Dot Net Guidelines

Version No.: 1.1

Date: 21-May-2013

Revision History

Version No.	Date	Prepared by / Modified by	Significant Changes
1.1	21-May-2013	QMG	Formatting done as per Org. standards
1.0	23-Jan-2013	QMG	QMS-IT Phase 4 release

Glossary

Abbreviation	Description

Table of Contents

1 Namespace

1.1 Guidelines

The general rule for naming namespaces is to use the company name followed by the technology name and optionally the feature and design as follows.

CompanyName.TechnologyName[.Feature][.Design]

Usage:

- Microsoft.Media
- Microsoft.Media.Design

2 Example of Using Namespace

2.1 C#

```
namespace N1.N2
{
  Class A
  {
  }
}
```

Vb.net

```
Namespace N1.N2
  Class B
  End Class
End Namespace
```

- Prefixing namespace names with a company name or other well-established brand avoids the possibility of two published namespaces having the same name. For example, Microsoft.Office is an appropriate prefix for the Office Automation Classes provided by Microsoft.
- Use a stable, recognized technology name at the second level of a hierarchical name. Use organizational hierarchies as the basis for namespace hierarchies. Name a namespace that contains types that provide design-time functionality for a base namespace with the .Design suffix. For example, the System.Windows.Forms.Design Namespace contains designers and related classes used to design System.Windows.Forms-based applications.
- A nested namespace should have a dependency on types in the containing namespace. For example, the classes in the System.Web.UI.Design depend on the classes in System.Web.UI. However, the classes in **System.Web.UI** do not depend on the classes in **System.UI.Design**.
- You should use Pascal case for namespaces, and separate logical components with periods, as in Microsoft.Office.PowerPoint. If your brand employs nontraditional casing, follow the casing defined by your brand, even if it deviates from the prescribed Pascal case. For example, the namespaces NeXT.WebObjects and ee.cummings illustrate appropriate deviations from the Pascal case rule.
- Use plural namespace names if it is semantically appropriate. For example, use System.Collections rather than System.Collection. Exceptions to this rule are brand names and abbreviations. For example, use System.IO rather than System.IOs.
- Do not use the same name for a namespace and a class. For example, do not provide both a Debug namespace and a Debug class.

3 Type Usage Guidelines

3.1 Base Class Usage Guidelines

A class is the most common kind of type. A class can be **abstract** or **sealed**. An abstract class requires a derived class to provide an implementation. A sealed class does not allow a derived class. It is recommended that you use classes over other types.

Base classes are a useful way to group objects that share a common set of functionality. Base classes can provide a default set of functionality, while allowing customization though extension.

You should add extensibility or polymorphism to your design only if you have a clear customer scenario for it. For example, providing an interface for data adapters is difficult and serves no real benefit. Developers will still have to program against each adapter specifically, so there is only marginal benefit from providing an interface.

Base Classes vs. Interfaces

An interface type is a partial description of a value, potentially supported by many object types. Use base classes instead of interfaces whenever possible. From a versioning perspective, classes are more flexible than interfaces. With a class, you can ship Version 1.0 and then in Version 2.0 add a new method to the class. As long as the method is not abstract, any existing derived classes continue to function unchanged.

Because interfaces do not support implementation inheritance, the pattern that applies to classes does not apply to interfaces. Adding a method to an interface is equivalent to adding an abstract method to a base class; any class that implements the interface will break because the class does not implement the new method.

Interfaces are appropriate in the following situations:

- Several unrelated classes want to support the protocol.
- These classes already have established base classes (for example, some are user interface (UI) controls, and some are XML Web services).
- Aggregation is not appropriate or practical.

In all other situations, class inheritance is a better model.

3.2 Sealed Class Usage Guidelines

The following rules outline the usage guidelines for sealed classes:

- Use sealed classes if it will not be necessary to create derived classes. A class cannot be derived from a sealed class.

C#

```
public sealed class Runtime
{
// Private constructor prevents the class from being created.
private Runtime();

// Static method.
public static string GetCommandLine()
{
// Implementation code goes here.
}
}
```

Vb.net

```
NotInheritable Public Class Runtime
 ' Private constructor prevents the class from being created.
 Private Sub New()
 End Sub

 ' Static method.
 Public Shared Sub GetCommandLine() As String
   ' Implementation code goes here.
 End Sub
End Class
[C#]
public sealed class Runtime
{
  // Private constructor prevents the class from being created.
  private Runtime();

  // Static method.
 public static string GetCommandLine()
 {
    // Implementation code goes here.
  }
}
```

3.3 Value Type Usage Guidelines

A value type describes a value that is represented as a sequence of bits stored on the stack. For a description of all the .NET Framework's built-in data types, see Value Types. This section provides guidelines for using the structure (**struct**) and enumeration (**enum**) value types.

3.3.1 Struct Usage Guidelines

It is recommended that you use a **struct** for types that meet any of the following criteria:

- Act like primitive types.
- Have an instance size under 16 bytes.

- Are immutable.
- Value semantics are desirable.

When using a **struct**, do not provide a default constructor. The runtime will insert a constructor that initializes all the values to a zero state. This allows arrays of structs to be created more efficiently. You should also allow a state where all instance data is set to zero, false, or null (as appropriate) to be valid without running the constructor.

C#

```
public struc Employee
{
  ' Public members, accessible throughout declaration region.
  public string FirstName;
  public string MiddleName;
  Public string LastName;
  ' Friend members, accessible anywhere within the same assembly.
  Internal int EmployeeNumber;
  Internal long BusinessPhone ;
  ' Private members, accessible only within the structure itself.
  private long HomePhone;
  Private double Salary;
  Private double Bonus;
  ' Procedure member, which can access structure's private members.
  internal CalculateBonus(short Rate)
  {
    Bonus = Salary * CDbl(Rate)
  }
}
```

Vb.net

```
Public Structure Employee
  ' Public members, accessible throughout declaration region.
  Public FirstName As String
  Public MiddleName As String
  Public LastName As String
  ' Friend members, accessible anywhere within the same assembly.
  Friend EmployeeNumber As Integer
  Friend BusinessPhone As Long
  ' Private members, accessible only within the structure itself.
  Private HomePhone As Long
  Private Salary As Double
  Private Bonus As Double
  ' Procedure member, which can access structure's private members.
  Friend Sub CalculateBonus(ByVal Rate As Single)
    Bonus = Salary * CDbl(Rate)
  End Sub

End Structure
```

3.3.2 Enum Usage Guidelines

The following rules outline the usage guidelines for enumerations:

- Use an **enum** to strongly type parameters, properties, and return types. Always define enumerated values using an **enum** if they are used in a parameter or property. This allows development tools to know the possible values for a property or parameter. The following example shows how to define an enum type.

C#

```
public enum FileMode
{
  Append,
  Create,
  CreateNew,
  Open,
  OpenOrCreate,
  Truncate
}
```

Vb.net

```
Public Enum FileMode
        Append
        Create
        CreateNew
        Open
        OpenOrCreate
        Truncate
End Enum
```

The following example shows the constructor for a **FileStream** object that uses the **FileMode** enum.

C#

```
public FileStream(string path, FileMode mode);
```

Vb.net

```
Public FileStream(path As String,mode As FileMode)
```

Use the System.FlagsAttribute Class to create custom attribute for an **enum** if a bitwise OR operation is to be performed on the numeric values. This attribute is applied in the following code example.

C#

```
public enum Bindings
{
  IgnoreCase = 0x01,
  NonPublic = 0x02,
  Static = 0x04,
  InvokeMethod = 0x0100,
  CreateInstance = 0x0200,
  GetField = 0x0400,
  SetField = 0x0800,
  GetProperty = 0x1000,
  SetProperty = 0x2000,
  DefaultBinding = 0x010000,
  DefaultChangeType = 0x020000,
  Default = DefaultBinding | DefaultChangeType,
  ExactBinding = 0x040000,
  ExactChangeType = 0x080000,
  BinderBinding = 0x100000,
  BinderChangeType = 0x200000

  }
```

Vb.net

```
Public enum Bindings
```

```
        IgnoreCase = 0x01,
        NonPublic = 0x02,
        Static = 0x04,
        InvokeMethod = 0x0100,
        CreateInstance = 0x0200,
        GetField = 0x0400,
        SetField = 0x0800,
        GetProperty = 0x1000,
        SetProperty = 0x2000,
        DefaultBinding = 0x010000,
        DefaultChangeType = 0x020000,
        Default = DefaultBinding | DefaultChangeType,
        ExactBinding = 0x040000,
        ExactChangeType = 0x080000,
        BinderBinding = 0x100000,
        BinderChangeType = 0x200000

    End enum
```

- Use an **enum** with the **flags** attribute only if the value can be completely expressed as a set of bit flags. Do not use an **enum** for open sets (such as the operating system version).
- Do not assume that **enum** arguments will be in the defined range. Perform argument validation as illustrated in the following code example.

C#

```
public  void SetColor (Color color)
{
    if (!Enum.IsDefined (typeof(Color), color)
    throw new ArgumentOutOfRangeException();
}
```

Vb.net

```
Public Sub SetColor (color As Color)

   If (Not Enum.IsDefined (typeof(Color), color) Then
          throw new ArgumentOutOfRangeException()
    End If
End Sub
```

- Use an **enum** instead of static final constants.
- Use type **Int32** as the underlying type of an **enum** unless either of the following is true:
  - The **enum** represents flags and there are currently more than 32 flags, or the **enum** might grow to many flags in the future.
  - The type needs to be different from **int** for backward compatibility.
- Do not use a nonintegral enum type. Use only Byte, Int16, Int32, or Int64
- Do not define methods, properties, or events on an enum.
- Do not use an Enum suffix on enum types

3.4 Delegate Usage Guidelines

A delegate is a powerful tool that allows the managed code object model designer to encapsulate method calls. Delegates are useful for event notifications and callback functions.

4 Convention

Name end Delgate functions with the suffix Delegate.

4.1 C#

```
Public delegate void SimpleDelegate()


void F()
{
    System.Console.WriteLine("Test.F");
}
Main()
{
   SimpleDelegate d = New SimpleDelegate(F);
   d();
}

     public void MultiCall(SimpleDelegate d, int count)
     {
       int I;
       for(i = 0;I<count;I++)
     {
      d()
     }
}
```

4.2 Vb.net

```
Delegate Sub SimpleDelegate()

Module Test

  Sub F()
    System.Console.WriteLine("Test.F")
  End Sub

  Sub Main()
    Dim d As New SimpleDelegate(F)
    d()
  End Sub
End Module

Sub MultiCall(d As SimpleDelegate, count As Integer)
    Dim i As Integer
    For i = 0 To count - 1
      d()
    Next i
  End Sub
```

Event notifications

Use the appropriate event design pattern for events even if the event is not user interface-related. For more information on using events, see the Event Usage Guidelines.

Callback functions

Callback functions are passed to a method so that user code can be called multiple times during execution to provide customization. Passing a Compare callback function to a sort routine is a classic example of using a callback function. These methods should use the callback function conventions described in Callback Function Usage.

Name end callback functions with the suffix Callback.

Callback function Usage:

Delegates, Interfaces and Events allow you to provide callback functionality. Each type has its own specific usage characteristics that make it better suited to particular situations.

Events

Use an event if the following are true:

- A method signs up for the callback function up front, typically through separate **Add** and **Remove** methods
- Typically, more than one object will want notification of the event
- You want end users to be able to easily add a listener to the notification in the visual designer

Delegates

Use a delegate if the following are true:

- You want a C language style function pointer
- You want a single callback function
- You want registration to occur in the call or at construction time, not through a separate **Add** method

Interfaces

Use an interface if the callback function requires complex behavior.

4.3 Attribute Usage Guidelines

The .NET Framework enables developers to invent new kinds of declarative information, to specify declarative information for various program entities, and to retrieve attribute information in a run-time environment. For example, a framework might define a HelpAttribute attribute that can be placed on program elements such as classes and methods to provide a mapping from program elements to their documentation. New kinds of declarative information are defined through the declaration of attribute classes, which might have positional and named parameters.

The following rules outline the usage guidelines for attribute classes:

- Add the Attribute suffix to custom attribute classes, as shown in the following example.

C# :

```
public class ObsoleteAttribute{}
```

Vb.net :

```
Public Class ObsoleteAttribute
   Inherits Attribute
   ' Insert code here.
End Class
```

- Specify AttributeUsage on your attributes to define their usage precisely, as shown inthe following example.

C# :

```
[AttributeUsage(AttributeTargets.All, Inherited = false, AllowMultiple = true)]
```

```
public class ObsoleteAttribute: Attribute {}
```

Vb.net :

```
<AttributeUsage(AttributeTargets.All, Inherited := False, AllowMultiple := True)>  _
```

- Seal attribute classes whenever possible, so that classes cannot be derived from them.
- Use positional arguments for required parameters.
- Use named arguments for optional parameters.
- Do not name a parameter with both named and positional arguments.
- Provide a read-only property with the same name as each positional argument, but change the case to differentiate between them.
- Provide a read/write property with the same name as each named argument, but change the case to differentiate between them.

C# :

```
public class NameAttribute: Attribute
{
  public NameAttribute (string username)
  {
     // Implement code here.
  }
  public string UserName
  {
    get
    {
      return UserName;
    }
  }
  public int Age
  {
    get
    {
      return Age;
    }
    set
    {
      Age = value;
    }
  }
  // Positional argument.
}
```

Vb.net :

```
Public Class NameAttribute Inherits Attribute
  Public Sub NameAttribute (username As String)

     // Implement code here.
  End Sub

  Public Property UserName As String
    Get
      Return UserName;
    End Get
```

```
        End Property

          Public Property Age As Integer
         Get

            Return Age;
          End Get
         Set(ByVal Value As String)
               Age=value
        End Set
      End Property
        // Positional argument.
    End Class
```

4.4 Nested Type Usage Guidelines

A nested type is a type defined within the scope of another type. Nested types are very useful for encapsulating implementation details of a type, such as an enumerator over a collection, because they can have access to private state.

Public nested types should be used rarely. Use them only in situations where both of the following are true:

- The nested type logically belongs to the containing type.
- The nested type is not used often, or at least not directly.

The following examples illustrates how to define types with and without nested types:

```
// With nested types.
ListBox.SelectedObjectCollection
// Without nested types.
ListBoxSelectedObjectCollection

// With nested types.
RichTextBox.ScrollBars
// Without nested types.
RichTextBoxScrollBars
```

Do not use nested types if the following are true:

- The type is used in many different methods in different classes. The FileMode Enumeration is a good example of this kind of type.
- The type is commonly used in different APIs. The StringCollection Class is a good example of this kind of type.

4.5 Class Members and its Usage

4.5.1 Property Naming Guidelines

The following rules outline the naming guidelines for properties:

- Use a noun or noun phrase to name properties.
- Use Pascal Case(The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. You can use Pascal case for identifiers of three or more characters. For example:**B**ack**C**olor)
- Do not use Hungarian notation.

Read-Only and Write-Only Properties

You should use a read-only property when the user cannot change the property's logical data member. Do not use write-only properties.

Provide a read-only property with the same name as each positional argument, but change the case to differentiate between them.

Provide a read/write property with the same name as each named argument, but change the case to differentiate between them.

C# :

```
public readonly string UserName
{
   get
   {
  return UserName;
 }
}

 public int Age
   get
   {
      return Age;
   }
```

Vb.net :

```
Public ReadOnly Property UserName() As String
   Get
 Return UserName
   End Get
 End Property

 Public Property Age() As Integer
   Get
      Return Age
   End Get
```

Indexed Property Usage

The following rules outline guidelines for using indexed properties:

- Use only one indexed property per class, and make it the default-indexed property for that class.
- Do not use nondefault indexed properties.
- Name an indexed property Item. For example, see the DataGrid.Item Property. Follow this rule, unless there is a name that is more obvious to users, such as the Chars property on the **String** class.
- Use an indexed property when the property's logical data member is an array.
- Do not provide an indexed property and a method that are semantically equivalent to two or more overloaded methods.
- Consider creating a property with the same name as its underlying type. For example, if you declare a property named Color, the type of the property should likewise be Color.

For Example:

C# :

```
public class SampleClass
{
   public Color BackColor
```

```
  {
    // Code for Get and Set accessors goes here.
  }
}
```

VB.net :

```
Public class SampleClass
  Public Sub  BackColor As Color
        // Code for Get and Set accessors goes here.
  End Sub
End Class
```

4.6 Event Naming Guidelines

- Use an EventHandler suffix on event handler names.
- Specify two parameters named *sender* and *e*. The *sender* parameter represents the object that raised the event. The *sender* parameter is always of type **object**, even if it is possible to use a more specific type. The state associated with the event is encapsulated in an instance of an event class named *e*. Use an appropriate and specific event class for the *e* parameter type.
- Name an event argument class with the EventArgs suffix.
- Consider naming events with a verb.
- Use a gerund (the "ing" form of a verb) to create an event name that expresses the concept of pre-event, and a past-tense verb to represent post-event. For example, a **Close** event that can be canceled should have a Closing event and a Closed event. Do not use the BeforeXxx/AfterXxx naming pattern.
- Do not use a prefix or suffix on the event declaration on the type. For example, use Close instead of OnClose.
- In general, you should provide a protected method called OnXxx on types with events that can be overridden in a derived class. This method should only have the event parameter *e*, because the sender is always the instance of the type.

C# :

```
public class MouseEventArgs : EventArgs
{
  int x;
  int y;
  public MouseEventArgs(int x, int y)
    {
      this.x = x;
      this.y = y;
    }
  public int X { get { return x; } }
  public int Y { get { return y; } }
}
```

Vb.net :

```
Public Class MouseEventArgs Inherits EventArgs

  Dim x As Integer
  Dim y As Integer
```

```
        Public Sub MouseEventArgs(x As Integer , y As Integer)
               Me.x = x;
              Me.y = y;
        End Sub

        Public Property X As Integer
         Get
           Return x
         End Get
        End Property
           Public Property Y As Integer
         Get
           Return y
         End Get
        End Property

      End Class
```

4.6.1 Method Naming Guidelines

The following rules outline the naming guidelines for methods:

- Use verbs or verb phrases to name methods.
- Use Pascal case

The following are examples of correctly named methods.

```
RemoveAll()
GetCharArray()
Invoke()
```

Example for Methods:

C# :

```
public void SubComputeArea(double Length, double Width)

{
  double Area ;   // Declare local variable.
   if(Length == 0 Or Width == 0)
   {
      exit;     // If either argument = 0.
   } // Exit Sub immediately.

   Area = Length * Width   ' Calculate area of rectangle.
   Debug.WriteLine(Area)   ' Print Area to Immediate window.
}
```

Vb.net :

```
Sub SubComputeArea(ByVal Length As Double, ByVal Width As Double)
   Dim Area As Double   ' Declare local variable.
   If Length = 0 Or Width = 0 Then
      ' If either argument = 0.
      Exit Sub ' Exit Sub immediately.
   End If
   Area = Length * Width   ' Calculate area of rectangle.
   Debug.WriteLine(Area)   ' Print Area to Immediate window.
End Sub
```

<u>Note:</u>

In **Vb.net** a **Sub** procedure, like a **Function** procedure, is a separate procedure that can take arguments and perform a series of statements. However, unlike a **Function** procedure, which returns a value, a **Sub** procedure cannot be used in an expression **Sub** procedures are **Public** by default. To specify a different accessibility, include **Protected**, **Friend**, **Protected Friend**, or **Private** in the declaration.

Use a method when:

- The operation is a conversion, such as **Object.ToString**.
- The operation is expensive enough that you want to communicate to the user that they should consider caching the result.
- Obtaining a property value using the **get** accessor would have an observable side effect.
- Calling the member twice in succession produces different results.
- The order of execution is important. Note that a type's properties should be able to be set and retrieved in any order.
- The member is static but returns a value that can be changed.
- The member returns an array. Properties that return arrays can be very misleading. Usually it is necessary to return a copy of the internal array so that the user cannot change internal state. This, coupled with the fact that a user can easily assume it is an indexed property, leads to inefficient code. In the following code example, each call to the Methods property creates a copy of the array. As a result, 2^n+1 copies of the array will be created in the following loop.

C# :

```
 Type type = // Get a type.
     for (int i = 0; i < type.Methods.Length; i++)
     {
               if (type.Methods[i].Name.Equals ("text"))
               {
                  // Perform some operation.
               }
     }
```

Vb.net :

```
For I = 0 TO type.Methods.Length
   If type.Methods[i].Name.Equals ("text") Then
      ‘ Perform some operation
     End If
Next
```

4.7 Method Overloading Guidelines

Method overloading occurs when a class contains two methods with the same name, but different signatures. This section provides some guidelines for the use of overloaded methods.

- Use method overloading to provide different methods that do semantically the same thing.
- Use method overloading instead of allowing default arguments. Default arguments do not version well and therefore are not allowed in the Common Language Specification (CLS). The following code example illustrates an overloaded String.IndexOf method.

4.8 Static Field Naming Guidelines

The following rules outline the naming guidelines for static fields:

- Use nouns, noun phrases, or abbreviations of nouns to name static fields
- Use Pascal case
- Use a Hungarian notation prefix on static field names
- It is recommended that you use static properties instead of public static fields whenever possible

4.9 Parameter Naming Guidelines

The following rules outline the naming guidelines for parameters:

- Use descriptive parameter names. Parameter names should be descriptive enough that the name of the parameter and its type can be used to determine its meaning in most scenarios.
- Use camel case (The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized. For example) for parameter names.
- Use names that describe a parameter's meaning rather than names that describe a parameter's type. Development tools should provide meaningful information about a parameter's type. Therefore, a parameter's name can be put to better use by describing meaning. Use type-based parameter names sparingly and only where it is appropriate.
- Do not use reserved parameters. Reserved parameters are private parameters that might be exposed in a future version if they are needed. Instead, if more data is needed in a future version of your class library, add a new overload for a method.
- Do not prefix parameter names with Hungarian type notation.

The following are examples of correctly named parameters:

C# :

```
Type GetType(string typeName)
string Format(string format, object args() )
```

Vb.net :

```
GetType(typename as String) As Type
Format(format As String, args As object) As String
```

5 Interface Guidelines

An Interface is an entity, which behaves much like a class with the exceptions that:

- it cannot implement a constructor
- it cannot implement a destructor
- it cannot defined the variables
- it can only define the prototype for functions

Convention For Interfaces

Begin interface names with the prefix "I", followed by a noun or a noun phrase, like IComponent, or with an adjective describing the interface's behavior, like ISerialize. Do not use the underscore, and use abbreviations sparingly, because abbreviations can cause confusion.

You can implement the Interface just like that.

C# :

```
Interface ISerialize{
// define the function prototype here
}
```

Example

```
Interface Iserialize
{
public int ReadFile(String) ;   // read from a text file
public int WriteFile(String) ;  // write to a text file
}
```

Vb.net:

```
  Interface Iserialize
        ‘define the function prototype here
  End Interface
```

```
Interface Iserialize
   Public ReadFile(String) As Integer ’read from a text file
   Public WriteFile(String)As Integer ‘ write to a text file
End Interface
```

Keep in mind that you just only need to define the prototypes for your functions so if any one chooses to inherit your Interface must implement the functions defined and this definitively goes in your favor.

6 Array Usage Guidelines

Arrays vs. Collections

Class library designers might need to make difficult decisions about when to use an array and when to return a collection. Although these types have similar usage models, they have different performance characteristics. You should use a collection in the following situations:

- When **Add**, **Remove**, or other methods for manipulating the collection are supported
- To add read-only wrappers around internal arrays

For more information on using collections, see Grouping Data in Collections.

Using Indexed Properties in Collections

You should use an indexed property only as a default member of a collection class or interface. Do not create families of functions in noncollection types. A pattern of methods, such as **Add**, **Item**, and **Count**, signal that the type should be a collection.

Array Valued Properties

You should use collections to avoid code inefficiencies. In the following code example, each call to the myObj property creates a copy of the array. As a result, 2^n+1 copies of the array will be created in the following loop.

C# :

```
for (int i = 0; i < obj.myObj.Count; i++)
    DoSomething(obj.myObj[i]);
```

Vb.net :

```
 For I =0 To obj.myObj.Count
    DoSomething(obj.myObj[i])
Next
```

For more information, see the Properties vs. Methods topic.

Returning Empty Arrays

String and **Array** properties should never return a null reference. Null can be difficult to understand in this context. For example, a user might assume that the following code will work.

C# :

```
public void DoSomething()
{
  string s = SomeOtherFunc();
  if (s.Length > 0)
  {
    // Do something else.
  }
}
```

Vb.net :

```
Sub DoSomething()

  Dim s As String  = SomeOtherFunc();
  If s.Length > 0

    // Do something else.
  End If
End Sub
```

The general rule is that null, empty string (""), and empty (0 item) arrays should be treated the same way. Return an empty array instead of a null reference.

7 Enumeration Type Naming Guidelines

The enumeration (**Enum**) value type inherits from the Enum Class. An **Enum** is a named constant whose underlying type is any integral type except Char. If no underlying type is explicitly declared, Int32 is used. **Enum** derives from ValueType, but is not a value type. Programming languages typically provide syntax to declare an enumeration that consists of a set of named constants and their values.

Enum provides methods to compare instances of this class, convert the value of an instance to its string representation, convert the string representation of a number to an instance of this class, and create an instance of a specified enumeration and value.

You can also treat an **Enum** as a bit field.

The following rules outline the naming guidelines for enumerations:

- Use Pascal case for **Enum** types and value names
- Use abbreviations sparingly
- Do not use an Enum suffix on **Enum** type names
- Use a singular name for most **Enum** types, but use a plural name for **Enum** types that are bit fields
- Always add the **FlagsAttribute** to a bit field **Enum** type

8 Operator Usage Guidelines

The following rules outline the guidelines for operator overloading:

- Define operators on value types that are logical built-in language types, such as the System.Decimal Structure.
- Provide operator-overloading methods only in the class in which the methods are defined.
- Use the names and signature conventions described in the Common Language Specification (CLS).
- Use operator overloading in cases where it is immediately obvious what the result of the operation will be. For example, it makes sense to be able to subtract one Time value from another Time value and get a TimeSpan. However, it is not appropriate to use the **or** operator to create the union of two database queries, or to use **shift** to write to a stream.
- Overload operators in a symmetric manner. For example, if you overload the equality operator (==), you should also overload the not equal operator(!=).
- Provide alternate signatures. Most languages do not support operator overloading. For this reason, always include a secondary method with an appropriate domain-specific name that has the equivalent functionality. It is a Common Language Specification (CLS) requirement to provide this secondary method. The following example is CLS-compliant.

C# :

```
class Time
{
  TimeSpan operator -(Time t1, Time t2) { }
  TimeSpan Difference(Time t1, Time t2) { }
}
```

Vb.net :

```
Class Time

            Public function Difference(t1 As Time , t2 As Time) As TimeSpan

            End function
    End Class
```

9 Threading Guidelines

The following rules outline the design guidelines for implementing threading:

- Avoid providing static methods that alter static state. In common server scenarios, static state is shared across requests, which means multiple threads can execute that code at the same time. This opens up the possibility for threading bugs. Consider using a design pattern that encapsulates data into instances that are not shared across requests.
- Static state must be thread safe.
- Instance state does not need to be thread safe. By default, a library is not thread safe. Adding locks to create thread-safe code decreases performance, increases lock contention, and creates the possibility for deadlock bugs to occur. In common application models, only one thread at a time executes user code, which minimizes the need for thread safety. For this reason, the .NET Framework is not thread safe by default. In cases where you want to provide a thread-safe version, use a **GetSynchronized** method to return a thread-safe instance of a type. For examples, see the System.Collections Namespace.
- Design your library with consideration for the stress of running in a server scenario. Avoid taking locks whenever possible.
- Be aware of method calls in locked sections. Deadlocks can result when a static method in class A calls static methods in class B and vice versa. If A and B both synchronize their static methods, this will cause a deadlock. You might discover this deadlock only under heavy threading stress.

Performance issues can result when a static method in class A calls a static method in class A. If these methods are not factored correctly, performance will suffer because there will be a large amount of redundant synchronization. Excessive use of fine-grained synchronization might negatively impact performance. In addition, it might have a significant negative impact on scalability

Be aware of issues with the **lock** statement (**SyncLock** in Visual Basic). It is tempting to use the **lock** statement to solve all threading problems. However, the System.Threading.Interlocked Class is superior for updates that must be made automatically. It executes a single **lock** prefix if there is no contention. In a code review, you should watch out for instances like the one shown in the following example.

C# :

```
lock(this)
{
  myField++;
}
```

Vb.net :

```
SyncLock Me
      myField += 1
End SyncLock
```

Alternatively, it might be better to use more elaborate code to create rhs outside of the lock, as in the following example. Then, you can use an interlocked compare exchange to update x only if it is still null. This assumes that creation of duplicate rhs values does not cause negative side effects.

C# :

```
if (x == null)
{
  lock (this)
  {
        if (x == null)
        {
          // Perform some elaborate code to create rhs.
          x = rhs;
        }
      }
    }
```

Vb.net :

```
If x Is Nothing Then
  SyncLock Me
    If x Is Nothing Then
      ' Perform some elaborate code to create rhs.
      x = rhs
    End If
  End SyncLock
End If
```

- Avoid the need for synchronization if possible. For high traffic pathways, it is best to avoid synchronization. Sometimes the algorithm can be adjusted to tolerate race conditions rather than eliminate them.

10 Asynchronous Programming Guidelines

Asynchronous programming is a feature supported by many areas of the common language runtime, such as Remoting, ASP.NET, and Windows Forms. Asynchronous programming is a core concept in the .NET Framework. This topic introduces the design pattern for asynchronous programming.

The philosophy behind these guidelines is as follows:

- The client should decide whether a particular call should be asynchronous.
- It is not necessary for a server to do additional programming in order to support its clients' asynchronous behavior. The runtime should be able to manage the difference between the client and server views. As a result, the situation where the server has to implement IDispatch and do a large amount of work to support dynamic invocation by clients is avoided.
- The server can choose to explicitly support asynchronous behavior either because it can implement asynchronous behavior more efficiently than a general architecture, or because it wants to support only asynchronous behavior by its clients. It is recommended that such servers follow the design pattern outlined in this document for exposing asynchronous operations.
- Type safety must be enforced.
- The runtime provides the necessary services to support the asynchronous programming model. These services include the following:
    - Synchronization primitives, such as critical sections and ReaderWriterLock instances
    - Synchronization constructs such as containers that support the WaitForMultipleObjects method
    - Thread pools
    - Exposure to the underlying infrastructure, such as Message and ThreadPool objects

11 Type Casting Guidelines

The following rules outline the usage guidelines for casts:

- Do not allow implicit casts that will result in a loss of precision. For example, there should not be an implicit cast from **Double** to **Int32**, but there might be one from **Int32** to **Int64**.
- Do not throw exceptions from implicit casts because it is very difficult for the developer to understand what is happening.
- Provide casts that operate on an entire object. The value that is cast should represent the entire object, not a member of an object. For example, it is not appropriate for a **Button** to cast to a string by returning its caption.
- Do not generate a semantically different value. For example, it is appropriate to convert a Time or TimeSpan into an **Int32**. The **Int32** still represents the time or duration. It does not, however, make sense to convert a file name string, such as, "c:\mybitmap.gif" into a **Bitmap** object.
- Do not cast values from different domains. Casts operate within a particular domain of values. For example, numbers and strings are different domains. It makes sense that an **Int32** can cast to a **Double**. However, it does not make sense for an **Int32** to cast to a **String**, because they are in different domains.

Boxing

Boxing is an implicit conversion of a value type to the type **object** or to any interface type implemented by this value type.

For Example :

C# :

```
Implicit Conversion (No loss of data)
 Int I ; short j=20;
 I=J;
```

Vb.net :

```
Implicit Conversion (No loss of data)
 Dim I As Integer ;Dim j as Short =20;
 I=J;
```

C# :

```
Explicit Conversion(Loss of data may occur)
Int I = 3000
Short j = I ;
```

Vb.net :

```
Explicit Conversion(Loss of data may occur)
Dim I As Int  = 3000
Dim j As Short  = I ;
```

Conversion from value type to reference type (Boxing)

C#

```
Short j;
```

```
Object o;
O=j;
```

Vb.net :

```
Dim j As Short
Dim o As Object ' Value type base object type
o = j ' BOXING occurs here
```

12 Error Handling Guidelines

The following rules outline the guidelines for raising and handling errors:

- All code paths that result in an exception should provide a method to check for success without throwing an exception. For example, to avoid a **FileNotFoundException** you can call **File.Exists**. This might not always be possible, but the goal is that under normal execution no exceptions should be thrown.
- End **Exception** class names with the Exception suffix as in the following code example:

C# :

```
public class FileNotFoundException : Exception
{
    // Implementation code goes here.
}
```

Vb.net :

```
 Public Class FileNotFoundException
   Inherits Exception
  ' Implementation code goes here.
End Class
```

- Use the three common constructors shown in the following code example when creating exception classes in C# and the Managed Extensions for C++.

C# :

```
public class XxxException : ApplicationException
{
  XxxException() {... }
  XxxException(string message) {... }
  XxxException(string message, Exception inner) {... }
 }
```

Vb.net :

```
Public Class XxxException
        Inherits ApplicationException

    Public Sub New()
      ' Implementation code goes here.
    End Sub

        Public Sub New(message As String)
          My.Base.New(message)
         End Sub

         Public Sub New(message As String, inner As Exception)
          My.Base.New(message, inner)
         End Sub
  End Class
```

- In most cases, use the predefined exception types. Only define new exception types for programmatic scenarios, where you expect users of your class library to catch exceptions of this new type and perform a programmatic action based on the exception type itself. This is in lieu of parsing the exception string, which would negatively impact performance and maintenance.

For example, it makes sense to define a FileNotFoundException because the developer might decide to create the missing file. However, a FileIOException is not something that would typically be handled specifically in code.

- Do not derive new exceptions directly from the base class Exception.
- Group new exceptions derived from the base class **Exception** by namespace. For example, there will be derived classes for XML, IO, Collections, and so on. Each of these areas will have their own derived classes of exceptions as appropriate. Any exceptions that other library or application writers want to add will extend the **Exception** class directly. You should create a single name for all related exceptions, and extend all exceptions related to that application or library from that group.
- Use a localized description string in every exception. When the user sees an error message, it will be derived from the description string of the exception that was thrown, and never from the exception class.
- Create grammatically correct error messages with punctuation. Each sentence in the description string of an exception should end in a period. Code that generically displays an exception message to the user does not have to handle the case where a developer forgot the final period.
- Provide exception properties for programmatic access. Include extra information (other than the description string) in an exception only when there is a programmatic scenario where that additional information is useful. You should rarely need to include additional information in an exception.
- Do not use exceptions for normal or expected errors, or for normal flow of control.
- You should return null for extremely common error cases. For example, a **File.Open** command returns a null reference if the file is not found, but throws an exception if the file is locked.
- Design classes so that in the normal course of use an exception will never be thrown. In the following code example, a **FileStream** class exposes another way of determining if the end of the file has been reached to avoid the exception that will be thrown if the developer reads past the end of the file.

C# :

```
class FileRead
{
    void Open()
    {
      FileStream stream = File.Open("myfile.txt", FileMode.Open);
      byte b;

      // ReadByte returns -1 at end of file.
     while ((b = stream.ReadByte()) != true)
     {
        // Do something.
     }
    }
}
```

Vb.net :

```
Class FileRead
      Sub Open()
    Dim stream As FileStream = File.Open("myfile.txt", FileMode.Open)
          Dim b As Byte
```

```
            ' ReadByte returns -1 at end of file.
           While b = stream.ReadByte() <> true
                ' Do something.
           End While
       End Sub
End Class
```

- Throw the InvalidOperationException exception if a call to a property **set** accessor or method is not appropriate given the object's current state.
- Throw an ArgumentException or create an exception derived from this class if invalid parameters are passed or detected.
- Be aware that the stack trace starts at the point where an exception is thrown, not where it is created with the **new** operator. Consider this when deciding where to throw an exception.
- Use the exception builder methods. It is common for a class to throw the same exception from different places in its implementation. To avoid repetitive code, use helper methods that create the exception using the **new** operator and return it. The following code example shows how to implement a helper method.

C# :

```
class File
{
   string fileName;
   public byte[] Read(int bytes)
   {
      if (!ReadFile(handle, bytes))
           throw NewFileIOException();
   }

   FileException NewFileIOException()
   {
      string description =
         // Build localized string, include fileName.
      return new FileException(description);
   }
}
```

Vb.net :

```
Class File
  Filename As String
  Public function Read(bytes As Integer) As byte()
        If (Not ReadFile(handle, bytes)) Then
           throw NewFileIOException()
        End If
   End function

   Public function NewFileIOException() As FileException

       Dim description As String =
         ‘ Build localized string, include fileName.
      return new FileException(description)
   End function
End Class
```

- Throw exceptions instead of returning an error code or HRESULT.
- Throw the most specific exception possible.

- Create meaningful message text for exceptions, targeted at the developer.
- Set all fields on the exception you use.
- Use Inner exceptions (chained exceptions). However, do not catch and re-throw exceptions unless you are adding additional information and/or changing the type of the exception.
- Do not create methods that throw NullReferenceException or IndexOutOfRangeException.
- Perform argument checking on protected (Family) and internal (Assembly) members. Clearly state in the documentation if the protected method does not do argument checking. Unless otherwise stated, assume that argument checking is performed. There might, however, be performance gains in not performing argument checking.
- Clean up any side effects when throwing an exception. Callers should be able to assume that there are no side effects when an exception is thrown from a function. For example, if a **Hashtable.Insert** method throws an exception, the caller can assume that the specified item was not added to the **Hashtable**.

13 Standard Exception Types

The following table lists the standard exceptions provided by the runtime and the conditions for which you should create a derived class.

Exception type	Base type	Description	Example
Exception	Object	Base class for all exceptions.	None (use a derived class of this exception).
SystemException	Exception	Base class for all runtime-generated errors.	None (use a derived class of this exception).
IndexOutOfRangeException	SystemException	Thrown by the runtime only when an array is indexed improperly.	Indexing an array outside of its valid range: arr[arr.Length+1]
NullReferenceException	SystemException	Thrown by the runtime only when a null object is referenced.	object o = null; o.ToString();
InvalidOperationException	SystemException	Thrown by methods when in an invalid state.	Calling Enumerator.GetNext() after removing an Item from the underlying collection.
ArgumentException	SystemException	Base class for all argument exceptions.	None (use a derived class of this exception).
ArgumentNullException	ArgumentException	Thrown by methods that do not allow an argument to be null.	String s = null; "Calculate".IndexOf (s);
ArgumentOutOfRangeException	ArgumentException	Thrown by methods that verify that arguments are in a given range.	String s = "string"; s.Chars[9];
ExternalException	SystemException	Base class for exceptions that occur or are targeted at environments outside of the runtime.	None (use a derived class of this exception).
COMException	ExternalException	Exception encapsulating COM Hresult information.	Used in COM interop.
SEHException	ExternalException	Exception encapsulating Win32 structured Exception Handling information.	Used in unmanaged code Interop.

Wrapping Exceptions

Errors that occur at the same layer as a component should throw an exception that is meaningful to target users. In the following code example, the error message is targeted at users of the TextReader class, attempting to read from a stream.

C# :

```
public class TextReader
{
  public string ReadLine()
  {
    try
    {
      // Read a line from the stream.
    }
    catch (Exception e)
    {
      throw new IOException ("Could not read from stream", e);
    }
  }
}
```

14 Guidelines for implementing Finalize and Dispose

Class instances often encapsulate control over resources that are not managed by the runtime, such as window handles (HWND), database connections, and so on. Therefore, you should provide both an explicit and an implicit way to free those resources. Provide implicit control by implementing the protected Finalize Method on an object (destructor syntax in C# and the Managed Extensions for C++). The garbage collector calls this method at some point after there are no longer any valid references to the object.

In some cases, you might want to provide programmers using an object with the ability to explicitly release these external resources before the garbage collector frees the object. If an external resource is scarce or expensive, better performance can be achieved if the programmer explicitly releases resources when they are no longer being used. To provide explicit control, implement the Dispose method provided by the IDisposable Interface. The consumer of the object should call this method when it is done using the object. **Dispose** can be called even if other references to the object are alive.

Note that even when you provide explicit control by way of **Dispose**, you should provide implicit cleanup using the **Finalize** method. **Finalize** provides a backup to prevent resources from permanently leaking if the programmer fails to call **Dispose**.

For more information about implementing **Finalize** and **Dispose** to clean up unmanaged resources, see Programming for Garbage Collection. The following code example illustrates the basic design patter for implementing Dispose.

C# :

```
// Design pattern for a base class.
public class Base: IDisposable
{
   //Implement IDisposable.
   public void Dispose()
   {
     Dispose(true);
      GC.SuppressFinalize(this);
   }

   protected virtual void Dispose(bool disposing)
   {
      if (disposing)
      {
         // Free other state (managed objects).
      }
      // Free your own state (unmanaged objects).
      // Set large fields to null.
   }

   // Use C# destructor syntax for finalization code.
   ~Base()
   {
      // Simply call Dispose(false).
      Dispose (false);
   }

   // Design pattern for a derived class.
   public class Derived: Base
   {
      protected override void Dispose(bool disposing)
      {
         if (disposing)
```

```
            {
            // Release managed resources.
         }
         // Release unmanaged resources.
         // Set large fields to null.
         // Call Dispose on your base class.
         base.Dispose(disposing);
      }
      // The derived class does not have a Finalize method
      // or a Dispose method with parameters because it inherits
      // them from the base class.
   }
```

Vb.net :

```
Public Class Base
   Implements IDisposable
   ' Implement IDisposable.
   Public Sub Dispose()
      Dispose(True)
      GC.SuppressFinalize(Me)
   End Sub

   Protected Overloads Overridable Sub Dispose(disposing As Boolean)
      If disposing Then
         ' Free other state (managed objects).
      End If
      ' Free your own state (unmanaged objects).
      ' Set large fields to null.
   End Sub

   Protected Overrides Sub Finalize()
      ' Simply call Dispose(False).
      Dispose (False)
   End Sub
End Class

' Design pattern for a derived class.
Public Class Derived
   Inherits Base

   Protected Overloads Overrides Sub Dispose(disposing As Boolean)
      If disposing Then
         ' Release managed resources.
      End If
      ' Release unmanaged resources.
      ' Set large fields to null.
      ' Call Dispose on your base class.
      Mybase.Dispose(disposing)
   End Sub
   ' The derived class does not have a Finalize method
   ' or a Dispose method with parameters because it inherits
   ' them from the base class.
```

Customizing a Dispose Method Name

Occasionally a domain-specific name is more appropriate than **Dispose**. For example, a file encapsulation might want to use the method name **Close**. In this case, implement **Dispose** privately and create a public **Close** method that calls **Dispose**. The following code example illustrates this pattern. You can replace **Close** with a method name appropriate to your domain.

C# :

```
// Do not make this method virtual.
// A derived class should not be allowed
// to override this method.
public void Close()
{
  // Call the Dispose method with no parameters.
  Dispose();
 }
```

Vb.net :

```
'Do not make this method virtual.
' A derived class should not be allowed
' to override this method.
Public Sub Close()

 ' Call the Dispose method with no parameters.
  Dispose();
End Sub
```

Finalize

The following rules outline the usage guidelines for the **Finalize** method:

- Only implement **Finalize** on objects that require finalization. There are performance costs associated with **Finalize** methods.
- If you require a **Finalize** method, you should consider implementing **IDisposable** to allow users of your class to avoid the cost of invoking the **Finalize** method.
- Do not make the **Finalize** method more visible. It should be **protected**, not **public**.
- An object's **Finalize** method should free any external resources that the object owns. Moreover, a **Finalize** method should release only resources that are held onto by the object. The **Finalize** method should not reference any other objects.
- Do not directly call a **Finalize** method on an object other than the object's base class.
- Call the **base.Finalize** method from an object's **Finalize** method.

Note : The base class's **Finalize** method is called automatically with the C#.

Dispose

The following rules outline the usage guidelines for the **Dispose** method:

- Implement the dispose design pattern on a type that encapsulates resources that explicitly need to be freed. Users can free external resources by calling the public **Dispose** method.
- Implement the dispose design pattern on a base type that commonly has derived types that hold on to resources, even if the base type does not. If the base type has a close method, often this indicates the need to implement Dispose. In such cases, do not implement a Finalize method on the base type. Finalize should be implemented in any derived types that introduce resources that require cleanup.
- Free any disposable resources a type owns in its Dispose method.

- After Dispose has been called on an instance, prevent the Finalize method from running by calling the GC.SuppressFinalize Method. The exception to this rule is the rare situation in which work must be done in Finalize that is not covered by Dispose.
- Call the base class's Dispose method if it implements IDisposable.
- Do not assume that Dispose will be called. Unmanaged resources owned by a type should also be released in a Finalize method in the event that Dispose is not called.
- Throw an ObjectDisposedException when resources are already disposed. If you choose to reallocate resources after an object has been disposed, ensure that you call the GC.ReRegisterForFinalize Method.
- Propagate the calls to Dispose through the hierarchy of base types. The Dispose method should free all resources held by this object and any object owned by this object. For example, you can create an object like a TextReader that holds onto a Stream and an Encoding, both of which are created by the TextReader without the user's knowledge. Furthermore, both the Stream and the Encoding can acquire external resources. When you call the Dispose method on the TextReader, it should in turn call Dispose on the Stream and the Encoding, causing them to release their external resources.
- You should consider not allowing an object to be usable after its Dispose method has been called. Recreating an object that has already been disposed is a difficult pattern to implement.
- Allow a Dispose method to be called more than once without throwing an exception. The method should do nothing after the first call.

15 Vb.net

```
Public Class TextReader
    Public Function ReadLine() As String
     Try
        ' Read a line from the stream.
     Catch e As Exception
          Throw New IOException("Could not read from stream", e)
      End Try
     End Function
     End Class
```

16 Guidelines for COM Interoperating

The common language runtime provides rich support for interoperating with COM components. A COM component can be used from within a managed type and a managed instance can be used by a COM component. This support is the key to moving unmanaged code to managed code one piece at a time; however, it does present some issues for class library designers. In order to fully expose a managed type to COM clients, the type must expose functionality in a way that is supported by COM and abides by the COM versioning contract.

Mark managed class libraries with the ComVisibleAttribute attribute to indicate whether COM clients can use the library directly or whether they must use a wrapper that shapes the functionality so that they can use it.

Types and interfaces that must be used directly by COM clients, such as to host in an unmanaged container, should be marked with the **ComVisible(true)** attribute. The transitive closure of all types referenced by exposed types should be explicitly marked as **ComVisible(true)**; if not, they will be exposed as **IUnknown**.

Note : Members of a type can also be marked as ComVisible(false); this reduces exposure to COM and therefore reduces the restrictions on what a managed type can use.

Types marked with the **ComVisible(true)** attribute cannot expose functionality exclusively in a way that is not usable from COM. Specifically, COM does not support static methods or parameterized constructors. Test the type's functionality from COM clients to ensure correct behavior. Make sure that you understand the registry impact for making all types cocreateable.

Marshal By Reference

Marshal-by-reference objects are Remotable Objects. Object remoting applies to three kinds of types:

- Types whose instances are copied when they are marshaled across an AppDomain boundary (on the same computer or a different computer). These types must be marked with the **Serializable** attribute.
- Types for which the runtime creates a transparent proxy when they are marshaled across an AppDomain boundary (on the same computer or a different computer). These types must ultimately be derived from System.MarshalByRefObject Class.
- Types that are not marshaled across AppDomains at all. This is the default.

Follow these guidelines when using marshal by reference:

- By default, instances should be marshal-by-value objects. This means that their types should be marked as Serializable.
- Component types should be marshal-by-reference objects. This should already be the case for most components, because the common base class, System.Component Class, is a marshal-by-reference class.
- If the type encapsulates an operating system resource, it should be a marshal-by-reference object. If the type implements the IDisposable Interface it will very likely have to be marshaled by reference. System.IO.Stream derives from MarshalByRefObject. Most streams, such as FileStreams and NetworkStreams, encapsulate external resources, so they should be marshal-by-reference objects.
- Instances that simply hold state should be marshal-by-value objects (such as a DataSet).
- Special types that cannot be called across an AppDomain (such as a holder of static utility methods) should not be marked as Serializable