Table of Contents (Overview)

Directory Structure

- **The course software installs to the root directory *C:\OIC\NetCs*.**
  - Example programs for each chapter are in named subdirectories of chapter directories **Chap01**, **Chap02**, and so on.
  - The **Labs** directory contains one subdirectory for each lab, named after the lab number. Starter code is frequently supplied, and answers are provided in the chapter directories.
  - The **Demos** directory is provided for hand-on work during lectures.
- **Data files install to the directory *C:\OIC\Data*.**
- **The directory *C:\OIC\Deploy* is provided to practice deployment.**

Table of Contents (Detailed)

Chapter 1

.NET Fundamentals

.NET Fundamentals

Objectives

After completing this unit you will be able to:

- **Understand the problems Microsoft .NET is designed to solve.**
- **Understand the basic programming model of Microsoft .NET.**
- **Understand the basic programming tools provided by Microsoft .NET.**

What Is Microsoft .NET?

- **Microsoft .NET was developed to solve three fundamental problems.**
- **First, the Microsoft Windows programming model must be unified to remove the widely varied programming models and approaches that exist among the various Microsoft development technologies.**
- **Second, Microsoft based solutions must be capable of interacting with the modern world of heterogeneous computing environments.**
- **Third, Microsoft needs a development paradigm that is capable of being expanded to encompass future development strategies, technologies, and customer demands.**

Open Standards and Interoperability

- **The modern computing environment contains a vast variety of hardware and software systems.**
  - Computers range from mainframes and high-end servers, to workstations and PCs, and to small mobile devices such as PDAs and cell phones.
  - Operating systems include traditional mainframe systems, many flavors of Unix, Linux, several versions of Windows, real-time systems and special systems such as PalmOs for mobile devices.
  - Many different languages, databases, application development tools and middleware products are used.
- **Applications need to be able to work in this heterogeneous environment.**
  - Even shrink-wrapped applications deployed on a single PC may use the Internet for registration and updates.
- **The key to interoperability among applications is the use of *standards*, such as HTML, HTTP, XML, SOAP, and TCP/IP.**

Windows Development Problems

- **In classic Windows development design and language choice often clashed.**
  - Visual Basic vs. C++ approach
  - IDispatch, Dual, or Vtable interfaces
  - VB vs. MFC
  - ODBC or OLEDB or ADO
- **Application deployment was hard.**
  - Critical entries in Registry for COM components
  - No versioning strategy
  - DLL Hell
- **Security was difficult to implement.**
  - No way to control code or give code rights to certain actions and deny it the right to do other actions.
  - Security model is difficult to understand. Did you ever pass anything but NULL to a LPSECURITY_ATTRIBUTES argument?
- **Too much time is spent in writing plumbing code that the system should provide.**
  - MTS/COM+ a step in the right direction.

Common Language Runtime

- **The first step in solving the three fundamental problems is for Microsoft .NET to provide a set of underlying services available to all languages.**
- **The runtime environment provided by .NET that provides these services is called the *Common Language Runtime* or CLR.**
  - A runtime provides services to executing programs.
  - Traditionally there are different runtimes for different programming environments. Examples of runtimes include the standard C library, MFC, the Visual Basic 6 runtime and the Java Virtual Machine.
- **These services are available to all languages that follow the rules of the CLR.**
  - C# and Visual Basic are examples of Microsoft languages that are fully compliant with the CLR requirements.
  - Not all languages use all the features of the CLR.
- **As a terminology note, beginning with .NET 2.0, Microsoft has dropped the ".NET" in the Visual Basic language.**
  - The pre-.NET version of the language is now referred to as Visual Basic 6 or VB6.

Serialization Example

- **Let us use serialization to illustrate how the CLR provides a set of services that unifies the Microsoft development paradigm.**
  - Every programmer has to do it.
  - It can get complicated with nested objects, complicated data structures, and a variety of data storages.
  - The programmer should also be able to override the system service if necessary.
- **See the *Serialize* example in this chapter.**

Serialization Example (Cont'd)

- **Ignore the language details covered in a later chapter.**

```
[Serializable]
class Customer
{
   public string name;
   public long id;
}
class Test
{
   static void Main(string[] args)
   {
      ArrayList list = new ArrayList();

      Customer cust = new Customer();
      cust.name = "Charles Darwin";
      cust.id = 10;
      list.Add(cust);

      cust = new Customer();
      cust.name = "Isaac Newton";
      cust.id = 20;
      list.Add(cust);

      foreach (Customer x in list)
         Console.WriteLine(x.name + ": " + x.id);

      Console.WriteLine("Saving Customer List");
      FileStream s = new FileStream("cust.txt",
                           FileMode.Create);
      SoapFormatter f = new SoapFormatter();
      SaveFile(s, f, list);
```

Serialization Example (Cont'd)

```
      Console.WriteLine("Restoring to New List");
      s = new FileStream("cust.txt",
         FileMode.Open);
      f = new SoapFormatter();
      ArrayList list2 =
         (ArrayList)RestoreFile(s, f);

      foreach (Customer y in list2)
         Console.WriteLine(y.name + ": " + y.id);
   }

   public static void SaveFile(Stream s,
   IFormatter f, IList list)
   {
      f.Serialize(s, list);
      s.Close();
   }

   public static IList RestoreFile(Stream s,
   IFormatter f)
   {
      IList list = (IList)f.Deserialize(s);
      s.Close();
      return list;
   }
}
```

Attribute-Based Programming

- **We add two Customer objects to the collection, and print them out. We save the collection to disk and then restore it. The identical list is printed out.**

```
Charles Darwin: 10
Isaac Newton: 20
Saving Customer List
Restoring to New List
Charles Darwin: 10
Isaac Newton: 20
Press enter to continue...
```

- **We wrote no code to save or restore the list!**
  - We just annotated the class we wanted to save with the **Serializable** attribute.
  - We specified the format (SOAP) that the data was to be saved.
  - We specified the medium (disk) where the data was saved.
  - This is typical class partitioning in the .NET Framework.
- **Attribute-based programming is used throughout .NET to describe how code and data should be treated by the framework.**

Metadata

- **The compiler adds the *Serializable* attribute to the *metadata* of the Customer class.**
- **Metadata provides the Common Language Runtime with information it needs to provide services to the application.**
  - Version and locale information
  - All the types
  - Details about each type, including name, visibility, etc.
  - Details about the members of each type, such as methods, the signatures of methods, etc.
  - Attributes
- **Metadata is stored with the application so that .NET applications are self-describing. The registry is not used.**
  - The CLR can query the metadata at runtime. It can see if the **Serializable** attribute is present. It can find out the structure of the Customer object in order to save and restore it.

Types

- ***Types* are at the heart of the programming model for the CLR.**
  - Most of the **metadata** is organized by type.
- **A type is analogous to a class in most object-oriented programming languages, providing an abstraction of data and behavior, grouped together.**
- **A type in the CLR contains:**
  - Fields (data members)
  - Methods
  - Properties
  - Events (which are now full fledged members of the Microsoft programming paradigm).

NET Framework Class Library

- **The *SoapFormatter* and *FileStream* classes are two of the thousands of classes in the .NET Framework that provide system services.**
- **The functionality provided includes:**
  - Base Class Library (basic functionality such as strings, arrays and formatting).
  - Networking
  - Security
  - Remoting
  - Diagnostics
  - I/O
  - Database
  - XML
  - Web Services
  - Web programming
  - Windows User Interface
- **This framework is usable by all CLR compliant languages.**

Interface-Based Programming

- **Interfaces allow you to work with abstract types in a way that allows for extensible programming.**
- **The *SaveFile* and *RestoreFile* routines are written using the *IList* and *IFormatter* interfaces.**
- **These routines will work with all the collection classes that support the *IList* interface, and the formatters that support the *IFormatter* interface.**
- **Implementation inheritance permits code reuse.**
- **You can implement the *ISerializable* interface to override the framework's implementation.**
  - The metadata for the type tells the framework that the class has implemented the interface.
- **Interface-based programming allows classes to provide implementations of standard functionality that can be used by the framework.**

Everything is an Object

- **Every type in .NET derives from *System.Object*.**[1]
- **Every type, system or user defined, has metadata.**
  – In the sample the framework can walk through the ArrayList of Customer objects and save each one as well as the array itself.
- **All access to objects in .NET is through object references.**

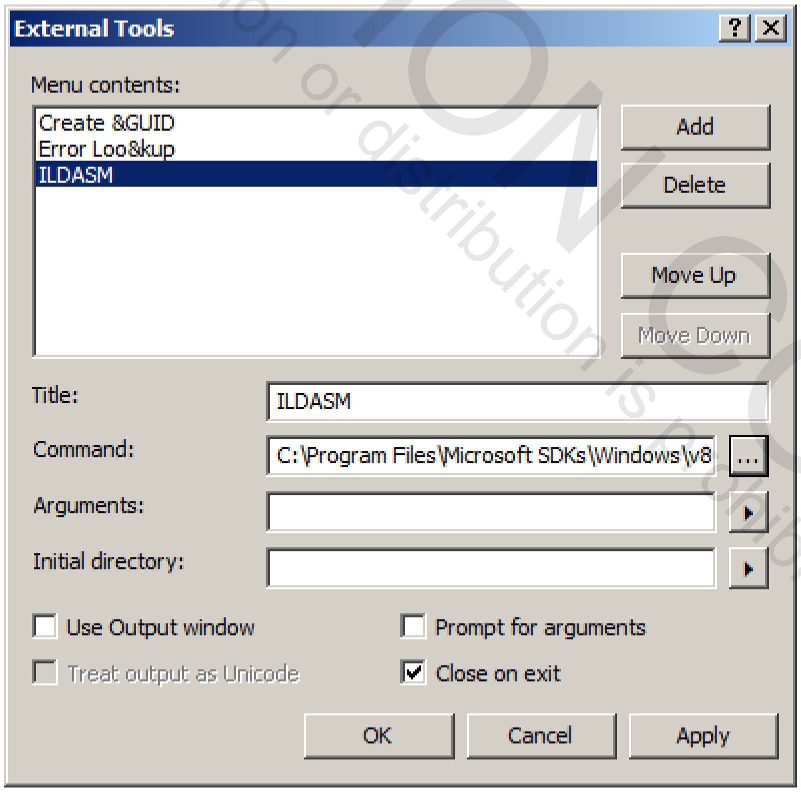
[1] An exception is the pointer type, which is rarely used in C#.

Common Type System

- **The Common Type System (CTS) defines the rules for the types and operations that the CLR will support.**
  - The CTS limits .NET classes to single implementation inheritance.
  - The CTS is designed for a wide range of languages, not all languages will support all features of the CTS.
- **The CTS makes it *possible* to guarantee type safety.**
  - Access to objects can be restricted to object references (no pointers), each reference refers to a defined memory. Access to that layout is only through public methods and fields.
  - By performing a local analysis of the class, you can verify to make sure that the code does not perform any inappropriate memory access. You do not have to analyze the users of the class.
- **.NET compilers emit Microsoft Intermediate Language (MSIL or IL) not native code.**
  - MSIL is platform independent.
  - Type-safe code can be restricted to a subset of verifiable MSIL expressions.
  - Once code is verified, it is verified for all platforms.

ILDASM

- **The Microsoft Intermediate Language Disassembler (ILDASM) can display the metadata and MSIL instructions associated with .NET code.**
  - It is a very useful tool both for debugging and increasing your understanding of the .NET infrastructure.
- **You may wish to add ILDASM to your Tools menu in Visual Studio 2013.**
  - Use the command Tools | External Tools. Click the Add button, enter ILDASM for the Title, and click the ... button to navigate to the folder \Program Files\Microsoft SDKs\Windows\v8.1A\bin\NETFX 4.5.1 Tools.



Menu contents: Create &GUID, Error Loo&kup, ILDASM

Title: ILDASM

Command: C:\Program Files\Microsoft SDKs\Windows\v8

Arguments:

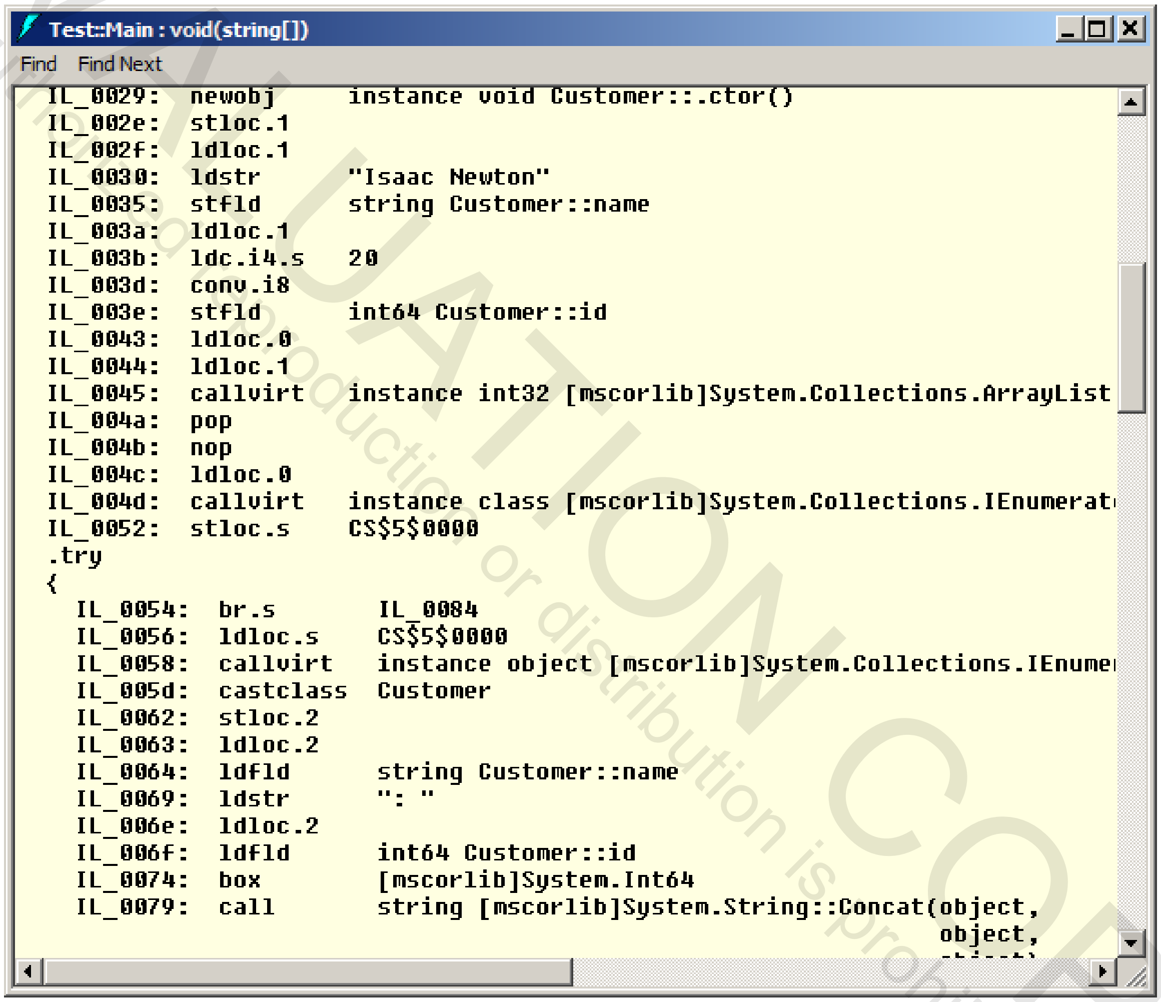
Initial directory:

☐ Use Output window ☐ Prompt for arguments

☐ Treat output as Unicode ☑ Close on exit

ILDASM (Cont'd)

- **You can use ILDASM to examine the .NET framework code.**
  – Here is a fragment of the MSIL from the **Serialize** example.

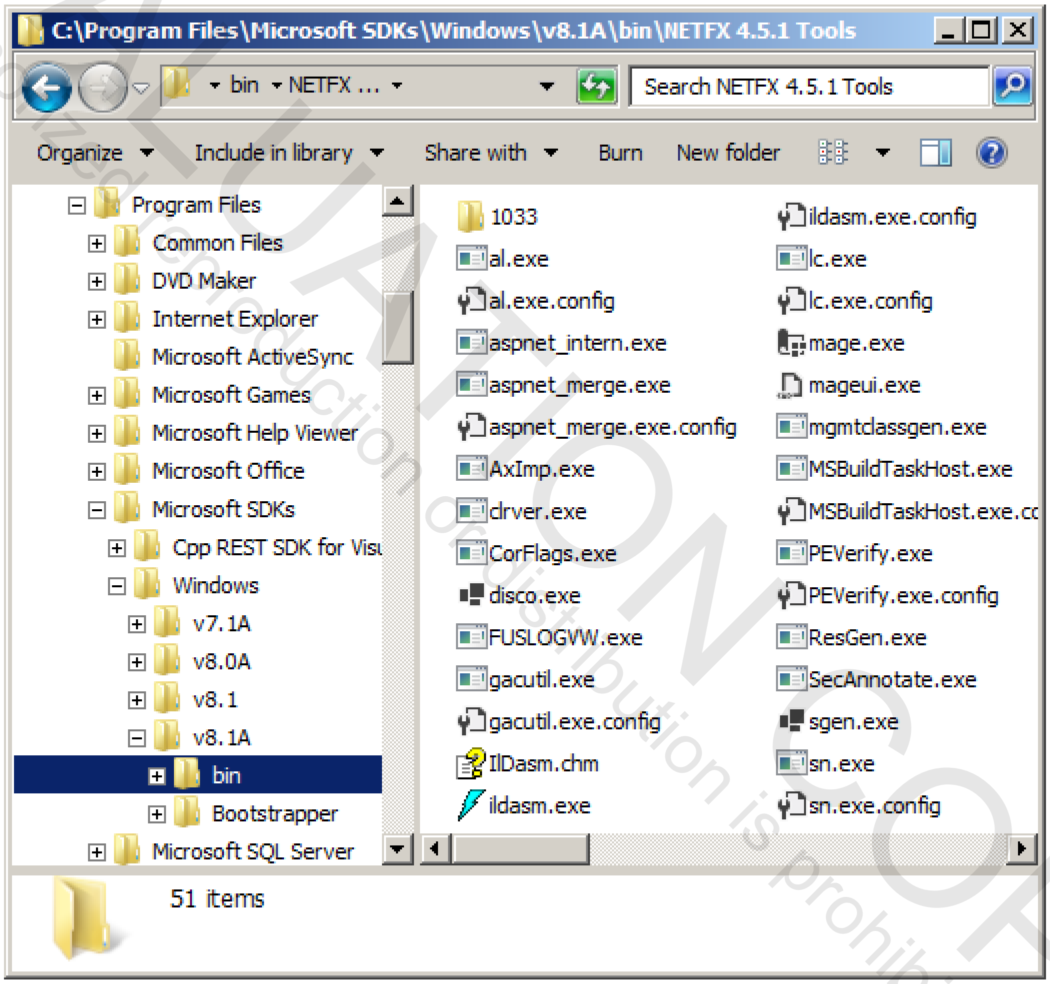

Test::Main : void(string[])

Find Find Next

```
IL_0029:  newobj     instance void Customer::.ctor()
IL_002e:  stloc.1
IL_002f:  ldloc.1
IL_0030:  ldstr      "Isaac Newton"
IL_0035:  stfld      string Customer::name
IL_003a:  ldloc.1
IL_003b:  ldc.i4.s   20
IL_003d:  conv.i8
IL_003e:  stfld      int64 Customer::id
IL_0043:  ldloc.0
IL_0044:  ldloc.1
IL_0045:  callvirt   instance int32 [mscorlib]System.Collections.ArrayList
IL_004a:  pop
IL_004b:  nop
IL_004c:  ldloc.0
IL_004d:  callvirt   instance class [mscorlib]System.Collections.IEnumerat
IL_0052:  stloc.s    CS$5$0000
.try
{
  IL_0054:  br.s       IL_0084
  IL_0056:  ldloc.s    CS$5$0000
  IL_0058:  callvirt   instance object [mscorlib]System.Collections.IEnume
  IL_005d:  castclass  Customer
  IL_0062:  stloc.2
  IL_0063:  ldloc.2
  IL_0064:  ldfld      string Customer::name
  IL_0069:  ldstr      ": "
  IL_006e:  ldloc.2
  IL_006f:  ldfld      int64 Customer::id
  IL_0074:  box        [mscorlib]System.Int64
  IL_0079:  call       string [mscorlib]System.String::Concat(object,
                                                               object,
```

.NET Framework SDK Tools

- **Installing Visual Studio 2013 will also install the .NET Framework SDK, version 8.1A.**
  - These tools are located in the folder \Program Files\Microsoft SDKs\Windows\v8.1A\bin\NETFX 4.5.1 Tools.



  - They can be run at the command line from the Visual Studio 2013 Command Prompt[2], which can be started from All Programs | Microsoft Visual Studio 2013 | Visual Studio Tools | Developer's Command Prompt for VS2013.

[2] You may need to run the Command Prompt as Administrator in some cases.

Language Interoperability

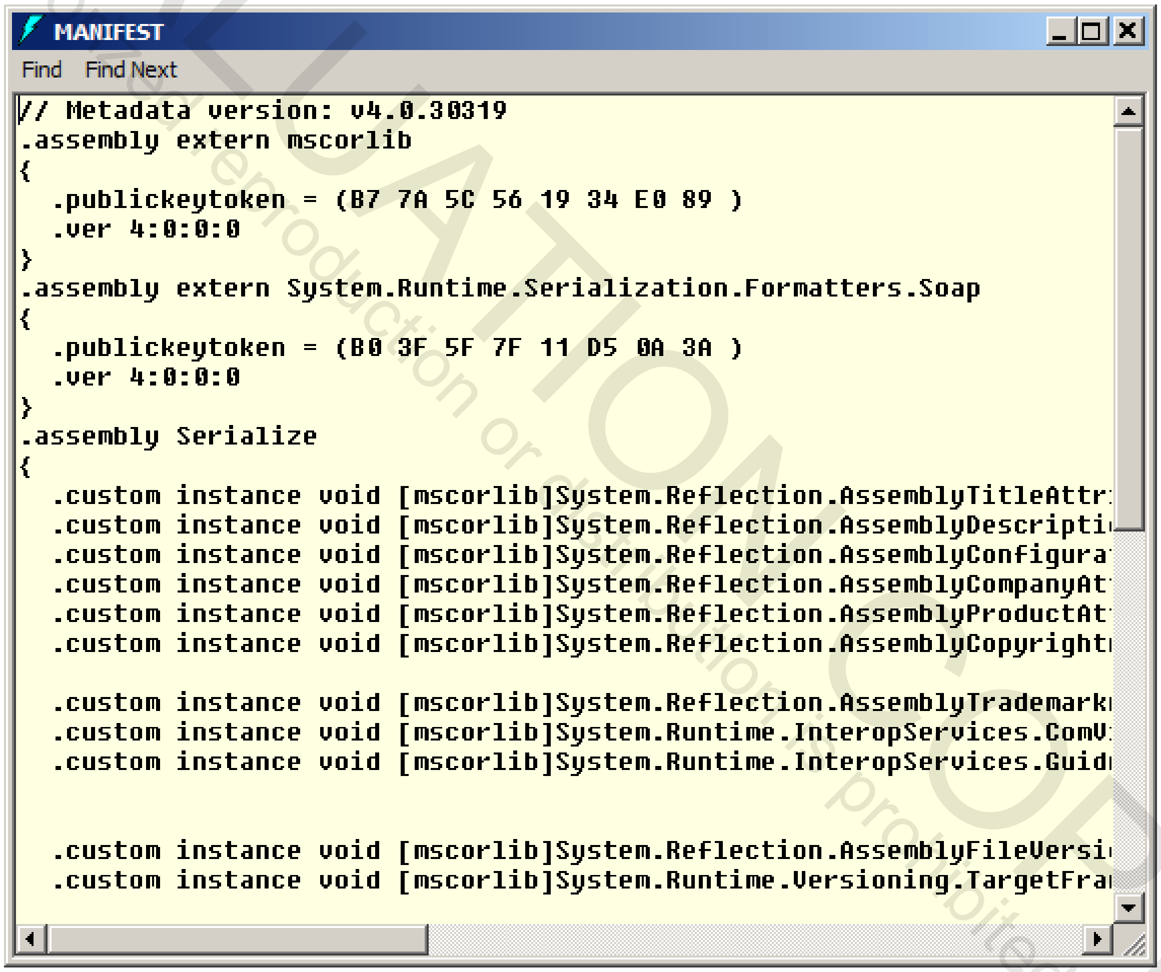
- **Having all language compilers use a common intermediate language and common base class makes it *possible* for languages to interoperate.**
  - All languages need not implement all parts of the CTS.
  - One language can have a feature that another does not.
- **The *Common Language Specification* (CLS) defines a subset of the CTS that represents the basic functionality that all .NET languages should implement if they are to interoperate with each other.**
  - For example, a class written in Visual Basic can inherit from a class written in C#.
  - Interlanguage debugging is possible.
  - CLS rule: Method calls need not support a variable number of arguments even though such a construct can be expressed in MSIL.
  - CLS prohibits the use of pointers.
- **CLS compliance only applies to public features.**
  - C# code should not define public and protected class names that differ only by case sensitivity since languages as Visual Basic are not case sensitive. Private C# fields could have such names.

Managed Code

- **In the serialization example we never freed any allocated memory.**
  - Memory that is no longer referenced can be reclaimed by the CLR's garbage collector.
  - Automatic memory management eliminates the common programming error of memory leaks.
  - Garbage collection is one of the services provided to .NET applications by the Common Language Runtime.
- **Managed code uses the services of the CLR.**
  - MSIL can express access to unmanaged data in legacy code.
- **Type-safe code cannot be subverted.**
  - For example, a buffer overwrite is not able to corrupt other data structures or programs. Security policy can be applied to type-safe code.
- **Type-safe code can be secured.**
  - Access to files or user interface features can be controlled.
  - You can prevent the execution of code from unknown sources.
  - You can prevent access to unmanaged code to prevent subversion of .NET security.
  - Paths of execution of .NET code to be isolated from one another.

Assemblies

- **.NET programs are deployed as an *assembly*.**
  - The metadata about the entire assembly is stored in the assembly's manifest.
  - An assembly has one or more EXEs or DLLs with associated metadata information.



```
// Metadata version: v4.0.30319
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 4:0:0:0
}
.assembly extern System.Runtime.Serialization.Formatters.Soap
{
  .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A )
  .ver 4:0:0:0
}
.assembly Serialize
{
  .custom instance void [mscorlib]System.Reflection.AssemblyTitleAttr
  .custom instance void [mscorlib]System.Reflection.AssemblyDescripti
  .custom instance void [mscorlib]System.Reflection.AssemblyConfigura
  .custom instance void [mscorlib]System.Reflection.AssemblyCompanyAt
  .custom instance void [mscorlib]System.Reflection.AssemblyProductAt
  .custom instance void [mscorlib]System.Reflection.AssemblyCopyright

  .custom instance void [mscorlib]System.Reflection.AssemblyTrademark
  .custom instance void [mscorlib]System.Runtime.InteropServices.ComV
  .custom instance void [mscorlib]System.Runtime.InteropServices.Guid


  .custom instance void [mscorlib]System.Reflection.AssemblyFileVersi
  .custom instance void [mscorlib]System.Runtime.Versioning.TargetFra
```

Assembly Deployment

- **The assemblies can be uniquely named.**
  - Assemblies can be versioned and the version is part of the assembly’s name.
  - Unique (strong) names use a public/private encryption scheme.
  - The culture used can also be made part of the assembly name.
- **Assemblies are self-describing. Information is in the metadata associated with the assembly, not in the System Registry.**
- **Private, or xcopy deployment requires only that all the assemblies an application needs are in the same directory.**
  - This makes deployment of components much simpler.
- **Public assemblies require a strong name and an entry in the Global Assembly Cache (GAC).**
- **Either approach means the end of DLL Hell!**
  - Components with different versions can be deployed side by side and need not interfere with each other.

JIT Compilation

- **Before executing on the target machine, MSIL is translated by a just-in-time (JIT) compiler to native code.**
- **Some code typically will never be executed during a program run.**
  - Hence it may be more efficient to translate MSIL as needed during execution, storing the native code for reuse.
- **When a type is loaded, the loader attaches a stub to each method of the type.**
  - On the first call the stub passes control to the JIT, which translates to native code and modifies the stub to save the address of the translated native code.
  - On subsequent calls to the method transfer is then made directly to the native code.
- **As part of JIT compilation code goes through a verification process.**
  - Type safety is verified, using both the MSIL and metadata.
  - Security restrictions are checked.

ASP.NET and Web Services

- **.NET includes a totally redone version of the popular Active Server Pages technology, known as *ASP.NET*.**
- **Whereas ASP relied on interpreted script code interspersed with page formatting commands, ASP.NET relies on *compiled* code.**
  - The code can be written in any .NET language, including C#, Visual Basic, JScript.NET and C++/CLI.
- **ASP.NET provides *Web Forms* which vastly simplifies creating Web user interfaces.**
  - Drag and drop in Visual Studio 2013 makes it very easy to lay out forms.
  - Also supported are ASP.NET MVC and Web API.
- **For application integration across the internet, *Web services* use the SOAP protocol.**
  - The beautiful thing about a Web service is that from the perspective of a programmer, a Web service is no different from any other kind of service implemented by a class in a .NET language.
  - Or you can use third-party web services which you did not know existed when you designed your application.
- **Web services and C# (or Visual Basic) as a scripting language allows Web programming to follow an object-oriented programming model.**

The Role of XML

- **XML is ubiquitous in .NET and is highly important in Microsoft's overall vision.**
- **Some uses of XML in .NET include:**
  - XML is used for encoding requests and responses in the SOAP protocol.
  - XML is the serialization format for disconnected datasets in ADO.NET.
  - XML is used extensively in configuration files.
  - XML documentation can be automatically generated by .NET languages.
  - .NET classes provide a very convenient API for XML programming as an alternative to DOM or SAX.

Performance

- **Concerns about performance of managed code are similar to the concerns assembly language programmers had with high level languages.**
- **Garbage collection usually produces faster allocation than C++ unmanaged heap allocation. Deallocation is done on a separate thread by the garbage collector.**
- **JIT compilation takes a hit the first time when verification and translation take place, but subsequent executions pay no penalty.**
- **There is a penalty when security checks have to be made that require a stack walk.**
- **Compiled ASP.NET code is going to be a lot faster than interpreted ASP pages.**
- **Bottom line: for most of the code that is written, any small loss in performance is far outweighed by the gains in reliability and ease of development.**
  - High performance servers might still have to use technologies such as ATL Server and C++.

Summary

- **.NET solves problems of past Windows development.**
- **One development paradigm for all languages exists.**
- **Design and programming language no longer conflict.**
- **Web services provide an API for applications across the Internet, typically using the SOAP protocol.**
- **SOAP supports a high degree of interoperability, since it is based on widely adopted standards such as HTTP and XML.**
- **.NET has many features which will create a much more robust Windows operating system.**
- **.NET uses managed code with services provided by the Common Language Runtime that uses the Common Type System.**
- **The .NET Framework is a very large class library available consistently across many languages.**
- **Deployment is more rational and includes a versioning strategy.**
- **Metadata, attribute-based security, code verification, and type-safe assembly isolation make developing secure applications much easier.**
- **Plumbing code for fundamental system services is there, yet you can extend it or replace it if necessary.**

Chapter 6

.NET Programming Model

.NET Programming Model

Objectives

After completing this unit you will be able to:

- **Describe garbage collection in .NET and implement the finalize/dispose pattern where appropriate.**
- **Describe the process and thread model for .NET applications, including thread isolation.**
- **Explain asynchronous programming and use asynchronous delegates to implement asynchronous calls.**
- **Use *async* and *await* keywords in C# 5.0 with the *Task* class to implement asynchronous programs.**
- **Describe how application domains support application isolation, and implement programs that use multiple application domains.**

Garbage Collection

- **Memory in managed applications is reclaimed through a garbage collection algorithm.**
  - While this prevents code from using already freed memory, or from failing to free memory, it makes the reclamation of non-memory resources harder.
  - For example, files or database connections that have to be closed, or server login connections that have to be disconnected.
- **The CLR tracks the use of memory that is allocated on the managed heap. Memory that is no longer referenced is marked as garbage.**
- **Classes that require non-memory resources to be freed must implement the *IDisposable* interface. *IDisposable* has one member, the *Dispose* method.**

```
public interface IDisposable
{
   void Dispose();
};
```

  - The **Dispose** method is designed to be called by a client program when it is done with an object, or knows that it is safe to free the resources associated with the object.
  - If some other name besides **Dispose** makes sense for the class (such as **Close** or **Cleanup**), add that method and have it call **Dispose**.
  - These classes should also implement a **Finalize** method.

Finalize Method

- ***Finalize*** **is called by the garbage collector when it frees the class' memory resources.**
- ***Finalize*** **is a protected member of the** ***System.Object*** **class, so it is only accessible within the class or a derived class.**
  - The default implementation does nothing.
  - A derived class should always call the base class' **Finalize** method.
- **Managed objects should not be referenced in** ***Finalize*****.**
  - The object's class might itself implement **Finalize**, which could get called first, leaving the object in an unpredictable state.

C# Destructor Notation

- **The C# language provides a special tilde notation *~SomeClass* to represent the overridden *Finalize* method and this special method is called a *destructor*.**
- **The C# destructor automatically calls the base class *Finalize*. Thus the following C# code**

```
~SomeClass()
{
   // perform cleanup
}
```

  - generates code that could be expressed

```
protected override void Finalize()
{
   // perform cleanup
   base.Finalize();
}
```

  - The second code fragment is actually not legal C# syntax, and you must use the destructor notation.

- **Although C# uses the same notation and terminology for destructor as C++, the two are very different.**
  - The C++ destructor is called deterministically when a C++ object goes out of scope or is deleted.
  - The C# destructor is called during the process of garbage collection, a process which is not deterministic.
  - It’s a good idea to minimize using the term “destructor” in C#!

Dispose

- **Since there are no guarantees exactly when the *Finalize* method will be called, resources that are scarce should be freed by the client through a *Dispose* method.**
- **A *Dispose* method should also call the base class *Dispose* as well to make sure that all its resources are freed.**
- **It should also be written so that if a *Dispose* method is called after the resources have been already freed, no exception is thrown.**
- **A *Finalize* method is implemented even if a *Dispose* method exists, just in case *Dispose* is not called.**
  - Since finalization is expensive, any objects that will no longer acquire any more resources should call the static method *GC.SuppressFinalize* method and pass it the *this* reference.
- **The classic case for a finalizer is a class that contains some unmanaged resource, such as a database connection. If they are not released when no longer needed, the scalability of your application can be affected.**

Finalize/Dispose Example

- **The example program *DisposeDemo* provides an illustration of finalization and the dispose pattern.**
  - The class **SimpleLog** implements logging to a file, making use of the **StreamWriter** class.

```
using System;
using System.IO;

public class SimpleLog : IDisposable
{
   private StreamWriter writer;
   private string name;
   private bool disposeCalled = false;
   public SimpleLog(string fileName)
   {
      name = fileName;
      writer = new StreamWriter(fileName, false);
      writer.AutoFlush = true;
      Console.WriteLine("logfile " + name + "
         created");
   }
   public void WriteLine(string str)
   {
      writer.WriteLine(str);
      Console.WriteLine(str);
   }
   ...
```

Finalize/Dispose Example (Cont'd)

```
      ...
      public void Dispose()
      {
         if(disposeCalled)
            return;
         // During finalization you should avoid
         // accessing other managed objects. To see a
         // problem, uncomment next line
         // writer.Close();
         GC.SuppressFinalize(this);
         Console.WriteLine("logfile " + name +
            " disposed");
         disposeCalled = true;
      }
      ~SimpleLog()
      {
         Console.WriteLine("logfile " + name +
            " finalized");
         Dispose();
      }
   }
```

- **The class *SimpleLog* supports the *IDisposable* interface, and thus implements *Dispose*.**
  - The **Dispose** method displays a message at the console. To make sure that a disposed object will not also be finalized, **GC.SuppressFinalize** is called.
  - The finalizer simply delegates to **Dispose**. To help monitor object lifetime, a message is written to the console in the constructor and in the finalizer, as well as in **Dispose**.

Finalize/Dispose Test Program

- **Here is the code for the test program:**

```
using System;
using System.Threading;

public class DisposeDemo
{
   public static void Main()
   {
      SimpleLog log = new SimpleLog(@"log1.txt");
      log.WriteLine("First line");
      Pause();
      log.Dispose();
      log.Dispose();
      log = new SimpleLog(@"log2.txt");
      log.WriteLine("Second line");
      Pause();
      log = new SimpleLog(@"log3.txt");
      log.WriteLine("Third line");
      Pause();
      log = null;
      GC.Collect();
      Thread.Sleep(100);
   }
   private static void Pause()
   {
      Console.Write("Press enter to continue");
      string str = Console.ReadLine();
   }
}
```

Test Program (Cont'd)

- **The *SimpleLog* object reference *log* is assigned in turn to three different object instances.**
  - The first time, it is properly disposed. The second time, log is reassigned to refer to a third object before the second object is disposed, resulting in the second object becoming garbage. The **Pause** method provides an easy way to pause the execution of this console application, allowing us to investigate the condition of the files **log1.txt**, **log2.txt**, and **log3.txt** at various points in the execution of the program.
- **Running the program results in the following output:**

```
logfile log1.txt created
First line
Press enter to continue
logfile log1.txt disposed
logfile log2.txt created
Second line
Press enter to continue
logfile log3.txt created
Third line
Press enter to continue
logfile log3.txt finalized
logfile log3.txt disposed
logfile log2.txt finalized
logfile log2.txt disposed
```

Garbage Collection Performance

- **Allocation is *very* fast. Space on the managed heap is always contiguous, so allocating a new object is equivalent to incrementing a pointer.**
  - Allocation on an unmanaged heap is relatively slow, because a list of data structures must be walked to find a block that is large enough.
  - The CLR uses *generations* during garbage collecting, reducing the number of objects that are typically checked for being garbage.
- **Some empirical experience suggests that performance may be very fast, but managed code may use significantly more memory.**
  - For a fascinating report on one experiment, see ".NET Versus COM" by Robert Gunion, *Dr. Dobbs Journal*, October, 2002.
- **.NET 4.5 introduces background server garbage collection, which can improve performance for servers.**
  - For an in-depth discussion of .NET memory management, see the article "Memory Management and Garbage Collection in the .NET Framework" on MSDN:

http://msdn.microsoft.com/en-us/library/vstudio/hh156531.aspx

Generations

- **As an optimization, every object on the managed heap is assigned to a generation.**
  - A new object is in generation 0 and is considered a prime candidate for garbage collection.
  - Older objects are in generation 1. Since such an older object has survived for a while, the odds favor its having a longer lifetime than a generation 0 object.
  - Still older objects are assigned to generation 2 and are considered even more likely to survive a garbage collection. The maximum generation number in the current implementation of .NET can be found from the **GC.MaxGeneration** property.
- **In a normal sweep of the garbage collector, only generation 0 will be examined. It is here that the most likely candidates are for memory to be reclaimed.**
- **All surviving generation 0 objects are promoted to generation 1. If not enough memory is reclaimed, a sweep will next be performed on generation 1 objects, and the survivors will be promoted.**
- **Then, if necessary, a sweep of generation 2 will be performed, and so on up until *MaxGeneration*.**

Processes

- **A process is the environment in which a program executes.**
  - Part of this environment is the address space in which the code and data of the program reside.
  - A process also has a set of environmental variables that is associated with the program.
  - A process has a current drive and directory.
  - A process has one or more threads. A thread is what actually executes the program’s code.
- **Traditionally, processes are used to provide application isolation, so that a fault in one process will not corrupt another process.**
- **In .NET application isolation can be achieved by a lighter weight entity called an *application domain*.**
  - We will discuss application domains later in the chapter.

Threads

- **Operating systems use processes to separate the different applications that they are executing. Threads run inside of processes to allow for multiple execution paths inside of a process.**
- **Threads are what are scheduled by the operating system, not processes or application domains.**
- **The .NET Framework provides extensive support for multiple thread programming in the *System.Threading* namespace.**
  - The core class is **Thread**, which encapsulates a thread of execution.
  - The **Thread** object that represents the current executing thread can be found from the static property **Thread.CurrentThread**.
- **The use of multiple threads can enable greater responsiveness to shorter tasks, especially those requiring user responses.**
- **With the advent of multiple-core CPUs, the use of multiple threads can speed up computations through parallel programming.**
- **Threads will be discussed in some detail in the next chapter.**

Asynchronous Calls

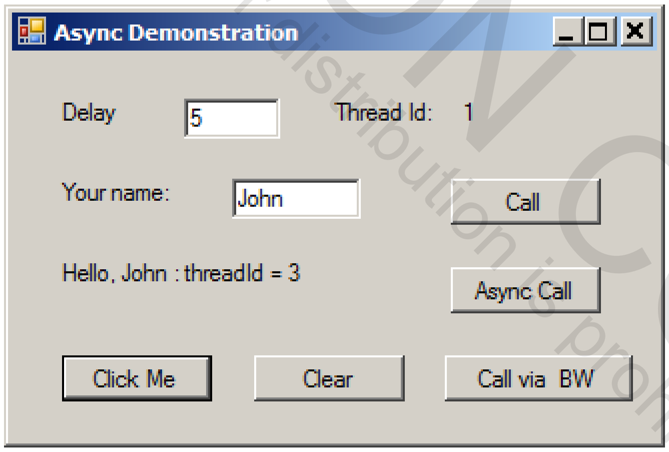
- **Normal method calls are *synchronous*, which means that the calling thread cannot continue until the called method returns.**
- **In an *asynchronous* call, the called method runs on a different thread, and the method call returns immediately.**
  - The calling program continues executing until either a specified callback is made or the calling program polls or waits for completion.
- **Asynchronous programming is supported in many areas of the .NET Framework, including :**
  - File I/O
  - Networking
  - Remoting
  - Web services
  - Asynchronous delegates

Asynchronous Delegates

- **Threads in .NET enable a powerful mechanism of *asynchronous delegates*, which enable you to call any target method asynchronously.**
  - You must define a delegate which has the same signature as the target method.
  - The common language runtime will synthesize **BeginInvoke** and **EndInvoke** methods for this delegate, having suitable signatures.
  - Call **BeginInvoke** to start the asynchronous call. The target method will start up on a new thread, and **BeginInvoke** will return immediately.
  - Call **EndInvoke** to obtain the results of the asynchronous call.
- ***BeginInvoke* returns an *IAsyncResult* interface reference that can be used to track the results of the asynchronous invocation. *IAsyncResult* has the following public properties:**
  - AsyncState
  - AsyncWaitHandle
  - CompletedSynchronously
  - IsCompleted

Using a CallBack Method

- **A useful pattern for using an asynchronous delegate is to provide a callback method.**
  - The calling thread continues after **BeginInvoke**.
  - The callback is invoked when the target method completes. However, any method called by an external thread is not guaranteed to by thread safe, and so .NET 2.0 and later block accessing any Windows controls in the callback method.
  - A pattern for safe calls to Windows controls is also presented in our example program.
- **Use of a callback method is illustrated in an example program.**
  - See **AsyncGui\Step2**.



Hello, John : threadId = 3

  - The target method GetGreeting returns a greeting string personalized by your name. The thread ID under which it runs is also returned. The target method sleeps for the number of seconds specified in the Delay box.

Using a CallBack Method (Cont’d)

- A second method SetText assigns text to a control in a thread-safe manner. It uses a second delegate call if necessary to assure thread-safe access to Windows controls.
- The synchronous call blocks, and the Click Me button will not be responsive. The target method is running under the same thread.
- The asynchronous call returns immediately, and Click Me responds. The target method is running under a different thread.

- **There are four main pieces to the callback method pattern.**
  - Define a delegate with the same signature as the target method.

```
private delegate string GreetingDelegate(
   string name, int delay, out int threadId);
```

  - Create the delegate object and call **BeginInvoke**. Note the two extra parameters.

```
GreetingDelegate delg =
   new GreetingDelegate(GetGreeting);

int threadId;
int delay = Convert.ToInt32(txtDelay.Text);
IAsyncResult ar = delg.BeginInvoke(txtName.Text,
   delay,
   out threadId,
   new AsyncCallback(CallbackMethod),
   delg);
```

Using a CallBack Method (Cont’d)

- In the callback method, retrieve the delegate from **IAsyncResult** and obtain the results by calling **EndInvoke**, which includes the **out** and **ref** parameters of the target method.

```
private void CallbackMethod(IAsyncResult ar)
{
   GreetingDelegate delg =
      (GreetingDelegate) ar.AsyncState;
   int threadId;
   string greeting =
      delg.EndInvoke(out threadId, ar);

   greeting = greeting + " : "
      + "threadId = " + threadId;
   SetText(greeting, lblGreeting);
}
```

- Take steps to assign results to controls in a thread-safe manner. The method **SetText** uses a control property **InvokeRequired** to determine whether this can be performed directly or if the callback method must be used.

```
//Thread-safe method for calling Windows controls
private void SetText(string MyText, Control ctl)
{
   if (ctl.InvokeRequired)
   {
      SetTextCallback d =
         new SetTextCallback(SetText);
      this.Invoke(d, new object[] {MyText, ctl});
   }
   else
   {
      ctl.Text = MyText;
   }
}
```

BackgroundWorker

- **The *BackgroundWorker* control encapsulates the asynchronous process discussed above.**
- **Placing the control on a form allows immediate use without creating delegates and callback methods.**
  - Method used is **RunWorkerAsync.** Events called **DoWork** and **RunWorkerCompleted** provide easy control and access to results.
  - **DoWork** replaces the delegate and the callback function. An event argument, *Result*, allows passing of results.
  - **RunWorkerCompleted** allows access to the results of **DoWork** through the event argument *Result.*

```
private void cmdSafeAsync_Click(object sender,
System.EventArgs e)
{
   BackgroundWorker1.RunWorkerAsync();
}
private void BackgroundWorker1_DoWork(object sender,
System.ComponentModel.DoWorkEventArgs e)
{
   int threadId = 0;
   int delay = Convert.ToInt32(txtDelay.Text);
   string answer = GetGreeting(txtName.Text, delay,
      out threadId);
   answer = answer + " : " + "threadId = " + threadId;
   e.Result = answer;
}
private void BackgroundWorker1_RunWorkerCompleted(
object sender,
System.ComponentModel.RunWorkerCompletedEventArgs e)
{
   lblGreeting.Text = e.Result.ToString();
}
```

Asynchronous Programs in C# 5.0

- **In situations where an activity may be blocked (such as file I/O), application performance and responsiveness may be improved with asynchronous programming.**
- **.NET 4.5 introduces new C# keywords *async* and *await* to simplify asynchronous programming.**
  - The result is a *much* simpler asynchronous programming model than provided with previous approaches.
- **The essence of the new model is extremely simple:**
  - Apply the **async** keyword to a method that is to be called asynchronously. Such a method is referred to as an “async method.” As a naming convention, the method name should end in “Async”.
  - When you call an async method, use the **await** keyword. The call will not complete until the asynchronous operation has completed.

Task and Task<TResult>

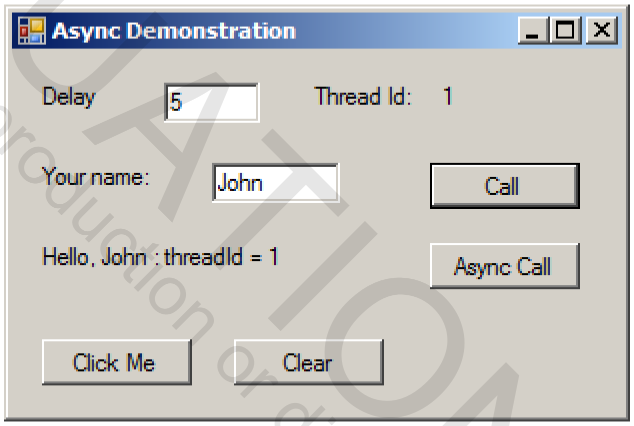
- **The *Task* class represents an asynchronous operation.**
- **The *Task<TResult>* class represents an asynchronous operation that returns a result.**
- **The *Task* class has several static *Delay()* methods.**
  - **Task.Delay(Int32)** creates a task that will complete after a time delay specified in milliseconds.
  - **Task.Delay(TimeSpan)** creates a task that will complete after a time delay specified as a **TimeSpan**.
- **The *Task* class is the fundamental class in the Task Parallel Library (TPL), which was introduced with .NET 4.**
  - We will cover the **Task** class and TPL in more detail in the next chapter.

Aysnc Methods

- **An async method has these characteristics:**
  - The method signature has an **async** modifier.
  - By convention, the name of the method ends with “Async”.
  - The return type is **Task<TResult>** if the method returns a value of type **TResult**.
  - The return type is **Task** if the method does not return a value.
  - The return type is void for an async event handler.
  - The method usually contains at least one **await** expression. This marks a point where the method is suspended until a pending asynchronous operation completes.
- **In .NET 4.5 many .NET Framework classes contain async methods.**

New Async Example

- **See *NewAsyncGui\Step2* in the chapter folder.**
  - You can make either a synchronous or an asynchronous call.
  - Both calls are delayed for the number of seconds specified by the user.
  - The synchronous call is blocking, and so “Click Me” is not responsive. The async call is responsive.



Hello, John : threadId = 1

- **Try it out!**

Synchronous Call

```
private void cmdCall_Click(object sender, EventArgs
e)
{
   int threadId;
   int delay = Convert.ToInt32(txtDelay.Text);
   string greeting = GetGreeting(txtName.Text,
      delay,
      out threadId);
   greeting = greeting + " : " + "threadId = " +
      threadId;
   lblGreeting.Text = greeting;
}

private string GetGreeting(string name, int delay,
out int threadId)
{
   Thread.Sleep(delay * 1000);
   Thread t = Thread.CurrentThread;
   threadId = t.ManagedThreadId;
   return "Hello, " + name;
}
```

- **Initialization of thread info is done when the form is loaded.**

```
private void Form1_Load(object sender, EventArgs e)
{
   Thread t = Thread.CurrentThread;
   int threadId = t.ManagedThreadId;
   lblThreadId.Text = threadId.ToString();
}
```

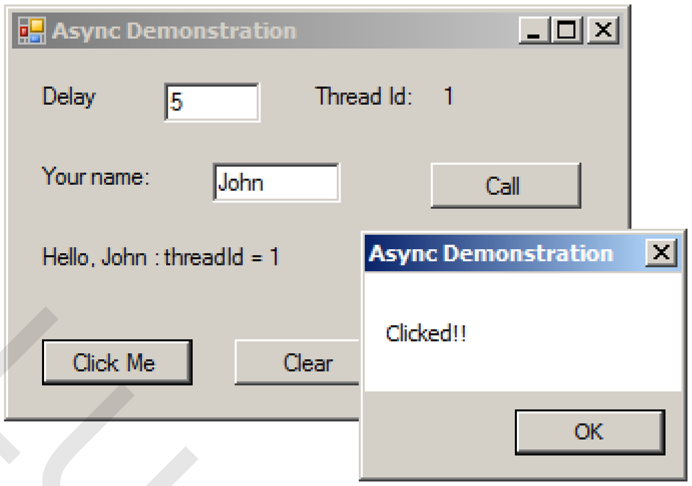
Async Call

```
private async Task<string> GetGreetingAsync(string
name, int delay)
{
   Thread t = Thread.CurrentThread;
   int threadId = t.ManagedThreadId;
   await Task.Delay(delay * 1000);
   return "Hello, " + name + " : " + "threadId = "
      + threadId;
}

private async void cmdAsync_Click(object sender,
EventArgs e)
{
   int delay = Convert.ToInt32(txtDelay.Text);
   lblGreeting.Text =
      await GetGreetingAsync(txtName.Text, delay);
}
```

- **Notice how simple this code is!**

Threading



- **The *async* and *await* keywords do not result in additional threads being created.**
  - An async method does not run on its own thread.
  - The method runs on the current synchronization context. It uses time on the thread only when the method is active.
- **This approach to asynchronous programming using async methods is almost always preferable to older techniques, and is *much* simpler.**

Lab 6A

Using an Asynchronous Delegate

In this lab you will take an existing program that has a target method called synchronously and implement an asynchronous call via the callback pattern.

Detailed instructions are contained in the Lab 6A write-up at the end of the chapter.

Suggested time: 30 minutes

Lab 6B

Using *async* and *await* Keywords

In this lab you will take an existing program that has a target method called synchronously and implement an asynchronous call using the **Task** class and the new keywords in C# 5.0.

Detailed instructions are contained in the Lab 6B write-up at the end of the chapter.

Suggested time: 20 minutes

Application Isolation

- **When writing applications it is often necessary to isolate parts of the applications so that a failure of one part does not cause a failure in another part of the application.**
- **In Windows, application isolation has been at the process level.**
  - In other words, if a process is stopped or crashes, other processes will be unaffected.
  - One process cannot directly address memory in another process' address space.
- **For one application to use separate processes to achieve isolation is expensive. To switch from one process to another the process state must be saved.**
- **Process switch overhead includes:**
  - Thread switch (saving call stack, registers such as the instruction pointer
  - Loading the information for a new thread
  - Updating the scheduling information for the threads
  - Any process state that must be saved and loaded (such as accounting information and processor rights).

Application Domain

- **The .NET *application domain* is a lightweight unit for application isolation, fault tolerance, and security.**
- **Multiple app domains can run in one process. Since the CLR checks code to be type-safe and verifies security, app domains can run independently of each other.**
  - No process switch is required to achieve application isolation.
- **A thread runs in one app domain at a time.**
  - Each app domain starts with a single thread. Additional threads can be added as needed.
  - There is no relationship between the number of app domains and threads. A Web server might require an app domain for each hosted application that runs in its process. The number of threads in that process would be far fewer depending on how much actual concurrency the process can support.
  - A new app domain runs on the thread that created it.
- **Code in one application domain cannot make direct calls into the code (or even reference resources) in another application domain. They must use *proxies*.**
  - Proxies are also used in calling to another process or even another computer, using .NET Remoting.
  - Proxies and .NET Remoting are discussed in Appendix A.

Application Domains and Assemblies

- **Applications are built from one more assemblies.**
- **Each application domain can be unloaded independent of the others.**
  - You cannot unload an individual assembly from an app domain.
  - Unloading an app domain also frees all resources associated with that app domain.
- **By default if an assembly is loaded into several app domains in a process, each app domain will get a separate copy.**
- **Each process has a default application domain that is created when the process is started.**
  - This default domain can only be unloaded when the process shuts down.
- **Hosts of the Common Language Runtime such as ASP.NET or Internet Explorer critically depend on this functionality.**
- **While you may never write code with application domains, understanding them is critical to understanding how .NET programs execute.**

AppDomain

- **Application domains are represented by a class *AppDomain*.**
- **This class has static methods for creating and unloading application domains:**

```
AppDomain domain2  = AppDomain.CreateDomain(
   "Domain2", null, null);
...
AppDomain.Unload(domain2);
```

- **The sample program *AppDomainDemo* illustrates working with app domains.**

CreateDomain

- **While the *CreateDomain* method is overloaded, one signature illustrates application domain isolation:**

```
public static AppDomain CreateDomain(
   string friendlyName,
   Evidence securityInfo,
   AppDomainSetup info
);
```

- **The *Evidence* parameter is a collection to the security constraints on the application domain.**
  - The domain's creator can modify this collection to control the permissions that the executing app domain can have.
- **The *AppDomainSetup* parameter specifies setup information about the domain.**
  - Among the information specified is the location of the app domain's configuration file and where private assemblies are loaded.
  - Each app domain can be configured independently of each other.
- **Code isolation, setup isolation, and control over security combine to ensure application domains are independent of each other.**
- **In the Security chapter we will illustrate another overload of *CreateDomain()*, which is used to create a sandbox with restricted permissions.**

App Domain Events

- **To help in maintaining isolation, the *AppDomain* class allows you to setup event handlers**
  - when an assembly loads
  - when the domain unloads
  - when an unhandled exception occurs
  - when attempts to resolve assemblies, types and resources fail.

```
...
// Add event handlers
domain2.AssemblyLoad += new
   AssemblyLoadEventHandler(LoadEventHandler);
domain2.DomainUnload += new
   EventHandler (DomainUnloadHandler);
...

public static void  LoadEventHandler(object sender,
   AssemblyLoadEventArgs args)
{
   Console.WriteLine("ASSEMBLY LOADED: " +
      args.LoadedAssembly.FullName);
   Console.WriteLine();
}

public static void DomainUnloadHandler(
   object sender, EventArgs args)
{
   Console.WriteLine("DOMAIN UNLOADED");
   Console.WriteLine();
}
```

Lab 6C

Working with App Domains

In this lab you will incrementally create a demonstration program that illustrates a number of features in working with application domains. The exercise also reviews some important concepts concerning program startup, command-line arguments, assemblies and reflection.

Detailed instructions are contained in the Lab 6C write-up at the end of the chapter.

Suggested time: 60 minutes

Summary

- **The CLR provides automatic garbage collection, but you can implement the finalize/dispose pattern for greater control over how memory is managed in your application.**
- **.NET has a sophisticated process and thread model whose features include thread isolation and thread synchronization.**
- **Asynchronous delegates enable you to call any target method asynchronously.**
- **You can use *async* and *await* keywords with the *Task* class to implement asynchronous programs in C# 5.0.**
- **Application domains provide a lightweight means of supporting application isolation.**

Lab 6A

Using an Asynchronous Delegate

Introduction

In this lab you will take an existing program that has a target method called synchronously and implement an asynchronous call via the callback pattern.

Suggested Time: 30 minutes

Root Directory: **OIC\NetCs**

Directories:	**Labs\Lab6A\AsyncGui**	(do your work here)
	Chap06\AsyncGui\Step1	(backup of starter code)
	Chap06\AsyncGui\Step2	(answer)

Instructions

1. Build the starter project and observe the working of the Call button, which makes a synchronous call. If you click “Click Me,” nothing will happen (but the request will be queued, and when the call completes, you will see both the results of the call displayed and a message box pop up!). Study the code.

2. Declare a delegate **GreetingDelegate** in Form1 that has the same signature as the **GetGreeting** method, which we will use as our callback.

3. Implement the handler for the Async Call button. You will need to instantiate a **GreetingDelegate** object and then make the call to **BeginInvoke**.

4. Define a method **CallbackMethod** that has the same signature as the **AsyncCallback** delegate that is provided by the .NET Framework.

5. Retrieve the **GreetingDelegate** from the **IAsyncResult** that is passed as a parameter to **CallbackMethod**.

6. Call **EndInvoke**. Note that the method synthesized by .NET takes the **out** parameter from the target method as well as the **IAsyncResult**. Use the results to set the label control with the greeting string and the thread ID used in the called method.

7. Build and run. Try out both synchronous and asynchronous calls. Notice that for synchronous calls the same thread is used in the called method, and a different thread is used in the asynchronous calls. Try the “Click Me” button in each case.

8. Optional: Add a BackgroundWorker control and an additional button to call the **RunWorkerAsync** method and display the result. Implement the **DoWork** and **RunWorkerCompleted** event handlers to call **GetGreeting.** Build and run.

Lab 6B

Using *async* and *await* Keywords

Introduction

In this lab you will take an existing program that has a target method called synchronously and implement an asynchronous call using the **Task** class and the new keywords in C# 5.0.

Suggested Time: 20 minutes

Root Directory: **OIC\NetCs**

Directories:	**Labs\Lab6B\NewAsyncGui**	(do your work here)
	Chap06\NewAsyncGui\Step1	(backup of starter code)
	Chap06\NewAsyncGui\Step2	(answer)

Instructions

1. Build the starter project and observe the working of the Call button, which makes a synchronous call. If you click "Click Me," nothing will happen (but the request will be queued, and when the call completes, you will see both the results of the call displayed and a message box pop up!). Review the code.

2. Create a helper method **GetGreetingAsync()** which returns a **Task<string>** and has the **async** keyword. Input parameters are a string for a name and an integer for the number of milliseconds to delay.

```
private async Task<string> GetGreetingAsync(string name, int delay)
{
}
```

3. Obtain an integer thread ID in the same manner as in Form1_Load().

```
private async Task<string> GetGreetingAsync(string name, int delay)
{
    Thread t = Thread.CurrentThread;
    int threadId = t.ManagedThreadId;
}
```

4. Use the **await** keyword and the **Delay()** method of the **Task** class to delay for the specified interval.

```
private async Task<string> GetGreetingAsync(string name, int delay)
{
    Thread t = Thread.CurrentThread;
    int threadId = t.ManagedThreadId;
    await Task.Delay(delay * 1000);
}
```

5. Return a string with a greeting and the thread ID.

```
private async Task<string> GetGreetingAsync(string name, int delay)
{
    Thread t = Thread.CurrentThread;
    int threadId = t.ManagedThreadId;
    await Task.Delay(delay * 1000);
    return "Hello, " + name + " : " + "threadId = " + threadId;
}
```

6. Add a handler for the Async Call button.

```
private void cmdAsync_Click(object sender, EventArgs e)
{
}
```

7. Obtain **delay** from the textbox. Set the label to the result of calling **GetGreetingAsync()** using the **await** keyword.

```
private void cmdAsync_Click(object sender, EventArgs e)
{
    int delay = Convert.ToInt32(txtDelay.Text);
    lblGreeting.Text = await GetGreetingAsync(txtName.Text, delay);
}
```

8. Build the project. You will get a compile error.

```
'await' can only be used when contained within a method or lambda
expression marked with the 'async' modifier.
```

9. Mark the handler with the **async** keyword. Now you should get a clean compile.

```
private async void cmdAsync_Click(object sender, EventArgs e)
{
    int delay = Convert.ToInt32(txtDelay.Text);
    lblGreeting.Text = await GetGreetingAsync(txtName.Text, delay);
}
```

10. Build and run. Try out both synchronous and asynchronous calls. Notice that the same thread is used in the called method for both the synchronous and asynchronous calls. Try the “Click Me” button in each case. The coding is *much* simpler than in Lab 6A!

Lab 6C

Working with App Domains

Introduction

In this lab you will incrementally create a demonstration program that illustrates a number of features in working with application domains. The exercise also reviews some important concepts concerning program startup, command-line arguments, assemblies and reflection.

Suggested Time: 60 minutes

Root Directory: **OIC\NetCs**

Directories:	**Labs\Lab6C**	(do your work here)
	Chap06\AppDomainDemo	(answer)

Step 1. Launch an Assembly from the Default Domain

In this first step you will create the demonstration console program and also a test console application. You will run the test app both independently and also launched from the default app domain of the main program.

1. In the **Lab6C** directory create a new console project **AppDomainDemo**. To make the projects a little cleaner, you may wish to take out the default namespace created by Visual Studio.

2. Add code to display the friendly name of the default domain.

3. Add a second console project **TestApp** to your solution, in a directory **TestApp** just below **AppDomainDemo**. To make it easy to find the executables, you might want to configure them all to be built in the **AppDomainDemo** source directory.

4. Add code to display the friendly name of the domain the test app is running in. Also, display the command-line arguments, and return the value 100.

5. Temporarily make **TestApp** the startup project of the solution. Build and test. You may specify command line arguments from the Properties for the project. See Debugging under Configuration Properties.

6. Add code to **AppDomainDemo** to launch **TestApp** in the default domain by calling the **ExecuteAssembly** method of the **AppDomain** class. Use the overloaded version which passes no arguments. Print out the return value.

7. Restore the startup project to be **AppDomainDemo**. Build and test.

Step 2. Launch an Assembly from a Second Domain

In the second step you will create a second app domain and launch an assembly from this second app domain. You will pass arguments.

1. Add code to **AppDomainDemo** to create a second domain called “Domain2”. Display the friendly name of this domain.

2. We are going to execute the **TestApp** assembly from this second domain, but this time we will pass some arguments. To this end declare an array of **string**, initialized with “one” and “two”.

3. Add code to **AppDomainDemo** to launch **TestApp** in the second domain by calling the **ExecuteAssembly** method of the **AppDomain** class. Use the overloaded version which passes an array of arguments. Print out the return value.

4. Modify **TestApp** to return 100 if no arguments were passed and 200 otherwise. Build and test.

Step 3. Load a Class Library and Call Its Methods

In the third step you will add a third project to the solution, a class library **AddLib** that implements a class **AddLib**. In **TestApp** you will load this class library and call its methods (both an instance method and a static method). You will create an instance of **AddLib** using both the default constructor and a non-default constructor.

1. Add a third project **AddLib**, a class library, to your solution, in a directory **AddLib** just below **AppDomainDemo**. In order to make it easy for **TestApp** to locate the DLL, set the Output Path to be the **AppDomainDemo** directory, where the other assemblies are also created.

2. Adjust the Project Dependencies of the solution so that the assemblies build in the order: **AddLib**, **TestApp**, **AppDomainDemo**.

3. Change the name of the **Class1.cs** file to **AddLib.cs** and the class to **AddLib**.

4. Implement the following features in the **AddLib** class.

    a. A private member variable **m_number** that holds an integer.

    b. A read-only property **Number** that exposes this integer.

    c. A default constructor that initializes the encapsulated number to 50.

    d. A constructor that initializes the encapsulated number to an integer passed as a parameter.

   e. An instance function that that adds an integer passed as a parameter to the encapsulated number and returns the sum.

   f. A static function that takes two integer parameters and returns the sum.

5. Add code to **TestApp** to load the **AddLib.dll** assembly and call its methods. Don't forget to add a reference to **AddLib.dll**. (You could use more Reflection code to make the call, as discussed in Chapter 4. The reference makes it easier.) Illustrate creating instances of **AddLib** using each constructor and call the instance method for each object. Also call the static method. Print out the results of the additions.

6. Build and test.

Step 4. Handling Events from an App Domain

In the final step you will add code to **AppDomainDemo** to handle events associated with loading an assembly and unloading an application domain.

1. Add an event handler procedure **LoadEventHandler** that will print a message when an assembly is loaded. You should also display the name of the assembly.

2. Add code to hook this handler to the **AssemblyLoad** event of the second app domain.

3. Build and test.

4. Add an event handler **DomainUnloadHandler** that will display a message when an app domain is unloaded. Hook this event to the **DomainUnload** event of the second app domain.

5. Build and test. Why don't you see a message announcing the domain has been unloaded?

6. Add a call to the **Unload** method of **AppDomain**, and run again. This time you should see the message.