document windows, 122  
DocumentClosing event,  
102  
DocumentEvents, 102  
DocumentOpened event,  
102  
DocumentOpening event,  
102  
DocumentSaved event, 102  
Do..Loop structure, 573  
Domain up-down control,  
384  
Domains. *See* Applications  
policy. *See* Applications  
DomainUpDown control,  
384–386  
dotNETRedist, 630  
Double data type, 652, 658  
Double-click event, 361  
Download cache, 45  
Drawing, 267–268  
Drop-down boxes, 415  
Drop-down list style, 381  
Drop-down menus, 330  
Drop-down style buttons,  
330

DSA. *See* Data Signature  
Algorithm  
DSACryptoServiceProvider,  
601  
DTD. *See* Data Type  
Definition  
DTE. *See* Development  
Tool Environment  
Dyalog APL, 38  
Dynamic arrays, 191–192  
Dynamic compilation, 654  
Dynamic Data Exchange  
(DDE), 6, 664  
Dynamic Help window, 112  
Dynamic HTML  
(DHTML), 490  
applications, 14, 112, 653,  
655  
Dynamic Link Library  
(DLL), 18, 21, 222,  
225, 542  
avoidance, 467  
base address, 546  
breaking, 42  
classes, 224  
compiling, 678  
files, 36, 44, 61  
function declaration. *See*  
C++  
hell, ending/escape, 58,  
259, 653  
need, 617  
observing, 473  
problems, 58–59  
Dynamic reference, 46

## **E**

Early binding, 436. *See also*  
Variables  
E-commerce application,  
464  
EditCommandColumn,  
445, 446  
Eiffel, 38  
Else  
block, 179  
statement, 178, 179  
ElseIf statement, 178  
Else...If statement, 182  
E-mail  
change, 603

- folders, 369
- manager application, 352
- /embed parameter, 637
- Empty (keyword), 690
- Encapsulation, 196–198
- End function, 193
- End Function statement, 697
- End If statement, 135
- End Interface statement, 697
- End Property statement, 684
- End Sub statement, 697
- Enhanced scalability, 133
- Enhanced state management, 133
- Enterprise
  - files, 627
  - security level, 563, 586
- Entities, 698
- Entries, sorting, 123
- Entry points, 41
- Enumerations, 226
- EnvDTE assembly, 109
- Environment automation model. *See* Integrated Development Environment
- Environment
  - E-procurement, 504
- Error. *See* Compile-time errors; Logic errors; Runtime errors
  - description, 62
  - handling, 172, 210–212, 216
  - understanding, 690–693
  - usage, 692–693
- locating. *See* Build errors; Compile errors
  - message, 670
- Error-checking options, 545
- Error-handling code, 525
- 722 Index**
- Event-driven model, 466
- Event-driven programming model, 467
- EventLog, 119
- EventLogTraceListener, 538
- Events, 56, 232–236, 266.  
*See also* Forms;

- Properties methods
  - and events
  - contract, 55
  - manipulation, 133
  - objects, 101
  - programming, 236
  - statement, 697
- Everything permission set, 586
- Exception
  - class, 61
  - handling, 60–62, 103–104.  
*See also* Structured exception handling
  - usage, 210
  - window, 525, 532–534
- Exception-based format, 648
- EXE. *See* ActiveX; Standard EXE
- Executable code, 635
- Executables. *See* Portable executables
- compiling, 20–21
- Execute method, 109
- Execution
  - environments. *See* Managed execution environments
  - permission set, 569, 586
- Exit Do statement, 185
- Exit function, 24, 193
- Exit Sub statement, 669, 670
- Expressionlist, 183
- Extensibility. *See* Platform models, 100. *See also* Visual Studio .NET
- eXtensible Markup Language Data Reduced (XDR), 139, 412
- eXtensible Markup Language Schema Definition (XSD), 412
- schema, 119
- tool, usage, 416–417, 455
- XSD.EXE, 438
- eXtensible Markup Language (XML),

126, 439, 656  
data validation, 416  
documents, 411, 415  
encoding/decoding,  
implementation, 578  
files, 47, 466, 617  
leveraging, 415  
overview, 411–412,  
454–455  
schema files, 412  
support, 15, 414–415  
tags, 17  
usage, 410. *See also*  
Populating  
XML-coded configuration  
files, 593, 623  
XML-coded file, 592  
XML-coded permission  
sets, 570, 588, 589  
XML-coded resource file,  
637  
eXtensible Stylesheet  
Language (XSL),  
411–412  
External assembly references,  
52  
External procedure  
declaration, 668, 671  
External process, debugger  
(attachment), 541

## **F**

Family access, 57  
FAT16 file system, 100  
FAT32 file system, 100  
Fields, 53, 198, 334  
File. *See* Active Server  
Pages; Data;Text  
appending, 246  
I/O, 243, 266–267  
management. *See* Visual  
Basic  
manipulation. *See* Images  
name variables, 242  
opening, 129  
operations, 221, 239–246,  
266  
options. *See* Output  
table, 259–260  
version information, 58  
File Signing tool (signcode).

exe), 639  
FileCodeGroup, 594  
FileIOPermission, 564–565,  
569–570, 575,  
590–591  
permission, 577  
FileIOPermissionAttribute,  
591  
Filename filter string, 310  
File-opening logic, 307  
File-saving logic, 310  
FileStream, 267  
FileSystemObject, 240  
FileSystemWatcher, 119  
Filter  
property, 311  
string. *See* Filename filter  
string  
Finalization, 542  
**Index 723**  
Finalize method, 76, 682.  
*See also* Raw finalize  
method  
Finally  
block, 103, 691, 692  
statement, 104  
FindEvents, 102  
Firewalls, 15  
Fixed-length strings, 23,  
660–661  
Floating toolbar, 136  
Focus, 130  
Font property, 275  
FontDialog control,  
311–313  
For loops, 178, 186–187,  
189, 196  
For...Each...Next loop, 187  
Form class, 294  
Format property, 391, 393,  
394  
Form.paint event, 249  
Form.PrintForm method,  
662  
Forms. *See* Multiple  
Document Interface;  
Windows forms;  
World Wide Web  
forms  
arranging/determination.  
*See* Active MDI child

- forms
  - borders, changing, 289–291
  - controls
    - addition, 300–305, 341
    - anchoring, 301–303
    - docking, 303–304
    - positioning, 304–305
    - creation. *See* Multiple Document Interface;
    - Windows forms Designer, 526
    - displaying. *See* Modal forms; Modeless forms; Multiple Document Interface;
    - Top-most forms
  - events, 294–297, 341
  - layout toolbar, 130–132
  - location, setting, 292–294
  - menus, addition, 323–327
  - objects, layering, 304
  - properties, 271–275. *See also* Windows forms
  - resizing, 291–292
  - status bars, addition, 328–330, 342
  - submission, 468
  - toolbars, addition, 330–332, 343
- Framework. *See* .NET framework
- permissions, 577
- SDK, 420
- security, 80–84
- Free store/freestore, 69
- Free threading, 262–264, 267
- Friend
  - (keyword), 684
  - modifier, 697
- FromFile method, 388
- FromODate method, 659
- FrontPage 2000
- Server Extensions, 99
- Web extensions client, 93
- FtpChannel, 600
- FullTrust permission set, 569, 586
- Function
  - definition, 235

signature, 674  
statement, 697  
Functionality, sharing, 238  
Functions, 192–196, 215  
calls, parameters, 273  
differentiation, 675  
overloading, 674  
procedures, overloading,  
675  
values, 24–25

## **G**

GAC. *See* Global Assembly  
Cache  
Gacutil. *See* General  
Assembly Cache  
utility  
Garbage Collection (GC),  
8, 62, 69, 544  
advantage, 682  
managed heap, interaction,  
71–78  
usage, 257, 262, 274–279  
Garbage Collector, 258,  
274, 544  
GC. *See* Garbage Collection  
GDI+. *See* Graphics Design  
Interface  
General Assembly Cache  
utility (Gacutil), 619,  
620, 631  
Generated code, 498  
Generation 0, 77, 275  
Generation 1, 77, 275  
Generation 2, 77, 275  
Generations, 274  
assignment, 77  
GenericIdentity object, 581  
GenericPrincipal, 579–583  
Get  
method, 230  
statement, 26  
GET protocol. *See*  
HyperText Transport  
Protocol  
GetOrders, 508, 509  
function, 511, 513  
**724 Index**  
GIF, 253  
GIF icon, 388  
Global Assembly Cache

(GAC), 45, 51, 55,  
622–623  
deployment, 40  
usage, 50, 631  
Global variables, 25  
Global.asax, 463  
GoSub statement, 666, 669  
Goto statement, 666  
Grants, 556  
Graphical user identifications  
(GUIDs), 36  
Graphical user interface  
(GUI), 51, 348  
design, 364  
Graphics  
commands, 663  
display, 396  
Graphics Design Interface  
(GDI+)  
engine, 221  
functions, 267  
Windows API, 267  
Group radio buttons, 365  
GroupBox control, 364,  
396–397  
GUI. *See* Graphical user  
interface  
GUIDs. *See* Graphical user  
identifications

## **H**

HACK, 127  
Handles (keyword), 236,  
475  
Hanging indents, 367  
Hardware-specific machine  
code, 21  
Hash, 602  
Hash algorithm, 602. *See*  
*also* One-way hash  
algorithm; Secure  
Hash Algorithm  
Haskell, 38  
Headers, 378. *See also*  
Clickable headers;  
Non-clickable  
headers  
Headers/footers, 321  
HeaderStyle property, 378,  
482  
Heap. *See* Managed heap

- Height property, 291, 482
- Help filter, 112
- /help option, 546
- HelpLink, 62
- Hidden attribute, 394
- Hide method, 275
- Hide/show code elements, 132–133
- Hierarchical system, 64
- High-performance machines, 464
- High-volume transaction, 464
- HiveKey, 593
- HKEY\_LOCAL\_MACHINE, 591
- Home page, 110–112
- Horizontal control, 328
- HotTrack property, 399
- HREF link, 634
- HRESULTS, 36
- HTML. *See* HyperText Markup Language
- HTTP. *See* HyperText Transport Protocol
- Hyperlinks, 96, 112
- HyperText Markup Language (HTML), 119, 126, 139, 461.  
*See also* Browsercompliant HTML;  
Dynamic HTML  
code, 471, 500, 501  
controls, 117  
elements, 476, 478, 500  
extensions, 14  
files, 47  
generation, 467, 504  
output, 133, 445  
rendering, 463  
sending, 497  
server controls, 477  
source, 481  
table, 443, 482  
usage, 15, 411, 439, 450
- HyperText Transport Protocol (HTTP), 16, 504
- HTTP-GET protocol, 505, 508, 509
- HTTP-POST protocol,

505, 509  
methods, 656  
network traffic, 465  
protocol, 16, 460, 695  
remoting channels, 440  
usage, 17

## I

ICalculator interface, 700  
ID attribute, 140  
IDE. *See* Integrated  
Development  
Environment  
Identifier type characters,  
175  
Identity, 51. *See also* Code  
permissions, 556  
IDTextExtensibility2 interface,  
104  
#if statement, 537  
**Index 725**  
If...Then...Else statement,  
178–183  
IIS. *See* Internet  
Information Server  
ILDASM. *See* Intermediate  
Language  
Disassembler  
IList interface, 441  
IListSource interface, 441  
Image property, 389  
ImageAlign (property), 687  
ImageList control, 369  
Images, 253–256  
class, 388  
controls, 664  
cropping, prevention, 389  
files, manipulation, 253  
Imaging namespace, 221  
Imperative security, 559,  
564–565  
Implements  
interface, 13  
(keyword), 229, 696, 698  
statement, 229, 698  
Imports  
command, 226–228  
(keyword), 226–229, 265  
statement, 226–228, 241,  
256–257  
Increment property, 387

Indents, 367. *See also*  
Hanging indents  
Index property, 687  
InFile, 245  
Information box, 376  
Informational version, 59  
InheritAct permission, 571  
Inheritance, 11, 196–197.  
*See also* Pseudo-inheritance  
demand, 571  
support, 10  
usage, 12  
Inherits  
(keyword), 229  
statement, 697  
Initialization code, 235  
In-memory copy, 426  
In-memory data, 51, 52  
In-memory manifest items,  
optimization, 52  
In-memory type library,  
680  
InnerException, 61–62  
Input/output (I/O), 60. *See*  
*also* File  
INSERT statement, 421  
InsertCommand, 423  
Insertion point, 383  
Instance ID, 401  
intClassInstanceCount, 401  
Integers, 658, 661  
Integrated Development  
Environment (IDE),  
2, 100–136, 143, 461.  
*See also* Visual Basic  
capability, 123  
changes, 92  
commands, execution, 534  
compiler, usage, 20  
configuration, 120  
customization, 135–141,  
143  
debugging tools. *See*  
Visual Basic .NET  
development accelerators,  
5  
environment automation  
model, 100–104  
examination, 3–5, 28  
execution, 438  
features, 5

- improvements, 3–4
- opening, 117
- starting, 107
- usage, 470, 495, 706
- Integration testing, 547
- Intellisense, 134–135
- Interface, 226
  - contract, 54
  - defining, 260
  - implementation, 229–232, 266
  - model, 10
  - name, 52
  - statement, 229, 696, 697
  - type, 53
  - upgrading, 696–703
  - visibility, 52
- Intermediate Language Disassembler (ILDASM), 680
- Internationalization projects, 704
- Internet
  - applications, 17, 57, 653–655
  - permission set, 569, 586
  - protocols, 504
- Internet Explorer
  - version 5.5, 633–634
  - version 6.0, 93
- Internet Information Server (IIS), 99, 600
  - applications, 653, 654
- Internet\_Zone, 597
  - code, 599
- Interoperability. *See* Applications; Language
  - managed code, interaction, 263–264
- Interoperation. *See* COM+
  - 1.0 security
- Interval property, 663
- intNextInstanceID, 401
- 726 Index**
- Intranet
  - applications, 17, 653–655
  - zone, 563
- Intrinsic controls, 476–478
- Intrinsic datatypes, 542
- Intrinsic functions, colorcoding,

527

Invoke method, 232

I/O. *See* Input/output

IPermission interface,  
implementation, 577

Is (operator), 180

IsAnonymous, 585

IsAuthenticated, 578  
field, 584

IsGuest, 585

IsInRole method, 584

IsNull function, 23

ISO C/C++, 35

IsSystem, 585

IsValid property, 489

isValid property, 492

Items

addition/removal, 118

collection, 333

marking, 123

property, 385

renaming, 118

repositioning, 119

sorting, 118

view, choice, 118

IUnrestrictedPermission  
interface, implementation,  
577

## **J**

Java

exposure, 60, 272

(language), 18

relationship. *See* C++

JavaScript.NET, 9

JIT. *See* Just-in-time

JOIN clause, 694

JPEG, 253

formats, 388

JPG image, 637

Jscript.NET, 37

Just-in-time (JIT), 465

code, 40

compilation, 58

phase, 558, 570

compiler, 34, 63–64

debugger, 541

debugging, 679

updating, 71

## **K**

Kerberos, 585

usage, 600  
version 5.0, 557  
Key pair, 602  
Keyboard scheme, 111  
/keycontainer option, 545  
Key-value pairs, 689  
Keywords. *See* Any;  
Handles; Imports  
changes, 665–667  
color-coding, 527

## **L**

Label control, 351–353, 686  
usage, 358  
Language  
interoperability, 237–238,  
266  
variable, 537  
Large heap, creation, 71  
LargeChange property, 390,  
391  
Last in first out (LIFO), 61  
Last known good system, 58  
Late binding. *See* Object  
usage, 650  
Late-bound objects, 649,  
687  
Layering, 300  
Layers. *See* Connected layer;  
Disconnected layer  
Layout, 130. *See also*  
Window  
toolbar. *See* Forms  
LayoutMDI method, 299  
LBound function, 189  
Left property, 305  
Legacy applications, 649  
Legacy-code platform,  
usage, 79  
Legacy-code-based platform  
invoke, 79  
Let  
procedure, 689  
statement, 26  
Let/Set statements, 26  
LevelFinal, 599, 600  
Libraries  
projects, creation. *See*  
Class Library; Control  
library project  
references. *See*

- Unmanaged libraries
- LIFO. *See* Last in first out
- Like (operator), 180
- Line code, 508
- Line graphics, 249
- Lines property, 687
- LINK reference, 638
- LINK tag, 633, 634
- LinkArea property, 355
- LinkArea.X property, 355
- LinkBehavior property, 356
- LinkColor property, 354
- LinkLabel Control,  
354–357
- LinkVisited property, 356
- Index 727**
- List style. *See* Drop-down  
list style
- Listbox, 117
- ListBox control, 371–381
- ListItems property, 378
- ListView control, 376–381
- ListView option, 118
- LiteralControl API, 500
- Load event, 475
- Loader optimization, 39
- Local assembly members, 52
- LocalIntranet permission  
set, 569, 586, 588–589
- LocalIntranet\_Zone  
group, 595  
permission, 600
- Location, 39
- object, 293, 305
- option, 44
- property, 293, 304
- setting. *See* Forms  
structure, 293, 305
- Lock  
implementation, 696  
types, 414
- Logic errors, 525
- Logical operations, 666
- LogonUser, 582
- Long data type, 658
- Loop While, usage, 186
- Looping, 172
- Loops. *See* For loops;While  
loops  
counter, 186
- Lower-level languages, 9

Lset statement, 666–667

## **M**

Machine

code. *See* Hardware-specific

machine code

configuration files,

624–625

files, 627

policy, 1237

security level, 563, 586

Machinewide code cache,

45

Macros, 109–110

Mail server, 352

MainMenu control, 323

Maintaining state, 415–416

Major/minor revision, 59

Makecab.exe, 620, 632

Malicious code, 555

Managed code, 36, 586

interaction. *See*

Interoperability

unmanaged code, contrast,

257, 261–264, 278

Managed execution environments,

654

Managed extensions, 37

Managed heap, 69–71

interaction. *See* Garbage

collection

Managed providers, 410,

418

usage. *See* SQL Server

Managed types, 56

Manifest. *See* Standalone

manifest

custom attributes, 43

items, optimization. *See*

In-memory manifest

items

usage, 42–45

Margin adjustments, 321

Marshalling, 681. *See also*

ADO

Master-detail relationship,

696

MaxDate property, 394

Maximize button, 305

MaximizeBox property, 291

MaxLength property, 359

MD5. *See* Message Digest 5  
MD5CryptoService  
Provider, 601  
MDI. *See* Multiple  
Document Interface  
MDILayout enumeration,  
299  
Members. *See* Local assembly  
members; Public  
members; Shared  
members  
defining, 54  
listing, 134  
Membership condition, 562  
usage, 563  
Memory  
leaks, 10  
repair, 72  
Management, 8  
Memory-intensive objects,  
78  
Menus  
addition. *See* Forms  
creation/usage, 323–327,  
342  
dynamic creation, 326  
enhancements, 326  
Mercury, 38  
Message, 61, 62. *See also*  
Error  
boxes, displaying, 306  
digest, 602  
Queue, 119  
Message Digest 5 (MD5),  
601  
MessageBox class, 323  
Metadata, 36, 41, 260,  
679–680, 683. *See also*  
Type  
APIs, 52  
benefits, 52  
**728 Index**  
insertion, 51  
mark, 53  
storage, 42  
understanding, 51–59, 86  
usage. *See* Assemblies  
MethodInfo object, 570  
Methods, 53, 56. *See also*  
Properties methods  
and events

- addition, 200–201
- contract, 54
- definition, 54
- implementation, 668–676
- Microsoft Intermediate Language (MSIL), 21, 34, 63–64, 87
- code. *See* PE MSIL code
- conversion. *See* Native code
- format, 39
- relation, 79
- usage, 68
- Microsoft Management Console (MMC), 585, 595
- Microsoft Message Queue Services (MSMQ), 262
- Microsoft Transaction Server (MTS), 6, 15, 655–656
- Middle-tier components, 348, 655
- Migration, architecture consideration, 653–657
- MinDate property, 394
- Minimize button, 305
- MinimizeBox property, 291
- Minimum property, 387
- Miscellaneous options, 546
- ML, 38
- MMC. *See* Microsoft Management Console
- Modal forms
  - creation, 287
  - displaying, 288
- Modeless forms
  - creation, 287
  - displaying, 289
- Modifiers property, 686, 687
- Module/assembly location, 57
- Modules, 221–222, 265
  - enumeration, 57
- Mondrian, 38
- MousePointer property, 663, 687
- MSDN, 111–112, 133, 535, 599
- subscribers, 596

- .msi files, creation, 620
- MSIL. *See* Microsoft Intermediate Language
- MSMQ. *See* Microsoft Message Queue Services
- MTS. *See* Microsoft Transaction Server
- Multiassembly, 46
- scenarios, 40
- Multicast delegates, 236
- MultiColumn property, 371
- Multidimensional arrays, 189–191
- Multidomain host setting, 39
- Multidomain setting, 39
- Multifile assembly, 44
- MultiLine property, 359, 399
- Multiple branching, 666
- Multiple Document Interface (MDI), 120
  - applications, 325
  - creation, 297–300, 341
  - child forms, 297, 325
  - arranging. *See* Active MDI child forms
  - creation, 298–300
  - determination. *See* Active MDI child forms
  - displaying, 298–299
  - forms, 270
  - mode, 120–123
  - parent form, creation, 297, 298
  - windows, 271
- Multiple selections, 129
- Multiply function, 195
- Multi-terabyte server, 435
- Multithreaded applications, 221
- Multi-tier applications, 348, 655–656
- Multiuser environments, 538
- MustInherit, usage, 205
- MustOverride
  - member, 205

modifier, 698

## **N**

Name

changes. *See* Controls;

Properties

property, 350, 663

value pairs, 411

variables. *See* File

Named permission set, 588

Namespaces, 13, 51,

222–226, 265. *See also*

System;

System.Drawing;

System.Threading;

System.Windows

alias, setting, 497

creation, 222–226

**Index 729**

drawing, 248–261, 266

programmatic side, 220

root, 222

system, usage. *See* Classes

Naming, 39

Narrowing, 19

Native code, MSIL code

conversion, 262

Navigation. *See* Data

.NET. *See* Visual Basic

.NET

architecture, 7

assembly options, 545

class, 65

code execution, 35

component, coding/compiling,

678

graphics, 516

history, 6

installation. *See* Visual

Studio .NET

internal functionality, 35

platform, object-oriented

nature, 265

servers, 8

.NET Framework, 6–8, 29,

33, 93

classes, 8

configuration, 623–630,

641

definition, 34–35, 85

FAQs, 88–89

- installation, 415
- introduction, 34
- solutions, 85–88
- usage, 306, 410, 511
- validation, 489
- .NET programming fundamentals, 171
- FAQs, 217
- introduction, 172–173
- solutions, 214–216
- NetCodeGroup, 594
- .NET-compliant languages, 37
- .NET-compliant programming languages, usage, 37–39, 86
- Netscape, 460
- Network resources, 694
- traffic. *See* HyperText Transport Protocol
- New () procedure, 201
- New (keyword), 200, 201, 681
- Node images, 369
- Non-clickable headers, 378
- Non-deterministic finalization, 276
- Nonpublic types, enumeration, 57
- Nonspaghetti code, 466
- Non-TrueType font, 663
- Normal attribute, 394
- Normal mode, 48
- NorthWind database, 478
- /nostdlib option, 545
- NOT operator, 666
- Notepad, 241
- Nothing permission set, 569, 586
- NotOverridable (keyword), 205
- nSize parameter, 662
- NTLM, 557, 585
- usage, 600
- Null (keyword), 651, 690
- usage, 668, 690
- NULL propagation, 23–24
- Null Propagation, avoidance, 648, 650–651

Number property, 690  
Number-of-windows property,  
200  
Numeric field, 391  
NumericUpDown control,  
386–387  
NumInstances variable, 206

## O

Oberon, 38  
Object, 65. *See also* Add-ins;  
Build objects;  
Clipboard; Code;  
Command;  
Component Object  
Model; Debugger;  
Events; Identity;  
Project; Properties;  
System; Window  
allocation/deallocation,  
275–276  
data types, 175, 187, 657,  
671  
destruction, 74  
implementation, 274  
late binding, 649  
layering. *See* Forms  
manual termination, 75  
orientation, history, 13  
placement, 70  
type, 53  
variable, 650  
OBJECT reference, 638  
OBJECT tag, 634  
Object-based components,  
460  
Object-based methodology,  
554  
Object-oriented (OO)  
compliant syntax, 273  
Object-oriented (OO)  
concepts, 10–11  
**730 Index**  
Object-oriented (OO)  
design, advantages,  
11–12  
Object-oriented (OO)  
functionality, 196  
Object-oriented (OO) languages,  
10, 257, 259,  
691

- Object-oriented (OO)
  - nature. *See* .NET
- Object-oriented (OO)
  - principles, 11
- Object-oriented programming (OOP), 65, 172, 196–206, 215–216
  - language, 196
- Object-oriented-based (OO-based) language, 10–13, 29
- OCX, 18
- ODBC providers, 418, 436
- OLE Container control, 663
- OLE DB, 418, 420, 436
  - libraries, 426
- OLEDB provider, 410
- OleDbCommand, 421, 424
- OleDbConnection, 418, 420, 480
- OleDbDataAdapter, 480
- On Error Goto, 103
  - statement, 210
- OnAddInsUpdate method, 104
- OnBeginShutdown
  - event, 102
  - method, 104
- OnCancelCommand, 445
- OnChange event, 102
- OnConnection method, 104
- OnDisconnection method, 104
- OnEditCommand, 445
- One-way hash algorithm, 602–603
- OnMacrosRuntimeReset
  - event, 102
- OnPagePrint, 258
- OnStartupcomplete event, 102
- OnStartupComplete
  - method, 104
- OnUpdateCommand, 445
- OO. *See* Object-oriented
- OOP. *See* Object-oriented programming
- Opacity property, 275
- Open File dialog box, 308,

- 309, 369
- Opened event, 103
- OpenFileDialog control, 306–308
- OpenType font, 663
- Optimization. *See* Code FAQs, 552
- introduction, 524
- options, 544
- solutions, 551
- OPTION BASE command, 23
- Option Base statement, 659–660, 666
- Optional function parameter, 195
- Optional parameters, 202, 668–669, 676
- /optioncompare option, 546
- /optionexplicit option, 546
- Options. *See* Compilers; Error-checking options; Miscellaneous options; .NET; Optimization; Output file options; Preprocessor options
- /optionstrict option, 546
- OR operator, 666
- OrderDetail tables, 428
- /out parameter, 637
- OutFile, 245
- Outlook bar, 307
- Output
  - directory, 707
  - file options, 544–545
  - parameters, 424
- OutputWindowEvents, 102
- Overload procedures, 674
- Overloaded functions, 134, 232
- Overloading, 202–203, 668, 674–676. *See also* Functions
  - implementation, 676
- OverLoads (keyword), 697
- Overloads (keyword), 202, 203
- Overridable (keyword), 203, 204. *See also*

NotOverridable  
Overrides  
(keyword), 203  
modifier, 698  
Overriding, 203–205  
Oz, 38

## **P**

P1, 1216  
P1A, 1216  
Packaging code, 618–623,  
641  
Page directive, 473  
Page object, inheriting, 474  
Page orientation, 321  
Pagelets, 493  
Page\_Load event, 444, 452  
**Index 731**  
PageSetupDialog control,  
321–322  
Paging, 486  
PaneAdded event, 102  
PaneClearing event, 102  
Panel control, 394–396  
PaneUpdated event, 102  
ParamArray, 668, 672–674  
function parameter, 195  
parameters, 672, 676  
Parameters, 667. *See also*  
Command-line  
parameters; Default  
parameter; Functions;  
Optional parameters;  
Output; ParamArray;  
Passing parameters  
array, 109. *See also* Custom  
parameters  
data types, 674  
information, 134  
modifiers, 674  
names, 674  
number, 674  
object, 423  
order, 674  
passing methods, 542–543  
Parent  
class, functionality, 13  
form, 298  
creation. *See* Multiple  
Document Interface  
Parent-child relationship,

- 428–430
- Partial references, usage, 49
- Pascal, 38
- Passing methods. *See*
- Parameter passing methods
- Passing parameters, 668, 672
- Password text box, 357
- PasswordChar property, 359
- PE MSIL code, 41
- Peek() method, 259
- Performance
  - Counter, 119
  - monitoring, 548–549
- PerformClick event, 363
- Perl, 38
- Permission, 79, 555–556.
  - See also* Code;
  - Custom permissions;
  - Identity; Role-based security; Security class, design, 577
  - demands, 559, 570–571, 573
  - granting, 556
  - level, 82
  - policy, 80
  - requests, 41, 81, 559, 565–570
  - sets, 560. *See also* eXtensible Markup Language; Internet;
  - Named permission set creation, 588–593
  - management/configuration, 559
- PermissionState, 584
- PermitOnly, 572
- override, 576
- Person class, 230, 231
- PictureBox control, 388–389
- Platform
  - extensibility, 262
  - independence, 15
  - invoke. *See* Legacy-codebased platform invoke
  - Platform-specific code, 63
- PMEs. *See* Properties methods and events
- Pointer

creation, 70  
type, 53  
Policy. *See* Machine policy;  
Security; Users  
assemblies, 587  
level, 599  
utility. *See* Code access  
Polymorphism, 196, 197  
Pooling. *See* Connection  
Populating. *See*  
Programmatic populating  
DataSetCommand, usage,  
432–433  
XML, usage, 433–434  
Portable executables, 618  
POST protocol. *See*  
HyperText Transport  
Protocol  
Preprocessor options, 546  
PreRender trigger, 475  
Preserve (keyword), 192  
Primary keys, 430  
PrimaryKey property, 426  
Primitive data types, 241  
Princauthenticated, 584  
Principal, 556–557,  
578–582  
policy, 80  
usage, 82–83  
PrincipalObject, 581, 583  
PrincipalPermission, 578,  
583  
objects, 584  
PrincState, 583  
Print() method, 256  
PrintDialog control,  
315–316  
PrintDocument object, 256  
Printer object, 256  
Printing, 256–261, 268–269  
namespace, 221  
PrintPage  
**732 Index**  
event, 256  
method, 257  
PrintPreviewDialog control,  
316–321  
Print-related settings, 315  
Private  
(keyword), 684  
modifier, 697

- Private assemblies, 626
- files, 51
- Private key, 602, 603
- PrivateGroup, 597, 599
- PrivatePath attribute, 47, 48, 50
- privatePath directories, 622
- PrivatePermissions, 595
- Probing, usage. *See* Assemblies
- Procedure
  - calls, 668, 670–671
  - declaration. *See* External procedure declaration
  - variables, 667
- Process, 119
  - control, performing, 262
  - Process.Start method, 356
- Profile, 111
- Profiling services, 262
- PROGID/ProgID. *See* Programmatic identifier
- Programmatic identifier
- Program execution, tracing, 467
- Program flow control, 178–187, 214–215
- Programmatic functionality, 348
- Programmatic identifier (PROGID/ProgID), 64, 678
- Programmatic populating, 434–435
- Programming. *See* Events
  - concepts. *See* Advanced programming concepts
  - differences, 668–690
  - languages
    - usage. *See* .NET-compliant programming languages
    - users. *See* Third-party programming language users
- Project Explorer, 127
- ProjectAdded event, 103
- ProjectRemoved event, 103
- ProjectRenamed event, 103
- Projects. *See* Class Library; Console Application
  - projects; Startup; Windows

- collection objects, 101
- debugging, 116, 536
- files, list, 707
- name, 707
- options, 112–116
- Propagation. *See* NULL propagation
- Properties, 25–27, 53, 56, 668, 684–689. *See also* Default properties; Forms; Opacity property; TopMost property; Windows forms
  - addition, 198–200. *See also* Classes
  - contract, 55
  - grouping, 696
  - listing, 130
  - names, 689
  - changes, 668. *See also* Controls
- Controls
  - objects, 101
  - procedures
  - syntax, 689
  - usage, 684–685
  - windows, 129–130
- Properties methods and events (PMEs), 339
- Property statement, 684, 697
- Protected
  - (keyword), 684
  - modifier, 697
- Proxy class, 510–511
- Pseudo-inheritance, 196
- Public
  - (keyword), 684
  - modifier, 697
- Public access, 57
- Public Constructor, inclusion, 56
- Public key, 602, 603
- token, 622
- Public members, 57
- Public types, 57
  - enumeration, 57
- Publicly declared variables, 530
- Python, 38

## Q

QFE. *See* Quick Fix  
Engineering  
QueryCloseSolution event,  
103  
Quick Fix Engineering  
(QFE), 48, 50, 59, 621  
Quick Launch toolbar, 617  
Quickinfo, 135  
Quotation marks, usage, 360

## **R**

RAD. *See* Rapid  
Application  
Development  
**Index 733**  
RadioButton control,  
365–367  
Random number generation,  
1240  
Random Number  
Generator (RNG),  
601  
RangeValidator, 488  
Rapid Application  
Development (RAD),  
461  
Raw finalize method, usage,  
76  
RC2. *See* Rivest's Cipher 2  
RC2CryptoServiceProvider,  
601  
RCW. *See* Runtime  
Callable Wrapper  
RDBMS, 436  
RDO  
applications, 657  
code, 651  
controls, 657  
README file, 369  
ReadOnly  
attribute, 394  
(keyword), 200  
Read-only rich text box,  
348  
Read-only text box, 348,  
359  
Read-only up-down control,  
386  
Recessed border, 305  
Recordset, 184, 694  
Redim (keyword), 191

- ReDim statement, 674
- Reference
  - counting, 69
  - locating, 46
  - scope boundary, 41
  - type, 53
  - usage. *See* Weak references
- Reflection, 56–58
  - API. *See* Common Language Runtime
  - emit service, 52
  - service, 52
  - ReflectionPermission,
    - usage, 57
  - Registry, 555
    - dependence, 42
    - keys, 573
    - usage, 36, 259
  - RegistryPermission, 565, 570, 590
  - Regression testing, 548
  - REGTOOL, 228
  - RegularExpressionValidator, 488, 490
  - Relational data, 414, 440
  - Relational schema, 428–430
  - Remoting, 262, 439–440, 456. *See also* ADO.NET
    - channels. *See* HyperText Transport Protocol
    - section, 623
  - Remove method, 208, 373
    - /removeintchecks option, 545
  - Renamed event, 103
  - Repeater, 410, 450–453
    - control, 478
  - Report view, 376, 378
  - Representation, gaining, 82–83
  - RequestMinimum, 566
  - RequestOptional, 566
  - RequestRefuse, 566
  - Requests, 556
  - RequiredFieldValidator, 488, 490
  - Resgen. *See* Resource Generator
  - Resource, 618
    - assembly, 636

files, 634–638. *See also*  
eXtensible Markup  
Language  
Resource Generator  
(Resgen), 637  
Resource X Generator  
(Resxgen), 637  
Resurrection, 76  
Resxgen. *See* Resource X  
Generator  
Return  
command, 24  
(keyword), 193  
statement, 668–670, 676  
value, data type, 662  
values/types, 674  
Revert method, 572  
RevertAssert, 572  
RevertDeny, 572  
RevertPermitOnly, 572  
RFC 3075, 601  
RichTextBox control,  
367–369  
Right property, 305  
Rivest Shamir Adleman  
(RSA), 602  
Rivest's Cipher 2 (RC2),  
601  
RNG. *See* Random  
Number Generator  
RNGCryptoService  
Provider, 601  
Role-based security, 80, 81,  
554, 565, 578–585,  
609  
checks, 583–585  
permissions, 556  
usage, 600  
Role-based validation, 579  
Role-based verification,  
580, 581  
Rows property, 316  
**734 Index**  
RPC ports, 414  
RSA. *See* Rivest Shamir  
Adleman  
RSACryptoServiceProvider,  
602  
Runat clause, 471, 477  
Runtime. *See* Common  
Language Runtime

- code, 41
- errors, 524–525
- files. *See* Setup runtime files
- performance issues, 524
- properties, 129, 348
- requirement, 9–10
- section, 623
- Runtime Callable Wrapper (RCW), 263–264, 679, 681–683

## **S**

- Satellite assemblies, 635
- Save As Type box, 310, 311
- SaveFileDialog control, 309–311
- ScaleMode property, 663
- Schedule, 119
- Schema files. *See* eXtensible Markup Language
- Scheme, 38
- Scope, 173, 188, 530
  - boundary. *See* Reference
- Screen size/location, 293
- Scripting Runtime Library, 239
- Scripting.FileSystemObject object, 239
- ScrollBars property, 359
- SDK. *See* Software development kit
- SDL. *See* Service Description Language
- Secure Hash Algorithm 1 (SHA1), 602
- Security, 17–18, 30, 464, 466, 553. *See also*
  - Code access security;
  - Declarative security;
  - Imperative security;
  - Role-based security
- boundary, 41
- checks
- completion, 572
- overriding, 559, 572–576
- concepts, 555–558, 607–608
- configuration files, 627–630

- FAQs, 611–613
- features. *See* Common Language Runtime
- level. *See* Enterprise; Machine; User
- permissions, 51
- policy, 80, 83–84, 558, 585–600, 609–610
  - level, 598
- remoting, 600
- section, 623
- services, 79–84, 88, 262
- solutions, 607–610
- tools, 603–605, 610
- security.config file, 593
- security.config.cch file, 593
- security.config.old file, 593
- SecurityPermission, 581, 590
- SECUTIL, 18
- SelColor property, 369
- Select Case statement, 182–184
- Select statement, 178, 482
- SelectCommand, 423, 424
- SelectedIndex property, 373
- Selection. *See* Multiple selections
- SelectionEvents, 102
- SelectionLength property, 360–361
- SelectionMode property, 373
- SelectionStart property, 360–361
- Self-describing applications, 58
- Self-describing code, 126
- SelFont property, 369
- SelFontSize property, 369
- SendToBack method, 304
- Server. *See* Multi-terabyte server; World Wide Web
- applications, 653, 655–656
- controls. *See* Active Server Pages; HyperText Markup Language
- process, 681
- Server-side cursor, 695
- Service Controller, 119

Service Description  
Language (SDL),  
509–510  
Services, usage. *See*  
Namespaces  
Session keys, 602  
Set  
(keyword), 198  
method, 230  
statement, 26, 684  
argument, 685  
usage, 688  
Setup files, 41  
Setup runtime files, 93  
SHA1. *See* Secure Hash  
Algorithm 1  
**Index 735**  
SHA1CryptoService  
Provider, 602  
ShadowCopy attribute, 47  
Shared assembly files, 51  
Shared members, 205–206,  
221  
Short circuiting, 25  
Short data type, 658  
Shortcut keys, customization,  
135–136  
Show method, 289, 306  
ShowColor function, 233  
ShowDialog method, 288,  
308, 323  
ShowNetwork property,  
322  
Side-by-side deployment,  
58–59  
Side-by-side execution  
units, 41  
Signcode.exe. *See* File  
Signing tool  
Simple data binding,  
332–333  
Simple delegates, 235  
Simple Object Access  
Protocol (SOAP),  
16–17, 415, 439, 505,  
508  
usage, 510  
SimpleCustomControl, 495,  
496  
Single dimension arrays,  
190, 440

- Single domain setting, 39
- Single-tier applications, 653, 656
- Size
  - object, 292
  - property, 292
  - SizeMode property, 389
- SkipVerification permission set, 569, 586
- SLN file type, 115
- SmallChange property, 390
- SmallTalk, 38
- sn.exe utility, 618
- SOAP. *See* Simple Object Access Protocol
- Software development kit (SDK), 45, 636. *See also* Framework
- Software Publisher Certificate (SPC), 638
- Solution Explorer, 127–129, 487
  - window, 274
- SolutionEvents, 102
- SomeSub, 262
- Sorting property, 378
- Sound files, 635
- Source code, 536
- Spaghetti code, 461
- Span tag, 472
- SPC. *See* Software Publisher Certificate
- SQL managed provider, 433
- SQL namespace, 480
- SQL Server, 410, 418
  - connection, 443
  - string, 419
  - managed provider, usage, 435–439, 456
  - setup, 480
- SQL statement, 422, 425
- SqlClient connections, 420
- SqlCommand, 421, 435
- SqlConnection, 435, 480
  - object, 437
- SqlDataAdapter, 435
  - usage, 441
- SqlDataReader, 435, 439
- SqlDataSetCommand, 480
- SSL, usage, 596
- Stack walking, 559–561,

572

StackTrace, 61

Standalone manifest, 44

Standard EXE, 112

Start page, customization,  
139–141

StartPosition property, 292

Startup

projects, 129

section, 623

State bag, 133

State management, 466. *See*  
*also* Enhanced state  
managed

Statements, 184, 198

Static modifier, 668, 669,  
676

Static reference, 46

Static variables, 25

Status bars, addition. *See*

Forms

StatusBar control, 328

Step clause, 186

Stored procedures, 421

str (variable), 208

StreamReader, 267

object, 257, 259

Stress testing, 548–549

Strikeout, 311

String, 508, 543–544. *See*  
*also* Connection;

Filename filter string;

Fixed-length strings

buffer, 667

class, 209

handling, 206–209, 216

initialization, 662

object, intrinsic type conversion

method, 273

padding, 667

StringBuilder object, 543

### **736 Index**

Strong name, 562

Strong typed language,  
465–466

Strong typing, 173

StrPtr, 667

Structured exception handling,  
103

Structures, 176–177, 214  
(keyword), 176

- usage, 177
- Sub New procedure, 682
- Sub statement, 697
- subOne method, 229
- SUO file type, 115
- Switch classes, 540
- Symmetric key algorithm, 602
- SyncLock, 263–264
- Syntax-related bugs, 524
- System
  - controls, 117
  - namespace, 257, 264–269, 278, 690
  - derivation, 477
  - object, 34
  - services, usage, 60–63, 86–87
  - System attribute, 394
  - System.Array object, 188
  - System.Collections namespace, 689
  - System.Diagnostics namespace, 356
  - System.Drawing
    - namespace, 253, 268
    - object, 248–250
  - System.Drawing.Imaging namespace, 253
  - System.Drawing.Printing namespace, 256, 268
  - System.Drawing.Printing
    - .PrintPageEventArgs, 258
  - System.Exception class, 692
  - System.IO
    - class, 220
    - namespace, 226, 239–241, 245
    - System.IO.Directory class, 240
    - System.IO.File, 245
    - System.IO.StreamReader
      - class, 246
      - object, 245
    - System.IO.StreamWriter
      - class, 246
      - object, 245
    - System.Object, 201, 375–376
    - System.Security

- .Cryptography, 601
- System.Security
  - .Cryptography.X509
  - certificates, 601
  - System.Security
    - .Cryptography.Xml, 601
    - System.Text.StringBuilder, 543
    - System.Threading namespace, 262
    - System.Web.HttpContext, 487
    - System.Windows namespace, 226

## T

- TabControl control, 397–399
- Tables, 334, 336
  - collection, 430–431
- TablesCollection object, 694
- TabPage property, 397
- Tabs
  - addition/removal, 118
  - displaying, 118
  - hiding, 118
  - renaming, 118
- Task list, 123–127
  - filtering, 123
- TaskList
  - views, 124–126
  - window, 126
- TaskList by Priority, 123
- TaskListEvents, 102
- TCP, 439, 440
  - port 80, 505
- TDS. *See* TypedDataSet
- Template tags, 449, 452
- TemplateColumn, 484, 485
- Terminate event, 276
- Testing. *See* Beta testing;  
Integration testing;  
Regression testing;  
Stress testing; Unit testing
- FAQs, 552
- introduction, 524
- phases, 546–549
- solutions, 551

strategies, 546–549, 551  
Text  
boxes, 351. *See also*  
Password text box;  
Read-only rich text  
box; Read-only text  
box  
files, 243–246  
property, 333, 385  
Text Editor folder, 135  
TextAlign property, 353  
TextBox, 472  
control, 357–361, 367,  
502, 686–687  
Textexpression, 183  
Text-only view, 376, 378  
**Index 737**  
Text-with-large-icons view,  
376, 378  
Text-with-small-icons view,  
376, 378  
TextWriterTraceListener,  
538  
Third-party controls, 654  
Third-party programming  
language users, 37  
Third-party vendors, 348  
Thread, 262  
creation, 263  
management, 262  
Threading. *See* Free threading  
ThreeDCheckBoxes property,  
375  
Throughput, 465  
Throw statement, 693  
TickFrequency property,  
391  
TickStyle property, 391  
TIF, 253  
Timer, 119  
Title bar, 305  
TLBIMP, 418  
TLBIMP.EXE, 651  
TODO, 124, 127  
Toggle-style buttons, 330  
Token, 124. *See also*  
Comment tokens;  
Public key  
property, 585  
setup. *See* Custom token  
ToOADate function, 27

- ToODate method, 659
- Tool windows, 122
- Toolbars. *See* Floating toolbar;
- Forms
  - addition, 136–137. *See also*
  - Forms
    - commands, addition, 137
    - customization, 136–137
  - Toolboxes, 116–120
- Tools. *See* Security
- Top-most forms
  - creation, 287
  - displaying, 289
- TopMost property, 275, 289
- Trace
  - class, 538, 539
  - Trace Listeners, 538, 540
  - Traces, 538–540
    - addition, 525
  - TraceSwitch, 540
  - Trace.WritelineIf method, 539
- TrackBar control, 389–391
- Transitions, 542
- TreeView control, 369–371
- TripleDESCryptoService Provider, 602
- TrueType font, 663. *See also*
- Non-TrueType font
- Trusted code, 572
- Try
  - block, 211, 691, 692
  - (keyword), 211
  - Try/catch
    - set, 60
    - statement, 61
  - Try...Catch...Finally block, 103, 533
  - Try...Catch...Finally syntax, 691
- Type, 65
  - boundary, 41
  - casting, 271–273
  - (keyword), 176
  - libraries, 259
  - metadata, 618
  - safety, 18–20, 30, 68, 558, 654
- Type Library Importer, 680
- Type Mismatch, 270
- Typed language. *See* Strong

typed language  
TypedDataSet (TDS), 412,  
436–439  
usage, 437–439, 441–446  
TypeLibConverter class, 680  
Types, 53–55  
exportation, 51

## U

UBound function, 189  
UDDI. *See* Universal  
Description Discovery  
Integration  
UDT. *See* User-Defined  
Types  
UIPermission, 559, 560,  
573, 575–576  
UNC names/path, 591  
Underline, 311  
Underscores, 173  
UNDONE, 127  
Unit testing, 547  
Universal Description  
Discovery Integration  
(UDDI), 512  
UnLoad trigger, 475  
Unmanaged assembly code,  
55–56  
Unmanaged code, 36, 55  
contrast. *See* Managed  
code  
references, 668  
Unmanaged COM interop,  
79  
Unmanaged libraries, references,  
677–683  
Unrestricted matches, 583  
UPDATE statement, 421  
UpdateCommand, 423  
Up-down control, 385–387.  
*See also* Read-only  
up-down control  
**738 Index**  
Upgrade  
error, 667  
settings, 707  
time, 707  
tool, usage, 703–707  
Wizard, usage, 655, 688,  
702–707  
UPGRADE\_ISSUE, 687,

706  
UPGRADE\_NOTE, 707  
UPGRADE\_TODO, 706  
UPGRADE\_WARNING,  
707  
URLAuthorizationModule,  
466  
URN, 62  
UseLatestBuildVersion, 48  
User. *See* Concurrent users  
controls, 493  
files, 627  
interface, 130, 464  
capabilities, 399, 403  
notes, addition, 123  
security level, 563, 586  
User-created controls, 654  
User-defined customizations,  
100  
User-defined data type, 666  
User-Defined Types (UDT),  
660, 661  
User-interface layer, 655  
User-interface object, 137  
Utilities. *See* World Wide  
Web services  
class, creation, 221  
programs, 656

## **V**

Validation controls, 476,  
488–489  
usage, 489–492  
ValidationSummary, 488  
control, 492  
Value property, 390  
Variable lengths, 25–26  
Variable lower bounds, 23  
Variable names, 530  
Variables, 25–27, 173–175,  
214, 667. *See also*  
Boolean variables;  
Global variables;  
Publicly declared  
variables; Static variables  
early binding, 648–650  
pointers, 194  
Variant data type, 23, 648,  
657  
Variants, 657–658  
VarPtr, 667

VB. *See* Visual Basic  
VBG. *See* Visual Basic Group  
VB.NET. *See* Visual Basic .NET  
editions, 92–93, 142  
installation/configuration  
FAQs, 143  
introduction, 92  
solutions, 142–143  
VBProjectEvents, 103  
VBScript, 7, 465  
VCProjectEvents, 103  
Verification. *See* Role-based verification  
process, 71  
Version  
boundary, 41  
policy, checking, 47–49  
/version option, 545  
Versioning, 463. *See also* Assemblies  
support, 59  
VES. *See* Virtual Execution System  
View property, 378  
Virtual Execution System (VES), 66, 79, 262  
VisitedLinkColor property, 354, 356  
Visual Basic Group (VBG), 115  
Visual Basic .NET (VB.NET), 34, 37, 676  
data access changes, 693–696  
FAQs, 31–32  
features, 1  
functionality, 3  
IDE debugging tools, 529  
introduction, 2–3  
items, removal, 24–25  
solutions, 28–32  
Visual Basic (VB), 654  
6.0 ActiveX DLLs, 348  
applications, upgrading  
considerations, 648–653  
FAQs, 712  
introduction, 648  
solutions, 709–712

- class, 237
- code, 9, 425, 473–476
- coders, 46
- compiler, 436
- architecture, 21
- Developer, 111
- file management, 21–22
- forms, conversion. *See* Windows forms
- history, 13
- IDE, 525
- programming tasks, 225
- runtime, 261
- version 6.0, 440, 663
- Index 739**
- changes, 23–27, 31
- interfaces, upgrading, 699–703
- Visual C++, 105
  - environment, 100
  - version 6.0, 135
- Visual C++ 6.0 component, 655
- Visual Interdev, 93
- Visual Studio IDE, 436
- Visual Studio .NET (VS.NET), 55, 417, 462
  - code editor, 132
  - command window, 110
  - editions, 92–93
  - extensibility model, 656
  - installation, 93–100, 142–143. *See also* Windows 2000
  - VPD, 14
  - .VSDISCO file, 510
- VS.NET. *See* Visual Studio .NET

## **W**

- W3C guidelines, 433
- Watches, 525, 529–530
- Weak references, usage, 77–78
- WebBrowser control, 663
- WebClasses, 14, 112
  - applications, 654
- Web.config, 487
- WebControl, 495
- WebForm

- control, 446
- DataGrid, 444
- WebMethod attribute, 506
- Web-style links, 354
- Where clause, 428
- While loops, 178, 184–186
- While...Wend syntax, 184
- Widening, 19
- Width property, 291, 482
- Win forms, 119
- Win32 native applications, debugging, 656
- Win32API, 36
- Window
  - configuration objects, 101
  - layout, 111–112
  - style, 130
  - types, 122
- WindowEvents, 102
- Windows. *See* Child; Document windows; Tool windows
- API. *See* Graphics Design Interface
- data types, 661–662
- arrangement, 123
- authentication, 580
- components
- creation, 399–403. *See also* Custom Windows
- update, 94
- Control projects, 116
- controls, creation. *See* Custom Windows
- Windows 2000
- Component services, 420
- domain, 557
- environment, 556, 557
- operating system, 100
- servers, 602
- Service Pack 2, 93
- Visual Studio .NET, installation, 99–100
- Windows CE, 6, 63
- Windows Form Designer, 222
- Windows Forms, 475
- ActiveX Control Importer, usage, 338–339, 343–344
- applications, 354

- Class Viewer, usage, 338
- components/controls, 347
- FAQs, 408
- solutions, 407–408
- contrast. *See* World Wide
- Web forms
  - controls, 117
  - creation, 287–289
  - FAQs, 344–345
  - introduction, 270
  - solutions, 340–344
  - framework, 306, 324, 332
  - implementation, 440
- Label, 351
  - manipulation, 275–294, 340–341
  - methods, 276–287
  - properties, 275–276
  - support, 663
  - usage. *See* Distributed applications
- VB forms, conversion, 662–665
- Windows Installer, 631–632
  - version 2.0, 93
- Windows NT
  - environment, 557
  - version 4.0, 93
- Windows-based applications, 226, 270
- WindowsDefaultLocation, 292
- WindowsIdentity
  - object, 579, 580, 582
  - token, 582
- WindowsImpersonation
  - Context
    - instance, 582
- 740 Index**
  - object, 582
  - WindowsPrincipal,
    - 579–581, 583
  - WindowsPrincipal.Identity
    - .IsAuthenticated, 584
  - WindowsPrincipal.Identity.
    - Name, 584
  - WindowState property, 321
  - WinMain, 41
  - WithEvents (keyword), 475
  - Wizard\_Machine\_Policy, 593–594

Wizards, 109  
usage. *See* Add-ins  
Word completion, 134  
WordWrap property, 359  
World Wide Web (WWW  
// Web)  
classes, 654  
extensions client. *See*  
FrontPage 2000  
page loads, 488  
servers, 13, 632  
site, 573, 633  
World Wide Web (WWW  
// Web) applications,  
13–17, 30, 410  
development, 459  
FAQs, 521–522  
introduction, 460  
overview, 13–14  
solutions, 519–520  
World Wide Web (WWW  
// Web) forms,  
14–15, 119, 133–134,  
461–467, 511–513,  
519  
classic ASP, contrast,  
465–467  
control, 450, 471  
addition, 467–492,  
519–520  
creation, 492–504. *See*  
*also* Custom Web  
form controls  
example, 493–497  
counterpart, 440  
creation, 462–464  
deployment, 476  
performance, improvement,  
465  
simplicity, 465  
usage, 654. *See also* Active  
Server Pages  
Windows forms  
contrast, 464–465  
controls, contrast, 476  
World Wide Web (WWW  
// Web) services,  
15–17, 460, 466,  
504–513, 520  
consuming, 511–513  
developing, 505–509

function, explanation, 505  
usage, 432, 440  
utilities, 509–511  
Wrapper classes, 263, 660.  
*See also* Component  
Object Model  
Callable Wrapper;  
Managed wrapper  
classes  
WriteXML method, 433

## **X**

x86 machine, 63  
XCOPY, 630  
XDR. *See* eXtensible  
Markup Language  
Data Reduced  
XML. *See* eXtensible  
Markup Language  
XMLDataReader, 421  
XMLNavigator object, 412  
XOR operator, 666  
XPath, 412  
XSD. *See* eXtensible  
Markup Language  
Schema Definition  
XSL. *See* eXtensible  
Stylesheet Language

### ***Train with Global Knowledge***

The right content, the right method,  
delivered anywhere in the world, to any  
number of people from one to a  
thousand. Blended Learning Solutions™  
from Global Knowledge.

### ***Train in these areas:***

Network Fundamentals  
Internetworking  
A+ PC Technician  
WAN Networking and Telephony  
Management Skills  
Web Development  
XML and Java Programming  
Network Security  
UNIX, Linux, Solaris, Perl  
Cisco  
Enterasys  
Entrust

Legato  
Lotus  
Microsoft  
Nortel  
Oracle

Only Global Knowledge offers so much content in so many formats—Classroom, Virtual Classroom, and e-Learning. This flexibility means Global Knowledge has the IT learning solution you need. Being the leader in classroom IT training has paved the way for our leadership in technology-based education. From CD-ROMs to learning over the Web to e-Learning live over the Internet, we have transformed our traditional classroom-based content into new and exciting forms of education. Most training companies deliver only one kind of learning experience, as if one method fits everyone. Global Knowledge delivers education that is an exact reflection of you. No other technology education provider integrates as many different kinds of content and delivery