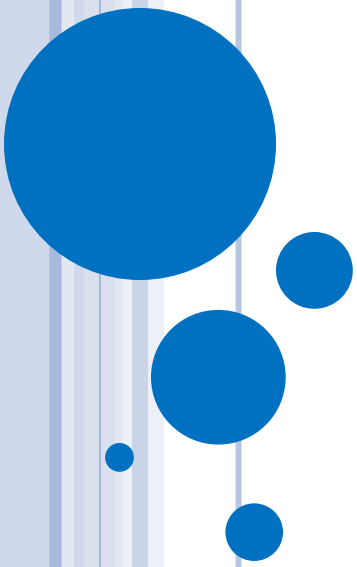
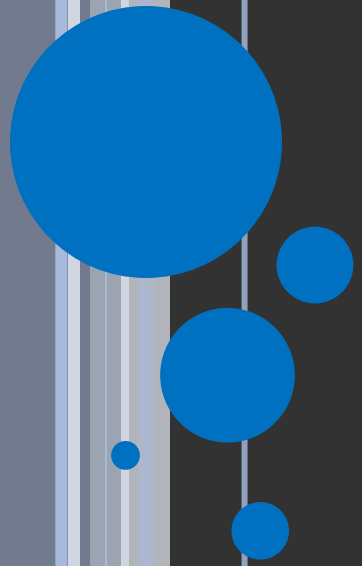


File IO - Text

Why use file input and output?



Why File IO?

Why would anyone choose to use text files?

- Why would we want to store data in files on the hard drive?
- Aren't variables good enough?
- Isn't hard-coding good enough?

Why File IO?

Imagine that every time you turn on your computer, it prompted you to enter a new password. That password works until you restart or shut down.

Every time you log into a website, it asks you to make a new account. You've made 36 new Facebook accounts today.

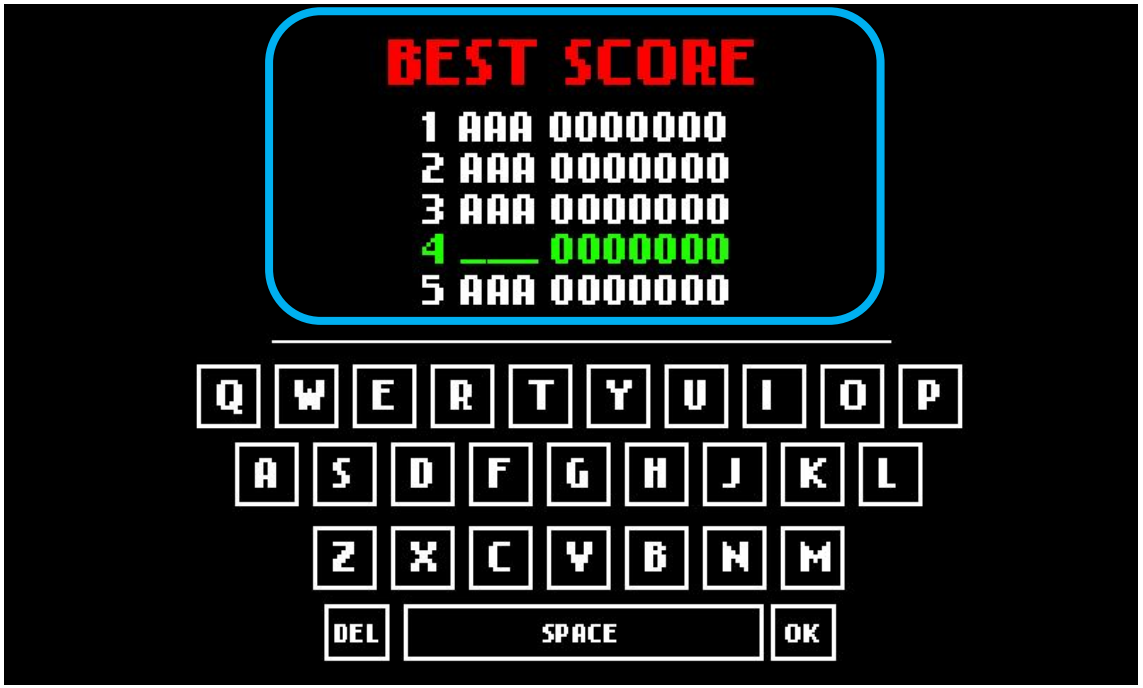
Every time you tried to read old text messages, it asked you to type them all in from memory.

Uh… Oops.

It's cool. It's totally cool.

You just have to enter your own high score.

I totally planned it that way.



Why File IO?

We use file IO because it's a great way to store information!!!

1. Persistence
   - The data remains after the program ends
   - Don't have to hard-code information every. time. you. run. the. program.
   - Examples: Save files (high scores in a game), settings (font size)

2. Allows for easy use of data
   - Store "stuff" we can load into data structures, like lists or dictionaries

Disadvantages of File IO

There are some drawbacks to using files to store data.

1. Variables are fast.
   - Memory (RAM) is fast

2. Traditional hard drives are SLOW in comparison
   - They are mechanical devices
   - You have to wait for a little motor to move a physical piece of metal inside the drive

Inside a Hard Drive

Types of Files

Text-based files

- Contain only human-readable characters
- Can be opened in any text editor
- We *will* work with this

```
The quick brown fox
jumped over the lazy
dog.
```

Binary files

- Contain raw data as 1's and 0's
- Mostly unreadable on their own
- Must be written by a program
- We *won't* be doing any of this

```
00101001  10111001
10010011  10001001
00010011  10011110
```

Are we going to use File IO?

What is File IO?

- **File I**nput/**O**utput

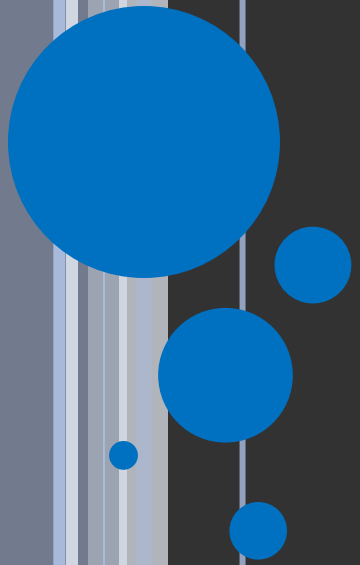
Beginning of semester → Now

- All code comes from you!
  - Type in all variables
  - Hard-code values

Now → End of semester

- Possibility of using text files to get information

Using files with our code



Using Files

Three steps to *reading* data from a file:

1. Open file
2. *Read* data
3. Close file

Three steps for *writing* data to a file:

4. Open file
5. *Write* data
6. Close file

Either read or write.

- Do not do both at once!

Text File IO in C#

Requires System.IO namespace

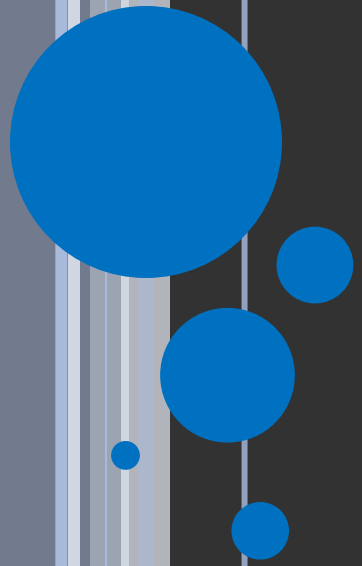
- Add the following at the top of your code:

```
using System.IO;
```

Two built-in classes for interacting with text files

- `StreamReader`
- `StreamWriter`

File Streams



Streams

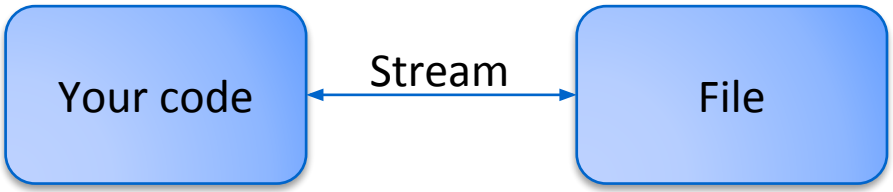
An abstract way of interacting with a sequence of data

Could represent data retrieved from:

- Files on your computer
- Files on another computer
- Data from the internet
- The computer's memory

Many programming languages utilize streams

- Some languages call them "buffers"



StreamReader & StreamWriter

Constructors can take file path parameter

- String representing file to work with

```
StreamReader sr = new StreamReader("filepath.txt");
```

Can be absolute or relative path

- Absolute: C:\Windows\Dropbox\files\calc.exe
- Relative: files\calc.exe

Console Reading versus Stream Reading

Console:

- Only text
- Always ready for use
- No data limit

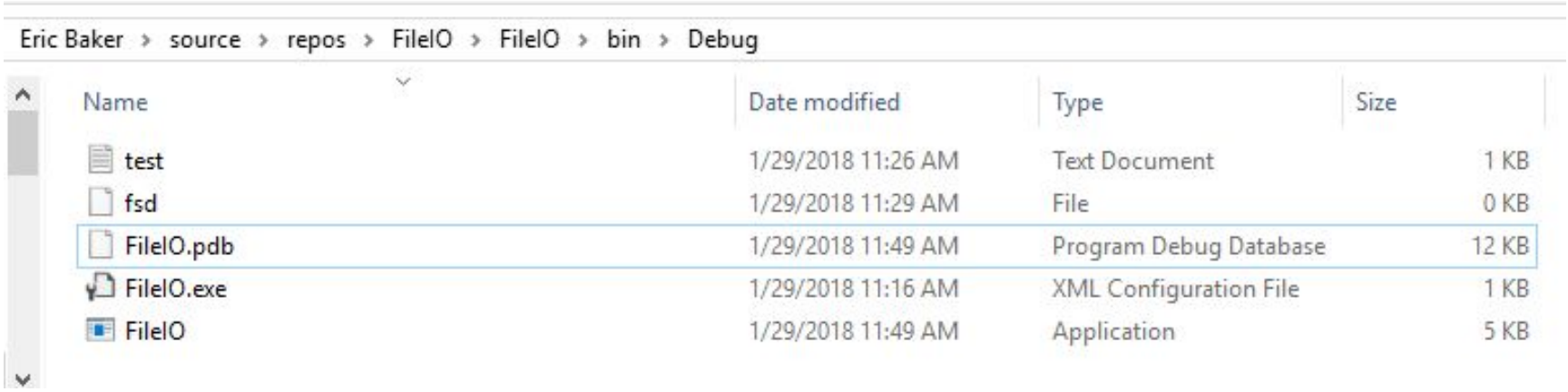
StreamReader

- Only text
- Have to create it
- Has a read limit
- File may not exist
  - If so, cannot open it to read from
- File may be “corrupted”

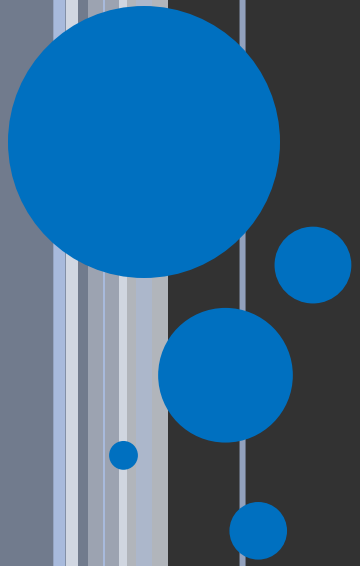
Where do files live?

ProjectFolder/bin/Debug

(where your class files are)



File reading



Basic File Reading Example

```
//***Note: Should use try/catch/finally!

// Create the stream reader
StreamReader input =
            new StreamReader( “filepath/filename.txt”);

// Read a single line from the file
string line = input.ReadLine();

// Close the stream, which closes the file
input.Close();
```

ReadLine() method can be called on a stream.

We’ve been using it with the Console stream.

Step 1: Creating the Reader

```
StreamReader input =
      new StreamReader("filepath/filename.txt");
```

Can throw exceptions!

- File might not exist
- File might be in use
- Might not have permission to read or write to the file

Use try/catch!

Step 2: Reading Data

```
string line = input.ReadLine();
```

Returns the next line of text from the file

- Advances the stream
- Each ReadLine() will "use up" a line from the file

Returns null if there's no more data

How would you read a whole file?

- Call this ReadLine() method in a loop until there are no more lines to read from.

Reading Data in a Loop

Can read a line, store in a variable and check the result of that assignment in one step

```
string line = null;
while((line = input.ReadLine()) != null)
{
    // This loop will continue until it
       reaches the end of file
    // No need to call ReadLine() inside
       here – that happens inside the while
       loop conditional
}
```

Reading Data in a Loop

```
string line = null;

while( (line = input.ReadLine()) != null ){

}
```

Start with an empty string to read into

- Tries to read a line from the data file.
- Returns null is the line was not read (meaning no more lines left).

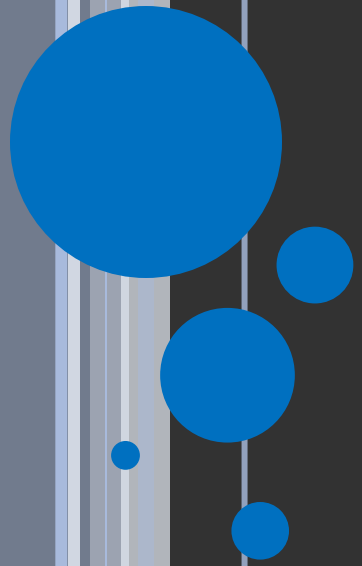
- Compare the line that was read to null.
- Will return true is it wasn't null (meaning there are more lines left).
- Will return false when all lines of text have been read.

Step 3: Closing the file

```
input.Close();
```

- File is considered “in use” until closed
- While open, file can’t be used by another StreamReader, StreamWriter or your program

File Reading and Exceptions



File Reading and Exceptions

If there is an issue with files, C# will throw an exception

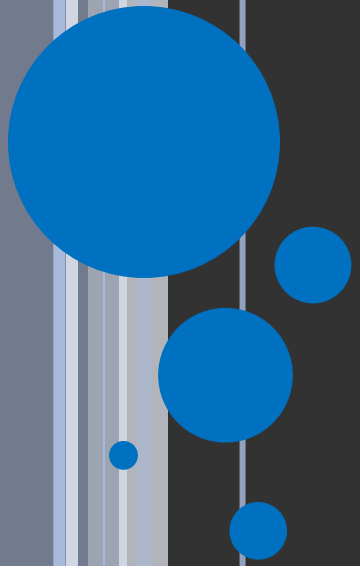
We could:

- Ignore it and our program crashes
- Print an error message and let it crash
- Print a message and work around the crash
- Catch the exception and let the program continue

Handle the exception:

- Enclose code that may break in a try block
- Catch the exception in a catch block

Try
Catch
Finally



Why Finally?

- Sometimes a try will fail
- But or code has stuff it must do
  - return a value
  - release memory
  - close a file
- Finally will ALWAYS run

Finally

```
int num;

try {
    nun = 12 / 0;
}
catch (DivideByZeroException e) {
    Console.WriteLine("Can't divide by ZERO!");
    num = 0;
}
finally {
    return num;
}
```

Finally – a new part of the try/catch block

```
StreamReader input = null;

try{
   input = new Str
   String line = i
}

catch(Exception e)
   Console.WriteLine("Error with file: " + e.Message);
}
```

- Releases the resources from a try/catch block
- Good for closing files
- Code will run regardless if there was an exception thrown or not

```
finally{
   if( input != null )
      input.Close();
}
```

File Reading – Try/Catch Example

```
StreamReader input = null;

try{
   input = new StreamReader("path/file.txt");
   String line = input.ReadLine();
}

catch(Exception e){
   Console.WriteLine("Error with file: " + e.Message);
}

finally{
   if( input != null )
      input.Close();
}
```

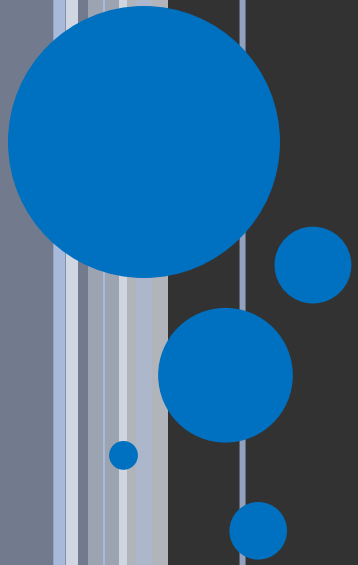
Create stream outside of try/catch

Both lines could throw exceptions

Print error message

If file didn't exist, StreamReader constructor would fail and the file can't be closed.
Must check for NULL down here.

Writing to a file



Basic File Writing Example


```
//***NOTE: Should use try/catch/finally!
//Create the writer
StreamWriter output = new StreamWriter(

"filepath/filename.txt");

//Write some data
output.Write("Hello ");
output.WriteLine("there!");
output.WriteLine("Another line");

//Close the stream, which closes the file.
//NOTE: Data isn't written until stream is closed!
output.Close();
```

Writing Data

Write() & WriteLine() – Similar to Console

- Written to the stream writer object instead of console

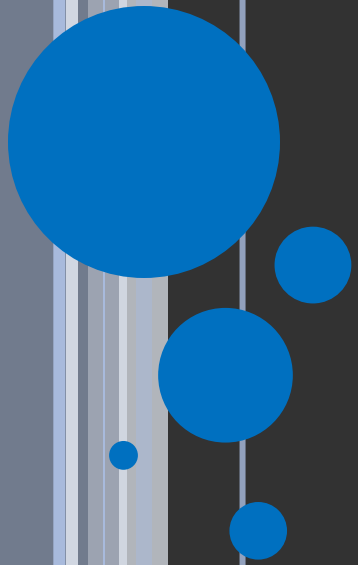
Must close StreamWriter for data to be written

- Data is not written immediately
- Stored in memory until right before file is closed
- Nothing is written if program ends before file is closed

By default, file contents are *overwritten every time*

- Regardless of how much data you're writing

I read data from a text file – what do I do now?



Dealing with Text File Data

Data on a line is often separated by *delimiters*

- Usually commas or pipes

How do we split up that line?

- We read an entire line at a time
- And then use the Split method of the String class!

- Data,goes,here,separated,by,commas,wherever,necessary.,
- Can|also|separate|data|with|“pipes”|
- Sometimes data is not separated by special characters.

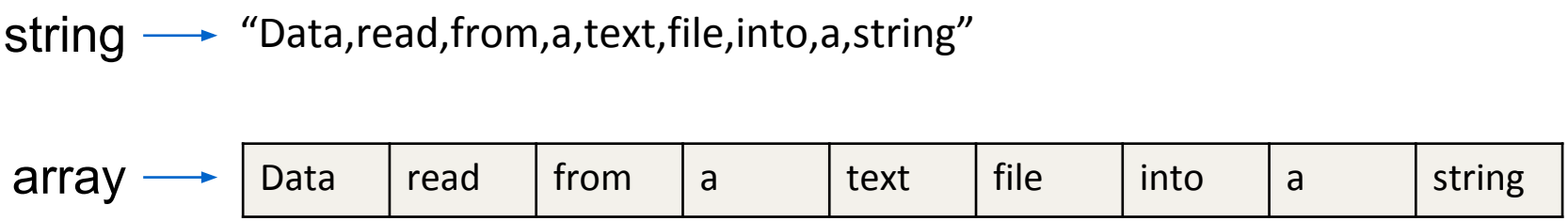
Split() Method

Splits up a string into an *array* of strings

- Returns the array
- Based on a delimiter

Takes a single character as a parameter

- The character to look for when splitting
- That character will be REMOVED from the results

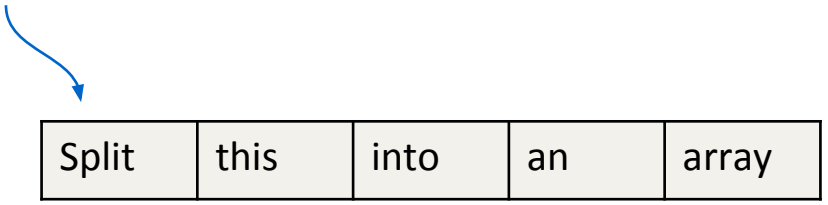


Split() Example

Using a String that is hard-coded with words separated by pipes:

```
// String with words separated by pipes
string sentence = “Split|this|into|an|array”;

// Split the string into an array of strings
// Use a pipe when splitting the string
// Split returns the array it created
string[] data = sentence.Split(‘|’);
```



Another Split() Example

Using a String that holds a line of data read from a file:

```
// Get text from the user
string line = input.ReadLine();

// Split the string using a space
string[] data = line.Split(‘ ’);

// Loop through resulting strings, which will not
contain commas
// Print each word in the array on separate lines
for(int i = 0; i < data.Length; i++){
    Console.WriteLine(“First word: ” + data[i]);
}
```