It is impossible to cover in one chapter, or even in one book, all of the .NET Framework classes. Although coverage is incomplete, the .NET classes cover a large fraction of the Win32 API, as well as much else. While a lot of attention has been focused on the Internet-related functionality, the development model for Windows applications has changed as well.

This chapter focuses on those classes that illustrate the key concepts and patterns that appear throughout the .NET Framework. You will find this approach more fruitful over the long run than our attempting to explain a little about every class that you might need without giving you much insight. Other chapters will go into more depth about other parts of the framework, such as Windows Forms, ASP.NET, ADO.NET security, and Web Services.

We start out by exploring the concept of reflection and metadata. Metadata appears everywhere in .NET and is critical to understanding how the Common Language Runtime (CLR) can provide services for your applications. Next, we explore file input/output for several reasons. First, it introduces the important topic of serialization. Second, the Path class provides an example of how some framework classes provide some or all of their functionality through static methods. Third, the formatter classes are used in several places in .NET.

Understanding serialization will give you a concrete idea of how the framework can handle objects transparently for you. It also appears in a supporting role any place where objects have to be stored or transported. Our discussion of the ISerializable interface again demonstrates how much easier it is to implement an interface in .NET than with COM.

To further develop an understanding of the .NET model for applications, we introduce programming with threads under .NET and several .NET synchronization techniques to handle multithreading conflicts. The various
synchronization techniques illustrate the tradeoffs of using attributes supplied by the framework versus doing it yourself.

To further your understanding of the .NET programming model, we introduce context and the use of proxies and stubs to implement system services. We also look at using application domains, which are more efficient than Win32 processes in achieving application isolation.

The asynchronous design pattern appears throughout .NET and is discussed in some detail. We give some examples of remoting because it is a key technology and it summarizes many of the concepts developed in this chapter. The chapter uses several attributes provided by the .NET Framework, and we show how to implement and use custom attributes. We discuss finalization so that you can understand how to make sure resources are properly freed in your applications.

Metadata and Reflection

The Serialization example in Chapter 2 demonstrates how metadata makes many of the services of the CLR possible. Many of the technologies we cover in the rest of the book rely on metadata, although we will not always stop and point this out.

Metadata is information about the assemblies, modules, and types that constitute .NET programs. If you have ever had to create IDL to generate a type library so that your C++ COM objects could be called by Visual Basic, or to create proxies and stubs, you will appreciate how useful metadata is and will be grateful that it comes “for free.”

Compilers emit metadata, and the CLR, the .NET Framework, or your own programs can use it. Since we want to give you an understanding of how metadata works, we will focus our discussion on the use of metadata, not the creation of metadata. Metadata is read using classes in the System::Reflection namespace.\(^1\)

When you load an assembly and its associated modules and types, the metadata is loaded along with it. You can then query the assembly to get those associated types. You can also call GetType on any CLR type and get its metadata. GetType is a method on System::Object, which every CLR type inherits from. After you get the Type associated with an object, you can use the reflection methods to get the related metadata.

The Reflection sample program takes the case study’s Customer assembly and prints out some of the metadata available. You should examine the output and source code as you read the next sections. You should espe-

\(^1\) There is a lower level set of unmanaged COM interfaces for accessing metadata, but we will not discuss them here. See “Metadata in .NET” by Matt Pietrek in the October 2000 MSDN Magazine.
cially compare the output of the program with the source code in the file *customer.h*.

The program clearly shows that it is possible to retrieve all of the types in an assembly and reconstruct the structures, interfaces, properties, events, and methods associated with those types.

First we load the assembly into memory and write out its name.

```csharp
Assembly *a = Assembly::Load(assemblyName);
Console::WriteLine("Assembly {0} found.", a->FullName);
```

The output for this statement is appropriate for an unsigned assembly:

```
Assembly Customer, Version=1.0.643.18973, Culture=neutral,
PublicKeyToken=null found.
```

One of the properties of the `Assembly` class is the `CodeBase`, discussed Chapter 7, “Assemblies and Deployment.” The security evidence associated with this assembly is another property.

The following code tries to get the entry point for the assembly:

```csharp
MethodInfo *entryMethodInfo = a->EntryPoint;
```

Since this is a typical C++ component assembly, the entry point is `__DllMainCRTStartup@12`. If this was an executable program, we could use the `Invoke` method on the `MethodInfo` class to run the startup code in the assembly.

The sample uses the assembly’s `GetModules` method to find associated modules with this assembly. In this case we have only one, named `customer.dll`. We could then find the types associated with the module. Instead, we use the assembly’s `GetTypes` method to return an array of the assembly’s types.

**Type**

The abstract class `Type` in the `System` namespace defines .NET types. Since there are no functions outside of classes or global variables in .NET, getting all the types in an assembly will allow us to get all the metadata about the code in that assembly. `Type` represents all the types present in .NET: classes, structs, interfaces, values, arrays, and enumerations.

The `Type` class is also returned by the `GetType` method on the `System::Object` class and the static `GetType` method on the `Type` class itself. The latter method can be used only with types that can be resolved statically.

---

2 You can also load and execute the assembly from the AppDomain, as we discuss later in this chapter.
One of Type's properties is the assembly to which it belongs. You can get all the types in the containing assembly once you have the Type of one object. Type is an abstract class, and at runtime an instance of System::RuntimeType is returned.

If you examine the program's output, you will see that each type in the assembly, CustomerListItem, ICustomer, Customer, and Customers, is found and its metadata is printed out. We can find out the standard attributes and the type from which the class derives for each type through the Attributes and BaseType properties.

The methods associated with the Type class enable you to get the associated fields, properties, interfaces, events, and methods. For example, the Customer type has no interfaces, properties, or events, but it has four fields, three constructors, and the methods inherited from its BaseType System::Object:

- Interfaces:
- Fields:
  - CustomerId
  - FirstName
  - LastName
  - EmailAddress
- Properties:
- Events:
- Constructors:
  - public .ctor(System.String first, System.String last,
    System.String email)
  - public .ctor()
  - public .ctor(System.Int32 id)
- Methods:
  - public Int32 GetHashCode()
  - public Boolean Equals(System.Object obj)
  - public String ToString()
  - public Type GetType()

The type Customers inherits from one interface and has one constructor and four of its own methods in addition to the four it inherited from its BaseType System::Object:

- Interfaces:
- ICustomer
- Fields:
- Properties:
- Events:
- Constructors:
Metadata and Reflection

public .ctor()
Methods:
 public Void ChangeEmailAddress(System.Int32 id,  
    System.String emailAddress)
 public ArrayList GetCustomer(System.Int32 id)
 public Void UnregisterCustomer(System.Int32 id)
 public Int32 RegisterCustomer(System.String firstName,  
    System.String lastName, System.String emailAddress)
 public Int32 GetHashCode()
 public Boolean Equals(System.Object obj)
 public String ToString()
 public Type GetType()

These were obtained with the GetInterfaces, GetFields, GetProperties, GetEvents, GetConstructors, and GetMethods methods on the Type class. Since an interface is a type, GetInterfaces returns an array of Types representing the interfaces inherited or implemented by the Type queried. Since fields, properties, events, and methods are not types, their accessor methods do not return Types. Each of their accessor methods returns an appropriate class: FieldInfo, PropertyInfo, EventInfo, ConstructorInfo, and MethodInfo. All these classes, as well as the Type class, inherit from the MemberInfo class that is the abstract base class for member metadata.

Let us examine some of the metadata associated with a class method. Using the reflection methods, we were able to reconstruct the signatures for all the classes and interfaces in the Customer assembly. Here is the output for the methods of the Customers class:

public Void ChangeEmailAddress(System.Int32 id,  
    System.String emailAddress)
 public ArrayList GetCustomer(System.Int32 id)
 public Void UnregisterCustomer(System.Int32 id)
 public Int32 RegisterCustomer(System.String firstName,  
    System.String lastName, System.String emailAddress)
 public Int32 GetHashCode()
 public Boolean Equals(System.Object obj)
 public String ToString()
 public Type GetType()

Here is the code from the example that produced the output:

for (int j = 0; j < methodInfo.Length; j++)
{
    if (MethodInfo[j]->IsStatic)
        Console::Write(" static ");
}
Except that a constructor does not have a return type, the exact same code reconstitutes the calling sequences for the class’s constructors.

The `MethodInfo` class has properties that help us determine if the method is static, public, protected, internal, or private as well as determine the return type and method name. The method parameters are stored in a property array of type `ParameterInfo` class.

This example should also make it clear that types are assembly relative. The same type name and layout in two different assemblies are treated by the runtime as two separate types. When versioning assemblies, you have to be careful when mixing versioned types or the same types in two different assemblies.

All this metadata allows the CLR and the framework to provide services to your applications, because they can understand the structure of your types.

### Late Binding

Reflection can also be used to implement late binding. Late binding is where the method to be called is determined during execution rather than at compi-
lation time. It is one example of how metadata can be used to provide functionality. As the previous example demonstrates, you can extract the signature of a method associated with a type. The `MethodInfo` object has all the needed metadata for a class method. The Dynamic sample demonstrates a very simple example of late binding.³

We dynamically load an assembly and get the metadata for a method of a particular type:

```c++
// Load Customer assembly
Assembly *a = Assembly::Load("Customer");

// Get metadata for Customers class and one method
Type *t = a->GetType("OI.NetCpp.Acme.Customers");
MethodInfo *mi = t->GetMethod("GetCustomer");
```

One thing that C++ programmers need to remember when doing reflection programming is that when you deal with strings that contain namespaces or classes, you must use properly formatted strings that are understood by the reflection class methods. So, the fully qualified class name in the code above is `OI.NetCpp.Acme.Customers` rather than the C++ style format `OI::NetCpp::Acme::Customers`. Thus, the format used is like that of C#, not C++.

Using the reflection classes, we could have made this completely dynamic by arbitrarily picking types, methods, and constructors from the `Customer` assembly using the techniques of the last example, but we wanted to keep the Dynamic example simple. A more ambitious program could do something much more interesting, such as implement an assembly decompiler that generates Managed C++, C#, or VB.NET source code directly from a compiled assembly.

The System namespace has an `Activator` class that has overloaded `CreateInstance` methods to create an instance of any .NET type using the appropriate constructor. The `Activator` class is discussed in this chapter's section on remoting. We invoke a constructor with no arguments to create an instance of the `Customers` object.

```c++
Type *t = a->GetType("OI.NetCpp.Acme.Customers");
...
Object *customerInstance =
   Activator::CreateInstance(t);
```

³ After you build the Dynamic project, you must copy `Customer.dll` from the `Dynamic\Customer\Debug` directory to the `Dynamic\Debug` directory before running `Dynamic.exe`.
We then build an argument list and use the **Invoke** method of the **MethodInfo** instance to call the **GetCustomer** method.

```csharp
// invoke the method
Object *arguments [] = new Object*[1];
int customerId = -1;
arguments[0] = __box(customerId);
Object *returnType = mi->Invoke(
    customerInstance, arguments);
```

Using the reflection methods, we get the type information for each field in a return structure. Note that the **GetValue** method of **FieldInfo** returns the data for a particular field in an object.

```csharp
if (returnType->GetType() ==
    Type::GetType("System.Collections.ArrayList"))
{
    ArrayList *arrayList =
        dynamic_cast<ArrayList*>(returnType);
    for (int i = 0; i<arrayList->Count; i++)
    {
        Type *itemType =
            arrayList->get_Item(i)->GetType();
        FieldInfo *fi [] = itemType->GetFields();
        for (int j = 0; j < fi->Length; j++)
        {
            Object *fieldValue =
                fi[j]->GetValue(arrayList->get_Item(i));
            Console::Write(
                "{0, -10} = {1, -15}",
                fi[j]->Name, fieldValue);
        }
        Console::WriteLine();
    }
}
```

Again, note the use of the single periods rather than double colons in the string `System.Collections.ArrayList`.

This code did not use any specific objects or types from the **Customer** assembly. We did use some knowledge about the assembly to keep the code simple to illustrate the main points. It should be clear, however, how to make this completely general.

You can take this one step further and use the classes that emit metadata (in `System::Reflection::Emit`). You can even dynamically create an assembly in memory, and then load and run it!
Input and Output in .NET

To make a crude generalization, the input/output functions in the .NET Framework can be divided into two broad categories, irrespective of the data storage (disk, memory, etc.) that is being written to or read from.

Data can be treated as a stream of bytes or characters. For example, we could read 500 bytes from a file and write them to a memory buffer. Data can also be treated as a set of objects. Reading and writing the objects is referred to as deserializing and serializing the objects. We can serialize (write) the list of Customer objects to disk. We can then deserialize (read) the list of Customer objects back into memory.

The System::IO namespace has several classes for reading and writing to various types of storage while treating the data as bytes or characters. Serialization functionality can be found in various places in the .NET framework. The System::Runtime::Serialization namespace handles serialization of the Common Type System. The System::Xml::Serialization namespace handles XML serialization.

Stream Classes

Stream is an abstract class that is the basis for reading from and writing bytes to some storage such as a file. It supports both synchronous and asynchronous reading and writing. Asynchronous methods are discussed later in this chapter. The Stream class has the typical methods that you would expect: Read, Write, Seek, Flush, and Close.

The FileStream class is derived from Stream to represent the reading and writing of files as a series of bytes. The FileStream constructor builds the actual stream instance. The overridden Stream methods implement the reading and writing to the file.

Other classes derived from Stream include MemoryStream, BufferedStream, and NetworkStream (in System::Net::Sockets).

The FileStream example (in the FileIO directory with the IO examples) illustrates how to use the Stream classes. If the file does not exist, a new file is created and the numbers from 0 to 9 are written to the file. If the file already exists, the code starts reading 5 bytes from the end of the file and then writes them out. (You should run the example twice. The first time creates and writes the file, and the second time reads and displays the file.)

```csharp
unsigned char data __gc[] =
    new unsigned char __gc [10];
FileStream *fs = new FileStream(
    "FileStreamTest.txt", FileMode::OpenOrCreate);
if (fs->Length == 0)
{
    Console::WriteLine("Writing Data...");
```
Chapter 8  • .NET Framework Classes

for (short i = 0; i < 10; i++)
    data[i] = (unsigned char)i;
fs->Write(data, 0, 10);
}
else
{
    fs->Seek(-5, SeekOrigin::End);
    int count = fs->Read(data, 0, 10);
    for (int i = 0; i < count; i++)
    {
        Console::WriteLine(data[i]);
    }
}
fs->Close();

Primitive Data Types and Streams

The Stream-derived classes work well if you are reading and writing bytes of data as a block. If you need to read and write the primitive common types, such as Boolean, String, and Int32, in and out of a stream, you should use the BinaryReader and the BinaryWriter classes. The Binary example in the FileIO directory shows how to use these classes. You create the appropriate stream (FileStream in the example) and pass it to the BinaryReader or BinaryWriter constructor. You can then use one of the overloaded Read or Write methods to read or write a data type to or from the stream. (Again, you should run the example twice. The first time creates and writes the file, and the second time reads the file.)

FileStream *fs = new FileStream("BinaryTest.bin", FileMode::OpenOrCreate);
if (fs->Length == 0)
{
    Console::WriteLine("Writing Data...");
    BinaryWriter *w = new BinaryWriter(fs);
    for (short i = 0; i < 10; i++)
        w->Write(i);
    w->Close();
}
else
{
    BinaryReader *r = new BinaryReader(fs);
    for (int i = 0; i < 10; i++)
        Console::WriteLine(r->ReadInt16());
    r->Close();
}
TextReader and TextWriter

The TextReader and TextWriter abstract classes treat the data as a sequential stream of characters (i.e., as text). TextReader has methods such as Close, Peek, Read, ReadBlock, ReadLine, and ReadToEnd. TextWriter has methods such as Close, Flush, Write, and WriteLine. The overloaded Read methods read characters from the stream. The overloaded Write and WriteLine methods write various types to the stream. If an object is written to the stream, the object's ToString method is used.

StringReader and StringWriter are derived from TextReader and TextWriter, respectively. StringReader and StringWriter read and write data in a character string, which is stored in an underlying StringBuilder object. The StringWriter constructor can take a StringBuilder object. The StringBuilder class was discussed in Chapter 3.

StreamReader and StreamWriter are also derived from TextReader and TextWriter. They read and write text to and from a Stream object. As with the BinaryReader and BinaryWriter class, you can create a Stream object and pass it to the StreamReader or StreamWriter constructor. Hence, these classes can use any Stream-derived class data storage. The Text example in the FileIO directory uses the StreamWriter and StreamReader classes. Run the program twice, first to create the file, and then to read it.

```cpp
FileStream *fs = new FileStream(
    "TextTest.txt", FileMode::OpenOrCreate);
if (fs->Length == 0)
{
    Console::WriteLine("Writing Data...");
    StreamWriter *sw = new StreamWriter(fs);
    sw->Write(100);
    sw->WriteLine(" One Hundred");
    sw->WriteLine("End of File");
    sw->Close();
}
else
{
    String *text;
    StreamReader *sr = new StreamReader(fs);
text = sr->ReadLine();
    while (text != 0)
    {
        Console::WriteLine(text);
    }
}
fs->Close();
```
File Manipulation

The framework has two classes named File and FileInfo that are very useful for working with files. If you need to manipulate the file in addition to reading and writing to it, the File class provides the basic functionality. Since the File class just has static members, you have to provide the name of the file as an argument. The FileInfo class has a constructor that creates an object that represents a file. You then use the methods to manipulate that particular file.

The File class methods always perform a security check. If you are going to continually access a particular file, you may want to use the FileInfo class because the security check is made only once in the constructor. Security is discussed in more detail in Chapter 13.

FILE CLASS

The File class has methods for creating and opening files that return FileStream, StreamWriter, or StreamReader objects that do the actual reading and writing. The overloaded Create methods return a FileStream object. The CreateText method returns a StreamWriter. The overloaded Open method can either create a new file or open an existing one for reading or writing, depending on the method parameters. The object returned is a FileStream object. The OpenText method returns a StreamReader. The OpenRead method returns a FileStream object. The OpenWrite method returns a FileStream.

The File class also has methods for copying, deleting, and moving files. You can also test for the existence of a file. File attributes such as the following can be read or modified:

- creation time
- last access time
- last write time
- archive, hidden, normal, system, or temporary
- compressed, encrypted
- read-only
- is the file a directory?

PATH CLASS

Many of the file names needed for input arguments have to be full paths. Or, you might want to manipulate only parts of the path. The Path class has static
methods that make this easier. The Path class has static fields that indicate various platform-specific aspects of pathnames, such as the separator characters for directories, paths, and volumes and the illegal characters for pathnames.

Its static methods let you change the extension of a file or find the directory where temporary files reside. The GetFullPath method is particularly useful. You can pass it a relative path, such as \foo.txt, and it will return the full path of the file. This is very useful for the File or security classes that require the full file path.

FILEINFO CLASS

The FileInfo constructor creates an object that represents a disk file. The constructor takes one argument, a string representing the name of the file. The class has properties that represent file properties such as the creation time, full pathname and the size of the file. It has creation and open methods that are analogous to the File class methods, but operate on this file instance and therefore do not need a filename parameter. The FileInfo class also has methods to move and copy the file.

FILE EXAMPLE

The File example in the FileIO directory illustrates the use of the File and FileInfo classes. In this example, the static Delete method of the File class is used to remove a specified file. The static CreateText method then creates a new file and returns a StreamWriter instance that is used to write some text to the file. The stream is then closed, and the static Move method then renames the file. A FileInfo instance is constructed to represent this renamed file. The complete file name, size, and creation date for the file are written to the console. The file is opened as text and a StreamReader instance is used to read and write out the contents of the file.

```csharp
File::Delete("file2.txt");

StreamWriter *sw = 
    System::IO::File::CreateText("file.txt");
sw->Write("The time has come the Walrus said, ");
sw->WriteLine("to talk of many things.");
sw->Write("Of shoes, and ships, and sealing wax, ");
sw->WriteLine("of cabbages and kings.");
sw->Write("And why the sea is boiling hot, ");
sw->WriteLine("and whether pigs have wings.");
sw->Close();

File::Move("file.txt", "file2.txt");
```
FileInfo *fileInfo = new FileInfo("file2.txt");

Console::WriteLine("File (0) is (1) bytes in length, created on (2)",
    __box(fileInfo->FullName),
    __box(fileInfo->Length),
    __box(fileInfo->CreationTime));
Console::WriteLine("\n");

StreamReader *sr = fileInfo->OpenText();
String *s = sr->ReadLine();
while (s != 0)
{
    Console::WriteLine(s);
    s = sr->ReadLine();
}
sr->Close();
Console::WriteLine("\n");

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. Serialization converts objects, such as classes, structs, and arrays, to a byte stream. Deserialization converts the byte stream back into the objects. Serializing and deserializing can be done on different machines as long as they both host the CLR.

Objects can be serialized without writing special code because, as we have seen, the runtime can query the object’s metadata to allow it to understand the memory layout of the object. To inform the framework that a class can be serialized, mark the class with the System::Serializable attribute. Any field or property that should not be serialized can be marked with the System::NonSerialized attribute. For example, fields that represent cached values need not be serialized. All you have to do is mark the class with the serializable attribute, and you then do not have to write any other code to serialize the object’s fields.

The Serialization example shows how to apply serialization to the case study’s HotelBroker class in the Hotel assembly. The Serializable attribute
has been applied to the **HotelBroker** class definition. The **Serializable** attribute has also been applied to all the classes that get used by **HotelBroker** or that **HotelBroker** derives from—**Broker**, **Hotel**, **HotelReservation**, **Reservable**, and **Reservation**—because in order for **HotelBroker** to be serializable, those classes must be serializable as well. If any of those classes were not marked, a runtime exception would be thrown when the framework tries to serialize an object of that type.

```csharp
[Serializable]
public __gc class HotelBroker :
    public Broker,
    public IHotelInfo,
    public IHotelAdmin,
    public IHotelReservation
{
private:
    const int MAXDAY;
    const int MAXUNIT;
    [NonSerialized] ArrayList *cities;
    ...
};

[Serializable]
public __gc class Hotel : public Reservable
{
    ...
};

[Serializable]
public __gc class HotelReservation : public Reservation
{
    ...
};

[Serializable]
public __gc __abstract class Reservable
{
    ...
};
```

---

4 After you build the **Serialization** project, you must copy **Hotel.dll** from the **Serialization\Hotel\Debug** directory to the **Serialization\Debug** directory before running **Serialization.exe**.
public __gc __abstract class Reservation
{

...
}

[Serializable]
public __gc __abstract class Broker
{

...
}

The cities field has been marked as NonSerialized, since the hotel's city is saved with the serialized hotels, and therefore can be restored as the modified AddCity method demonstrates. The cities field would be null if the HotelBroker class had been deserialized, because the cities field was not saved.5

private:
void AddCity(String *city)
{
  if (cities == 0)
  {
    cities = new ArrayList;
    IEnumerator *pEnum = units->GetEnumerator();
    while (pEnum->MoveNext())
    {
      Hotel *h =
      dynamic_cast<Hotel *>(pEnum->Current);
      AddCity(h->City);
    }
  }
  // check if city already on list, add if not
  if (!cities->Contains(city))
    cities->Add(city);
}

Serialization Objects

Although the framework knows how to save an object marked with the Serializable attribute, you still have to specify the format in which the object is

5 Of course we could have serialized the cities field and not have to deal with the case where cities could be null, but we wanted to demonstrate the NonSerialized attribute.
saved and the storage medium. To specify the format that an object is saved in, you use an instance of an object that supports the `IFormatter` interface.\(^6\)

The Framework ships with two such classes: `System::Runtime::Serialization::Formatters::Binary::BinaryFormatter` and `System::Runtime::Serialization::Formatters::Soap::SoapFormatter`. The `BinaryFormatter` uses a binary, compact format for serializing and deserializing on platforms that support the CLR. The `SoapFormatter` uses the industry standard SOAP protocol that is discussed in Chapter 11, “Web Services.” Since it is an XML-based, and therefore text-based, protocol, it can be used to communicate with a non-CLR-based platform. The binary format is faster when serializing and deserializing data.

You can, of course, implement your own formatter classes. You might do this if you have to talk to a foreign system with its own persistent object byte format.

The `Serialization` example has code to demonstrate saving and restoring both binary and SOAP formats using a `FileStream`. Of course you could use any `Stream`-based class representing some data medium. Notice the special care taken to ensure that the `Load` method is able to modify the parameter that points to the `HotelBroker`. To enable this, the parameter is declared to be a reference to a pointer to a `HotelBroker`.

```csharp
static void Save(HotelBroker *broker, String *formatter)
{
    FileStream *s;
    if (String::Equals(formatter, "b"))
    {
        s = new FileStream("hotels.bin", FileMode::Create);
        BinaryFormatter *b = new BinaryFormatter;
        b->Serialize(s, broker);
    }
    else
    {
        s = new FileStream("hotels.txt", FileMode::Create);
        SoapFormatter *sf = new SoapFormatter;
        sf->Serialize(s, broker);
    }
    s->Close();
}
```

\(^6\) How does the runtime know if a class supports the `IFormatter` interface? It queries the metadata!
static void Load(
    HotelBroker *&broker, /*ref to pointer */
    String *formatter)
{
    FileStream *s;
    if (String::Equals(formatter, "b"))
    {
        s = new FileStream("hotels.bin", FileMode::Open);
        BinaryFormatter *b = new BinaryFormatter;
        broker =
            dynamic_cast<HotelBroker *>(b->Deserialize(s));
    }
    else
    {
        s = new FileStream("hotels.txt", FileMode::Open);
        SoapFormatter *sf = new SoapFormatter;
        broker =
            dynamic_cast<HotelBroker *>(sf->Deserialize(s));
    }
    s->Close();
    ShowHotelList(broker->GetHotels());
}

Here is some sample output from the Serialization example: First, we add a hotel and save it with the SOAP formatter. We then exit the program.

Enter command: cities
Atlanta
Boston
Commands: quit, cities, list, add, fetch, save

Enter command: list
City       Name     Rooms  Rate
Atlanta    Dixie    100    115
Atlanta    Marriott 500    70
Boston     Sheraton 250    95
Commands: quit, cities, list, add, fetch, save

Enter command: add
Hotel City: Philadelphia
Hotel Name: Franklin
Number Rooms: 100
Room Rate: 200
Commands: quit, cities, list, add, fetch, save

Enter command: save
Formatter: b(binary), s(soap)s
Commands: quit, cities, list, add, fetch, save

Enter command: cities
Atlanta
Boston
Philadelphia
Commands: quit, cities, list, add, fetch, save

Enter command: list

<table>
<thead>
<tr>
<th>City</th>
<th>Name</th>
<th>Rooms</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atlanta</td>
<td>Dixie</td>
<td>100</td>
<td>115</td>
</tr>
<tr>
<td>Atlanta</td>
<td>Marriot</td>
<td>500</td>
<td>70</td>
</tr>
<tr>
<td>Boston</td>
<td>Sheraton</td>
<td>250</td>
<td>95</td>
</tr>
<tr>
<td>Philadelphia</td>
<td>Franklin</td>
<td>100</td>
<td>200</td>
</tr>
</tbody>
</table>

Commands: quit, cities, list, add, fetch, save

Enter command: quit

We then run the program again and restore what we saved in the first run.

Enter command: cities
Atlanta
Boston
Commands: quit, cities, list, add, fetch, save

Enter command: list

<table>
<thead>
<tr>
<th>City</th>
<th>Name</th>
<th>Rooms</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atlanta</td>
<td>Dixie</td>
<td>100</td>
<td>115</td>
</tr>
<tr>
<td>Atlanta</td>
<td>Marriot</td>
<td>500</td>
<td>70</td>
</tr>
<tr>
<td>Boston</td>
<td>Sheraton</td>
<td>250</td>
<td>95</td>
</tr>
</tbody>
</table>

Commands: quit, cities, list, add, fetch, save

Enter command: fetch
Formatter: b(binary), s(soap)s

<table>
<thead>
<tr>
<th>City</th>
<th>Name</th>
<th>Rooms</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atlanta</td>
<td>Dixie</td>
<td>100</td>
<td>115</td>
</tr>
<tr>
<td>Atlanta</td>
<td>Marriot</td>
<td>500</td>
<td>70</td>
</tr>
</tbody>
</table>

7 If you look at the hotels.txt file, you will see a huge file with a lot of “empty” entries. This stems from the simplistic array data structure we used for reservations, which is very sparse.
Sometimes the serialization provided by the framework is not satisfactory. You can provide custom serialization for a class by implementing the `ISerializable` interface and adding a constructor to the class, as shown in the `Serialization` project in the `ISerializable` directory. The `ISerializable` interface has one member: `GetObjectData`. This method is used when data is serialized.

The `ISerializable` example demonstrates how this is done. As before, the class has to be marked as `Serializable`.

```csharp
[Serializable]
public __gc class HotelBroker :
    public Broker,
    public IHotelInfo,
    public IHotelAdmin,
    public IHotelReservation,
    public ISerializable
{
    ...
};
```

The `SerializationInfo` class is used to store all the data that needs to be saved in the `ISerializable::GetObjectData` method. The `AddValue` method of `SerializationInfo` is overloaded to handle the saving of various types, including `Object*`. When you save the type, you provide a name so that it can be restored later. The `StreamingContext` class gives you information

---

8. Again, after you build the `Serialization` project, you must copy `Hotel.dll` from the `Serialization\Hotel\Debug` directory to the `Serialization\Debug` directory before running `Serialization.exe`.

9. Some of the `AddValue` overloads are not CLS-compliant when the types being saved are not CLS-compliant types, such as unsigned integers. Be careful not to use those types where .NET language interoperability is required. You have to watch for this in other places in the framework, such as the `Convert` class or the `Parse` methods of the various CTS types, or any other place where data is formatted, converted, read, or written (such as the `TextWriter` classes).
about the stream being used in the serialization. For example, you can find out if the stream being used is a file or is being remoted to another computer.

```c++
public:
    void GetObjectData(SerializationInfo *info, StreamingContext context)
    {
        long numberHotels = units->Count;
        info->AddValue("NumberHotels", numberHotels);
        info->AddValue("Hotels", units);
    }
```

You also have to implement a special constructor that is used by the framework when the object is deserialized. It has the same arguments as `GetObjectData` has. Here you use the various `GetXXX` methods on `SerializationInfo` to restore the data. Note that since we did not save the `cities` field, we had to manually restore it. The constructor is private because only the framework uses it. If you forget to add the constructor, you will get a `SerializationException` when you try to restore the object.

```c++
private:
    HotelBroker(SerializationInfo *info, StreamingContext context)
    : Broker(366, 10), MAXDAY(366), MAXUNIT(10)
    {
        long numberHotels =
            info->GetInt32("NumberHotels");
        units = dynamic_cast<ArrayList *>(
            info->GetValue("Hotels", units->GetType()));
        if (numberHotels == units->Count)
            Console::WriteLine("All hotels deserialized.");
        else
            Console::WriteLine("Error in deserialization.");
        cities = new ArrayList;
        IEnumerator *pEnum = units->GetEnumerator();
        while (pEnum->MoveNext())
        {
            Hotel *h =
                dynamic_cast<Hotel *>(pEnum->Current);
            AddCity(h->City);
        }
    }
```
Remember, after you make any changes and build the Hotel project, you will need to copy the `Hotel.dll` to the directory of the `Serialization.exe` client program. This does not get done automatically, as in C#, unless you add a custom build step to the project.

In this example we only did custom serialization for the `HotelBroker` object. For all the other objects, we still relied on the framework's serialization. This example works the same way that the `Serialization` example did. The sample output would therefore look the same.

**.NET Application Model**

Serialization gave you a concrete example of the flexible environment the .NET Framework provides for writing code. Now let us take a look at the model in which .NET applications run. The Win32 environment in which a program runs is called its process. This environment consists of:

- An address space in which the code and data of the program resides.
- A set of environment variables that is associated with the program.
- A current drive and directory.
- One or more threads.

**Threads**

A thread is the actual execution path of a program's code. One or more threads run inside of a process to allow for multiple execution paths inside of a process. For example, with multiple threads a program can update the user interface with partial results on one thread as a calculation proceeds on another thread. All threads in the same process share the process environment so that all the threads can access the process memory.

Threads are scheduled by the operating system. Processes and application domains are not scheduled. Threads are periodically given a limited timeslice in which to run so that they can share the processor with other threads. Higher priority threads will get to run more often than lower priority threads. After some time elapses, a thread will get another chance to run. When a thread is swapped back in, it resumes running from the point it was at when it was swapped out.

Threads maintain a context that is saved and restored when the operating system's scheduler switches from one thread to another. A thread's context includes the CPU registers and stack that contain the state of the executing code.

---

10 Application domains are discussed later in this chapter.
The `System::Threading::Thread` class models an executing thread. The `Thread` object that represents the current executing thread can be found from the static property `Thread::CurrentThread`.

Unless your code runs on a multiprocessor machine, or you are trying to use time while a uniprocessor waits for some event such as an I/O event, multiple threads do not result in any time saved for your computing tasks. It does, however, make the system seem more responsive to tasks requiring user interaction. Using too many threads can decrease performance, as thread management overhead and contention between competing threads burdens the CPU.

To help you understand threads, we have a multiple-step `Threading` example\(^{11}\) that uses the `Customer` and `Hotel` assemblies from the case study to make reservations. Let us look first at Step 0.

.NET threads run as delegates defined by the `ThreadStart` class. The delegate returns void and takes no parameters.

```csharp
public __gc __delegate void ThreadStart();
```

The `NewReservation` class has a public member function, `MakeReservation`, that will define the thread function. Since the thread function takes no parameters, any data that this function uses is assigned to fields in the `NewReservation` instance.

The thread delegate is created and passed as a parameter to the constructor that creates the `Thread` object. The `Start` method on the `Thread` object is invoked to begin the thread’s execution. When we discuss the asynchronous programming model, we will show you how to pass parameters to a thread delegate. The program now has two threads: the original one that executed the code to start the thread and now the thread we just created that attempts to make a hotel reservation.

```csharp
public __gc class NewReservation
{
    ...
    public:
    ...
    void MakeReservation()
    {
        Console::WriteLine(
            "Thread {0} starting.",
            Thread::CurrentThread->
            GetHashCode().ToString());
    ...
```

\(^{11}\) In each step, you will need to copy the `Customer` and `Hotel` DLLs into the `Debug` directory of `TestThreading`
ReservationResult *result =
    hotelBroker->MakeReservation(
        customerId, city, hotel, date, numberDays);
    ...
};

Then, in the Main method, the following code starts the thread, using a delegate:

NewReservation *reserve1 =
    new NewReservation(customers, hotelBroker);

reserve1->customerId = 1;
reserve1->city = "Boston";
reserve1->hotel = "Presidential";
reserve1->sdate = "12/12/2001";
reserve1->numberDays = 3;

// create delegate for threads
ThreadStart *threadStart1 = new ThreadStart(
    reserve1,
    reserve1->MakeReservation);

Thread *thread1 = new Thread(threadStart1);
Console::WriteLine(
    "Thread \{0\} starting a new thread.",
    Thread::CurrentThread->
    GetHashCode().ToString());
thread1->Start();

To cause the original thread to wait until the second thread is done, the Join method on the Thread object is called. The original thread now blocks (waits) until the reservation thread is complete. The results of the reservation request are written to the console by the reservation thread.

    // Block this thread until the worker thread is done
    thread1->Join();
    Console::WriteLine("Done!");

THREAD SYNCHRONIZATION

An application can create multiple threads. Look at the code in Step 1 of the Threading example. Now multiple reservation requests are being made simultaneously.
NewReservation *reserve1 =
   new NewReservation(customers, hotelBroker);
...

NewReservation *reserve2 =
   new NewReservation(customers, hotelBroker);
...

// create delegate for threads
ThreadStart *threadStart1 = new ThreadStart(
   reserve1,
   reserve1->MakeReservation);
ThreadStart *threadStart2 = new ThreadStart(
   reserve2,
   reserve2->MakeReservation);

Thread *thread1 = new Thread(threadStart1);
Thread *thread2 = new Thread(threadStart2);

Console::WriteLine(
   "Thread {0} starting a new thread.",
   Thread::CurrentThread->
   GetHashCode().ToString());
thread1->Start();
thread2->Start();

// Block this thread until the worker thread is done
thread1->Join();
thread2->Join();

The problem with our reservation system is that there is no guarantee
that one thread will not interfere with the work being done by the other
thread. Since threads run for only a small period before they give up the pro-
cessor to another thread, they may not be finished with whatever operation
they were working on when their timeslice is up. For example, they might be
in the middle of updating a data structure. If another thread tries to use the
information in that data structure or to update the data structure, the results
of those operations will be inconsistent and incorrect, or a program crash could
result (i.e., if references to obsolete structures were not yet updated, an
unhandled exception could occur).

Let us look at one of several places in the customer and reservation
code where we could have a problem. Examine the code for Broker::Reserve in Broker.h. First, a check is made of the existing bookings for
a given hotel for a given date to see if there are rooms available. Then, if a
room is available, it is reserved. Note that we have added a call to 
Thread::Sleep between the code that checks for room availability and the 
code that reserves a room, which will be explained shortly.

... 

// Check if rooms are available for all dates 
for (int i = day; i < day + numDays; i++) 
{ 
    if (numCust[i, unitid] >= unit->capacity) 
    { 
        result->ReservationId = -1; 
        result->Comment = "Room not available"; 
        return result; 
    } 
} 

Console::WriteLine(
    "Thread {0} finds the room is available in 
    Broker::Reserve",
    Thread::CurrentThread->
    GetHashCode().ToString());

Thread::Sleep(0);

// Reserve a room for requested dates 
for (int i = day; i < day + numDays; i++) 
    numCust[i, unitid] += 1;
...

This code can produce inconsistent results! One of the threads could be 
swapped out after it finds that the last room is available, but before it gets a 
chance to make the booking. The other thread could then run, find the same 
available room and make the booking. When the first thread runs again, it starts 
from where it left off, and it will also book the same last room at the hotel.

To simulate this occurrence, this step (Step 1) of the Threading example puts a 
Thread::Sleep call between the code that checks for room availability and the code that makes the booking. The Sleep(0) call will cause the 
thread to stop executing and give up the remainder of its timeslice.

To ensure that we see the thread contention problem occur, we have only one room in the entire hotel! We then set up our program so that the two

---

You can see that special care must be taken to make this threading problem manifest itself consistently to make a good example! In real programming, such a problem typically expresses itself inconsistently, which makes it all the more difficult to detect during testing. Of course, it is certain to express itself sooner or later after it has been deployed.
threads try to reserve the only room at the hotel for the same date. Examine the code in the Main routine that sets this up:

```csharp
hotelBroker->AddHotel(
    "Boston",
    "Presidential",
    1, //only one room in entire hotel!
    (Decimal) 10000);
...
NewReservation *reserve1 =
    new NewReservation(customers, hotelBroker);
reserve1->customerId = 1;
reserve1->city = "Boston";
reserve1->hotel = "Presidential";
reserve1->sdate = "12/12/2001";
reserve1->numberDays = 3;

NewReservation *reserve2 =
    new NewReservation(customers, hotelBroker);
reserve2->customerId = 2;
reserve2->city = "Boston";
reserve2->hotel = "Presidential";
reserve2->sdate = "12/13/2001";
reserve2->numberDays = 1;

Running the program will give results that look something like this:

Added Boston Presidential Hotel with one room.
Thread 3 starting a new thread.
Thread 5 starting.
Thread 6 starting.
Reserving for Customer 2 at the Boston Presidential Hotel on 12/13/2001 12:00:00 AM for 1 days
Reserving for Customer 1 at the Boston Presidential Hotel on 12/12/2001 12:00:00 AM for 3 days
Thread 6 entered Broker::Reserve
Thread 6 finds the room is available in Broker::Reserve
Thread 5 entered Broker::Reserve
Thread 5 finds the room is available in Broker::Reserve
Thread 6 left Broker::Reserve
Reservation for Customer 2 has been booked
ReservationId = 1
Thread 5 left Broker::Reserve
Reservation for Customer 1 has been booked
ReservationId = 2
ReservationRate = 10000
ReservationCost = 30000
Comment = OK
ReservationRate = 10000
ReservationCost = 10000
Comment = OK
Done!

Unfortunately, both customers get to reserve the last (only) room on December 13! Note how one thread enters the `Reserve` method and finds the room is available before it gets swapped out. Then, the other thread enters Reserve and also finds the room is available before it gets swapped out. Both threads then book the same room.

Operating systems provide means for synchronizing the operation of multiple threads or multiple processes accessing shared resources. The .NET Framework provides several mechanisms to prevent threading conflicts.

Every object in the .NET Framework can be used to provide a synchronized section of code (critical section). Only one thread at a time can execute within such a section. If one thread is already executing inside that synchronized code section, any other threads that attempt to access that section will block (wait) until the executing thread leaves that code section.

**SYNCHRONIZATION WITH MONITORS**

The `System::Threading::Monitor` class allows threads to synchronize on an object to avoid the data corruption and indeterminate behavior that can result from race conditions. Step 2 of the Thread example demonstrates the use of the Monitor class with the `this` pointer of the HotelBroker instance.

```
ReservationResult *Reserve(Reservation *res)
{
    Console::WriteLine(
        "Thread {0} trying to enter Broker::Reserve",
        Thread::CurrentThread->
            GetHashCode().ToString());
    Monitor::Enter(this);
    ... (thread contentious code goes here)
    Monitor::Exit(this);
    return result;
}
```

The thread that first calls the `Monitor::Enter(this)` method will be allowed to execute the code of the `Reserve` method because it will acquire the Monitor lock based on the `this` pointer. Subsequent threads that try to
execute the same code will have to wait until the first thread releases the lock with `Monitor::Exit(this)`. At that point they will be able to call `Monitor::Enter(this)` and acquire the lock.

A thread can call `Monitor::Enter` several times, but each call must be balanced by a call to `Monitor::Exit`. If a thread wants to try to acquire a lock, but does not want to block, it can use the `Monitor::TryEnter` method.

Now that we have provided synchronization, the identical case tried in Step 1 does not result in one reservation too many for the hotel. Notice how the second thread cannot enter the `Reserve` method until the first thread that entered has left. This solves our thread contention problem and enforces the rule that the same room cannot be booked by two customers for the same date.

Added Boston Presidential Hotel with one room.
Thread 3 starting a new thread.
Thread 5 starting.
Thread 6 starting.
Reserving for Customer 2 at the Boston Presidential Hotel on 12/13/2001 12:00:00 AM for 1 days
Thread 6 trying to enter Broker::Reserve
Thread 6 entered Broker::Reserve
Thread 6 finds the room is available in Broker::Reserve
Thread 6 left Broker::Reserve
Reservation for Customer 2 has been booked
ReservationId = 1
Reserving for Customer 1 at the Boston Presidential Hotel on 12/12/2001 12:00:00 AM for 3 days
Thread 5 trying to enter Broker::Reserve
Thread 5 entered Broker::Reserve
Reservation for Customer 1 could not be booked
Room not available
ReservationRate = 10000
ReservationCost = 10000
Comment = OK
Done!

**NOTIFICATION WITH MONITORS**

A thread that has acquired a `Monitor` lock can wait for a signal from another thread that is synchronizing on that same object without leaving the synchronization block. The thread invokes the `Monitor::Wait` method and relinquishes the lock. When notified by another thread, it reacquires the synchronization lock.

A thread that has acquired a `Monitor` lock can send notification to another thread waiting on the same object with the `Pulse` or the `PulseAll`
methods. It is important that the thread be waiting when the pulse is sent; otherwise, if the pulse is sent before the wait, the other thread will wait forever and will never see the notification. This is unlike the reset events discussed later in this chapter. If multiple threads are waiting, the \textbf{Pulse} method will put only one thread on the ready queue to run. The \textbf{PulseAll} will put all of them on the ready queue.

The pulsing thread no longer has the \textbf{Monitor} lock, but is not blocked from running. Since it is no longer blocked, but does not have the lock, to avoid a deadlock or race condition, this thread should try to reacquire the lock (through a \textbf{Monitor::Enter} or \textbf{Wait}) before doing any potentially damaging work.

The \textbf{PulseAll} example illustrates the \textbf{Pulse} and \textbf{PulseAll} methods. Running the example produces the following output:

First thread: 2 started.
Thread: 5 started.
Thread: 5 waiting.
Thread: 6 started.
Thread: 6 waiting.
Thread 5 sleeping.
Done.
Thread 5 awake.
Thread: 5 exited.
Thread 6 sleeping.
Thread 6 awake.
Thread: 6 exited.

The class \textbf{X} has a field “\textit{o}” of type \textbf{Object} that will be used for a synchronization lock.

The class also has a method \textbf{Test} that will be used as a thread delegate. The method acquires the synchronization lock, and then waits for a notification. When it gets the notification, it sleeps for half a second, and then relinquishes the lock.

The main method creates two threads that use the \textbf{X::Test} method as their thread delegate and share the same object to use for synchronization. It then sleeps for 2 seconds to allow the threads to issue their wait requests and relinquish their locks. It then calls \textbf{PulseAll} to notify both waiting threads and relinquishes its hold on the locks. Eventually, each thread will reacquire the lock, write a message to the console, and relinquish the lock for the last time.

\begin{verbatim}
__gc class X
{
private:
  Object *o;
\end{verbatim}
public:
    X(Object *o)
    {
        this->o = o;
    }

    void Test()
    {
        try
        {
            long threadId =
                Thread::CurrentThread->GetHashCode();
            Console::WriteLine(
                "Thread: {0} started.", threadId.ToString());
            Monitor::Enter(o);
            Console::WriteLine(
                "Thread: {0} waiting.", threadId.ToString());
            Monitor::Wait(o);
            Console::WriteLine(
                "Thread (0) sleeping.", threadId.ToString());
            Thread::Sleep(500);
            Console::WriteLine(
                "Thread (0) awake.", threadId.ToString());
            Monitor::Exit(o);
            Console::WriteLine(
                "Thread: {0} exited.", threadId.ToString());
        }
        catch(Exception *e)
        {
            long threadId =
                Thread::CurrentThread->GetHashCode();
            Console::WriteLine(
                "Thread: {0} Exception: {1}",
                threadId.ToString(), e->Message);
            Monitor::Exit(o);
        }
    }
};

__gc class Class1
{
    public:
        static Object *o = new Object;
        static void Main()
        {

Console::WriteLine("First thread: {0} started.",
    Thread::CurrentThread->GetHashCode().ToString());

X *a = new X(o);
X *b = new X(o);

ThreadStart *ats = new ThreadStart(a, X::Test);
ThreadStart *bts = new ThreadStart(b, X::Test);

Thread *at = new Thread(ats);
Thread *bt = new Thread(bts);

at->Start();
bt->Start();

// Sleep to allow other threads to wait on the
// object before the Pulse
Thread::Sleep(2000);
Monitor::Enter(o);

Monitor::PulseAll(o);
//Monitor::Pulse(o);
Monitor::Exit(o);

Console::WriteLine("Done.");
};

Comment out the PulseAll call and uncomment the Pulse call, and only
one thread completes because the other thread is never put on the ready
queue. Remove the Sleep(2000) from the main routine, and the other threads
block forever because the pulse occurs before the threads get a chance to call
the Wait method, and hence they will never be notified. These methods can
be used to coordinate several threads’ use of synchronization locks.

The Thread::Sleep method causes the current thread to stop execution
(block) for a given time period. Calling Thread::Suspend will cause the
thread to block until Thread::Resume is called by another thread on that
same thread. Threads can also block because they are waiting for another
thread to finish (Thread::Join). This method was used in the Threading
examples so that the main thread could wait until the reservation requests
were completed. Threads can also block because they are waiting on a syn-
chronization lock (critical section).
Calling `Thread::Interrupt` on the blocked thread can awaken it. The thread will receive a `ThreadInterruptedException`. If the thread does not catch this exception, the runtime will catch the exception and kill the thread.

If, as a last resort, you have to kill a thread outright, call the `Thread::Abort` method on the thread. `Thread::Abort` causes the `ThreadAbortException` to be thrown. This exception cannot be caught, but it will cause all the `finally` blocks to be executed. In addition, `Thread::Abort` does not cause the thread to wake up from a wait.

Since `finally` blocks may take a while to execute, or the thread might be waiting, aborted threads may not terminate immediately. If you need to be sure that the thread has finished, you should wait on the thread's termination using `Thread::Join`.

SYNCHRONIZATION CLASSES

The .NET Framework has classes that represent the standard Win32 synchronization objects. These classes all derive from the abstract `WaitHandle` class. This class has static methods, `WaitAll` and `WaitAny`, that allow you to wait on a set or just one of a set of synchronization objects being signaled. It also has an instance method, `WaitOne`, that allows you to wait for this instance to be signaled. How the object gets signaled depends on the particular type of synchronization object that is derived from `WaitHandle`.

A `Mutex` object is used for interprocess synchronization. `Monitors` and synchronized code sections work only within one process. An `AutoResetEvent` and `ManualResetEvent` are used to signal whether an event has occurred. An `AutoResetEvent` remains signaled until a waiting thread is released. A `ManualResetEvent` remains signaled until its state is set to unsignaled with the `Reset` method. Hence, many threads could be signaled by this event. Unlike `Monitors`, code does not have to be waiting for the signal before the pulse is set for the reset events to signal a thread.

The framework has provided classes to solve some standard threading problems. The `Interlocked` class methods allow atomic operations on shared values such as increment, decrement, comparison, and exchange. `ReaderWriterLock` is used to allow single-writer, multiple-reader access to data structures. The `ThreadPool` class can be used to manage a pool of worker threads.

AUTOMATIC SYNCHRONIZATION

You can use attributes to synchronize the access to instance methods and fields of a class. Access to static fields and methods is not synchronized. To do this, you derive the class from the class `System::ContextBoundObject` and apply a `Synchronization` attribute to the class. This attribute cannot be applied to an individual method or field.
The attribute is found in the `System::Runtime::Remoting::Contexts` namespace. It describes the synchronization requirements of an instance of the class to which it is applied. You can pass one of four values, which are static fields of the `SynchronizationAttribute` class, to the `SynchronizationAttribute` constructor: NOT_SUPPORTED, SUPPORTED, REQUIRED, REQUIRES_NEW. The Threading example Step 3 illustrates how to do this.

```csharp

using namespace System::Runtime::Remoting::Contexts;

//SynchronizationAttribute::REQUIRED is 4
[Synchronization(4)]
public __gc __abstract class Broker :
    public ContextBoundObject
{
    ...
}
...
```

In order for the CLR to make sure that the thread in which an object’s methods run is synchronized properly, the CLR has to track the threading requirements of the object. This state is referred to as the context of the object. If one object needs to be synchronized, and another does not, they are in two separate contexts. The CLR has to acquire a synchronization lock on behalf of the code when a thread that is executing a method on the object that does not need to be synchronized starts executing a method on an object that does. The CLR knows that this has to be done because it can compare the threading requirements of the first object with the threading requirements of the second object by comparing their contexts.

Objects that share the same state are said to live in the same context. For example, two objects that do not need to be synchronized can share the same context. `ContextBoundObject` and contexts are discussed in more detail in the section on contexts.

With this intuitive understanding of contexts, we can now explain the meaning of the various `Synchronization` attributes. NOT_SUPPORTED means that the class cannot support synchronization of its instance methods and fields and therefore must not be created in a synchronized context. REQUIRED means that the class requires synchronization of access to its instance methods and fields. If a thread is already being synchronized, however, it can use the same synchronization lock and live in an existing synchronization context. REQUIRES_NEW means that not only is synchronization required, but access to its instance methods and fields must be with a unique synchronization lock and context. SUPPORTED means that the class does not
require synchronization of access to its instance methods and fields, but a new context does not have to be created for it.

You can also pass a bool flag to the constructor to indicate if reentrancy is required. If required, call-outs from methods are synchronized. Otherwise, only calls into methods are synchronized.

With the **Synchronization** attribute, there is no need for ```Monitor::Enter``` and ```Monitor::Exit``` in the ```Broker::Reserve``` method.

Just as in Step 2, this example attempts to make two reservations for the last room in a hotel. Here is the output from running this example:

```
Added Boston Presidential Hotel with one room.
Thread 13 starting a new thread.
Thread 28 starting.
Thread 29 starting.
Reserving for Customer 1 at the Boston Presidential Hotel on 12/12/2001 12:00:00 AM for 3 days
Thread 28 entered Broker::Reserve
Thread 28 finds the room is available in Broker::Reserve
Thread 28 left Broker::Reserve
Reservation for Customer 1 has been booked
ReservationId = 1
ReservationRate = 10000
ReservationCost = 30000
Comment = OK
Reserving for Customer 2 at the Boston Presidential Hotel on 12/13/2001 12:00:00 AM for 1 days
Thread 29 entered Broker::Reserve
Thread 29 left Reserve.
Reservation for Customer 2 could not be booked
Room not available
Done!
```

As in the previous case, the second thread could not enter the **Reserve** method until the thread that entered first finished. Only one reservation is made.

What is different about using this automatic approach is that you get the synchronization in all the methods of the class whether you need it or not. Accessing other shared data via other methods in the class is also blocked. You get this behavior whether you want it or not.

Note how only one thread can be in any method of the class at any given time. Assume that you added another method to the class named **CancelReservation**. If one thread called **CancelReservation**, it would block all other threads from calling other methods in the class, including **MakeReservation**.
With a reservation system, this is the behavior you would want, since you do not want the `MakeReservation` attempt to use a data structure that might be in the middle of being modified. In some situations, however, this is not desirable and reduces performance due to unnecessary blocking. The resulting increase in contention can interfere with scalability, since you are not just locking around the specific areas that need synchronizing.

The attribute approach is simpler than using critical sections. You do not have to worry about the details of getting the synchronization implemented correctly. On the other hand, you get behavior that reduces scalability. Different applications or different parts of the same application will benefit from the approach that makes the most sense.

### Thread Isolation

An exception generated by one thread will not cause another thread to fail. The `ThreadIsolation` example demonstrates this.

```csharp
__gc class tm
{
public:
    void m()
    {
        Console::WriteLine(
            "Thread (0) started",
            Thread::CurrentThread->GetHashCode().ToString());
        Thread::Sleep(1000);
        for(int i = 0; i < 10; i++)
            Console::WriteLine(i);
        Console::WriteLine(
            "Thread (0) done",
            Thread::CurrentThread->GetHashCode().ToString());
    }
};

__gc class te
{
public:
    void tue()
    {
        Console::WriteLine(
            "Thread (0) started",
            Thread::CurrentThread->GetHashCode().ToString());
        Exception *e = new Exception("Thread Exception");
        throw e;
    }
};
```
The following output is generated. Note how the second thread can continue to write out the numbers even though the first thread has aborted from the unhandled exception. Note also how the “main” thread that spawned the other two threads can finish without causing the other threads to terminate.

Thread 2 starting new threads.
Thread 2 done.
Thread 5 started
Thread 6 started

Unhandled Exception: System.Exception: Thread Exception
 at te.tue() in c:\oi\netcpp\chap8\threadisolation\threadisolation.h:line 32
The `AppDomain` class (discussed later in the chapter) allows you to set up a handler to catch an `UnhandledException` event.

### Synchronization of Collections

Some lists, such as `TraceListeners`, are thread safe. When this collection is modified, a copy is modified and the reference is set to the copy. Most collections, like `ArrayList`, are not thread safe. Making them automatically thread safe would decrease the performance of the collection even when thread safety is not an issue.

An `ArrayList` has a static `Synchronized` method to return a thread-safe version of the `ArrayList`. The `IsSynchronized` property allows you to test if the `ArrayList` you are using is the thread-safe version. The `SyncRoot` property can return an object that can be used to synchronize access to a collection. This allows other threads that might be using the `ArrayList` to be synchronized with the same object.

### Context

In order for us to understand how the runtime is able to enforce a threading requirement based on an attribute, we have to introduce the concept of context. Step 4 of the `Threading` example is the same code as Step 3, but with some additional output:

```
Is the customer object a proxy? False
Is the bookings object a proxy? True
Added Boston Presidential Hotel with one room.
Thread 13 ContextId 0 launching threads.
MakeReservation: Thread 28 ContextId 0 starting.
MakeReservation: Thread 29 ContextId 0 starting.
```
Reserving for Customer 2 at the Boston Presidential Hotel on 12/13/2001 12:00:00 AM for 1 days
Thread 29 ContextId 1 entered Reserve.
Thread 29 finds the room is available in Broker::Reserve
Reserving for Customer 1 at the Boston Presidential Hotel on 12/12/2001 12:00:00 AM for 3 days
Thread 29 ContextId 1 left Reserve.
Thread 28 ContextId 1 entered Reserve.
Thread 28 ContextId 1 left Reserve.
Reservation for Customer 1 could not be booked
Room not available
Reservation for Customer 2 has been booked
ReservationId = 1
ReservationRate = 10000
ReservationCost = 10000
Comment = OK

In this last step (Step 4) of the Threading example, we see that when a thread enters a method, such as Reserve of the Broker class, it has a different ContextId than when it runs outside of the Broker class. It runs in a different context. That is why you see above that thread 28 has the ContextId set to 0 before it calls Reserve, and then within Reserve, its ContextId changes to 1. This is due to the Synchronization attribute being applied to the Broker class.

Broker objects have different runtime requirements than the other objects in the program, since access to Broker objects must be synchronized and access to other objects should not be synchronized. The environment that represents the runtime requirements of an object is called a context. There are two contexts in the Threading Step 3 and Step 4 examples: Context 1 where the Broker object lives and Context 0 where all other objects live. Every thread in the program runs in Context 1 when executing inside a Broker object, Context 0 everywhere else. Contexts are independent of threads.

A context is a collection of one or more objects that have identical concurrency requirements. The .NET concept of a context is similar to the COM concept of an apartment and the COM+ concept of a context. In general, you cannot say what the runtime must do in a given context because it depends on exactly what the runtime requirements are. A context that has transactional requirements requires different action than one that does not. Or, a context that has to maintain a REQUIRED synchronization requirement

---

At this point in time, COM+ contexts and .NET contexts are different. For a discussion of contexts in COM+, see Understanding and Programming COM+ by Robert J. Oberg.
is different than one that has to maintain a REQUIRES_NEW synchronization requirement.

You can get the `Context` class instance that represents the current context from the static property `Thread::CurrentContext`. `ContextId` is a property of the `Context` class.

### Proxies and Stubs

How does the runtime enforce the different requirements of different contexts? When an object that resides in another context (such as the `HotelBroker` object in the `NewReservation` instance), a pointer to a proxy object is returned instead of a pointer to the object itself. The actual object resides in its original, or home, context. The proxy is an object that represents the original object in a different context. The static method `RemotingServices::IsTransparentProxy` determines if an object reference points to a real object instance or a proxy. Look at the code in the main routine in `Threading.h` of the `TestThreading` project in Step 4:

```cpp
bool bTrans;
bTrans = RemotingServices::IsTransparentProxy(
customers);
Console::WriteLine(
    "Is the customer object a proxy? {0}",
bTrans.ToString());

bTrans = RemotingServices::IsTransparentProxy(
    hotelBroker);
Console::WriteLine(
    "Is the bookings object a proxy? {0}",
bTrans.ToString());
```

This causes the following output:

```
Is the customer object a proxy? False
Is the bookings object a proxy? True
```

When a program starts up, it is given a default context.\(^{14}\) All objects, like the `Customers` object, that do not have any special requirements are created inside of that context (context 0). An object, such as the `HotelBroker` object, that has a different set of requirements (synchronization) is created in a differ-

---

\(^{14}\) As will be clear in the next section, the sentence should really read, “When a new application domain starts up, it is given a new default context.” Contexts are application domain relative. Two different application domains will have two separate default contexts, each with id 0.
ent context (context 1), and a proxy is returned to the creating context (context 0).

Now when you access the MakeReservation method in the HotelBroker object, you are actually accessing a method on the proxy. The proxy method can then apply the synchronization lock and then delegate to the actual HotelBroker object’s method. When the actual object’s method returns, it returns to the proxy. The proxy can then remove the synchronization lock and return to the caller. This technique, where the runtime uses a proxy to intercept method calls to the actual object, is called interception.

You may also want to look at a similar example named MarshalByReference in the example directory; it shows how an object can be created in different ways and the effect on whether or not a proxy object is created.

**ContextBoundObject**

The Broker class has to derive from the class ContextBoundObject so that the runtime knows to set up a different context if one is required. If you remove the derivation of Broker from ContextBoundObject, you will once again get the threading conflict, and both customers will be able to reserve the last room at the hotel, even though the class is still marked with the Synchronization attribute. Objects that do not derive from ContextBoundObject can run in any context (agile objects).

Since other contexts work with a proxy or a reference to the actual object, the runtime must translate (marshal) the call from one context to another. Hence, ContextBoundObject inherits from MarshalByRefObject. MarshalByRefObject is the base class for objects that need to be able to be marshaled by reference.

One advantage of using synchronization techniques such as a Monitor is that a Monitor can be called from any context. Another potential disadvantage of using automatic synchronization is the performance hit from marshaling and using proxies rather than the actual object.

As will be clear when we discuss application domains, since the customer object has no dependency on context, it is the actual object that is accessed, not a proxy. It can be copied to any context within the same application domain.

**Application Isolation**

When writing an application, it is often necessary to isolate parts of the application so that a failure of one part does not cause a failure in another part of the application. In Windows, application isolation has traditionally been at the process level. In other words, if a process is stopped or crashes, other processes will continue running. One process cannot directly address memory in
another process’s address space; however, several interprocess communication mechanisms may be utilized.

Unfortunately, it is expensive for an application to use separate processes to achieve such isolation. To switch from one process to another, the process state information (context) must be saved and restored. This includes a thread and a process switch. A thread switch requires saving CPU registers, such as the call stack and instruction pointer, and loading the information for a new thread, as well as updating the scheduling information for the threads. A process switch includes IO buffers, accounting information, and processor rights that have to be saved for the old process and restored for the new one.

**Application Domain**

The .NET application domain (sometimes called the AppDomain) is a more lightweight unit for application isolation, fault tolerance, and security. Multiple application domains can run in one process. Since the .NET code can be checked for type safety and security, the CLR can guarantee that one application domain can run independently of another application domain in the same process. No process switch is required to achieve application isolation.

Application domains can have multiple contexts, but a context exists in only one application domain. Although a thread runs in one context of one application domain at a time, the Threading examples Step 3 and Step 4 demonstrate that a thread can execute in more than one context. One or more threads can run in an application domain at the same time. An object lives in only one context.

Each application domain starts with a single thread and one context. Additional threads and contexts are created as needed.

There is no relationship between the number of application domains and threads. A Web server might require an application domain for each hosted application that runs in its process. The number of threads in that process would be far fewer, depending on how much actual concurrency the process can support.

To enforce application isolation, code in one application domain cannot make direct calls into the code (or even reference resources) in another application domain. They must use proxies.

**Application Domains and Assemblies**

Applications are built from one or more assemblies, but each assembly is loaded into an application domain. Each application domain can be unloaded independently of the others, but you cannot unload an individual assembly from an application domain. The assembly will be unloaded when the application domain is unloaded. Unloading an application domain also frees all resource associated with that application domain.
Each process has a default application domain that is created when the process is started. This default domain can only be unloaded when the process shuts down.

Applications such as ASP.NET or Internet Explorer critically depend on preventing the various applications that run under it from interfering with each other. By never loading application code into the default domain, they can ensure that a crashing program will not bring down a host.

**AppDomain Class**

The **AppDomain** class abstracts application domains. The **AppDomain** sample illustrates the use of application domains. This class has static methods for creating and unloading application domains:

```csharp
AppDomain *domain = AppDomain::CreateDomain("CreatedDomain2", 0, 0);
...
AppDomain::Unload(domain);
```

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

```csharp
static AppDomain CreateDomain(
    String *friendlyName,
    Evidence *securityInfo,
    AppDomainSetup *info);
```

The **Evidence** parameter is a collection of the security constraints on the application domain. While we will discuss this in greater detail in Chapter 13, the domain's creator can modify this collection to control the permissions that the executing application domain can have. The **AppDomainSetup** parameter specifies setup information about the domain. Among the information specified is the location of the application domain's configuration file and where private assemblies are loaded. Hence, application domains can be configured independently of each other. Code isolation, setup isolation, and control over security combine to ensure that application domains are independent of each other.

**AppDomain Events**

To help in maintaining application isolation, the **AppDomain** class allows you to set up event handlers for

- when a domain unloads.
- when the process exits.
Chapter 8  •  .NET Framework Classes

- when an unhandled exception occurs.
- when attempts to resolve assemblies, types, and resources fail.

AppDomain Example

If you run the AppDomain example,15 you will get the following output in Figure 8–1.

Figure 8–1  Output of AppDomain example.

15 The TestApp subproject in the AppDomain example is implemented in C# rather than C++. The C++ version is provided in a separate subdirectory that is not part of the AppDomain project. Unfortunately, the C++ version causes a BadImageFormatException to be thrown in the call to ExecuteAssembly, stating that the format of TestApp.exe is invalid. This problem does not occur with the C# version of TestApp.exe. The problem with the C++ version is due to the fact that Managed C++ images can contain unmanaged relocations and unmanaged imports. The initial release of Visual C++.NET cannot load images that contain these unmanaged constructs; however, Microsoft is looking into fixing this in a future version.
First, the name, thread, and context of the default domain is written out.

```csharp
AppDomain *currentDomain = AppDomain::CurrentDomain;
Console::WriteLine("At startup, Default AppDomain is {0} ThreadId: {1}
ContextId (2)\n", currentDomain->FriendlyName,
Thread::CurrentThread->GetHashCode().ToString(),
Thread::CurrentContext->ContextID.ToString());
```

We then load and execute an assembly. This code in this assembly just prints out a string and its domain's name, thread, and context. Notice that it executes in the default domain.

```csharp
int val = domain->ExecuteAssembly("TestApp\bin\Debug\TestApp.exe", 0, args);
```

We then create an instance of the `Customers` type from the `Customer` assembly in the default domain. The `CreateInstance` method of the `AppDomain` class returns an `ObjectHandle` instance. You can pass this `ObjectHandle` instance between application domains without loading the metadata associated with the wrapped type. When you want to use the create object instance, you must unwrap it by calling the `Unwrap` method on the `ObjectHandle` instance.

```csharp
ObjectHandle *oh = currentDomain->CreateInstance("Customer", "OI.NetCpp.Acme.Customers");
...
Customers *custs =
    dynamic_cast<Customers *>(oh->Unwrap());
```

We then add a new customer, and then list all the existing customers. Notice that both the constructor of this type and its methods execute in the same thread and context as the default domain does.

We then create a new domain and create an instance of the same type as before in that new domain.

```csharp
AppDomain *domain = AppDomain::CreateDomain("CreatedDomain1", 0, 0);
...
oh = domain->CreateInstance("Customer", "OI.NetCpp.Acme.Customers");
...
Customers *custs2 = dynamic_cast
           <Customers *>(oh->Unwrap());

Note that the constructor call that results from the CreateInstance
method executes in the new domain and is therefore in a different context
from where the CreateInstance call was made, but is executing on the same
thread that made the CreateInstance call.

When we list the customers in this new object, we get a different list of
customers. This is not surprising, since it is a different Customers object.
Nonetheless, the customer list method executes in the default domain!

Using RemotingServices::IsTransparentProxy, we see that the
ObjectHandle is a proxy to the Customers object that lives in the newly created
AppDomain. However, when you unwrap the object to get an instance handle,
you do not get a proxy, but you get an actual object reference. By default, objects
are marshaled by value (copied) from one application domain to another.

If the Customers object is not serializable, you will get an exception
when you try to copy it. This exception would be thrown when you do the
Unwrap, not the CreateInstance. The latter returns a reference. The copy is
made only when the ObjectHandle is unwrapped. If the object cannot be
serialized, it cannot be copied from one application domain to another.

Next, we create a new thread, and that thread creates a new application
domain, and then loads and executes an assembly. The assembly starts exec-
cuting at its entry point, the Main routine of the AppDomainTest class.

AppDomain *domain = AppDomain::CreateDomain(
   "CreatedDomain2", 0, 0);
...
String * args[] = new String *[1];
args[0] = "MakeReservation";
...
int val = domain->ExecuteAssembly(
   "TestApp\bin\Debug\TestApp.exe", 0, args);
...
AppDomain::Unload(domain);

The Main routine loads the Hotel assembly into the newly created
application domain. In this example, the TestApp.exe application is imple-
mented in C#. It then queries the metadata of the assembly for the HotelBro-
ker type information. It then uses that type information to create a
HotelBroker object. The HotelBroker class is marked with a synchroniza-
tion attribute. As a result, the HotelBroker constructor and the MakeReser-
vation method run in a different context than the default context.

Assembly a = AppDomain.CurrentDomain.Load("Hotel");
Type typeHotelBroker =
a.GetType("OI.NetCpp.Acme.HotelBroker");
HotelBroker hotelBroker =
    (HotelBroker)Activator.CreateInstance(typeHotelBroker);
DateTime date = DateTime.Parse("12/2/2001");
ReservationResult rr = hotelBroker.MakeReservation(1,
    "Boston", "Sheraton", date, 3);
Console.WriteLine("Reservation Id: {0}", rr.ReservationId);

Marshaling, AppDomains, and Contexts

By default, objects are copied from one application domain to another (marshal by value). The section "Remoting" shows how to marshal by reference between application domains. This ensures that code in one application domain is isolated from another.

Objects are marshaled by reference between contexts. This allows the CLR to enforce the requirements (such as synchronization or transactions) of different objects. This is true whether the client of the object is in the same application domain or not.

Since most objects do not derive from ContextBoundObject, they can reside or move from one context to another as required. Threads can cross application domain and context boundaries within the same Win32 process.

Asynchronous Programming

.NET supports a design pattern for asynchronous programming. This pattern is present in many places in .NET (including I/O operations, as noted earlier, and as we will see in Chapter 11). Asynchronous programming provides a way for you to provide a method call without blocking the method caller. From the perspective of the client, the asynchronous model is easier to use than threading. It offers much less control over the synchronization than using synchronization objects, however, and the class designer would probably find threading to be much easier to use.

The Asynchronous Design Pattern

This design pattern is composed of two parts: a set of methods and an interface IAsyncResult. The methods of the pattern are

IAsyncResult *BeginXXX(
    [InputParams], AsyncCallback *cb, Object *AsyncObject)

[ReturnValue] EndXXX([OutputParams], IAsyncResult *ar);
As a design pattern, the XXX represents the actual method being called asynchronously (e.g., BeginRead/EndRead for the System::IO::FileStream class). The BeginXXX should pass all input parameters of the synchronous version (in, in/out, and ref) as well as the AsyncCallback and AsyncObject parameters. The EndXXX should have all the output parameters of the synchronous version (ref, out, and in/out) parameters in its signature. It should return whatever object or value that the synchronous version of the method would return. It should also have an IAsyncResult parameter. A CancelXXX can also be provided if it makes sense.

The AsyncCallback is a delegate that represents a callback function.

```csharp
public __delegate void AsyncCallback(IAsyncResult *ar);
```

The AsyncObject is available from IAsyncResult. It is provided so that in the callback function you can distinguish which asynchronous read generated the callback.

The framework uses this pattern so that the FileStream synchronous Read method can be used asynchronously. Here is the synchronous FileStream::Read method:

```csharp
int Read(
  __in unsigned char* array __gc[],
  int offset, int count);
```

Here is the asynchronous version using the design pattern:

```csharp
IAAsyncResult *BeginRead(
  __in unsigned char* array __gc[],
  int offset, int numBytes,
  AsyncCallback *userCallback,
  Object *stateObject);
```

```csharp
int EndRead(IAAsyncResult *asyncResult);
```

Any exception thrown from BeginXXX should be thrown before the asynchronous operation starts. Any exceptions from the asynchronous operation should be thrown from the EndXXX method.

IAsyncResult

IAsyncResult is returned by a BeginXXX method (such as BeginRead). This interface has four elements:

```csharp
public __gc __interface IAsyncResult {
```
public:
    bool get_IsCompleted();
    bool get_CompletedSynchronously();
    WaitHandle* get_AsyncWaitHandle();
    Object* get_AsyncState();
}

The get_IsCompleted field is set to true after the call has been processed by the server. The client can destroy all resources after get_IsCompleted is set to true. If BeginXXX completed synchronously, get_CompletedSynchronously is set to true. Most of the time this will be ignored and set to the default value of false. In general, a client never knows whether the BeginXXX method executed asynchronously or asynchronously. If the asynchronous operation is not finished, the EndXXX method will block until the operation is finished.

The get_AsyncWaitHandle returns a WaitHandle that can be used for synchronization. As we discussed previously, this handle can be signaled so that the client can wait on it. Since you can specify a wait time period, you do not have to block forever if the operation is not yet complete.

The get_AsyncState is the object provided as the last argument in the BeginXXX call. It allows you to differentiate asynchronous reads in the callback.

Using Delegates for Asynchronous Programming

Any developer of .NET objects who wants to provide an asynchronous interface should follow the pattern just described. Nonetheless, there is no need for most developers to develop a custom asynchronous solution for their objects. Delegates provide a very easy way to support asynchronous operations on any method without any action on the class developer's part. Of course, this has to be done with care because the object may have been written with certain assumptions about which thread it is running on and its synchronization requirements.

The two Asynch examples16 use the Customers object from our case study Customer assembly. The first example, AsynchWithoutCallback, registers new customers asynchronously and does some processing while waiting for each registration to finish. The second example, AsynchWithCallback, uses a callback function with the asynchronous processing. In addition to allowing the program to do processing while waiting for the registrations to finish, the callback allows the system to take some asynchronous action for each individual registration.

In the examples, we just print out to the console to show where work could be done. To increase the waiting time to simulate longer processing

16 As usual, you will need to copy the DLL to the Debug directory of the test program.
times, we have put calls to `Thread::Sleep()` in `Customers::RegisterCustomer` as well as in the sample programs. Now let us look at the code within the examples.

Suppose the client wants to call the `RegisterCustomer` method asynchronously. The caller simply declares a delegate with the same signature as the method.

```csharp
public __delegate int RegisterCustomerCbk(
    String *FirstName,
    String *LastName,
    String *EmailAddress);
```

You then make the actual method the callback function:

```csharp
RegisterCustomerCbk *rcc = new
    RegisterCustomerCbk(
        customers,
        Customers::RegisterCustomer);
```

**BEGIN/END INVOKE**

When you declare a delegate, the compiler generates a class with three methods: `BeginInvoke`, `EndInvoke` and `Invoke`. The `BeginInvoke` and `EndInvoke` are type-safe methods that correspond to the `BeginXXX` and `EndXXX` methods and allow you to call the delegate asynchronously. The `Invoke` method is what the compiler uses when you call a delegate. To call `RegisterCustomer` asynchronously, just use the `BeginInvoke` and `EndInvoke` methods.

```csharp
RegisterCustomerCbk *rcc = new
    RegisterCustomerCbk(
        customers,
        Customers::RegisterCustomer);
for(int i = 1; i < 5; i++)
{
    firstName = String::Concat(
        "FirstName", i.ToString());
    lastName = String::Concat(
        "SecondName", (i * 2).ToString());
```

---

17 If you open the executable from the `DelegateAccount` example in Chapter 5 in ILDASM, you can observe this. The `NotifyCallback` class has the `BeginInvoke`, `EndInvoke`, and `Invoke` methods defined. If you look at the `Withdraw` method for `Account`, you will notice that the line `NotifyDlg(balance)` has been transformed to `instance void NotifyCallback::Invoke(valuetype [mscorlib]System.Decimal)`.
Asynchronous Programming

```csharp
emailAddress = String::Concat(i.ToString(), "\".biz\";
IAsyncResult *ar =
rcc->BeginInvoke(
    firstName,
    lastName,
    emailAddress,
    0,
    0);
while(!ar->IsCompleted)
{
    Console::WriteLine("Could do some work here while waiting for customer
registration to complete.");
ar->AsyncWaitHandle->WaitOne(1, false);
}
customerId = rcc->EndInvoke(ar);
Console::WriteLine("    Added CustomerId: {0}",
customerId.ToString());
}

The program waits on the AsyncWaitHandle periodically to see if the
registration has finished. If it has not, some work could be done in the
interim. If EndInvoke is called before RegisterCustomer is complete, End-
Invoke will block until RegisterCustomer is finished.

ASYNCALL BACK

Instead of waiting on a handle, you could pass a callback function to Begin-
Invoke (or a BeginXXX method). This is done in the AsynchWithCallback
example.

```
lastName = String::Concat("SecondName", (i * 2).ToString());
emailAddress =
    String::Concat(i.ToString(), ".biz");
objectState = __box(i);
ar = rcc->BeginInvoke(
    firstName, lastName,
    emailAddress, cb, objectState);
}

Console::WriteLine(  
    "Finished registrations...could do some work here.");
Thread::Sleep(25);
Console::WriteLine(  
    "Finished work..waiting to let registrations complete.");
Thread::Sleep(1000);

You then get the results in the callback function:

void CustomerCallback(IAsyncResult *ar)
{
    int customerId;
    AsyncResult *asyncResult =
        dynamic_cast<AsyncResult *>(ar);
    RegisterCustomerCbk *rcc =
        dynamic_cast<RegisterCustomerCbk *>(
            asyncResult->AsyncDelegate);

    customerId = rcc->EndInvoke(ar);
    Console::WriteLine(  
        "AsyncState: {0} CustomerId {1} added.";
        ar->AsyncState,
        customerId.ToString());
    Console::WriteLine(  
        "Could do processing here.");
    return;
}

You could do some work when each customer registration was finished.

**Threading with Parameters**

The asynchronous callback runs on a different thread from the one on which **BeginInvoke** was called. If your threading needs are simple and you want to
Asynchronous Programming

pass parameters to your thread functions, you can use asynchronous delegates
to do this. You do not need any reference to the **Threading** namespace. The
reference to that namespace in the **AsynchThreading** example is just for the
**Thread::Sleep** method needed for demonstration purposes.

**PrintNumbers** sums the numbers from the starting integer passed to it
as an argument to 10 greater than the starting integer. It returns that sum
to the caller. **PrintNumbers** can be used for the delegate defined by **Print**.

```csharp
//AsynchThreading.h

using namespace System;
using namespace System::Threading;

public __delegate int Print(int i);

public __gc class Numbers
{
    public:
        int PrintNumbers(int start)
        {
            int threadId = Thread::CurrentThread->GetHashCode();
            Console::WriteLine(
                "PrintNumbers Id: {0}",
                threadId.ToString());
            int sum = 0;
            for (int i = start; i < start + 10; i++)
            {
                Console::WriteLine(i.ToString());
                Thread::Sleep(500);
                sum += i;
            }
            return sum;
        }
}

The **Main** routine then defines two callbacks and invokes them explicitly
with different starting integers. It waits until both of the synchronization
handles are signaled. **EndInitinvoke** is called on both, and the results are written
to the console.

Numbers *n = new Numbers;
Print *pf n1 = new Print(n, Numbers::PrintNumbers);
Print *pf n2 = new Print(n, Numbers::PrintNumbers);
IAsyncResult *ar1 =
    pfn1->BeginInvoke(0, 0, 0);
Chapter 8 • .NET Framework Classes

The program's output:

MainThread Id: 2
PrintNumbers Id: 14
0
1
2
3
4
5
6
7
8
9
PrintNumbers Id: 14
100
101
102
103
104
105
106
107
108
109
Sum1 = 45 Sum2 = 1045

Remoting

Remoting technology uses all of the key concepts in the .NET Application Model. While a complete discussion of remoting is beyond the scope of this
book, a brief introduction provides a powerful example of how metadata and marshal by reference (MBR) work. Remoting also provides a mechanism that supports executable servers.

Unlike remoting in Microsoft’s COM technology, there is a minimal amount of infrastructure programming required. What infrastructure program is required allows programmers either a degree of flexibility or the ability to customize remoting for their particular applications.

The .NET Framework provides two ways to provide connections between two applications on different computers. Web Services, discussed in Chapter 11, enable computers that do not host the CLR to communicate with computers that do. The remoting technology discussed here builds distributed applications between computers that host the CLR.

**Remoting Overview**

The key parts of Remoting are

- **Interception**, which allows for message generation for communication over the channels.
- **Formatters** to put the messages into a byte stream that is sent over the channel. These are the same formatters that were discussed in the section on serialization.
- Communication **channels** for transport of messages.

**INTERCEPTION**

Proxies and stubs (referred to in .NET as dispatchers) transform the function calls on the client or server side into messages that are sent over the network. This is called interception, because the proxies and dispatchers intercept a method call to send it to its remote destination. Unlike COM, metadata provides the information so the CLR can generate the proxies and stubs for you.

A *proxy* takes the function call off the stackframe of the caller and transforms it into a message. The message is then sent to its destination. A *dispatcher* takes the message and transforms it into a stackframe so that a call can be made to the object.

For example, assume the **UnregisterCustomer** method from the **Customer** assembly runs in one application domain and is called from another. It makes no difference if the application domains are in the same process or on the same machine.

The proxy would take the integer **id** argument on the stackframe of the client making the call and put it in a message that encoded the call and its argument. On the server side, the dispatcher would take that message and create a function call on the server’s stack for the call **UnregisterCustomer(int id)** and make that call into the object. The client and server code do not need to know that they are being remoted.
CHANNELS AND FORMATTERS

The formatter converts the message into a byte stream. The .NET Framework comes with two formatters, binary and SOAP (text-based XML, discussed in the Chapter 11). The byte stream is then sent over a communication channel.

The .NET Framework comes with two channels, although you can write your own. The HTTP channel uses the HTTP protocol and is good for communicating over the Internet or through firewalls. The TCP channel uses the TCP (sockets) protocol and is designed for high-speed communication. You have four permutations of formatters and transport: binary over TCP, binary over HTTP, SOAP over HTTP, and SOAP over TCP.

Remote Objects

Clients obtain a proxy by activating a remote object. Remote objects must derive from MarshalByRefObject, because you work with a proxy to the object reference, not with the object reference itself. This is the same concept discussed in the section on contexts, where marshal by reference is also used to access context bound objects.

Local objects passed as method parameters from one application domain to another can be passed by value (copied) or by reference.

To be passed by value, they must be serializable. The object is serialized, sent across the transport layer, and re-created on the other side. We have already seen an example of this in the AppDomain example.

To be passed by reference, the class must derive from MarshalByRefObject. The Remoting example illustrates pass by reference.

Remote objects can be either server or client activated. Server-activated objects are not created until the first method call on the object. Server-activated objects come in two flavors, SingleCall and Singleton. SingleCall objects are stateless. Each method causes a new object to be created. Singleton objects can be used by multiple client activation requests. Singleton objects can maintain state. SingleCall objects will scale better than Singleton objects because they do not retain state and can be load balanced.

Client-activated objects are activated when the client requests them. While they can last for multiple calls and hold state, they cannot store information from different client activations. This is similar to calling CoCreateInstanceEx in DCOM.

Activation

Objects are activated on the client side in one of three ways by using the Activator class.

* Activator::GetObject is used to get a reference to a server-activated object.
* Activator::CreateInstance is used to create a client-activated object. You can pass parameters to the object's constructor using
one of the overloaded `CreateInstance` methods that takes an array of objects to be passed to the constructor.

- The C++ `new` syntax can be used to create a server or a client activated object. A configuration file is used to describe how `new` should be used.

**Sample Remotable Object**

For our remoting example, we remote our `Customers` object from the `Customer` assembly. In the Remoting example directory there are two solutions. One represents the client program, the other the server program. Build the server solution first, which will also build the dependent `Customer.dll`. Copy this DLL into the `Debug` directory of both `Server` and `Client`. Start the server program first. Notice it waits for a client request. You can then run the client program that will run against objects that live inside of the server. We will discuss the details of the client and server code and output in the next few sections.

Notice that we only had to make two simple changes to our object. The `Customers` class in the server project had to be made remotable by inheriting from `MarshalByRefObject`.

```csharp
public __gc class Customers :
    public MarshalByRefObject, public ICustomer
```

The `CustomerListItem` that was going to be transferred by value had to be made serializable.

```csharp
[Serializable]
public __value struct CustomerListItem
{
    public:
        int CustomerId;
        String *FirstName;
        String *LastName;
        String *EmailAddress;
};
```

**SAMPLE REMOTING PROGRAM**

In the Remoting example the client accesses a server-activated object. The server is the `TcpServerChannel` class that uses a binary format with the TCP protocol. The channel will use port 8085. The server registers the type being remoted, the endpoint name to refer to this object, and the type of activation. The server then waits for client requests.

```csharp
TcpServerChannel *chan = new TcpServerChannel(8085);
ChannelServices::RegisterChannel(chan);
```
The server has to be started before the client program can access the object.

The client sets up a **TcpClientChannel** object and then connects to the object. It specifies the type of the object it wants and the endpoint where the server is listening for object requests. If you want to run the client and server on separate machines, substitute the server machine name for localhost in the endpoint. Unlike COM location transparency, the client has to specify a specific endpoint; there is no redirection through an opaque registry entry.

```csharp
TcpClientChannel *chan = new TcpClientChannel;
ChannelServices::RegisterChannel(chan);
Customers *obj = dynamic_cast<Customers *>(
    Activator::GetObject(
        __typeof(Customers),
        "tcp://localhost:8085/AcmeCustomer"));
if (obj == 0)
    Console::WriteLine("Could not locate server");
else
{
    ...
}
```

The client then uses the proxy to make calls on the object as if it were a local instance.

```csharp
bool bRet =
    RemotingServices::IsTransparentProxy(obj);
...
ArrayList *ar;
ar = obj->GetCustomer(-1);
ShowCustomerArray(ar);
```

To run the program, start the server program in one console window, and then run the client program from another console window.

The output depends on what kind of server-activated object is being activated. If the server activation type is **Singleton**, which supports the maintaining state, you get the behavior you would expect from the non-remoted case. A new customer is added, and you find that new customer in the list when you ask for all the existing customers. As you would expect, the initial
activate call results in the **Customers** constructor being called once for each server invocation no matter how many times the client program is run.

If the activation type is **SingleCall**, which creates a new object instance for every method call, the results are quite different. Four different objects are created. The first object is created by the initial activate request. The second is created by the initial call to `GetCustomer`. The third object is created by the `RegisterCustomer` call. The fourth object is created by the second call to `GetCustomer`. The last object created never sees the new customer because no state is saved. Note that the static `nextCustId` member of the `Customer` class is treated as static with respect to the new object instances of the `Customer` class, just as you would expect. Same client code, different results!

Since the object is already activated, if you run the client program a second time for the same server invocation, the `Customers` constructor will be called only three times.

Since the client uses a proxy, the object executes inside the server’s application domain, but on a different thread than the main server thread. The object’s constructor is not called until the first method call on the object. Notice how in both cases we have remoted an **ArrayList** of types without any special work aside from making the type serializable. The presence of metadata makes the programmer’s work much easier.

**Metadata and Remoting**

In order for the client to request an object of a specific type, metadata about the type has to be available to the client. For the remoting example shown in
this chapter, a reference is simply made to the actual assembly where the object is stored. However, for many applications, you do not want to give the client access to your actual assembly, since it can be decompiled into source code via reflection (i.e., reverse engineering). For the metadata that the client needs, a reference need only be made to an object that contains interface information, but no actual implementation details.

One way to do this is to build a version of the object that has methods with no implementation. This interface class can then be built into an assembly that can be given to the client. You can throw the `System::NotSupportedException` in the methods if you wish to make sure that they are never used by mistake for the real object.

For Web Services, you use the SOAPSUDS tool to extract the metadata from the service, and then generate an assembly that has the required metadata. You can then build a proxy DLL and have the client program refer to it. This is conceptually equivalent to the first approach. The server, of course, has to reference the real object’s assembly.

Unlike the COM model, there is no reference counting, interface negotiation, building and registering separate proxies and stubs, worrying about global identifiers, or use of the registry. Because of metadata, all you have to do is inherit from `MarshalByRefObject` to make an object remotable.

**Remoting Configuration Files**

You use configuration files to define where the object is activated. The client can then use the `new` operator to create the object. The big advantage in doing this is that as the object location changes (such as a URL or TCP channel), or the formatter you want to use changes, the client does not have to be rebuilt.

Multiple classes can be configured on the client. Configuration files are loaded into the client using the `RemotingConfiguration::Configure` method.

**Custom Attributes**

Chapter 5 introduced the concept of attributes, which have already appeared in several examples. In this chapter we used the `Serializable` and `Synchronization` attributes, which are provided by .NET Framework classes. The .NET Framework makes the attribute mechanism entirely extensible, allowing you to define custom attributes, which can be added to the class’s metadata. This custom metadata is available through reflection and can be used at runtime. To simplify the use of custom attributes, you may declare a base class to do the work of invoking the reflection API to obtain the metadata information.
The example CustomAttribute illustrates the custom attribute InitialDirectory. InitialDirectory controls the initial current directory where the program runs. By default, the current directory is the directory containing the solution, which in this case is C:\OI\NetCpp\Chap08\CustomAttribute.

Using a Custom Attribute

Before we discuss implementing the custom attribute, let us look at how the InitialDirectory attribute is used. 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. Here is the code for our test program.

When you run this sample on your system, you can change the directory in the attribute to one that exists on your machine.

//AttributeDemo.h

using namespace System;
using namespace System::IO;

__gc class Normal : public DirectoryContext
{
};

[InitialDirectory("C:\OI\NetCpp\Chap08")]
__gc class Special : public DirectoryContext
{
};

public __gc class AttributeDemo
{
public:
    static void Main()
    {
        Normal *objNormal = new Normal;
        Console::WriteLine(
            "path = {0}", objNormal->DirectoryPath);
        ShowDirectoryContents(objNormal->DirectoryPath);
        Special *objSpecial = new Special;
        Console::WriteLine(
            "path = {0}", objSpecial->DirectoryPath);
private:
    static void ShowDirectoryContents(String *path)
    {
        DirectoryInfo *dir = new DirectoryInfo(path);
        FileInfo *files[] = dir->GetFiles();
        Console::WriteLine("Files:");
        IEnumerator *pEnum = files->GetEnumerator();
        while (pEnum->MoveNext())
        {
            FileInfo *f =
                dynamic_cast<FileInfo *>(pEnum->Current);
            Console::WriteLine("   {0}", f->Name);
        }
        DirectoryInfo *dirs[] = dir->GetDirectories();
        Console::WriteLine("Directories:");
        pEnum = dirs->GetEnumerator();
        while (pEnum->MoveNext())
        {
            DirectoryInfo *d =
                dynamic_cast<DirectoryInfo *>(pEnum->Current);
            Console::WriteLine("   {0}", d->Name);
        }
    }
};

Here is the output:

path = c:\OI\NetCpp\Chap08\CustomAttribute
Files:
    CustomAttribute.vcproj
    CustomAttribute.ncb
    ReadMe.txt
    CustomAttribute.cpp
    AssemblyInfo.cpp
    stdafx.cpp
    stdafx.h
    CustomAttribute.sln
    CustomAttribute.suo
    AttributeDemo.h
    DirectoryContext.h
    DirectoryAttribute.h
Directories:
Debug
path = C:\OI\NetCpp\Chap08
Files:
Directories:
  Reflection
  Dynamic
  FileIO
  Serialization
  Hotel
  ISerialization
  Threading
  PulseAll
  ThreadIsolation
  AppDomain
  Asynch
  AsynchThreading
  CustomAttribute
  MarshalByReference
  Remoting

Defining an Attribute Class

To create a custom attribute, you must define an attribute class, derived from the base class `Attribute`. By convention, give your class a name ending in “Attribute.” The name of your class without the “Attribute” suffix will be the name of the custom attribute. In our example the class name is `InitialDirectoryAttribute`, so the attribute’s name 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 the name=value syntax.

You may also provide properties to read the parameter information. In our example, we have a property `Path`, which is initialized in the constructor.

```csharp
//DirectoryAttribute.h
using namespace System;

public __gc class InitialDirectoryAttribute :
    public Attribute
{
    private:
        String *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 using the reflection classes. You can obtain the Type of any object by calling the method GetType, which is provided in the root class Object. Using the class's method GetCustomAttributes you can read the custom attribute information.

To make the coding of the client program as simple as possible, it is often useful to provide a base class that does the work of reading the custom attribute information.18 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:

```c++
//DirectoryContext.h

using namespace System;
using namespace System::Reflection;
using namespace System::IO;
using namespace System::Collections;

__gc class DirectoryContext
{
    public:
    __property String *get_DirectoryPath()
    {
        Type *t = this->GetType();
        IEnumerator *pEnum = 
            t->GetCustomAttributes(true)->GetEnumerator();
        }
    }
};
```

18 With single implementation inheritance, there is a cost to providing a base class. If you need to derive from another class, such as ContextBoundObject, the base class has to derive from that class.
while (pEnum->MoveNext())
{
    Attribute *a =
        dynamic_cast<Attribute *>(pEnum->Current);
    InitialDirectoryAttribute *da =
        dynamic_cast<InitialDirectoryAttribute *>(a);
    if (da != 0)
    {
        return da->Path;
    }
}
return Directory::GetCurrentDirectory();
};

We must import 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. Since this collection is heterogeneous, consisting of different types, the dynamic_cast operator is used to test if a given collection element is of the type InitialDirectoryAttribute. If we find such an element, we return the Path property. Otherwise, we return the default current directory, obtained from GetCurrentDirectory.

Garbage Collection

Managed memory allocations are automatically reclaimed through a garbage collection algorithm. The CLR tracks the use of memory that is allocated on the managed heap, and any memory that is no longer referenced is marked as “garbage.” When memory is low, the CLR traverses its data structure of tracked memory and reclaims all the memory marked as garbage. Thus the programmer is relieved of this responsibility.

While this prevents memory leaks in the managed heap, it does not help with the reclamation of other types of allocated resources. Examples of these resources include open files or database connections, or server login connections that have to be disconnected. The programmer may need to write explicit code to perform cleanup of these other resources. This can be done in your class destructor or in a specialized cleanup method. The CLR calls your destructor when the memory allocated for an object is reclaimed.

19 The .NET Framework describes a method named Finalize in the Object base class for this purpose. However, in Managed C++ you do not use the name Finalize. Instead, you simply define a destructor for your class.
Another concern with garbage collection is performance. There is some overhead associated with automatic garbage collection. However, the CLR does provide an efficient multi-generational garbage collection algorithm.

**Object Destruction**

Unless you explicitly use the delete operator on a managed object, destruction time is non-deterministic. The destructor for a particular unreferenced object may run at any time during the garbage collection process, and the order of calling destructors for different objects cannot be predicted. Moreover, under exceptional circumstances, a destructor may not run at all (for example, a thread goes into an infinite loop or a process aborts without giving the runtime a chance to clean up). Also, unless you explicitly use the delete operator, the thread on which the destructor is called is not deterministic.

The fact that the call to the destructor is synchronous, and therefore deterministic, when you explicitly delete a managed pointer is demonstrated in the **ExplicitDelete** example. The following code shows two objects being created. The first one is then passively finalized by assigning it to zero. The garbage collector will call the destructor on its own thread asynchronously. The second object is explicitly deleted with the delete operator, and the destructor is called synchronously. The program displays details on what is happening to each object, and on what thread, using hash codes. From the output, you can see that with the passively disposed object, the destructor is run on a different thread than the **Main** method thread. In contrast, you can see that with the explicitly deleted object, the destructor is run on the same thread as the **Main** method.

```csharp
//ExplicitDelete.h

using namespace System::Threading;

public __gc class SomeClass
{
 public:
 ~SomeClass()
 {
  Console::Write(
   "Destructor running in thread: (0), ",
   __box(Thread::CurrentThread->GetHashCode()));
  Console::WriteLine(
   "Destroying object: (0)*",
   __box(this->GetHashCode()));
  }
};
```
public __gc class ExplicitDelete
{
public:
    static void Main()
    {
        Console::WriteLine(  
            "Main thread: {0} ",  
            __box(Thread::CurrentThread->GetHashCode()));  

        SomeClass *sc = new SomeClass;  
        Console::WriteLine(  
            "Main thread creating object: {0} ",  
            __box(sc->GetHashCode()));  
        Console::WriteLine(  
            "Nulling pointer to object: {0} ",  
            __box(sc->GetHashCode()));  
        sc = 0;  
        GC::Collect();  
        GC::WaitForPendingFinalizers();  

        sc = new SomeClass;  
        Console::WriteLine(  
            "Main thread creating object: {0} ",  
            __box(sc->GetHashCode()));  
        Console::WriteLine(  
            "Deleting pointer to object: {0} ",  
            __box(sc->GetHashCode()));  
        delete sc;  

        Console::WriteLine("All done.");
    }
};

Here is the output.

Main thread: 2
Main thread creating object: 5
Nulling pointer to object: 5
Destructor running in thread: 6, Destroying object: 5
Main thread creating object: 7
Deleting pointer to object: 7
Destructor running in thread: 2, Destroying object: 7
All done.
To avoid unnecessary overhead, you should not implement a destructor for a class unless you have a good reason for doing so. And if you do provide a destructor, you should probably provide an alternate, deterministic mechanism for a class to perform necessary cleanup. The .NET Framework recommends a **Dispose** design pattern for deterministic cleanup, which is described next.

### Unmanaged Resources and Dispose

Consider an object that has opened a file, and is then no longer needed and marked for garbage collection. Eventually, the object’s destructor will be called, and the file could be closed in that destructor. But, as we discussed, garbage collection is non-deterministic, and the file might remain open for an indefinitely long time. It would be more efficient to have a deterministic mechanism for a client program to clean up the object’s resources when it is done with it. The .NET Framework recommends the **IDisposable** interface for this purpose.

```csharp
public __gc __interface IDisposable
{
    void Dispose();
};
```

This design pattern specifies that a client program should call **Dispose** on the object when it is done with it. In the **Dispose** method implementation, the class does the appropriate cleanup. As backup assurance, the class should also implement a destructor in case **Dispose** never gets called, perhaps due to an exception being thrown. Since both **Dispose** and the destructor perform the cleanup, the cleanup code can be placed in **Dispose**, and the destructor can be implemented by calling **Dispose**. The **Dispose** method is designed such that a client program can call it when it is done with the object or knows that it is safe to free the resources associated with the object. One detail is that once **Dispose** has been called, the object’s destructor should not be called, because that would involve cleanup being performed twice. The object can be removed from the garbage collection queue by calling **GC::SuppressFinalize**. Also, it is a good idea for the class to maintain a bool flag such as **disposeCalled**, so that if **Dispose** is called twice, cleanup will not be performed a second time.

A **Dispose** method should also call the base class **Dispose** to make sure that all its resources are freed. It should also be written so that if a **Dis-

---

20 One of the virtues of the exception handling mechanism is that as the call stack is unwound in handling the exception, local objects go out of scope and so can get marked for finalization. We provide a small demo later in this section.
pose method is called after the resources have been already freed, no exception is thrown.

Since finalization is expensive, any objects that will no longer acquire any more resources should call the static method GC::SupressFinalize and pass it the this pointer. If you have a try/finally block in your code, you can place a call to the object’s Dispose method in the finally block to make sure that resources are freed.

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

//SimpleLog.h

using namespace System;
using namespace System::IO;

public __gc class SimpleLog : public IDisposable
{
private:
    StreamWriter *writer;
    String *name;
    bool disposeCalled;
public:
    SimpleLog(String *fileName) : disposeCalled(false)
    {
        name = fileName;
        writer = new StreamWriter(fileName, false);
        writer->AutoFlush = true;
        Console::WriteLine(
            String::Format("logfile {0} created", name));
    }
    void WriteLine(String *str)
    {
        writer->WriteLine(str);
        Console::WriteLine(str);
    }
    void Dispose()
    {
        if(disposeCalled)
            return;
        writer->Close();
        GC::SuppressFinalize(this);
        Console::WriteLine(
            String::Format("logfile {0} disposed", name));
    }
}
The class SimpleLog supports the IDisposable interface and thus implements Dispose. The cleanup code simply closes the StreamWriter object. To make sure that a disposed object will not also be finalized, GC::SuppressFinalize is called. The finalizer simply delegates to Dispose. To help monitor object lifetime, a message is written to the console in the constructor, in Dispose, and in the finalizer.21

Here is the code for the test program:

```csharp
using namespace System;
using namespace System::Threading;

using namespace System;
using namespace System::Threading;

class DisposeDemo
{
public:
    static void Main()
    {
        SimpleLog *log = new SimpleLog("log1.txt");
        log->WriteLine("First line");
        Pause();
        log->Dispose(); // first log disposed
        log->Dispose(); // test Dispose twice
        log = new SimpleLog("log2.txt");
        log->WriteLine("Second line");
        Pause();
        log = new SimpleLog("log3.txt"); // previous (2nd) log released
        log->WriteLine("Third line");
        Pause();
        log = 0; // last log released
    }
};
```

21 The Console::WriteLine in the destructor is provided purely for didactic purposes and should not be done in production code, for reasons we shall discuss shortly.
The SimpleLog object pointer log is assigned in turn to three different object instances. The first time, it is properly disposed. The second time, log is reassigned to refer to a third object, before the second object is disposed, resulting in the second object becoming garbage. The Pause method provides an easy way to pause the execution of this console application, allowing us to investigate the condition of the files log1.txt, log2.txt, and log3.txt at various points in the execution of the program.

Running the program results in the following output:

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

After the first pause, the file log1.txt has been created, and you can examine its contents in Notepad. If you try to delete the file, you will get a sharing violation, as illustrated in Figure 8–2.

At the second pause point, log1.txt has been disposed, and you will be allowed to delete it. The log2.txt file has been created (and is open). At the third pause point, log3.txt has been created. But the object reference to log2.txt has been reassigned, and so there is now no way for the client program to dispose of the second object. If Dispose were the only mechanism to clean up the second object, we would be out of luck. Fortunately, the
SimpleObject class has implemented a destructor, so the next time garbage is collected, the second object will be disposed of properly. We can see the effect of finalization by running the program through to completion. The second object is indeed finalized, and thence disposed. In fact, as the application domain shuts down, the destructor is called on all objects not exempt from finalization, even on objects that are still accessible.

In our code we explicitly make the third object inaccessible by the assignment \( \text{log} = \text{null} \), and we then force a garbage collection by a call to \text{GC::Collect}. Finally, we sleep briefly to give the garbage collector a chance to run through to completion before the application domain shuts down. Coding our test program in this way is a workaround for the fact that the order of garbage collection is non-deterministic. The garbage collector will be called automatically when the program exits and the application domain is shut down. However, at that point, system objects, such as \text{Console}, are also being closed. Since you cannot rely on the order of finalizations, you may get an exception from the \text{WriteLine} statement within the finalizer. The explicit call to \text{GC::Collect} forces a garbage collection while the system objects are still open. If we omitted the last three lines of the \text{Main} method, we might well get identical output, but we might also get an exception.

**ALTERNATE NAME FOR DISPOSE**

The standard name for the method that performs cleanup is \text{Dispose}. The convention is that once an object is disposed, it is finished. In some cases, the same object instance may be reused, as in the case of a file. A file may be

---

22 This example illustrates that it is the client’s responsibility to help the scalability of the server by cleaning up objects (using \text{Dispose}) before reassigning them. Once an object has been reassigned, there is no way to call \text{Dispose}, and the object will hang around for an indeterminate period of time until garbage is collected. Effective memory management involves both the server and client.
opened, closed, and then opened again. In such a case the standard naming convention is that the cleanup method should be called `Close`. In other cases some other natural name may be used.

Our `SimpleLog` class could plausibly have provided an `Open` method, and then it would have made sense to name our cleanup method `Close`. For simplicity, we did not provide an `Open` method, and so we stuck to the name `Dispose`.

**GENERATIONS**

As an optimization, every object on the managed heap is assigned to a generation. A new object is in generation 0 and is considered a prime candidate for garbage collection. Older objects are in generation 1. Since such an older object has survived for a while, the odds favor its having a longer lifetime than a generation 0 object. Still older objects are assigned to generation 2 and are considered even more likely to survive a garbage collection. The maximum generation number in the current implementation of .NET is 2, as can be confirmed from the `GC::MaxGeneration` property.

In a normal sweep of the garbage collector, only generation 0 will be examined. It is here that the most likely candidates are for memory to be reclaimed. All surviving generation 0 objects are promoted to generation 1. If not enough memory is reclaimed, a sweep will next be performed on generation 1 objects, and the survivors will be promoted. Then, if necessary, a sweep of generation 2 will be performed, and so on up until `MaxGeneration`.

**Finalization and Stack Unwinding**

As mentioned earlier, one of the virtues of the exception-handling mechanism is that as the call stack is unwound in handling the exception, local objects go out of scope and so can get marked for finalization. The program `FinalizeStackUnwind` provides a simple illustration. It uses the `SimpleLog` class discussed previously, which implements finalization.

```csharp
//FinalizeStackUnwind.h
using namespace System;
using namespace System::Threading;

public __gc class FinalizeStackUnwind
{
  public:
    static void Main()
    {
      try
```
private:
static void SomeMethod()
{
    // local variable
    SimpleLog *alpha = new SimpleLog("alpha.txt");
    // force an exception
    throw new Exception("error!!");
}
};

A local pointer variable alpha of type SimpleLog* is allocated in SomeMethod. Before the method exits normally, an exception is thrown. The stack-unwinding mechanism of exception handling detects that alpha is no longer accessible, and so is marked for garbage collection. The call to GC::Collect forces a garbage collection, and we see from the output of the program that finalization is indeed carried out.

logfile alpha.txt created
error!!
logfile alpha.txt finalized
logfile alpha.txt disposed

Controlling Garbage Collection with the GC Class

Normally, it is the best practice simply to let the garbage collector perform its work behind the scenes. Sometimes, however, it may be advantageous for the program to intervene. The System namespace contains the class GC, which enables a program to affect the behavior of the garbage collector. We summarize a few of the important methods of the GC class.

SUPPRESSFINALIZE

This method requests the system to not finalize (i.e., not call the destructor) for the specified object. As we saw previously, you should call this method in
your implementation of **Dispose** to prevent a disposed object from also being finalized.\(^{23}\)

### COLLECT

You can force a garbage collection by calling the **Collect** method. An optional parameter lets you specify which generations should be collected. Use this method sparingly, since normally the CLR has better information on the current state of memory. A possible use would be a case when your program has just released a number of large objects, and you would like to see all this memory reclaimed right away. Another example was provided in the previous section, where a call to **Collect** forced a collection while system objects were still valid.

### MAXGENERATION

This property returns the maximum number of generations that are supported.

### GETGENERATION

This method returns the current generation number of an object.

### GETTOTALMEMORY

This method returns the number of bytes currently allocated (not the free memory available, and not the total memory size of the heap). A parameter lets you specify whether the system should perform a garbage collection before returning. If no garbage collection is done, the indicated number of bytes is probably larger than the actual number of bytes being used by live objects.

### Sample Program

The program **GarbageCollection** illustrates using these methods of the **GC** class. The example is artificial, simply illustrating object lifetime and the effect of the various **GC** methods. The class of objects that are allocated is called **Member**. This class has a **String** property called **Name**. Write statements are provided in the constructor, **Dispose**, and destructor. A **Committee** class maintains an array list of **Member** instances. The **RemoveMember** method

---

\(^{23}\) You should be careful in the case of an object that might be “closed” (like a file) that is later re-opened again. In such a case it might be better not to suppress finalization. Once finalization is suppressed, it can be made eligible for finalization again by calling `<GC::ReRegisterForFinalize>` For a discussion of advanced issues in garbage collection and finalization, refer to the Jeffrey Richter article previously cited.
simply removes the member from the array list. The **DisposeMember** method also calls **Dispose** on the member being expunged from the committee. The **ShowGenerations** method displays the generation number of each **Member** object. **GarbageCollection.h** is a test program to exercise these classes, showing the results of various allocations and deallocations and the use of methods of the **GC** class. The code and output should be quite easy to understand.

All the memory is allocated locally in a method named **DemonstrateGenerations**. After this method returns and its local memory has become inaccessible, we make an explicit call to **GC::Collect**. This forces the destructors to be called before the application domain shuts down, and so we avoid a possible random exception of a stream being closed when a **WriteLine** method is called in a finalizer. This is the same point mentioned previously for the earlier examples.

**Summary**

This chapter introduced the .NET application model. Through metadata and reflection, the framework can understand enough about your application to provide many services that you do not have to implement. On the other hand, we have seen how the framework is structured so that you can substitute your own objects and implementations where needed.

Type safety enables application domains to provide an effective yet economical form of application isolation. Contexts, proxies, and interception allow the runtime to transparently provide services to parts of applications that require them.

Another aspect of the .NET application model is the pervasive use of attributes, which can be easily added to source code and is stored with the metadata. We saw examples of the use of attributes for serialization and for synchronization, and we demonstrated how to implement and use custom attributes.

.NET simplifies the programming of memory management through an efficient, generational, automatic garbage collection facility. Finalization is non-deterministic, but you can support deterministic cleanup by implementing the dispose pattern or using the delete operator explicitly.