Contents

Files

The code for this lecture is available in this archive[1].

Motivation

Files are useful for permanency: when a program terminates, all the objects created, strings entered by the user, and even the messages displayed on the screen, are lost. *Saving* some information as files allows to retrieve this information after the program terminates, possibly using a different tool. *Retrieving* information from a file allows to continue works that was started by a different program, or to use a better-suited tool to carry on some task (typically, counting the number of words in a document).

This lecture is concerned with files: how to write into them, how to read from them, how to make sure that our program does not throw exceptions when dealing with "I/O" (read: input / output) operations? We will only consider *textual* files, with the `.txt` extension, for now.

Warm-Up: Finding a Correct Path

Each file has a (full, or absolute) *path*, which is a string describing where it is stored: it is made of a directory path and a file (or base) name. Paths are complicated because they can

- Vary with the operating system (windows uses `\` as a path separator, but macOS and Unix systems use `/`),
- Vary with the user (windows store information in the `C:\Users\<username>\` folder, that will change based on your username),

[1] https:/princomp.github.io/code/projects/FileDemo.zip

- Vary with the language (windows calls the "Downloads" folder "Overførsler" in Danish),
- Point to a folder that the current user is not allowed to explore (`/root/` in Unix systems),
- Not exist,
- etc.

A fairly reliable way of handling this diversity is to select the folder `/bin/Debug` naturally present in your solution (it is where the executable is stored). We can access its directory path using:

```
string directoryPath = AppDomain
  .CurrentDomain
  .BaseDirectory;
Console.WriteLine(
  "Directory path is " + directoryPath + "."
);
```

On most Unix systems, this would display at the screen something like

```
~/source/code/projects/FileDemo/FileDemo/bin/Debug/
```

To add to this directory path the file name, we will be using the `Combine` method from the `Path` class, as follows:

```
string filePath = Path.Combine(
  directoryPath,
  "Hello.txt"
);
Console.WriteLine("File path is " + filePath + ".");
```

⚠ Warning

Unless otherwise stated, we will always use this folder in our examples.

Writing Into a File

Writing into a file is done using the `StreamWriter` class and a couple of methods:

- a constructor, that takes a string describing a path as an argument,
- the `WriteLine` method, that write its `string` argument to the file, followed by a new line,
- the `Write` method, that write its `string` argument to the file,
- the `Close` method, that closes the file.

Even if we will not go into details about the role of the `Close` method, it is extremely important and should never be omitted.

As an example, we can create a `HelloWorld.txt` file containing

```
Hello World!!
From the StreamWriter class0123456789
```

using the following code:

```
StreamWriter sw = new StreamWriter(filePath);
sw.WriteLine("Hello World!!");
sw.Write("From the StreamWriter class");
for (int x = 0; x < 10; x++)
{
  sw.Write(x);
}
sw.Close();
```

Reading From a File

Reading from a file is very similar to writing from a file, as we are using a `StreamReader` class and a couple of methods:

- a constructor, that takes a string describing a path as an argument,
- the `ReadLine` method, that read the current line of the file and move the cursor to the next line,
- the `Close` method, that closes the file.

The `Close` method is similarly important and should never be omitted.

As an example, we can open a file located at `filePath` and display its content at the screen using:

```
string line;
StreamReader sr = new StreamReader(filePath);
line = sr.ReadLine();
while (line != null)
{
  Console.WriteLine(line);
  line = sr.ReadLine();
}
sr.Close();
```

What Can Go Wrong?

When manipulating files, *many* things can go wrong.

What if we are trying to read a file that does not exist?

If the `StreamReader` constructor is given a path to a file that does not exist, it will raise an exception. We can catch this exception, but a better

mechanism is to simply warn the user, using the `File.Exists` method that return **`true`** if its string argument points to a file, **`false`** otherwise.

```
if (!File.Exists(filePath))
{
  Console.WriteLine("File does not exist.");
}
```

What if we are trying to write into a file that already exist?

A dual problem is if the path we are using as an argument to the `StreamWriter` constructor points to a file that already exists: by default, that file will be overwritten. An alternative mechanism is to use the overloaded `StreamWriter` constructor that additionally takes a `bool` as argument[2]: if the `bool` is set to **`true`**, then the new content will be *appended* into the existing file instead of overwriting it.

Hence, we can use the following:

```
StreamWriter sw = new StreamWriter(filePath, true);
```

with benefits:

- If the file at `filePath` already exists, we will append to it (add at its end) instead of overwriting it,
- If the file at `filePath` does not exist, it is created.

The previous code

```
StreamWriter sw = new StreamWriter(filePath);
```

should be used only if we do not care about existing files.

Many Things Can Go Wrong!

We will not list in detail all the ways things can go wrong with manipulating files (memory shortage, access right limitations, concurrent access to a file, etc.), but **read and write access to files should always take place in `try{…}catch{…}` blocks**.

[2] https://learn.microsoft.com/en-us/dotnet/api/system.io.streamwriter.-ctor?view=net-8.0#system-io-streamwriter-ctor(system-string-system-boolean)