

# I/O

## C# Programming

February 21

# I/O

- Up until now, the only reading/writing to the file system we have seen is by way of settings files
- We will now look at:
  - Navigating directories
  - Binary file I/O
  - Text file I/O
  - Asynchronous I/O
  - Serialization
- The classes we need for I/O are in the `System.IO` namespace

# Part I

## Directories/Paths

# Directories

- The `Directory` class provides static methods for creating directories and navigating the file system
- The `DirectoryInfo` class provides information about a directory – including creation time, last access time, etc.
- We will look at a couple examples that walk the file system and print directories and files

## An aside – string literals

- As usual with string literals, in C# the backslash and other special characters need to be escaped
- For example:  
`"C:\\Users\\Ravi"`  
would be the path of a Windows directory
- There are many occasions when we would rather the characters be taken verbatim (like when we are dealing with file paths, which have a lot of backslashes)
- Prefixing a string literal with @ results in the characters of the string being parsed as-is
- For example:  
`@"C:\Users\Ravi"`

## Printing top-level directory structure

```
void PrintDirectoryStructure() {  
    DirectoryInfo di = new DirectoryInfo(@"C:\");  
    Console.WriteLine(di.Name);  
    foreach (DirectoryInfo d in di.GetDirectories()) {  
        Console.WriteLine("  " + d.Name);  
    }  
}
```

## Printing project directory structure

- There are a few ways to get the path from which the program is being run:
  - `Directory.GetCurrentDirectory()`
  - `Environment.CurrentDirectory`
  - `Application.StartupPath`
- If the program is running in the default location where it was built, the path will be something like  
`..\bin\Debug`
- Since most of the contents of the project directory are a couple levels up, we can navigate relative to this startup directory

```
DirectoryInfo di = new DirectoryInfo(  
    Directory.GetCurrentDirectory());  
di = di.Parent.Parent;
```

## Printing project directory structure

```
void PrintProjectStructure() {
    DirectoryInfo di = new DirectoryInfo(
        Directory.GetCurrentDirectory());
    di = di.Parent.Parent;
    RecPrintDir(di, 1);
}

void RecPrintDir(DirectoryInfo di, int depth) {
    string indent = "";
    for (int i = 0; i < depth; i++)
        indent += " ";
    Console.WriteLine(indent + di.Name + "/");
    foreach (DirectoryInfo d in di.GetDirectories())
        PrintDir(d, depth + 1);
    foreach (FileInfo f in di.GetFiles())
        Console.WriteLine(indent + f.Name);
}
```

## Part II

### Reading and writing data

## Reading and writing data

- We now begin looking at reading and writing data to files
- For all examples, we will place our input and output files in the startup directory of the application
- I.e., in the Debug folder of the project

# Binary I/O

- If we know that a particular file is text, we can use specialized classes to operate on it
- In the general case, however, a file is just an array of bytes
- The most general way to read and write files is using the `Stream` class
- We will look at an example that copies the contents of one file to another
- Here we will not worry about exceptions that can be thrown – but I/O exceptions should be handled in practice!

```
static void CopyFile() {
    string dir = Directory.GetCurrentDirectory();
    Stream istream =
        File.OpenRead(dir + @"\testfile.txt");
    Stream ostream =
        File.OpenWrite(dir + @"\testfile2.txt");
    byte[] buffer = new byte[1024];
    int bytesRead = istream.Read(buffer, 0, 1024);
    while (bytesRead > 0) {
        ostream.Write(buffer, 0, bytesRead);
        bytesRead = istream.Read(buffer, 0, 1024);
    }
    istream.Close();
    ostream.Close();
}
```

# Binary I/O

- The `Stream` class gives complete control over how to access the file: where to read/write from, the exact number of bytes to manipulate at a time
- The downside to calling `Stream`'s `Read()` and `Write()` methods is that these disk operations are only performed when explicitly stated

## Buffered streams

- Instead, we can use **buffered streams**, which decide how much data to read and write to the disk and when
- Using the `BufferedStream` class makes reads and writes more efficient
- Since it may have already fetched more data from disk than previously requested, a `Read()` might only have to read in-memory data instead of going to the disk
- Similarly, the buffered stream may choose to not immediately write all the contents of a `Write()` to disk, instead waiting for a better time
- We need to make only a few changes to our previous example to make use of buffered streams

```
static void CopyFile() {
    string dir = Directory.GetCurrentDirectory();
    Stream istream =
        File.OpenRead(dir + @"\testfile.txt");
    Stream ostream =
        File.OpenWrite(dir + @"\testfile2.txt");
    BufferedStream bistream = new BufferedStream(istream);
    BufferedStream bostream = new BufferedStream(ostream);
    byte[] buffer = new byte[1024];
    int bytesRead = bistream.Read(buffer, 0, 1024);
    while (bytesRead > 0) {
        bostream.Write(buffer, 0, bytesRead);
        bytesRead = bistream.Read(buffer, 0, 1024);
    }
    bistream.Close();
    bostream.Flush();
    bostream.Close();
}
```

## Buffered streams

- Notice the call to `Flush()` on the output stream
- Since the buffered stream may not write the data to disk immediately, we need to explicitly make sure they have been written before we exit
- If there is still data in the buffer waiting to get written, simply closing the buffer will not write these to disk

## Text I/O

- If we know that the bytes of a file are to be interpreted as characters, we can use another type of stream
- The `StreamReader` and `StreamWriter` classes allow reading and writing of entire lines of text with a simpler method interface

```
static void CopyFile() {
    string dir = Directory.GetCurrentDirectory();
    StreamReader istream =
        new StreamReader(dir + @"\testfile.txt");
    StreamWriter ostream =
        new StreamWriter(dir + @"\testfile2.txt", false);
    String text;
    do {
        text = istream.ReadLine();
        ostream.WriteLine(text);
    } while (text != null);
    istream.Close();
    ostream.Close();
}
```

## Text I/O

- Note that the boolean flag in the `StreamWriter` constructor signifies whether or not to append to the file if it already exists
- Again, we can make use of a buffered stream to improve efficiency
- Instead of passing the file path as a string to the `StreamReader` and `StreamWriter` constructors, pass them `BufferedStream` objects

## Asynchronous I/O

- All of the examples we have seen so far performed **synchronous** I/O
- You can also make **asynchronous** I/O requests
- That is, you can start an I/O operation and then continue executing some other logic
- When the file operation is complete, a callback delegate can be invoked

## Asynchronous I/O

- Instead of using a Stream's Read() method, you use the BeginRead() method:

```
inputStream.BeginRead(  
    buffer,      // array of bytes to hold result  
    0,          // offset in file  
    1024,       // number of bytes to read  
    myCallBack, // callback delegate  
    null);     // state object
```

- Note that the state object can be anything

## Part III

# Serialization

# Serialization

- **Serialization** is the process of writing an object out to a stream as a series of bytes
- This allows objects to be saved to disk and reloaded in the future
- The CLR takes care of this process
- All the programmer has to do is mark the class as `[Serializable]`
- If the class contains references to other classes, each of these must be marked `[Serializable]` as well

# Serialization

- Serialization is performed by the `BinaryFormatter` class in the `System.Runtime.Serialization.Formatters.Binary` namespace
- Consider an example with the simple class:

```
[Serializable]
class Foo {
    public int bar = 1;
}
```

## Serializing an object

```
static void Main(string[] args) {  
    Foo foo = new Foo();  
    foo.state = 99;  
    Serialize(foo);  
}
```

```
static void Serialize(Foo foo) {  
    String dir = Directory.GetCurrentDirectory();  
    Stream ostream = File.OpenWrite(dir + @"\foo.txt");  
    BinaryFormatter bf = new BinaryFormatter();  
    bf.Serialize(ostream, foo);  
    ostream.Close();  
}
```

## Deserializing an object

```
static void Main(string[] args) {  
    Foo foo = Deserialize();  
    Console.WriteLine(foo.state.ToString());  
}  
  
static Foo Deserialize() {  
    String dir = Directory.GetCurrentDirectory();  
    Stream istream = File.OpenRead(dir + @"\foo.txt");  
    BinaryFormatter bf = new BinaryFormatter();  
    Foo foo = (Foo)bf.Deserialize(istream);  
    istream.Close();  
    return foo;  
}
```

## Time/space tradeoff

- If the object being serialized contains a lot of data, then the size of the file will grow large
- If there is certain data that can be recomputed, you can choose not to serialize that particular data
- The tradeoff is that since that data would not be recovered from the deserialization process, it would need to be recomputed

## Time/space tradeoff

- To tell the serializer to not serialize particular data in the class, mark that member as `[NonSerialized]`
- You can also tell the serializer to call a specified method when deserialization is finished (you can put logic for recomputing the missing data here)
- To achieve this, implement the `IDeserializationCallback` interface
- This requires you to implement `void OnDeserialization(object sender);`