Pentalog

I/O and Exceptions

DATA ANALYSIS
UX/UI
FRONT-END
TECHNOLOGY
NEAR/OFFSHORE
AGILITY
BACK-END
SPRINT
KANBAN
CAMPAIGNS
GROWTH HACKING
SCRUM
BACKLOG
DEVOPS
DESIGN
SEO
CONTINUOUS INTEGRATION
MOBILE
QA
AUTOMATION
RESPONSIVE
UNIT TESTING

I/O and Exceptions

- Input/Output
  - Path
  - File
  - Directory
  - Environment
  - IDisposable
  - Streams
  - Stream readers and writers
- Exceptions

Path, File and Directory

- These are classes defined in the System.IO namespace.
  - The Path class provides methods for working with strings that represent a file or directory path
  - The File class provides methods for interacting with files
  - The Directory class provides methods for interacting with directories and their contents
- These are static classes. They cannot be instantiated.
- They contain static methods that perform operations

Path

- The Path class contains methods for working with strings that represent filesystem paths
- When working with paths, you should use the methods inside the Path class instead of manually concatenating strings, or using substring
- Some useful methods are:
  - Combine: is used to combine multiple strings to form a path
  - GetFileNameWithoutExtension: extracts the file name without the extension from a path
  - GetExtension: extracts the extension of a file from a path
  - GetFullPath: it returns the absolute path of the specified path
  - GetTempFileName: creates a unique, empty temporary file on disk and returns the path to that file
  - GetTempPath: returns the path of the user's temporary folder

Path

- Other useful methods are:
  - GetInvalidPathChars: gets a list of characters that cannot be used in a path
  - GetInvalidFileNameChars: gets a list of characters that cannot be used in the name of a file
- You can get a list of all the methods from Path in the documentation: https://docs.microsoft.com/en-us/dotnet/api/system.io.path?view=netframework-4.8

File

- The File class contains methods used for creating, copying, moving and opening of files
- Some useful methods are:
  - ReadAllText, ReadAllLines: reads all the text inside a file and returns it in a string object or as an array of string objects
  - WriteAllText, WriteAllLines: writes a string, or an array of strings to a file
  - AppendAllText, AppendAllLines: appends a string, or an array of strings to a file
  - Copy, Move, Delete: copies, moves or deletes a file
  - Exists: checks if a file exists
  - Create, CreateText: creates a new file, returning a FileStream or StreamWriter object
  - Open, OpenRead, OpenWrite: opens an exising file, returning a FileStream object
- You can get a list of all the methods from the documentation: https://docs.microsoft.com/en-us/dotnet/api/system.io.file?view=netframework-4.8

Directory

- The Directory class contains methods for creating, moving and enumerating directories
- Some useful methods are:
  - Exists: checks if a directory exists
  - CreateDirectory: creates a new directory
  - Move: moves the directory to another location on the same volume
  - GetFiles: gets a list of files in the directory
  - GetDirectories: gets a list of subdirectories in the directory
- You can get a list of all the methods from the documentation: https://docs.microsoft.com/en-us/dotnet/api/system.io.directory?view=netframework-4.8

Environment

- This is a class used to access the current environment.
- This class is not really used for input/output, but it has some useful methods we can use when working with files
- Some useful methods are:
  - GetEnvironmentVariables: gets a list of the defined environment variables
  - ExpandEnvironmentVariables: replaces the name of environment variables with their value
  - GetFolderPath: gets the path to special system folders
- You can get a list of all the methods from the documentation: https://docs.microsoft.com/en-us/dotnet/api/system.environment?view=netframework-4.8

IDisposable

- There are multiple types of resources we use in an application: memory, database connections, files, network ports and other operating system resources
- The design of the .NET Framework automatically manages memory, so we don’t have to do release it ourselves like in other languages
- Other types of resources still have to be managed manually, but the .NET Framework has implemented a way to make this easier for us
- The IDisposable interface is defined in the System namespace, and it contains a single method: void Dispose();
- Classes can implement this interface and they can release the resources they represent when the user calls the Dispose method. This is a way to specify to the user that they should manually release these resources

The using statement

- The C# language implements the using keyword that will initialize an object that implements IDisposable and will automatically dispose it when the user leaves the using block
- The syntax of the using statement is:

```
using (MyDisposableClass object2 = new MyDisposableClass())
{
     object2.UseResource();
}
```

Streams

- A stream is an object used to transfer data. It provides a generic view of a sequence of bytes.
- This is an abstract class. Classes that need to read or write bytes from a source will derive from the Stream class.
- The Stream class implements the IDisposable interface
- Concrete classes that derive from Stream:
  - FileStream: reads or writes bytes to or from a physical file on disk
  - MemoryStream: reads or writes bytes to or from memory
  - BufferedStream: reads or writes bytes to or from other streams, improving performance of certain operations
  - NetworkStream: reads or writes bytes to or from a network socket
  - PipeStream: reads or writes bytes to or from processes
  - CryptoStream: is used for performing cryptographic operations on data read from streams

Stream readers and writers

- These are objects used for reading and writing to or from streams
  - StreamReader: it is used to read characters or strings from a stream by converting bytes into characters
  - StreamWriter: it is used to write characters or strings to a stream by converting characters into bytes
  - BinaryReader: it is used to read primitive data types from streams
  - BinaryWriter: it is used to write primitive data types to streams

Exceptions

- C# provides error handling by using exceptions
- An exception is an object that contains information about an abnormal situation. It is used to convey this information to the developer in order to help him handle that situation
- Exceptions can be thrown by the CLR, or by the user
- C# provides built-in classes for a lot of possible exceptions, and they are all derived from the Exception class
- C# uses the try…catch statement in order to handle exceptions

System exceptions

- The .NET Framework defines some exceptions that can be used for different situations:
  - NullReferenceException: Thrown when we try to access a property or method of an object that is null
  - InvalidOperationException: Thrown when a method call is invalid for the object's current state
  - ArgumentException: Thrown when there is a problem with an argument
    - ArgumentNullException: Thrown when a required argument is null
    - ArgumentOutOfRangeException: Thrown when an argument is outside the required range
  - IndexOutOfRangeException: Thrown when you try to access an element of an array or collection with an index that is out of bounds
  - AccessViolationException: Thrown when you try to read or write protected memory
  - StackOverflowException: Thrown when the execution stack overflows
  - OutOfMemoryException: Thrown when there is not enough memory to continue the execution

Exception properties

- Exceptions have properties that contain information about the unexpected situation that occurred:
  - Message: a string containing a description of the exception
  - Source: a string containing the name of the application or the object that throws the error
  - StackTrace: a string containing the frames on the call stack
  - Data: a dictionary of key/value pairs that provide additional information about the exception
  - InnerException: the exception instance that caused the current exception

Throwing an exception

- The user can signal an abnormal situation by creating a new exception and throwing it, using the throw statement
- Throwing an exception will stop the execution of the current method. The CLR will start looking for an exception handler (a catch block)
- If an exception handler cannot be found in the current method, then the CLR will recursively look in the caller method
- If there was no handler found in the call stack, then the CLR will terminate the current execution thread.

Handling an exception

- Catching and handling an exception is done using the catch block

```
try
{
     Console.WriteLine("Before throwing");
     throw new Exception("Exception message");
     Console.WriteLine("After throwing");
}
catch (Exception ex)
{
     Console.WriteLine("Caught exception with the message: " + ex.Message);
}
```

The finally block

- We can use the finally block in order to guarantee that a piece of code is being executed
- This is used to release resources even when exceptions are thrown

```
try
{
     // Acquire resource
     // Use resource
}
catch(Exception ex)
{
     // Handle exception
}
finally
{
     // Release resource
}
```

Rethrowing an exception

- After catching an exception inside a catch block, you can optionally throw it again
- This is useful when you want to intercept an exception in a lower level component of the application and then you want to send it to a higher level component
- There are two ways of rethrowing an exception:
  - throw ex : will reset the stack trace to the location when you are throwing
  - throw : will preserve the original exception’s stacktrace

User-defined exceptions

- We can create our own exception types by creating a new class that derives from System.Exception
- Custom exceptions are created to add extra properties that the predefined exceptions do not already provide
- When adding new properties, we should override the ToString() method in order to display the new properties

Best practices

- Exceptions should not be used to change the flow of the program as part of a normal execution. They should only be used to report and handle error conditions and exceptional situations.
- Exceptions should not be returned as a return value or parameter. Instead, they should be thrown immediately after being created
- You should not throw System.Exception, System.SystemException, System.NullReferenceException, System.OutOfMemoryException or System.IndexOutOfRangeException intentionally from your own code
- You should not throw exceptions only in debug mode, but not in release mode. This makes debugging an application very difficult
- Use built-in exception types whenever you can

Homework

- Load a list of numbers from a file named numbers.txt.
- Read each string in the file and try to parse the numbers.
- For the parsing, create your own TryParse method, which would handle the operation. Using the original int.TryParse method is not allowed (but using int.Parse is allowed). Your code should handle the exception raised by int.Parse
- Write all the correct numbers inside a file named correctNumbers.txt.
- Write all the strings which are not numbers inside a file named incorrectNumbers.txt.
- If numbers.txt is missing, display an error.
- If the correct & incorrect files are missing, they should be created by your code.

Reference

- https://support.microsoft.com/en-us/help/304430/how-to-do-basic-file-i-o-in-visual-c
- https://docs.microsoft.com/en-us/dotnet/api/system.io.path?view=netframework-4.8
- https://docs.microsoft.com/en-us/dotnet/api/system.io.file?view=netframework-4.8
- https://docs.microsoft.com/en-us/dotnet/api/system.idisposable?view=netframework-4.8
- https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/using-statement
- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/exceptions/
- https://docs.microsoft.com/en-us/dotnet/standard/exceptions/best-practices-for-exceptions