* 1. 

Hands-On Lab

Module 05: Parsers and Formatters

Lab version: 1.0.0

Last updated: 10/23/2011

Information in this document is subject to change without notice. The example companies, organizations, products, people, and events depicted herein are fictitious. No association with any real company, organization, product, person or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user.

Microsoft may have patents, patent applications, trademarked, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, or other intellectual property.

© 2011 Microsoft Corporation. All rights reserved.

Microsoft, MS, Windows, MSDN, Visual Basic, Visual C++, Visual C#, and Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries.

Other product and company names herein may be the trademarks of their respective owners.

[Creative Commons License](http://creativecommons.org/licenses/by/3.0/)Microsoft Biology Initiative Training by [Microsoft Corporation](http://research.microsoft.com/bio) is licensed under a [Creative Commons Attribution 3.0 Unported License](http://creativecommons.org/licenses/by/3.0/). Based on a work at [research.microsoft.com/bio](http://research.microsoft.com/bio).

[Module 05: Parsers and Formatters 1](#_Toc293908510)

[Loading and Saving data in .NET Bio 4](#_Toc293908511)

[Objectives 4](#_Toc293908512)

[System Requirements 4](#_Toc293908513)

[Task 1: Introducing the .bio Format 5](#_Toc293908514)

[General Formatting 5](#_Toc293908515)

[Comments 5](#_Toc293908516)

[Sequence Data 5](#_Toc293908517)

[Full Example 6](#_Toc293908518)

[Task 2: Building a Sequence Parser 7](#_Toc293908519)

[Task 3: Building a Sequence Formatter 19](#_Toc293908520)

[Task 4: Creating an .NET Bio Add-In 23](#_Toc293908521)

[Summary 27](#_Toc293908522)

Loading and Saving data in .NET Bio

* 1. This Hands-On Lab walks you through building a simple custom parser and formatter for a fabricated biology data format. You will start by building a simple Console-based application to test your parser and formatter and then finally, plug it into .NET Bio and try it out with the sequence viewer/editor you created in Lab #3.
  2. Objectives
  3. In this Hands-On Lab, you will get some experience building an application that uses .NET Bio
  + Create a Sequence Parser using .NET Bio
  + Manage Metadata in the Parser
  + Identify the alphabet and encoding type
  + Support read-only and read/write sequences
  + Build a Sequence Formatter
  + Register your parser and formatter with .NET Bio

# System Requirements

* 1. You must have the following items to complete this lab:
  + Microsoft Visual Studio 2010
  + .NET Bio 1.0 or later
  + Windows XP SP3 or better

Task 1: Introducing the .bio Format

* 1. In this first task you will examine a fabricated bioinformatics file format called “.bio”. It is a simple text-based format that you can store DNA, RNA or Protein sequences in. It supports metadata and comments and multiple sequences of the same type.

## General Formatting

Blank lines are ignored and each line is prefaced with a symbol indicating the type of data contained on that particular line.

## Comments

The bio format supports comments at the top of the file (before any data is present). Each comment is marked with an exclamation point (‘!’). An example of a comment would be:

* 1. ! This is a comment line
  2. !This is also a comment line  
     ! As is this

Comments are only allowed at the top of the file – before data.

## Sequence Data

Each sequence is presented in a formatted block – you can have blank lines around a sequence block or between sections of the sequence block. The sequence block is made up of three sections:

1. Name section
2. Metadata section
3. Data section

The **Name** section is prefaced with an equals sign (‘=’). That is followed by a short identifier for the sequence. There can only be a single name line in the section. An example might be:

* 1. =E.Coli

The **Metadata** section must follow the name, it is optional (that is, it does not need to be present) and each line in the metadata is prefaced with a colon (‘:’). Each line contains two data elements – a key and a value. They are separated with colons, so a sample bit of metadata might be:

* 1. :Date:01/02/1996  
     :FullName:Escherichia coli

Finally, the **Data** section contains the actual sequence items. Each line is prefaced with a vertical bar (‘|’) and is limited to 80 characters. An example might be:

* 1. |----------AAAAAAAAAACCCCCCCCCCUUUUUUUUUUGGGGGGGGGG----------AAAAAAAAAACCCCCCCCC
  2. |CUUUUUUUUUUGGGGGGGGGG----------

## Full Example

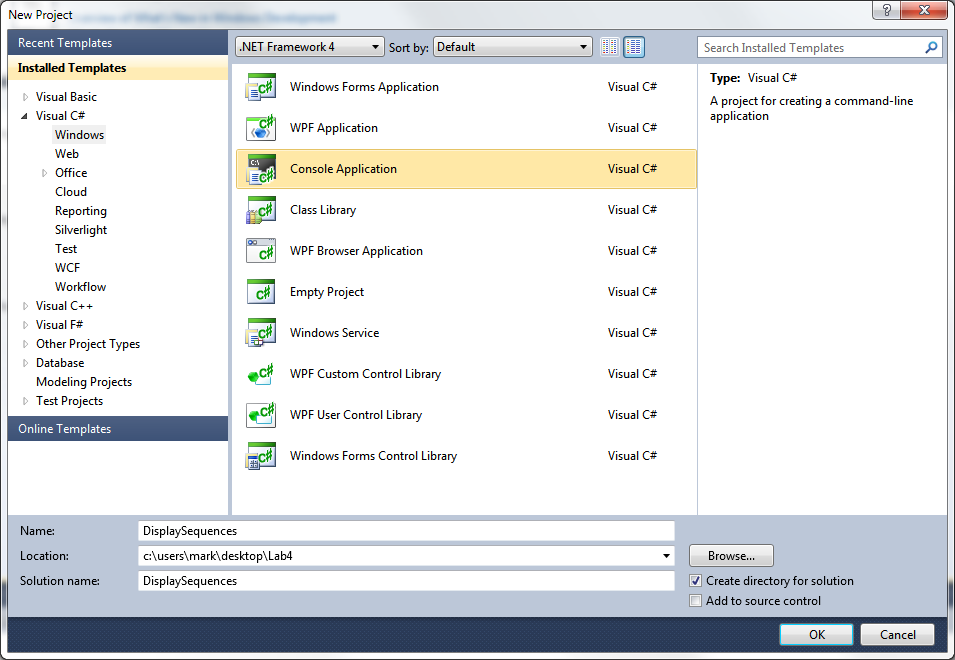
Here is a full example of a .bio data file describing two sequences of data, recognize that this is sample data and not really accurate.

* 1. ! This is a sample data file
  2. ! Comment line
  3. =E.coli
  4. :FullName:Escherichia coli
  5. :Date:01/01/1980
  6. |----------AAAAAAAAAACCCCCCCCCCUUUUUUUUUUGGGGGGGGGG----------AAAAAAAAAACCCCCCCCC
  7. |CUUUUUUUUUUGGGGGGGGGG----------
  8. =CR937012
  9. :FullName:uncultured archaeon
  10. |GC-GGC--CAUA-GC-GG-AGGG--GA-UA-CUCCUGU-ACCCAUCCCGAACACAG-CA-GAU-AAG--CCUUCC--C
  11. |--AC-G-UUCCUUA-C-UGUA-CUAAGG-U-GCGAG-AGC-CCUUGGGA-A-C-UC-UGGAU--C--GCUGCU---A-
  12. |UGCUCAC

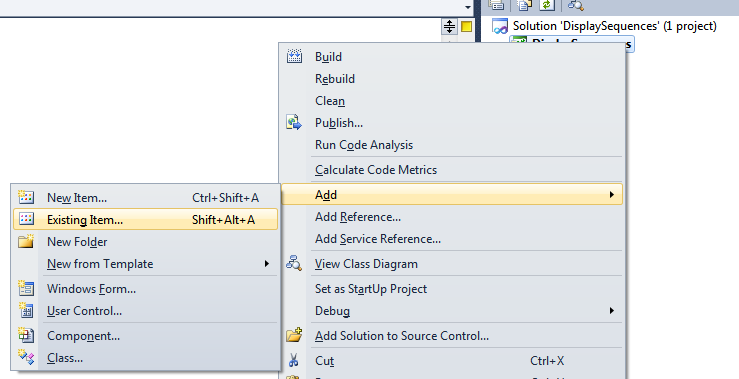
Task 2: Building a Sequence Parser

* 1. We will start by building our sequence parser. The goal will be to take a sample .bio file (shown above) and read it into a **Sequence** object and display the data.

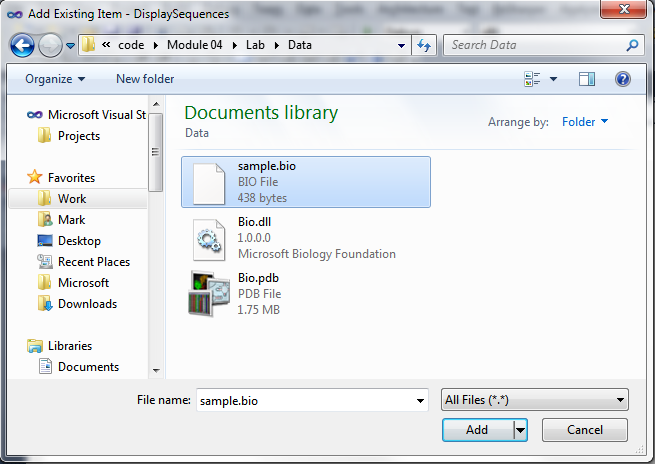
1. Start by creating a new Console-based Windows application.
   1. Name it DisplaySequences



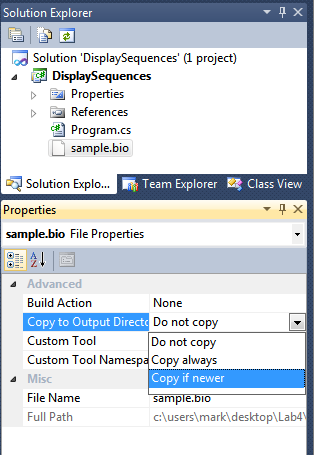
1. Next, in the lab directory included with this manual, there is a Data directory. Inside that directory you will find a sample.bio file. Add it to your project.
   1. Right click on the project, select **Add | Existing Item…**



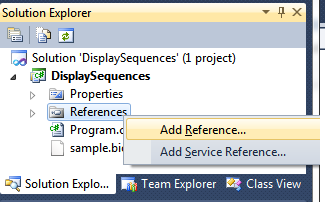
* 1. Navigate to the **Lab4/Data** directory and select **sample.bio**



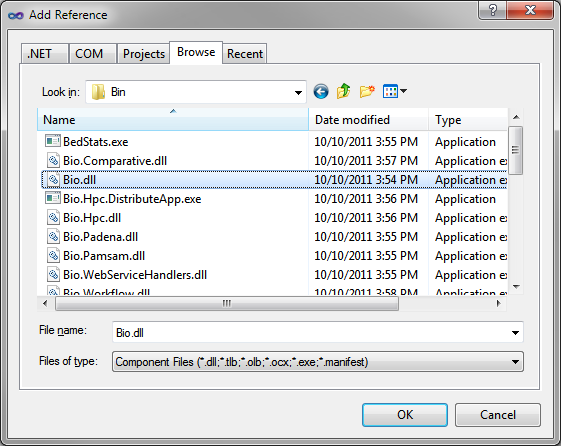
1. Change the properties for this file to have it be copied into our **bin** folder. This way it will be in the current directory when we run the application.
   1. Select the **sample.bio** file in the **Solution Explorer**.
   2. Down in the **Property Explorer**, change the “**Copy to Output Directory**” field to “**Copy if newer**”. The **Build Action** should be “None” – we do not want any compilers trying to process this file!



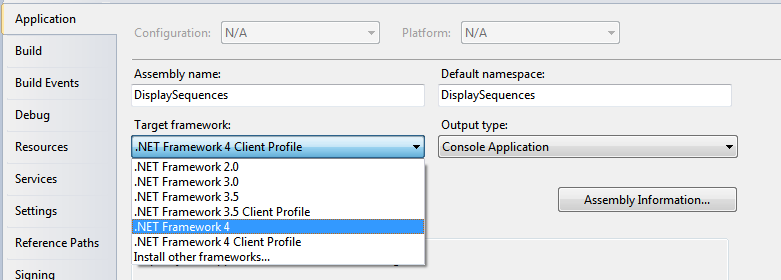
1. Next, we need a reference to the core **.NET Bio** assembly where .NET Bio classes are located.
   1. Right-click on the **References** folder in the **Solution Explorer**.
   2. Select “**Add Reference**”



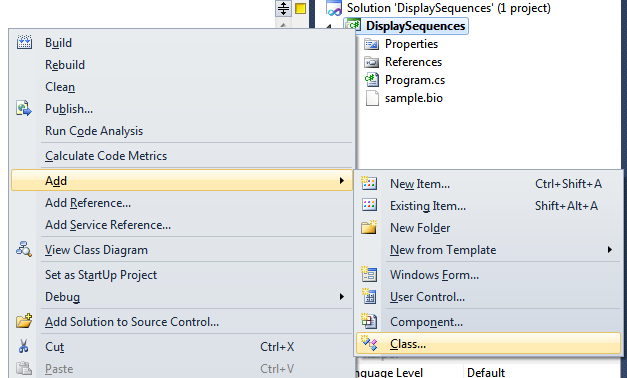
1. In the dialog, select the “**Browse**” tab.
2. Change the directory to the “**.NET Bio**” folder. On a 32-bit machine this will normally be located in the “**C:\Program Files**” and on a 64-bit machine it will be in “**C:\Program Files (x86)**”. Select the version of .NET Bio you want to use by navigating to the next directory (as of this writing, the current version is “1.0”), and then finally select the **Bio.dll** file in the **Tools\Bin** directory. Below, the full path is **“C:\Program Files\.NET Bio\1.0\Tools\Bin”**

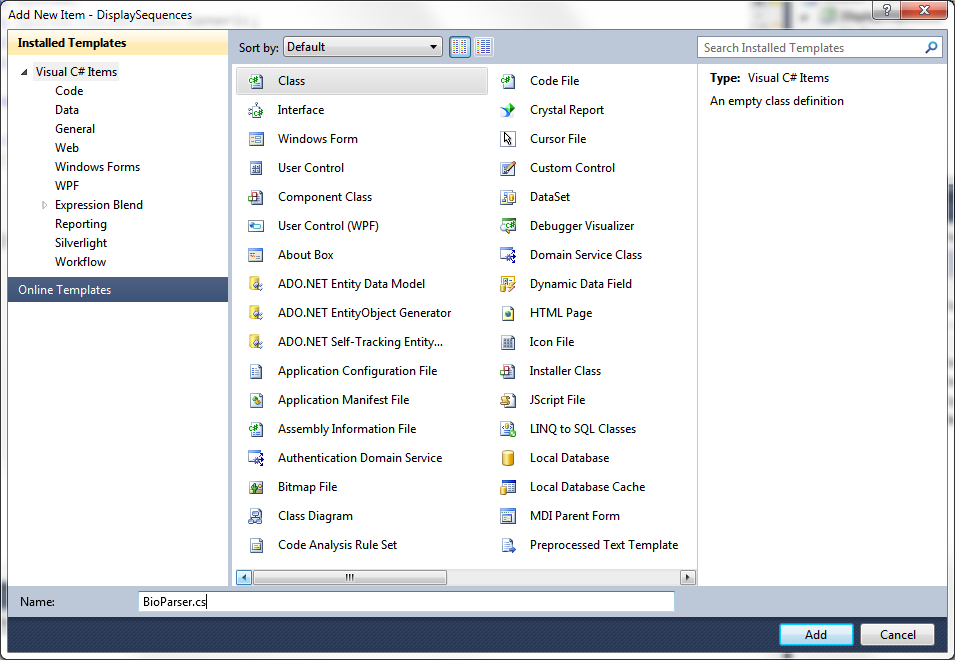


1. Finally, remember that most client projects use the .NET client profile. We need to switch the version of .NET to be the full .NET 4.0 Framework. Double-Click on the **Properties** folder and on the **General** Tab, select the proper .NET version (.NET Framework 4):

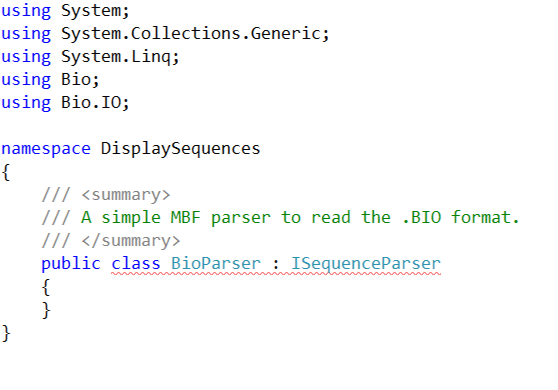


1. Now we are ready to start coding! Create a new C# source file in your project – call it **BioParser.cs**
   1. Right-click on the project, select **Add | Class**

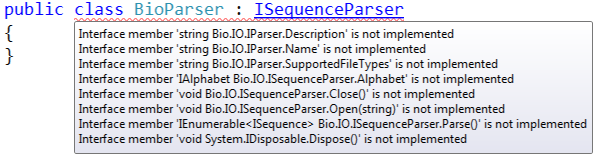




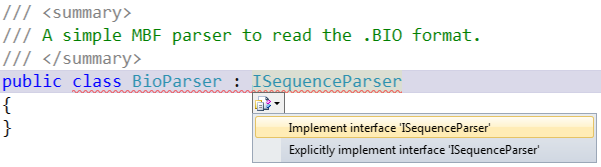
1. Make the class public.
2. Indicate that you want to implement the **ISequenceParser** interface which makes this a parser.
   1. You will need the **Bio.IO** namespace for this.
3. You should notice that Visual Studio will underline the type in a red squiggly line – this is because we have not implemented the members for this interface:



1. If you hover the mouse over the class name, a tooltip will display the problem:

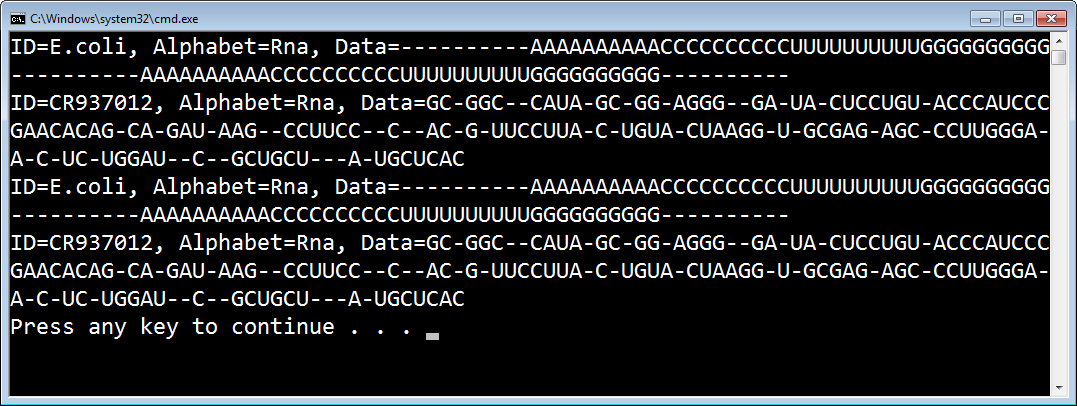


1. If you click on the interface type, a menu box will appear and prompt you to implement the members automatically:



1. Clicking on this choice will add dummy implementations for each missing interface method. This is a great time-saving feature of the editor.
2. Examine the generated members; most of them are fairly easy and you should be able to drop in an implementation yourself
   1. **Name**, **Description** and **SupportedFileTypes** are self-explanatory; return a string for each. **SupportedFileTypes** should return “.bio” to indicate the file extension(s) supported by this parser.
   2. public class BioParser : ISequenceParser
   3. {
   4. public string Description
   5. {
   6. get { return "BioParser Sample - From Lab4 of the .NET Bio course"; }
   7. }
   8. public string SupportedFileTypes
   9. {
   10. get { return ".bio"; }
   11. }
   12. public string Name
   13. {
   14. get { return "BioParser"; }
   15. }
3. The **Alphabet** property can be a simple auto-property, we will add some code later to auto-determine the alphabet.
   1. public IAlphabet Alphabet { get; set; }
4. Add two constructors – one that takes no parameters (a default constructor), and a second that takes a filename. You can simply call **Open**(filename) with the second constructor.
   1. public BioParser()
   2. {
   3. }
   5. public BioParser(string filename)
   6. {
   7. Open(filename);
   8. }
5. The **Dispose** method is used to close the file when you are finished with the parser. It allows the parser to be used with a **using** construct in C# - go ahead and just call **Close()** in that method:
   1. public void Dispose()
   2. {
   3. Close();
   4. }
6. Next, create a field to hold our filename. In the **Open** method, check the input and then set the filename field and verify that the filename exists.
   1. private string \_filename;
   2. public void Open(string filename)
   3. {
   4. if (string.IsNullOrEmpty(filename))
   5. throw new ArgumentException("Missing filename");
   7. if (\_filename != null &&
   8. \_filename != filename)
   9. throw new InvalidOperationException("Parser already open");
   11. if (!File.Exists(filename))
   12. throw new FileNotFoundException("File does not exist.");
   13. \_filename = filename;
   14. }
7. The “meat” of the code will be in a new **ParseOne** method. This method will be used to pull each sequence from the file and returning it to the caller. The input will be a **StreamReader** (something that can read the file line by line). Here is a couple of notes:
   1. The**StreamReader** will read the file one line at a time.
   2. We will want to skip blank lines and lines that start with our comment marker.
   3. There is a **GetMoleculeType** and **IdentifyAlphabet** method included in the **BasicSequenceParser** class you are deriving from. You can call these to dynamically pickup required elements you need to set on the sequence.
   4. You want to create a new **Bio.Sequence** object and set the various properties (**ID**, **DisplayID**, **MoleculeType**, **Alphabet**, **Metadata**) using the information in the .bio file. The completed lab uses the name for the **ID** and looks for a **FullName** metadata key to use for the **DisplayID**. Everything else gets placed into the **Metadata** dictionary.
   5. Make sure to set the **IsReadOnly** flag prior to returning from the method – using the Boolean passed into the method.
8. Next, we will implement the two **Parse** methods – this is really the meat of the parser itself. We have to do the following tasks:
   1. Open the filename using a **StreamReader** – recall that this allows us to read the file line-by-line.
   2. Add a loop, parsing each sequence out of the file and generate a new **ISequence** from it, and then use the **yield return** keyword to return it.
9. Let’s start with the default **Parse** method that takes no parameters. For this form, we assume you have passed a filename and we just need to open a **StreamReader** and pass it along to the second form of **Parse**.
   1. public IEnumerable<ISequence> Parse()
   2. {
   3. return Parse(new StreamReader(\_filename));
   4. }
10. Next, let’s work on the real implementation method. Start by placing the supplied **StreamReader** into a using block – we assume we will be reading the entire stream since this format has no termination marker. If this were not the case, you might decide to **not** close the stream, but require the client perform this action.
    1. public IEnumerable<ISequence> Parse(StreamReader reader)
    2. {
    3. using (reader)
    4. {
    5. }
    6. }
11. Next, we will create a simple method that will use our **StreamReader** and read a non-blank line from it, skipping any comments such as lines that start with ‘!’. We will call it **ReadLine.** It should look something like this:
    1. private static string ReadLine(StreamReader reader)
    2. {
    3. string line;
    5. do
    6. {
    7. line = reader.ReadLine();
    9. } while (line != null
    10. && (string.IsNullOrWhiteSpace(line) || line.StartsWith("!")));
    12. return line;
    13. }
12. Now, back in our **Parse** method, read a line and check it for an error condition such as if it is null, or does not start with our sequence starter marker ‘=’. If either of these tests fail, use the **yield break** keyword to exit the function:
    1. using (reader)
    2. {
    3. string line = ReadLine(reader);
    4. if (line == null || !line.StartsWith("="))
    5. yield break;
13. We should now have the first line of the sequence in the line string. Use a **do/while** loop and at the end of the loop perform the same validity test to exit the loop. We will use this loop to read each sequence:
    1. string line = ReadLine(reader);
    2. if (line == null || !line.StartsWith("="))
    3. yield break;
    5. do
    6. {
    7. }
    8. while (line != null && line.StartsWith("="));
14. Now, in the do loop, we will parse out a single sequence.
    1. The first line has the textual identifier of the sequence. Store the name in an **id** field so we can populate the sequence with it when we create it.
    2. The next few lines *might* be metadata, key/value pairs we need to parse. Read each line and use **String.Split** to split the key/value pairs and store them in a **Dictionary** to add to the sequence when it is created.
    3. Finally, we have all the sequence data itself. We need to read each line and generate a **string** or **byte[]** from it to create the sequence. For this file, you could just append strings together as the data is small. A more performant option (shown below) is to take each line and begin building a **byte[]** from it using **Array.Resize** to make the array large enough to hold the lines read, **Array.Copy** to copy the read line into your buffer and **Encoding.ASCII.GetBytes** to transform a string into a byte[].
    4. If you are feeling adventurous, try doing the above, otherwise, go through the code below which shows an example implementation:
    5. public IEnumerable<ISequence> Parse(StreamReader reader)
    6. {
    7. using (reader)
    8. {
    9. // Read the first non-blank line
    10. string line = ReadLine(reader);
    11. if (line == null || !line.StartsWith("="))
    12. yield break;
    14. do
    15. {
    16. // Get the name of the sequence.
    17. string id = line.Substring(1);
    19. // Look for Metadata
    20. var metadata = new Dictionary<string, string>();
    21. while ((line = ReadLine(reader)) != null)
    22. {
    23. if (line.StartsWith(":"))
    24. {
    25. string[] keyValue = line.Split(new[] {':'});
    26. metadata.Add(keyValue[1], string.Join(":",
    27. keyValue.Skip(2)));
    28. }
    29. else break;
    30. }
    32. // Now read the data.
    33. if (line == null)
    34. yield break;
    36. if (!line.StartsWith("|"))
    37. throw new FormatException("Missing Sequence Data");
    39. int count = 0;
    40. byte[] data = new byte[line.Length-1];
    42. while (line != null && line.StartsWith("|"))
    43. {
    44. int newDataSize = line.Length - 1;
    46. // Not enough space - increase our array size.
    47. if (newDataSize + count > data.Length)
    48. Array.Resize(ref data, newDataSize + count);
    50. // Add the bytes - skip the first byte
    51. Array.Copy(Encoding.ASCII.GetBytes(line),
    52. 1, data, count, newDataSize);
    54. count += newDataSize;
    55. line = ReadLine(reader);
    56. }

    59. // Create the sequence
    60. Sequence sequence = new Sequence(Alphabet, data, false)
    61. {
    62. ID = id
    63. };
    65. // Add the metadata to the sequence
    66. foreach (var kvp in metadata)
    67. sequence.Metadata.Add(kvp.Key, kvp.Value);
    69. // Return it as part of our enumerable.
    70. yield return sequence;
    71. }
    72. while (line != null && line.StartsWith("="));
    73. }
    74. }
15. As a last step, we will identify the alphabet to use – just in case it has not been set on the parser prior to calling **Parse**:
    1. }
    3. if (Alphabet == null)
    4. {
    5. // Try DNA, RNA and then finally Protein.
    6. Alphabet = DnaAlphabet.Instance;
    7. if (!Alphabet.ValidateSequence(data, 0, count))
    8. {
    9. Alphabet = RnaAlphabet.Instance;
    10. if (!Alphabet.ValidateSequence(data, 0, count))
    11. {
    12. Alphabet = ProteinAlphabet.Instance;
    13. if (!Alphabet.ValidateSequence(data, 0, count))
    14. throw new FormatException(
    15. "Failed to identify proper alphabet for symbols.");
    16. }
    17. }
    18. }
    20. // Create the sequence
    21. Sequence sequence = new Sequence(Alphabet, data, false) {ID = id};
16. You can use the following code to test it. Place the code in your Main method. You will need to add **System.Linq** to the **using** list.
    1. using System;
    2. using System.IO;
    3. using System.Linq;
    5. namespace DisplaySequences
    6. {
    7. class Program
    8. {
    9. static void Main(string[] args)
    10. {
    11. // Test the Open/Parse/Close style
    12. using (var parser = new BioParser("sample.bio"))
    13. {
    14. var sequences = parser.Parse();
    15. foreach (var **seq** in sequences)
    16. Console.WriteLine("ID={0}, Alphabet={1}, Data={2}",
    17. **seq**.ID, **seq**.Alphabet.Name,
    18. new string(**seq**.Select(b => (char) b).ToArray()));
    19. }
    21. // Test the new/Parse style
    22. foreach (var **seq** in new BioParser().Parse(
    23. new StreamReader("sample.bio")))
    24. {
    25. Console.WriteLine("ID={0}, Alphabet={1}, Data={2}",
    26. **seq**.ID, **seq**.Alphabet.Name,
    27. new string(**seq**.Select(b => (char)b).ToArray()));
    28. }
    29. }
    30. }
    31. }
17. When you run the application, it should look like:



In the next task, we will build the formatter to go along with this parser! You can see a full completely solution to this task at: [task2/after/DisplaySequences.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp4_MBF.V2.zip\MBF.V2\Module%2004\Lab\Task2\after\DisplaySequences.sln).

Task 3: Building a Sequence Formatter

* 1. In this task, we will complete the parser/formatter pair by adding a Bio formatter to our project. The goal will be to take a set of sequence(s) and then write them out in our .bio format.

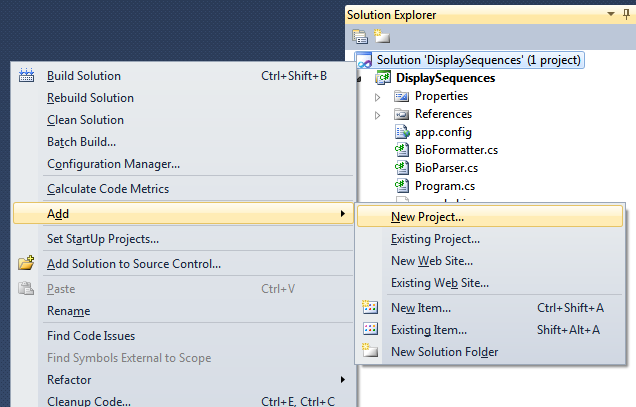
1. You can either continue from Task 2, or open the starter project located at [Task3/before/DisplaySequences.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp4_MBF.V2.zip\MBF.V2\Module%2004\Lab\Task3\before\DisplaySequences.sln).
2. Start by creating a new C# class – name it **BioFormatter.cs** (see step #4 from Task 2 if you need help with this).
3. Make the class public and have it implement **ISequenceFormatter**.
   1. Remember you will need the **Bio.IO** namespace for this.
   2. Go ahead and generate the members using the Visual Studio tip you learned in Task #2 (see steps 7-10 if you need a refresher).
4. Once you have done that, the class should look like:
   1. public class BioFormatter : ISequenceFormatter
   2. {
   3. public string Description
   4. {
   5. get { throw new **NotImplementedException**(); }
   6. }
   7. public string SupportedFileTypes
   8. {
   9. get { throw new **NotImplementedException**(); }
   10. }
   11. public string Name
   12. {
   13. get { throw new **NotImplementedException**(); }
   14. }
   15. public void Open(string filename)
   16. {
   17. throw new **NotImplementedException**();
   18. }
   19. public void Open(StreamWriter outStream)
   20. {
   21. throw new **NotImplementedException**();
   22. }
   23. public void Write(ISequence sequence)
   24. {
   25. throw new **NotImplementedException**();
   26. }
   27. public void Close()
   28. {
   29. throw new **NotImplementedException**();
   30. }
   31. public void Dispose()
   32. {
   33. throw new **NotImplementedException**();
   34. }
   35. }
5. As before, some of the members have obvious implementations. Go ahead and return appropriate strings for each property (**Description**, **SupportedFileTypes** and **Name**).
6. Add two constructors, just like on the parser:
   1. One default, no-parameter constructor
   2. A constructor that takes a string filename. Pass the filename onto the **Open**() method.
7. Implement the **Dispose** method by calling **Close** as before.
8. In the **Open** method which takes a filename:
   1. Test the input filename for the null / empty case and throw an exception if it is not valid.
   2. Call the second Open method which takes a stream writer – pass a newly constructed **StreamWriter** which opens the filename.
   3. public void Open(string filename)
   4. {
   5. if (string.IsNullOrEmpty(filename))
   6. throw new ArgumentException("Missing filename");
   8. Open(new StreamWriter(filename, false));
   9. }
9. In the second **Open** method (which takes the **StreamWriter**)
   1. Assign the passed **StreamWriter** into a field of the class (the lab calls it “\_writer”).
   2. Before the assignment, test the **\_writer** field for a non-null condition – if it already has a value then the user of the formatter already called **Open** without closing it which will be considered an error.
   3. StreamWriter \_writer;
   4. public void Open(StreamWriter outStream)
   5. {
   6. if (\_writer != null)
   7. throw new InvalidOperationException("Formatter already open.");
   9. \_writer = outStream;
   10. }
10. In the **Close** method
    1. Test the **\_writer** for null. If it does not have a value we do not have to do anything.
    2. Call **Dispose** on the **StreamWriter** to close the writer.
    3. Set the field to **null** to indicate we do not have an active stream.
    4. public void Close()
    5. {
    6. if (\_writer != null)
    7. {
    8. \_writer.Dispose();
    9. \_writer = null;
    10. }
    11. }
11. Now, we will focus on the primary method, **Write**. This is called for each sequence to write it out to the open file. You are only responsible for a single sequence in this method. If there are multiple sequences then the user will call this method more than once to write each one out.
12. Our goal in this method is to write out the elements of the Sequence block described in Task 1. You can refer to the documentation there, or even to the **sample.bio** file included in the project to get a sense of what each line should look like.
13. Here are a couple of notes to help:
    1. **StreamWriter** has a **WriteLine** method to write a single CRLF terminated line out, and a **Write** method to write a partial line (it will not be CRLF terminated). You do not need to supply the CRLF.
    2. You should call **StreamWriter.Flush** at the end of all your writes to force the stream to flush to disk.
    3. You need to write out the following items:
       1. **Name (Sequence.ID)**, prefixed with “=”
       2. Metadata (**Sequence.Metadata**, and **Sequence.DisplayID as “FullName”**), each key prefixed with “:”.
       3. Data (the string from the Sequence), remember we want it to be capped at 80 characters per line, with the prefix “|”.
14. See if you can implement the **Write** method given the tips above. Use the following code to test the formatter, and then compare the generated file with the original **sample.bio**:
    1. static void Main(string[] args)
    2. {
    3. // Read them in
    4. using (var parser = new BioParser("sample.bio"))
    5. {
    6. var sequences = parser.Parse().ToList();
    7. foreach (var seq in sequences)
    8. Console.WriteLine("ID={0}, Alphabet={1}, Data={2}",
    9. seq.ID, seq.Alphabet.Name,
    10. new string(seq.Select(b => (char) b).ToArray()));
    12. // Write them out.
    13. using (BioFormatter formatter = new BioFormatter("testout.bio"))
    14. {
    15. foreach (var seq in sequences)
    16. formatter.Write(seq);
    17. }
    18. }
    19. }
15. Here is an example implementation if you need some help:
    1. public void Write(ISequence sequence)
    2. {
    3. if (\_writer == null)
    4. throw new InvalidOperationException("Formatter has not been opened.");
    6. // Write the header (comment)
    7. if (!\_writtenHeader)
    8. {
    9. \_writer.WriteLine("! Created At {0}", DateTime.Now);
    10. \_writtenHeader = true;
    11. }
    13. // Write the name block
    14. \_writer.WriteLine("={0}", sequence.ID);
    16. // Write out all the additional metadata
    17. foreach (var kvp in sequence.Metadata)
    18. \_writer.WriteLine(":{0}:{1}", kvp.Key, kvp.Value);
    20. // Write the data - in 80 character blocks.
    21. string data = new string(sequence.Select(b => (char)b).ToArray());
    22. int pos = 0;
    23. while (pos < data.Length)
    24. {
    25. int len = Math.Min(79, data.Length-pos);
    26. string line = data.Substring(pos, len);
    28. \_writer.Write('|');
    29. \_writer.WriteLine(line);
    31. pos += len;
    32. }
    34. // Add a line separator and flush the writer.
    35. \_writer.WriteLine();
    36. \_writer.Flush();
    37. }
16. As a final step, try adding a comment header to the file. You will need to track whether a sequence has been written out or not, and also reset it when you open a new file with the formatter (if it gets reused). The final solution has this code in place.

In the final step, we will create an add-in with the parser and formatter and then add it to the Sequence Visualizer you built in Lab 3. You can view a completed solution for this task at [Task3/after/DisplaySequences.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp4_MBF.V2.zip\MBF.V2\Module%2004\Lab\Task3\after\DisplaySequences.sln).

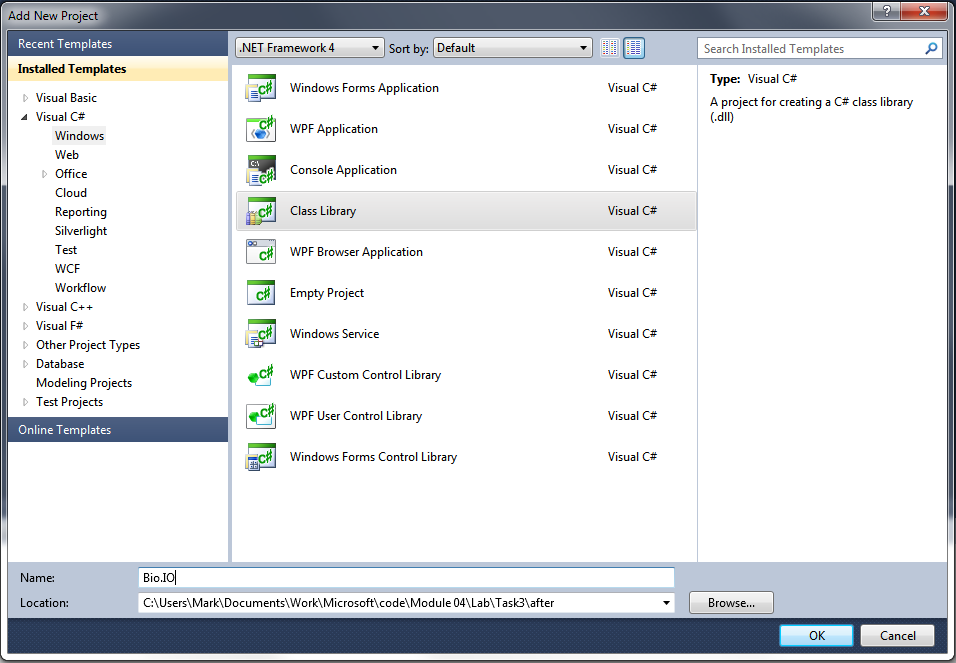
Task 4: Creating an .NET Bio Add-In

* 1. In this final task, we will move your new parser and formatter into an add-in assembly and try it out with the Sequence Viewer you created in Lab 3.

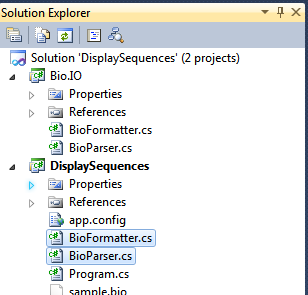
1. You can either continue from Task 3, or open the starter project located at [Task4/before/DisplaySequences.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp4_MBF.V2.zip\MBF.V2\Module%2004\Lab\Task4\before\DisplaySequences.sln).
2. Start by creating a new Class Library project in the solution.
   1. Right Click on the **Solution** item in the **Solution Explorer** and select **Add | New Project**.



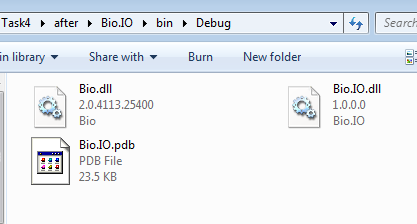
1. Select Class Library and name it **Bio.IO**. Click OK.



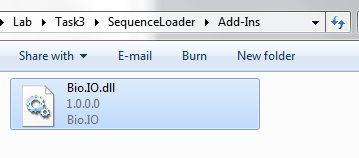
1. Delete the **Class1.cs** file that was included by default. You can just select it in the project explorer and press DEL, or right-click and select **Delete**.
2. Select the **BioParser.cs** file in your main project and while holding the mouse button, drag it into the class library project (specifically, onto the Project item). This will *copy* the source file into the other project.
3. Do the same for the **BioFormatter.cs** file. When you are finished it should look like:



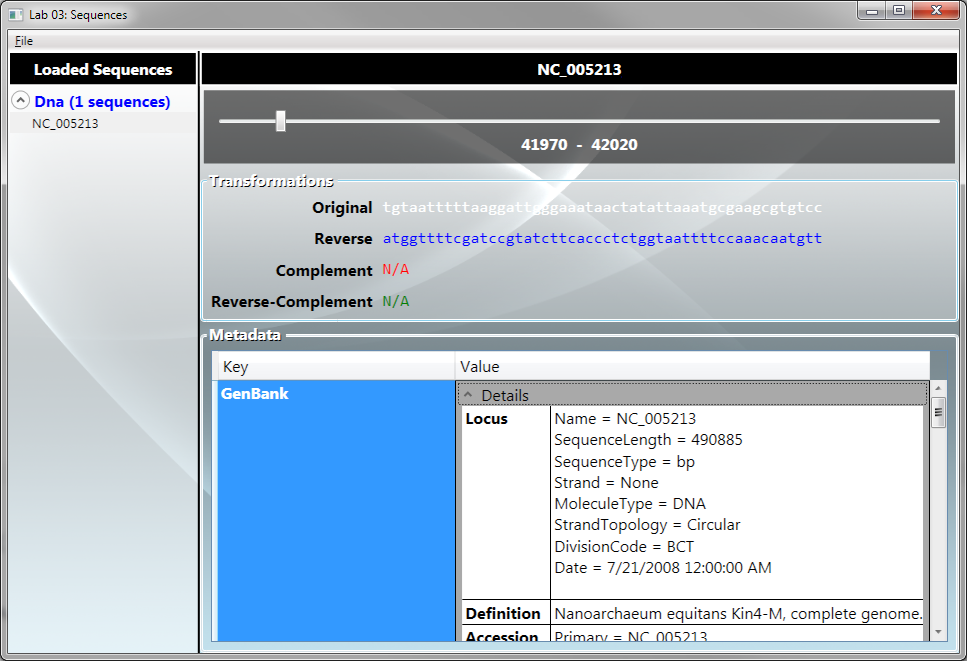
1. Note the source files still exist in the **DisplaySequences** project. Go ahead and delete them.
2. Next, add a reference to the **Bio** assembly to the new class library project. It will need this to compile. It should already be a full .NET Framework assembly so you shouldn’t need to modify that.
3. Try compiling. The main project should fail because the parser and formatter are no longer in the project. To fix this, add a reference to the class library.
   1. Right click on the **References** folder in the main project.
   2. Select the “**Projects**” tab.
   3. Select the Bio.IO assembly listed there and click OK.
4. It should compile and run now.
5. Now, open the **BioParser.cs** source file in the class library.
6. To turn this into a discoverable parser, we need to add a single attribute to the top of the class file – **Bio.Registration.RegisterableAttribute**: (remember the namespace here)
   1. [Registrable(true)]
   2. public class BioParser : ISequenceParser
   3. {
7. Do the same for the **BioFormatter** class.
8. Build the project, now right click on the solution and select “**Open Folder in Windows Explorer**” to get to a command shell. Navigate to the **Bio.IO/bin/debug** folder (assuming you built a debug build). You want to find a built version of the **Bio.IO.dll** assembly.



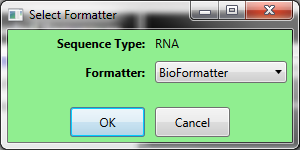
1. Copy the file by selecting it and pressing CTRL+C, or right-clicking on it and selecting **Copy**.
2. Back up to the Task 4 folder. There is a **SequenceLoader** folder located here containing a pre-built version of the previous lab (essentially the final solution). It has not been modified from the final solution you used. We are going to add the new parser and formatter to the existing application without changing its source code!
3. In here, create a folder called “**Add-Ins**” and place your Bio.IO.dll into this subdirectory (press CTRL+V or right-click and select **Paste**).



1. Back up a directory and run the Sequence Loader application. You should be able to open a .bio file using the **File** | **Open** dialog. You should see the Metadata, and be able to edit it.



1. Finally, try saving it. It should show your formatter in the list of selections:



1. If you open the saved file you should see your altered data with one sequence:
   1. =E.coli
   2. :FullName:Escherichia coli
   3. :Date:01/01/1980
   4. :NewData:
   5. |----------AAAAAAAAAACCCCCCCCCCUUUUUUUUUUGGGGGGGGGG----------AAAAAAAAAACCCCCCCCC
   6. |CUUUUUUUUUUGGGGGGGGGG----------

Summary

* 1. In this lab you have created a new .NET Bio parser and formatter using a file specification described in this document. You have used it to load and save sequences, and then modified it to be supplied as an add-in to .NET Bio and used it on an existing application without recompiling it.
  2. You can get a final solution from [Task4/after/DisplaySequences.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp4_MBF.V2.zip\MBF.V2\Module%2004\Lab\Task4\after\DisplaySequences.sln). You can also try the completed sample using the [Task4/after/SequenceLoader/SequenceLoader.exe](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp4_MBF.V2.zip\MBF.V2\Module%2004\Lab\Task4\SequenceLoader\SequenceLoader.exe).