Migrating from COM and VB6 to VB.NET

Agenda

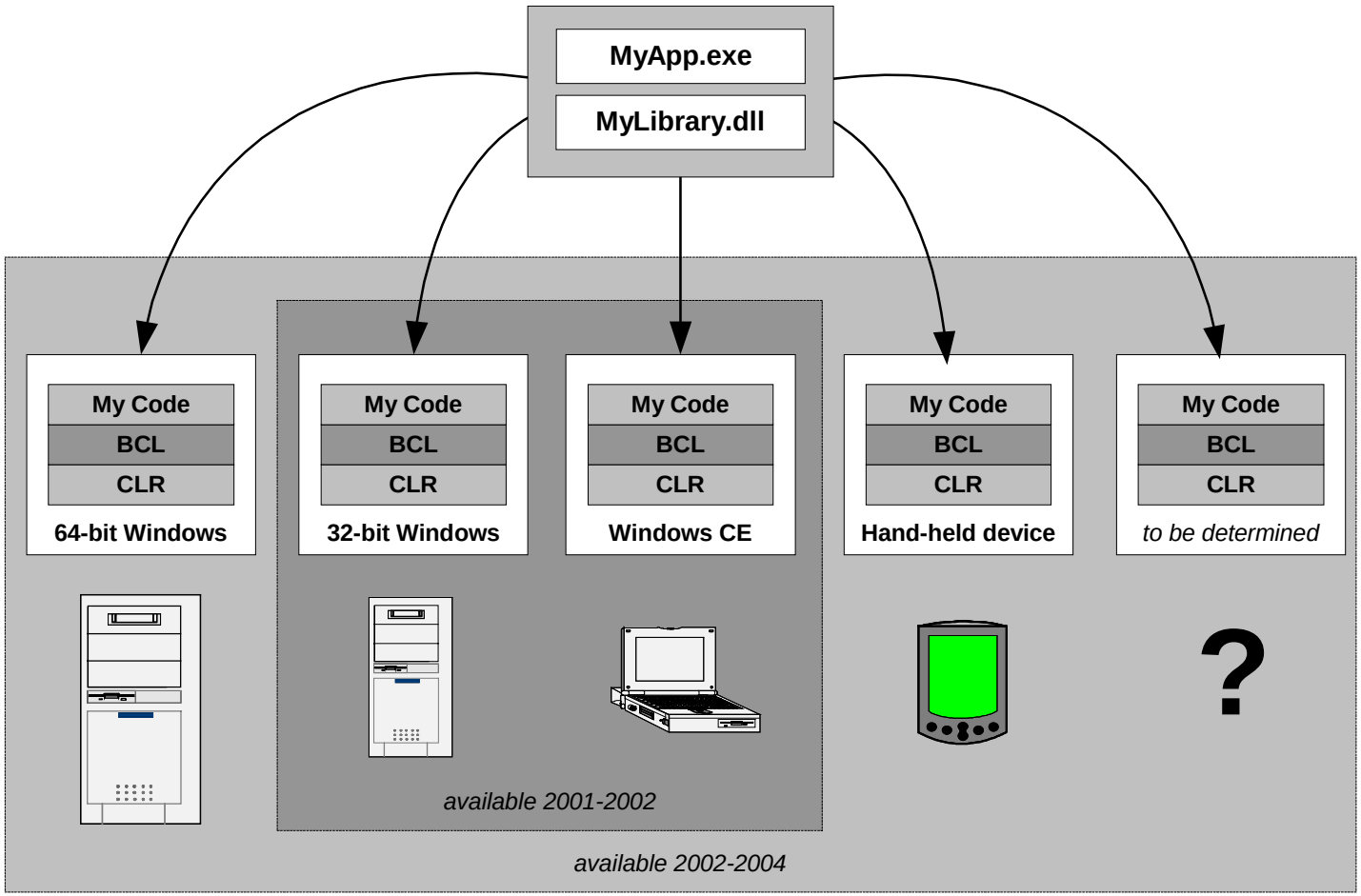
- Intro to the CLR and VB.NET
- Adjusting to a new type system
- New OOP features in class design
- Interfaces and inheritance
- Structured exception handling
- Objects and values
- Delegates and events

Intro to the CLR and VB.NET

The .NET framework as a development platform

- The .NET framework is a platform based on the premise of building applications by leveraging reusable components
  - .NET is based on the Common Language Runtime (CLR)
  - The CLR is an execution engine (i.e. virtual machine)
  - The CLR ships with the Base Class Libraries (BCL)
  - .NET eliminates dependencies on any specific hardware platform
  - .NET eliminates dependencies on any specific operating system
  - .NET represents new era in component software evolution

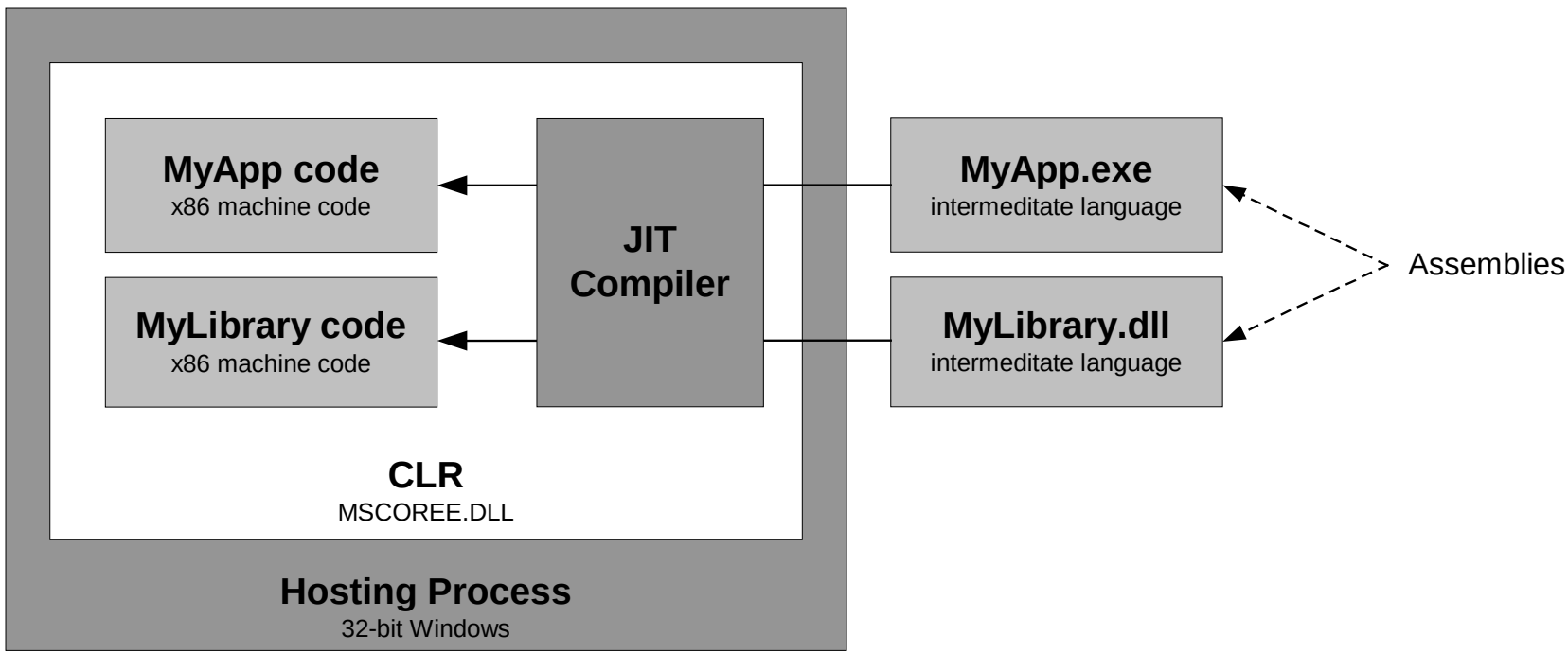
Code which targets the CLR can be run on several platforms



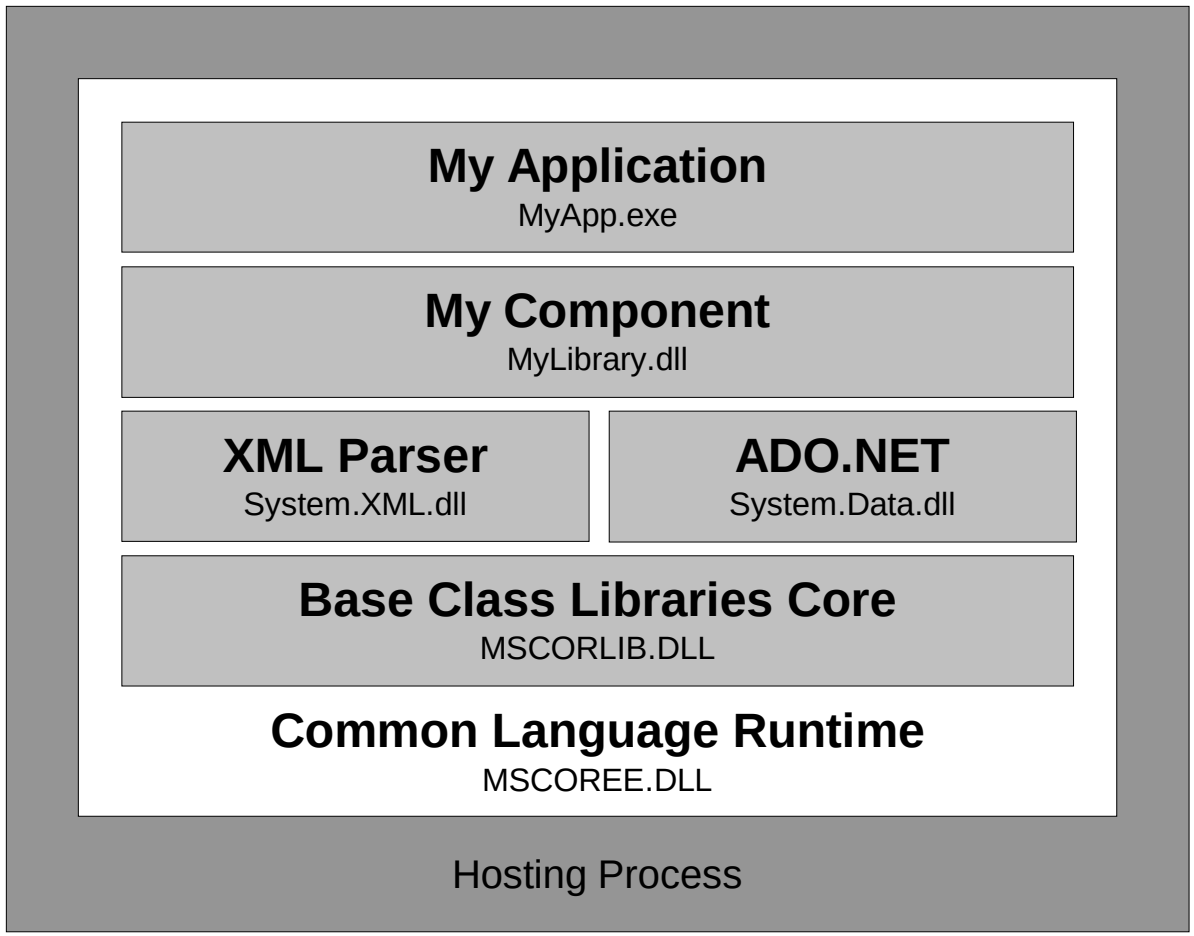
Common Language Runtime (CLR)

- In the CLR, code is always loaded and run under strict control of the system
  - Code written to target the CLR is known as managed code
  - Managed code must be written in a language designed for the CLR
  - Managed code is distributed and reused in terms of assemblies
  - Managed code is compiled into intermediate language (IL)
  - Code in IL form undergoes JIT compilation at load time

Code in the form of IL must undergo JIT compilation



CLR Architecture



A sampling of the Base Class Libraries (BCL)

Namespace	Assembly	Purpose
System	mscorlib.dll	Core type system
System.Reflection	mscorlib.dll	Reflection services
System.Security	mscorlib.dll	Access control
System.Text	mscorlib.dll	Text encoding/munging
System.Collections	mscorlib.dll	Collection types
System.IO	mscorlib.dll	Binary and text I/O
System.Threading	mscorlib.dll	Threading/locking
System.Runtime.Serialization	mscorlib.dll	Object persistence
System.Runtime.Remoting	mscorlib.dll	SOAP & Proxy support
System.Runtime.InteropServices	mscorlib.dll	Native code support
System.Data	System.Data.dll	Access to DBMS
System.Xml	System.Xml.dll	XML, XPath and XSLT support
System.Web	System.Web.dll	Server-side HTTP support
System.Windows.Forms	System.Windows.Forms.dll	Forms-based UI support
System.Diagnostics	System.dll	Assertion/tracing

A Universal Type System

- The CLR was designed to facilitate interoperability across languages
  - The CLR defines a universal type system
  - The CLR defines a set of modern, object-oriented programming features
  - Manages types are often defined inside namespaces
  - The CLS defines the minimum required set of types and features
  - The Base Class Libraries are CLS-compliant

Component Metadata and Reflection

- The .NET programming model is highly dependant upon component metadata and reflection
  - Every assembly must carry an extensive set of component metadata
  - ILDASM.EXE is a useful utility for examining an assembly's metadata
  - Metadata fully describes each assembly including its types and its dependencies
  - The CLR and other .NET services use reflection to read component metadata at runtime
  - The CLR's metadata format is extensible through custom attributes

VB.NET code with custom attributes applied

```
Imports System
Imports System.Reflection
Imports System.EnterpriseServices

<Assembly: AssemblyCulture("en-US")>
<Assembly: AssemblyKeyFile("MyKey.snk")>
<Assembly: AssemblyVersion("2.0.12.0")>

<Serializable> Class Class1
  Private Field1 As Integer
  Private <NonSerialized> Field2 As Integer
  Private Field3 As Integer
End Class

<Transaction(TransactionOption.Required)> _
Class Class2 : Inherits ServicedComponent
  Sub <AutoComplete> Method1()
     '*** implementation
  End Sub
End Class
```

Compiling and testing code

- As a .NET developer, you must be able to compile and test your code
  - VBC.EXE is the command-line compiler for VB.NET
  - CSC.EXE is the command-line compiler for C#
  - Compilers require references when using types from other assemblies
  - NMAKE.EXE is a essential utility for .NET developers
  - Visual Studio .NET is a valuable tool for .NET developers

A VB.NET application, a component library and a MAKEFILE

```
'*** MyApp.vb

Imports System
Imports AcmeCorp

Module MyApp
  Sub Main()
    Dim obj As New Class1
    Console.WriteLine(obj.Foo())
  End Sub
End Module
```

```
'*** MyLibrary.vb

Namespace AcmeCorp
  Public Class Class1
    Function Foo() As String
      Return "Hi from a VB.NET DLL"
    End Function
  End Class
End Namespace
```

```
All : MyApp.exe MyLibrary.dll

MyApp.exe : MyApp.vb MyLibrary.dll
  vbc /t:exe /r:MyLibrary.dll MyApp.vb

MyLibrary.dll : MyLibrary.vb
  vbc /t:library MyLibrary.vb
```

Migrating from VB6 to VB.NET

- In many ways, VB.NET can be seen as an entirely new language
  - Lots of programmatic conveniences have been added
  - Inconsistencies and idiosyncrasies have been removed
  - Many new object-oriented programming features have been added
  - Much higher levels of type safety are enforced at compile time
  - VB.NET projects should favor managed libraries over unmanaged libraries
  - **`Microsoft.VisualBasic.dll`** is a .NET library for VB programmers

VB.NET includes many new conveniences

```
Function Foo() As String

  '*** declare and initialize variables on the same line
  Dim x As Integer = 10
  Dim obj1 As Class1 = New Class1
  Dim obj2 As New Class1

  '*** use new C-style syntax
  x += 5 ' this is the same as x = x + 5

  '*** Return statement can be used in functions
  Return "This is the method's return value"

End Function
```

VB.NET removes inconsistencies that were allowed in VB6

```
Sub Foo()
  ' Set no longer required/allowed on assignment statements
  Dim obj As Class1
  obj = New Class1       ' compiles
  Set obj = New Class1   ' doesn't compile
  ' Argument lists in all calls must be enclosed in parentheses
  Bar(3)  ' compiles
  Bar 4   ' doesn't compile
End Sub

Sub Bar(ByVal i As Integer)
  ' implementation
End Sub
```

ASP.NET: a platform on a platform

- ASP.NET is a managed framework that facilitates building server-side applications based on HTTP, HTML, XML and SOAP
  - ASP.NET replaces original ASP framework
  - ASP.NET adds integrated compilation support to the CLR
  - ASP.NET lets you work with VB.NET and C# instead of scripting languages
  - ASP.NET facilitates creating HTML-based applications with Web forms, server-side controls and data binding
  - ASP.NET facilitates creating HTTP handlers, filters and Web services

Hello, World using VB.NET in an ASP.NET page

```
<%-- MyPage1.aspx --%>
<%@ page language="VB" %>

<html>
<body>

  <h3>Test1 response page</h3>
  This page contains boring static content and
  <%
    Dim s As String
    s =  "<i>"
    s += "this page also contains "
    s += "exciting dynamic content"
    s += "</i>"
    Response.Write(s)
  %>

</body>
</html>
```

A component library that's accessible to ASP.NET pages

```
'*** WebUtilities.vb
'*** WebUtilities.dll placed in \bin directory

Imports System
Imports System.Collections

Namespace AcmeCorp.WebUtilities

  Public Class MyDataClass
    Function GetCustomerList() As IList
      Dim list As IList = New ArrayList()
      list.Add("John")
      list.Add("Paul")
      list.Add("George")
      list.Add("Pete")
      Return list
    End Function
  End Class

End Namespace
```

An ASP.NET page that uses component libraries

```
<%-- MyPage2.aspx --%>
<%@ Page language="VB" %>

<%@ Import namespace="System" %>
<%@ Import namespace="System.Collections" %>
<%@ Import namespace="AcmeCorp.WebUtilities" %>

<html>
<body>
  <h3>Customer list</h3>
  <%
    '*** ASP.NET automatically references DLLs in \bin directory
    Dim MyDataObject As New MyDataClass
    Dim Customer As String
    For Each Customer in MyDataObject.GetCustomerList()
      Response.Write("<li>" & s & "</li>")
    Next
  %>
</body>
</html>
```

ASP.NET code can be partitioned using code-behind features

```
<%-- MyPage3.aspx --%>
<%@ page language="VB" src=" MyPage3.aspx.vb" %>
<%@ Import Namespace="AcmeCorp" %>
<html>
<body>
  <h3>Splitting code between a .aspx file and a .vb file</h3>
  This text was generated by MyPage3.aspx<br>
  <%
    Dim obj As New ContentGenerator
    Response.Write(obj.GetContent())
  %>
</body>
</html>
```

```
'*** source file: MyPage3.aspx.vb
Imports System

Namespace AcmeCorp
  Public Class ContentGenerator
    Function GetContent() As String
      Return "This text was generated by MyUtilityCode.vb"
    End Function
  End Class
End Namespace
```

An example of a Web form and server-side controls

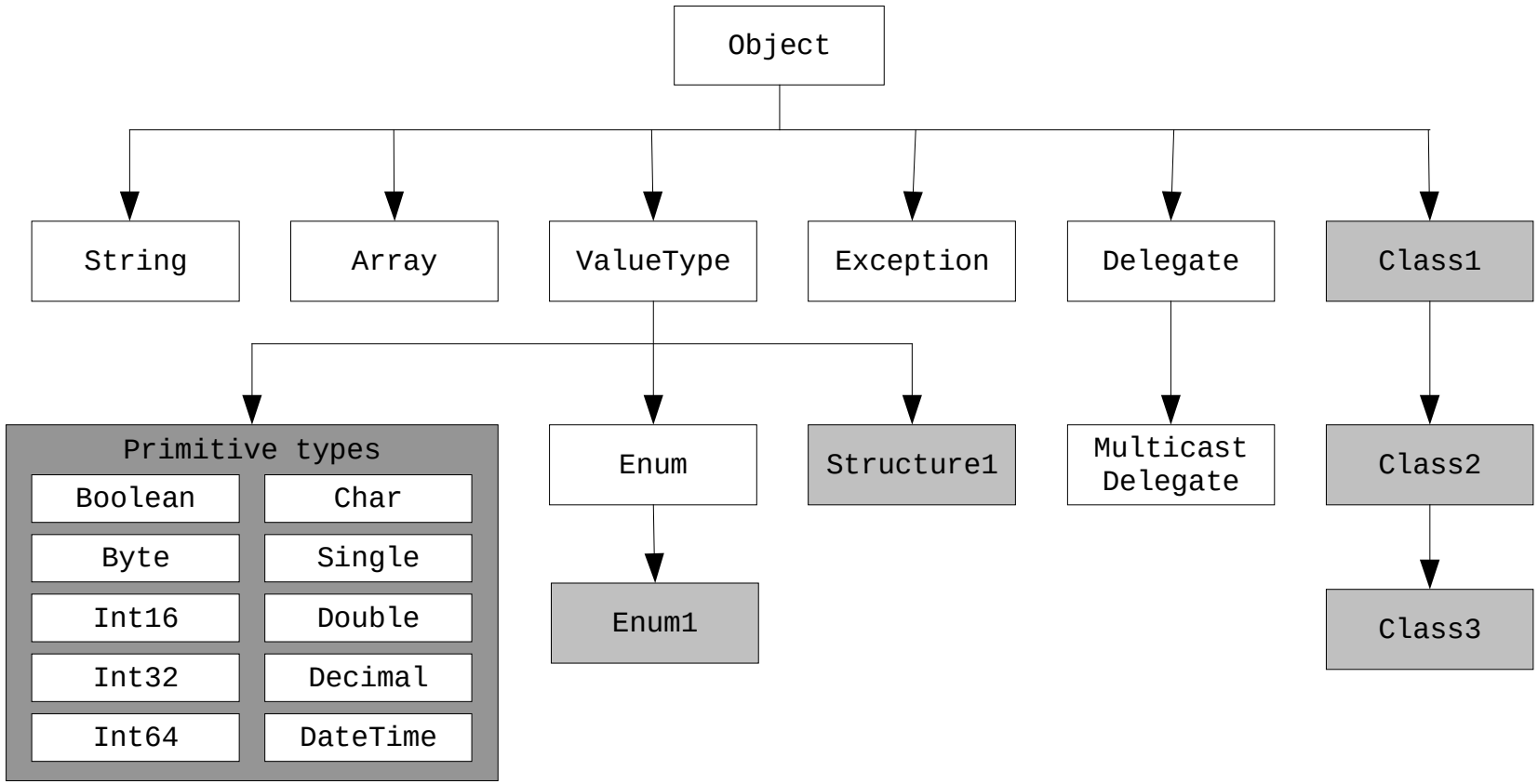
```
<%-- MyPage4.aspx --%>
<%@ page language="VB" %>

<html>
<body>
  <form runat="server">
    <p><ASP:Textbox id="txtValue1" runat="server" /></p>
    <p><ASP:Textbox id="txtValue2" runat="server" /></p>
    <p><ASP:Button  text="Add Numbers"
            runat="server"  OnClick="cmdAdd_OnClick" /> </p>
    <p><ASP:Textbox id="txtValue3" runat="server" /></p>
  </form>
</body>
</html>

<script runat="server">
  Sub cmdAdd_OnClick(sender As Object , e As System.EventArgs)
    Dim x, y As Integer
    x = CInt(txtValue1.Text)
    y = CInt(txtValue2.Text)
    txtValue3.Text = CStr(x + y)
  End Sub
</script>
```

Adjusting to a new type system

Managed types

- The programming model of the CLR is based on managed types
  - Managed types exist within the scope of a singly-root inheritance hierarchy
  - All VB.NET code lives inside the scope of managed types
  - VB.NET types include classes, modules, structures, enums and interfaces
  - Types must be public to be accessible from outside their assembly
  - Types can contain members
  - Types can contain nested types

Core CLR types in the System namespace



Using some of the built-in managed types

```
'*** these three variable declarations
Dim x1 As System.Int32
Dim y1 As System.String
Dim z1 As System.Object

'*** are equivalent to these
Dim x2 As Integer
Dim y2 As String
Dim z2 As Object
```

Namespaces and nested classes help to avoid naming conflicts

```
Namespace DevelopMentor

  Public Class CustomerList
    Class Enumerator
      '*** definition
    End Class
  End Class

  Public Class EmployeeList
    Class Enumerator
      '*** definition
    End Class
  End Class

End Namespace
```

Compile-time type checking

- The VB.NET compiler enforces higher levels of type safety than VB6.
  - Every variable, parameter, field and property is based on a type
  - VB.NET compiler can (optionally) perform extensive type-safety checks
  - **`Option Explicit`** requires all variables to be explicitly defined
  - **`Option Strict`** requires everything to be defined with an explicit type
  - Disabling **`Option Strict`** allows late binding through **`Object`** type

Strongly-typed binding versus late binding

```
Option Strict Off

Module MyApp
  Sub Main

    '*** programming against a strongly-typed variable
    Dim ref1 As Class1 = New Class1
    ref1.Foo()
    ref1.Bar() '*** compile-time error (method doesn't exist)

    '*** programming that uses late binding
    Dim ref2 As Object = New Class1
    ref2.Foo()
    ref2.Bar() '*** run-time error (method doesn't exist)

  End Sub
End Module

Public Class Class1
  Sub Foo
    '*** implementation
  End Sub
End Class
```

Primitive types

- The CLR's type system includes several familiar primitive types
  - The CLR defines a predictable set of primitive types
  - All primitive types are scoped in the **`System`** namespace
  - VB.NET auto-initializes primitive types to a well-known default value
  - VB.NET supports all primitive types that are CLS-compliant
  - VB.NET does not supports some non-CLS-compliant primitive types

CLS-compliant primitive types

VB keyword	CLR Type	Default value	Description
Boolean	Boolean	False	32-bit true/false value
Byte	Byte	0	8-bit unsigned integer value
Short	Int16	0	16-bit signed integer value
Integer	Int32	0	32-bit signed integer value
Long	Int64	0	64-bit signed integer value
Char	Char	ChrW(0)	16-bit UNICODE character value (0-65535)
Single	Single	0.0	32-bit floating-point number
Double	Double	0.0	64-bit floating-point number
Decimal	Decimal	0.0	96-bit number with 28 points of precision
Date	DateTime	#01/01/0001#	64-bit long integer in IEEE date format
N/A	TimeSpan	00:00:00	64-bit long integer in IEEE time span format
N/A	Guid	(all zeros)	128-bit integer vlaue (often used as unique ID)

Strings

- String values are managed with instances of the **`System.String`** class
  - The **`String`** keyword in VB.NET maps to the **`System.String`** class
  - The VB.NET compiler treats the **`String`** class differently than other types
  - **`System.String`** provides string-related utility functions
  - Instances of types can be converted to strings using the **`ToString`** method
  - **`StringBuilder`** class provides a more efficient buffering technique for parsing together large strings

Creating instances of the System.String type

```
'*** create a new string object
Dim s1 As String = "Hello"

'*** call String.Concat explicitly
Dim s2 As String = String.Concat(s1, ", World")

'*** call String.Concat implicitly
Dim s3 As String = s1 & ", World"

'*** string concatentation performed at compile time
Dim s4 As String = "Hello" & ", World"

'*** computing a string value at runtime
Dim s5 As New String(CChar("A"), 10)
```

Comparing strings

```
Dim s1 As String = "Bob"
Dim s2 As String = "BOB"

'*** perform case-insenstive comparison
Dim b1 As Boolean = ( String.Compare(s1, s2, True) = 0 )

'*** perform case-senstive comparison
Dim b2 As Boolean = ( String.Compare(s1, s2, False) = 0 )
Dim b3 As Boolean = ( String.CompareOrdinal(s1, s2) = 0 )
Dim b4 As Boolean = (s1 = s2)

'*** perform comparison with = operator
'*** Option Compare { Binary | Text } determines case senstivity
If (s1 = s2) Then
  ' statement block
End If
```

Using functions from the `System.String` class

```
Dim s1 As String = "Hello world"

' convert to an upper-case string
Dim s2 As String = s1.ToUpper

'*** extract the string "lo wo"
Dim s3 As String = s1.Substring(3, 5)

'*** insert one string value inside another
Dim s4 As String = s1.Insert(6, "brave new ")

'*** extract a character from a string
Dim c As Char = s1.Chars(3)

'*** determine if a string starts with a certain pattern
Dim b As Boolean = s1.StartsWith("H")
```

Using ToString

```
Dim d As Double = 3.141592
Dim s1 As String = d.ToString()

Dim s2 As String = 714.ToString()

Dim d1 As Date = Date.Now
Dim s3 As String = d1.ToString("MMMM d, yyyy")

Dim MyAccountBalance As Decimal = 324.31D
Dim s4 As String = MyAccountBalance.ToString("($#,###)")
```

Using the class `System.Text.StringBuilder`

```
Imports System
Imports System.Text
Imports Microsoft.VisualBasic

Module MyApp
  Const LB As String = ControlChars.CrLf
  Sub Main()
    '*** employ StringBuilder to parse together large string
    Dim sb As New StringBuilder(512)
    sb.Append("How have you been doing?")
    sb.Append(LB)
    sb.Append("I haven't seen you in a while.")
    sb.Insert(0, "Hi John," & LB)
    '*** extract string from StringBuilder object
    Dim s As String = sb.ToString()
    Console.WriteLine(s)
  End Sub
End Module
```

System.Object

- **System.Object** is the super type of the CLR's type system
  - The **Object** keyword in VB.NET maps to **System.Object**
  - **Object** is the type at the root of the type hierarchy
  - Any other type can be implicitly converted to the **Object** type
  - **Object** is somewhat similar to the VB6 **Variant** type

Instance of all types are assignable to variables of type `Object`

```
'*** initialize instances using various types
Dim a As Integer = 10
Dim b As String = "Hey there, Bob"
Dim c As Class1 = New Class1

'*** assign instances to System.Object variables
Dim x As Object = a
Dim y As Object = b
Dim z As Object = c
```

Parameters of type `Object` are flexible yet not type-safe

```
Sub Main()

  Dim x As Class1
  ProcessCheeseburger(x) '*** x Is Nothing

  Dim y As New Class1
  ProcessCheeseburger(y) '*** y references Class1 instance

  Dim z As String = "Hi there"
  ProcessCheeseburger(z) '*** z holds a string value

End Sub

Sub ProcessCheeseburger(param1 As Object)
  '*** test to see if param1 Is Nothing
  If param1 Is Nothing Then
    Console.WriteLine("param1 is Nothing")
  '*** test to see if param1 references a Class1 instance
  ElseIf TypeOf param1 Is Class1 Then
    Console.WriteLine("param1 references Class1 instance")
  '*** handle all other cases
  Else
    Console.WriteLine("param1 references instance of some other type")
  End If
End Sub
```

Casting between types

- Knowing when and how to explicitly cast between types is essential
  - VB.NET allows implicit casting between convertible types
  - **`Option Strict`** doesn't permit implicit casting non-convertible types
  - **`Option Strict`** doesn't permit implicit conversion when there's potential loss of data/precision
  - Explicit casting is performed using **`CType`**, **`CInt`**, **`CStr`**, etc.

Implicit narrowing conversions are prohibited

```
Dim x As Double = 20.1

'*** implicit narrowing conversions are not allowed
Dim y As Integer = x ' doesn't compile

'*** An explicit cast makes the VB.NET compiler happy
'*** runtime exception is thrown if cast attempt fails
Dim Z As Integer = CInt(x)
```

Explicit casts are required when converting from System.Object

```
'*** initialize instances using various types
Dim a As Integer = 10
Dim b As String = "Hey there, Bob"
Dim c As Class1 = New Class1

'*** assign instances to System.Object variables
Dim x As Object = a
Dim y As Object = b
Dim z As Object = c

'*** cast from System.Object back to specific types
Dim d As Integer = CInt(x)
Dim e As String = CStr(Y)
Dim f As Class1 = CType(z, Class1)
```

Logical comparisons versus bitwise operations

- Know the difference between logical comparisons and bitwise operations
  - Under beta 1, logical comparisons and bitwise operations are much different than VB6
  - With the beta 2 release, VB.NET restores VB6-like functionality
  - **`Not`**, **`And`**, **`Or`** and **`XOr`** work the same way as in VB6
  - **`BitNot`**, **`BitAnd`**, **`BitOr`** and **`BitXOr`** have been removed from VB.NET
  - **`AndAlso`** and **`OrElse`** are logical comparison operators with short-circuiting

Performing bitwise operations

```
Dim x As Integer = 3
Dim y As Integer = 6

'*** use bitwise Or to find union of bits
Dim union = x Or y '*** union = 7

'*** use bitwise And to find intersection of bits
Dim intersection = x And y '*** intersection = 2
```

`AndAlso` and `OrElse` are new logical comparisons operators

```
'*** Function2 doesn't execute if (Function1() = False)
If Function1() AndAlso Function2() Then
  ' statement block
End If

'*** Function2 doesn't execute if (Function1() = True)
If Function1() OrElse Function2() Then
  ' statement block
End If
```

New OOP features in class design

Designing classes

- Classes are the fundamental unit of design in object-oriented programming
  - Classes make it possible to transform abstractions into implementations
  - Classes are often used as templates for creating objects
  - Class can contain shared members and/or instance members
  - It's often desirable to encapsulate the implementation details of a class
  - Class members can be defined as public, friend, private or protected

Classes provide the basic building blocks for writing software

```
Public Class Customer
  '*** fields
  Public ID As Integer
  Public Name As String
  '*** methods
  Public Function GetInfo() As String
    Dim LineBreak, ReturnValue As String
    LineBreak = System.Convert.ToChar(10)
    ReturnValue = "ID: " & ID & LineBreak
    ReturnValue &= "Name: " & Name
    Return ReturnValue
  End Function
End Class
```

Possible levels of member accessibility

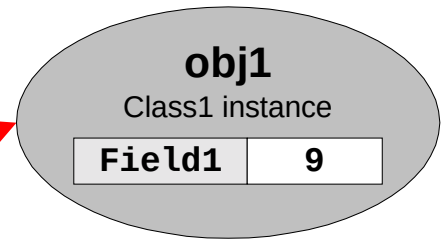
```
Public Class Class1
  '*** accessible to everyone
  Public Field1 As Integer
  '*** accessible from within current assembly
  Friend Field2 As Integer
  '*** accessible from this class only
  Private Field3 As Integer
  '*** accessible from this class and from child classes
  Protected Field4 As Integer
  '*** union of Protected and Friend accessibility
  Protected Friend Field5 As Integer
End Class
```

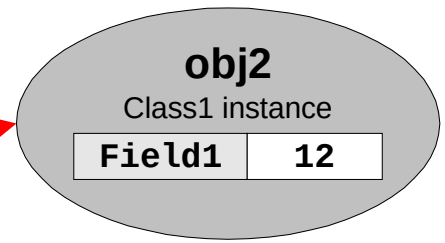
Shared members versus instance members

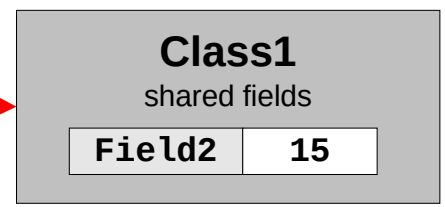
- Shared members are defined at class scope while instance members are defined at object scope
  - Instance members are only accessible through objects
  - Shared members are directly accessible through a class definition
  - Shared members must be defined using the **`Shared`** keyword
  - Instance members can access instance members and shared members
  - Shared members can only access other shared members
  - Memory for shared fields is allocated on a per-class basis

Memory is allocated differently for instance and shared fields

```
Public Class Class1
  '*** instance field
  Public Field1 As Integer
  '*** shared field
  Public Shared Field2 As Integer
End Class

Module MyApp
  Sub Main()

    Dim obj1 As New Class1()
    obj1.Field1 = 9

    Dim obj2 As New Class1()
    obj2.Field1 = 12

    Class1.Field2 = 15

  End Sub
End Module
```







Accessing shared methods from the `Math` and `Console` classes

```
Imports System
Module MyApp
  Sub Main()
    Dim result1, result2, result3 As Double
    result1 = Math.Sqrt(225)
    result2 = Math.Pow(15, 2)
    result3 = Math.Log(225)
    Console.WriteLine("r1={0}, r2={1}, r3={2}", _
                      result1, result2, result3 )
  End Sub
End Module
```

A class can contain instance members and shared members

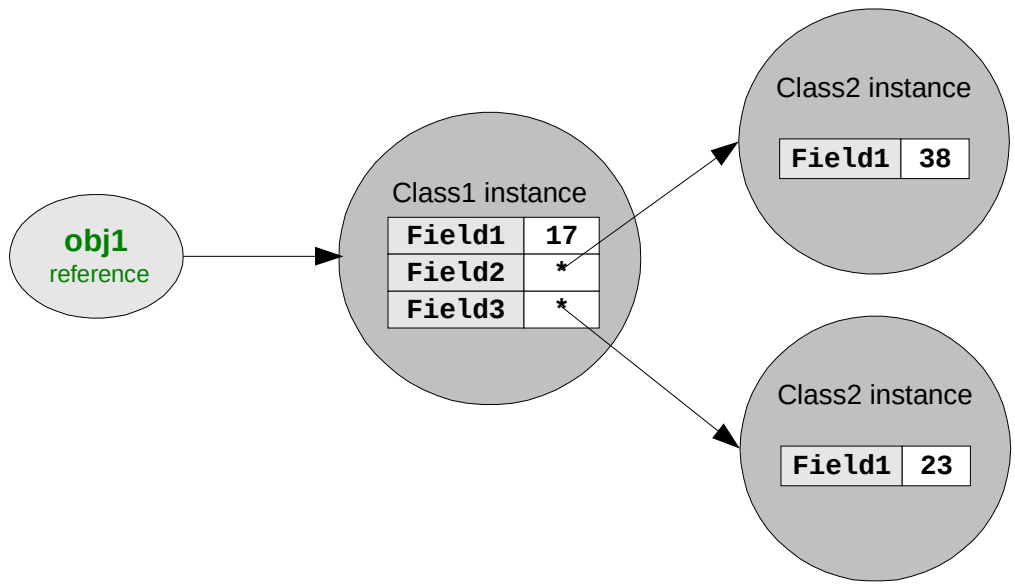
```
Class Class1

  '*** instance members
  Private objectID As Integer
  Function GetObjectInfo() As String
    Return "Object #" & CStr(objectID)
  End Function

  '*** shared members
  Private Shared objectCount As Integer
  Shared Function GetNextObject() As Class1
    objectCount += 1
    Dim temp As New Class1()
    temp.objectID = objectCount
    Return temp
  End Function

End Class
```

Fields

- Fields are named, typed units of storage
  - Each field must be either a shared member or an instance member
  - Fields can be explicitly initialized with in-line syntax
  - Object graphs are created when fields reference other objects
  - **`Const`** fields must be initialized using compile-time constant expressions
  - **`Const`** fields are implicitly **`Shared`**
  - **`Readonly`** fields must be initialized during construction

An object graph is a set of object which reference one another

```
Public Class Class1
  Public Field1 As Integer
  Public Field2 As Class2
  Public Field3 As Class2
End Class

Public Class Class2
  Public Field1 As Integer
End Class

Module MyApp
  Sub Main()
    Dim obj1 As New Class1
    obj1.Field1 = 17
    obj1.Field2 = New Class2
    obj1.Field2.Field1 = 38
    obj1.Field3 = New Class2
    obj1.Field3.Field1 = 23
  End Sub
End Module
```



Methods

- Methods are executable units of code that represent callable operations
  - Methods can be defined with typed parameters and/or a return value
  - Method parameters are defined as either **`ByVal`** or **`ByRef`**
  - The default convention for parameter passing is **`ByVal`**
  - Methods can be defined with optional parameters
  - Methods can be defined with parameter arrays

Methods can take several different forms of parameters

```
Class Class1

  Function Method1(ByVal i As Integer) As String
    ' implementation
  End Function

  Sub Method2(ByVal s As String, ByRef x As Double)
    ' implementation
  End Sub

  Sub Method3(Optional ByVal i As Integer = 33)
    ' implementation
  End Sub

  Sub Method4(ByVal ParamArray data() As String)
    ' implementation
  End Sub

End Class
```

Properties

- Properties are special methods that simulate public fields
  - Each property has a name and a type
  - Each property must implement a **`Get`** block and/or a **`Set`** block
  - Properties can be defined as **`ReadOnly`** or **`WriteOnly`**
  - Properties can take parameters
  - Parameterized properties can be defined with the **`Default`** keyword

A VB.NET property definition

```
Public Class Customer

  '*** private field
  Private m_Name As String

  '*** property provides controlled access to private field
  Public Property Name As String

    Set '*** perform validation here if necessary
      m_Name = Value
    End Set

    Get '*** perform calculations here if necessary
      Return m_Name
    End Get

  End Property

End Class
```

```
'*** client-side code
Dim s As String
Dim obj As New Customer
obj.Name = "Bob" '*** triggers Set block
s = obj.Name     '*** triggers Get block
```

Declaring ReadOnly and WriteOnly properties

```
'*** private fields are inaccessible to client
Private m_FirstName As String
Private m_LastName As String
Private m_Password As String

'*** ReadOnly property has Get but no Set
Public ReadOnly Property FullName as String
  Get
      Return m_FirstName & " " & m_LastName
  End Get
End Property

'*** WriteOnly property has Set but no Get
Public WriteOnly Property Password as String
  Set
    m_Password = Value
  End Set
End Property
```

Properties can be defined with parameters

```
Public Class Class1
  Default ReadOnly Property Item(ByVal Index As Integer) As String
    Get
      Select Case Index
        Case 0
          Return "Larry"
        Case 1
          Return "Mo"
        Case 2
          Return "Curly"
        Case Else
          Return "<unknown>"
      End Select
    End Get
  End Property
End Class
```

```
'*** client-side code
Dim Name As String
Dim obj As New Class1
Name = obj.Item(2) '*** access parameterized property by name
Name = obj(2)      '*** access parameterized property as default
```

Overloading methods and properties

- Overloading occurs when two or more class members have the same name
  - You can overload a set of methods or a set of properties
  - You cannot overload a method and a property to the same name
  - All members that share the same name must have different parameter lists
  - Parameter lists must differ in size or in sequence of parameter types
  - You cannot overload based on return type or parameter name
  - **`Overloads`** is an optional keyword to document member overloading

Method overloading

```
Class CustomerManager

  Function GetCustomer(ID As Integer) As Customer
    ' implementation
  End Function

  Function GetCustomer(Name As String) As Customer
    ' implementation
  End Function

End Class
```

```
‘*** client-side code
Dim c1, c2 As Customer
Dim mgr As New CustomerManager
c1 = mgr.GetCustomer(23)    '**** calls GetCustomer(Integer)
c2 = mgr.GetCustomer("Bob") '**** calls GetCustomer(String)
```

Method overloading can simulate optional parameters

```
Class Class1

  Shared Sub Foo()
    Me.Foo(100) ' forward call passing default value
  End Sub

  Shared Sub Foo(i As Integer)
    ' implementation
  End Sub

End Class
```

```
'*** client-side code
Class1.Foo()      '*** calls Foo(Integer) with parameter value of 100
Class1.Foo(100)   '*** calls Foo(Integer) with parameter value of 100
```

Constructors

- Constructors are special methods designed to initialize fields
  - The CLR executes an instance constructor whenever **`New`** is called on a class
  - Every creatable class must have at least one accessible instance constructor
  - Instance constructors are defined using sub procedures named **`New`**
  - Instance constructors can be parameterized and overloaded for flexibility
  - A non-parameterized instance constructor is called the "default" constructor
  - At times, the VB.NET compiler automatically adds a default constructor
  - A class can contain a shared constructor that executes once per class

Designing class with instance constructors

```
Public Class Class1
  Private x As Integer
  '*** parameterized constructor
  Public Sub New(i As Integer)
    x = i
  End Sub
End Class
```

```
Public Class Class2
  Private x As Integer
  '** default constructor
  Public Sub New()
    x = 99
  End Sub
End Class
```

```
'*** client-side code
Dim a As Class1 = New Class1(14)
Dim b As New Class1(27)
Dim c As Class2 = New Class2
Dim d As Class2 = New Class2()
Dim e As New Class2
Dim f As New Class2()
```

Overloading constructors

```
Public Class Class1
  Private x As Integer
  '*** default constructor
  Public Sub New()
    '*** forward call to parameterized constructor
    MyClass.New(99)
  End Sub
  '*** parameterized constructor
  Public Sub New(i As Integer)
    x = i
  End Sub
End Class
```

Shared constructors

```
Public Class Class1
  Shared Private LastTimeIGotLoaded As Date
  '*** shared constructor
  Shared Sub New()
    LastTimeIGotLoaded = Date.Now
  End Sub
End Class
```

Interfaces and inheritance

What is an interface?

- An interface is an abstract type that defines a contract of behavior
  - Client-side code can be written against interfaces instead of classes
  - Concrete classes required to provide implementation for interface
  - Multiple classes can implement the same interface
  - Interfaces are the key to achieving polymorphism
  - Client-side code can easily switch between compatible implementations
  - Interfaces decouple client-side code from specific class implementations

An interface defines a contract of behavior without implementation

```
Public Interface ICustomerManager
  Function AddCustomer(Name As String) As Integer
  Function GetCustomerName(ByVal ID As Integer) As String
  Function GetCustomerNames() As String()
End Interface

'*** uses hashtable to store customer data
Public Class HashtableCustomerManager
             Implements ICustomerManager
  '*** implementation omitted for clarity
End Class

'*** uses SQL Server to store customer data
Public Class SqlServerCustomerManager
             Implements ICustomerManager
  '*** implementation omitted for clarity
End Class
```

```
'*** client-side code using ICustomerManager interface

'*** method can be called with any ICustomerManager-compatible object
Sub MyUtilityMethod(mgr As ICustomerManager)

  mgr.AddCustomer("Bob")
  mgr.AddCustomer("Shannon")
  mgr.AddCustomer("Jose")

  Dim CustomerList As String() = mgr.GetCustomerNames()
  Dim Customer As String
  For Each Customer In CustomerList
    Console.WriteLine(Customer)
  Next

End Sub

Sub Main
  Dim obj1, obj2 As ICustomerManager
  obj1 = New HashtableCustomerManager
  obj2 = New SqlServerCustomerManager
  MyUtilityMethod(obj1)
  MyUtilityMethod(obj2)
End Sub
```

Interface syntax

- VB.NET provides a special syntax for defining and implementing interfaces
  - Interfaces are defined with the **`Interface`** construct
  - Interfaces usually contain method and/or property signatures
  - All interface members are implicitly public
  - A class declares support for an interface with the **`Implements`** keyword
  - An object implements the same interfaces as the class used to create it
  - A class definition can declare support for multiple interfaces
  - Class definition can hide or rename interface members if necessary

Using the Implements keyword

```
Interface Interface1
  '*** contains no members
End Interface

Class Class1
      Implements Interface1
  '*** implementation omitted
End Class

Class Class2 : Implements Interface1
  '*** implementation omitted
End Class
```

The Programmer class implements the IPerson interface

```
Interface IPerson
  Property Name() As String
  Sub Speak()
End Interface
```

```
Public Class Programmer : Implements IPerson
  '*** private implementation details
  Private m_Name As String

  '*** entry point to implementation for IPerson.Name
  Property Name() As String Implements IPerson.Name
    Set
      m_Name = value
    End Set
    Get
      Return m_Name
    End Get
  End Property

  '*** entry point to implementation for IPerson.Speak
  Sub Speak() Implements IPerson.Speak
    Console.WriteLine("Hi. I write code. My name is " & m_Name)
  End Sub

End Class
```

`Implements` clauses map interface members to implementations

```
Interface Interface1
  Sub Foo()
End Interface

Interface Interface2
  Sub Bar()
End Interface

Class Class1 : Implements Interface1, Interface2
  Sub Foo() Implements Interface1.Foo
    '*** method implementation
  End Sub
  Sub Bar() Implements Interface2.Bar
    '*** method implementation
  End Sub
End Class

Class Class2 : Implements Interface1, Interface2
  Sub FooBar() Implements Interface1.Foo, Interface2.Bar
    '*** method implementation
  End Sub
End Class
```

Interface members can be renamed or hidden within public member list for a class

```
Module MyApp
  Sub Main()
    Dim refA As New Class1
    Dim refB As Interface1 = refA
    '*** legal
    refA.Bob
    refB.Foo
    refB.Bar
    '*** illegal
    refA.Foo
    refA.Bar
  End Sub
End Module
```

```
Interface Interface1
  Sub Foo()
  Sub Bar()
End Interface

Class Class1 : Implements Interface1
  '*** method renamed
  Sub Bob() Implements Interface1.Foo
    '*** impl
  End Sub
  '*** method hidden
  Private Sub BarImpl() _
              Implements Interface2.Bar
    '*** impl
  End Sub
End Class
```

Interfaces and inheritance

- An interface can inherit from one or more interfaces
  - Derived interface takes on all constraints of base interface(s)
  - Derived interface can add addition constraints of it own
  - Implementing derived interface implies also implementing base interface(s)
  - Objects that are type-compatible with derived interface can be used anywhere a base interface is expected

Two variations on implementing a derived interface

```
Interface IAnimal
  Sub Breathe()
End Interface

Interface IHuman : Inherits IAnimal
  Sub Speak()
End Interface

Class Manager : Implements IAnimal, IHuman
  Sub Breathe() Implements IAnimal.Breathe
    '*** method implementation
  End Sub
  Sub Speak() Implements IHuman.Speak
    '*** method implementation
  End Sub
End Class

Class Programmer : Implements IHuman
  Sub Breathe() Implements IHuman.Breathe
    '*** method implementation
  End Sub
  Sub Speak() Implements IHuman.Speak
    '*** method implementation
  End Sub
End Class
```

Derived type compatibility implies base type compatibility

```
Module MyApp

  Sub Main()
    Dim obj1 As IHuman = New Manager
    ProcessAnimal(obj1)
    ProcessHuman(obj1)
    Dim obj2 As IHuman = New Programmer
    ProcessAnimal(obj2)
    ProcessHuman(obj2)
  End Sub

  '*** method accepts any IAnimal-compatible object
  Sub ProcessAnimal(obj As IAnimal)
    obj.Breathe()
  End Sub

  '*** method accepts any IHuman-compatible object
  Sub ProcessHuman(obj As IHuman)
    obj.Speak()
  End Sub

End Module
```

Interface conversion and discovery

- It's often necessary to convert between interface types and to test for interface support
  - Implicit casting is always allowed between compatible types
  - **`Option Strict`** requires explicit casting between non-compatible types
  - VB.NET provides the **`CType`** function for explicit casting
  - You can query an object at runtime to see if it supports a specific interface
  - VB.NET provides **`TypeOf`** and **`Is`** keywords to test for interface support

Option Strict forces explicit casting between non-compatible types

```
'*** compiles with or without Option Strict
Dim ref1 As New Programmer
Dim ref2 As IHuman = ref1
Dim ref3 As IAnimal = ref2

'*** doesn't compile unless Option Strict is disabled
Dim ref4 As IAnimal = New Programmer
Dim ref5 As IHuman = ref4      '*** illegal implicit conversion
Dim ref6 As Programmer = ref5  '*** illegal implicit conversion

'*** compiles with or without Option Strict
Dim ref7 As IAnimal = New Programmer
Dim ref8 As IHuman = CType(ref7, IHuman)
Dim ref9 As Programmer = CType(ref8, Programmer)
```

Converting between interfaces and testing for interface support

```
Sub ProcessAnimalAsHuman(param1 As IAnimal)

  '*** Example 1: testing for interface support
  Try
    Dim ref1 As IHuman = CType(param1, IHuman)
    ref1.Speak
  Catch ex As System.InvalidCastException
    '*** degrade gracefully if interface isn't supported
    Console.WriteLine("Oops, this animal doesn't support IHuman!")
  End Try

  '*** Example 2: testing for interface support
  If TypeOf param1 Is IHuman Then
    Dim ref2 As IHuman = CType(param1, IHuman)
    ref2.Speak
  Else
    '*** degrade gracefully if interface isn't supported
    Console.WriteLine("Oops, this animal doesn't support IHuman!")
  End If

End Sub
```

Base classes and inheritance

- Every class (except **`System.Object`**) inherits from a base class
  - Each class inherits from one (and only one) base class
  - Class definition uses **`Inherits`** keyword to specify base class
  - Class declared without explicit base class inherits from **`System.Object`**
  - Class defined as **`NotInheritable`** cannot be used as a base class

Specifying a base class

```
'*** implicitly derive from System.Object
Public Class Class1
  '*** class member declarations go here
End Class

'*** explicitly derive from System.Object
Public Class Class2
             Inherits System.Object
  '*** class member declarations go here
End Class

'*** derive one user-defined class from another
Public Class Class3
             Inherits Class1
  '*** class member declarations go here
End Class
```

```
‘*** C++ wanna-be syntax
Public Class Class1 : Inherits System.Object
  '*** class member declarations go here
End Class

Public Class Class2 : Inherits Class1
  '*** class member declarations go here
End Class
```

You cannot inherit from a class marked as `NotInheritable`

```
Public NotInheritable Class Class1
  '*** class member declarations go here
End Class

'*** compile error – cannot inherit from a sealed class
Public Class Class2 : Inherits Class1
  '*** class member declarations go here
End Class
```

Base/derived relationship

- Derived class inherits all base class members
  - Base class **`Private`** members are hidden from derived class
  - Base class **`Protected`** members are accessible to derived class
  - Base class **`Public`** members are part of derived class contract
  - Base class interface list is part of derived class contract
  - Derived type is compatible with everything the base type is

Private, Protected and Public members

```
'*** base class
Public Class Class1
  Private x As Integer
  Protected y As Integer
  Public z As Integer
End Class

'*** derived class
Public Class Class2 : Inherits Class1
  Shared Sub Foo()
    x = 10 '*** illegal
    y = 20 '*** legal
    z = 30 '*** legal
  End Sub
End Class

'*** client
Module MyApp
  Sub Main()
    Dim obj As New Class2
    obj.x = 10 '*** illegal
    obj.y = 20 '*** illegal
    obj.z = 30 '*** legal
  End Sub
End Module
```

```
Interface IPerson
  Property Name() As String
  Sub Speak()
End Interface

Public Class Person : Implements IPerson
  Protected m_Name As String
  Property Name() As String Implements IPerson.Name
    Set
      m_Name = value
    End Set
    Get
      Return m_Name
    End Get
  End Property
  Sub Speak() Implements IPerson.Speak
    Console.WriteLine("Hi. I am a person named " & m_Name)
  End Sub
End Class

Class Programmer : Inherits Person
  '*** extended definition goes here
End Class

Class Manager : Inherits Person
  '*** extended definition goes here
End Class
```

Derived types are compatible with their base types

```
Module MyApp

  Sub Main()
    '*** Programmer instance is compatible with IPerson and Person type
    Dim obj1 As New Programmer
    obj1.Name = "Bob"
    Foo(obj1)
    Bar(obj1)
    '*** Manager instance is compatible with IPerson and Person type
    Dim obj2 As New Manager
    obj2.Name = "Lucy"
    Foo(obj2)
    Bar(obj2)
  End Sub

  Sub Foo(param1 As IPerson)
    param1.Speak()
  End Sub

  Sub Bar(param1 As Person)
    param1.Speak()
  End Sub

End Module
```

Base types and constructors

- Constructors and base types have "issues"
  - Derived type's contract does not include base type constructor(s)
  - Derived type must provide its own constructor(s)
  - Derived type constructor(s) must call base type constructor(s)
  - Constructors fire in order of least derived to most derived

Base classes and constructors

```
Public Class Person

  Protected m_Name As String

  Sub New(Name As String)
    '*** implicit call to default constructor of System.Object
    m_Name = Name
  End Sub

End Class

Class Programmer : Inherits Person

  Sub New(Name As String)
    MyBase.New(Name) '*** explicit call to base class constructor
    '*** class-specific initialization goes here
  End Sub

End Class
```

Static binding and member shadowing

- Standard class members are accessed based on static type information
  - Static binding based on reference variable type (not object type)
  - Derived class cannot override base class members
  - Derived class can shadow base class members
  - Shadowed members must be marked with **Shadows** keyword
  - Shadowing a method hides all base class methods of the same name
  - Shadowing can cause confusion and should be used with caution

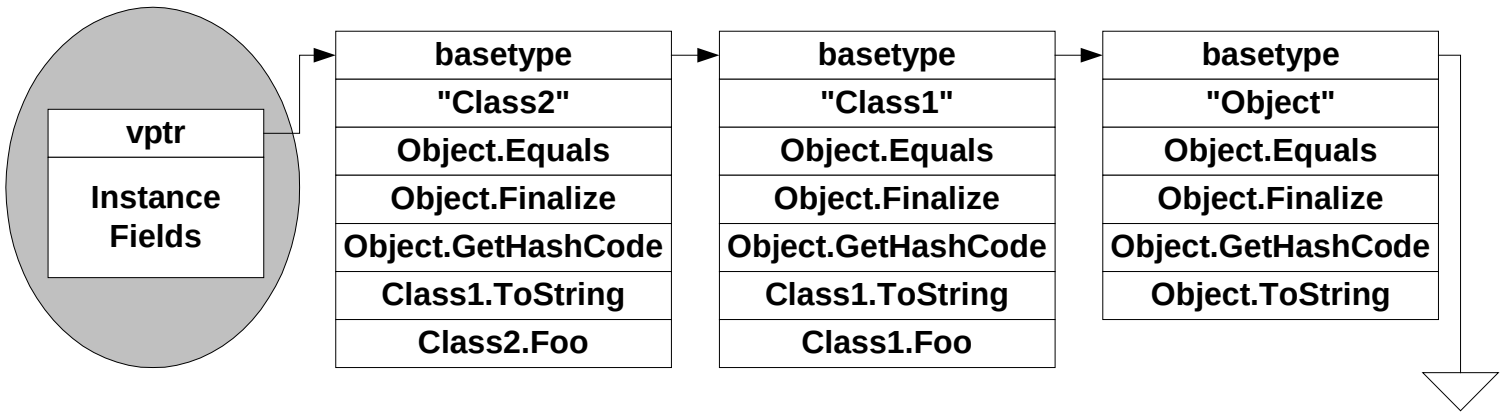
Field and method shadowing

```
Public Class Class1
  Public x As Integer = 10
  Sub Foo()
    '*** implementation
  End Sub
End Class

Public Class Class2 : Inherits Class1
  Shadows Public x As Integer = 20
  Shadows Sub Foo()
    '*** implementation
  End Sub
End Class

Module MyApp
  Sub Main()
    '*** access object through Class2 ref
    Dim refA As Class2 = New Class2
    Dim i As Integer = refA.x '*** accesses Class2.x
    refA.Foo() '*** executes Class2.Foo
    '*** access same object through Class1 ref
    Dim refB As Class1 = refA
    Dim j As Integer = refB.x '*** accesses Class1.x
    refB.Foo() '*** executes Class1.Foo
  End Sub
End Module
```

Dynamic binding and virtual methods

- Virtual methods are overridable and invoked based on dynamic type information
  - Dynamic binding based on object type (not reference variable type)
  - Virtual methods are marked with **`Overridable`** keyword
  - Derived class can inherit or override base class implementation
  - Derived class must use **`Overrides`** keyword when overriding
  - Overriding method can explicitly call base class implementation if needed

Virtual function tables



Virtual method dispatching in action

```
Public Class Class1
  Overridable Sub Foo()
    '*** implementation
  End Sub
  Overrides Function ToString() As String
    '*** implementation
  End Function
End Class

Public Class Class2 : Inherits Class1
  Overrides Sub Foo()
    '*** implementation
  End Sub
End Class

Module MyApp
  Sub Main()
    '*** access object through Class2 ref
    Dim refA As Class2 = New Class2
    refA.Foo() '*** executes most derived impl (Class2.Foo)
    Dim s1 As String = refA.ToString() '*** calls Class1.ToString
    '*** access same object through Class1 ref
    refB.Foo() '*** executes most derived impl (Class2.Foo)
    Dim s2 As String = refB.ToString() '*** calls Class1.ToString
  End Sub
End Module
```

Chaining a call to the base class

```
Public Class Class1
  Overridable Sub Foo()
    '*** implementation
  End Sub
End Class

Public Class Class2 : Inherits Class1
  Overrides Sub Foo()
    '*** Class2-specific code can go here
    MyBase.Foo() '*** chain a call to base class
    '*** Class2-specific code can go here
  End Sub
End Class
```

Abstract classes

- Abstract classes can only be used as a base class
  - **`MustInherit`** keyword marks a class as abstract
  - Abstract classes cannot be used with the **`New`** operator
  - **`MustInherit`** classes can contain members marked **`MustOverride`**
  - **`MustOverride`** methods provide no implementation and must be overridden
  - Restricting derivation using access modifiers also useful technique

Base classes and design

- Abstract bases can be used as a poor man's interface - don't take a vow of poverty naively
  - Mandating a base class is often a bad design choice
  - Mandated base classes restrict options for derived type
  - Writing bullet-proof base classes is really hard (really)
  - Primary reason for base classes is refactoring within a component

Refactoring common functionality

```
Public Class Manager : Implements IPerson
  Private m_Name As String
  Property Name() As String _
           Implements IPerson.Name
    '*** implementation
  End Property
  Sub Speak() Implements IPerson.Speak
    '*** implementation
  End Sub
End Class
```

```
Public Class Programmer : Implements IPerson
  Private m_Name As String
  Property Name() As String _
           Implements IPerson.Name
    '*** implementation
  End Property
  Sub Speak() Implements IPerson.Speak
    '*** implementation
  End Sub
End Class
```

Before Refactoring

After Refactoring

```
 Public Class Person : Implements IPerson
  Private m_Name As String
  Property Name() As String Implements IPerson.Name
    '*** implementation
  End Property
  Overridable Sub Speak() Implements IPerson.Speak
    '*** implementation
  End Sub
End Class
```

```
Public Class Manager : Inherits Person
  Overrides Sub Speak()
    '*** custom implementation
  End Sub
End Class
```

```
Public Class Programmer : Inherits Person
  Overrides Sub Speak()
    '*** custom implementation
  End Sub
End Class
```

Structured exception handling

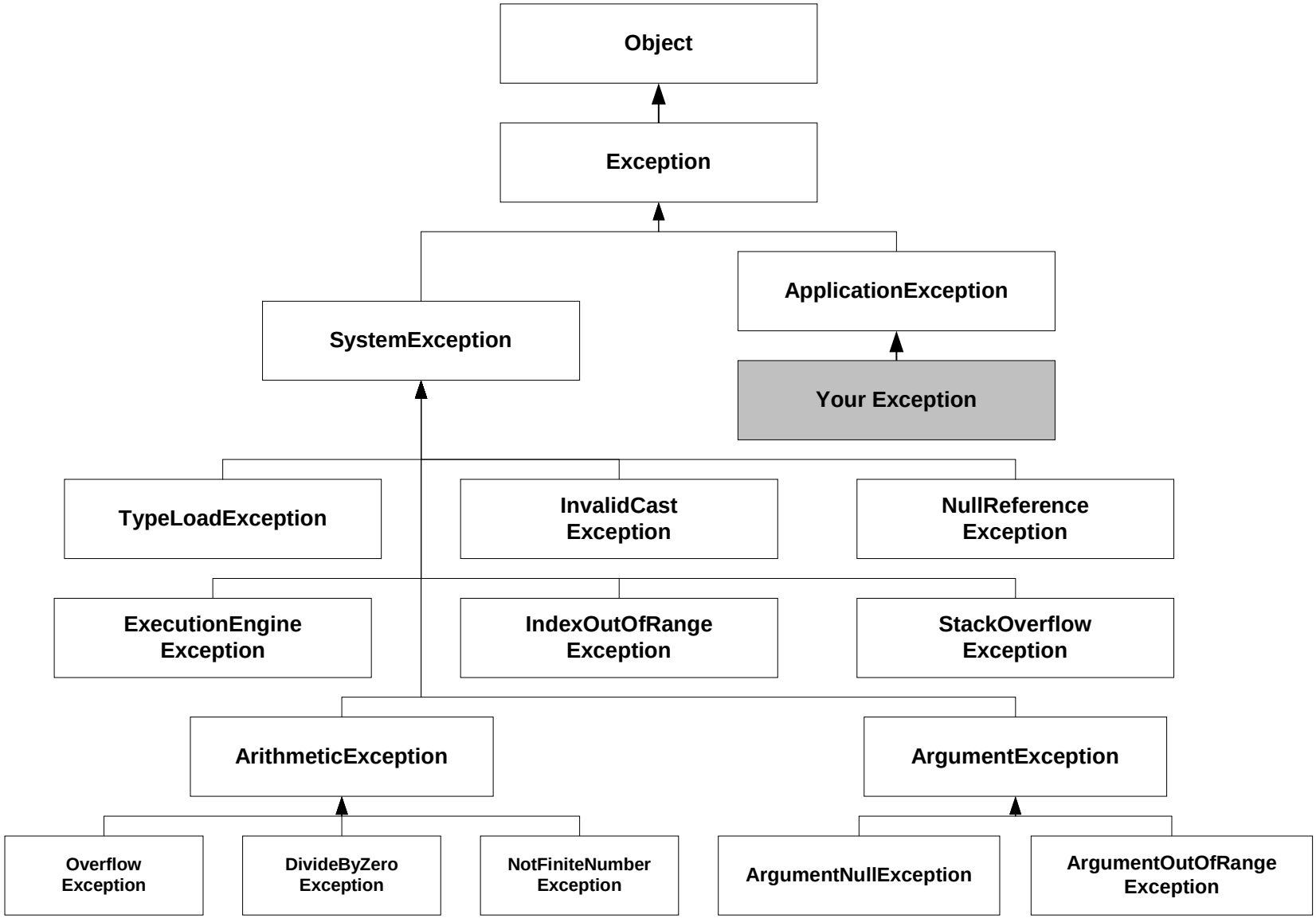
Error handling

- Operations raise exceptions to indicate abnormal operation
  - Normal method termination achieved using **`Return`** statement
  - Abnormal method termination achieved using **`Throw`** statement
  - Abnormal termination loses normal return value and out parameters
  - Abnormal termination returns an exception object describing the error
  - Caller may handle exception or allow it to propagate

Exceptions defined

- Exceptions are instances of classes that derive from **`System.Exception`**
  - System-generated exceptions derive from **`System.SystemException`**
  - Application-specific exceptions derive from **`System.ApplicationException`**
  - Exceptions maintain a chain of cascaded exceptions
  - Derived exception types can add public properties and methods

Exceptions and the CLR type system



System.Exception

```
Namespace System
  Public Class Exception
    '*** constructors
    Overloads Public Sub New()
    Overloads Public Sub New(message As String)
    String (message As String, inner As Exception)
    ' members
    Overridable ReadOnly Property InnerException As Exception
    ReadOnly Property Message As String
    ReadOnly Property TargetSite As MethodBase
    MethodBase StackTrace As String
    Overridable Property Source As String
    Overridable ReadOnly Property HelpLink As String
    Overridable Function SetHelpLink(url As String) As Exception
    Overridable Function GetBaseException() As Exception
    Protected HResult As Integer
  End Class
End Namespace
```

Raising exceptions

- Exceptions are raised implicitly by the runtime or explicitly via the Throw statement
  - The runtime signals abnormal conditions by raising exceptions
  - The .NET framework classes signal abnormal conditions using an explicit Throw statement
  - Your methods may raise exceptions using the Throw statement as well
  - The **`Throw`** statement requires an object of at least **`System.Exception`**

Catching exceptions

- Exceptions are caught using language-specific constructs
  - The Try-Catch statement protects the Try-block with one or more exception handlers
  - Exception handlers are selected based on exception types
  - Handled exceptions do not propagate beyond the Try-Catch statement
  - Unhandled exceptions propagate to the next Try-Catch statement in scope

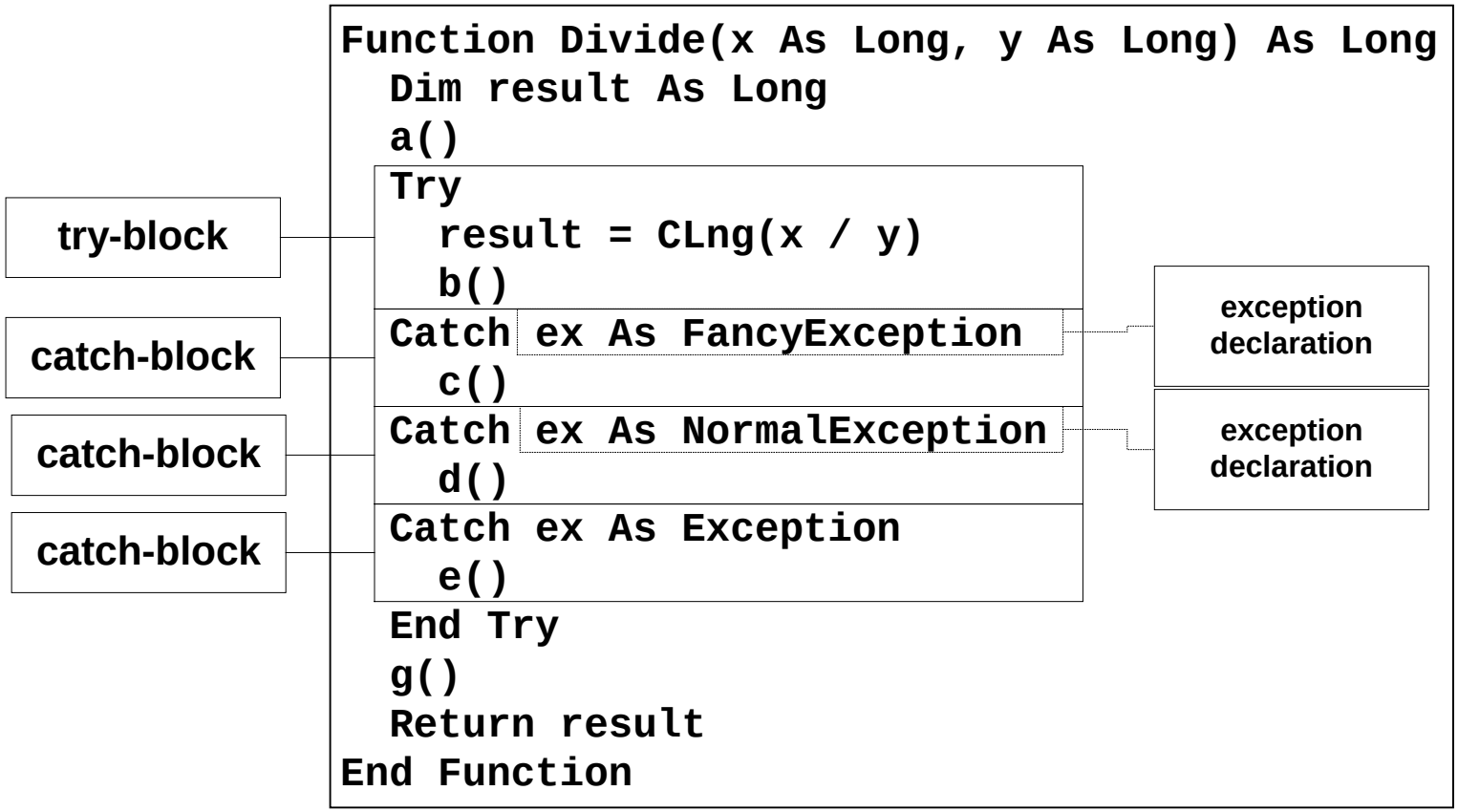
Try-Catch statement

Try-block

Catch-block

```
Function Divide(x As Long, y As Long) As Long
  Dim result As Long
  a()
  Try
    result = CLng(x / y)
    b()
  Catch ex As Exception
    e()
  End Try
  g()
  Return result
End Function
```

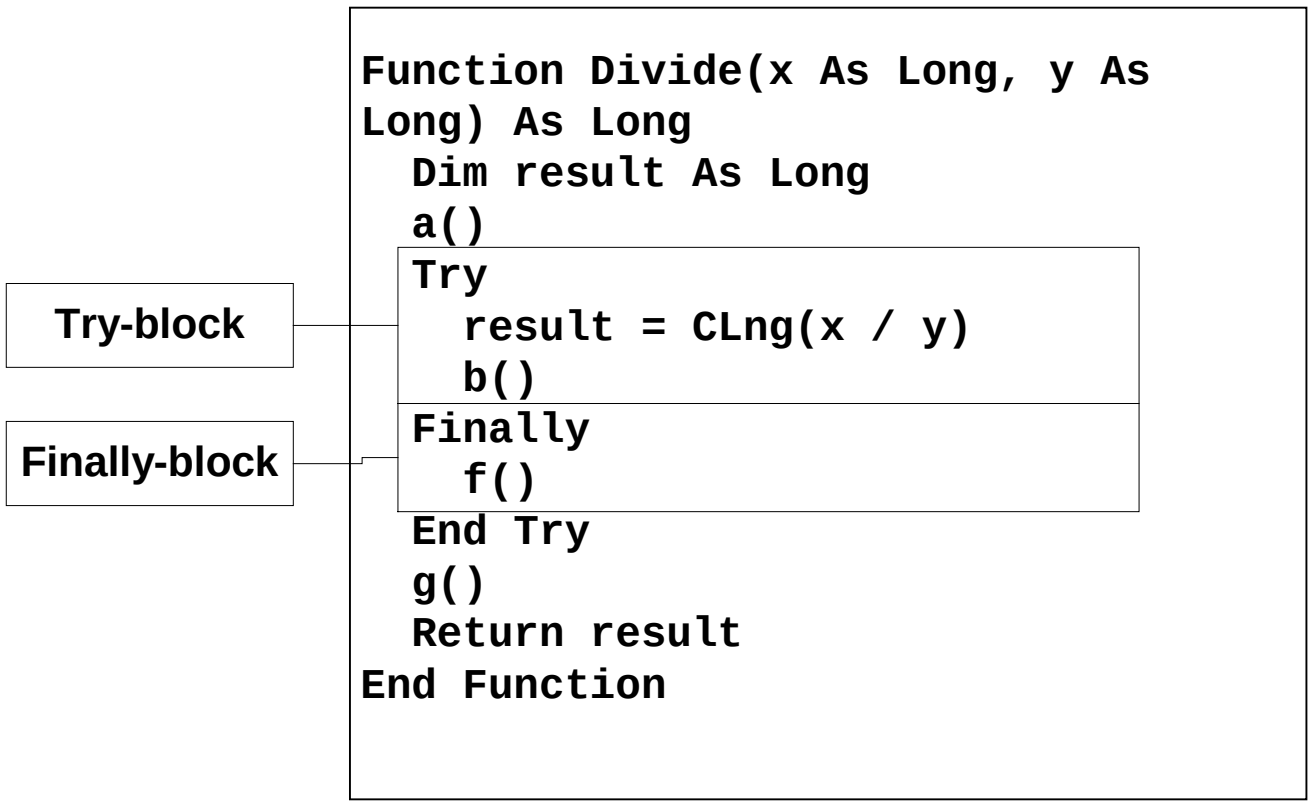
Try-Catch statement with declarations



Termination handlers

- Termination handlers ensure cleanup in the face of multiple exit points
  - Expressed using Try-Finally or Try-Catch-Finally statements
  - Allows cleanup code to run when exceptions are raised
  - Allows cleanup code to run when multiple **`Return`** statements are used

Try-Finally statement



Try-Catch-Finally statement revisited

try-block

catch-block

catch-block

catch-block

finally-block

```
Function Divide(x As Long, y As Long) As Long
  Dim result As Long
  a()
  Try
    result = CLng(x / y)
    b()
  Catch ex As FancyException
    c()
  Catch ex As NormalException
    d()
  Catch ex As Exception
    e()
  Finally
    f()
  End Try
  g()
  Return result
End Function
```

exception declaration

exception declaration

User-defined exception

```
Public Class Account

  Sub Withdraw()
    '*** throw custom exception if business rule is violated
    Throw New AccountException("You don't have enough money)
  End Sub

End Class

Class AccountException : Inherits ApplicationException

  '*** explicit constructor required
  Sub New(msg As String)
    MyBase.New(msg) '*** call base constructor
  End Sub

End Class
```

Objects and values

Classes, reference variables and objects

- Classes are used to create objects while reference variables are used to access objects
  - Objects are instances of classes (also known as reference types)
  - Objects are heap-based and carry an object header
  - Reference variables may be declared using classes or interfaces
  - Reference variables have a default value of **`Nothing`**
  - Reference variables must contain a value other than **`Nothing`** to be useful
  - Reference variables may refer to any object that is compatible with their declared type

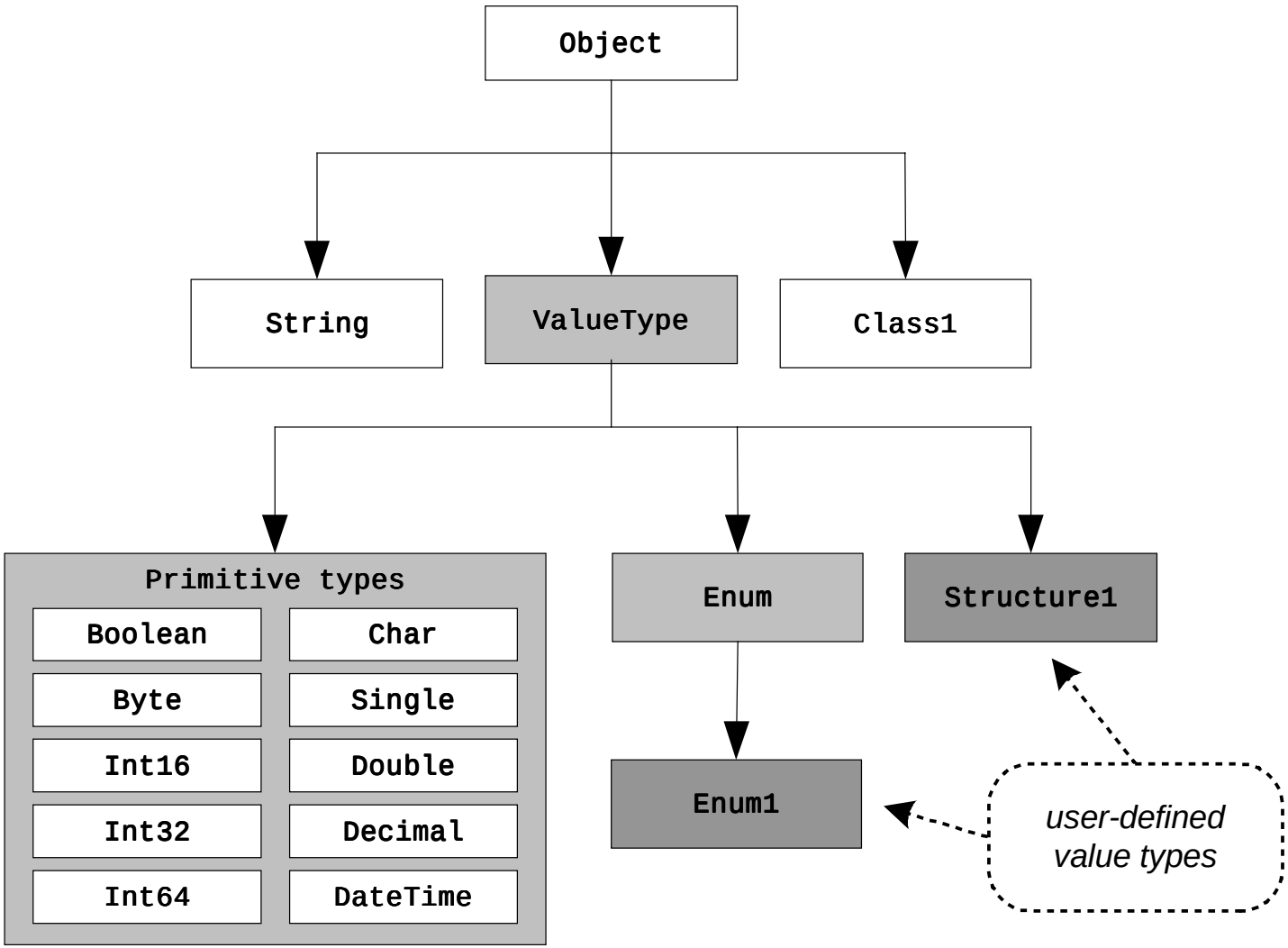
Figure 6.2: Objects and object references

```
Public Class Person
  Sub HealThyself()
    ' miracle implementation
  End Sub
End Class

Module MyApp
  Sub Main
    Dim don As Person    ' don is a reference
    don.HealThyself()    ' runtime exception - null reference
    don = New Person()   ' don now refers to an object
    don.HealThyself()    ' legal - valid reference
    don = Nothing        ' don no longer references the object
    don.HealThyself()    ' runtime exception - null reference
  End Sub
End Module
```

Reference types vs. value types

- The runtime distinguishes between reference and value types
  - Reference types yield "true" objects
  - Value types yield formatted memory
  - All classes are reference types
  - Primitives, structures and enumerations are value types
  - Instances of value types do not have an object header
  - Instances of value types are not independently garbage collected
  - Value types have several limitations that reference types do not have

The CLR type system



User-defined value types

```
Public Structure Size
  Public Height As Integer
  Public Weight As Integer
End Structure
```

```
Public Enum Breath
  None
  Sweet
  Garlicy
  Rancid
  Horrid
End Enum
```

```
Public Enum <System.Flags> Organs As Short
  None = 0
  Heart = 1
  Lung = 2
  Liver = 4
  Kidney = 8
End Enum
```

Value types and memory

- Value types are allocated using embedded memory based on where the instance is declared
  - Stack-based memory used for local variables and method parameters
  - Memory allocation embedded inside heap-based object for value type field
  - Value types do not require **`New`** operator
  - **`New`** operator can be used to call value type's parameterized constructor
  - Value type assignment yields second copy, not second reference

Using value and reference types

```
Public Structure Size
  Public Height As Integer
  Public Weight As Integer
End Structure

Public Class CSize
  Public Height As Integer
  Public Weight As Integer
End Class

Sub Main
  Dim var1 As Size     ' var1 is an instance of Size
  var1.Height = 100    ' legal
  Dim var2 As CSize    ' var2 is a reference
  var2.Height = 100    ' illegal, var2 = Nothing
  var2 = New CSize()   ' var2 refers to an instance of CSize
  var2.Height = 100    ' legal, var2 <> Nothing
End Sub
```

Cloning

- Some objects can be deep or shallow copied
  - Objects typically implement the **`System.ICloneable`** interface
  - Built-in method **`Object.MemberwiseClone`** performs shallow copy
  - Deep copy implemented by hand

Implementing System.ICloneable

```
Namespace System
  Public Interface ICloneable
    Function Clone() As Object
  End Interface
End Namespace
```

```
Imports System

Public Class Person : Implements ICloneable

  Function Clone() As Object Implements ICloneable.Clone
    ' this implementation produces a shallow copy
    Return Me.MemberwiseClone()
  End Function

End Class
```

Implementing Clone with a deep copy

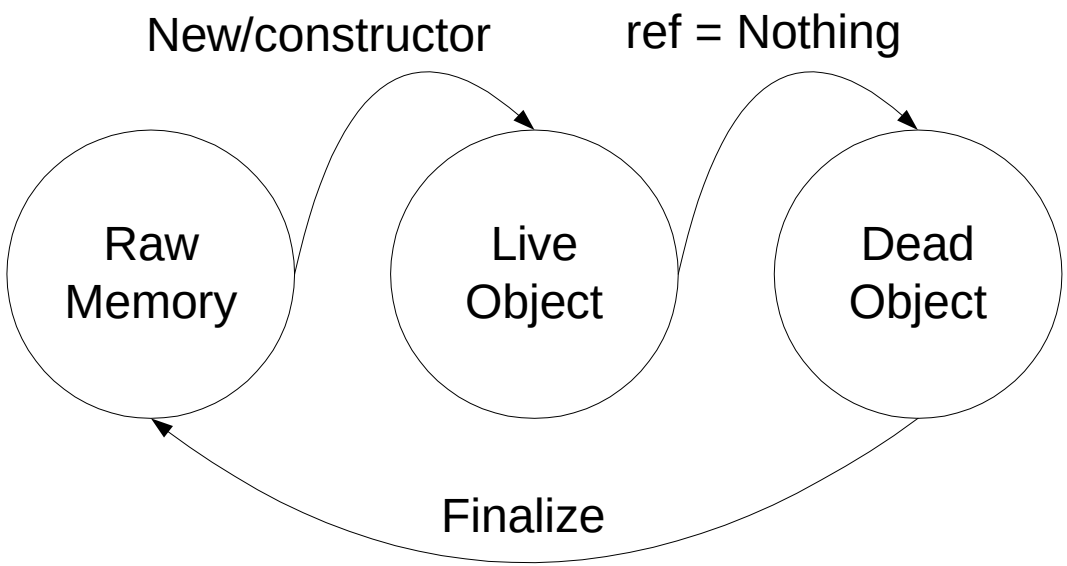
```
Public Class Marriage : Implements ICloneable
  Public Girl As Person
  Public Boy As Person
  Function Clone() As Object Implements ICloneable.Clone
    ' shallow copy first
    Dim result As Marriage
    result = CType(Me.MemberwiseClone(), Marriage)
    ' deep copy each field
    result.Girl = CType(Me.Girl.Clone(), Person)
    result.Boy = CType(Me.Boy.Clone(), Person)
    Return result
  End Function
End Class
```

Object lifecycle and finalization

- Objects that are no longer referenced may be garbage collected
  - Manual resource reclamation not part of model
  - Simplifies resource ownership issues
  - Garbage collector exposed via shared members on **`System.GC`** class
  - Objects are notified when they are garbage collected
  - GC calls well-known **`Finalize`** method
  - Overriden version of **`Finalize`** should call base class **`Finalize`**

References and finalization

```
Dim obj1 As Object = New Person()
Dim obj2 As Object = obj1
obj2 = Nothing
System.GC.Collect() ' nothing happens
obj1 = Nothing      ' object available for GC
System.GC.Collect() ' memory reclaimed
```



Programmer hygiene

- Deterministic finalization is (largely) the programmer's responsibility
  - Expensive resources require special attention
  - Explicit **`Dispose`** method common idiom
  - **`IDisposable`** interface added to CLR in Beta 2 to formalize idiom
  - Can manually suppress finalization or retrigger using **`System.GC`** class
  - Beware multiple exit points (use try/finally)

Value types and boxing

- Instances of value types can be "boxed" to support object references to value types
  - Value types lack "objectness" until boxed
  - Boxed object is an independent clone
  - Boxed object can be copied back into instance (unboxing)
  - Boxing key to using System.Object as universal type

Boxing in action

```
Sub Main()
  Dim var1 As Structure1
  Foo(var1) – boxing occurs
End Sub

Sub Foo(param1 As Object)
  '*** param1 is a ref pointing to
  '*** a heap-based copy (i.e. box)
  '*** of the structure variable var1
End Sub
```

Delegates and events

Delegates defined

- Delegates are objects that invoke methods on another object (or class)
  - Delegates act as type-safe function pointers
  - Delegates are objects that are instances of a delegate type
  - Delegates were originally architected for the MSJVM

Delegate signatures

- A delegate type supports exactly one method signature
  - Declaring a delegate type is language-specific
  - VB.NET uses the **`Delegate`** keyword and function declaration syntax
  - Syntax for invoking delegates is language-specific
  - VB.NET uses function call syntax treating the delegate variable as a function name

Figure 9.3: Using a delegate type

```
Delegate Sub Delegate1(s As String)

Module Module1
  Sub FireDelegate(del As Delegate1)
    del("Here we go")
  End Sub
End Module
```

Delegate constructors

- Delegate constructors accept a method address
  - Each language has its own syntax
  - VB.NET uses the **`AddressOf`** keyword to retrieve a method address
  - Use **`AddressOf ObjRef.MethodName`** for instance methods
  - Use **`AddressOf ClassName.MethodName`** for shared methods

Creating a delegate

```
Delegate Sub Delegate1(s As String)

Class Class1
  Sub JoeBobsHandler(s As String)
    Console.WriteLine("Joe Bob's handler: " & s)
  End Sub
End Class

Module MyApp
  Sub Main
    Dim handler As New Class1
    Dim del1, del2 As Delegate1
    ' longhand syntax for creating and using a delegate
    del1 = New Delegate1(AddressOf handler.JoeBobsHandler)
    del1.Invoke("Test message 1")
    ' shorthand syntax for creating and using a delegate
    del2 = AddressOf handler.JoeBobsHandler
    del2("Test message 2")
  End Sub
End Module
```

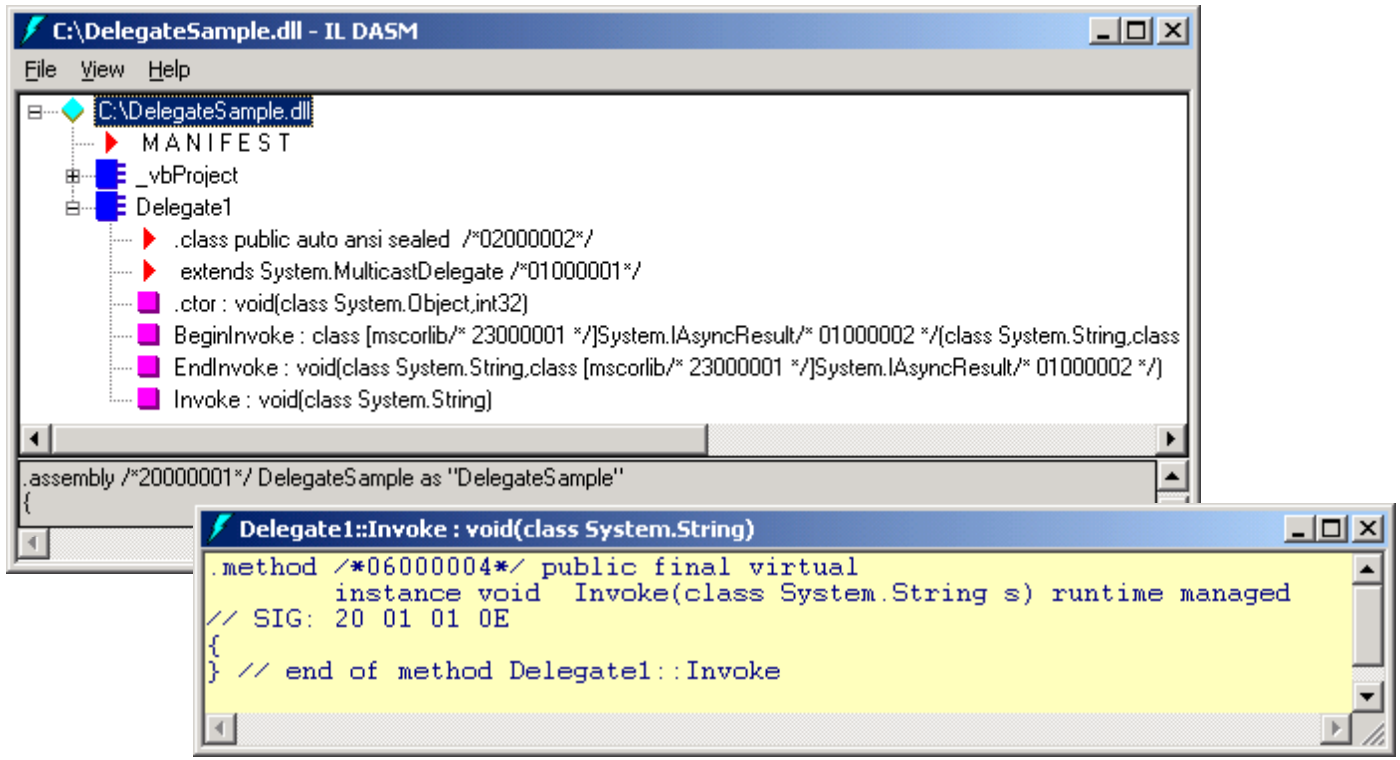
Binding a delegate to a shared method

```
Delegate Sub Delegate1(s As String)

Class Class2
  Shared Sub JimBobsHandler(s As String)
    Console.WriteLine("Jim Bob's handler: " & s)
  End Sub
End Class

Module MyApp
  Sub Main
    Dim del As Delegate1
    del = AddressOf Class2.JimBobsHandler
    del("Test message")
  End Sub
End Module
```

System.Delegate

- Delegates must extend System.Delegate (or a subclass of System.Delegate)
  - Abstract **`System.Delegate`** type simply signals core plumbing in MSCOREE
  - Derived type must provide an **`Invoke`** method of the appropriate signature
  - Compilers/languages perform magic to make this usable

Delegates revealed



System.MulticastDelegate

- **System.MulticastDelegate** forwards one invocation to multiple targets
  - **System.Delegate.Combine** supports combining delegates together
  - Default implementation throws **System.MulticastNotSupportedException**
  - **System.MulticastDelegate** overrides to support chain of delegates
  - **System.MulticastDelegate** does not perform magic

Using multicast delegates

```
Delegate Sub Delegate1(s As String)

Class Class1
  Shared Sub JoeBobsHandler(s As String)
    Console.WriteLine("Joe Bob's handler: " & s)
  End Sub
End Class

Class Class2
  Shared Sub JimBobsHandler(s As String)
    Console.WriteLine("Jim Bob's handler: " & s)
  End Sub
End Class

Module MyApp
  Sub Main
    Dim d1, d2, d3 As Delegate1
    d1 = AddressOf Class1.JoeBobsHandler
    d2 = AddressOf Class2.JimBobsHandler
    ' create d3 which is a mutlicast of d1 and d2
    d3 = CType(System.Delegate.Combine(d1, d2), Delegate1)
    d3("Here we go!")
    Console.ReadLine()
  End Sub
End Module
```

Asynchronous invocation

- Asynchronous method invocation is exposed by the runtime using Delegates
  - Delegate type's **`BeginInvoke`** method enqueues request onto a pool of worker threads
  - An **`IAsyncResult`** object is returned by the runtime to represent the call in progress
  - **`ByRef`** parameters and return value must be harvested using **`EndInvoke`** method
  - You can poll for completion using **`IAsyncResult.IsCompleted`**
  - You can register callback object for completion notifcation

Synthesized synchronous/asynchronous delegate methods

```
Public Delegate Function FooDelegate() As String
```

```
'*** pseudo-VB.NET/IL
public class FooDelegate : Inherits MulticastDelegate
  '*** synchronous invocation
  Function Invoke() As String
  '*** issue asynchronous invocation
  Function BeginInvoke(cb As AsyncCallback, _
                       asyncState As Object) As IAsyncResult
  '*** harvest asynchronous invocation results
  Function EndInvoke(result As IAsyncResult) As String
End Class
```

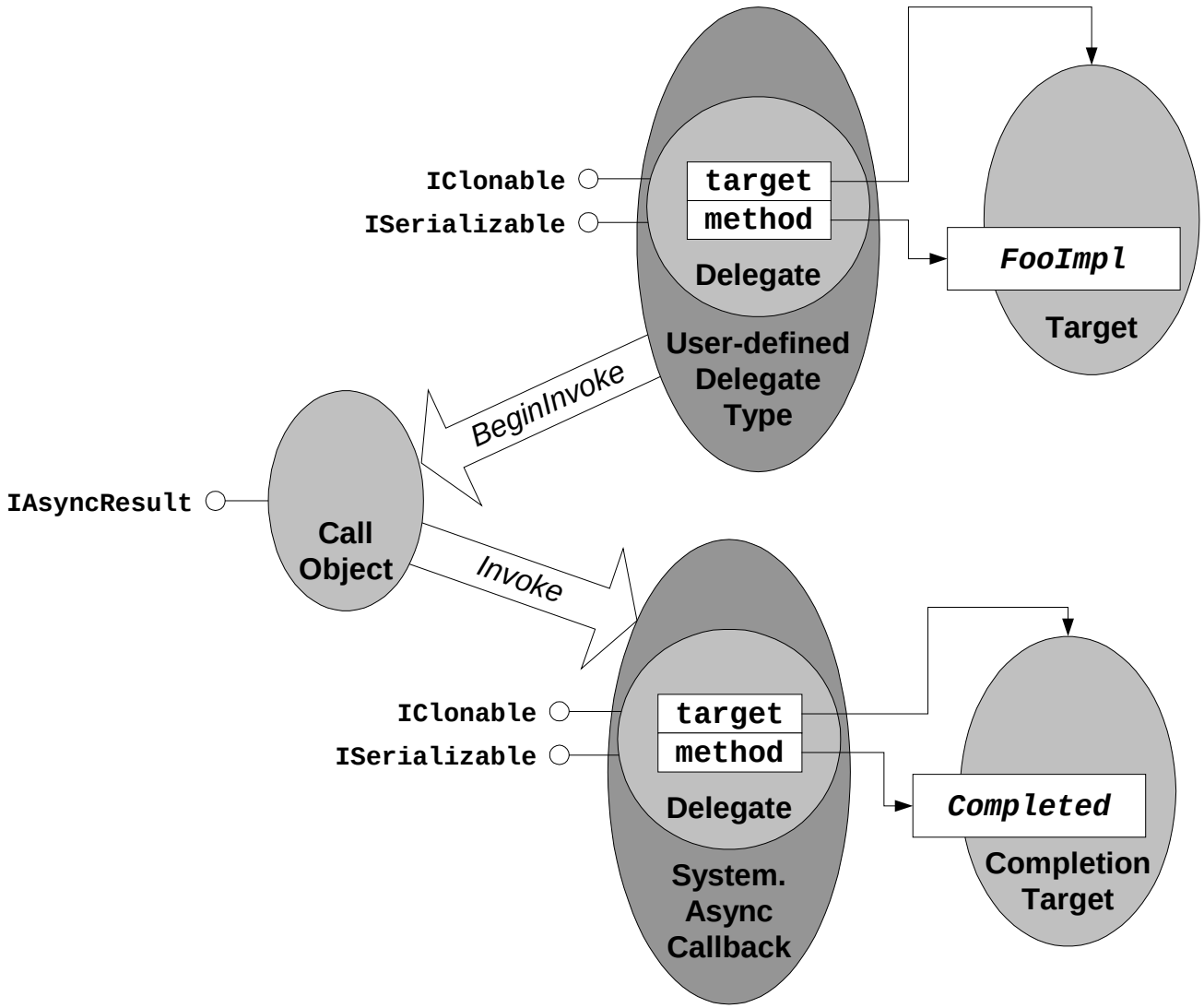
Asynchronous invocation with polling

```
Public Delegate Function FooDelegate() As String

Module MyApp

  Function FooImpl() As String
    Console.WriteLine("FooImpl")
    Return "Hi there"
  End Function

  Sub Main()
    Dim d1 As New FooDelegate(AddressOf FooImpl)
    Dim ar As IAsyncResult = d1.BeginInvoke(Nothing, Nothing)
    '*** poll until call is complete
    Do Until ar.IsCompleted
      System.Threading.Thread.Sleep(0)
    Loop
    '*** harvest ref/out parameters
    Dim s As String = d1.EndInvoke(ar)
  End Sub

End Module
```

Asynchronous invocation with callback

```
Public Delegate Function FooDelegate() As String

Module MyApp

  Function FooImpl() As String
    Return "Hi there from FooImpl"
  End Function

  Sub Main()
    Dim d1 As New FooDelegate(AddressOf FooImpl)
    Dim cb As New AsyncCallback(AddressOf CallbackMethod1)
    '*** issue asynchronous call and register callback method
    Dim ar As IAsyncResult = d1.BeginInvoke(cb, d1)
  End Sub

  '*** this method will be called at call completion
  Sub CallbackMethod1(ar As IAsyncResult)
    Dim del As FooDelegate = CType(ar.AsyncState, FooDelegate)
    Dim s As String = del.EndInvoke(ar)
  End Sub

End Module
```

Asynchronous invocation



Events

- Events are pseudo-properties that automate delegate registration
  - Events are "properties" of delegate type
  - Events are based on delegates either implicitly or explicitly
  - Event declaration results in transparent creation of register/unregister methods
  - Compilers typically do syntactic magic if not genuine magic

Declaring an event

```
' class with event(s) defined acts as source
Public Class EventClass1
  Private MyData As String
  ' define event in terms of implicit delegate
  Event Sub OnDataChanged(NewValue As String)
End Class

' define explicit delegate for event
Delegate Sub DataChangedDelegate(NewValue As String)

Public Class EventClass2
  Private MyData As String
  ' define an event in terms of an existing delegate
  Event OnDataChanged As DataChangedDelegate
End Class
```

Firing an event

```
Public Class EventClass1
  Private MyData As String
  ' define event in terms of implicit delegate
  Event OnDataChanged(NewValue As String)
  Public Sub UpdateData(s As String)
    ' raise event when listeners require notification
    RaiseEvent OnDataChanged(s)
  End Sub
End Class
```

```
Public Class EventClass1
  Private MyData As String
  Event OnDataChanged(NewValue As String)
  Public Sub UpdateData(s As String)
    RaiseEvent OnDataChanged(s)
  End Sub
End Class

Class ListenerClass1
  Private WithEvents Source As EventClass1
  Public Sub New(ByVal s As EventClass1)
    Source = s
  End Sub
  Private Sub BobsHandler(NewValue As String) _
              Handles Source.OnDataChanged
    System.Console.WriteLine("Handler data update: " & NewValue)
  End Sub
End Class

Module MyApp
  Public Sub Main()
    Dim ec As New EventClass1
    Dim listener1 As New ListenerClass1(ec)
    ec.UpdateData("some new value")
    Console.ReadLine()
  End Sub
End Module
```

Agenda

- Intro to the CLR and VB.NET
- Adjusting to a new type system
- New OOP features in class design
- Interfaces and inheritance
- Structured exception handling
- Objects and values
- Delegates and events