

CS233S Lab 5

Input and Output

Goals:

- Practice with text and binary files.
- Use the converter classes to read and write the simple types.
- Work with files and directories in the filesystem.

Overview

The `System.IO` namespace contains two broad categories of types: those that work with the filesystem and those that manipulate file contents.

The filesystem classes allow examination and manipulation of files and directories. For example, an individual file can be tested for existence, moved, copied, deleted, have its attributes examined, etc. Similarly, a directory can be created, moved, deleted, have its contents examined, etc.

The file content classes deal with two types of files: binary and text. The binary file classes offer methods to read and write bytes. The text file classes offer methods to write all the basic types (converting them to characters) but only methods to read characters and strings. The data can be located on disk or in memory and separate classes are used for the different cases.

Converters are supplied to store the basic types such as `short`, `int`, `float`, `double`, etc. in a binary file. The converters work for both input and output: the output converter converts to bytes while the input converter reads raw bytes and reassembles them into types. The converters are separate class that must be used together with a binary file.

Object serialization can be used to save/restore objects to a stream. Applications for this technology include persistence (i.e. saving state to disk) and remoting (i.e. sending an object across a network connection).

Part 1 – Basic text file

Here we experiment with a basic text file: writing into the file and then reading back what was written.

steps:

1. Use a `StreamWriter` to open a text file for writing. Use the constructor that takes the name of the file as an argument.
2. Use the `WriteLine` methods to write data into the file. There are several overloaded `Write` and `WriteLine` methods available for many of the common types. The data is converted to characters and written into the file. `WriteLine` adds a line terminator to the output while `Write` does not.
3. Close the file.

4. Use a `StreamReader` to open the same file for reading. Use the constructor that takes the name of the file as an argument.
5. Use the `ReadLine` method to read a line at a time from the file. `ReadLine` returns the next line in the file as a `string`. Print each line to the Console as you read it. `ReadLine` will return `null` when it reaches the end of the file.
6. Close the file.

Part 2 – Basic binary file

Here we experiment with a binary file, writing some bytes and then reading back what was written.

steps:

1. Use a `FileStream` to open a binary file. Use the constructor that takes the name of the file and the mode, here we want to create a new file.
2. There are two options for writing: write a single byte or write an array of bytes. Use the `WriteByte` method to write a few bytes of data into the file.
3. Close the file.
4. Open the same file, again using a `FileStream`. There are not separate classes for reading and writing binary files as there are for text files. Use the constructor that takes the name of the file and the mode, here we want to open an existing file.
5. There are two options for reading: read a single byte or read an array of bytes. Use the `ReadByte` method to read each byte in the file. `ReadByte` returns the next byte of data embedded in a 4 byte integer. When the end of the file is reached, `-1` is returned.
6. Close the file.

Part 3 – Simple type converters

Here we employ converters to write and read structured types such as `int`, `float`, etc. to a binary file on disk.

steps:

1. Use a `FileStream` to open a binary file. Use the constructor that takes the name of the file and the mode, here we want to create a new file.
2. Create a `BinaryWriter` to convert from structured types to bytes. Pass the `FileStream` to the constructor.
3. Call a few of the overloads of `Write` on the converter. The converter will convert to bytes and write the bytes to the underlying stream.
4. Call `Close` on the converter. This will also close the underlying stream.
5. Open the same file, again using a `FileStream`. Use the constructor that takes the name of the file and the mode, here we want to open an existing file.

6. Create a `BinaryReader` to convert from bytes to structured types. Pass the `FileStream` to the constructor.
7. Call the various read methods on the converter. The converter will read some bytes from the underlying stream and convert them into the requested type. For example, the `ReadInt32` method will read the next 4 bytes of the file and convert them into an `int` and return the integer value. Be careful to read the values back in the same order they were written. If you are unsure about the file length, you can wait for the stream to throw an `EndOfStreamException` when you attempt to read. Print each value to the console as you read it.
8. Call `Close` on the converter. This will also close the underlying stream.

Part 4 – Directory search

Here we use the filesystem manipulation classes to write a utility to search for a particular directory.

steps:

1. Implement the `FindDirectory` method shown below. Traverse the entire directory tree below the specified starting point and search for all directories whose name matches the target. For each match, print out the full name of the directory (the full name will include the path).

```
public static void FindDirectory(string target, DirectoryInfo current)
{
    // ...
}
```

Part 5 – Serialization

Applications often need to save data to persistent storage such as a database. Databases are powerful, but are often expensive to buy and time consuming to administer. An alternative approach for simple applications is to use object serialization to save data to a disk file. In this exercise, we code a simple address book that uses serialization to save the book to disk.

steps:

1. Create a class to represent a “contact” i.e. an address book entry. Store the name and email address of the contact. Feel free to add any methods that would be convenient; for example, a constructor or an override of `ToString` might be convenient for testing. The contacts will be saved to disk using object serialization. Apply the `Serializable` attribute to the `Contact` class to make it eligible for serialization.

```
class Contact
{
    string name;
    string email;
    ...
}
```

2. Create a class to represent the address book.

```
class AddressBook
{
    ...
}
```

We need to store an arbitrary number of contacts inside our address book. Luckily, the .NET Framework Class Library has a class called `ArrayList` from the `System.Collections` namespace that is perfect for our needs. The `ArrayList` will handle all the memory management issues for us, allowing us to add as many contacts as necessary without having to worry about running out of room to store them. Add a field of type `ArrayList` to the address book class.

```
class AddressBook
{
    ArrayList contacts = new ArrayList();
    ...
}
```

Code an `Add` method for the address book. Add the specified contact to the `ArrayList` using the `Add` method of the `ArrayList` class.

```
class AddressBook
{
    public void Add(Contact contact)
    {
        ...
    }
    ...
}
```

3. Next we need to save all the contacts to disk. Fortunately, the `ArrayList` class is serializable so we can write the entire `ArrayList` all at once. Serialization will write the entire object graph which means the `ArrayList` and all the contacts will be stored on disk. Add a `Save` method to the address book class that saves the contacts to a disk file named `MyContacts.dat`.

```
class AddressBook
{
    public void Save()
    {
        ...
    }
    ...
}
```

4. Add a Load method to the address book class that reads the contacts in from the MyContacts.dat disk file. Since we wrote the entire ArrayList as a single object, reloading should be very simple: a single deserialize will reconstruct the ArrayList and all the contacts.

```
class AddressBook
{
    public void Load()
    {
        ...
    }
    ...
}
```

5. Add a Print method to the address book class that prints out all the contacts. A foreach loop can be used to step through the contents of the ArrayList.

```
class AddressBook
{
    public void Print()
    {
        ...
    }
    ...
}
```

6. Create a driver class with a Main method to test the address book. A sample driver is shown below.

```
class SerializationDriver
{
    static void Main()
    {
        AddressBook friends = new AddressBook();

        friends.Add(new Contact("Ann", "ann@develop.com"));
        friends.Add(new Contact("Bob", "bob@develop.com"));

        friends.Print();
        friends.Save();
        friends.Load();
        friends.Print();
    }
}
```