Chapter 20

Advanced Features

Advanced Features

Objectives

After completing this unit you will be able to:

- **Use C# to access directories and files.**
- **Write multithreaded programs in C#.**
- **Build C# classes that can *serialize* themselves to and from files.**
- **Extend the .NET framework to define your own attributes.**
- **Use *reflection* to discover information in C# class metadata.**

Overview

- **In this chapter we discuss a number of advanced features of C# programming using the .NET Framework.**
  - Directories and files
  - Multithreaded programming
  - Serialization
  - Custom attributes
  - Reflection
  - "Unsafe" code
- **These topics are important in their own right, and they provide further examples of important C# concepts that we have been studying.**

Directories And Files

- **In this section we will first examine the *System.IO.Directory* class in the .NET Framework that allows us to work with directories.**
- **We will then look at file input and output, which makes use of an intermediary called a *stream*.**

Directories

- **The classes supporting input and output are in the namespace *System.IO*.**
- **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.
  - A complete discussion of security is beyond the scope of this course.

    For a discussion of security in .NET you may wish to refer to the book *Application Development Using C# and .NET*, in the Prentice Hall/Object Innovations series on .NET technology.

Directories (Cont'd)

- **We illustrate both classes with a simple program *DirectoryDemo*, which contains DOS-like commands to show the contents of the current directory ("dir") and to change the current directory ("cd").**
  - 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.
  - In a sample run of the program notice that the current directory starts out as the directory containing the program's executable.

Files and Streams

- **Programming languages have undergone an evolution in how they deal with the important topic of input/output (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*.**

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 C# strikes a nice balance, with classes that make performing common operations quite simple.

File

- **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

- **Here is the code for the "read" command. 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;
   str = reader.ReadLine();
   while (str != null)
   {
      Console.WriteLine(str);
      str = reader.ReadLine();
   }
   reader.Close();
}
...
```

Write

- **Here is the code for the "write" command.**

```
...
else if (cmd.Equals("write"))
{
   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 = iw.getString(">>");
   while (str != "")
   {
      writer.WriteLine(str);
      str = iw.getString(">>");
   }
   writer.Close();
}
...
```

- **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.

Sample Run

- **We first obtain a listing of existing files in the current directory.**
- **We then create a new text file,** one.txt**, and enter a couple of lines of text data.**
- **We again do "dir", and our new file shows up.**
- **We try out the "read" command.**
- **You could also open up the file in a text editor to verify that it has been created and has the desired data.**
- **Next we write out another line of text to this same file, this time saying "yes" for append mode.**
- **We conclude by reading the contents of the file.**

Sample Run Output

```
path = C:\OI\CSharp\Chap20\FileDemo\bin\Debug
Enter command, quit to exit
> dir
Files:
   FileDemo.exe
   FileDemo.pdb
Directories:
> write
file name: one.txt
append (yes/no): no
Enter text, blank line to terminate
>>hello, world
>>this is second line
>>
> dir
Files:
   FileDemo.exe
   FileDemo.pdb
   one.txt
Directories:
> read
file name: one.txt
hello, world
this is second line
> write
file name: one.txt
append (yes/no): yes
Enter text, blank line to terminate
>>and a third line
>>
> read
file name: one.txt
hello, world
this is second line
and a third line
```

Multiple Thread Programming

- **Modern programming environments allow you to program with multiple threads.**
- **Threads run inside of processes and allow multiple concurrent execution paths.**
  - If there are multiple CPUs, you can achieve parallel processing through the use of threads.
  - On a single processor machine, you can often achieve greater efficiency by using multiple threads, because when one thread is blocked, for example, waiting on an I/O completion, another thread can continue execution.
  - Also, the use of multiple threads can make a program more responsive to shorter tasks, such as tasks requiring user responses.
- **Along with the potential benefit of programming with multiple threads, there is greater program complexity, because you have to manage the issues of starting up threads, controlling their lifetimes, and synchronizing among threads.**
  - Since threads are within a common process and share an address space, it is possible for two threads to concurrently access the same data.
  - Such concurrent access, known as a "race condition," can lead to erroneous results when non-atomic operations are performed, a topic we will discuss in detail later in this section.

.NET Threading Model

- **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.**
  - This class provides methods to start and suspend threads, to sleep, and to perform other thread management functions.
  - The method that will execute for a thread is encapsulated inside a delegate of type **ThreadStart**.
  - As we saw in Chapter 19, a delegate can wrap either a static or an instance method.
- **When starting a thread, it is frequently useful to define an associated class, which will contain instance data for the thread, including initialization information.**
  - A designated method of this class can be used as the **ThreadStart** delegate method.

Console Log Demonstration

- **The *ThreadDemo* program provides an illustration of this architecture.**
- **The *ConsoleLog* class encapsulates a thread ID and parameters specifying a sleep interval and a count of how many lines of output will be written to the console.**
  - It provides the method **ConsoleLog** that writes out logging information to the console, showing the thread ID and number of elapsed (millisecond) ticks.
- **The program is configured with a "slow" thread and a "fast" thread.**
  - The slow thread will sleep for 1 second between outputs, and the fast thread will sleep for only 400 milliseconds.
  - A **ConsoleLog** object is created for each of these threads, initialized with appropriate parameters. Both will do five lines of output.
- **Next, appropriate delegates are created of type *ThreadStart*.**
- **Notice that we use an instance method, *ConsoleThread*, as the delegate method.**
  - Use of an instance method rather than a static method is appropriate in this case, because we want to associate parameter values (sleep interval and output count) with each delegate instance.

Console Log Demonstration (Cont'd)

- **We then create and start the threads. We write a message to the console just before and just after starting the threads.**
  - When do you think the message "Threads have started" will be displayed, relative to the output from the threads themselves? Here is the output from running the program. You will notice a slight delay as the program executes, reflecting the sleep periods.

```
Starting threads ...
Threads have started
Thread 1: ticks = 0
Thread 2: ticks = 0
Thread 2: ticks = 400
Thread 2: ticks = 800
Thread 1: ticks = 1000
Thread 2: ticks = 1200
Thread 2: ticks = 1600
Thread 1: ticks = 2000
Thread 2 is terminating
Thread 1: ticks = 3000
Thread 1: ticks = 4000
Thread 1 is terminating
```

- **The "Threads have started" message is displayed immediately, reflecting the asynchronous nature of the two additional threads.**
  - The **Start** calls return immediately, and the second message prints.
  - Meanwhile, the other threads get started by the system, which takes a little bit of time, and then they each start producing output.

Race Conditions

- **A major issue in concurrency is shared data.**
- **If two computations access the same data, different results can be obtained depending on the timing of the different accesses, a situation known as a race condition.**
  - Race conditions present a programming challenge because they can occur unpredictably.
  - Careful programming is required to ensure they do not occur.
- **Race conditions can easily arise in multithreaded applications, because threads belonging to the same process share the same address space and thus can share data.**
- **Consider two threads making deposits to a bank account, where the deposit operation is not atomic:**
  - Get balance.
  - Add amount to balance.
  - Store balance.

Race Conditions (Cont'd)

- **The following sequence of actions will then produce a race condition, with invalid results.**
  - Balance starts at $100.
  - Thread 1 makes deposit of $25 and is interrupted after getting balance and adding amount to balance, but before storing balance.
  - Thread 2 makes deposit of $5000 and goes to completion, storing $5100.
  - Thread 1 now finishes, storing $125, overwriting the result of thread 2. The $5000 deposit has been lost!

Race Condition Illustration

- **The program *ThreadAccount\Race* illustrates this race condition.**
  - The **Account** class has a method **DelayDeposit**, which updates the balance non-atomically.
  - The thread sleeps for 5 seconds in the middle of the update operation, leaving open a window of vulnerability for another thread to come in.
- **The call *t2.Join* blocks the current thread until thread *t2* finishes.**
  - This technique enables us to show the balance after a thread has definitely completed.
  - Here is the output.
  - As you can see, we have exactly replicated the race condition scenario outlined at the beginning of this section.

```
balance = $100.00
delay deposit of $25.00 on thread 1
deposit of $5,000.00 on thread 2
balance = $5,100.00 (thread 2 done)
balance = $125.00 (thread 1 done)
```

Thread Synchronization Programming

- **Such race conditions can be avoided by serializing access to the shared data.**
  - Suppose only one thread at a time is allowed to access the bank account.
  - Then the first thread that starts to access the balance will complete the operation before another thread begins to access the balance (the second thread will be blocked).
  - In this case threads synchronize based on accessing data.
- **Another way threads can synchronize is for one thread to block until another thread has completed.**
- **The *Join* method is a means for accomplishing this kind of thread synchronization, as illustrated above.**
- **The *System.Threading* namespace provides a number of thread synchronization facilities.**
- **In this section we will illustrate use of the *Monitor* class.**
- **We will also look at use of the C# keyword *lock*, which uses monitors under the hood.**

Monitor

- **You can serialize access to shared data using the *Enter* and *Exit* methods of the *Monitor* class.**
- ***Monitor.Enter* obtains the monitor lock for an object.**
  - An object is passed as a parameter.
  - This call will block if another thread has entered the monitor of the same object.
  - It will not block if the current thread has previously entered the monitor.
- ***Monitor.Exit* releases the monitor lock.**
  - If one or more threads are waiting to acquire the lock, and the current thread has executed **Exit** as many times as it has executed **Enter**, one of the threads will be unblocked and allowed to proceed.
  - An object reference is passed as the parameter to **Monitor.Enter** and **Monitor.Exit**.
  - This is the object on which the monitor lock is acquired or released. To acquire a lock on the current object, pass **this**.

Monitor (Cont'd)

- **The program *ThreadAccount\Monitor* illustrates the use of monitors to protect the critical section where the balance is updated.**
  - In this program we also place the calls to **ShowBalance** just before exiting the monitor. This technique will ensure that we see the balance as soon as it has been updated.
- **The test program is the same as in the previous example, except we are now doing the display of the new balance immediately after the thread has updated the balance.**
  - Here is the output. Notice that the synchronization is successful—we did not have the $5000 deposit wiped out!

```
balance = $100.00
delay deposit of $25.00 on thread 1
deposit of $5,000.00 on thread 2
balance = $125.00 (Thread 1)
balance = $5,125.00 (Thread 2)
```

lock

- **C# provides the *lock* statement as a convenient way to obtain a mutual exclusion lock implemented by the *Monitor* class.**
- **The program *ThreadAccount\Lock* illustrates use of the lock statement.**
  - Besides using **lock**, this example in the relevant code for the modified **Account** class introduces a property **Owner**. The owner is completely independent of the balance, and so we do not need any lock around the code to get the owner.
- **The test program is similar to the program for testing monitors, but it adds code at the end of *Main* to get the owner name, after sleeping briefly to make sure the threads have started and have hit the locks.**
- **Here is the output. It is the same as for the previous example, with an additional line of output showing the owner.**
  - This extra line of output is displayed almost immediately, with very little pause.
  - The output from the threads comes afterwards.

```
balance = $100.00
delay deposit of $25.00 on thread 1
deposit of $5,000.00 on thread 2
owner = Tom Thread
balance = $125.00 (Thread 1)
balance = $5,125.00 (Thread 2)
```

Attributes

- **We have seen various ways to *program* proper synchronization so that we avoid a race condition.**
  - Although the code in our example was quite straightforward, in more elaborate situations code involving multiple threads may become quite complex, especially when exceptional conditions are taken into account.
- **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, including the methods and signatures.**
- **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 is necessary to give more information than when they are passed with the same process.
  - 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.

Attributes (Cont'd)

- **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 .NET, attributes are declared with square brackets, as in IDL.**
- **But unlike IDL, the attributes are part of the program source code.**
  - When compiled into intermediate language, the attributes become part of the metadata.

Synchronization Attribute

- **What does all this have to do with synchronization?**
- **The .NET Framework provides many different kinds of attributes, which your source code may use in order to obtain automatic use of services of the Framework.**
- **An example is the Synchronization attribute, which is applied to a class and automatically synchronizes method calls, allowing only one thread to call into the class at a time.**
- **The program *ThreadAccount\Attribute* illustrates synchronization of the *Account* class by means of an attribute. The code is shown below, and only three lines of code are involved:**
  - A new namespace, **System.Runtime.Remoting.Contexts**.
  - The attribute **[Synchronization(SynchronizationAttribute.REQUIRED)].**
  - The class derives from **ContextBoundObject**.

Synchronization Attribute (Cont'd)

- **The base class *ContextBoundObject* extracts the attribute information from the metadata, using a mechanism known as *reflection*.**
  - It then provides the appropriate code to ensure serial invocation of methods.
  - You do not need to understand how the base class carries out its work; it is abstracted for you.
- **Later in this chapter we will show how you can implement your own custom attributes, and we will also introduce reflection.**
- **We will illustrate how you can define a base class, which enables use of the custom attribute in any derived class.**

```
balance = $100.00
delay deposit of $25.00 on thread 1
deposit of $5,000.00 on thread 2
balance = $125.00 (Thread 1)
balance = $5,125.00 (Thread 2)
owner = Tom Thread
```

Synchronization Attribute (Cont'd)

- **As you can see, the race condition has been avoided in the output, and there is no explicit thread synchronization code using monitors, the lock statement, or a similar construct.**
- **However, the behavior is somewhat different from our previous program, *ThreadAccount\Lock*.**
  - Now all method calls into **Account** are serialized, including calling the property to obtain the owner.
  - When you ran the program, you should have noticed a pronounced delay before the owner was displayed, whereas this happened almost immediately in the **Lock** example.
  - This automatic synchronization is coarse grained, so we obtained greater concurrency, in this example, by implementing the synchronization ourselves.

Serializable Attribute

- **A second example of a useful attribute provided by the .NET Framework is *Serializable*.**
- **When this attribute is applied to a class, the Framework provides code to serialize object instances of the class.**
- **"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.

SerializeAccount

- **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.

```
// Account.cs

using System;

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

- **The *SerializeAccount* class illustrates serializing and deserializing a collection of *Account* objects.**

Serializing a Class

- **What is powerful about the serialization mechanism is that you can serialize complex graphs of objects simply by making each object serializable, which in turn can be done in the .NET Framework by using an attribute.**
  - A composite object is serialized by serializing each of its constituent objects.
  - The .NET Framework collection classes, such as **ArrayList**, have serialization support built in.
- **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.**
  - Instantiate a **FileInfo** object where the data will be saved.
  - Open up a **Stream** object for writing to the file.
  - Instantiate a formatter object for laying out the objects in a suitable format.
  - Apply the formatter's **Serialize** method to the root object and the stream.

De-Serializing

- **There is similar code for deserializing. There are two built-in formatters provided by the .NET Framework:**
  - **BinaryFormatter** lays out object data in a binary format.
  - **SOAPFormatter** lays out object data in an XML format.

Serialization Namespaces

- **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 Sample Run

- **Here are two consecutive sample runs of the program.**
  - In the first run, we add two accounts, remove one, then save.
  - In the second run, we load and verify that we have gotten back the modified collection of accounts.

```
1   Bob         $100.00
2   Mary        $200.00
3   Charlie     $300.00
Enter command, quit to exit
> add
balance: 400
owner: David
id: 4
> add
balance: 500
owner: Ellen
id: 5
> remove
id: 3
> save
> quit
---- Second run ----
1   Bob         $100.00
2   Mary        $200.00
3   Charlie     $300.00
Enter command, quit to exit
> load
> show
1   Bob         $100.00
2   Mary        $200.00
4   David       $400.00
5   Ellen       $500.00
```

Custom Attributes

- **We have seen two examples, synchronization and serialization, of useful attributes provided by the system.**
- **The .NET Framework makes the attribute mechanism entirely extensible, allowing you to define custom attributes, which will cause information to be written to the metadata.**
- **Using *reflection*, you can then extract this information from the metadata at run-time, and modify the behavior of your program appropriately.**
  - In order to simplify the use of the custom attribute, you may declare a base class to do the work of invoking the reflection API to obtain the attribute information stored in the metadata.
  - We will illustrate getting custom attribute information using reflection in this section, and in the following discussion we will discuss reflection in more generality.

Custom Attributes (Cont'd)

- **We illustrate this whole process of defining and using custom attributes with a simple example, *AttributeDemo*, which uses the custom attribute *InitialDirectory* to control the initial current directory when the program is run.**
  - As we saw in the first section of this chapter, by default the current directory is the directory containing the program's executable.
  - In the case of a Visual Studio C# project, built in Debug mode, this directory is **bin\Debug**, relative to the project source code directory.

Using a Custom Attribute

- **First, let's see an example of using the *InitialDirectory* custom attribute.**
  - We will then see how to define and implement it.
- **To be able to control the initial directory for a class, we derive the class from the base class *DirectoryContext*.**
- **We may then apply to the class the attribute *InitialDirectory*, which takes a *string* parameter giving a path to what the initial directory should be.**
  - The property **DirectoryPath** extracts the path from the metadata.
  - If our class does not have the attribute applied, this path will be the default.

Defining an Attribute Class

- **To create a custom attribute, you must define an attribute class, derived from the base class *Attribute*.**
  - The convention is to give your class a name ending in "Attribute."
  - The name of your class without the "Attribute" suffix will then be the name of the custom attribute.
  - In our example, the name of our class is **InitialDirectoryAttribute**, and the name of the corresponding attribute is **InitialDirectory**.
- **You may provide one or more constructors for your attribute class.**
- **The constructors define how to pass positional parameters to the attribute (provide a parameter list, separated by commas).**
  - It is also possible to provide "named parameters" for a custom attribute, where the parameter information will be passed using syntax name = value.
  - You may also provide properties to read the parameter information.

Defining an Attribute Class

- **In our example, we have a property *Path*, which is initialized in the constructor.**

```
// DirectoryAttribute.cs

using System;

public class InitialDirectoryAttribute : Attribute
{
   private string path;
   public InitialDirectoryAttribute(string path)
   {
      this.path = path;
   }
   public string Path
   {
      get
      {
         return path;
      }
   }
}
```

Defining a Base Class

- **The last step in working with custom attributes is to provide a means to extract the custom attribute information from the metadata.**
- **As we shall see in the next section, the .NET Framework provides an elaborate API, the Reflection API, for precisely this purpose.**
  - The root class for metadata information is **Type**, and you can obtain the Type of any object by calling the method **GetType**, which is provided in the root class object.
  - To read custom attribute information, you need only one method, **Type.GetCustomAttributes**.
- **Although it would be quite feasible for the program using the custom attribute to perform this operation directly, normally you will want to make the coding of the client program as simple as possible.**
  - Hence it is useful to provide a base class to do the work of reading the custom attribute information from the metadata.
  - In the previous section we saw an example of such a base class, **ContextBoundObject**, which was used when we wanted a class to be able to use the **Synchronization** attribute.

Defining a Base Class (Cont'd)

- **In our case, we provide a base class *DirectoryContext*, which is used by a class wishing to take advantage of the *InitialDirectory* attribute.**
- **This base class provides the property *DirectoryPath* to return the path information stored in the metadata. Here is the code for the base class:**

```
// DirectoryContext.cs

using System;
using System.Reflection;
using System.IO;

public class DirectoryContext
{
   virtual public string DirectoryPath
   {
      get
      {
         Type t = this.GetType();
         foreach (Attribute a
            in t.GetCustomAttributes(true))
         {
            InitialDirectoryAttribute da =
               a as InitialDirectoryAttribute;
            if (da != null)
            {
               return da.Path;
            }
         }
         return Directory.GetCurrentDirectory();
      }
   }
}
```

Defining a Base Class (Cont'd)

- **We require the *System.Reflection* namespace. *GetType* returns the current *Type* object, and we can then use the *GetCustomAttributes* method to obtain a collection of *Attribute* objects from the metadata.**
  - This collection is heterogeneous, consisting of different types.
  - We can use the C# as operator to test if a given element in the collection is of type **InitialDirectoryAttribute**.
  - If we find such an element, we return the **Path** property.
  - Otherwise, we return the default current directory, obtained from **GetCurrentDirectory**.

Reflection

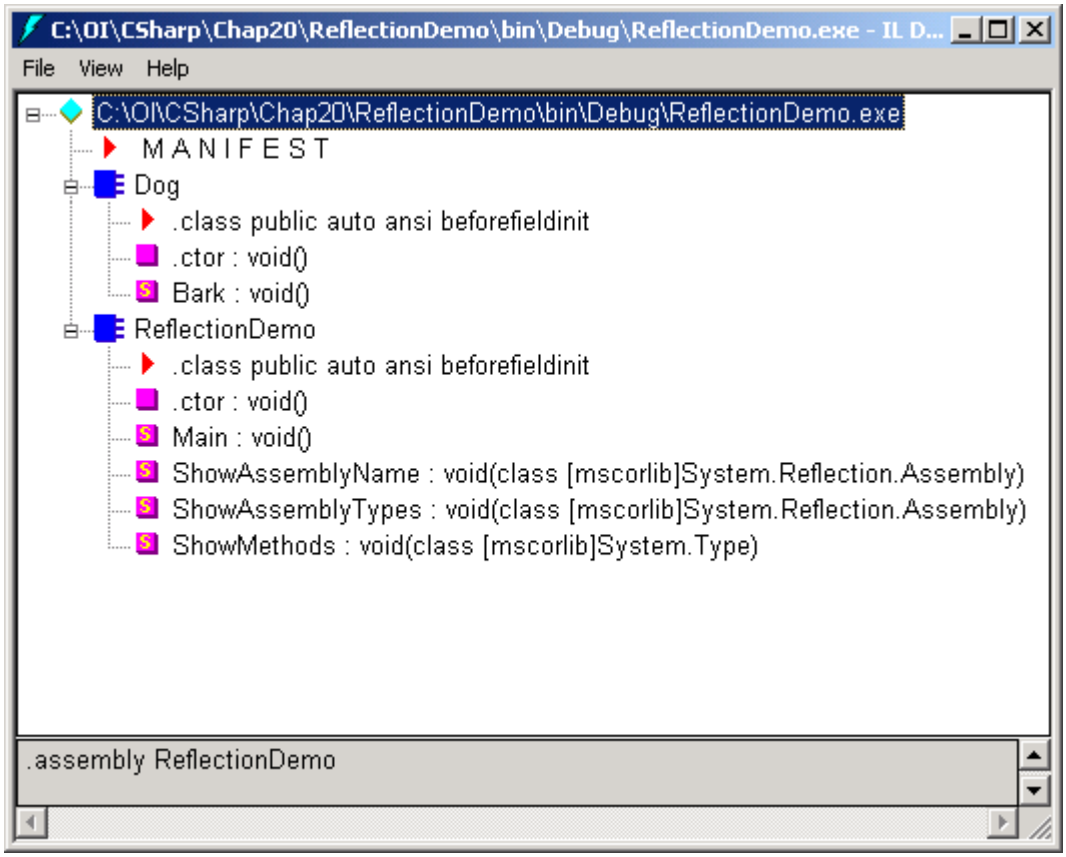
- **At the heart of .NET is metadata, which stores very complete type information.**
- **The Reflection API permits you to query this information at runtime.**
- **You can also dynamically invoke methods, and through the *System.Reflection.Emit* namespace, you can even dynamically create and execute MSIL code.**
  - In the previous section we saw an illustration of using reflection to obtain custom attribute information.
  - In this section we will provide a further example of dynamically obtaining a listing of all types defined in an assembly. (A program's EXE file is an example of an assembly.
  - We will discuss assemblies in the next chapter, and see some further examples, such as dynamic link libraries, or DLLs.)
  - We are only scratching the surface of a very large subject, but our example should help you get started.

Reflection Demo

- **As befits the name "reflection", our demo program *ReflectionDemo* shows information about itself.**
  - The program contains two classes, **Dog** and **ReflectionDemo**.
  - The program calls the **Bark** method of **Dog**.
  - It then calls **Assembly.GetExecutingAssembly** to obtain its assembly. It shows all the types in the assembly, and all the methods of each type.

Using ILDASM

- **You can examine information about any assembly by using the *ILDASM* utility.**
  - To run **ILDASM**, simply enter **ildasm** at a command prompt or use Start | Run.
  - To examine a particular assembly, use the File | Open command.
  - The figure shows the types for **ReflectionDemo.exe**, which compares with the output from our ReflectionDemo program.



Unsafe Code

- **The mainstream use of C# is to write *managed code*, which runs on the Common Language Runtime.**
- **As we shall see in the next chapter, it is quite possible for a C# program to call unmanaged code, such as a legacy COM component, which runs directly on the operating system.**
  - This facility is important, because a tremendous amount of legacy code exists, which is all unmanaged.
  - There is overhead in transitioning from a managed environment to an unmanaged one and back again.
- **C# provides another facility, called *unsafe code*, which allows you to bypass the .NET memory management and get at memory directly, while still running on the CLR.**
  - In particular, in unsafe code you can work with **pointers**, which we will discuss later in this section.

Unsafe Blocks

- **The most circumspect use of unsafe code is within a block, which is specified using the C# keyword unsafe.**
- **The program *UnsafeBlock* illustrates using the *sizeof* operator to determine the size in bytes of various data types.**
  - You will get a compiler error if you try to use the **sizeof** operator outside of unsafe code.

```
// UnsafeBlock.cs
using System;
struct Account
{
   private int id;
   private decimal balance;
}

public class UnsafeBlock
{
   public static void Main()
   {
      unsafe
      {
         Console.WriteLine("size of int = {0}",
                           sizeof(int));
         Console.WriteLine("size of decimal = {0}",
                           sizeof(decimal));
         Console.WriteLine("size of Account = {0}",
                           sizeof(Account));
      }
   }
}
```

Unsafe Blocks (Cont'd)

- **To compile this program at the command line, open up a DOS window and navigate to the directory *c:\OI\CSharp\Chap20\UnsafeBlock*.**
- **You can then enter the following command to compile using the */unsafe* compiler option:**

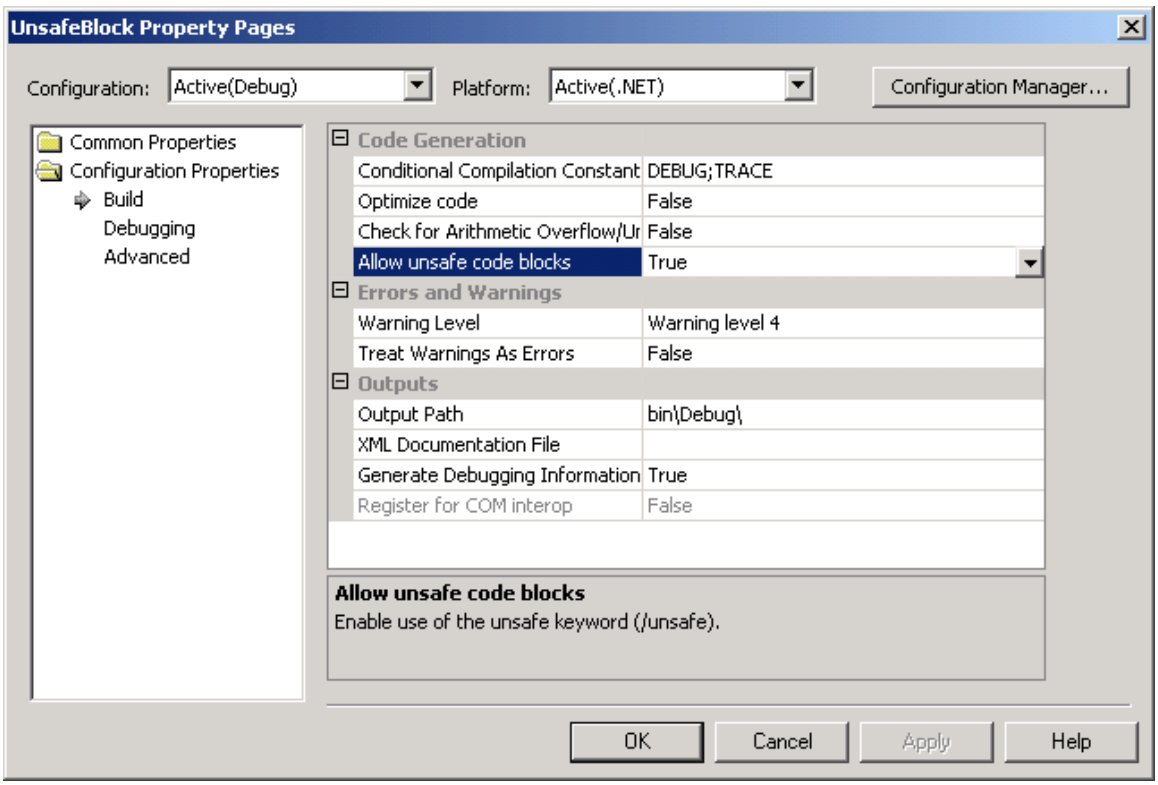
```
csc /unsafe UnsafeBlock.cs
```

  - (You may ignore the warning messages, as our program does not attempt to use fields of **Account**. It only applies the **sizeof** operator.)
  - To run the program, type **unsafeblock** at the command line, obtaining the output shown below:

```
C:\OI\CSharp\Chap20\UnsafeBlock>unsafeblock
size of int = 4
size of decimal = 16
size of Account = 20
```

Unsafe Option in Visual Studio

- **To set the unsafe option in Visual Studio, perform the following steps:**
  - Right-click over the project in the Solution Explorer, and choose Properties.
  - In the Property Pages window that comes up, click on Configuration Proper-ties and then on Build.
  - In the dropdown for Allow unsafe code blocks, choose True. See Figure 20–2.
  - Click OK. You can now compile your project in unsafe mode.



Pointers

- **In Chapter 9 we saw that C# has three kinds of data types:**
  - Value types, which directly contain their data
  - Reference types, which refer to data contained somewhere else
  - Pointer types
- **Pointer types can only be used in unsafe code.**
  - A pointer is an **address** of an actual memory location.
  - A pointer variable is declared using an asterisk after the data type.
  - To refer to the data a pointer is pointing to, use the **dereferencing** operator, which is an asterisk before the variable.
  - To obtain a pointer from a memory location, apply the **address of** operator, which is an ampersand in front of the variable.
- **Here are some examples.**

```
int* p;      // p is a pointer to an int
int a = 5;   // a is an int, with 5 stored
p = &a;      // p now points to a
*p = 12;     // 12 is now stored in location
pointed
             // to by p. So a now has 12 stored
```

Pointers (Cont'd)

- **Pointers were widely used in the C programming language, because functions in C only pass data by value.**
- **Thus if you want a function to return data, you must pass a pointer rather than the data itself.**
- **The program *UnsafePointer* illustrates a *Swap* method, which is used to interchange two integer variables.**
  - Since the program is written in C#, we can pass data by reference.
  - We illustrate with two overloaded versions of the **Swap** method, one using **ref** parameters and the other using pointers.
  - Rather than using an **unsafe** block, this program uses **unsafe** methods, which are defined by including **unsafe** among the modifiers of the method.
  - Both the **Main** method and the one **Swap** method are unsafe.
- **Again you should compile the program using the unsafe option, either at the command line or in the Visual Studio project.**
  - The first swap interchanges the values. The second swap brings the values back to their original state.

Fixed Memory

- **When working with pointers, there is a pitfall.**
  - Suppose you have obtained a pointer to a region of memory that contains data you are working on.
  - Since you have a pointer, you are accessing memory directly.
  - But suppose the garbage collector collects garbage and moves data about in memory.
- **Then your object may now reside at a different location, and your pointer may no longer be valid.**
- **To deal with such a situation, C# provides the keyword *fixed*, which declares that the memory in question is "pinned" and cannot be moved by the garbage collector.**
  - Note that you should use **fixed** only for temporary, local variables, and you should keep the scope as circumscribed as possible.
  - If too much memory is pinned, the CLR memory management system cannot manage memory efficiently.

Fixed Memory Illustration

- **The program *UnsafeAccount* illustrates working with *fixed* memory.**
  - This program declares an array of five **Account** objects, and then assigns them all the same value.
  - The attempt to determine the size of this array is commented out, because you cannot apply the **sizeof** operator to a managed type such as **Account[]**.
- **It also illustrates the arrow operator for dereferencing a field in a struct, when you have a pointer to the struct.**
  - For example, if **p** is a pointer to an instance of the struct **Account** shown below, the following code will assign values to the account object pointed to by p.

```
p->id = 101;          // assign the id field
p->balance = 50.00m;      // assign the balance
field
```

Summary

- **We first looked at directories and files, which were used for examples later in the chapter.**
- **The .NET threading model is built on delegates.**
- **Synchronization can be accomplished through .NET Framework classes such as *Monitor* and through the C# *lock* statement.**
- **Attributes are a very powerful concept in .NET, making it possible to accomplish much without writing code.**
- **We looked at the built-in attributes *Synchronized* and *Serializable*, and we saw how to create a custom attribute.**
- **Custom attribute information can be read from metadata using the Reflection API.**
- **Unsafe code allows C# programs to access legacy code and use pointers.**