impera

Objektorienterad programmering med C#

Kursträff 8

Exception-handling, I/O, using

Today's lecture

1. **Exception-handling**
    - Catching exceptions
    - Throwing exceptions
    - Creating exception classes
2. **Input/Output**
    - System.IO
    - Streams
        - FileStream
        - StreamReader and StreamWriter
        - BinaryReader and BinaryWriter
    - Handling I/O exceptions
        - IDisposable and using

Exceptions

- Exceptions are *thrown* when errors occur
  - An exception is an object of the class `Exception` or one of its subclasses
- To handle exceptions you *catch* them
- Unhandled exceptions will crash your application
  - If you're debugging your application, Visual Studio will display the Exception
- Exceptions are handled using `try/catch` blocks

Catching exceptions

- To handle exceptions use the `try/catch` block:

  -
    ```
    try {
        var num = decimal.Parse("Not a number");
    } catch {
        Console.WriteLine("An error occured!");
    }
    ```
- `try/catch` block that accesses the `Exception` object:

  -
    ```
    try {
        var num = decimal.Parse("Not a number");
    } catch (Exception ex) {
        Console.WriteLine(ex.Message);
    }
    ```

Catching exceptions

- Catch specific exception types:

  -
    ```
    try {
        var num = decimal.Parse("Not a number");
    } catch (FormatException) { // No variable
        Console.WriteLine("FormatException!");
    } catch (OverflowException ex) { // Named variable
        Console.WriteLine("OverflowException!");
    } catch (Exception) {
        Console.WriteLine("Some other exception.");
    } finally {
        // This always happens, whether an exception
        // was thrown or not.
    }
    ```

Throwing exceptions

- You throw exceptions using the keyword throw:

  -
    ```
    public static class Helper {
        public static int ConvertStringToInt(string input) {
            if (int.TryParse(input, out int result)) {
                return result;
            } else {
                throw new Exception("TryParse-error!");
            }
        }
    }
    ```
- You can also re-throw an exception you’ve caught:

  -
    ```
    public static class Helper {
        public static int ConvertStringToInt(string input) {
            try {
            } catch (FormatException ex) {
                Console.WriteLine("Couldn’t convert string to int!");
                throw ex;
            }
        }
    }
    ```

Creating Exception classes

- Custom Exception classes are created by inheriting Exception:

  -
    ```
    public class FetchException : Exception {
        public int TimeElapsed { get; set; }
        public FetchException() {}
        public FetchException(int timeElapsed) {
            TimeElapse = timeElapsed;
        }
    }
    ```
- Throwing and catching a custom exception:

  -
    ```
    if (data == null) {
        throw new FetchException(10);
    }
    ```

  -
    ```
    try {
        doSomething();
    } catch (FetchException ex) {
        Console.WriteLine("doSomething() failed after " + ex.TimeElapsed + " seconds");
    }
    ```

Input/Output

- The namespace `System.IO` contains classes for handling files
  - Useful static classes in `System.IO` include
    - `Directory` which is used to handle directories
    - `File` which is used to handle files
    - `Path` which is used to handle paths
  - Example:

    -
      ```
      var dir = @"C:\C#-HT-18";
      var filePath = dir + "\\test.txt";
      if (!Directory.Exists(dir)) { // static method
          Directory.CreateDirectory(dir);
      }
      if (File.Exists(filePath)) {
          File.Delete(filePath);
      }
      ```

Files and Streams

- Data is read and written using *streams*
  - Two types of streams:
    - Binary
    - Text
  - Classes used to work with streams:
    - `FileStream`
    - `StreamReader`
    - `StreamWriter`
    - `BinaryReader`
    - `BinaryWriter`

FileStream

- FileStreams are opened in different *modes, access-types,* and *share-types*:
  - `var fs = new FileStream(path, mode, access, share);`
    `// access and share are optional parameters`

FileMode	FileAccess	FileShare
Append	Read	None
Create	**ReadWrite**	**Read**
CreateNew	Write	ReadWrite
Open		Write
OpenOrCreate		
Truncate		

StreamWriter

- FileStreams opened with `FileAccess.Write` or `FileAccess.ReadWrite` can be written to with `StreamWriter`:

  -
    ```
    var fs = new FileStream(
        @"C:\Temp\test.txt",
        FileMode.Create,
        FileAccess.Write
    );

    var sw = new StreamWriter(fs);

    sw.Write("Hello!");
    sw.WriteLine("Hi!");
    sw.Close();
    ```

impera

StreamReader

- FileStreams opened with `FileAccess.Read` or `FileAccess.ReadWrite` can be read from with `StreamReader`:

    -
      ```
      var fs = new FileStream(
          @"C:\Temp\test.txt",
          FileMode.Open,
          FileAccess.Read
      );

      var sr = new StreamReader(fs);

      var character = sr.Read();
      var line = sr.ReadLine();
      var wholeFile = sr.ReadToEnd();

      sr.Close();
      ```

BinaryWriter

```
var fs = new FileStream(
    @"C:\Temp\test.txt",
    FileMode.Create,
    FileAccess.Write
);

var bw = new BinaryWriter(fs);

bw.Write(true); // write a boolean value to file
bw.Write(3.0m); // write a decimal value to file
bw.Write(0);    // write an int value to file

bw.Close();
```

BinaryReader

```
var fs = new FileStream(
    @"C:\Temp\test.txt",
    FileMode.Create,
    FileAccess.Write
);

var br = new BinaryReader(fs);

var c = sw.Read();
var b = sw.ReadBoolean();
var d = sw.ReadDecimal();
var i = sw.ReadInt32();
var s = sw.ReadString();

sw.Close();
```

Handling I/O exceptions

```
FileStream fs = null;

try {
    fs = new FileStream(@"C:\Temp\test.txt", FileMode.Open);
    // Read from or write to file here
} catch (FileNotFoundException) {
    // The file does not exist
} catch (DirectoryNotFoundException) {
    // The directory does not exist
} catch (IOException) {
    // Some other I/O error
} finally {
    if (fs != null) {
        fs.Close(); // FileStreams must be closed after use!
    }
}
```

IDisposable

- Streams, writers, and readers all implement the `IDisposable` interface:

    -
      ```
      public interface IDisposable {
          void Dispose();
      }
      ```
- `Dispose()`-methods free any resources used by the class
    - Equivalent to `Close()` on streams and writers/readers

using

- Using is a control structure for working with `IDisposable` objects:

  -
    ```
    using (var fs = new FileStream("test.txt", FileMode.Read)) {
        // Do something with file.
    }
    ```
- Using is the equivalent of try/finally.

  -
    ```
    FileStream fs = null;

    try {
        fs = new FileStream("test.txt", FileMode.Read);
        // Do something with file.
    } finally {
        if (fs != null) {
            fs.Dispose();
        }
    }
    ```

using - Example

```
using (var fs = new FileStream(@"C:\Temp\test.txt", FileMode.Write)) {
    using (var sw = new StreamWriter(fs)) {
        sw.WriteLine("It works!");
    }
}
```