

CHAPTER 4

Implement data access

A program can use variables to store values as it is running, but it needs a means of persisting data. We have already touched on data storage at several points in this book, in Skill 3.1 in the section “Choose the appropriate data collection type,” we considered how best to map information in our application onto the data types that C# offers. Later in the section “Using JSON,” we saw how data in objects can be serialized. And finally, in the “Use the Entity Framework to design your data storage” section we created classes that were automatically mapped into database tables for use in an application. Before you read this chapter, it is worth going over those topics so that they are fresh in your mind.

In this chapter we’re going to bring all of these different data storage and access abilities together and consider how a C# program can store and manipulate data; starting with the use and management of file storage before moving into databases, and then onto the use of Language Integrated Query (LINQ). Then we’ll take a look at serialization in detail and finally explore the data collection facilities offered by the .NET framework.

Skills in this chapter:

- Skill 4.1: Perform I/O operations
- Skill 4.2: Consume data
- Skill 4.3: Query and manipulate data and objects by using LINQ
- Skill 4.4: Serialize and deserialize data
- Skill 4.5: Store data in and retrieve data from collections

SKILL 4.1: PERFORM I/O OPERATIONS

In this section we will look at the fundamental input/output (I/O) operations that underpin data storage in applications. You will discover how files are managed by the operating system and the .NET Framework libraries that allow programs to store and load data. File access is a very slow activity when compared with the speed of modern processors, so we are also going to investigate the use of asynchronous i/o, which can be used to keep an application responsive even when it is reading or writing large amounts of data.

This section covers how to:

- Read and write files and streams
- Read and write from the network by using classes in the System.Net namespace
- Implement asynchronous I/O operations

Read and write files and streams

A stream is a software object that represents a stream of data. The .NET framework provides a `Stream` class that serves as the parent type for a range of classes that can be used to read and write data. There are three ways that a program can interact with a stream:

- Write a sequence of bytes to a stream
- Read a sequence of bytes from a stream
- Position the “file pointer” in a stream

The file pointer is the position in a stream where the next read or write operation will take place. A program can use the `Seek` method provided by the stream to set this position. For example, a program can search through a file stream connected to formatted records for one that has a particular name.

The `Stream` class is abstract and serves as a template for streams that connect to actual storage resources. It is a very good example of how C#



of an encryption process into a byte array. It would be very easy to redirect the encrypted data produced in that program to a file or a network connection by using a different type of stream object. Figure 4-2 shows how the `System.IO.Stream` type is the base type for a large number of classes that provide different forms of stream connections.

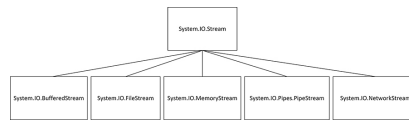


FIGURE 4-1 Some of the Stream types

The child classes all contain the stream behaviors that allow data to be transferred, for example the `Stream` method `Write` can be used on any of them to write bytes to that stream. However, how each type of stream created is dependent on that stream type. For example, to create a `FileStream` a program must specify the path to the file and how the file is going to be used. To create a `MemoryStream` a program must specify the buffer in memory to be used.

Use FileStream

The `FileStream` object provides a stream instance connected to a file. The stream object instance converts calls into the stream into commands for the filesystem on the computer running the program. The file system provides the interface to the physical device performing the data storage for the computer. Figure 4-2 shows how this works. A call of the `Write` method in a stream object will generate a request to the file system to write the data to the storage device.

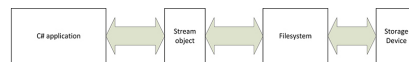


FIGURE 4-2 The Stream object

Listing 4-1 shows how a program can use a `FileStream` to create an output stream connected to a new or existing file. The program writes a block of bytes to that stream. It then creates a new stream that is used to read the bytes from the file. The bytes to be written are obtained by encoding a text string.

LISTING 4-1 Using a `FileStream`

[Click here to view code image](#)

```

using System;
using System.IO;
using System.Text;

namespace LISTING_4_1_Using_a_FileStream
{
    class Program
    {
        static void Main(string[] args)
        {
            // Writing to a file
            FileStream outputStream = new FileStream(
                OpenOrCreate, FileAccess.Write);
            string outputMessageString = "Hello wor
            byte[] outputMessageBytes = Encoding.UTF8.GetBytes(outputMessageString);
            outputStream.Write(outputMessageBytes, 0, outputMessageBytes.Length);
            outputStream.Close();

            FileStream inputStream = new FileStream(
                File.OpenRead, FileAccess.Read);
            long fileLength = inputStream.Length;
            byte[] readBytes = new byte[fileLength];
            inputStream.Read(readBytes, 0, (int)fileLength);
            string readString = Encoding.UTF8.GetString(readBytes);
            inputStream.Close();
            Console.WriteLine("Read message: {0}", readString);

            Console.ReadKey();
        }
    }
}
  
```

Control file use with `FileMode` and `FileAccess`

A stream can be associated with reading, writing, or updating a file. The base `Stream` class provides properties that a program can use to determine the abilities of a given stream instance (whether a program can read, write, or seek on this stream).

The `FileMode` enumeration is used in the constructor of a `FileStream` to indicate how the file is to be opened. The following modes are available:

- **FileMode.Append** Open a file for appending to the end. If the file exists, move the seek position to the end of this file. If the file does not exist; create it. This mode can only be used if the file is being opened for writing.
- **FileMode.Create** Create a file for writing. If the file already exists, it is overwritten. Note that this means the existing contents of the file are lost.



- **FileMode.Open** Open an existing file. An exception is thrown if the file does not exist. This mode can be used for reading or writing.
- **FileMode.OpenOrCreate** Open a file for reading or writing. If the file does not exist, an empty file is created. This mode can be used for reading or writing.
- **FileMode.Truncate** Open a file for writing and remove any existing contents.

The `FileAccess` enumeration is used to indicate how the file is to be used. The following access types are available:

- **FileAccess.Read** Open a file for reading.
- **FileAccess.ReadWrite** Open a file for reading or writing.
- **FileAccess.Write** Open a file for writing.

You can see these used in [Listing 4-1](#). If a file stream is used in a manner that is incompatible with how it is opened, the action will fail with an exception.

Convert text to binary data with Unicode

A stream can only transfer arrays of bytes to and from the storage device, so the program in [Listing 4-1](#) uses the `Encoding` class from the `System.Text` namespace. The `UTF8` property of this class provides methods that will encode and decode Unicode text. We looked at Unicode in the section “String comparison and cultures,” in [Skill 2-7](#).

Unicode is a mapping of character symbols to numeric values. The `UTF8` encoding maps Unicode characters onto 8-bit values that can be stored in arrays of bytes. Most text files are encoded using `UTF8`. The `Encoding` class also provides support for other encoding standards including `UTF32` (Unicode encoding to 32-bit values) and `ASCII`.

The `GetBytes` encoding method takes a `C#` string and returns the bytes that represent that string in the specified encoding. The `GetString` decoding method takes an array of bytes and returns the string that a buffer full of bytes represents.

Dispose and FileStream objects

The `Stream` class implements the `IDisposable` interface shown in [Skill 2-4](#). This means that any objects derived from the `Stream` type must also implement the interface. This means that we can use the `C#` `using` construction to ensure that files are closed when they are no longer required. [Listing 4-2](#) shows how this works.

LISTING 4-2 FileStream and IDisposable

[Click here to view code image](#)

```
using (FileStream outputStream = new FileStream("Out.txt",
    FileAccess.Write))
{
    string outputMessageString = "Hello world";
    byte[] outputMessageBytes = Encoding.UTF8.GetBytes(outputMessageString);
    outputStream.Write(outputMessageBytes, 0, outputMessageBytes.Length);
}
```

Work with text files

The filesystem makes no particular distinction between *text* files and *binary* files. We have already seen how we can use the `Encoding` class to convert Unicode text into an array of bytes that can be written into a binary file. However, the `C#` language provides stream classes that make it much easier to work with text. The `TextWriter` and `TextReader` classes are abstract classes that define a set of methods that can be used with text.

The `StreamWriter` class extends the `TextWriter` class to provide a class that we can use to write text into streams. [Listing 4-3](#) shows how the `StreamWriter` and `StreamReader` classes can be used with text files. It performs the same task as the program in [Listing 4-1](#), but it is much more compact.

LISTING 4-3 StreamWriter and StreamReader

[Click here to view code image](#)

```
using (StreamWriter writeStream = new StreamWriter("Out.txt"))
{
    writeStream.Write("Hello world");
}

using (StreamReader readStream = new StreamReader("Out.txt"))
{
    string readString = readStream.ReadToEnd();
    Console.WriteLine("Text read: {0}", readString);
}
```

Chain streams together

The `Stream` class has a constructor that will accept another stream as a parameter, allowing the creation of chains of streams. [Listing 4-4](#) shows how to use the `GZipStream` from the `System.IO.Compression` namespace.



[Click here to view code image](#)

```
using (FileStream writeFile = new FileStream("CompT
FileAccess.Write))
{
    using (GZipStream writeFileZip = new GZipStream
CompressionMode.Compress))
    {
        using (StreamWriter writeFileText = new Str
        {
            writeFileText.Write("Hello world");
        }
    }
}

using (FileStream readFile = new FileStream("CompTe
FileAccess.Read))
{
    using (GZipStream readFileZip = new GZipStream(
CompressionMode.Decompress))
    {
        using (StreamReader readFileText = new Stre
        {
            string message = readFileText.ReadToEnd
            Console.WriteLine("Read text: {0}", mes
        }
    }
}
```

Use the File class

The `File` class is a "helper" class that makes it easier to work with files. It contains a set of static methods that can be used to append text to a file, copy a file, create a file, delete a file, move a file, open a file, read a file, and manage file security. [Listing 4-5](#) shows some of the features of the `File` class in action.

LISTING 4-5 The File class

[Click here to view code image](#)

```
File.WriteAllText(path: "TextFile.txt", contents:

File.AppendAllText(path: "TextFile.txt", contents

if (File.Exists("TextFile.txt"))
    Console.WriteLine("Text File exists");

string contents = File.ReadAllText(path:"TextFile.t
Console.WriteLine("File contents: {0}", contents);

File.Copy(sourceFileName: "TextFile.txt", destFile

using (TextReader reader = File.OpenText(path: "Cop
{
    string text = reader.ReadToEnd();
    Console.WriteLine("Copied text: {0}", text);
}
```

Handle stream exceptions

Exceptions are situations where it is not meaningful for a given thread of execution to continue. The thread can raise an exception and pass control to a handler that will attempt to resolve the situation in a sensible way. You first saw exceptions in [Skill 1.5, "Implement exception handling."](#) When creating applications that use streams you need to ensure that your code can deal with any exceptions that might be thrown by the stream. These can happen at any time during the use of a stream. Our application may try to open a file that does not exist, or a given storage device may become full during writing. It is also possible that threads in a multi-threaded application can "fight" over files. If one thread attempts to access a file already in use by another, this will lead to exceptions being thrown.

With this in mind you should ensure that production code that opens and interacts with streams is protected by `try-catch` constructions. There are a set of file exceptions that are used to indicate different error conditions. [Listing 4-6](#) shows how a program can detect the `FileNotFoundException` and respond to that in a different way to other file exceptions.

LISTING 4-6 Stream exceptions


[Click here to view code image](#)

```
try
{
    string contents = File.ReadAllText(path: "Testf
    Console.WriteLine(contents);
}
catch(FileNotFoundException notFoundEx)
{
    // File not found
    Console.WriteLine(notFoundEx.Message);
}
```



[illegible]

◀ ▶



attributes assigned to the file.

- **FileAttributes.ReadOnly** The file cannot be written.
- **FileAttributes.System** The file is part of the operating system and is used by it.
- **FileAttributes.Temporary** The file is a temporary file that will not be required when the application has finished. The file system will attempt to keep this file in memory to improve performance.

This information is exposed to C# programs by means of the `FileInfo` class. Listing 4-8 shows how a program can obtain the `FileInfo` information about a file and then work with the attribute information. The program creates a new file and then obtains the `FileInfo` object that represents the file. It uses the `Attributes` property of the `FileInfo` object to make the file `readOnly` and then removes the `readOnly` attribute.

LISTING 4-8 Using `FileInfo`

[Click here to view code image](#)

```
string filePath = "TextFile.txt";

File.WriteAllText(path: filePath, contents: "This t
FileInfo info = new FileInfo(filePath);
Console.WriteLine("Name: {0}", info.Name);
Console.WriteLine("Full Path: {0}", info.FullName);
Console.WriteLine("Last Access: {0}", info.LastAcce
Console.WriteLine("Length: {0}", info.Length);
Console.WriteLine("Attributes: {0}", info.Attribute
Console.WriteLine("Make the file read only");
info.Attributes |= FileAttributes.ReadOnly;
Console.WriteLine("Attributes: {0}", info.Attribute
Console.WriteLine("Remove the read only attribute")
info.Attributes &= ~FileAttributes.ReadOnly;
Console.WriteLine("Attributes: {0}", info.Attribute
```

You can use a `FileInfo` instance to open a file for reading and writing, moving a file, renaming a file, and also modifying the security settings on a file. Some of the functions provided by a `FileInfo` instance duplicate those provided by the `File` class. The `File` class is most useful when you want to perform an action on a single file. The `FileInfo` class is most useful when you want to work with a large number of files. In the next section you will discover how to get a collection of `FileInfo` items from a directory and work through them.

Use the `Directory` and `DirectoryInfo` classes

A file system can create files that contain collections of file information items. These are called *directories* or *folders*. Directories can contain directory information about directories, which allows a user to *nest* directories to create tree structures.

As with files, there are two ways to work with directories: the `Directory` class and the `DirectoryInfo` class. The `Directory` class is like the `File` class. It is a static class that provides methods that can enumerate the contents of directories and create and manipulate directories. Listing 4-9 shows how a program can use the `Directory` class to create a directory, prove that it exists, and then delete it. Note that if a program attempts to delete a directory that is not empty an exception will be thrown.

LISTING 4-9 The `Directory` class

[Click here to view code image](#)

```
Directory.CreateDirectory("TestDir");

if (Directory.Exists("TestDir"))
    Console.WriteLine("Directory created successful

Directory.Delete("TestDir");

Console.WriteLine("Directory deleted successfully")
```

An instance of the `DirectoryInfo` class describes the contents of one directory. The class also provides methods that can be used to create and manipulate directories. Listing 4-10 performs the same functions as Listing 4-9 using the `DirectoryInfo` class.

LISTING 4-10 The `DirectoryInfo` class

[Click here to view code image](#)

```
DirectoryInfo localDir = new DirectoryInfo("TestDir

localDir.Create();

if(localDir.Exists)
    Console.WriteLine("Directory created successful

localDir.Delete();
```



Files and paths

A *path* defines the location of a file on a storage device. In all the example programs above we have simply given the path as a string of text. In this case the file or directory being created will be located in the same directory as the program that is running and will have the name given. If you want to store files in different places on the computer you need to create more complex paths.

Paths can be *relative* or *absolute*. A relative path specifies the location of a file relative to the folder in which the program is presently running. Up until now all the paths that we have specified have been relative to the current directory. When expressing paths, the character "." (period) has a special significance. A single period "." means the current directory. A double period ".." means the directory above the present one. You can use relative paths to specify a file in a parent directory, or a file in a directory in another part of the tree. Next you can see the path used to locate the image directory that is provided with the sample programs for this text. The @ character at the start of the string literal marks the string as a *verbatim* string. This means that any escape characters in the string will be ignored. This is useful because otherwise the backslash characters in the string might be interpreted as escape characters.

[Click here to view code image](#)

```
string imagePath = @"..\..\..\images\";
```

The program is running in the debug directory. The path must "climb" up through four parent directories to find the `images` directory. The diagram in [Figure 4-4](#) shows how the directories are structured.

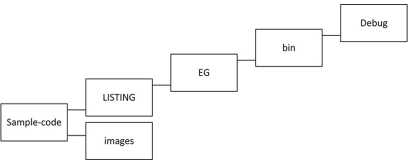


FIGURE 4-4 Navigating a relative path

An absolute path includes the drive letter and identifies all the sub-directories in the path to the file. The statement here gives the path to the document `test.txt` in the `Documents` folder on a machine.

[Click here to view code image](#)

```
string absPath = @"c:\users\rob\Documents\test.txt"
```

The path to a file contains two elements: the directories in the path and the name of the file in the directory. The `Path` class provides a lot of very helpful methods that can be used to work with paths in programs. It provides methods to remove filenames from full paths, change the extension on a filename, and combine filenames and directory paths. [Listing 4-11](#) shows some of the ways that `Path` can be used.

LISTING 4-11 Using Path

[Click here to view code image](#)

```
string fullName = @"c:\users\rob\Documents\test.txt";

string dirName = Path.GetDirectoryName(fullName);
string fileName = Path.GetFileName(fullName);
string fileExtension = Path.GetExtension(fullName);
string lisName = Path.ChangeExtension(fullName, ".1");
string newTest = Path.Combine(dirName, "newtest.txt");

Console.WriteLine("Full name: {0}", fullName);
Console.WriteLine("File directory: {0}", dirName);
Console.WriteLine("File name: {0}", fileName);
Console.WriteLine("File extension: {0}", fileExtension);
Console.WriteLine("File with lis extension: {0}", lisName);
Console.WriteLine("New test: {0}", newTest);
```

When you run the program in [Listing 4-11](#) it produces the following output:

[Click here to view code image](#)

```
Full name: c:\users\rob\Documents\test.txt
File directory: c:\users\rob\Documents
File name: test.txt
File extension: .txt
File with lis extension: c:\users\rob\Documents\test.1
New test: c:\users\rob\Documents\newtest.txt
```

The `Path` class is very useful and should **always** be used in preference to manually working with the path strings. The `Path` class also provides methods that can generate temporary filenames.



Searching for files

directory. One overload of `GetFiles` can accept a search string. Within the search string the character `*` can represent any number of characters and the character `?` can represent a single character.

The program in **Listing 4-12** uses this form of `GetFiles` to list all of the `C#` source files that are in the example programs. Note that this program also provides a good demonstration of the use of recursion, in that the `FindFiles` method calls itself to deal with any directories found inside a given directory.

LISTING 4-12 C sharp programs

Click here to view code image

```
static void FindFiles(DirectoryInfo dir, string sea
{
    foreach (DirectoryInfo directory in dir.GetDire
    {
        FindFiles(directory, searchPattern);
    }

    FileInfo[] matchingFiles = dir.GetFiles(searchP
    foreach(FileInfo fileInfo in matchingFiles)
    {
        Console.WriteLine(fileInfo.FullName);
    }
}
static void Main(string[] args)
{
    DirectoryInfo startDir = new DirectoryInfo(@"..
    string searchString = "*.cs";

    FindFiles(startDir, searchString);

    Console.ReadKey();
}
```

The `Directory` class provides a method called `EnumerateFiles` that can also be used to enumerate files in this way.

Read and write from the network by using classes in the `System.Net` namespace

The .NET Framework provides a range of application programming interfaces that can interact with a TCP/IP (Transmission Control Protocol/Internet Protocol) network. `C#` programs can create network *socket* objects that can communicate over the network by sending unacknowledged *datagrams* using UDP (User Datagram Protocol) or creating managed connections using TCP (Transmission Control Protocol).

In this section we are going to focus on the classes in the `System.Net` namespace that allow a program to communicate with servers using the HTTP (HyperText Transport Protocol). This protocol operates on top of a TCP/IP connection. In other words, TCP/IP provides the connection between the server and client systems and HTTP defines the format of the messages that are exchanged over that connection.

An HTTP client, for example a web browser, creates a TCP connection to a server and makes a request for data by sending the HTTP GET command. The server will then respond with a page of information. After the response has been returned to the client the TCP connection is closed.

The information returned by the server is formatted using HTML (HyperText Markup Language) and rendered by the browser. In the case of an ASP (Active Server Pages) application (for example the one that we created at the beginning of **Chapter 3**) the HTML document may be produced dynamically by software, rather than being loaded from a file stored on the server.

HTTP was originally used for the sharing of human-readable web pages. However, now an HTTP request may return an XML or JSON formatted document that describes data in an application.

The REST (REpresentational State Transfer) architecture uses the GET, PUT, POST and DELETE operations of HTTP to allow a client to request a server to perform functions in a client-server application. The fundamental operation that is used to communicate with these and other servers is the sending of a “web request” to a server to perform an HTML command on the server, and now we are going to discover how to do this. Let’s look at three different ways to interact with web servers and consider their advantages and disadvantages. These are `WebRequest`, `WebClient`, and `HttpClient`.

WebRequest

The `WebRequest` class is an abstract base class that specifies the behaviors of a web request. It exposes a static factory method called `Create`, which is given a *universal resource identifier* (URI) string that specifies the resource that is to be used. The `Create` method inspects the URI it is given and returns a child of the `WebRequest` class that matches that resource. The `Create` method can create `HttpWebRequest`, `FtpWebRequest`, and `FileWebRequest` objects. In the case of a web site, the URI string will start with “http” or “https” and the `Create` method will return an `HttpWebRequest` instance.

The `GetResponse` method on an `HttpWebRequest` returns a `WebResponse` instance that describes the response from the server. Note that this response is not the web page itself, but an object that describes the



a stream from which the webpage text can be read. [Listing 4-13](#) shows how this works.

LISTING 4-13 httpWebRequest

[Click here to view code image](#)

```
WebRequest webRequest = WebRequest.Create("https://  
WebResponse webResponse = webRequest.GetResponse();  
  
using (StreamReader responseReader = new StreamRead  
{  
    string siteText = responseReader.ReadToEnd();  
    Console.WriteLine(siteText);  
}
```

Note that the use of using around the StreamReader ensures that the input stream is closed when the web page response has been read. It is important that either this stream or the WebResponse instance are explicitly closed after use, as otherwise the connection will not be reused and a program might run out of web connections.

Using WebRequest instances to read web pages works, but it is rather complicated. It does, however, have the advantage that a program can set a wide range of properties on the web and request to tailor it to particular server requirements. This flexibility is not available on some of the other methods we are going to consider.

The code in [Listing 4-13](#) is synchronous, in that the program will wait for the web page response to be generated and the response to be read. It is possible to use the WebRequest in an asynchronous manner so that a program is not paused in this way. However, the programmer has to create event handlers to be called when actions are completed.

WebClient

The WebClient class provides a simpler and quicker way of reading the text from a web server. [Listing 4-14](#) shows how this is achieved. There is now no need to create a stream to read the page contents (although you can do this if you wish) and there is no need to deal with the response to the web request before you can obtain the reply from the server. [Listing 4-14](#) shows how this works.

LISTING 4-14 WebClient

[Click here to view code image](#)

```
WebClient client = new WebClient();  
string siteText = client.DownloadString("http://www  
Console.WriteLine(siteText);
```

The WebClient class also provides methods that can be used to read from the server asynchronously. [Listing 4-15](#) is used in a Windows Presentation Foundation (WPF) application to read the contents of a web page for display in a window.

LISTING 4-15 WebClient async

[Click here to view code image](#)

```
async Task<string> readWebpage(string uri)  
{  
    WebClient client = new WebClient();  
    return await client.DownloadStringTaskAsync(uri)  
}
```

HttpClient

The HttpClient is important because it is the way in which a Windows Universal Application can download the contents of a website. Unlike the WebRequest and the WebClient classes, an HttpClient only provides asynchronous methods. It can be used in a very similar manner to the WebClient, as shown in [Listing 4-16](#).

LISTING 4-16 HttpClient

[Click here to view code image](#)

```
async Task<string> readWebpage(string uri)  
{  
    HttpClient client = new HttpClient();  
    return await client.GetStringAsync(uri);  
}
```

Exception handling

As with file handling, loading information from the Internet is prone to error. Network connections may be broken or servers may be unavailable. This means that web request code should be enclosed in appropriate exception handlers. The code next is part of the program in [Listing 4-16](#); it catches exceptions thrown by the asynchronous loading method and displays a MessageBox containing error information.

[Click here to view code image](#)



```

try
{
    string webText = await readWebpage(PageUriTextB
ResultTextBlock.Text = webText;
}
catch (Exception ex)
{
    var dialog = new MessageDialog(ex.Message, "Req
await dialog.ShowAsync();
}
}

```

Implement asynchronous I/O operations

Up until now all the file input/output in our example programs has been *synchronous*. A program calling a method to perform a file operation must wait for the method to complete before it can move onto the next statement. A user of the program has to wait for the file action to complete before they can do anything else, which might lead to a very poor user experience.

In [Chapter 1](#), in the Skill "Using `async` and `await`," you saw that a program can use tasks to perform asynchronous background execution of methods. It is worth taking a look at that section to refresh your understanding of these elements before reading further.

The file operations provided by the `File` class do not have any asynchronous versions, so the `FileStream` class should be used instead. [Listing 4-17](#) shows a function that writes an array of bytes to a specified file using asynchronous writing.

LISTING 4-17 Asynchronous file writing

[Click here to view code image](#)

```

async Task WriteBytesAsync(string filename, byte [])
{
    using (FileStream outStream = new FileStream(fi
{
        await outStream.WriteAsync(items, 0, items.
    }
}

```

The demonstration program is a Windows Presentation Foundation application that contains both synchronous and asynchronous file writing methods. [Figure 4-5](#) shows the program display. The user can write a large number of values to a file either synchronously or asynchronously depending on which start button they select. They can also test the responsiveness of the application by selecting the `Get Time` button, which will display the current time. When the synchronous version of the writer is running, they should note that the user interface becomes unresponsive for a short while.

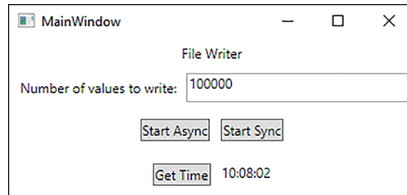


FIGURE 4-5 Async File Writer demo

Handling exceptions in asynchronous methods

If any exceptions are thrown by the asynchronous file write method they must be caught and a message displayed for the user. This will only happen if the `WriteBytesAsync` method returns a `Task` object that is awaited when the `WriteBytesAsync` method is called. [Listing 4-18](#) shows a button event handler that does this correctly and catches exceptions that may be thrown by the file write action.

LISTING 4-18 File exceptions

[Click here to view code image](#)

```

private async void StartTaskButton_Click(object sen
{
    byte[] data = new byte[100];

    try
    {
        // note that the filename contains an inval
        WriteBytesAsyncTask("demo.dat", data);
    }
    catch (Exception writeException)
    {
        MessageBox.Show(writeException.Message, "Fi
    }
}

```

When you run the example program for [Listing 4-18](#) the program displays



(see Figure 4-6) and display a message box. The other exception will not be handled correctly.

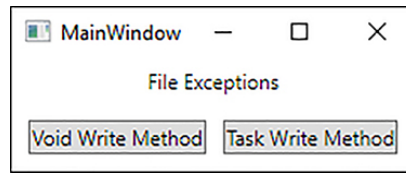


FIGURE 4-6 Catching file exceptions

The only asynchronous methods that should return `void` are the actual event handlers for the windows controls. Every other asynchronous method must return either a result or a `Task`, so that any exceptions thrown by the method can be dealt with correctly.

SKILL 4.2: CONSUME DATA

In the previous section we saw how to create files and store data in them. In this section you are going to look at data storage techniques that build on a file store to underpin working applications. Initially you will look at how a database engine can expose information to applications. Then you're going to take a look at how to use XML and JSON to impose structure on a file of stored data. Finally, you are going to investigate web services, which allow a program to expose data to other programs over a network in a portable manner.

This section covers how to:

- Retrieve data from a database
- Update data in a database
- Consume JSON and XML data
- Retrieve data by using web services

Retrieve data from a database

In Skill 3.2, in the "Use the Entity Framework to design your data storage" section you saw how to use a class-based design to express the data storage needs of an application to store information about music tracks. We designed a class called `MusicTrack` and then used the Entity Framework toolchain in Visual Studio to automatically create the underlying database.

Before you read any further you might like to revisit that section to remind yourself of what we were trying to do, and how we did it. If you want to use the database samples in this section you can perform the steps in Skill 3.2 to create your own local database file.

A single SQL database server can provide data storage to be shared between a large number of clients. The server can run on the same machine as the program that is using the database or the server can be accessed via a network connection. Developers can work with development version of the database server running on their machine, before creating a dedicated server for the published application. The MusicTracks application that we worked with in Skill 3.2 used a local database running on a machine in development mode. Later in this section we will consider how to manage the environment of an ASP application to allow it to use a remote database server when it runs on a production server.

Data in a database

The code next shows the `MusicTrack` class that is used in the MusicTracks application. The class contains data members that store the `Artist`, `Title`, and `Length` of a music track. The class also contains an integer `ID` value, which will be used by the database to allow it to uniquely identify each music track that is stored.

Click here to view code image

```
public class MusicTrack
{
    public int ID { get; set; }
    public string Artist { get; set; }
    public string Title { get; set; }
    public int Length { get; set; }
}
```

The Entity Framework toolchain uses the class definition to produce a table in a database that has the required data storage. Table 4.1 shows the database table that was created for the `MusicTrack` class. Each row in the table equates to an instance of the `MusicTrack` class. The table contains three songs. Each song has a different `ID` value. The database has been configured to automatically create `ID` values when a new `MusicTrack` entry is created.



TABLE 4-1 MusicTrack table

ID	Artist	Title	Length
1	Rob Miles	My Way	150
2	Fred Bloggs	His Way	150
3	The Bloggs Singers	Their Way	200

From a data design point of view a table in a database can be considered as a collection of objects. In other words, the table in Table 4-1 can be thought of as a list of references to `MusicTrack` objects in a C# program. Creating a program to read the data in a database table, however, is the same as accessing an element in a C# list. A program has to make a connection with the database and then send the database a command to request the `MusicTrack` information.

Read with SQL

The database in the MusicTracks application is managed by a server process that accepts commands and acts on them. The commands are given in a format called *Structured Query Language* (SQL). SQL dates back to the 1970's. It is called a *domain specific language* because it is used solely for expressing commands to tell a database what to do.

SQL is very useful. For example, one day the user of your music track program might ask you to make the program produce a list of all the tracks by a particular artist. This would be easy to do if all the music tracks are held in a list. You can write a `for` loop that works through the list looking for tracks with a particular name. Then, the next day the user might ask for a list of tracks ordered by track length, and on the next day you might get asked for all the artists who have recorded a track called "My Way" that is longer than 120 seconds. Each time you are asked for a new view of the data you have to write some more C# to search through the list of tracks and produce the result. However, if the music track information is held in a database, each of these requests can be satisfied by creating an SQL query to obtain the data.

The first SQL query that we are going to perform on the MusicTracks database has the form shown next. The `*` character is a "wildcard" that matches all of the entries in the table. This command tells the database server that our program wants to read all of the elements in the `MusicTrack` table.

```
SELECT * FROM MusicTrack
```

Now that we have our SQL command, let's discover how to present the command to the database. A program uses an SQL database in a similar way to a stream. The program creates an object that represents a connection to the database and then sends SQL commands to this object. The database connection mechanism is also organized in the same way as the input/output stream classes. The `DbConnection` class that represents a connection to a database is an abstract class that describes the behaviors of the connection in the same way that the `Stream` class is also abstract and describes the behaviors of streams. The `SqlConnection` class is a child of the `DbConnection` class and represents the implementation of a connection to an SQL database.

To make a connection to a database a program must create a `SqlConnection` object. The constructor for the `SqlConnection` class is given a *connection string* that identifies the database that is to be opened. Before we can begin to read from the database we need to consider how the connection string is used to manage the connection.

The connection string contains a number of items expressed as name-value pairs. For a server on a remote machine the connection string will contain the address of the server, the port on which the server is listening and a username/password pair that can authenticate the connection.

In the case of the MusicTracks application that we created earlier, this connection string was created automatically and describes a connection to a local database file held on the computer in the folder for the user. If you followed the steps in Chapter 3 to build your own application, the database file will be created on your machine.

The program in Listing 4-19 shows how a C# program can make a connection to a database, create an SQL query, and then execute this query on the database to read and print out information from the `MusicTrack` table. The program prints the Artist and Title of each music track in the database.

LISTING 4-19 Read with SQL

[Click here to view code image](#)

```
using System;
using System.Data.SqlClient;

namespace LISTING_4_19_Read_with_SQL
{
    class Program
    {
        static void Main(string[] args)
        {
```



```

using (SqlConnection connection = new SqlConnection(connectionString))
{
    connection.Open();
    SqlCommand command = new SqlCommand("SELECT * FROM Artists", connection);

    SqlDataReader reader = command.ExecuteReader();

    while (reader.Read())
    {
        string artist = reader["Artist"];
        string title = reader["Title"];

        Console.WriteLine("Artist: {0} Title: {1}", artist, title);
    }
    Console.ReadKey();
}
}

```

The connection string in the program makes a connection to the MusicTracks database. Note that it is important that the connection string is **not** "hard wired" into program source code. The program in Listing 4-19 is just an example, not how you should create production code. Anyone obtaining the source code of this program can view the connection string and get access to the database and anyone obtaining the compiled version of the program can use a tool such as `ildasm` to view the MSIL code and extract the connection. You would also have to change the code of the program each time you want to use a different database.

Connection string management in ASP.NET

The database connection string to be used by an ASP.NET application is held in the configuration information for the solution. This is stored in the file `appsettings.json` in the solution. The location of this file is shown in Figure 4-7, which shows the solution files for the MusicTracks application.

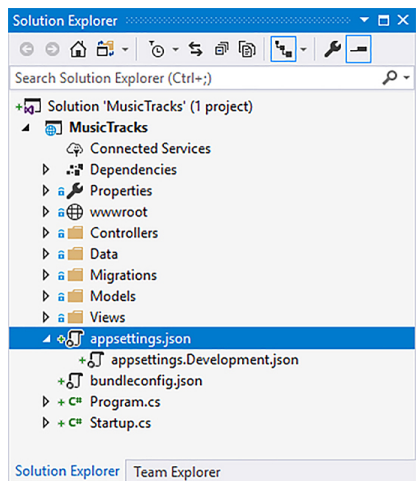


FIGURE 4-7 The `appsettings.json` file

The solution in Figure 4-7 has a setting file `appsettings.Development.json` that contains custom settings for use during development. If you add an `appsettings.Production.json` file to the solution, you can create settings information that will be used when the program is running on a production server. The `appsettings.json` file for the MusicTracks application on my machine contains the following:

[Click here to view code image](#)

```

{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "ConnectionStrings": {
    "MusicTracksContext": "Server=(localdb)\\mssql1;..."
  }
}

```

The `ConnectionStrings` element in the settings file contains the connection string for the MusicTracks database context. The database name that is created when the solution is created has a globally unique identifier (GUID) appended, so the name of the database on your machine will be different from mine. If you want to run the database samples that follow in this text you will have to copy your connection string into the example programs.



The setting information to be used is when a server is started and determined by an environment variable on the computer that is tested when the program starts running. The word environment is being used in two different contexts in this situation, which can be confusing.

-
- The screenshot shows the 'Main Tools' window in Visual Studio. The 'Online' tab is active. The 'Configuration' and 'Platform' dropdowns are both set to 'x64'. The 'Environment variables' section displays a table with one entry: 'Name: ASPNETCORE_ENVIRONMENT, Value: Development'. There are 'Add' and 'Remove' buttons to the right of the table. The left sidebar shows a tree view with 'Online' selected.

FIGURE 4-8 Setting the ASPNETCORE_ENVIRONMENT variable

In older ASP.NET applications the SQL settings are held in the web.config file, which is part of the solution. Developers then use XML transformations to override settings in the file to allow different SQL servers to be selected.

A program can make a query of a database by creating an `SqlCommand` instance. The constructor for an `SqlCommand` is given a string, which is the SQL query and the database connection, as shown in the statement from Listing 4-19.

```
SqlCommand command = new SqlCommand("SELECT * FROM !
```

The query is then executed as a reader command (because it is reading from the database). This operation returns an `SqlReader` instance as shown in the following statement.

```
SqlDataReader reader = command.ExecuteReader();
```

The `SqlReader` provides methods that can be used to move through the results returned by the query. Note that it is only possible to move forward through the results. The `Read` method moves the reader on to the next entry in the results. The `Read` method returns `False` when there are no more results. The individual items in the element can be accessed using their name, as shown here.

```
while (reader.Read())
{
    string artist = reader["Artist"].ToString();
    string title = reader["Title"].ToString();

    Console.WriteLine("Artist: {0} Title: {0}", art...
```

The first SQL command selected all of the elements in a table. You can change this so that you can filter the contents of the table using a query. Listing 4-20 shows a query that selects only music tracks from the artist “Rob Miles.”

[Click here to view code image](#)

```
SELECT * FROM MusicTrack WHERE Artist='Rob Miles'
```

A program can use the SQL UPDATE command to update the contents of



same ID. If the WHERE component of the command selected multiple entries in the database, they would all be updated.

[Click here to view code image](#)

```
UPDATE MusicTrack SET Artist='Robert Miles' WHERE ID=1
```

When the update is performed it is possible to determine how many elements are updated. The program in [Listing 4-21](#) shows how this is done.

LISTING 4-21 Update with SQL

[Click here to view code image](#)

```
using (SqlConnection connection = new SqlConnection(
{
    connection.Open();
    dumpDatabase(connection);
    SqlCommand command = new SqlCommand(
        "UPDATE MusicTrack SET Artist='Robert Miles'
    int result = command.ExecuteNonQuery();
    Console.WriteLine("Number of entries updated: {0}");
    dumpDatabase(connection);
})
```

SQL injection and database commands

Suppose you want to allow the user of your program to update the name of a particular track. Your program can read the information from the user and then construct an SQL command to perform the update. [Listing 4-22](#) shows how this can be done:

LISTING 4-22 SQL injection

[Click here to view code image](#)

```
Console.Write("Enter the title of the track to update: ");
string searchTitle = Console.ReadLine();
Console.Write("Enter the new artist name: ");
string newArtist = Console.ReadLine();
string sqlCommandText =
    "UPDATE MusicTrack SET Artist='" + newArtist + '";
```

If you run the program in [Listing 4-22](#), you will find that it does work. You can select a track by title and then change the artist for that track. However, this is very dangerous code. Consider the effect of a user running the program and then typing in the following information.

[Click here to view code image](#)

```
Enter the title of the track to update: My Way
Enter the new artist name: Fred'); DELETE FROM MusicTrack
```

The track title is fine, but the new artist name looks rather strange. What the user of the program has done is *injected* another SQL command after the UPDATE command. The command would set the new artist name to Fred and then perform an SQL DELETE command. The DELETE command does exactly what you would expect. It deletes the entire contents of the table.

This works because there is nothing to stop the user of the program from typing in the delimiter ' (single quote) to mark the end of a string and then adding new SQL statements after that. A malicious user can use SQL injection to take control of a database.

For this reason, you should never construct SQL commands directly from user input. You must use parameterized SQL statements instead. [Listing 4-23](#) shows how they work. The SQL command contains markers that identify the points in the query where text is to be inserted. The program then adds the parameter values that correspond to the marker points. The SQL server now knows exactly where each element starts and ends, making SQL injection impossible.

LISTING 4-23 Parameterized query

[Click here to view code image](#)

```
string sqlCommandText = "UPDATE MusicTrack SET Artist=" + newArtist + " WHERE ID=" + searchTitle + ";";

SqlCommand command = new SqlCommand(sqlCommandText, connection);
command.Parameters.AddWithValue("@artist", newArtist);
command.Parameters.AddWithValue("@searchTitle", searchTitle);
```

Asynchronous database access

The SQL commands that we have used have all used synchronous methods to evaluate queries and work through results. There are also asynchronous versions of the methods. A program can use the `SqlCommand.ExecuteNonQueryAsync()` method to execute a command asynchronously.



The `dumpDatabase` method in Listing 4-24 uses asynchronous database commands to create a listing of the contents of the MusicTrack database. This method is part of a WPF (Windows Presentation Foundation) application that also allows database editing. Note that to use this example on your machine you will have to set the database connection string to make a connection to a database on your machine.

LISTING 4-24 Asynchronous access

[Click here to view code image](#)

```
async Task<string> dumpDatabase(SqlConnection connec
{
    SqlCommand command = new SqlCommand("SELECT * F
    SqlDataReader reader = await command.ExecuteRea
    StringBuilder databaseList = new StringBuilder(
    while (await reader.ReadAsync())
    {
        string id = reader["ID"].ToString();
        string artist = reader["Artist"].ToString()
        string title = reader["Title"].ToString();

        string row = string.Format("{ID: {0} Artist:
        databaseList.AppendLine(row);
    }
    return databaseList.ToString();
}
```

Using SQL in ASP applications

If you examine the ASP `MusicTracks` application that we created at the start of Skill 3.2, you will not find any SQL commands in the code that implement the controller classes for this application. This is because database updates in an ASP application are performed using an `Update` method that accepts a modified instance of the class to be updated.

You can see this behavior in action by taking a look at the code in the `Edit` method in the `MusicTrackController.cs` source file, as shown next. The variable `musicTrack` contains the modified music track instance, as received from the web form. The variable `_context` contains the database context for this page. The code updates the database and catches any exceptions that may be thrown by this operation.

[Click here to view code image](#)

```
try
{
    _context.Update(musicTrack); // update the
    await _context.SaveChangesAsync(); // save all
}
catch (DbUpdateConcurrencyException)
{
    // deal with exceptions
}
```

The ASP application uses embedded C# code in the web page (using a feature called `Razor`), which iterates through the database contents and presents a view of the data. The code shown next is part of the `index.cshtml` file for the `MusicTracks` application. `Razor` enables C# statements to be inserted into the HTML page description to programmatically generate some of the content. The C# elements on the page are preceded by an `@` character. In the following code you can see how a `foreach` loop is used to iterate through the items in the database and call the `DisplayFor` method for each item.

[Click here to view code image](#)

```
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.)
        </td>
        <td>
            <a asp-action="Edit" asp-route-id="@
            <a asp-action="Details" asp-route-id
            <a asp-action="Delete" asp-route-id
        </td>
    </tr>
}
```

Figure 4-9 shows the web output generated by this code. Note that for each item that is displayed there are also links to the behaviors to edit, view the details, or delete the item. If you look at the earlier code you can see that the routing information for each link contains the ID of that item, which is how the server will know which item has been selected.



Rob Miles	My Way	150	Edit Details Delete
Fred Bloggs	Ha Way	150	Edit Details Delete
The Bloggs Singers	Their Way	200	Edit Details Delete

FIGURE 4-9 Track list in MusicTracks

Consume JSON and XML data

You have seen that it is very easy for a program to load the contents of a web page. In Listing 4-14 you found that the `WebClient` class can do this with a single method call. A lot of data resources are now provided to applications via a structured data documents that can be obtained in this way. There are two popular formats for such data, JavaScript Object Notation (JSON) and eXtensible Markup Language (XML). In this section you are going to see examples of each in use.

Consume JSON data

In Skill 3.1 in the “Using JSON” section, you saw that JSON (JavaScript Object Notation) is a means by which applications can exchange data. A JSON document is a plain text file that contains a structured collection of name and value pairs. You also discovered how to read and write JSON files using the `Newtonsoft.Json` library. Read that section again to refresh your memory about JSON before continuing with this section.

The program in Listing 4-25 consumes a JSON feed provided by the National Aeronautics and Space Administration (NASA). The feed is updated every day with a new picture or video from the NASA archives. The method `getImageOfDay` returns an object describing the image of the day for a particular address on the NASA site. It uses the `Newtonsoft` libraries described in Skill 3.1 to convert the text loaded from the NASA web page into an instance of the `ImageOfDay` class with all of the fields initialized.

LISTING 4-25 NASA.JSON

[Click here to view code image](#)

```
public class ImageOfDay
{
    public string date { get; set; }
    public string explanation { get; set; }
    public string hdurl { get; set; }
    public string media_type { get; set; }
    public string service_version { get; set; }
    public string title { get; set; }
    public string url { get; set; }
}

async Task<ImageOfDay> GetImageOfDay(string imageURL)
{
    string NASAJson = await readWebpage(imageURL);

    ImageOfDay result = JsonConvert.DeserializeObject<ImageOfDay>(NASAJson);

    return result;
}
```

For this to work I had to create an `ImageOfDay` class that exactly matches the object described by the JSON feed from NASA. One way to do this is to use the website <http://jsonzsharp.com> (<http://jsonzsharp.com>), which will accept a web address that returns a JSON document and then automatically generates a C# class as described in the document.

The code that displays the image when the user clicks a button to perform the load is shown next. The method `displayUrl` asynchronously loads the image and displays it on the screen. The code also gets the descriptive text for the image and displays it in a text box.

[Click here to view code image](#)

```
private async void LoadButtonClicked(object sender,
{
    try
    {
        ImageOfDay imageOfDay = await getImageOfDay(
            "https://api.nasa.gov/planetary/apo

        if (imageOfDay.media_type != "image")
        {
            MessageBox.Show("It is not an image tod
            return;
        }

        DescriptionTextBlock.Text = imageOfDay.expl

        await displayUrl(imageOfDay.url);
    }
    catch (Exception ex)
    {
        MessageBox.Show("Fetch failed: {0}", ex.Mes
    }
}
```

The program works well, although at the moment it always shows the same image, which is specified in the address in the previous code. It



NASA site. This can only be used for a few downloads an hour. If you want to use this service for your application you can apply for a free API key from NASA. Figure 4-10 shows the output from the program.

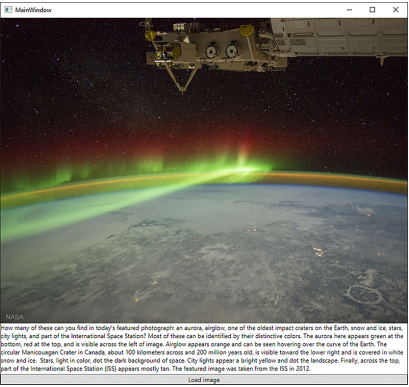


FIGURE 4-10 NASA image display

Consume XML data

In Skill 3.1 in the “JSON and XML” section we explored the difference between JSON and XML. The XML language is slightly more expressive than JSON, but XML documents are larger than equivalent JSON documents and this, along with the ease of use of JSON, has led to JSON replacing XML. However, a lot of information is still expressed using XML. In Skill 3.1 we saw an XML document that described a `MusicTrack` instance.

[Click here to view code image](#)

```
<?xml version="1.0" encoding="utf-16"?>
<MusicTrack xmlns:xsi="http://www.w3.org/2001/XMLSchema"
  <Artist>Rob Miles</Artist>
  <Title>My Way</Title>
  <Length>150</Length>
</MusicTrack>
```

One way to consume such a document is to work through each element in turn. The `System.Xml` namespace contains a set of classes for working with XML documents. One of these classes is the `XmlTextReader` class. An instance of the `XmlTextReader` class will work through a stream of text and decode each XML element in turn.

The program in Listing 4-26 shows how this is done. It creates a `StringReader` stream that is used to construct an `XmlTextReader`. The program then iterates through each of the XML elements that are read from the stream, printing out the element information.

LISTING 4-26 XML elements

[Click here to view code image](#)

```
string XMLDocument = "<?xml version='1.0\' encoding="
  "<MusicTrack xmlns:xsi='http://www.w3.org/2001/
  \"xmlns:xsd='http://www.w3.org/2001/XMLSchema\"
  \"<Artist>Rob Miles</Artist>\" +
  \"<Title>My Way</Title>\" +
  \"<Length>150</Length>\" +
  \"</MusicTrack>\";

using (StringReader stringReader = new StringReader
{
  XmlTextReader reader = new XmlTextReader(string)

  while (reader.Read())
  {
    string description = string.Format("Type:{0}
      reader.NodeType.ToString(),
      reader.Name,
      reader.Value);
    Console.WriteLine(description);
  }
}
```

The output from this program gives each item in turn as shown next. Note that not all items have both names and values.

[Click here to view code image](#)

```
Type:XmlDeclaration Name:xml Value:version="1.0" en
Type:Element Name:MusicTrack Value:
Type:Whitespace Name: Value:
Type:Element Name:Artist Value:
Type:Text Name: Value:Rob Miles
Type:EndElement Name:Artist Value:
Type:Whitespace Name: Value:
Type:Element Name:Title Value:
```



```
Type:Element Name:Length Value:
Type:Text Name: Value:150
Type:EndElement Name:Length Value:
Type:EndElement Name:MusicTrack Value:
```

XML documents

It is possible to read information from an XML document by decoding each individual element, but this can be hard work to code. An easier approach is to use an `XmlDocument` instance. This creates a Document Object Model (DOM) in memory from which data can be extracted. Another advantage of a DOM is that a program can change elements in the DOM and then write out a modified copy of the document incorporating the changes.

The program in Listing 4-27 creates an `XmlDocument` instance from a string of XML describing a `MusicTrack` and then reads the artist and title information from it. The program checks to make sure that the document describes a `MusicTrack` before writing the information.

LISTING 4-27 XML DOM

[Click here to view code image](#)

```
XmlDocument doc = new XmlDocument();
doc.LoadXml(XMLDocument);

System.Xml.XmlElement rootElement = doc.DocumentElement;
// make sure it is the right element
if (rootElement.Name != "MusicTrack")
{
    Console.WriteLine("Not a music track");
}
else
{
    string artist = rootElement["Artist"].FirstChild.Value;
    Console.WriteLine(artist);
    string title = rootElement["Title"].FirstChild.Value;
    Console.WriteLine("Artist:{0} Title:{1}", artist, title);
}
```

An `XmlDocument` contains a hierarchy of items with a `rootElement` object the top of the hierarchy. A program can access items in an element by using a string indexer that contains the name of the required item. In the next section we will discover how to use Language Integrated Queries (LINQ) to work with XML documents.

Retrieve data by using Windows Communication Foundation (WCF)

You have already seen how an application can consume data from a service by using an HTTP request to download a JSON or XML document. The NASA image reader in Listing 4-25 works by converting a response from a server into an instance of a class that contains a description of the image and the web address from which the image can be downloaded. To use the information provided by the NASA server, a client program contains a C# class that matches the JSON document that is received from the server.

A client of a web service also uses an instance of an object to interact with a server. However, in this case, the client can call methods on the object to read data and also update information on the server. The object that is created is called a "proxy object." A call to a method on the proxy object will cause a request to be created that is sent to the service on the server.

When the server receives the request, it will then call the method code in a server object. The result to be returned by the method will then be packaged up into another network message, which is sent back to the client and then sends the return value back to the calling software. Don't worry about how the messages are constructed and transferred. You just need to create server methods and calls from the client systems.

The service also exposes a description of the methods that it provides. This is used by the development tool (in our case Visual Studio) to actually create the proxy object in the client program. This means that you can easily create a client application.

Create a web service

To discover how all of this works let's create a "Joke of the Day" service. This will return a string containing something suitably rib-tickling on request from the client. The user will be able to select the "strength" of the joke, ranging from 0 to 2 where 0 is mildly amusing and 2 is a guaranteed roll on the floor laughing experience.

The application will be made up of two parts, the server program that exports the service, and the client program that uses it. The server is created as a WCF (Windows Communication Foundation) Service application. The client will be an application that connects to the service and requests a joke.

The code next shows the single method that is provided by the service. This is part of the code in the server application. The `GetJoke` method accepts an integer and returns a string of text. The attributes `[ServiceContract]` and `[OperationContract]` denote that the interface and method are to be exposed as services.

[Click here to view code image](#)



```
using System.ServiceModel;

namespace JokeOfTheDay
{
    [ServiceContract]
    public interface IJokeOfTheDayService
    {
        [OperationContract]
        string GetJoke(int jokeStrength);
    }
}
```

Once you have the interface you now need a class that implements the method. This method will provide the service that the interface describes. As you can see from the code next, the method is quite simple, and so are the jokes. The input parameter is used to select one of three joke strings and return it. If there is no match to the input the string "Invalid strength" is returned.

[Click here to view code image](#)

```
public class JokeOfTheDayService : IJokeOfTheDayService
{
    public string GetJoke(int jokeStrength)
    {
        string result = "Invalid strength";
        switch (jokeStrength)
        {
            case 0:
                result =
                    "Knock Knock. Who's there? Oh, you've heard it";
                break;
            case 1:
                result =
                    "What's green and hairy and goes up and down? A goo";
                break;
            case 2:
                result =
                    "A horse walks into a bar and the barman asks 'Why'";
                break;
        }
        return result;
    }
}
```

If you create a service you can use the WCF Test Client to invoke the methods and view the results. This tool allows you to call methods in the service and view the results that they return. Figure 4-11 shows the test client in use. You can see the results of a call to the method with a parameter of 1.

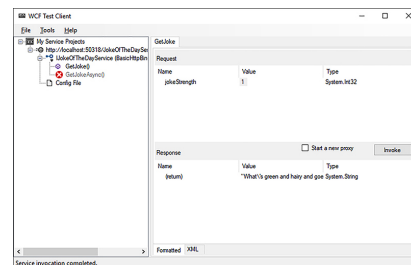


FIGURE 4-11 WCF Test Client

You can also view the service description in a browser as shown in Figure 4-12. This gives a link to the service description, as well as sample code that shows how to use it.



FIGURE 4-12 Service description in a browser

Joke of the Day Client

The client application can run on any machine that has a network connection and wants to consume the service. The client application needs to contain a connection to the `JokeOfTheDayService`. This is added to the client Visual Studio project as a reference like any other, by using the Solution Explorer pane in Visual Studio. Right-click on the **References** item and select **Add Service Reference** from the context menu, as shown in Figure 4-13.



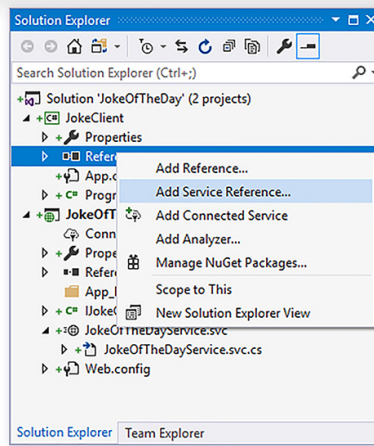


FIGURE 4-13 Adding a new service

At this point Visual Studio needs to find the description of the service that is to be used. The Add Service Reference dialog allows you to type in the network address of a service, and Visual Studio will then read the service description provided by the service. Figure 4-14 shows the dialog used to set up the client. You can see that the `JokeOfTheDay` service only provides one method. At the bottom of this dialog you can see the namespace that you want the service to have in the client application. Change this name to `JokeOfTheDayService`.

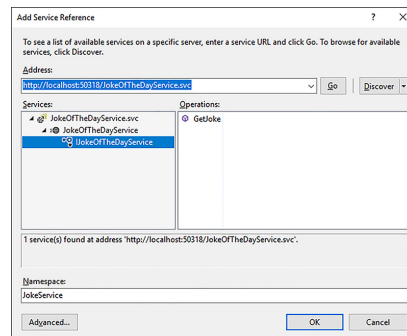


FIGURE 4-14 Adding a new service

Once the service has been added it now takes its place in the solution. Our application must now create a proxy object that will be used to invoke the methods in the service. The code next creates a service instance and calls the `GetJoke` method to get a joke from the service. Note that the example solution for Listing 4-28 contains two Visual Studio projects. One implements the server and the other the client.

LISTING 4-28 Web Service

[Click here to view code image](#)

```
using System;
using JokeClient.JokeService;

namespace JokeClient
{
    class Program
    {
        static void Main(string[] args)
        {
            using (JokeOfTheDayServiceClient jokeCl
                new JokeOfTheDayServiceClient())
            {
                Console.WriteLine(jokeClient.GetJok
            }

            Console.ReadKey();
        }
    }
}
```

SKILL 4.3: QUERY AND MANIPULATE DATA AND OBJECTS BY USING LINQ

In the previous section you saw how a program can work with stored data by using SQL commands. SQL commands are time consuming to create and processing the result of a query is hard work. Language INtegrated Query, or LINQ, was created to make it very easy for C# programmers to work with data sources. This section covers LINQ and how to use it.

Note that there are a lot of example programs in this section, and the LINQ statements might be a bit hard to understand at first. However, remember that you can download and run all of the example code.



- Query data by using operators, including projection, join, group, take, skip, aggregate
- Create method-based LINQ queries
- Query data by using query comprehension syntax
- Select data by using anonymous types
- Force execution of a query
- Read, filter, create, and modify data structures by using LINQ to XML

Sample application

In order to provide a good context for the exploration of LINQ we are going to develop the `MusicTracks` application used earlier in this book. This application allows the storage of music track data. [Figure 4-15](#) shows the classes in the system. Note that at the moment we are not using a database to store the class information. Later we will consider how to map this design onto a database.

In previous versions of the application, the `MusicTrack` class contained a string that gave the name of the artist that recorded the track. In the new design, a `MusicTrack` contains a reference to an `Artist` object that describes the artist that recorded the track. [Figure 4-15](#) shows how this works. If an artist records more than one track (which is very likely), the artist details will only be stored once and referred to by many `MusicTrack` instances.



FIGURE 4-15 Music tracks class design

The code next shows the C# code for the classes.

[Click here to view code image](#)

```

class Artist
{
    public string Name { get; set; }
}
class MusicTrack
{
    public Artist Artist { get; set; }
    public string Title { get; set; }
    public int Length { get; set; }
}
  
```

Now that you have the class design for the application, the next thing to do is create some sample data. This *must* be done programmatically. Testing a system by entering data by hand is a bad idea for a number of reasons. First, it is very time-consuming. Second, any changes to the data store design will mean that you will probably have to enter all the data again. And third, the act of creating the test data can give useful insights into your class design.

[Listing 4-29](#) shows code that creates some artists and tracks. You can increase the amount of test data by adding more artists and titles. In this version all of the artists recorded all of the tracks. A random number generator provides each track a random length in the range of 20 to 600 seconds. Note that because the random number generator has a fixed seed value the lengths of each track will be the same each time you run the program.

LISTING 4-29 Musictrack classes

[Click here to view code image](#)

```

string[] artistNames = new string[] { "Rob Miles", '
                                     "The Bloggs S.
string[] titleNames = new string[] { "My Way", "You.
                                     "Milky Way" };

List<Artist> artists = new List<Artist>();
List<MusicTrack> musicTracks = new List<MusicTrack>

Random rand = new Random(1);

foreach (string artistName in artistNames)
{
    Artist newArtist = new Artist { Name = artistName };
    artists.Add(newArtist);
    foreach (string titleName in titleNames)
    {
        MusicTrack newTrack = new MusicTrack
        {
            Artist = newArtist,
            Title = titleName,
            Length = rand.Next(20, 600)
        };
        musicTracks.Add(newTrack);
    }
}
}
  
```



```
Console.WriteLine("Artist:{0} Title:{1} Length:
track.Artist.Name, track.Title, track.Length
)
}
```

Use an object initializer

If you look at the code in [Listing 4-29](#) you will see that we are using *object initializer* syntax to create new instances of the music objects and initialize their values at the same time. This is a very useful C# feature that allows you to initialize objects when they are created without the need to create a constructor method in the class being initialized.

The code next shows how it works. The statement creates and initializes a new `MusicTrack` instance. Note the use of braces `{(and)}` to delimit the items that initialize the instance and commas to separate each value being used to initialize the object.

[Click here to view code image](#)

```
MusicTrack newTrack = new MusicTrack
{
    Artist = "Rob Miles",
    Title = "My Way",
    Length = 150
};
```

You don't have to initialize all of the elements of the instance; any properties not initialized are set to their default values (zero for a numeric value and null for a string). The properties to be initialized in this way must all be public members of the class.

Use a LINQ operator

Now that you have some data can use LINQ operators to build queries and extract results from the data. The code in [Listing 4-30](#) prints out the titles of all the tracks that were recorded by the artist with the name "Rob Miles." The first statement uses a LINQ query to create an enumerable collection of `MusicTrack` references called `selectedTracks` that is then enumerated by the `foreach` construction to print out the results.

LISTING 4-30 LINQ operators

[Click here to view code image](#)

```
IEnumerable<MusicTrack> selectedTracks = from track
foreach (MusicTrack track in selectedTracks)
{
    Console.WriteLine("Artist:{0} Title:{1}", track
}
```

The LINQ query returns an `IEnumerable` result that is enumerated by a `foreach` construction. You can find an explanation of `IEnumerable` in [Skill 2.4](#) in the "IEnumerable" section. The "Create method-based LINQ queries" section has more detail on how a query is actually implemented by C# code.

Use the var keyword with LINQ

The C# language is "statically typed." The type of objects in a program is determined at compile time and the compiler rejects any actions that are not valid. For example, the following code fails to compile because the compiler will not allow a string to be subtracted from a number.

```
string name = "Rob Miles";
int age = 21;
int silly = age - name;
```

This provides more confidence that our programs are correct before they run. The downside is that you have to put in the effort of giving each variable a type when you declare it. Most of the time, however, the compiler can infer the type to be used for any given variable. The `name` variable in the previous example must be of type `string`, since a `string` is being assigned to it. By the same logic, the `age` variable must be an `int`.

You can simplify code by using the `var` keyword to tell the compiler to infer the type of the variable being created from the context in which the variable is used. The compiler will define a string variable called `namev` in response to the following statement:

```
var namev = "Rob Miles";
```

Note, that this does not mean that the compiler cannot detect compilation errors. The statements in [Listing 4-31](#) still fails to compile:

LISTING 4-31 var errors

[Click here to view code image](#)

```
var namev = "Rob Miles";
var agev = 21;
var sillyv = agev - namev;
```



The `var` keyword is especially useful when using LINQ. The result of a simple LINQ query is an enumerable collection of the type of data element held in the data source. The statement next shows the query from Listing 4-30.

[Click here to view code image](#)

```
IEnumerable<MusicTrack> selectedTracks =  
    from track in musicTracks where track.Artist.Name == "Rob"
```

To write this statement you must find out the type of data in the `musicTracks` collection, and then use this type with `IEnumerable`. The `var` keyword makes this code much easier to write (see Listing 4-32).

LISTING 4-32 `var` and LINQ

[Click here to view code image](#)

```
var selectedTracks =  
    from track in musicTracks where track.Artist.Name == "Rob"
```

There are some situations where you won't know the type of a variable when writing the code. Later in this section you will discover objects that are created dynamically as the program runs and have no type at all. These are called anonymous types. The only way code can refer to these is by use of variables of type `var`.

You can use the `var` type everywhere in your code if you like, but please be careful. A statement such as the following will not make you very popular with fellow programmers because it is impossible for them to infer the type of variable `v` without digging into the code and finding out what type is returned by the `DoRead` method.

```
var v = DoRead();
```

If you really want to use `var` in these situations you should make sure that you select variable names that are suitably informative, or you can infer the type of the item from the code, as in the following statements.

```
var nextPerson = DoReadPerson();  
var newPerson = new Person();
```

LINQ projection

You can use the `select` operation in LINQ to produce a filtered version of a data source. In previous examples you discovered all of the tracks recorded by a particular artist. You can create other search criteria, for example by selecting the tracks with a certain title, or tracks longer than a certain length.

The result of a `select` is a collection of references to objects in the source data collection. There are a couple of reasons why a program might not want to work like this. First, you might not want to provide references to the actual data objects in the data source. Second, you might want the result of a query to contain a subset of the original data.

You can use *projection* to ask a query to "project" the data in the class onto new instances of a class created just to hold the data returned by the query. Let's start by creating the class called `TrackDetails` that will hold just the artist name and the title of a track. You will use this to hold the result of the search query.

```
class TrackDetails  
{  
    public string ArtistName;  
    public string Title;  
}
```

The query can now be asked to create a new instance of this class to hold the result of each query. Listing 4-33 shows how this works.

LISTING 4-33 LINQ projection

[Click here to view code image](#)

```
var selectedTracks = from track in musicTracks  
    where track.Artist.Name == "Rob"  
    select new TrackDetails  
    {  
        ArtistName = track.Artist.Name,  
        Title = track.Title  
    };
```

Projection results like this are particularly useful when you are using *data binding* to display the results to the user. Values in the query result can be bound to items to be displayed.

Anonymous types

You can remove the need to create a class to hold the result of a search



[Click here to view code image](#)

```
var selectedTracks = from track in musicTracks
                     where track.Artist.Name == '
                       select new // projection type
                       {
                           ArtistName = track.Artist.Name,
                           track.Title
                       };
```

The query in Listing 4-34 creates new instances of an anonymous type that contain just the data items needed from the query. Instances of the new type are initialized using the object initializer syntax. In this case the first property in the type is the name of the artist recording the track, and the second is the title of the track. For the first property you actually supply the name of the field to be created in the new type. For the second property the property is created with same name as the source property, in this case the property name will be `Title`.

The item that is returned by this query is an enumerable collection of instances of a type that has no name. It is an anonymous type. This means you *have* to use a `var` reference to refer to the query result. You can iterate through the collection in this result as you would any other. Note that each item in the `selectedTracks` collection must now be referred to using `var` because its type has no name. The code next shows how `var` is used for each item when printing out the results of the query in Listing 4-34.

[Click here to view code image](#)

```
foreach (var item in selectedTracks)
{
    Console.WriteLine("Artist:{0} Title:{1}", item.ArtistName, item.Title);
}
```

Note that the use of an anonymous type doesn't mean that the compiler is any less rigorous when checking the correctness of the code. If the program tries to use a property that is not present in the item, for example if it tries to obtain the `Length` property from the result of the query, this generates an error at compile time.

LINQ join

The class design used up to this point uses C# references to implement the associations between the objects in the system. In other words, a `MusicTrack` object contains a reference to the `Artist` object that represents the artist that recorded that track. If you store your data in a database however, you will not be able to store the associations in this way.

Instead, each item in the database will have a unique id (its *primary key*) and objects referring to this object will contain this ID value (a *foreign key*). Rather than a reference to an `Artist` instance, the `MusicTrack` now contains an `ArtistID` field that identifies the artist associated with that track. Figure 4-16 shows how this association is implemented.

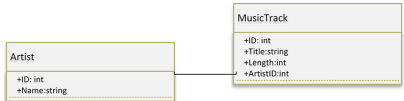


FIGURE 4-16 Music tracks and Artist ID

This design makes it slightly more difficult to search for tracks by a particular artist. The program needs to find the ID value for the artist with the name being searched for and then search for any tracks with that value of artist id. Fortunately, LINQ provides a `join` operator that can be used to join the output of one LINQ query to the input of another.

Listing 4-35 shows how this works. The first query selects the artist with the name "Rob Miles." The results of that query are joined to the second query that searches the `musicTracks` collection for tracks with an `ArtistID` property that matches that of the artist found by the first query.

LISTING 4-35 LINQ join

[Click here to view code image](#)

```
var artistTracks = from artist in artists where art.
                  join track in musicTracks on art.
                  select new
                  {
                      ArtistName = artist.Name,
                      track.Title
                  };
```

LINQ group

Another useful LINQ feature is the ability to group the results of a query to create a summary output. For example, you may want to create a query



Listing 4-36 shows how to do this. The group action is given the data item to group by and the property by which it is to be grouped. The `artistTrackSummary` contains an entry for each different artist. Each of the items in the summary has a `Key` property, which is the value that item is "grouped" around. You want to create a group around artists, so the key is the `ArtistID` value of each track. The `Key` property of the `artistTrackSummary` gives the value of this key. You can use behaviors provided by a summary object to find out about the contents of the summary, and the `Count` method returns the number of items in the summary. You will discover more summary commands in the discussion about the aggregate commands later in this section.

LISTING 4-36 LINQ group

[Click here to view code image](#)

```
var artistSummary = from track in musicTracks
                    group track by track.ArtistID
                    into artistTrackSummary
                    select new
                    {
                        ID = artistTrackSummary.Key,
                        Count = artistTrackSummary.Count;
                    };
```

You can print out the contents of the anonymous classes produced by this query by using a `foreach` loop as shown next.

[Click here to view code image](#)

```
foreach (var item in artistSummary)
{
    Console.WriteLine("Artist:{0} Tracks recorded:{0}",
        item.ID, item.Count);
}
```

The problem with this query is that when run it produces the results as shown next. Rather than generating the name of the artist, the program displays the `ArtistID` values.

[Click here to view code image](#)

```
Artist:0 Tracks recorded:5
Artist:6 Tracks recorded:5
Artist:12 Tracks recorded:5
Artist:18 Tracks recorded:5
```

You can fix this by making use of a join operation that will extract the artist name for use in the query. The needed joins shown next. You can then create the group keyed on the artist name rather than the ID to get the desired output.

[Click here to view code image](#)

```
var artistSummaryName = from track in musicTracks
                        join artist in artists on track.ArtistID equals artist.ID
                        group track by artist.Name
                        into artistTrackSummary
                        select new
                        {
                            ID = artistTrackSummary.Key,
                            Count = artistTrackSummary.Count;
                        };
```

The output from this query is shown here:

[Click here to view code image](#)

```
Artist:Rob Miles Tracks recorded:5
Artist:Fred Bloggs Tracks recorded:5
Artist:The Bloggs Singers Tracks recorded:5
Artist:Immy Brown Tracks recorded:5
```

Note that this is strong LINQ magic. It is worth playing with the sample code a little and examining the structure of the query to see what is going on.

LINQ Take and skip

A LINQ query will normally return all of the items that it finds. However, this might be more items that your program wants. For example, you might want to show the user the output one page at a time. You can use `take` to tell the query to take a particular number of items and the `skip` to tell a query to skip a particular number of items in the result before taking the requested number.

The sample program in Listing 4-37 displays all of the music tracks ten items at a time. It uses a loop that uses `skip` to move progressively further down the database each time the loop is repeated. The loop ends when the LINQ query returns an empty collection. The user presses a key at the end of each page to move onto the next page.



[Click here to view code image](#)

```
int pageNo = 0;
int pageSize = 10;

while(true)
{
    // Get the track information
    var trackList = from musicTrack in musicTracks.
                    join artist in artists on musicTrack
                    select new
                    {
                        ArtistName = artist.Name,
                        musicTrack.Title
                    };

    // Quit if we reached the end of the data
    if (trackList.Count() == 0)
        break;

    // Display the query result
    foreach (var item in trackList)
    {
        Console.WriteLine("Artist:{0} Title:{1}",
            item.ArtistName, item.Title);
    }

    Console.Write("Press any key to continue");
    Console.ReadKey();

    // move on to the next page
    pageNo++;
}
```

LINQ aggregate commands

In the context of LINQ commands, the word *aggregate* means “bring together a number of related values to create a single result.” You can use aggregate operators on the results produced by group operations. You have already used one aggregate operator in a LINQ query. You used the `Count` operator in [Listing 4-36](#) to count the number of tracks in a group extracted by artist. That provided the number of tracks assigned to a particular artist. You may want to get the total length of all the tracks assigned to an artist, and for that you can use the `Sum` aggregate operator.

The parameter to the `Sum` operator is a lambda expression that the operator will use on each item in the group to obtain the value to be added to the total sum for that item. To get the sum of `MusicTrack` lengths, the lambda expression just returns the value of the `Length` property for the item. [Listing 4-38](#) shows how this works.

LISTING 4-38 LINQ aggregate

[Click here to view code image](#)

```
var artistSummary = from track in musicTracks
                    join artist in artists on track.ArtistID
                    group track by artist.Name
                    into artistTrackSummary
                    select new
                    {
                        ID = artistTrackSummary.Key,
                        Length = artistTrackSummary.Sum(
                            t => t.Length
                        );
                    };
```

The result of this query is a collection of anonymous objects that contain the name of the artist and the total length of all the tracks recorded by that artist. The program produces the following output:

[Click here to view code image](#)

```
Name:Rob Miles   Total length:1406
Name:Fred Bloggs Total length:1533
Name:The Bloggs Singers Total length:1413
Name:Immy Brown Total length:1813
```

You can use `Average`, `Max`, and `Min` to generate other items of aggregate information. You can also create your own `Aggregate` behavior that will be called on each successive item in the group and will generate a single aggregate result.

Create method-based LINQ queries

The first LINQ query that you saw was in [Listing 4-30](#) as shown here.

[Click here to view code image](#)

```
IEnumerable<MusicTrack> selectedTracks =
    from track in musicTracks where track.Artist.N
```

The query statement uses *query comprehension syntax*, which includes the operators `from`, `in`, `where`, and `select`. The compiler uses these to



[Click here to view code image](#)

```
IEnumerable<MusicTrack> selectedTracks =  
    musicTracks.Where(track => track.Artist.Name == selectedArtistName);
```

The `Where` method accepts a lambda expression as a parameter. In this case the lambda expression accepts a `MusicTrack` as a parameter and returns `True` if the `Name` property of the `Artist` element in the `MusicTrack` matches the name that is being selected.

You first saw lambda expressions in [Skill 1.4, "Create and implement events and callbacks."](#) A lambda expression is a piece of behavior that can be regarded as an object. In this situation the `Where` method is receiving a piece of behavior that the method can use to determine which tracks to select. In this case the behavior is "take a track and see if the artist name is Rob Miles." You can create your own method-based queries instead of using the LINQ operators. [Listing 4-39](#) shows the LINQ query and the matching method-based behavior.

LISTING 4-39 Method based query

[Click here to view code image](#)

```
//IEnumerable<MusicTrack> selectedTracks = from tra  
    track.Artist.Name == "Rob Miles" select track  
// Method based implementation of this query  
IEnumerable<MusicTrack> selectedTracks = musicTracks.  
    track.Artist.Name == "Rob Miles");
```

Programs can use the LINQ query methods (and execute LINQ queries) on data collections, such as lists and arrays, and also on database connections. The methods that implement LINQ query behaviors are not added to the classes that use them. Instead they are implemented as *extension methods*. You can find out more about extension methods in [Skill 2.1](#), in the "Extension methods" section.

Query data by using query comprehension syntax

The phrase "query comprehension syntax" refers to the way that you can build LINQ queries for using the C# operators provided specifically for expressing data queries. The intention is to make the C# statements that strongly resemble the SQL queries that perform the same function. This makes it easier for developers familiar with SQL syntax to use LINQ.

[Listing 4-40](#) shows a complex LINQ query that is based on the LINQ query used in [Listing 4-38](#) to produce a summary giving the length of music by each artist. This uses the `orderby` operator to order the output by artist name.

LISTING 4-40 Complex query

[Click here to view code image](#)

```
var artistSummary = from track in musicTracks  
    join artist in artists on track.Artist equals artist  
    group track by artist.Name  
    into artistTrackSummary  
    select new  
    (  
        ID = artistTrackSummary.Key,  
        Length = artistTrackSummary.Sum(t => t.Length)  
    );
```

The SQL query that matches this LINQ is shown below:

[Click here to view code image](#)

```
SELECT SUM([t0].[Length]) AS [Length], [t1].[Name]  
FROM [MusicTrack] AS [t0]  
INNER JOIN [Artist] AS [t1] ON [t0].[ArtistID] = [t1].[ID]  
GROUP BY [t1].[Name]
```

This output was generated using the LINQPad application that allows programmers to create LINQ queries and view the SQL and method-based implementations. The standard edition is a very powerful resource for developers and can be downloaded for free from <http://www.linqpad.net/> (<http://www.linqpad.net/>).

Select data by using anonymous types

You first saw the use of anonymous types in the "Anonymous types" section earlier in this chapter. The last few sample programs have shown the use of anonymous types moving from creating values that summarize the contents of a source data object (for example extracting just the artist and title information from a `MusicTrack` value), to creating completely new types that contain data from the database and the results of aggregate operators.

It is important to note that you can also create anonymous type instances in method-based SQL queries. [Listing 4-41](#) shows the method-based implementation of the query from [Listing 4-40](#); anonymous type is shown



[Click here to view code image](#)

```
var artistSummary = MusicTracks.Join(
    Artists,
    track => track.ArtistID,
    artist => artist.ID,
    (track, artist) =>
        new
        {
            track = track,
            artist = artist
        }
)
.GroupBy(
    temp0 => temp0.artist.Name,
    temp0 => temp0.track
)
.Select(
    artistTrackSummary =>
        new
        {
            ID = artistTrackSum
            Length = artistTrac
        }
);
```

Force execution of a query

The result of a LINQ query is an item that can be iterated. We have used the `foreach` construction to display the results from queries. The actual evaluation of a LINQ query normally only takes place when a program starts to extract results from the query. This is called *deferred execution*. If you want to force the execution of a query you can use the `ToArray()` method as shown in Listing 4-42. The query is performed and the result returned as an array.

LISTING 4-42 Force query execution

[Click here to view code image](#)

```
var artistTracksQuery = from artist in artists
    where artist.Name == "Rob Miles"
    join track in musicTracks on ar
    select new
    {
        ArtistName = artist.Name,
        track.Title
    };

var artistTrackResult = artistTracksQuery.ToArray()

foreach (var item in artistTrackResult)
{
    Console.WriteLine("Artist:{0} Title:{1}",
        item.ArtistName, item.Title);
}
```

Note that in the case of this example the result will be an array of anonymous class instances. Figure 4-17 shows the view in Visual Studio of the contents of the result. The program has been paused just after the `artistTrackResult` variable has been set to the query result, and the debugger is showing the contents of the `artistTrackResult`.

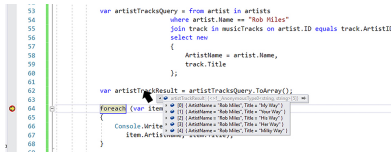


FIGURE 4-17 Immediate query results

A query result also provides `ToList` and `ToDictionary` methods that will force the execution of the query and generate an immediate result of that type. If a query returns a singleton value (for example the result of an aggregation operation such as `sum` or `count`) it will be immediately evaluated.

Read, filter, create, and modify data structures by using LINQ to XML

In this section we are going to investigate the LINQ to XML features that allow you to use LINQ constructions to parse XML documents. The classes that provide these behaviors are in the `System.Xml.Linq` namespace.

Sample XML document

The sample XML document is shown next. It contains two `MusicTrack` items that are held inside a `MusicTracks` element. The text of the sample document is stored in a string variable called `XMLText`.

[Click here to view code image](#)



```

string XMLText =
    "<MusicTracks> " +
        "<MusicTrack> " +
            "<Artist>Rob Miles</Artist> " +
            "<Title>My Way</Title> " +
            "<Length>150</Length>" +
        "</MusicTrack> " +

        "<MusicTrack>" +
            "<Artist>Immy Brown</Artist> " +
            "<Title>Her Way</Title> " +
            "<Length>200</Length>" +
        "</MusicTrack>" +
    "</MusicTracks>";

```

Read XML with LINQ to XML and XDocument

In the previous section, "Consume XML data" you learned how to consume XML data in a program using the `XDocument` class. This class has been superseded in later versions of .NET (version 3.5 onwards) by the `XDocument` class, which allows the use of LINQ queries to parse XML files.

A program can create an `XDocument` instance that represents the earlier document by using the `Parse` method provided by the `XDocument` class as shown here.

[Click here to view code image](#)

```
XDocument musicTracksDocument = XDocument.Parse(XMLText);
```

The format of LINQ queries is slightly different when working with XML. This is because the source of the query is a filtered set of XML entries from the source document. Listing 4-43 shows how this works. The query selects all the "MusicTrack" elements from the source document. The result of the query is an enumeration of `XElement` items that have been extracted from the document. The `XElement` class is a development of the `XMLElement` class that includes XML behaviors. The program uses a `foreach` construction to work through the collection of `XElement` results, extracting the required text values.

LISTING 4-43 Read XML with LINQ

[Click here to view code image](#)

```

IEnumerable<XElement> selectedTracks =
    from track in musicTracksDocument.Descendants("MusicTrack")
    foreach (XElement item in selectedTracks)
    {
        Console.WriteLine("Artist:{0} Title:{1}",
            item.Element("Artist").FirstNode,
            item.Element("Title").FirstNode);
    }

```

Filter XML data with LINQ to XML

The program in Listing 4-43 displays the entire contents of the XML document. A program can perform filtering in the query by adding a `where` operator, just as with the LINQ we have seen before. Listing 4-44 shows how this works. Note that the `where` operation has to extract the data value from the element so that it can perform the comparison.

LISTING 4-44 Filter XML with LINQ

[Click here to view code image](#)

```

IEnumerable<XElement> selectedTracks =
    from track in musicTracksDocument.Descendants("MusicTrack")
    where (string)track.Element("Artist") == "Rob Miles"
    select track;

```

The LINQ queries that we have seen so far have been expressed using query comprehension. It is possible, however, to express the same query in the form of a method-based query. The `Descendants` method returns an object that provides the `Where` behavior. The code next shows the query in Listing 4-44 implemented as a method-based query.

[Click here to view code image](#)

```

IEnumerable<XElement> selectedTracks =
    musicTracksDocument.Descendants("MusicTrack").Where(
        (string)element.Element("Artist") == "Rob Miles");

```

Create XML with LINQ to XML

The LINQ to XML features include very easy way to create XML documents. The code in Listing 4-45 creates a document exactly like the sample XML for this section. Note that the arrangement of the constructor calls to each `XElement` item mirror the structure of the document.



[Click here to view code image](#)

```
XElement MusicTracks = new XElement("MusicTracks",
    new List<XElement>
    {
        new XElement("MusicTrack",
            new XElement("Artist", "Rob Miles"),
            new XElement("Title", "My Way")),
        new XElement("MusicTrack",
            new XElement("Artist", "Immy Brown"),
            new XElement("Title", "Her Way"))
    }
);
```

Modify data with LINQ to XML

The `XElement` class provides methods that can be used to modify the contents of a given XML element. The program in [Listing 4-46](#) creates a query that identifies all the `MusicTrack` items that have the title "my Way" and then uses the `ReplaceWith` method on the title data in the element to change the title to the correct title, which is "My Way."

LISTING 4-46 Modify XML with LINQ

[Click here to view code image](#)

```
IEnumerable<XElement> selectedTracks =
    from track in musicTracksDocument.Descendants("MusicTrack")
    where (string) track.Element("Title") == "my Way"
    select track;

foreach (XElement item in selectedTracks)
{
    item.Element("Title").FirstNode.ReplaceWith("My Way");
}
```

As you saw when creating a new XML document, an `XElement` can contain a collection of other elements to build the tree structure of an XML document. You can programmatically add and remove elements to change the structure of the XML document.

Suppose that you decide to add a new data element to `MusicTrack`. You want to store the "style" of the music, whether it is "Pop", "Rock," or "Classical." The code in [Listing 4-47](#) finds all of the items in our sample data that are missing a `Style` element and then adds the element to the item.

LISTING 4-47 Add XML with LINQ

[Click here to view code image](#)

```
IEnumerable<XElement> selectedTracks =
    from track in musicTracksDocument.Descendants("MusicTrack")
    where track.Element("Style") == null
    select track;

foreach (XElement item in selectedTracks)
{
    item.Add(new XElement("Style", "Pop"));
}
```

SKILL 4.4: SERIALIZE AND DESERIALIZE DATA BY USING BINARY SERIALIZATION, CUSTOM SERIALIZATION, XML SERIALIZER, JSON SERIALIZER, AND DATA CONTRACT SERIALIZER

We have already explored the serialization process in [Skill 2.5](#), "The `Serializable` attribute," [Skill 3.1](#), "JSON and C#", [Skill 3.1](#), "JSON and XML," and [Skill 3.1](#), "Validate JSON data." You should read these sections before continuing with this section.

Serialization does not store any of the active elements in an object. The behaviors (methods) in a class are not stored when it is serialized. This means that the application deserializing the data must have implementations of the classes that can be used to manipulate the data after it has been read.

Serialization is a complex process. If a data structure contains a graph of objects that have a large number of associations between them, the serialization process will have to persist each of these associations in the stored file.

Serialization is best used for transporting data between applications. You can think of it as transferring the "value" of an object from one place to another. Serialization can be used for persisting data, and a serialized stream can be directed into a file, but this is not normally how applications store their state. Using serialization can lead to problems if the structure or behavior of the classes implementing the data storage changes during the lifetime of the application. In this situation developers may find that previously serialized data is not compatible with the new design.



We are going to use some sample music track data to illustrate how serialization works. The code shown next is the `MusicTrack`, `Artist`, and `MusicDataStore` objects that you are going to be working with. The `MusicDataStore` type holds lists of `MusicTrack` and `Artist` values. It also holds a method called `TestData` that creates a test music store that can be used in our examples.

[Click here to view code image](#)

```
class Artist
{
    public string Name { get; set; }
}

[Serializable]
class MusicTrack
{
    public Artist Artist { get; set; }
    public string Title { get; set; }
    public int Length { get; set; }
}

[Serializable]
class MusicDataStore
{
    List<Artist> Artists = new List<Artist>();
    List<MusicTrack> MusicTracks = new List<MusicTrack>();

    public static MusicDataStore TestData()
    {
        MusicDataStore result = new MusicDataStore();
        // create the same test data set as used for
        return result;
    }
}
```

Use binary serialization

There are essentially two kinds of serialization that a program can use: binary serialization and text serialization. In [Skill 4.1, "Convert text to binary data with Unicode,"](#) we noted that a file actually contains binary data (a sequence of 8-bit values). A UNICODE text file contains 8-bit values that represent text. Binary serialization imposes its own format on the data that is being serialized, mapping the data onto a stream of 8-bit values. The data in a stream created by a binary serializer can only be read by a corresponding binary de-serializer. Binary serialization can provide a complete "snapshot" of the source data. Both public and private data in an object will be serialized, and the type of each data item is preserved.

Classes to be serialized by the binary serializer must be marked with the `[Serializable]` attribute as shown below for the `Artist` class.

[Click here to view code image](#)

```
[Serializable]
class Artist
{
    public string Name { get; set; }
}
```

The binary serialization classes are held in the `System.Runtime.Serialization.Formatters.Binary` namespace. The code next shows how binary serialization is performed. It creates a test `MusicDataStore` object and then saves it to a binary file. An instance of the `BinaryFormatter` class provides a `Serialize` behavior that accepts an object and a stream as parameters. The `Serialize` behavior serializes the object to a stream.

[Click here to view code image](#)

```
MusicDataStore musicData = MusicDataStore.TestData();

BinaryFormatter formatter = new BinaryFormatter();
using (FileStream outputStream =
    new FileStream("MusicTracks.bin", FileMode.Open))
{
    formatter.Serialize(outputStream, musicData);
}
```

An instance of the `BinaryFormatter` class also provides a behavior called `Deserialize` that accepts a stream and returns an object that it has deserialized from that stream. [Listing 4-48](#) shows how to serialize an object into a binary file. The code uses a cast to convert the object returned by the `Deserialize` method into a `MusicDataStore`. This sample file for this listing also contains the previously shown `serialize` code.

LISTING 4-48 Binary serialization

[Click here to view code image](#)

```
MusicDataStore inputData;
```




```
        inputData = (MusicDataStore)formatter.Deseriali:
    }
}
```

If there are data elements in a class that should not be stored, they can be marked with the `NonSerialized` attribute as shown next. The `tempData` property will not be serialized.

[Click here to view code image](#)

```
[Serializable]
class Artist
{
    public string Name { get; set; }
    [NonSerialized]
    int tempData;
}
```

Binary serialization is the only serialization technique that serializes private data members by default (i.e. without the developer asking). A file created by a binary serializer can contain private data members from the object being serialized. Note, however, that once an object has serialized there is nothing to stop a devious programmer from working with serialized data, perhaps viewing and tampering with the values inside it. This means that a program should treat deserialized inputs with suspicion. Furthermore, any security sensitive information in a class should be explicitly marked `NonSerialized`. One way to improve security of a binary serialized file is to encrypt the stream before it is stored, and decrypt it before deserialization.

Use custom serialization

Sometimes it might be necessary for code in a class to get control during the serialization process. You might want to add checking information or encryption to data elements, or you might want to perform some custom compression of the data. There are two ways that to do this. The first way is to create our own implementation of the serialization process by making a data class implement the `ISerializable` interface.

A class that implements the `ISerializable` interface must contain a `GetObjectData` method. This method will be called when an object is serialized. It must take data out of the object and place it in the output stream. The class must also contain a constructor that will initialize an instance of the class from the serialized data source.

The code in [Listing 4-49](#) is an implementation of custom serialization for the `Artist` class in our example application. The `GetObjectData` method has two parameters. The `SerializationInfo` parameter `info` provides `AddValue` methods that can be used to store named items in the serialization stream. The `StreamingContext` parameter provides the serialization method with context about the serialization. The `GetObjectData` method for the `Artist` just stores the `Name` value in the `Artist` as a value that is called "name."

LISTING 4-49 Custom serialization

[Click here to view code image](#)

```
[Serializable]
class Artist : ISerializable
{
    public string Name { get; set; }

    protected Artist(SerializationInfo info, Stream
    {
        Name = info.GetString("name");
    }

    protected Artist ()
    {
    }

    [SecurityPermissionAttribute(SecurityAction.Dem
    public void GetObjectData(SerializationInfo info
    {
        info.AddValue("name", Name);
    }
}
```

The constructor for the `Artist` type accepts `info` and `context` parameters and uses the `GetString` method on the `info` parameter to obtain the name information from the serialization stream and use it to set the value of the `Name` property of the new instance.

The `GetObjectData` method must access private data in an object in order to store it. This can be used to read the contents of private data in serialized objects. For this reason, the `GetObjectData` method definition should be preceded by the security permission attribute you can see in [Listing 4-49](#) to control access to this method.

The second way of customizing the serialization process is to add methods that will be called during serialization. These are identified by attributes as shown in [Listing 4-50](#). The `OnSerializing` method is called before the serialization is performed and the `OnSerialized` method is called when the serialization is completed. The same format of attributes is used



the example program in Listing 4-50 you will see messages displayed as each stage of serialization is performed on the data.

LISTING 4-50 Customization methods

[Click here to view code image](#)

```
[Serializable]
class Artist
{
    [OnSerializing()]
    internal void OnSerializingMethod(StreamingContext context)
    {
        Console.WriteLine("Called before the artist is serialized.")
    }

    [OnSerialized()]
    internal void OnSerializedMethod(StreamingContext context)
    {
        Console.WriteLine("Called after the artist is serialized.")
    }

    [OnDeserializing()]
    internal void OnDeserializingMethod(StreamingContext context)
    {
        Console.WriteLine("Called before the artist is deserialized.")
    }

    [OnDeserialized()]
    internal void OnDeserializedMethod(StreamingContext context)
    {
        Console.WriteLine("Called after the artist is deserialized.")
    }
}
```

Manage versions with binary serialization

The `OnDeserializing` method can be used to set values of fields that might not be present in data that is being read from a serialized document. You can use this to manage versions. In Skill 4-3, in the "Modify data with LINQ to XML" section you added a new data element to `MusicTrack`. You added the "style" of the music, whether it is "Pop", "Rock," or "Classical." This causes a problem when the program tries to deserialize old `MusicTrack` data without this information.

You can address this by marking the new field with the `[OptionalField]` attribute and then setting a default value for this element in the `OnDeserializing` method as shown in Listing 4-51. The `OnDeserializing` method is performed during deserialization. The method is called before the data for the object is deserialized and can set default values for data fields. If the input stream contains a value for a field, this will overwrite the default set by `OnDeserializing`.

LISTING 4-51 Binary versions

[Click here to view code image](#)

```
[Serializable]
class MusicTrack
{
    public Artist Artist { get; set; }
    public string Title { get; set; }
    public int Length { get; set; }

    [OptionalField]
    public string Style;

    [OnDeserializing()]
    internal void OnDeserializingMethod(StreamingContext context)
    {
        Style = "unknown";
    }
}
```

Use XML serializer

You have already seen the XML serializer in use in Skill 3-1, in the "JSON and XML" section. A program can serialize data into an XML stream in much the same way as a binary formatter. Note, however, that when an `XmlSerializer` instance is created to perform the serialization, the constructor must be given the type of the data being stored. Listing 4-52 shows how this works.

LISTING 4-52 XML Serialization

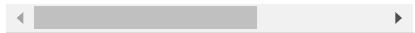
[Click here to view code image](#)

```
MusicDataStore musicData = MusicDataStore.TestData();

XmlSerializer formatter = new XmlSerializer(typeof(MusicTrack));
using (FileStream outputStream =
    new FileStream("MusicTracks.xml", FileMode.Open))
{
    formatter.Serialize(outputStream, musicData);
}
```



```
using (FileStream inputStream =
    new FileStream("MusicTracks.xml", FileMode.Open,
        FileAccess.Read))
{
    inputData = (MusicDataStore)formatter.Deserialize(inputStream);
}
```



XML serialization is called a *text* serializer, because the serialization process creates text documents.

The serialization process handles references to objects differently from binary serialization. Consider the class in [Listing 4-53](#). The `MusicTrack` type contains a reference to the `Artist` describing the artist that recorded the track.

LISTING 4-53 XML References

[Click here to view code image](#)

```
class MusicTrack
{
    public Artist Artist { get; set; }
    public string Title { get; set; }
    public int Length { get; set; }
}
```

If this track is serialized using binary serialization, the `Artist` reference is preserved, with a single `Artist` instance being referred to by all the tracks that were recorded by that artist. However, if this type of track is serialized using XML serialization, a copy of the `Artist` value is stored in each track. In other words, a `MusicTrack` value is represented as follows, with the contents of the artist information copied into the XML that is produced as shown.

[Click here to view code image](#)

```
<MusicTrack>
  <ID>1</ID>
  <Artist>
    <Name>Rob Miles</Name>
  <Title>My Way</Title>
  <Length>164</Length>
</MusicTrack>
```

When the XML data is deserialized each `MusicTrack` instance will contain a reference to its own `Artist` instance, which might not be what you expect. In other words, all of the data serialized using a text serializer is serialized by value. If you want to preserve references you must use binary serialization.

Note that the sample program in [Listing 4-51](#) uses an `ArtistID` value to connect a given `MusicTrack` with the artist that recorded it.

Use JSON Serializer

The JSON serializer uses the JavaScript Object Notation to store serialized data in a text file. Note that we have already discussed this serializer in detail in [Skill 3.1, "JSON and XML,"](#) and [Skill 3.1, "Validate JSON data."](#)

Use Data Contract Serializer

The data contract serializer is provided as part of the Windows Communication Framework (WCF). It is located in the `System.Runtime.Serialization` library. Note that this library is not included in a project by default. It can be used to serialize objects to XML files. It differs from the XML serializer in the following ways:

- Data to be serialized is selected using an "opt in" mechanism, so only items marked with the `[DataMember]` attribute will be serialized.
- It is possible to serialize private class elements (although of course they will be public in the XML text produced by the serializer).
- The XML serializer provides options that allow programmers to specify the order in which items are serialized into the data file. These options are not present in the `DataContract` serializer.

The classes here have been given the data contract attributes that are used to serialize the data in them.

[Click here to view code image](#)

```
[DataContract]
public class Artist
{
    [DataMember]
    public int ID { get; set; }
    [DataMember]
    public string Name { get; set; }
}

[DataContract]
public class MusicTrack
{
    [DataMember]
    public int ID { get; set; }
```



```

[DataMember]
public string Title { get; set; }
[DataMember]
public int Length { get; set; }
}

```

Once the fields to be serialized have been specified they can be serialized using a `DataContractSerializer`. [Listing 4-54](#) shows how this is done. Note that the methods to serialize and deserialize are called `WriteObject` and `ReadObject` respectively.

LISTING 4-54 Data contract serializer

[Click here to view code image](#)

```

MusicDataStore musicData = MusicDataStore.TestData(

DataContractSerializer formatter = new DataContractSeri

using (FileStream outputStream =
    new FileStream("MusicTracks.xml", FileMode.Open)
{
    formatter.WriteObject(outputStream, musicData);
}

MusicDataStore inputData;

using (FileStream inputStream =
    new FileStream("MusicTracks.xml", FileMode.Open)
{
    inputData = (MusicDataStore)formatter.ReadObject(
}

```

SKILL 4.5: STORE DATA IN AND RETRIEVE DATA FROM COLLECTIONS

C# programs can create variables that can store single data values, but there are many situations in which data values need to be grouped together. Objects provide one form of grouping. You can create types that describe a particular item, for example the `MusicTrack` type that has been used throughout the preceding examples.

Collections are a different way of grouping data. Collections are how to store a large number of objects that are usually all the same type, such as a list of `MusicTrack` instances. Unlike a database, which can be regarded as a service that provides data storage for an application, a collection is a structure that is stored in computer memory, alongside the program text and other variables.

You have used collections throughout the text so far. Now it is time to bring together your knowledge of how they work, the different collection types, and most importantly, how to select the right kind of collection for a particular job.

This Skill covers how to:

- Store and retrieve data by using dictionaries, arrays, lists, sets, and queues
- Choose a collection type
- Initialize a collection
- Add and remove items from a collection
- Use typed vs. non-typed collections
- Implement custom collections
- Implement collection interfaces

Store and retrieve data by using dictionaries, arrays, lists, sets, and queues

Before deciding what kind of data storage to use in your programs, you need an understanding of the collection types available to programs. Let's take a look at each in turn, starting with the simplest.

Use an array

An array is the simplest way to create a collection of items of a particular type. An array is assigned a size when it is created and the *elements* in the array are accessed using an *index* or *subscript* value. An array instance is a C# object that is managed by reference. A program creates an array by declaring the array variable and then making the variable refer to an array instance. Square brackets (`[` and `]`) are used to declare the array and also create the array instance. The statements next create an array variable called `intArray` that can hold arrays of integer values. The array variable `intArray` is then made to refer to a new array that contains five elements.

[Click here to view code image](#)

```

int [] intArray;
intArray = new int[5];

```



[Click here to view code image](#)

```
int [] intArray = new int[5];
```

An array of value types (for example an array of integers) holds the values themselves within the array, whereas for an array of reference types (for example an array of objects) each element in the array holds a reference to the object. When an array is created, each element in the array is initialized to the default value for that type. Numeric elements are initialized to 0, reference elements to null, and Boolean elements to false. Elements in an array can be updated by assigning a new value to the element.

Arrays implement the `IEnumerable` interface, so they can be enumerated using the `foreach` construction.

Once created, an array has a fixed length that cannot be changed, but an array reference can be made to refer to a new array object. An array can be initialized when it is created. An array provides a `Length` property that a program can use to determine the length of the array.

Listing 4-55 creates a new array, puts values into two elements and then uses a `for` loop to print out the contents of the array. It replaces the existing array with a new one, which is initialized to a four-digit sequence, and then prints out the contents of the new array using a `foreach` construction.

LISTING 4-55 Array example

[Click here to view code image](#)

```
// Array of integers
int[] intArray = new int[5];

intArray[0] = 99; // put 99 in the first element
intArray[4] = 100; // put 100 in the last element

// Use an index to work through the array
for (int i = 0; i < intArray.Length; i++)
    Console.WriteLine(intArray[i]);

// Use a foreach to work through the array
foreach (int intValue in intArray)
    Console.WriteLine(intValue);

// Initilaise a new array
intArray = new int [] { 1,2,3,4};

// Use a foreach to work through the array
foreach (int intValue in intArray)
    Console.WriteLine(intValue);
```

Any array that uses a single index to access the elements in the array is called a *one dimensional* array. It is analogous to a list of items. Arrays can have more than one dimension.

Multi-dimensional arrays

An array with two dimensions is analogous to a table of data that is made up of rows and columns. An array with three dimensions is analogous to a book containing a number of pages, with a table on each page. If you find yourself thinking that your program needs an array with more than three dimensions, you should think about arranging your data differently. The code next creates a two-dimensional array called `compass`, which holds the points of the compass. Elements in the array are accessed using a subscript value for each of the array dimensions. Note the use of the comma between the brackets in the declaration of the array variable. This denotes that the array has multiple dimensions.

[Click here to view code image](#)

```
string [,] compass = new string[3, 3]
{
    { "NW", "N", "NE" },
    { "W", "C", "E" },
    { "SW", "S", "SE" }
};

Console.WriteLine(compass[0, 0]); // prints NW
Console.WriteLine(compass[2, 2]); // prints SE
```

Jagged arrays

You can view a two-dimensional array as an array of one dimensional arrays. A "jagged array" is a two-dimensional array in which each of the rows are a different length. You can see how to initialize one here:

[Click here to view code image](#)

```
int[][] jaggedArray = new int[][]
{
    new int[] {1,2,3,4 },
    new int[] {5,6,7},
    new int[] {11,12}
}
```



The usefulness of an array is limited by the way you must decide in advance the number of items that are to be stored in the array. The size of an array cannot be changed once it has been created (although you can use a variable to set the dimension of the array if you wish). The `ArrayList` class was created to address this issue. An `ArrayList` stores data in a dynamic structure that grows as more items are added to it.

Listing 4-56 shows how it works. An `ArrayList` is created and three items are added to it. The items in an `ArrayList` can be accessed with a subscript in exactly the same way as elements in an array. The `ArrayList` provides a `Count` property that can be used to count how many items are present.

LISTING 4-56 `ArrayList` example

[Click here to view code image](#)

```
ArrayList arrayList = new ArrayList();

arrayList.Add(1);
arrayList.Add(2);
arrayList.Add(3);

for (int i = 0; i < arrayList.Count; i++)
    Console.WriteLine(arrayList[i]);
```

The `ArrayList` provides an `Add` method that adds items to the end of the list. There is also an `Insert` method that can be used to insert items in the list and a `Remove` method that removes items. There is more detail on the operations that can be performed on an `ArrayList` in the “[Add and remove items from a collection](#)” section.

Items in an `ArrayList` are managed by reference and the reference that is used is of type `object`. This means that an `ArrayList` can hold references to any type of object, since the `object` type is the base type of all of the types in C#. However, this can lead to some programming difficulties. This is discussed later in the “[Use typed vs. non-typed collections](#)” section.

Use a List

The `List` type makes use of the “generics” features of C#. You can find out more about generics in the “[Generic types](#)” section in [Skill 2.1](#). When a program creates a `List` the type of data that the list is to hold is specified using C# generic notation. Only references of the specified type can be added to the list, and values obtained from the list are of the specified type. **Listing 4-57** shows how a `List` is used. It creates a list of names, adds two names, and then prints out the list using a `for` loop. It then updates one of the names in the list and uses a `foreach` construction to print out the changed list.

LISTING 4-57 `List` example

[Click here to view code image](#)

```
List<string> names = new List<string>();

names.Add("Rob Miles");
names.Add("Immy Brown");

for (int i = 0; i < names.Count; i++)
    Console.WriteLine(names[i]);

names[0] = "Fred Bloggs";
foreach (string name in names)
    Console.WriteLine(name);
```

The `List` type implements the `ICollection` and `IList` interfaces. You can find out more about these interfaces later in the “[Implement collection interfaces](#)” section.

Use a dictionary

A `Dictionary` allows you to access data using a key. The name `Dictionary` is very appropriate. If you want to look up a definition of a word, you find the word in a dictionary and read the definition. In the case of an application, the key might be an account number or a username. The data can be a bank account or a user record.

Listing 4-58 shows how a `Dictionary` is used to implement bank account management. Each `Account` object has a unique `AccountNo` property that can be used to locate the account. The program creates two accounts and then uses the `AccountNo` value to find them. You can think of the key as being a *subscript* that identifies the item in the dictionary. The key value is enclosed in square brackets like for an array or a list.

LISTING 4-58 `Dictionary` example

[Click here to view code image](#)

```
BankAccount a1 = new BankAccount { AccountNo = 1, N
BankAccount a2 = new BankAccount { AccountNo = 2, N

Dictionary<int, BankAccount> bank = new Dictionary<

bank.Add(a1.AccountNo, a1);
bank.Add(a2.AccountNo, a2);
```



```
if (bank.ContainsKey(2))
    Console.WriteLine("Account located");
```

A dictionary can be used to implement data storage, but it is also useful in very many other contexts. [Listing 4-59](#) shows how to use a dictionary to count the frequency of words in a document. In this case the dictionary is indexed on a word and contains an integer that holds the count of that word. The document is loaded from a file into a string. The program extracts each word from the string. If the word is present in the dictionary the count for that word is incremented. If the word is not present in the dictionary, an entry is created for that word with a count of 1.

LISTING 4-59 Word counter

[Click here to view code image](#)

```
Dictionary<string, int> counters = new Dictionary<string, int>();

string text = File.ReadAllText("input.txt");

string[] words = text.Split(new char[] { ' ', '.', ',',
    StringSplitOptions.RemoveEmptyEntries });

foreach (string word in words)
{
    string lowWord = word.ToLower();
    if (counters.ContainsKey(lowWord))
        counters[lowWord]++;
    else
        counters.Add(lowWord, 1);
}
```

Here, you would like the word counter to produce a sorted list of word counts. List and array instances provide a `Sort` method that can be used to sort their contents. Unfortunately, the `Dictionary` class does not provide a sort behavior. However, you can use a LINQ query on a dictionary to produce a sorted iteration of the dictionary contents. This can be used by a `foreach` loop to generate sorted output. The code to do this is shown next. It requires careful study. Items in a `Dictionary` have `Key` and `Value` properties that are used for sorting and output. When trying the code on an early version of this text I found that the word “the” was used around twice as many times as the next most popular word, which was “a.”

[Click here to view code image](#)

```
var items = from pair in counters
            orderby pair.Value descending
            select pair;

foreach (var pair in items)
{
    Console.WriteLine("{0}: {1}", pair.Key, pair.Value);
}
```

Use a set

A set is an unordered collection of items. Each of the items in a set will be present only once. You can use a set to contain tags or attributes that might be applied to a data item. For example, information about a `MusicTrack` can include a set of style attributes. A track can be “Electronic,” “Disco,” and “Fast.” Another track can be “Orchestral,” “Classical,” and “Fast.” A given track is associated with a set that contains all of the attributes that apply to it. A music application can use set operations to select all of the music that meets particular criteria, for example you can find tracks that are both “Electronic” and “Disco.”

Some programming languages, such as Python and Java, provide a set type that is part of the language. C# doesn’t have a built-in set type, but the `HashSet` class can be used to implement sets. [Listing 4-60](#) shows how to create three sets that contain strings that contain the names of style attributes that can be applied to music tracks. The first two, `t1Styles` and `t2Styles`, give style information for two tracks. The third set is the `search` set that contains two style elements that you might want to search for in tracks. The `HashSet` class provides methods that implement set operations. The `IsSubsetOf` method returns true if the given set is a subset of another. The program uses this method to determine which of the two tracks matches the search criteria.

LISTING 4-60 Set example

[Click here to view code image](#)

```
HashSet<string> t1Styles = new HashSet<string>();
t1Styles.Add("Electronic");
t1Styles.Add("Disco");
t1Styles.Add("Fast");

HashSet<string> t2Styles = new HashSet<string>();
t2Styles.Add("Orchestral");
t2Styles.Add("Classical");
t2Styles.Add("Fast");
```



```
search.Add("Disco");

if (search.IsSubsetOf(t1Styles))
    Console.WriteLine("All search styles present in t1");

if (search.IsSubsetOf(t2Styles))
    Console.WriteLine("All search styles present in t2");
```

Another set methods can be used to combine set values to produce unions, differences, and to test supersets and subsets.

Use a queue

A queue provides a short term storage for data items. It is organized as a *first-in-first-out* (FIFO) collection. Items can be added to the queue using the `Enqueue` method and read from the queue using the `Dequeue` method. There is also a `Peek` method that allows a program to look at an item at the top of the queue without removing it from the queue. A program can iterate through the items in a queue and a queue also provides a `Count` property that will give the number of items in the queue. [Listing 4-61](#) shows the usage of a simple queue that contains strings.

LISTING 4-61 Queue example

[Click here to view code image](#)

```
Queue<string> demoQueue = new Queue<string>();

demoQueue.Enqueue("Rob Miles");
demoQueue.Enqueue("Immy Brown");

Console.WriteLine(demoQueue.Dequeue());
Console.WriteLine(demoQueue.Dequeue());
```

The program will print "Rob Miles" first when it runs, because of the FIFO behavior of a queue. One potential use of a queue is for passing work items from one thread to another. If you are going to do this you should take a look at the `ConcurrentQueue`, which is described in [Skill 1.1](#).

Use a stack

A stack is very similar in use to a queue. The most important difference is that a stack is organized as *last-in-first-out* (LIFO). A program can use the `Push` method to push items onto the top of the stack and the `Pop` method to remove items from the stack. [Listing 4-62](#) shows simple use of a stack. Note that the program prints out "Immy Brown" first, because that is the item on the top of the stack when the printing is first performed. There is a `ConcurrentStack` implementation that should be used if different Tasks are using the same stack.

LISTING 4-62 Stack example

[Click here to view code image](#)

```
Stack<string> demoStack = new Stack<string>();

demoStack.Push("Rob Miles");
demoStack.Push("Immy Brown");

Console.WriteLine(demoStack.Pop());
Console.WriteLine(demoStack.Pop());
```

Choose a collection type

The type of collection to use normally falls naturally from the demands of the application. If you need to store a list of values, use a `List` in preference to an array or `ArrayList`. An array is fixed in size and an `ArrayList` does not provide type safety. A `List` can only contain objects of the list type and can grow and contract. It is also very easy to remove a value from the middle of a list or insert an extra value.

Use an array if you are concerned about performance and you are holding value types, since the data will be accessed more quickly. The other occasion where arrays are useful is if a program needs to store two-dimensional data (for example a table of values made up of rows and columns). In this situation you can create an object that implements a row (and contains a `List` of elements in the row) and then stores a `List` of these objects.

If there is an obvious value in an object upon which it can be indexed (for example an account number or username), use a dictionary to store the objects and then index on that value. A dictionary is less useful if you need to locate a data value based on different elements, such as needing to find a customer based on their customer ID, name, or address. In that case, put the data in a `List` and then use LINQ queries on the list to locate items.

Sets can be useful when working with tags. Their built-in operations are much easier to use than writing your own code to match elements together. Queues and stacks are used when the needs of the application require FIFO or LIFO behavior.

Initialize a collection

The examples that you have seen have added values to collections by calling the collection methods to add the values. For example, use the `Add` method to add items to a `List`. However, there are other ways to



LISTING 4-63 Collection initialization

[Click here to view code image](#)

```
int[] arrayInit = { 1, 2, 3, 4 };

ArrayList arrayListInit = new ArrayList { 1, "Rob" };

List<int> listInit = new List<int> { 1, 2, 3, 4 };

Dictionary<int, string> dictionaryInit = new Dictionary<int, string>
{
    { 1, "Rob" },
    { 2, "Immy" } };

HashSet<string> setInit = new HashSet<string> { "Elmer", "Fudd" };

Queue<string> queueInit = new Queue<string> { new string { "1" }, new string { "2" } };

Stack<string> stackInit = new Stack<string> (new string { "1" }, new string { "2" } );
```

Add and remove items from a collection

Some of the collection types that we have discussed contain support for adding and removing elements from the types. Here are the methods for each collection type.

Add and remove items from an array

The array class does not provide any methods that can add or remove elements. The size of an array is fixed when the array is created. The only way to modify the size of an existing array is to create a new array of the required type and then copy the elements from one to the other. The array class provides a `CopyTo` method that will copy the contents of an array into another array. The first parameter of `CopyTo` is the destination array. The second parameter is the start position in the destination array for the copied values. Listing 4-64 shows how this can be used to migrate an array into a larger one. The new array has one extra element, but you can make it much larger than this if required. Note that because arrays are objects managed by reference, you can make the `dataArray` reference refer to the newly created array.

LISTING 4-64 Grow an array

[Click here to view code image](#)

```
int[] dataArray= { 1, 2, 3, 4 };
int[] tempArray = new int[5];
dataArray.CopyTo(tempArray, 0);
dataArray= tempArray;
```

Add and remove items in ArrayList and List

There are a number of methods that can be used to modify the contents of the `ArrayList` and `List` collections. Listing 4-65 shows them in action.

LISTING 4-65 List modification

[Click here to view code image](#)

```
List<string> list = new List<string>();
list.Add("add to end of list"); // add to the end
list.Insert(0, "insert at start"); // insert an item at start
list.Insert(1, "insert new item 1"); // insert at position 1
list.InsertRange(2, new string[] { "Rob", "Immy" });
list.Remove("Rob"); // remove first occurrence of "Rob"
list.RemoveAt(0); // remove element at index 0
list.RemoveRange(1, 2); // remove two elements starting at index 1
list.Clear(); // clear entire list
```

Add and remove items from a Dictionary

The Dictionary type provides Add and Remove methods, as shown in Listing 4-66.

LISTING 4-66 Dictionary modification

[Click here to view code image](#)

```
Dictionary<int, string> dictionary = new Dictionary<int, string>();
dictionary.Add(1, "Rob Miles"); // add an entry
dictionary.Remove(1); // remove the entry
```

Add and remove items from a Set

The Set type provides Add, Remove and RemoveWhere methods. Listing 4-67 shows how they are used. The RemoveWhere function is given a *predicate* (a behavior that generates either true or false) to determine which elements are to be removed. In the listing the predicate is a lambda expression that evaluates to true if the element in the set starts with the character 'R'.

LISTING 4-67 Set modification

[Click here to view code image](#)



```
set.Remove("Rob Miles"); // remove an item
set.RemoveWhere(x => x.StartsWith("R")); // remove
```

Add and remove items in Queue and Stack

The most important aspect of the behavior of queues and stacks is the way that items are added and removed. For this reason, the only actions that allow their contents to be changed are the ones you have seen earlier.

Use typed vs. non-typed collections

When comparing the behavior of the `ArrayList` collection types we noted that there is nothing to stop a programmer putting any type of object in the same `ArrayList`. The code next adds an integer, a string, and even an `ArrayList` to the `ArrayList` called `MessyList`. While this may be something you want to do, it is not good programming practice.

[Click here to view code image](#)

```
ArrayList messyList = new ArrayList();
messyList.Add(1); // add an integer to the list
messyList.Add("Rob Miles"); // add a string to the
messyList.Add(new ArrayList()); //add an ArrayList
```

Another difficulty caused by the untyped storage provided by the `ArrayList` is that all of the references in the list are references to objects. When a program removes an item from an `ArrayList` it must cast the item into its proper type before it can be used. In other words, to use the `int` value at the subscript 0 of the `messyList` above, cast it to an `int` before using it, as shown here:

[Click here to view code image](#)

```
int messyInt = (int) messyList[0];
```

Note that if you are confused to see the value type `int` being used in an `ArrayList`, where the contents are managed by reference, you should read the “Boxing and unboxing” section in [Skill 2.2](#).

These problems occur due to an `ArrayList` existing in an *untyped* collection. For this reason, the `ArrayList` has been superseded by the `List` type, which uses the generics features in later versions of C#. To allow a programmer to specify the type of item that the list should hold. It is recommended that you use the `List` type when you want to store collections of items.

Implement custom collections

A custom collection is a collection that you create for a specific purpose that has behaviors that you need in your application. One way to make a custom collection is to create a new type that implements the `ICollection` interface. This can then be used in the same way as any other collection, such as with a program iterating through your collection using a `foreach` construction. We will describe how to implement a collection interface in the next section.

Another way to create a custom collection is to use an existing collection class as the base (parent) class of a new collection type. You can then add new behaviors to your new collection and, because it is based on an existing collection type, your collection can be used in the same way as any other collection.

Listing 4-68 shows how to create a custom `MusicTrack` store, which is based on the `List` collection type. The method `RemoveArtist` has been added to the new type so that a program can easily remove all the tracks by a particular artist. Note how the `RemoveArtist` method creates a list of items to be removed and then removes them. This is to prevent problems that can be caused by removing items in a collection at the same time as iterating through the collection. If you investigate the sample program for **Listing 4-68**, you will find that the `TrackStore` class also contains a `ToString` method and also a static `GetTestTrackStore` method that can be used to create a store full of sample tracks.

LISTING 4-68 Custom collection

[Click here to view code image](#)

```
class TrackStore : List<MusicTrack>
{
    public int RemoveArtist(string removeName)
    {
        List<MusicTrack> removeList = new List<MusicTrack>();
        foreach (MusicTrack track in this)
            if (track.Artist == removeName)
                removeList.Add(track);

        foreach (MusicTrack track in removeList)
            this.Remove(track);

        return removeList.Count;
    }
}
```



The behavior of a collection type is expressed by the `ICollection` interface. The `ICollection` interface is a child of the `IEnumerator` interface. Interface hierarchies work in exactly the same way as class hierarchies, in that a child of a parent interface contains all of the methods that are described in the parent. This means that a type that implements the `ICollection` interface is capable of being enumerated. For a more details on the `IEnumerator` interface, consult the “[IEnumerable](#)” section in [Skill 2.4](#).

The class in [Listing 4-69](#) implements the methods in the `ICollection` interface. It contains an array of fixed values that give four points of a compass. It can be used in the same way as any other collection, and can be enumerated as it provides a `GetEnumerator` method. Note that the collection interface does not specify any methods that determine how (or indeed whether) a program can add and remove values.

LISTING 4-69 `ICollection` interface

[Click here to view code image](#)

```
class CompassCollection : ICollection
{
    // Array containing values in this collection
    string[] compassPoints = { "North", "South", "E", "W" };

    // Count property to return the length of the collection
    public int Count
    {
        get { return compassPoints.Length; }
    }

    // Returns an object that can be used to synchronize
    // access to this object
    public object SyncRoot
    {
        get { return this; }
    }

    // Returns true if the collection is thread safe
    // This collection is not
    public bool IsSynchronized
    {
        get { return false; }
    }

    // Provide a copyto behavior
    public void CopyTo(Array array, int index)
    {
        foreach (string point in compassPoints)
        {
            array.SetValue(point, index);
            index = index + 1;
        }
    }

    // Required for IEnumerable
    // Returns the enumerator from the embedded array
    public IEnumerator GetEnumerator()
    {
        return compassPoints.GetEnumerator();
    }
}
```

Note, that if you want the new collection type to be used with LINQ queries it must implement the `IEnumerable<type>` interface. This means that the type must contain a `GetEnumerator<string>()` method.

THOUGHT EXPERIMENTS

In these thought experiments, demonstrate your skills and knowledge of the topics covered in this chapter. You can find the answers to these thought experiments in the next section.

1 Perform I/O operations

Class hierarchies are an important element in the design of the data management mechanisms provided by .NET. The `Stream` class sets out the fundamental data movement commands and is used as the basis of classes that provide data storage on a variety of platforms. Streams can also be chained together so that character encoding, encryption and compression can be performed on data being moved to or from any stream-based store.

Here are some questions to consider:

1. Why can't a program create an instance of the `Stream` type?
2. When would a program modify the file pointer in a file?
3. What is the difference between UTF8 Unicode and UTF32 Unicode?
4. Is there such a thing as a "text file"?
5. Can a program both read from and write to a single file?
6. What is the difference between a `TextWriter` and a `StreamWriter`?



8. What is the `File` class for?
9. Will you need to re-compile a program for it to be used on a system with a different file system?
10. What happens to the files in a directory when you delete the directory?
11. How can you tell if a file path is an absolute path to the file?
12. Why should you use the `Path` class to construct file paths in your programs?
13. What is the difference between HTTP and HTML?
14. When should you use the `HttpWebRequest` class to contact a web server?
15. Can a program make web requests asynchronously?
16. Why is it a good idea to perform file operations asynchronously?

2 Consume data

Databases are the storage resource that underpins many applications. In this section you've seen how a program can use Structured Query Language (SQL) to interact with a database. You've also discovered more about the use of databases in Active Server Pages (ASP) and considered how JavaScript Object Notation (JSON) and eXtensible Markup Language (XML) documents allow programs to exchange structured data. Finally, you've taken a look at web services; a means by which a server can expose resources in the form of method calls.

Here are some questions to consider:

1. Does each user of a database need their own copy of the data?
2. What would happen if two users of a database updated the same record at the same time?
3. What would happen if two items in a database had the same ID value?
4. Why do you need to protect your database connection string?
5. Do you always have to write SQL queries to interact with a database?
6. What is an SQL injection attack, and how do you protect against one?
7. You can see how to read JSON data from a server, but is it possible to store JSON formatted data values on a server?
8. What is the difference between a web service and just downloading JSON document from a server?

3 Query and manipulate data and objects by using LINQ

LINQ allows developers to take their SQL skills in building queries and apply them to software objects. By exposing query elements as C# operators using the query comprehension syntax LINQ allows you to create C# statements that closely resemble SQL queries. These queries can be applied to structured data from a database, but they can also be applied to normal collections and XML documents. LINQ is useful for filtering data and extracting subsets; it also provides the group operator that can be used to summarize data. LINQ queries are actually implemented as method calls onto objects and can be created that way if preferred.

Here are some questions to consider:

1. What does "Language INtegrated Query" actually mean?
2. Does LINQ add new features for data manipulation?
3. Is it more efficient to build and perform our own SQL queries than to use LINQ?
4. Why would the statement `var counter;` cause a compilation error?
5. What is an anonymous type?
6. What does "deferred execution" of a LINQ query mean?
7. What does the group behavior do in a LINQ query?
8. What do the `take` and `skip` behaviors do?
9. What is the difference between query comprehension and method-based LINQ queries?
10. What is the difference between the `XDocument` and `XElement` types?
11. What is the difference between the `XmlDocument` and `XDocument` types?

4 Serialize and deserialize data

Serialization is very useful when you want to store or transfer structured data. It is susceptible to problems if the content or arrangement of data items that have been serialized change in later versions of an application. But, as you have seen, it is possible for these to be addressed with sensible data design.



2. Do you need a copy of the type to read a serialized object?
3. Can any data type be serialized?
4. Can value types and reference types be stored using serialization?
5. Does serialization store all of the elements of a class?
6. When should you use binary serialization and when should you use XML serialization?
7. Why should you be concerned about security when using binary serialization?
8. Can you use a custom serializer when serializing to an XML document?
9. Is it possible to encrypt a serialized class?
10. What is the difference between an XML serializer and a `DataContract` serializer?

5 Store data in and retrieve data from collections

You can think of the collection classes provided by .NET as a set of tools. Each tool is suited for a particular situation. When considering how to store data in a program it is important to consider the whole range of different kinds of collections. Most of the time storage demands tend to be met by the `List` or `Dictionary` types, but there have been occasions where you need the FIFO behavior provided by a queue, and a set can save a lot of work because of the behaviors that it provides. There are two things that are important when dealing with collections. The first is that extending a parent collection type is a great way to add custom behaviors. The second is that you should remember that it is possible to perform LINQ queries on collections, which can save you a lot of work writing code to search through them and select items.

Here are some questions to consider:

1. When should you use a collection, and when should you use a database?
2. Can you create an array of arrays?
3. Can you create a twenty-dimensional array?
4. Why does the program keep crashing with an array index error?
5. The array is one element too small. How do you add a new element on the end of the array?
6. Why do you use `Length` to get the length of an array and `Count` to get the number of items in an `ArrayList`?
7. How does a `Dictionary` actually work?
8. When would you use a set?
9. What is the difference between a stack and a queue?
10. Could you use a `List` to store every type of data in my program?

THOUGHT EXPERIMENT ANSWERS

This section provides the solutions for the tasks included in the thought experiments.

1 Perform I/O operations

1. It is not possible for a program to create an instance of the `Stream` type, because the `Stream` class is defined as abstract and intended to be used as the base class for child classes that contain implementations of the behaviors described by `Stream`.
2. The file pointer value is managed by a stream and specifies the position at which the next input/output transaction will be performed. When reading a file, the file pointer starts at the beginning of the file. If a program wishes to "skip" some locations in the file the file pointer can be updated to the new location. It is faster to update the file pointer than it is to move down a file by reading from it. When writing into a file the file pointer is updated as the file is written. When a file is opened in the "append" mode the file pointer is moved to the end of the file after the file has been opened.
3. Unicode is a mapping between values and text characters. As an example, the character π has the Unicode value 120587. The UTF8 standard maps Unicode characters onto one or more 8-bit storage locations. The UTF32 standard maps Unicode characters onto one or more 32-bit storage locations.
4. The file system on a computer does not make a distinction between a "text" file and a "binary" file. This distinction is made by the programs using the file system to store files. In the case of C#, a text file is one that contains values that represent text. The values in the text file will be encoded using a standard that maps numeric values onto characters. The Unicode standard is frequently used for this.
5. If a stream is opened for `ReadWrite` access, the program can both read from and write into the file. The file will usually hold fixed length records, so that a single record in the middle of a file can be changed without affecting any of the other records in the file.
6. The `TextWriter` class is an abstract class that specifies operations that can be performed to write text into a stream. The



7. A program can construct a stream from a stream to allow two streams to be "chained" together. The output of one stream can then be sent to the input of the next. The example used was that we could use a compression stream in conjunction with a file stream to create a stream that would compress data as it was written into a file.
8. The `File` class provides a set of very useful methods that can be used to create and populate files with only a small number of statements. It also provides a number of very useful file management commands.
9. The precise way in which a given file system works is completely hidden from the programs that are using it. There is need to recompile a program to change to a different file system.
10. If a program tries to delete a directory that is not empty, the delete action will fail with an exception. The program must delete all the files in a directory before the directory itself can be removed.
11. An absolute file path will start with the drive letter on which the file is stored and contain all the directories to be traversed to reach the file. A relative path will not start with a drive letter.
12. Programs frequently have to create filenames by adding drives, directory names, and filenames together. The `Path` class provides a static `Combine` method that makes sure that all the path separators are correctly inserted into the path that is created. This is much less prone to error than trying to make sure that the correct number of "backslash" characters has been included in the name.
13. HTTP stands for "Hyper Text Transfer Protocol." It defines the way that a client can give instructions to a web server. HTML stands for "Hyper Text Markup Language." It defines the format of documents that are sent back from the server in response to an HTTP request.
14. The `HttpRequest` class is extremely customizable and provides much more flexibility than the `WebClient` class when creating requests to be sent to web servers. Some forms of web request can only be sent using an `HttpRequest`.
15. The `WebClient` and `HttpClient` classes support the use of `async` and `await` for making asynchronous web requests.
16. It is a good idea to perform file operations asynchronously because they are frequently the slowest actions performed on a computer. Writing to a physical disk takes much longer than calculations, and disk transactions may take much longer if the system is busy.

2 Consume data

1. The idea behind a database server is that it can provide access to a central store of data. It is a bad idea for each user to have their own copy of the data because the copies can become updated in different ways. However, it is possible for a system to keep a local copy of a database for use when the system is not connected to a network and then apply the updates to a database when network connectivity is available.
2. A database server is designed to ensure that the stored data is always consistent. If two users update a particular record at the same time the updates will be performed in sequence and the server will make sure that data is not corrupted. It is possible for a user of a database to flag actions as *atomic* so that they are either completed or the database is "rolled back" to the state it had before the action.
3. If the ID of an item is being used as a "key" to uniquely identify the item, it is impossible to create two items with the same ID. The database will generate ID values for items automatically when they are added to the database.
4. The database connection string is used to create a connection to the database. If the database is not on the same computer as the program using it, the connection string will include the network address of the server and authentication information. Armed with this information anyone can create a connection to the database and send it SQL commands, which would be a bad thing.
5. SQL queries are the lowest level of communication with the database. However, you have seen that in an ASP.NET application you can perform actions on objects in the program and then update the contents of the database with the new objects.
6. An SQL command is a construction that contains command and data elements. An SQL injection attack is performed by "injecting" SQL commands into the data parts of the command. For example, if the user of a database is asked to enter a new name for the customer, they can add malicious SQL commands to the name text. If this text is used to build the command sent to the database, these commands are obeyed by the database. These attacks can be prevented by using parameterized SQL commands, which specifically separate the data from the command elements.
7. If your application needs to be able to upload data to a server application it can use the Representational State Transfer (REST) model. This makes use of the HTTP command set to allow an application to send data between the client and the server. The data can be XML or JSON documents.
8. A web service provides the client application with a "proxy object" that represents a connection to the service. The client can call methods on the proxy to interact with the service. This provides a



3 Query and manipulate data and objects by using LINQ

1. The phrase "Language Integrated Query" refers to the way that a data query can be expressed using language elements in a format called "query comprehension". The LINQ operators (`from`, `show`, `join`, etc.) allow the programmer to express their intent directly, without having to make a chain of method calls to create a query. That said, we know that the compiler actually converts "query comprehension" notation queries into a chain of method calls, and programmers can create "method-based" queries if they prefer.
2. LINQ doesn't create any new data manipulation features, but it does make it much, much easier for a programmer to express what they want to do.
3. Building your own SQL queries is slightly more efficient than using LINQ. The very first time a LINQ query is used it will be compiled into the method sequence that is then called to deliver the result. This is rarely a problem from a performance point of view, however, and LINQ makes programmers more efficient.
4. When you declare a variable of type `var` you are asking the compiler to infer the type of the variable from the value being assigned to it. If the declaration does not assign an initial value to the variable (as in this case) the compiler will refuse to compile the statement because it has no value from which it can infer the type of the variable.
5. An anonymous type is a type that has no name. This is not a very helpful thing to say, but it is true. Normally a type is defined and then the new keyword is used to make new instances of that type. In the case of an anonymous type the instance is created without the associated type. The object initializer syntax (which allows a programmer to initialize public properties of a class when an instance is created) also allows objects to be created without an associated type. In the case of LINQ these objects can be returned as the results of queries. The objects contain custom result types that exactly match the data requested. Note that these objects must be referred to using the `var` keyword.
6. A LINQ query will return a result that describes an iteration that can then be processed using a `foreach` construction. If the query result contains many result values, it takes a long time (and uses up a lot of memory) to actually generate that result when the query is performed. So instead, the iteration is evaluated and each result is generated in turn when it is to be consumed.
7. The group operator creates a new dataset grouped around the value of one item in the input dataset. You used it in the `MusicTracks` application to take a list of all the tracks (each track containing an `ArtistID` value) and create a group that contains one entry for each `ArtistID` value in the dataset. You can then use aggregate operations on the group to do things such as count the number of tracks and sum the total length of the tracks for each artist.
8. The `take` behavior creates a LINQ query that takes a particular number of items from the source dataset. The `skip` behavior skips down the dataset a given number of items before the query starts taking values. Used in combination they allow a program to work through a dataset one "page" at a time.
9. The "query comprehension" and "method-based" LINQ query formats can be used interchangeably in programs. You can use them both in a single solution, depending on which is most convenient at any given point in the program.
10. The `XDocument` class can hold an XML document, including the elements that can be used to express a "fully formed" XML document, including metadata about the document contents. An `XElement` can contain an element in an XML document (which can contain a tree of other `XElement` objects). An `XDocument` contains `XElement` objects that contain the data in the document.
11. The `XmlDocument` implements a Document Object Model (DOM) for XML documents. The `XDocument` builds on the ability of an `XmlDocument` to allow it to work with LINQ queries. The same relationship applies between `XmlElement` and `XElement`.

4 Serialize and deserialize data

1. Serialization is very useful if you are storing small amounts of structured data. The high score table for a game can be stored as a serialized object. Another use for a small serialized object can be the settings for an application. It would be less sensible to store a large data structure as a serialized object, particularly if the object has to be repeatedly updated.
2. Serialization takes a snapshot of the data elements in a particular type. During the deserialization process a new instance of the serialized type is created. This process requires the type to be deserialized to be available on the receiving machine. In the case of binary serialization, the binary file contains type information that is compared with type information in the destination class. If this information doesn't match, the deserialization fails. Note that in the case of serialization to XML and JSON text files, this matching does not take place. You can regard these two types as being more portable, at the expense of security. The `DataContract` serialization process allows the serialized object to contain type information that can be checked during deserialization, but the format of the serialized file is still human readable XML.

3. Binary serialization can store any type of data, but XML



4. A binary serializer can store reference types as references, whereas XML, JSON, and DataContract serialization will resolve references to obtain values that are then serialized.
5. Serialization does not store the methods in a class, or any static elements. Binary serialization will store private data members of a type. DataContract serialization can store private data members as XML text.
6. Binary serialization is very useful for taking a complete snapshot of the data content of a class. It forces the serialize and deserialize process to make use of identical classes. It is very useful for transferring an object from one process to another, where the serialized data stream will not be persisted. I'm not keen on using it to persist data for long periods of time because it is vulnerable to changes in the classes used. Text serialization such as XML is very useful if you want to transfer data from one programming language or host to another. It is highly portable. It is also very useful for storing small amounts of structured data.
7. Binary serialization produces a stream that represents the entire contents of a class, including all private content. However, with an understanding of the content, it is possible that this can be compromised, allowing the private contents of an object to be viewed and changed. Just because you can't view the contents of a binary serialized object with a text editor does not mean that it is not immune to tampering.
8. A custom serializer allows the programmer to get control of the serialization process either by creating their own serialization process or by getting control during the phases of the serialization process. These customizations are only possible when using binary data serialization.
9. All serializers use data streams to transfer data being serialized and deserialized. In Skill 3.2, in the "Encrypting data using AES symmetric encryption," section you saw that it is possible to send a data stream through an encrypting stream, making it possible to encrypt serialized data. It is also possible to compress serialized data in the same way.
10. Both XML and DataContract serializers produce XML output. In the case of XML, if a class is marked as serializable, all of the data elements in the class will be serialized and the programmer must mark as NonSerialized any data members that should not be serialized. In the case of DataContract serialization, the programmer must mark elements to be serialized. The other functional difference is that private data members can be serialized using DataContract serialization.

5 Store data in and retrieve data from collections

1. A database provides storage where database queries are used to manipulate the data stored. The data in the database is moved into the program for processing. A collection is stored in the memory of the computer and can therefore be accessed much more quickly. A database is good for very large amounts of data that won't necessarily fit in memory and have to be shared with multiple users. In-memory collections have performance many times that of data access from a database, but are limited in capacity to the memory of the computer. One major attraction of a database is the ease with which a database query can be used to extract data. However, you should remember that LINQ can be used on in-memory collections.
2. You can create an array of arrays. Each of the arrays in the array can consist of a different length, leading to the creation of what is called a "jagged" array.
3. The C# compiler will not complain if you make a twenty-dimensional array. However, it might use up a lot of computer memory, and it would certainly be very hard to visualize. Don't confuse adding array dimensions with adding properties to an object. To hold the name, birthday and address of a person you don't need a three-dimensional array, you need a one-dimensional array of Person elements.
4. Remember that C# arrays are indexed starting at 0. This means that if you have an array with 4 elements they are given the indices 0, 1, 2, and 3. In other words there is no element with the index value 4. This can be counter-intuitive, and it is also not how some other languages work, where array indices start at 1.
5. It is impossible to change the size of an array once it has been created. The only way to "add" an element is to create a new, larger, array and then copy the existing one into it.
6. The use of Length for the length of an array and Count for the number of elements in other collection types can be confusing, but the explanation is that an array has the same size at all times, so you can just get the length of it. However, a dynamic collection class such as an ArrayList can change in size at any time, and so the program will actually have to count the number of items to find out the current size.
7. Dictionaries decide where to store an item by using a hashing algorithm. You saw hashing in Skill 3.2, where a hashing function reduces a large amount of data to a single, smaller value that represents that data. The Dictionary class uses a hashing algorithm to convert the key value for the item being stored into a number that will give the location of the item. When searching for the location represented by a key, the dictionary doesn't have to search through a list of keys to find the one selected. It just has to



8. Sets are useful if you want to item properties that may grow and change over time, such as with a tag metadata. A user can generate new tags as the application is used, and the set provides operators that can search for items. The difficulty with tags is that it may be difficult to store them in fixed sized storage such as databases. Furthermore, LINQ operators can be used in place of set operations.
9. The prime difference between a stack and a queue is how the order of items is changed when they are pushed and popped. A queue retains the order, so the first item added to the queue is the first one to be removed from the queue. A stack reverses the order, so the first item to be pushed onto the stack will be the last one to be removed. If you think about it, you can reverse the order of a collection by pushing all of the elements onto a stack and then popping them off.
10. You can use the `List` type to meet all of your data storage needs, but you must put in substantial extra amounts of work to get a list to perform like a set or a dictionary.

CHAPTER SUMMARY

- A stream is an object that represents a connection to a data source. A stream allows a program to read and write sequences of bytes and set the position of the next stream operation.
- The `Stream` class is the abstract parent class that defines fundamental stream behaviors. A range of different child classes extend this base class to provide stream interaction with different data sources.
- The `FileStream` class provides a stream interface to file storage.
- A file contains a sequence of 8-bit values (bytes) that can be encoded into text using a particular character mapping. The `Encoding` class provides methods for different character mappings.
- The `TextWriter` and `TextReader` classes are abstract classes that define operations that can be performed with text in files. The `StreamWriter` and `StreamReader` class are implementations of this class that can be used to work with text files in C#.
- Stream classes have constructors that can accept other streams, allowing a program to create a "pipeline" of data processing behaviors that are ultimately connected to a storage device.
- The `File` class is a "helper" class that contains static methods that can be used to write, read, append, open, copy, rename, and delete files.
- It is important that a program using files deals with any exceptions that are thrown when the files are used. File operations are prone to throwing exceptions.
- The actual file storage on a computer is managed by a file system that interacts with a partition on a disk drive. The file system maintains information on files and directories which can be manipulated by C# programs.
- The `FileInfo` class holds information about a particular file in a filesystem. It duplicates some functions provided by the `File` class but is useful if you are working through a large number of files. The `File` class is to be preferred when working with individual files.
- The `DirectoryInfo` class holds information about a particular directory in a filesystem. This includes a list of `FileInfo` items describing the files held in that directory.
- A path describes a file on a filesystem. Paths can be absolute, thus starting at the drive letter, or relative. The `Path` class provides a set of methods that can be used to work with path strings, including extracting elements of the path and concatenating path strings.
- A C# program can use the `HttpRequest`, `WebClient`, and `HttpClient` classes to communicate with an HTTP server via the Internet. `HttpRequest` provides the most flexibility when assembling HTTP messages. `WebClient` is simpler to use and can be used with `await` and `async` to perform asynchronously. `HttpClient` only supports asynchronous use and must be used when writing Universal Windows Applications.
- Programs can (and should) perform file operations asynchronously. The `FileStream` class provides asynchronous methods. When catching exceptions thrown by file operations, ensure that the methods being awaited do not have a void return type.
- A database provides data storage for applications in the form of tables. A row in a table equates to a class instance. Each row can have a unique ID (called a *primary key*) which allows other objects to refer to that row.
- Programs interact with a database server by creating an instance of a connection object. The connection string is used to configure this connection, to identify the location of the server and to provide authentication details. ASP.NET applications can be configured with different environments for development and production, including the contents of the connection string.
- A database responds to commands expressed in Structured Query Language (SQL). SQL is plain text that contains commands and



data elements in SQL queries because a malicious user can inject additional SQL commands into the data.

- When creating ASP.NET applications, the SQL commands to update the database are performed by methods in that act on objects in the application.
- A program can download data from a web server in the form of a JSON or XML document that describes the elements in an object.
- A program can download data from a web server in the form of an XML document. XML documents can be parsed element by element or used to create a Document Object Model (DOM) instance, which provides programmatic access to the elements in the data.
- A web service takes the form of a server and a client. The server exposes a description of the service in the form of method calls that are implemented by a proxy object created by the client. The method calls in the client proxy object are translated into requests sent to the server. The server performs the requested action and then sends the response back to the client, which receives the response in the form of the result from the method call.
- LINQ allows programmers to express SQL-like queries using “query comprehension syntax.”
- LINQ queries can be performed against C# collections, database connections, and XML documents.
- A LINQ query generates an iteration. The execution of the query is deferred until the iteration is enumerated, although it is possible to force the execution of a query by requesting the query to generate a List, array, or dictionary as a result.
- LINQ queries are compiled into C# method calls. A programmer can express a query as methods if required.
- A LINQ query generates an iteration as a result. This may be an iteration of data objects or an iteration of anonymous types, which are created dynamically when the query runs.
- A program can work with anonymous types by using the `var` type, which requests that the compiler infer the type of the data from the context in which it is used. Using `var` types does not result in any relaxation of type safety because the compiler will ensure that the inferred type is not used incorrectly.
- The output from one query can be joined with a next, to allow data in different sources (C# collections or database tables) to be combined.
- The output from a query can be grouped on a particular property of the incoming data, allowing a query to create summary information that can be evaluated by the aggregate commands, which are sum, average, min, max, and count.
- LINQ to XML can be used to perform LINQ queries against XML documents held in the `XDocument` and `XElement` objects. These objects also provide behaviors that make it easy to create new XML documents and edit existing ones.
- Serialization involves sending (serializing) the contents of an object into a stream. The stream can be deserialized to create a copy of the object with all the data intact. The code content (the methods) in an object are not transferred by serialization.
- Classes that are to be serialized by the binary serializer must be marked using the `[Serializable]` attribute, which will request that all the data items in the class be serialized. It is possible to mark data items in a class with the `[NotSerialized]` attribute if it is not meaningful for them to be serialized.
- Binary serialization encodes the data into a binary file. Binary serialization serializes public and private data elements and preserves references. Sensitive data should not be serialized without paying attention to the security issues, because a binary serialized file containing private data can be compromised.
- A programmer can write their own serialization behaviors in a class, which save and restore the data items using the serialization stream. Note that customized serialization behaviors may be used to illicitly obtain the contents of private data in a class, and so must be managed in a secure way.
- A programmer can add methods that can modify the contents of a class during the serialization and deserialization process. This allows you to create classes that can create default values for missing attributes when old versions of serialized data are deserialized.
- The XML serializer serializes public elements of a class into XML text. The value of each element is stored in the file. References in objects that are serialized are converted into copies of the value at the end of the reference. There is no need for the `[Serializable]` attribute to be added to classes to be serialized using XML.
- The JSON serializer uses the JavaScript Object Notation to serialize data into a stream.
- The `DataContract` serializer can serialize public and private data elements into XML files. Classes to be serialized must be given the `[DataContract]` attribute and data elements to be serialized must be given the `[DataMember]` attribute.



of elements in it) is fixed when the array is created and cannot be changed. Array elements are accessed by the use of a subscript/index value which is 0 for the element at the start of the array. Arrays can have multiple dimensions.

- The `ArrayList` is a collection class that provides dynamic storage of elements. A program can add and remove elements. Elements in an `ArrayList` are managed in terms of references to the object type, which is the base type of all types in C#. This means that a program can store any type in an `ArrayList`.
- The `List` type uses generics to allow developers to create lists of a particular type. The list stores elements of the given type. It is used in exactly the same way as an `ArrayList`, with the difference that there is no requirement to cast elements removed from the `List` to their proper type.
- Dictionaries provide storage organized on a key value of a particular type. The key value must be unique for each item in the dictionary.
- Sets store a collection of unique values. They are useful because of the set functions that they provide. Sets are useful for storing tag values and other kinds of unstructured properties of an item.
- A `Queue` is a First-In-First-Out (FIFO) storage device that provides methods that can be used to Enqueue and Dequeue items.
- A program can customize a collection by extending the base collection type and adding additional behaviors.
- Programmers can create their own collection types by creating types that implement the `ICollection` interface.
- A `Stack` is a Last-In-First-Out (LIFO) storage device that provides methods that can be used Push items on the stack and Pop them off.

[Recommended](#) / [Playlists](#) / [History](#) / [Topics](#) / [Settings](#) / [Get the App](#) / [Sign Out](#)



PREV

[Chapter 3 Debug applications and implement security](#)

NEXT
[Index](#)

