.NET Framework
Using C#

Student Guide

Revision 4.0

Object Innovations Course 4112

.NET Framework Using C#
Rev. 4.0

Student Guide

® is a registered trademark of Object Innovations.

Author: Robert J. Oberg

Special Thanks: Michael Stiefel, Peter Thorsteinson, Dana Wyatt, Paul Nahay, Ed Strassberger, Robert Seitz, Sharman Staples

Object Innovations
877-558-7246
www.objectinnovations.com

Published in the United States of America.

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.**

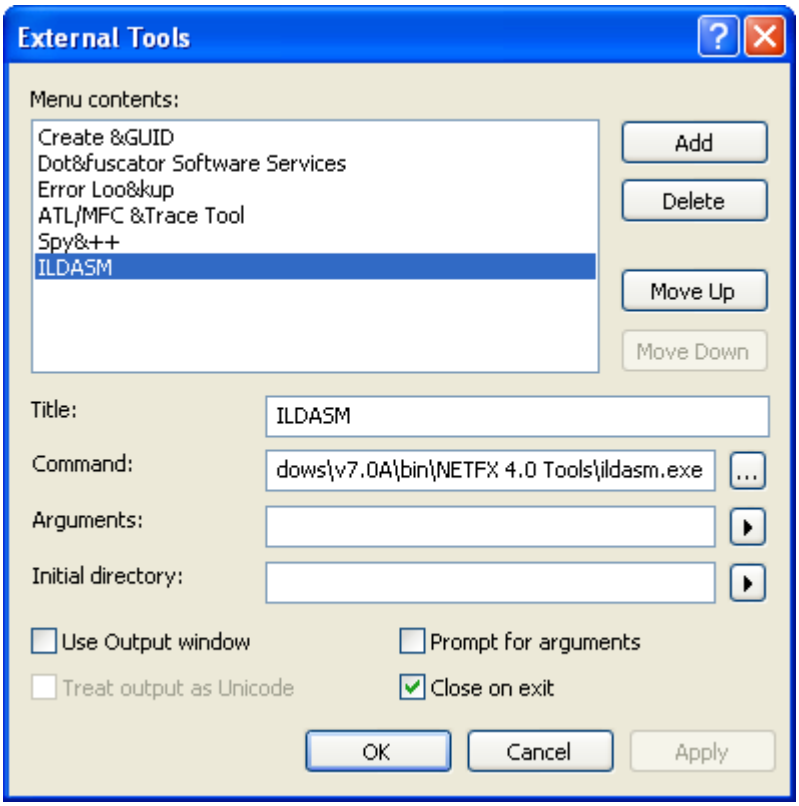
[1] An exception is the pointer type, which is rarely used in C#.

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.

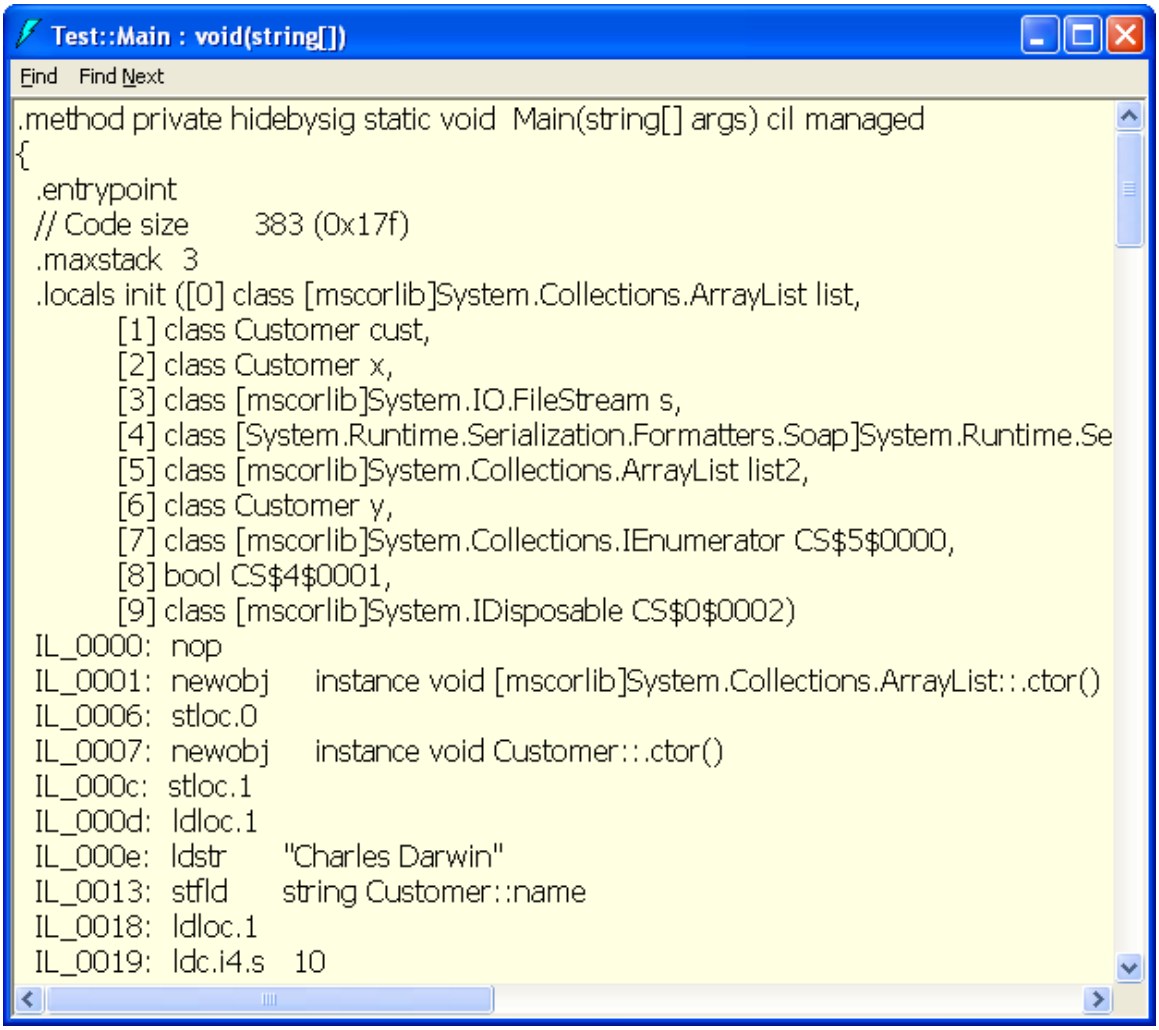
ILDASM

- **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 2010.**
  - 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\v7.0A\bin\NETFX 4.0 Tools.



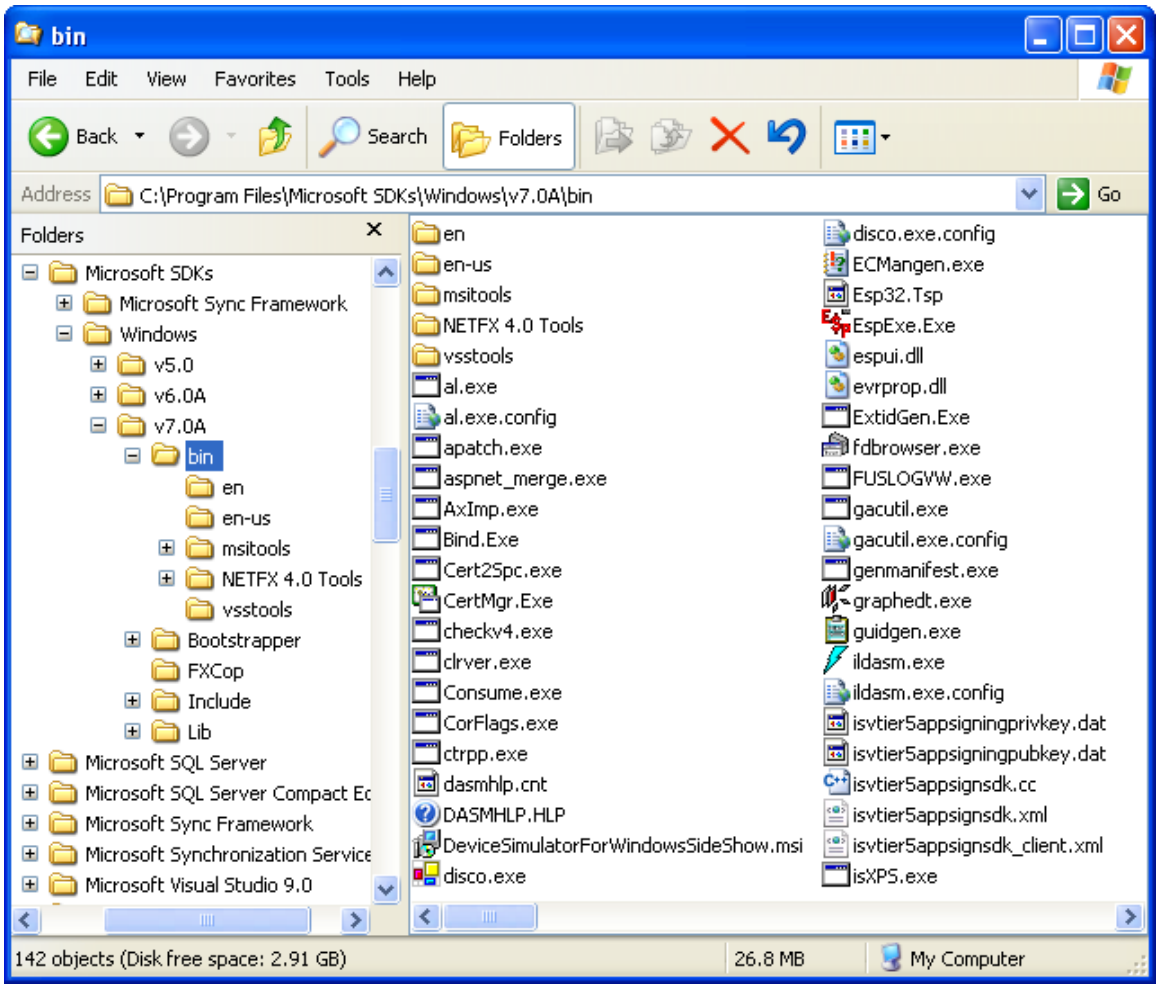
ILDASM (Cont'd)

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



.NET Framework SDK Tools

- **Installing Visual Studio 2010 will also install the .NET Framework SDK, version 7.0A.**
  - These tools are located in the folder C:\Program Files\Microsoft SDKs\Windows\v7.0A\bin.



  - They can be run at the command line from the Visual Studio 2010 Command Prompt, which can be started from All Programs | Microsoft Visual Studio 2010 | Visual Studio Tools | Visual Studio Command Prompt (2010).

Language Interoperability

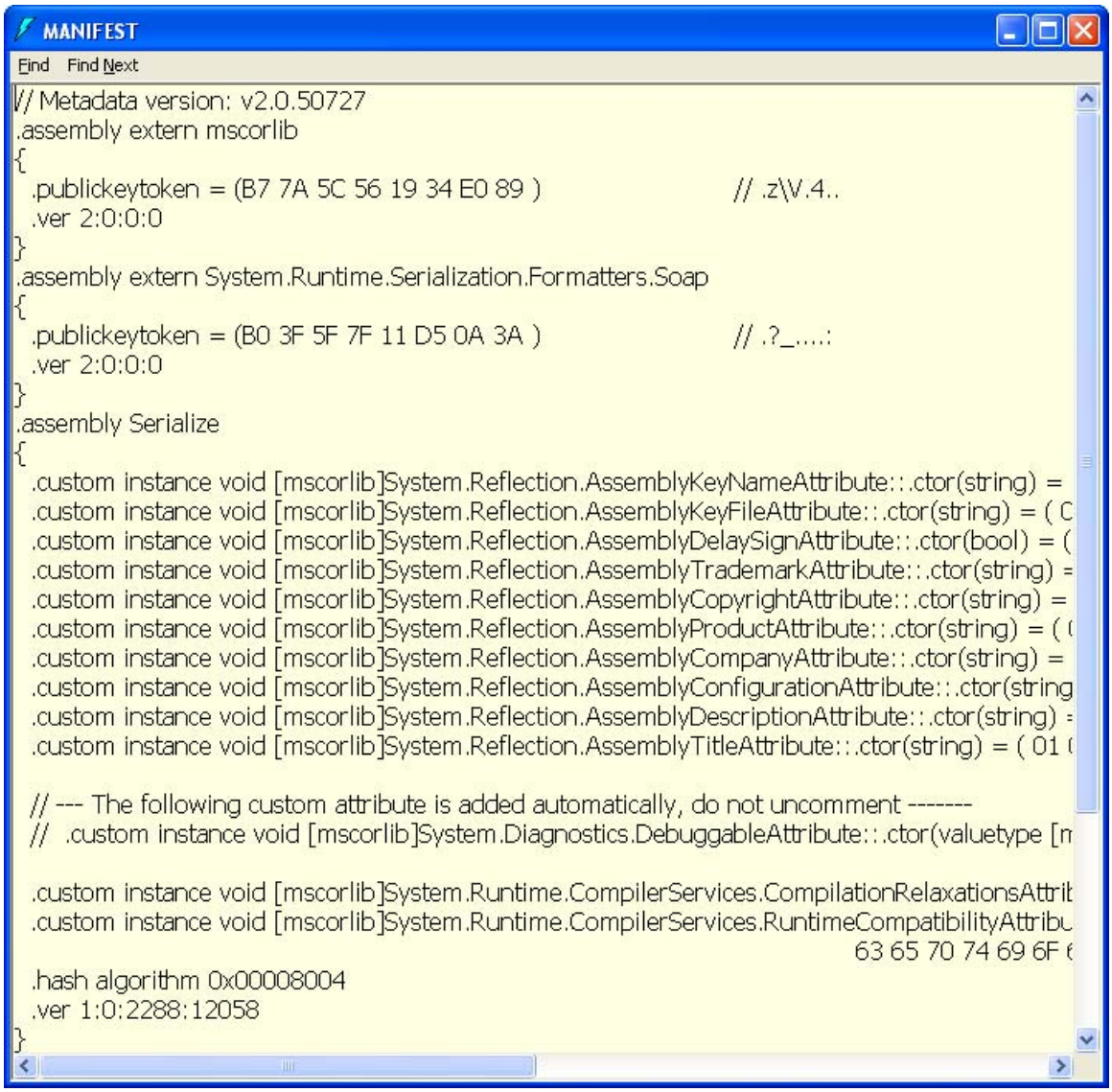
- **Having all language compilers use a common intermediate language and common base class makes it *possible* for languages to interoperate.**
  - All languages need not implement all parts of the CTS.
  - One language can have a feature that another does not.
- **The *Common Language Specification* (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.



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 2010 makes it very easy to lay out forms.
  - You can add code to form events such as a button click.
- **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 5

I/O and Serialization

I/O and Serialization

Objectives

After completing this unit you will be able to:

- **Describe and use .NET Framework classes for working with directories and files.**
- **Explain streams and use them for performing file I/O.**
- **Describe the use of metadata in serialization, and implement serialization in your classes.**
- **Use attributes in .NET to hook into code supplied by the system.**

Input and Output in .NET

- **The .NET Framework provides a very flexible and consistent framework for performing input and output.**
- **The classes supporting input and output are in the *System.IO* namespace.**
- **In this section, we will first examine how the .NET Framework handles directories.**
- **Then we will look at file I/O, which makes use of an intermediary called a *stream*.**
- **We conclude with a discussion of *serialization*, which we illustrated in Chapter 1 and which depends on the concept of metadata that we have discussed.**

Directories

- **The classes *Directory* and *DirectoryInfo* contain routines for working with directories.**
  - All the methods of **Directory** are static, and so you can call them without having a directory instance.
  - The **DirectoryInfo** class contains instance methods.
  - In many cases, you can accomplish the same objective using methods of either class.
  - The methods of **Directory** always perform a security check.
  - If you are going to reuse a method several times, it may be better to obtain an instance of **DirectoryInfo** and use its instance methods, because a security check may not always be necessary.
  - Security will be discussed in later in the course.

Directory Example Program

- **We illustrate both classes with a simple program *DirectoryDemo*, which contains DOS-like commands:**
  - **dir** to show the contents of the current directory
  - **cd** to change the current directory
- **A directory can contain both files and other directories.**
  - The method **GetFiles** returns an array of **FileInfo** objects, and the method **GetDirectories** returns an array of **DirectoryInfo** objects.
- **In this program we only use the *Name* property of *FileInfo*.**
- **In the following section we will see how to read and write files using streams.**

Directory Example Program (Cont'd)

```
using System;
using System.IO;

public class DirectoryDemo
{
   private static string path;
   private static DirectoryInfo dir;
   public static void Main()
   {
      path = Directory.GetCurrentDirectory();
      Console.WriteLine("path = {0}", path);
      dir = new DirectoryInfo(path);
      InputWrapper iw = new InputWrapper();
      string cmd;
      Console.WriteLine( _
         "Enter command, quit to exit");
      cmd = iw.getString("> ");
      while (! cmd.Equals("quit"))
      {
         try
         {
            if (cmd.Equals("cd"))
            {
               path = iw.getString("path: ");
               dir = new DirectoryInfo(path);
               Directory.SetCurrentDirectory(path);
            }
            ...
```

Directory Example Program (Cont'd)

```
...
else if (cmd.Equals("dir"))
{
   FileInfo[] files = dir.GetFiles();
   Console.WriteLine("Files:");
   foreach (FileInfo f in files)
      Console.WriteLine("   {0}",
         f.Name);
   DirectoryInfo[] dirs =
         dir.GetDirectories();
   Console.WriteLine("Directories:");
   foreach (DirectoryInfo d in dirs)
      Console.WriteLine("   {0}",
      d.Name);
}
else
   help();
...
```

- **Here is a sample run. Note that the current directory starts out as the directory containing the program's executable.**

```
path = C:\OIC\NetCs\Chap05\DirectoryDemo\bin\Debug
Enter command, quit to exit
> cd
path: ..
> dir
Files:
Directories:
   Debug
>
```

Files and Streams

- **Programming languages have undergone an evolution in how they deal with the important topic of file I/O.**
  - Early languages, such as FORTRAN, COBOL, and the original BASIC, had I/O statements built into the language.
  - Later languages have tended not to have I/O built into the language, but instead rely on a standard library for performing I/O, such as the **<stdio.h>** library in C. The library in languages like C works directly with files.
- **Still later languages, such as C++ and Java, introduced a further abstraction called a *stream*.**
  - A stream serves as an intermediary between the program and the file.
  - Read and write operations are done to the stream, which is tied to a file.
  - This architecture is very flexible, because the same kind of read and write operations can apply not only to a file, but to other kinds of I/O, such as network sockets.
  - This added flexibility introduces a slight additional complexity in writing programs, because you have to deal not only with files but also with streams, and there exists a considerable variety of stream classes.
  - But the added complexity is well worth the effort, and .NET strikes a nice balance, with classes that make performing common operations quite simple.

Files and Streams (Cont'd)

- **As with directories, the System.IO namespace contains two classes for working with files.**
  - The **File** class has all static methods, and the **FileInfo** class has instance methods.
- **The program *FileDemo* extends the *DirectoryDemo* example program to illustrate reading and writing text files.**
  - We will illustrate binary file I/O later in this chapter, when we discuss serialization.
- **The directory commands are retained so that you can easily exercise the program on different directories.**
- **The two new commands are "read" and "write."**
  - The "read" command illustrates using the **File** class.
  - The "dir" command, already present in the **DirectoryDemo** program, illustrates using the **FileInfo** class.

"Read" Command

- **The code for the "read" command is shown below.**
  - The user is prompted for a file name. The static **OpenText** method returns a **StreamReader** object, which is used for the actual reading.
  - There is a **ReadLine** method for reading a line of text, similar to the **ReadLine** method of the Console class. A **null** reference is returned by **ReadLine** when at end of file.
- **Our program simply displays the contents of the file at the console. When done, we close the *StreamReader*.**

```
...
else if (cmd.Equals("read"))
{
   string fileName = iw.getString("file name: ");
   StreamReader reader = File.OpenText(fileName);
   string str;

   while ((str = reader.ReadLine()) != null)
   {
      Console.WriteLine(str);
   }
   reader.Close();
}
...
```

Code for “Write” Command

- **The code for the “write” command is shown below.**
  - Again we prompt for a file name. This time we also prompt for whether or not to append to the file.
  - There is a special constructor for the **StreamWriter** class that will directly return a **StreamWriter** without first getting a file object.
  - The first parameter is the name of the file, and the second a **bool** flag specifying the append mode. If **true**, the writes will append to the end of an already existing file. If **false**, the writes will overwrite an existing file. In both cases, if a file of the specified name does not exist, a new file will be created.

```
...
else if (cmd.Equals("write"))
{
   string fileName = iw.getString("file name: ");
   string strAppend = iw.getString(
      "append (yes/no): ");
   bool append =
      (strAppend == "yes" ? true : false);
   StreamWriter writer = new StreamWriter(
      fileName, append);
   Console.WriteLine(
      "Enter text, blank line to terminate");
   string str;
   while ((str = iw.getString(">>") != "")
   {
      writer.WriteLine(str);
   }
   writer.Close();
}
...
```

Serialization

- **Using the *File* and *Stream* classes can be quite cumbersome if you have to save a complicated data structure with linked objects.**
  - You have to save the individual fields to disk, remembering which field belongs to which object, and which object instance was linked to another object instance.
  - When restoring the data structure you have to reconstitute that arrangement of fields and object references.
- **The serialization technology provided by the .NET Framework does this for you.**
  - "Serialize" means convert a graph of objects into a linear sequence of bytes.
  - This sequence of bytes can then be written to a stream or otherwise used to transmit all the data associated with the object instance.
  - Objects can be serialized without writing special code, because the metadata knows the object's memory layout.
- **To hook into the serialization mechanism provided by the .NET Framework, you mark your class with a special *attribute*.**

Attributes

- **The traditional approach to implementing functionality is to write code.**
  - A whole different approach to implementing complex code is to let the system do it for you. There must be a way for the programmer to inform the system of what is desired. In the .NET Framework such cues can be given to the system by means of *attributes*.
- **Microsoft introduced attribute-based programming in Microsoft Transaction Server.**
  - The concept was that MTS, not the programmer, would implement complex tasks such as distributed transactions.
  - The programmer would “declare” the transaction requirements for a COM class, and MTS would implement it.
  - This use of attributes was greatly extended in the next generation of MTS, known as COM+.
  - In MTS and COM+ attributes are stored in a separate repository, distinct from the program itself.

Attributes (Cont'd)

- **Attributes are also used in Interface Definition Language (IDL), which gives a precise specification of COM interfaces.**
  - Part of the function of IDL is to make it possible for a tool to generate proxies and stubs for remoting a method call across a process boundary or even across a network.
  - When parameters are passed remotely, it may be necessary to supply additional information.
  - For example, within a process, you can simply pass a reference to an array. But in passing an array across a process boundary, you must inform the tool of the size of the array.
  - This information is communicated in IDL by means of attributes, which are specified using a square bracket notation. Again, attribute information is stored in a location separate from program code.
- **A problem with attributes in both MTS/COM+ and IDL is that they are separate from the program source code. When the source code is modified, the attribute information may become out of synch with the code.**
- **In C#, attributes are declared with square brackets. Unlike IDL, the attributes are part of the program source code. When compiled into intermediate language, the attributes become part of the metadata.**

Serialization Example

- **The program *SerializeAccount* illustrates serialization of the *Account* class by means of an attribute.**
- **The code for the *Account* class is shown below, and only one line of code is involved:**
  - The attribute **[System.Serializable]** is placed before the class. In this case, all the data members of the class will be serialized.
  - If you want to exclude a data member from being serialized, for example, because it contained some kind of temporary cache of data, you can place the attribute **[System.NonSerialized]** in front of the member you want excluded.
  - Serialization applies only to instance data members; static members are never serialized.

```
using System;

[System.Serializable]
public class Account
{
   private decimal balance;
   private string owner;
   private int id;
   ...
```

Serialization Example (Cont'd)

- **While making a class serializable is simply a matter of using an attribute, you must write a little code to cause an object graph to serialize itself.**
- **If you are using serialization to implement persistence, you need to perform the following four basic steps to save the data.**

1. Instantiate a **FileInfo** object where the data will be saved.

2. Open up a **Stream** object for writing to the file.

3. Instantiate a formatter object for laying out the objects in a suitable format.

4. Apply the formatter's **Serialize** method to the root object and the stream.

- **There are various built-in formatters provided by the .NET Framework, including:**
  - **BinaryFormatter** lays out object data in a binary format.
  - **SOAPFormatter** lays out object data in an XML format.
- **If you use a binary formatter, you will need the following two namespaces in order to perform the serialization:**
  - **System.Runtime.Serialization**
  - **System.Runtime.Serialization.Formatters.Binary**

Serialization Example (Cont'd)

- **See the file *SerializeAccount.cs* for a test program that provides commands to save and load a collection of accounts.**

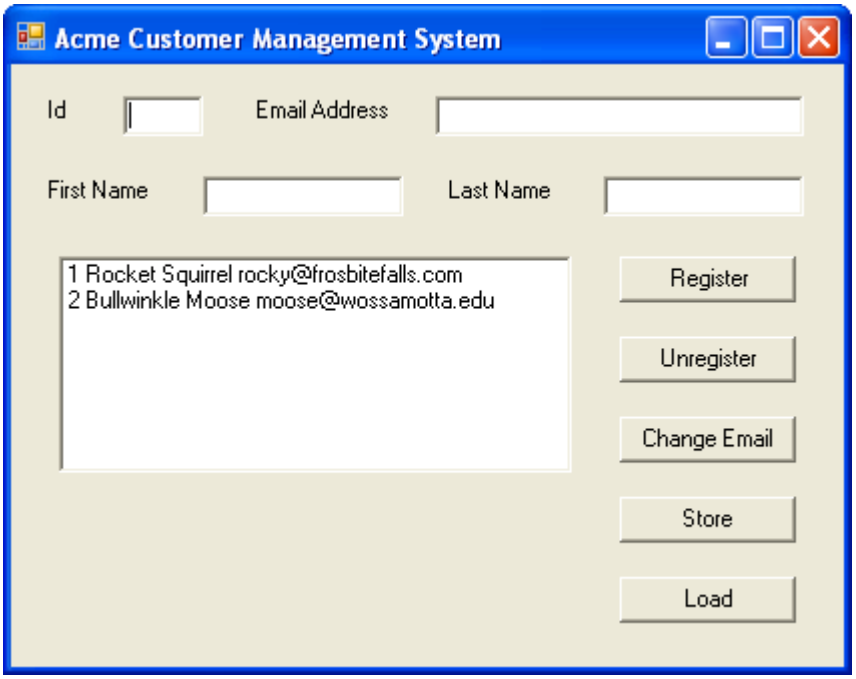
```
using System;
using System.Collections;
using System.IO;
using System.Runtime.Serialization;
using
System.Runtime.Serialization.Formatters.Binary;

public class SerializeAccount
{
   private static ArrayList accounts;
   public static void Main()
   {
      ...
      else if (cmd.Equals("save"))
      {
         FileInfo f = new FileInfo("accounts.bin");
         Stream s = f.Open(FileMode.Create);
         BinaryFormatter b = new BinaryFormatter();
         b.Serialize(s, accounts);
         s.Close();
      }
      else if (cmd.Equals("load"))
      {
         FileInfo f = new FileInfo("accounts.bin");
         Stream s = f.Open(FileMode.Open);
         BinaryFormatter b = new BinaryFormatter();
         accounts = (ArrayList) b.Deserialize(s);
         s.Close();
      }
     ...
```

Lab 5

Serializing the Customer Class

In this lab you will implement serialization code for the **Customer** class in the Acme Customer Management System, enabling you to store the list of customers persistently in a file. You will provide both a binary and a SOAP version of your solution.



Detailed instructions are contained in the Lab 5 write-up at the end of the chapter.

Suggested time: 30 minutes

Summary

- **The .NET Framework provides a flexible and consistent framework for performing input and output.**
- **The classes *DirectoryInfo* and *FileInfo* contain instance methods.**
- **The classes *Directory* and *File* contain static methods and always perform a security check.**
- **File I/O is accomplished with the aid of stream classes.**
- **Serialization makes effective use of metadata to persist data structures to streams.**
- **Attributes in .NET make it easy for your programs to hook into code supplied by the system.**

Lab 5

Serializing the Customer Class

Introduction

In this lab you will implement serialization code for the Customer class in the Acme Customer Management System, enabling you to store the list of customers persistently in a file. You will provide both a binary and a SOAP version of your solution.

Suggested Time: 30 minutes

Root Directory: **OIC\NetCs**

Directories:	**Labs\Lab5\SerializeCustomer**	(do your work here)
	Chap02\CustomerMonolithic	(backup of starter code)
	Chap05\SerializeCustomer\BinaryFormat	(answer, binary)
	Chap05\SerializeCustomer\SoapFormat	(answer, SOAP)

Instructions

1. In the working directory you will find a copy of the **CustomerMonolithic** program from Chapter 2. Build and verify that it is working.

2. In **Customer.cs** add the **[System.Serializable]** attribute to the **Customer** class.

3. Add **using** for the **Runtime.Serialization.Formatters.Binary** namespace.

4. Add public methods **Store** and **Load** to the **Customers** class, to implement serialization and deserialization using the binary formatter. Base your code on the **SerializeAccount** example.

5. Add two buttons to the form to Store and Load the customer list.

6. Implement handlers for these buttons that will call the **Store** and **Load** methods that you implemented. When you load, be sure to update the customers listbox.

7. Build and test. Be sure to verify that data is persisted between separate runs of the program.

8. Modify your solution to use a SOAP formatter. Note that you will need to add a reference to **System.Runtime.Serialization.Formatters.Soap.dll**.

9. Build and test. Examine the XML in the text file where the data is saved.

Chapter 10

Debugging Fundamentals

Debugging Fundamentals

Objectives

After completing this unit you will be able to:

- **Discuss the role of compile-time errors and runtime errors in program development.**
- **Configure Visual Studio for debug, release and special builds.**
- **Use the Visual Studio debugger, including features such as single step, breakpoints, and examining variables.**
- **Perform JIT (just-in-time) debugging in both console and Windows applications.**

Compile-Time Errors

- **Compile-time errors can be detected by the compiler as part of building the program.**
- **Modern strongly-typed languages, such as C#, enable many errors to be caught at compile-time.**
- **As an example of an error we would like to have caught at compile-time, consider this C program:**
  - See **CompileTimeVc**.

```
#include <stdio.h>

int main()
{
   char* onetwothree = "123";
   int number = (int) onetwothree;
   printf("number = %d\n", number);
   printf("Press Enter to exit");
   getchar();
   return 0;
}
```

  - If you build this using the C/C++ compiler in Visual Studio 2010 you will get no errors and no warnings!
  - But if you run it, you get nonsensical data (treating a pointer as an integer):

```
number = 4333612
```

Compile-Time Demo

- **As a simple demonstration of compile-time errors, consider a similar program in C#.**
  - See **Demos\CompileTime**, backed up in the chapter directory in **CompileTime\Step1**.

```
using System;

class CompileTime
{
   static void Main(string[] args)
   {
      string onetwothree = "123";
      int number = (int) onetwothree;
      Console.WriteLine("number = {1:F2}", number);
   }
}
```

  - The compiler will generate an error message, pinpointing the location of the error.

```
CompileTime.cs(10,16): error CS0030: Cannot convert
type 'string' to 'int'
```

- **Comment out the line with the error and do the conversion in the proper way with the *Convert* class.**

```
string onetwothree = "123";
//int number = (int) onetwothree;
int number = Convert.ToInt32(onetwothree);
Console.WriteLine("number = {1:F2}", number);
```

  - Now you will get a clean compile. Ship it, right?

Runtime Errors

- **If you run this program, you encounter a *runtime* error:**

```
Unhandled Exception: System.FormatException: Index
(zero based) must be greater than or equal to zero
and less than the size of the argument list.

    at System.Text.StringBuilder.AppendFormat(
IFormatProvider provider, String format, Object[]
args)
   at System.String.Format(IFormatProvider
provider, String format, Object[] args)
   at System.IO.TextWriter.WriteLine(String format,
Object arg0)
   at System.IO.SyncTextWriter.WriteLine(String
format, Object arg0)
   at System.Console.WriteLine(String format,
Object arg0)
   at CompileTime.Main(String[] args) in
c:\oic\netcs\chap10\compiletime\step1\
compiletime.cs:line 12
```

- **In this case, an exception was thrown, and the location of the error is pinpointed.**
  - Here is the fix, with the erroneous lines commented out.

```
string onetwothree = "123";
//int number = (int) onetwothree;
int number = Convert.ToInt32(onetwothree);
//Console.WriteLine("number = {1:F2}", number);
Console.WriteLine("number = {0:F2}", number);
```

  - Final program is in **CompileTime\Step2**.

Debugging

- **Run-time errors are typically referred to as “bugs”, and debugging is the art of finding and eliminating such errors in your code.**
- **This section of the course is concerned with debugging in the .NET environment.**
- **But erroneous functional behavior is not the only kind of problem that may appear in software.**
  - Software may fail to **scale** and behave improperly under heavy loads.
  - Software may give correct results, but have unacceptably slow **performance**.
  - Software may have defects that cause it to lack **security**.

Bytes Sample Program

- **To illustrate debugging, we will use another very simple example.**
  - See **Demos\Bytes** backed up in **Bytes\Step1Debug**.

```
class Bytes
{
   public static void Main(String[] args)
   {
      int kilo = 1024;
      int mega = kilo * kilo;
      int giga = kilo * mega;
      int tera = kilo * giga;
      Console.WriteLine("kilo = {0} bytes", kilo);
      Console.WriteLine("mega = {0} bytes", mega);
      Console.WriteLine("giga = {0} bytes", giga);
      Console.WriteLine("tera = {0} bytes", tera);
    }
}
```

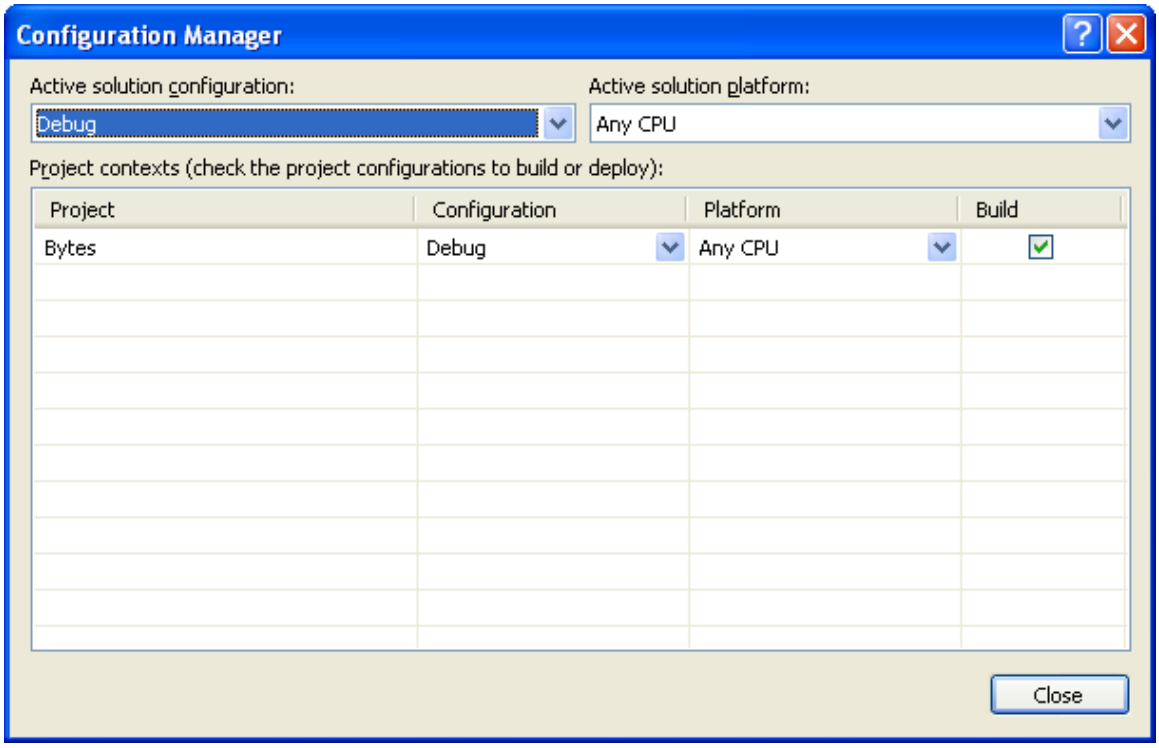
- **The output is a little strange:**

```
kilo = 1024 bytes
mega = 1048576 bytes
giga = 1073741824 bytes
tera = 0 bytes
```

Project Configurations

- **A project *configuration* specifies build settings for a project.**
- **Every project in a Visual Studio solution has two default configurations, *Debug* and *Release*.**
  - By default, a Visual Studio projects builds with the Debug configuration.
- **You can choose the configuration from the main toolbar** 

or use the menu Build | Configuration Manager, which will bring up the Configuration Manager dialog (shown below).



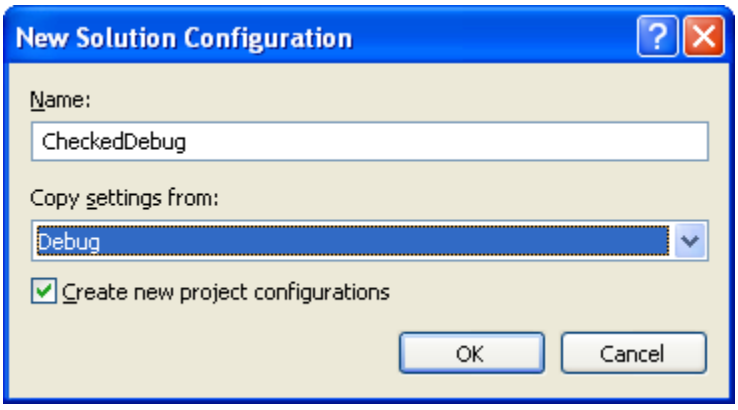
Release Configuration

- **After your program has been debugged, you will normally want to build your program with a Release configuration.**
- **A Release configuration by default performs a number of optimizations to make your code more efficient.**
- **When building with Release configuration, there will be no symbol information created (in a *.pdb* file), and you will not be able to use the debuggers for symbolic debugging.**
- **To make sure that your configuration setting persists, don't delete the *.suo* file of your project.**
- **As an example of a program that is built with a Release configuration, open the solution at *Bytes\Step1Release*.**

Creating a New Configuration

- **Sometimes it is useful to create additional configurations, which can save alternate build settings.**
- **As an example, let's create a configuration for a "checked" build. (The */checked+* compiler switch turns on runtime checking of integer overflow.)**

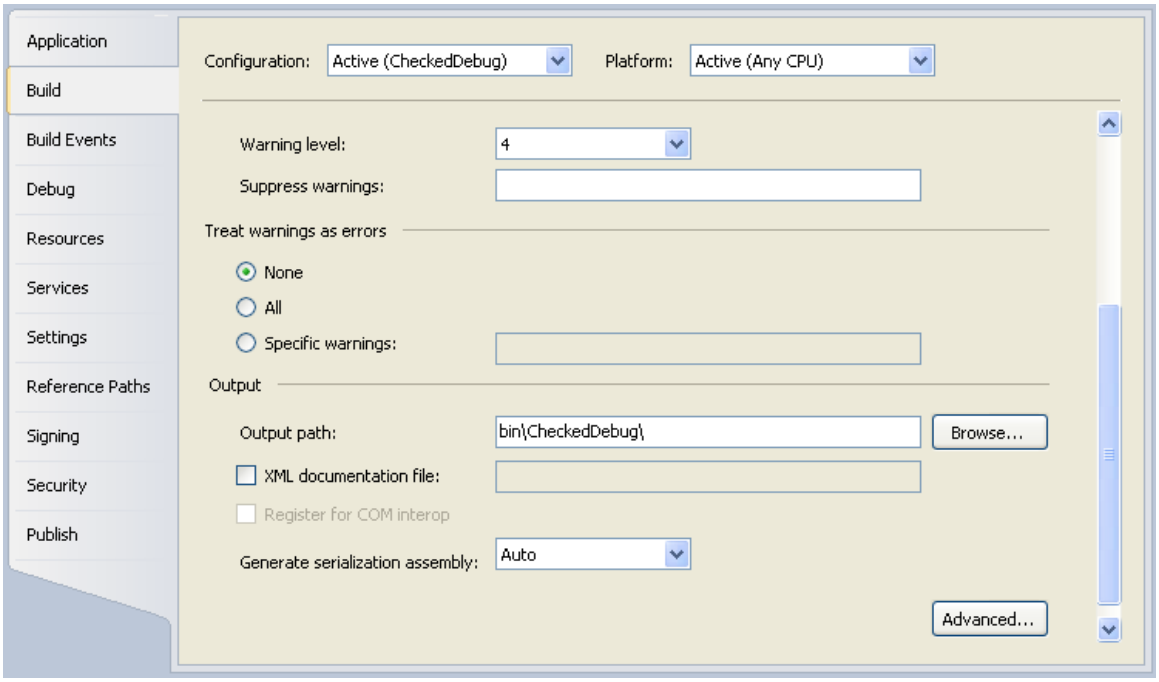
1. Bring up the Configuration Manager dialog.
2. From the Active Solution Configuration: dropdown, choose **<New...>**. The New Solution Configuration dialog will come up.
3. Type **CheckedDebug** as the configuration name. Choose Copy Settings from **Debug**. Check "Also create new project configuration(s)" (see below). Click OK. Then close the Configuration Manager dialog box.



Build Settings for a Configuration

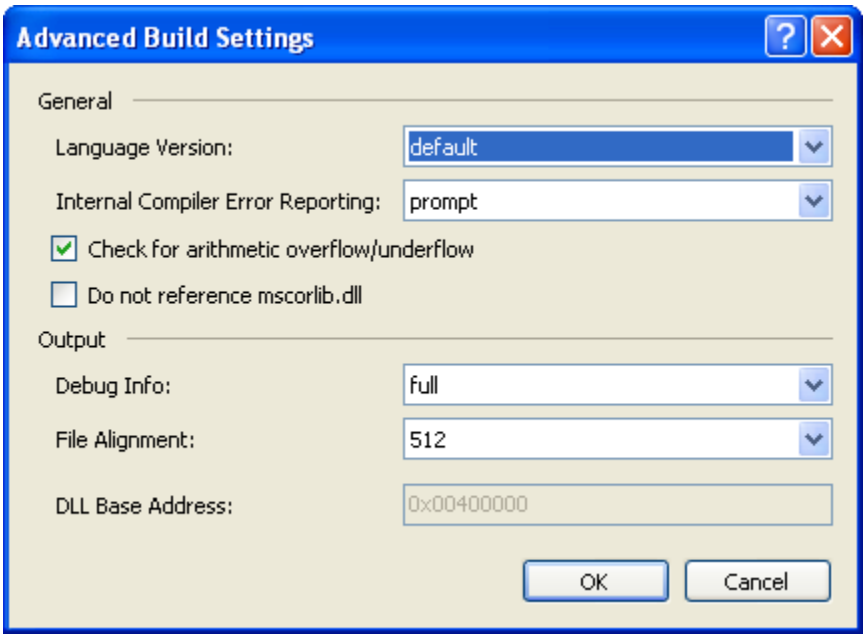
- **Next, we will set the build settings for the new configuration.**
  - Check the toolbar to verify that the new **CheckedDebug** is the currently-active configuration.

1. Right-click over **Bytes** in the Solution Explorer and choose Properties. The “Bytes Property Pages” dialog comes up.
2. Select the “Build” tab on the left. Scroll the window on the right until you see the Advanced button at the bottom right.



Build Settings (Cont'd)

3. Click the Advanced button. Change the setting for "Check for arithmetic overflow/underflow" to checked (see below). Click OK.

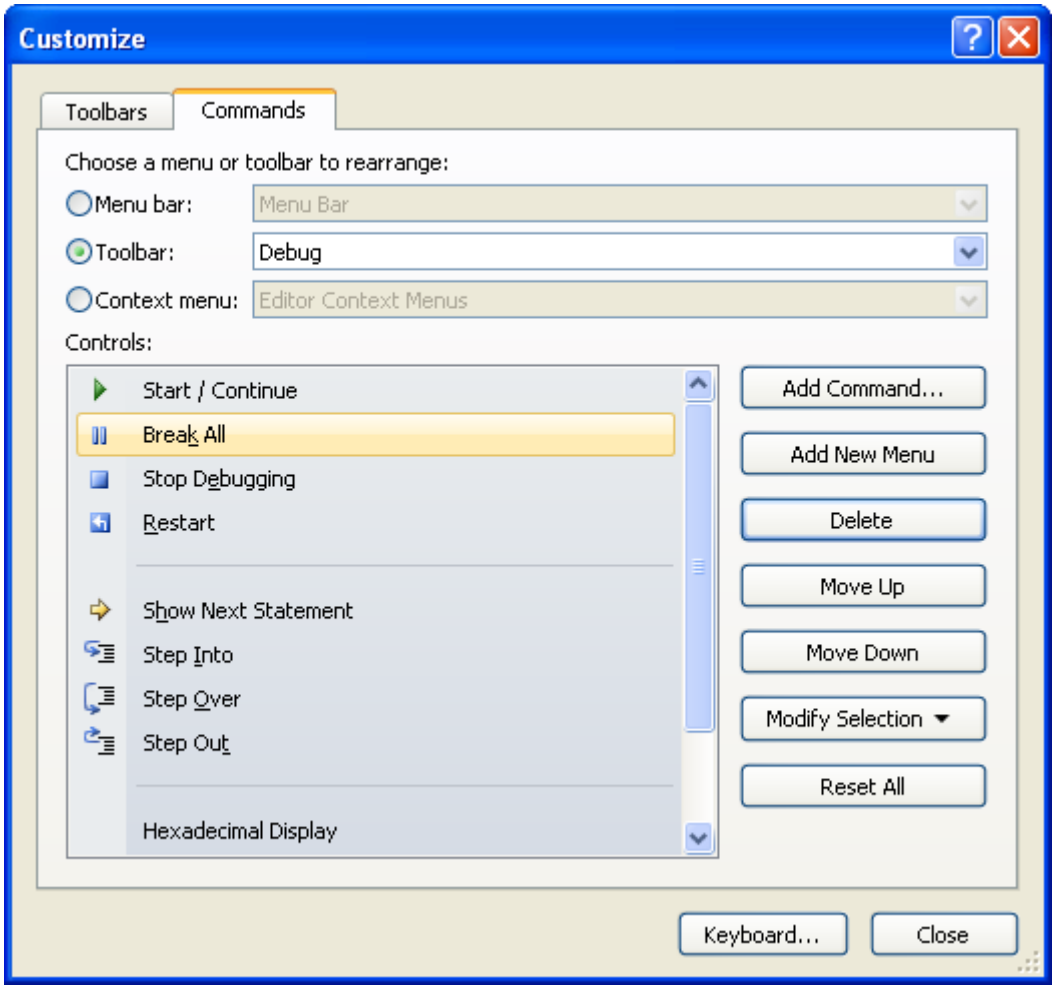


- **This special checked configuration of the program is saved in *Bytes\Step1Checked*.**
  - To make sure that the current configuration is saved, don't delete the file **Bytes.suo**.

Customizing a Toolbar

- **To exercise a program without the debugger, we will add the "Start Without Debugging" command (an open green wedge) to the Debug toolbar.**

1. Select menu Tools | Customize....
2. Select the Commands tab. Select the command at the position where you want the new command to appear.



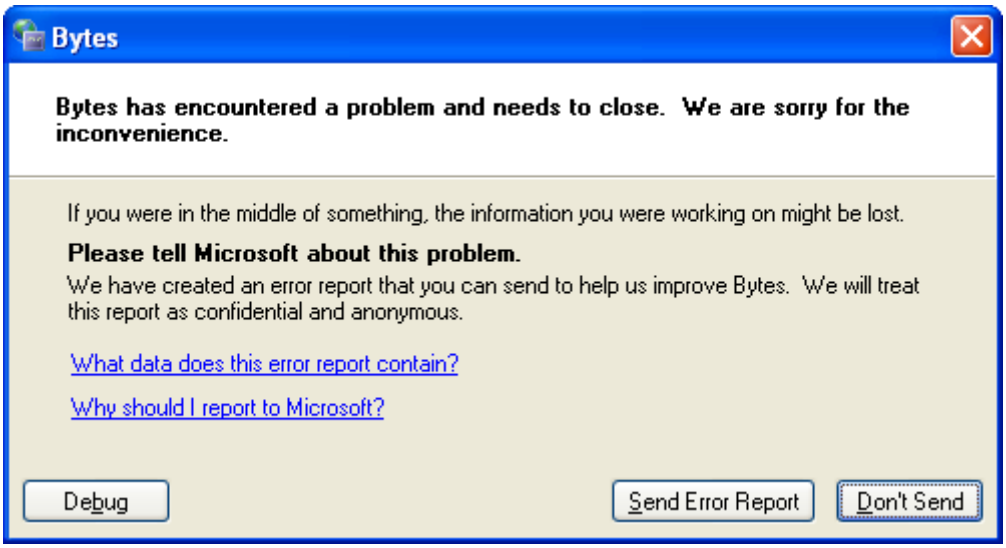
3. Click the Add Command button. Select the command you wish to add and click OK. Close the Customize dialog.

Using the Visual Studio Debugger

- **To be able to benefit from debugging at the source code level, you should have built your executable using a Debug configuration, as discussed previously.**
- **There are three ways to enter the debugger:**
  - Just-in-Time Debugging. You run normally and, if an exception occurs, you will be allowed to enter the debugger. The program has crashed, so you will not be able to run further from here to single step, set breakpoints, and so on. But you will be able to see the value of variables, and the point at which the program failed.
  - Standard Debugging. You start the program under the debugger. You may set breakpoints, single step, and so on.
  - Attach to a running process. You start the program normally, and then from Visual Studio you attach to it. We will discuss this approach later in the chapter.

Overflow Exception

- **Build and run (without debugging) the *Bytes* program from the previous section, making sure to use the *CheckedDebug* configuration.**
- **This time, the program will not run through smoothly to completion, but an exception will be thrown.**



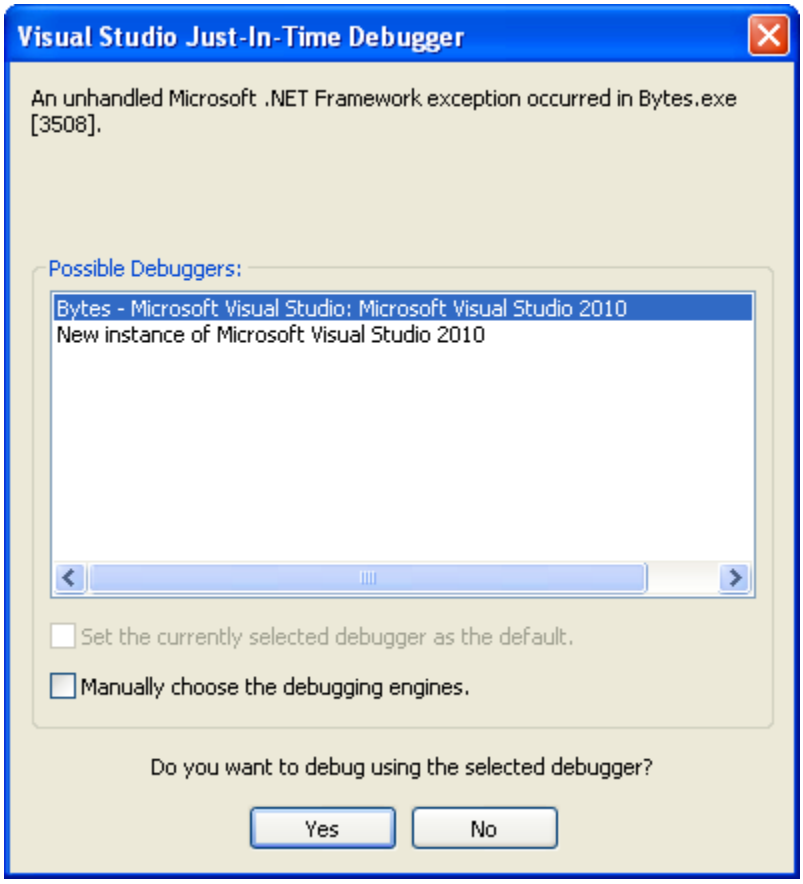
  - Click "Don't Send." You will then see an error message for the exception.

```
Unhandled Exception: System.OverflowException:
Arithmetic operation resulted in an overflow.
   at Bytes.Main(String[] args) in
C:\OIC\NetCs\Demos\Bytes\Bytes.cs:line 12
Press any key to continue . . .
```

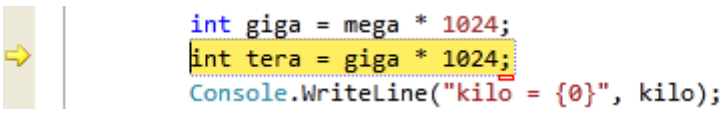
  - Press a key to exit.

Just-in-Time Debugging

- **Run again without the debugger. This time in response to the dialog about the problem click the Debug button.**
  - A "Visual Studio Just-In-Time Debugger" dialog will be shown (see below). Click Yes to debug.

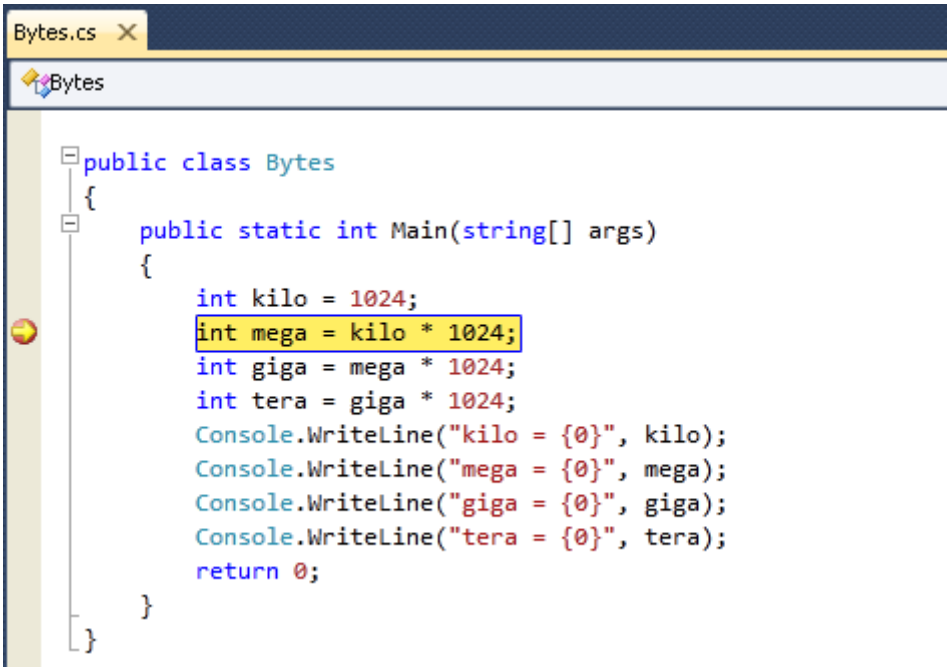


  - The program will break at the place where the problem occurred.



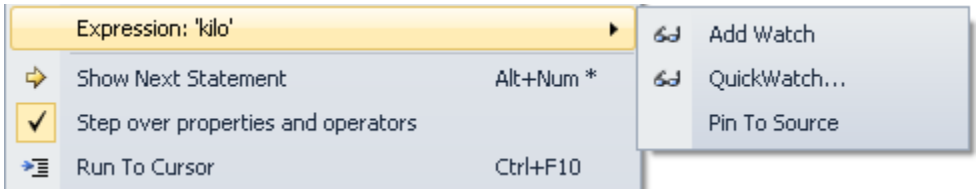
Standard Debugging – Breakpoints

- **The way you typically perform standard debugging is to set a breakpoint and then run, using the debugger.**
- **The easiest way to set a breakpoint is by clicking in the gray bar to the left of the source code window.**
  - You can also set the cursor on the desired line and hit the F9 key to toggle a breakpoint (set if not set, and remove if a breakpoint is set).
  - If you want to remove all breakpoints, you can use the menu Debug | Delete All Breakpoints.
  - A yellow arrow over the red dot of the breakpoint shows where the breakpoint has been hit.

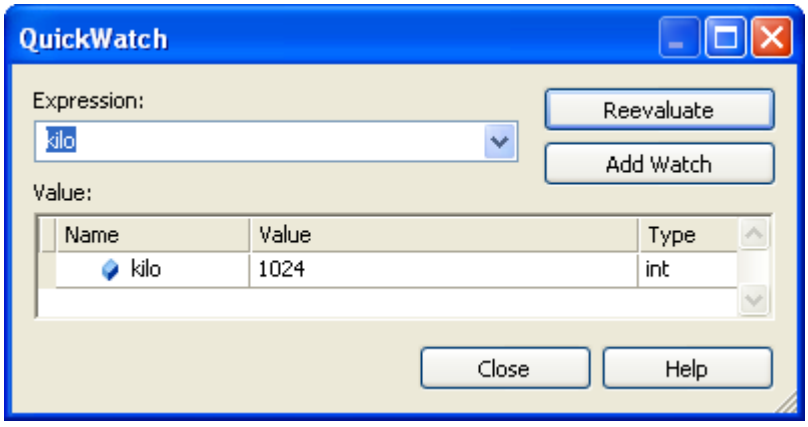


Standard Debugging – Watch Variables

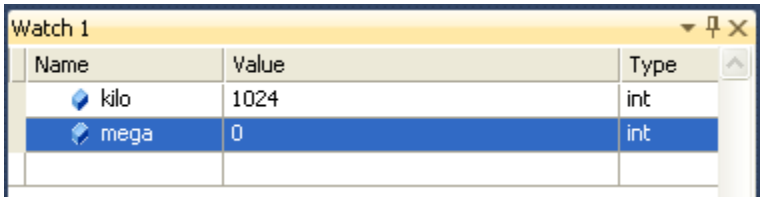
- **The easiest way to inspect the value of a variable is to slide the mouse over the variable you are interested in, and the value will be shown as a yellow tool tip.**
- **You can also right-click over a variable for Add Watch or QuickWatch.**



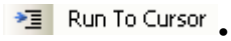
– This illustration shows a typical Quick Watch window.



– And here is a typical Watch window:



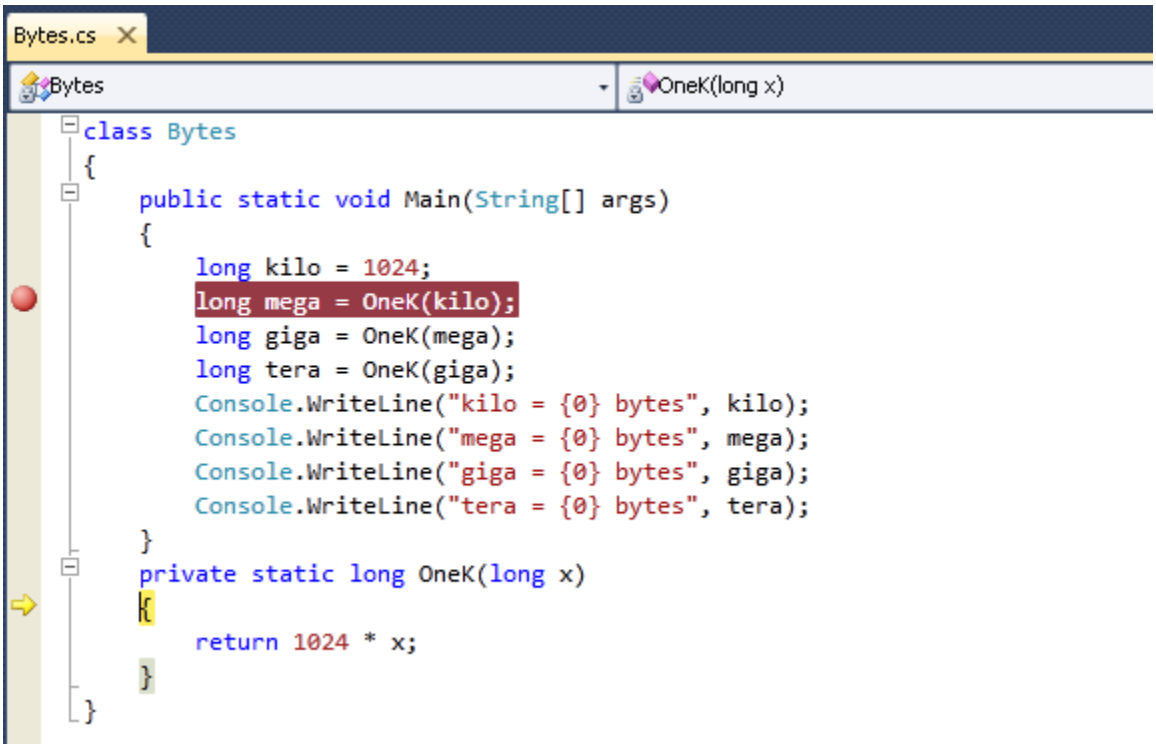
Stepping with the Debugger

- **When you are stopped in the debugger, you can step through instructions, either one at a time or by set amounts.**
- **There are a number of single step buttons**  **. The most common are (in the order shown on the toolbar):**
  - Step Into
  - Step Over
  - Step Out
- **There is also a Run to Cursor button that you can customize** 

.

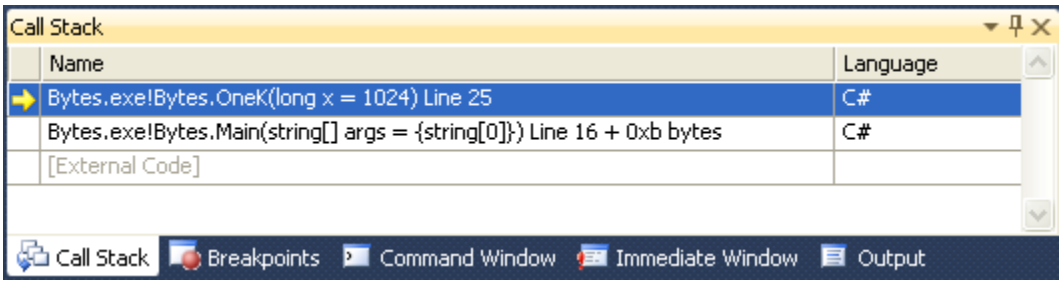
Demo: Stepping with the Debugger

- **Build the *Bytes\Step2* project.**
  - Multiplication by 1024 has been replaced by a function call. (Also, the **long** data type is used, resolving the integer overflow problem.)
- **Set a breakpoint at the first function call, start the program , and then Step Into .**
  - Note that there is a red dot at the breakpoint and a yellow arrow in the function.



The Call Stack

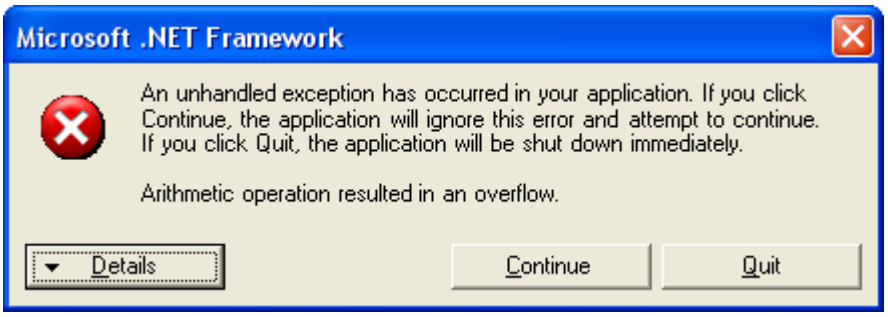
- **When debugging, Visual Studio maintains a Call Stack.**
- **In our simple example, the Call Stack is just two calls deep.**



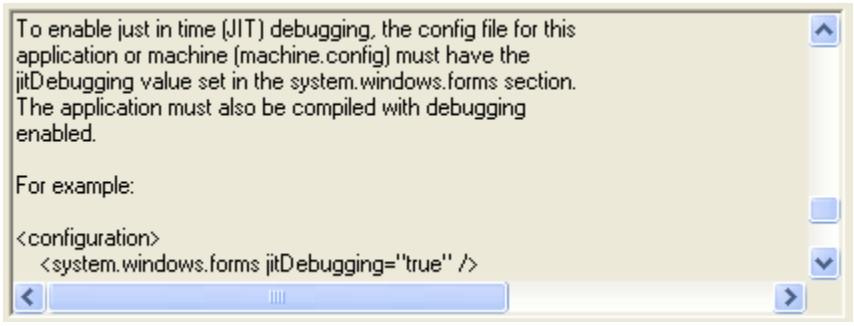
- **Note that the call stack also shows the language.**
  - In .NET it is quite possible to implement mixed-language program, and a called method may be in a different language from the calling method.

JIT Debugging in Windows Apps

- **Just-in-time (JIT) debugging is enabled by default in console applications.**
- **For Windows applications, you have to explicitly enable it.**
- **As a demo, open up a Windows version of the Bytes application, *WinBytes*, in the *Demos* directory.**
  - Build and run. You will see this dialog:



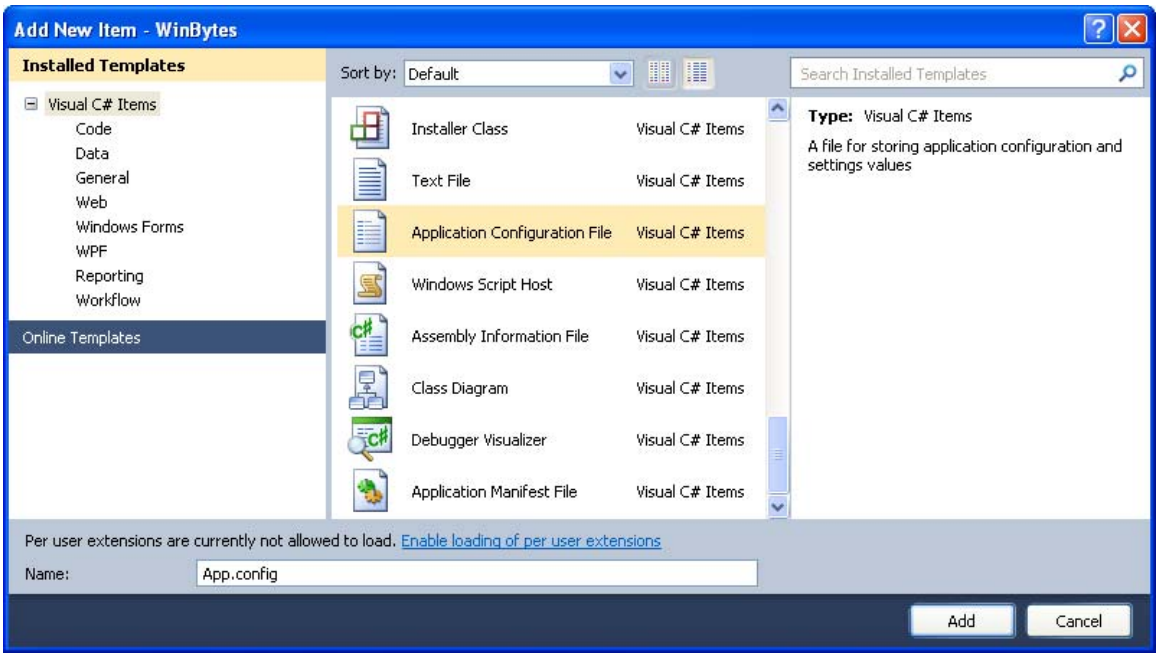
  - Click the Details button and you will see helpful directions for turning on JIT debugging:



Configuration File

- **We need to create a configuration file for our application.**

1. Right-click over the **WinBytes** project and from the context menu select Add | Add New Item. Choose Application Configuration File and click Add.

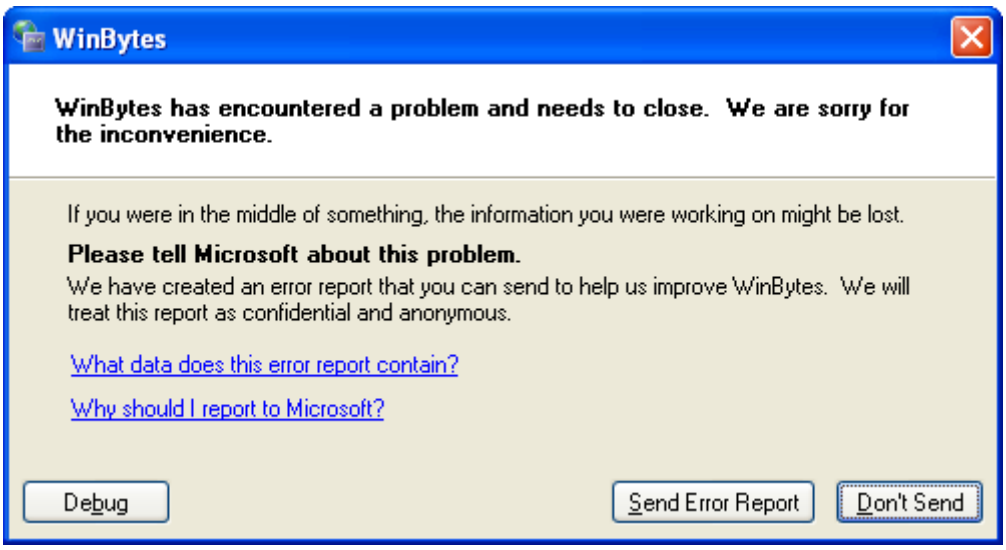


Configuration File (Cont'd)

2. A configuration file **App.config** will be created for you. Add the line that was suggested:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <system.windows.forms jitDebugging="true" />
</configuration>
```

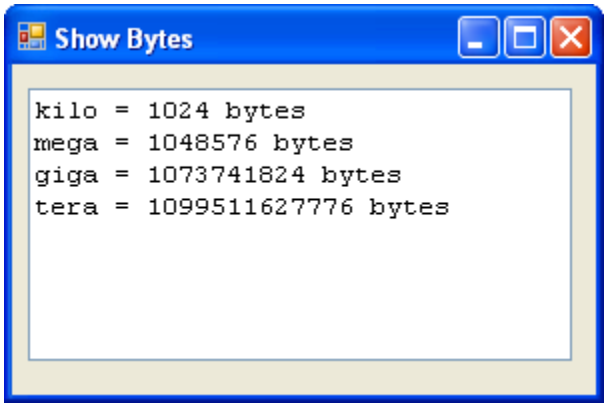
3. Build and run again. Now you should see the familiar problem dialog:



4. Click Debug, and you will start JIT debugging. This version of the program is saved in **WinBytes\Step1**.

Debugged Version of WinBytes

- **In the debugged version of the program you will see information displayed in a multi-line textbox.**
  - See **WinBytes\Step2**.




```
private void Form1_Load(object sender,
System.EventArgs e)
{
   string NL = Environment.NewLine;
   int kilo = 1024;
   txtBytes.Text = string.Format(
      "kilo = {0} bytes", kilo) + NL;
   int mega = kilo * kilo;
   txtBytes.Text += string.Format(
      "mega = {0} bytes", mega) + NL;
   int giga = kilo * mega;
   txtBytes.Text += string.Format(
      "giga = {0} bytes", giga) + NL;
   //int tera = kilo * giga;
   long tera = kilo * (long) giga;
   txtBytes.Text += string.Format(
      "tera = {0} bytes", tera) + NL;
}
```

Lab 10

Debugging Using Visual Studio

In this lab you will use Visual Studio to debug a simple application. Since you should already have basic experience with the Visual Studio debugger, only general instructions will be given. It will be up to you to figure out the problem(s)! This lab is intended to review some aspects of C# and the .NET Framework, as well as the Visual Studio debugger.

Detailed instructions are contained in the Lab 10 write-up at the end of the chapter.

Suggested time: 30 minutes

Summary

- **Strongly-typed languages such as C# facilitate early detection of many errors by the compiler.**
- **Debugging is the process of diagnosing runtime errors not detected at compile-time.**
- **Visual Studio can be configured for Debug, Release and special builds.**
- **Visual Studio provides an integrated debugger, including features such single step, breakpoints, and examining variables.**
- **You can enable JIT (just-in-time) debugging in Windows applications by adding a line to the application's configuration file.**

Lab 10

Debugging Using Visual Studio

Introduction

In this lab you will use Visual Studio to debug a simple application. Since you should already have basic experience with the Visual Studio debugger, only general instructions will be given. It will be up to you to figure out the problem(s)! This lab is intended to review some aspects of C# and the .NET Framework, as well as the Visual Studio debugger.

Suggested Time: 30 minutes

Root Directory: **OIC\NetCs**

Directories:	**Labs\Lab10\SimpleItem**	(do your work here)
	Chap10\SimpleItem\Step1	(backup copy of starter files)
	Chap10\SimpleItem\Step2	(contains lab solution)

Instructions

1. Build and run the starter application. What problems do you see?

2. The first problem is that the display for an item is rather uninformative, just “Item.” The compiler in fact gives a warning (not an error) message. Fix this problem and build and run again.

3. Another problem is that an exception is hit on the last statement of the program. Make sure that you can discover in the debugger that it is this last statement that causes the problem. Since this issue is rather obvious, just comment out the last statement.

4. Now investigate the substantive bug in the program. You will see that the main program adds 5 items to the list and prints them out sorted and unsorted. Then the same 5 items are added in two batches. After the first batch of 3 items are added, the list is sorted, and then the 2 remaining items are added. When the list of items is shown, two of the original items are missing! Why? Fix the problem.