CSC-200

Module 4

Implementing File & Database Access

Copyright ©
Symbolicon Systems
2008 – 2025

1 File Access

1.1 Directory Class

Use **Directory** class to examine and manage directories on a file system. There are methods are to create and delete a directory or get or set current directory and retrieve a list of files in the directory or a list of sub-directories. Note that because control characters can be inserted in C# strings you need to use double backslashes (\\) instead of one to indicate a separator instead of a control code. You can also place **@** character in front to tell the C# compiler that there are no control characters inside a string so the backlash becomes a normal character. Note that file access classes are defined in **System.IO** namespace. To get more information about a file or directory. If you need to obtain more information or perform directory operations using objects, then create a **DirectoryInfo** object using the directory name.

Accessing directories: Directory1\Program.cs

```
string path = @"C:\Windows";
string [] files = Directory.GetFiles(path);
string [] directories = Directory.GetDirectories(path);
foreach(string item in files) Console.WriteLine(item);
foreach(string item in directories) Console.WriteLine(item);
string source = @"C:\CSDEV\ABC";
string target = @"C:\ABC";
Directory.CreateDirectory(source);
if(Directory.Exists(target)) Directory.Delete(target);
Directory.Move(source, target);
Directory.SetCurrentDirectory(target);
Console.WriteLine(Directory.GetCurrentDirectory());
```

1.2 File Class

You may use the **File** class to create, open, delete, move and copy files on your file system. The following example will check whether a file already exists and removes it. It will then creates a text file and writes some text into the file and then makes a copy of the file. Finally the program will move the file to another folder. The **Move** method can also be used to rename a file by using the same location but a different name. To get more information about a file or perform file operations using objects, create a **FileInfo** object using the file name.

Access files: File1\Program.cs

```
string source="Original.txt";
string target=@"C:\Abc.txt";
StreamWriter writer = File.CreateText(source);
writer.WriteLine("Hello System.IO!"); writer.Close();
File.Copy(source,"Backup.txt");
if(File.Exists(target)) File.Delete(target);
File.Move(source,target);
```

1.3 Accessing Text Files

Rather than using the methods of a **File** class to create or open a file, you can directly create a **StreamWriter** or **StreamReader** to directly create, append or open a text file. When opening an existing file, you can pass **true** as the second parameter to append to the file rather than overwriting the contents of the file.

Using Stream classes directly File2\Program.cs

```
StreamWriter writer = new StreamWriter("C:\\ABC.txt",true);
writer.WriteLine("Hello System.IO.File!");
writer.WriteLine("Hello System.IO.StreamWriter!");
writer.Close();
```

Note that if a runtime error occurs while writing to the file, the file may not be property closed until the writer or reader is collected by the GC. For safety you can implement finalization using **try finally**. Code in the **finally** section will be executed regardless of whether the code in the **try** section is successful or not. Another way is to use a **using** block to obtain the writer or reader object. When the code block is executed, the object is guaranteed to be closed or disposed if it implements **IDisposable** interface. There is no need for you to manually call the **Close** method.

Ensuring the file is always closed

```
StreamWriter writer = new StreamWriter("C:\\ABC.txt",true);
try {
      writer.WriteLine("Hello System.IO.File!");
      writer.WriteLine("Hello System.IO.StreamWriter!");
}     finally { writer.Close(); }
```

A using block automatically closes and disposes an object

```
using (StreamWriter writer = new StreamWriter("C:\\ABC.txt",true)) {
      writer.WriteLine("Hello System.IO.File!");
      writer.WriteLine("Hello System.IO.StreamWriter!");
}
```

```
Console.WriteLine("Read line by line:");
StreamReader reader = new StreamReader("C:\\ABC.txt");
string line = reader.ReadLine(); // read 1st line
while(line != null) {
      Console.WriteLine(line);
      line = reader.ReadLine();  // read next line
}
reader.Close();
Console.WriteLine("Read all contents:");
reader = new StreamReader("C:\\ABC.txt");
string data = reader.ReadToEnd();
reader.Close();
Console.WriteLine(data);
```

The **File** class also provides very simple content loading and saving methods. If you have an array of text strings that you wish to save to a text file, call a **WriteAllLines** method. To load back the array of strings, call **ReadAllLines**. If you have one text string, use **WriteAllText** and **ReadAllText** instead.

Writing and reading string arrays

```
string[] lines = { "This is line 1.",
      "This is line 2.",
      "This is line 3." };
File.WriteAllLines(@"C:\File1.txt", lines);
lines = File.ReadAllLines(@"C:\File1.txt");
foreach (string line in lines) Console.WriteLine(line);
```

Writing and reading single text string

```
string text = "This is line 1.\r\n" +
      "This is line 2.\r\n" + "This is line 3.\r\n";
File.WriteAllText(@"C:\File2.txt", text);
text = File.ReadAllText(@"C:\File2.txt");
Console.WriteLine(text);
```

1.4 Accessing Binary Files

To read and write binary files, we can first use the base **FileStream** class to create a stream object for a file. The constructor will allow you to specify the filename and the access mode (**Create**, **Open**, **Append** etc). This is a low level object where you can read and write byte arrays. However, this would be a bit tedious if we need convert everything to byte arrays before writing to a file. You can then use a **BinaryReader** and **BinaryWriter** classes that can convert any data into binary before writing into a stream. The **Write** method is overloaded to convert and write any type of data into the stream.

Writing & reading binary data: File3\Program.cs

```
int id = 123;
string name = "Can of Coke";
double price = 1.20;
FileStream stream = new FileStream("C:\\ABC.bin",FileMode.Create);
BinaryWriter writer = new BinaryWriter(stream);
writer.Write(id); writer.Write(name); writer.Write(price);
writer.Close();
stream = new FileStream("C:\\ABC.bin",FileMode.Open);
BinaryReader reader = new BinaryReader(stream);
int id2 = reader.ReadInt32();
string name2 = reader.ReadString();
double price2 = reader.ReadDouble();
reader.Close(); stream.Close();
```

The base **FileStream** class provides additional methods to locate the size of the file using a **Length** property or retrieve the current position of the file pointer using the **Position** property and use a **Seek** method to move the file pointer to any location within the file.

Random file access

```
int id  = 220;
FileStream stream = new FileStream("C:\\ABC.bin",FileMode.Open);
BinaryReader reader = new BinaryReader(stream);
BinaryWriter writer = new BinaryWriter(stream);
Console.WriteLine(stream.Length);
stream.Seek(4, SeekOrigin.Begin);
Console.WriteLine(stream.Position);
Console.WriteLine(reader.ReadString());
stream.Seek(0, SeekOrigin.Begin);
writer.Write(id); reader.Close(); writer.Close();
```

1.5 Object Serialization

Serialization is a feature where objects can be written or read from a stream. This will allow you to save and load objects from disk or transfer them across a network. This applies only for data-oriented classes that are designed to store data and not for service-oriented classes that are designed for processing. Since serialization is provided by Microsoft .NET, all you need to do is to assign the **Serializable** attribute to your class.

Marking a class serializable: Time.cs

```
[Serializable]
public class Time :ICloneable,IComparable {
      :
```

To serialize and de-serialize objects in binary format, use the **BinaryFormatter** class in the **System.Runtime.Serialization.Formatters.Binary** namespace. The following example we use the binary formatter to serialize and de-serialize a **Time** object to and from a file. Alternatively, you can also make use of a **SoapFormatter** from the **System.Runtime.Serialization.Formatters.Soap** namespace to serialize in SOAP format.

Serializing to a file: Serialize1

```
Time t1 = new Time(10, 20, 30);
FileStream stream1 = new FileStream("Time.bin", FileMode.Create);
BinaryFormatter formatter1 = new BinaryFormatter();
formatter1.Serialize(stream1, t1); stream1.Close();
```

De-serializing from a file

```
FileStream stream2 = new FileStream("Time.bin", FileMode.Open);
BinaryFormatter formatter2 = new BinaryFormatter();
Time t2 = (Time)formatter2.Deserialize(stream2);
stream2.Close(); Console.WriteLine(t2.ToString());
```

2 Database Access

2.1 Creating a Database

Before we can create database applications, we would need to have a database first. Execute the SQL Server Management Studio tool to connect to a local SQL Server Express that has already been installed using the following details.

Connection Information

```
Server name      : .\SQLEXPRESS
Authentication   : Windows Authentication
```

Once the connection has been established, open and execute the following SQL script to create a new database named **SymBank** and then add a table named **Accounts** to the database.

SQL script to create a database and table: SQL\SymBank.sql

```
CREATE DATABASE SymBank
GO

USE SymBank
GO

CREATE TABLE Accounts (
      ID          INT              NOT NULL,
      Name        NVARCHAR(30)     NOT NULL,
      Balance     MONEY            NOT NULL)
GO

ALTER TABLE Accounts
      ADD CONSTRAINT PK_Accounts
      PRIMARY KEY (ID)
GO

ALTER TABLE Accounts
      ADD CONSTRAINT CK_Accounts_Name
      CHECK (LEN(Name) > 0)
GO

ALTER TABLE Accounts
      ADD CONSTRAINT CK_Accounts_Balance
      CHECK (Balance >= 0)
GO
```

Constraints are added to the database to ensure data integrity. A primary key constraint is added to ensures that no two rows can have the same **ID**. It also becomes the default order of the rows in the table and searching rows using the primary key would be the fastest. Two check constraints are added to ensure no operation can result in an empty **Name** or a negative **Balance**. Such operations will be cancelled and any changes already made will be rolled back. Notice also the use of a **NOT NULL** constraint to guarantee all columns are not optional so it is compulsory for each row to have all the values to be valid.

To ensure operational integrity you need to use stored procedures instead. This also ensures that multiple operations can be combined into a single transaction on the server so any changes can be rolled back in case the transaction fails to complete successfully. A transaction can be manually rolled back by using the **ROLLBACK TRANSACTION** command but any transaction that has not been committed using **COMMIT TRANSACTION** before the connection is closed will automatically be rolled back anyway. You can create methods in .NET that can be called to run code to perform operations and control transactions but this will not be as efficient as using a stored procedure. However for this example, only the transfer of funds between accounts is implemented as a stored procedure. All other operations will be done from .NET methods.

Stored procedure to transfer between accounts

```
CREATE PROCEDURE Transfer(@Source INT, @Target INT, @Amount MONEY) AS
      IF @Source = @Target BEGIN -- cannot transfer to same account
            RAISERROR('Cannot transfer to same account!',16,1)
            RETURN -1
      END
      IF @Amount <= 0 BEGIN -- amount cannot be less than 0
            RAISERROR('Invalid transfer amount!',16,1)
            RETURN -2
      END

      BEGIN TRANSACTION
      UPDATE Accounts SET Balance=Balance-@Amount WHERE ID = @Source
      IF @@ROWCOUNT = 0 BEGIN
            ROLLBACK TRANSACTION
            RAISERROR('Invalid source account!',16,1)
            RETURN -3
      END
      UPDATE Accounts SET Balance=Balance+@AMOUNT WHERE ID = @Target
      IF @@ROWCOUNT = 0 BEGIN
            ROLLBACK TRANSACTION
            RAISERROR('Invalid target account!',16,1)
            RETURN -4
      END
      COMMIT TRANSACTION
      RETURN
```

2.2 Connecting to Database

To access the database from Visual Studio, open the Server Explorer window. In the window you should see a **Data Connections** node. Right-click on the node and select **Add Connection...** option. Verify that you are connecting to a SQL Server instance and then enter the following details.

Connection information

```
Server name      : .\SQLEXPRESS
Authentication   : Windows Authentication
Database         : SymBank
```

You can then open that connection and see the **Accounts** table in the **Tables** section and **Transfer** procedure in the **Stored Procedures** section. Right-click on the table and select **Show Table Data** option to test adding, updating and removing rows. Add the following rows to the table.

Rows to add to table

```
100   ABC TRADING           50000.0
200   DEF LIMITED           30000.0
```

You can test the stored procedure by right-clicking over the stored procedure and select **Execute**. A dialog appears to prompt you to input the values for the parameters. You can view results in the **Output** window or open the tables. Try to transfer **10000.0** from account **100** to **200**. Refresh the table to verify that the balances have been updated.

2.3 Generating a Data Model

We will now write code to help perform data operations on the **Accounts** table in the **SymBank** database. However rather creating an application we will make a library instead. This would allow multiple applications to access the database using the code in the shared library. Use the following information to create the project in Visual Studio.

Class Library project information

```
Project Name     : SymBank.Data
Project Type     : Visual C# | Class Library
Location         : C:\CSDEV\SRC
Solution         : Module4
```

Once the project has been created you can now generate the data model from the database. A data model is a set of classes that can be generated to contain all the code necessary to connect to the database, send SQL commands to the database to perform operations including calling stored procedure. This makes it easy for you to perform data operations without knowing SQL.

Microsoft provides many different data model generation technologies including **ADO.NET Entity Data Model** (EDM) and **LINQ to SQL Classes**. EDM is more complicated and takes more time and effort to setup. If you are a beginner to data access you should start with LINQ to SQL since it is simple and rarely have any issues. To generate a LINQ to SQL data model, right-click over the project and select **Add New Item...** option. Select the **Data** node and choose **LINQ to SQL Classes** option. If this option is not available that means that the tools for LINQ to SQL has not been installed. Modify your Visual Studio installation to add in the tools. Enter **SymBank.dbml** as the filename and click the **Add** button.

Once the DBML file has been created and opened, you should see two panels. Drag and drop your tables from your database connection to the left panel and drop your stored procedures on the right panel. Save the file and the model will be generated for you immediately. You can now close the file. Two classes will be generated; **Account** class which is a data-oriented class to create objects to store data to be added to the table or to store data retrieved from the table and a **SymBankDataContext** class which is a service-oriented class to be used to perform data operations and call stored procedures. For example the following code is all that is neccessary to add a new account to the table.

Using LINQ to SQL to add a new row to a table

```
var item = new Account {
      ID = 300, Name = "XYZ PTE LTD", Balance = 50000m };
var dc = new SymBankDataContext();
dc.Accounts.InsertOnSubmit(item);
dc.SubmitChanges();
```

However rather than directly write the code in an application we will create a separate class named **AccountService** that contain methods that can be called to help perform such operations. Since the class is used to provide services and not for the purpose of storing data, you can decide to implement a **static** class where all members are forced to be static as there is no need to create objects. The data model already has a class named **Account** specifically to store data on each account.

Static class providing a static method to add an account: AccountService.cs

```
namespace SymBank.Data {
      public static class AccountService {
            public static void Add(Account item) {
                  var dc = new SymBankDataContext();
                  dc.Accounts.InsertOnSubmit(item);
                  dc.SubmitChanges();
            }
      }
}
```

Add method is not static

```
var item = new Account {
      ID = 300, Name = "XYZ PTE LTD", Balance = 50000m };
var service = new AccountService();
service.Add(item);
```

Add method is static

```
var item = new Account {
      ID = 300, Name = "XYZ PTE LTD", Balance = 50000m };
AccountService.Add(item);
```

We will now add additional methods to update and remove rows from the table. Add **Debit** and **Credit** methods to update the balance of an account. We can do checking to verify that transaction amount cannot be zero or less. To query the a database, use LINQ (Language Integrated Query) extension methods. To use the methods make sure you import the **System.Linq** namespace. The methods work for arrays and collections as well, not only for a database table.

Namespace for LINQ

```
using System.Linq;
```

To locate an account to update or remove use the LINQ **Single** method. Pass in a lambda expression to determine the account to retrieve. A runtime error will occur automatically if the account cannot be located.

Methods to debit, credit and remove an account

```
public static void Debit(int id, decimal amount) {
      if (amount <= 0) throw new Exception("Invalid debit amount.");
      var dc = new SymBankDataContext();
      var item = dc.Accounts.Single(account => account.ID == id);
      item.Balance += amount; dc.SubmitChanges();
}

public static void Credit(int id, decimal amount) {
      if (amount <= 0) throw new Exception("Invalid credit amount.");
      var dc = new SymBankDataContext();
      var item = dc.Accounts.Single(account => account.ID == id);
      item.Balance -= amount; dc.SubmitChanges();
}

public static void Remove(int id) {
      var dc = new SymBankDataContext();
      var item = dc.Accounts.Single(account => account.ID == id);
      dc.Accounts.DeleteOnSubmit(item);
      dc.SubmitChanges();
}
```

You can easily called stored procedures as well. The following method is used to transfer money between two accounts by calling the **Transfer** procedure.

Calling a stored procedure

```
public static void Transfer(int source, int target, decimal amount) {
      var dc = new SymBankDataContext();
      dc.Transfer(source, target, amount);
}
```

We have implemented all the operations to update the **Accounts** table, the rest is for querying data in the table. This is where LINQ extension methods will be mostly used. To retrieve and return a single account use the **Single** method as usual.

Method that returns a single account

```
public static Account GetItem(int id) {
      var dc = new SymBankDataContext();
      return dc.Accounts.Single(account => account.ID == id);
}
```

You can download multiple rows from a database into an array, list or dictionary by calling the LINQ **ToArray**, **ToList** or **ToDictionary** methods.

Method to return a collection of accounts

```
public static List<Account> GetList() {
      var dc = new SymBankDataContext();
      return dc.Accounts.ToList();
}
```

There are many other LINQ methods to help query a database; **Where** method can be used to filter rows so that not all rows will be selected, **OrderBy** method to change the order of the selected rows, **Select** to determine what is returned instead of the entire row, **Distinct** to ensure that the result has no duplicates. The following shows how to retrieve a list of account names without duplicates for accounts that have a non-zero balance ordered by the account name into an array.

Using LINQ extension methods

```
public static string[] GetNameList() {
      var dc = new SymBankDataContext();
      return dc.Accounts
            .Where(account => account.Balance > 0)
            .OrderBy(account => account.Name)
            .Select(account => account.Name)
            .Distinct().ToArray();
}
```

LINQ is integrated with the programming language so C# has keywords you can use to construct a query without having to call LINQ extension methods directly as show below. However not all LINQ methods have corresponding keyword so methods like Distinct and ToArray have to be called normally.

Using the C# query language

```
public static string[] GetNameList() {
      var dc = new SymBankDataContext();
      return (from account in dc.Accounts
            where account.Balance > 0
            orderby account.Name
            select account.Name)
            .Distinct().ToArray();
}
```

LINQ also provide aggregation methods like Min, Max, Sum, Average and Count that returns a single scalar value. The following demonstrates that the methods can be applied to any **IQueryable** instances at any stage of the query.

Using LINQ aggregation methods

```
var n0 = dc.Accounts.Count();
var n1 = dc.Count(item => item == host);
var n2 = dc.Sum(item => item.Balance);
```

Return total number of accounts and total balance

```
public static int Count {
      get {
            var dc = new SymBankDataContext();
            return dc.Accounts.Count();
      }
}

public static decimal TotalBalance {
      get {
            var dc = new SymBankDataContext();
                  return dc.Accounts
                        .Where(account => account.Balance > 0)
                        .Sum(account => account.Balance);
            }
      }
}
```