* 1. 

Hands-On Lab

Module 03: Programming with .NET Bio

Lab version: 1.0.0

Last updated: 10/14/2011Information 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 03: Programming with .NET Bio 1](#_Toc306367395)

[Programming with .NET Bio 4](#_Toc306367396)

[Objectives 4](#_Toc306367397)

[System Requirements 5](#_Toc306367398)

[Task 1: Creating a WinForms Application 6](#_Toc306367399)

[Task 2: Reading Sequences with .NET Bio 13](#_Toc306367400)

[Task 3: Running Algorithms 19](#_Toc306367401)

[Task 4: Saving Sequences 20](#_Toc306367402)

Programming with .NET Bio

* 1. This hands-on lab introduces you to the basics of programming with the .NET Bio framework by having you build a simple Windows Forms application that can load a set of sequence(s) from a file, transcribe them and then write them back out to the same or different file.
  2. Through this exercise you will get some experience working with sequences, parsers, formatters and the transcription algorithm that is supplied in .NET Bio.
  3. The final application you will build looks like:
  4. 
  5. Objectives
  6. In this Hands-On Lab, you will get some experience building an application that uses .NET Bio
  + Create a Windows Forms application
  + Use .NET Bio parsers to read sequence files from disk in any supported format.
  + Display sequences as string text.
  + Transcribe sequences DNA 🡪 RNA and RNA 🡪 DNA.
  + Save transcribed sequences back to disk in any supported format.

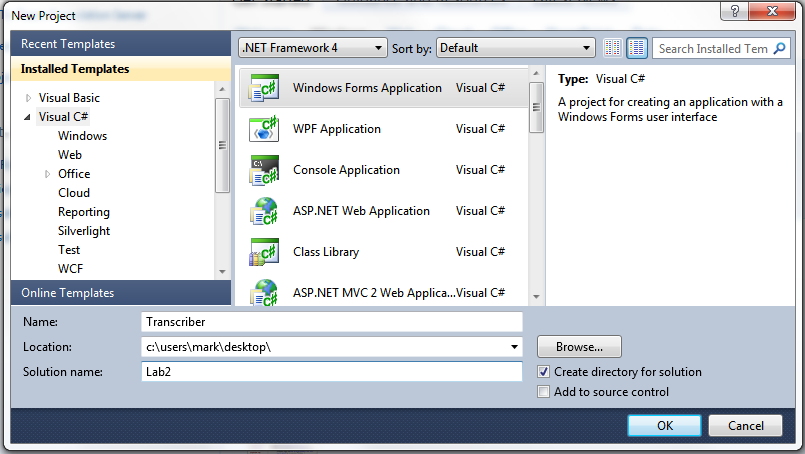
# System Requirements

* 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

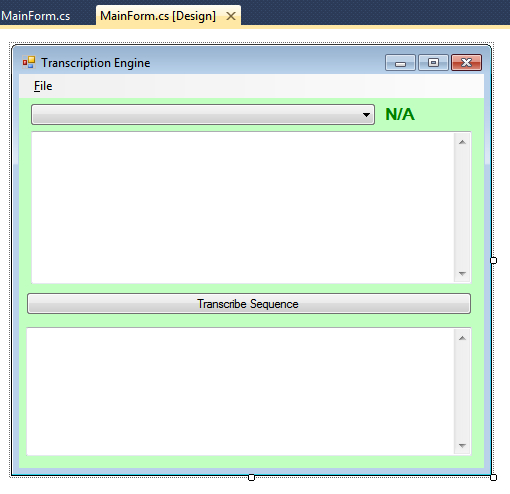
Task 1: Creating a WinForms Application

* 1. In this first task you will use Visual Studio 2010 to build a Windows Forms based application to display our .NET Bio data. If you would prefer to skip the UI generation, you can move onto Task 2 where we begin to add the .NET Bio support code.

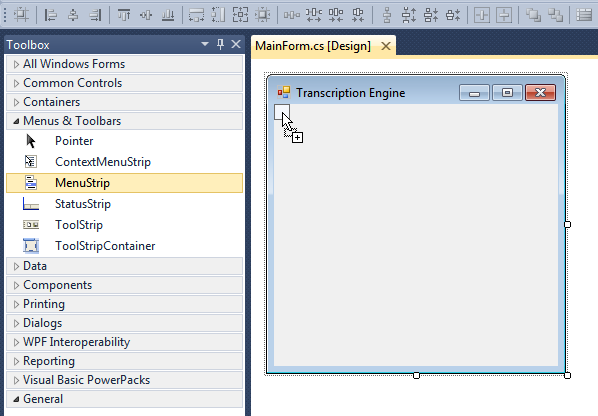
1. To begin the exercise, launch Visual Studio 2010 – it should be in **your Start Menu 🡪 All Programs 🡪 Microsoft Visual Studio 2010**.
   1. If it prompts you for developer settings (it does this the first time you open the application), make sure to choose “General Development Settings”.
2. Start by creating a new project – **File 🡪 New 🡪 Project…**
3. In the **New Project** Dialog:
   1. Locate the **Visual C#** section in the **Installed Templates** section.
   2. Make sure .NET Framework 4 is selected in the top combo box.
   3. Select **Windows Forms Application** from the list of templates.
   4. Name the application “Transcriber”
   5. Place it into a directory you have access to (the **Desktop** is a good choice).
   6. Name the solution “Lab2”
   7. Check the “Create Directory for solution” box – this will create a sub-directory for your project under the solution directory.
   8. Here is what your screen should look like:



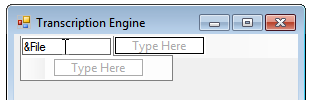
1. Click **OK** to create the project.
2. Visual Studio will create the project and open the **Form1** designer file.
3. As a first step, change the **Text** property of the form to “Transcription Engine”. You can do this by locating the **Text** property in the Property explorer.
4. We will also rename the form to be something other than “Form1”:
   1. Right-Click on the **Form1.cs** entry in the Solution Explorer and select the “Rename” option give it a different name – the lab will use “MainForm.cs”.
   2. Make sure to keep the “.cs” extension – Visual Studio will warn you if you leave this off – the file will not be buildable if you do this.
   3. As a future note, you can also click on the item in the **Solution Explorer** and hit “F2” to rename the file.
5. Now, let’s begin crafting the UI – here is ultimately what we want:



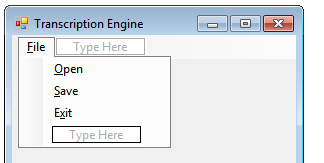
1. The first thing to add is the menu bar at the top. Open the **Tool Box** (if it isn’t already open) by clicking the **View** menu and selecting **Toolbox** (or CTRL+ALT+X).
2. Locate the **Menus & Toolbars** section and expand it.
3. Locate the **MenuStrip** control – click and drag one onto your form and position it at the top of the window:



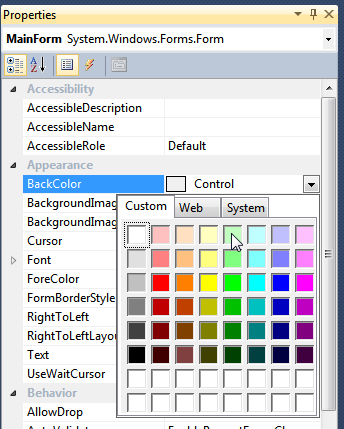
1. Now, add a File menu – click on the “Type Here” text and enter “&File”.
   1. The ampersand (&) that prefaces the text indicates we want the next letter (the “F” in this case) to be the accelerator for this menu item. So if we hold the **ALT** key and press “F” we want this menu item to drop down. Here is an example of what is happening:



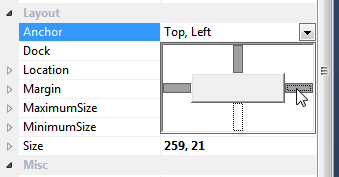
1. Next, under the File choice, add three other items:
   1. &Open
   2. &Save
   3. &Exit
2. When you are finished, it should look something like this:



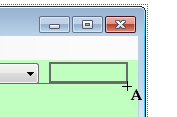
1. Next, click on the window itself and we will change the background color. The lab will use a green color, but feel free to choose anything you like.
   1. The **MainForm** object should be selected in properties (if not, you can use the ComboBox at the top of the **Property Explorer** to pick the proper visual element you want to work with).
   2. Locate the **BackColor** property and click it to get the color list. Change to the **Custom** tab and select a color. The background of the form should change accordingly.



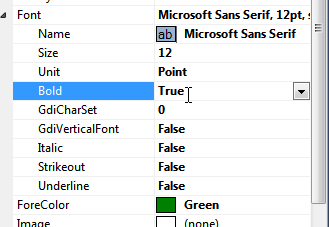
1. Next, add a **ComboBox** to the form by dragging it out of the **ToolBox**.
   1. Position it at the top of the form.
   2. Name it “cbSequences” by changing the **Name** property in the **Property Explorer**.
   3. Set the **DropDownStyle** to be “DropDownList” in the **Property Explorer**. This will allow the user to select an item from the list, but not edit the list directly.
   4. Set the **Anchor** property to dock it on the left, right and top. This property is towards the bottom of the **Property Explorer** – in the layout section. Here is what it will look like:



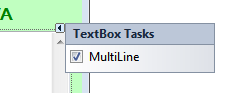
1. When you are finished, you should be able to resize the form in the designer and the **ComboBox** should resize with it.
2. Next, add a **Label** to the form – position it just to the right of the **ComboBox** (you may need to make the **ComboBox** width a little smaller).



1. Make the following property changes on the new label:
   1. Name it “sequenceTypeText”.
   2. Set the **Text** property to “N/A”
   3. Change the **ForeColor** property to some color – the lab will use “Green”.
   4. Set the **Anchor** property to be Top/Right.
   5. Change the **Font Size** to be “12” and the **Font Bold** to be “true”. You will need to expand the Font property in **Property Explorer** to make this change.



1. Next, drag a **TextBox** under the **ComboBox**. As seen in the “here’s what we want” picture, we would like it to take up a reasonable amount of space. However, the designer will not let you resize it beyond a single line. The reason is that, by default, the **TextBox** is always single-line, in order to take up multiple lines; you have to set the **MultiLine** property to **true**. You can do this in the **Property Explorer**, or you can click the little arrow on the right side of the **TextBox** in the designer which opens the “Tasks” menu and set it there:

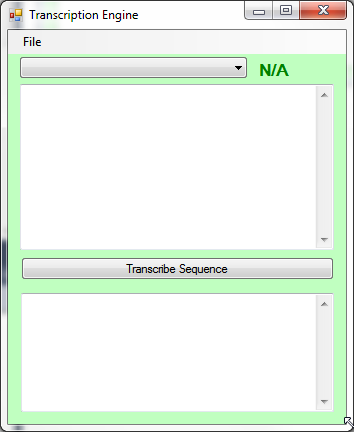


* 1. Set the **Name** to “sourceSequenceText”.
  2. Set the **ReadOnly** property to True.
  3. Set the **BackColor** property to White.
  4. Set the **ScrollBars** to Vertical.
  5. Set the **ForeColor** property to Blue.
  6. Set the **Font** Size property to “12”.
  7. Set the **Anchor** to be Top/Left/Right.

1. Drag a **Button** underneath the TextBox.
   1. Anchor it to the Top/Left/Right.
   2. Set the **Text** to be “Transcribe Sequence”
2. Finally, drag a second **TextBox** under the button.
   1. Set the **Name** to be “targetSequenceText”.
   2. Set the **ForeColor** property to be Red.
   3. Set the **Anchor** to be Top/Bottom/Left/Right.
   4. Set all the other properties to match Step 20 (MultiLine, Font, ReadOnly, etc.)
3. As a last step, click on the **File** menu item, and then double-click on the **Exit** child item. This will cause Visual Studio to generate an event handler that will be called when the user selects this menu item – it will be added to the code behind file and Visual Studio should switch you over to the new function called **exitToolStripMenuItem\_Click**.
   1. In the function invoke the **Close()** method – it is a method of this class and it will close the main window, and terminate the application.
4. Here is the new function if you need help:
   * 1. private void exitToolStripMenuItem\_Click(object sender, EventArgs e)
     2. {
     3. this.Close();

}

1. Build and Run the application by pressing F5 (or CTRL+F5 to run outside the debugger). It should look like:



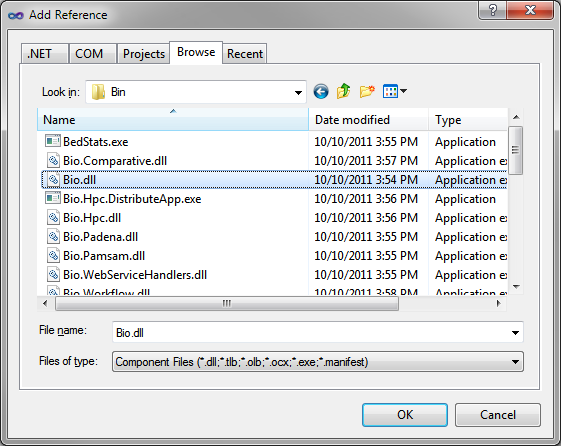
* 1. You should be able to resize the window and all the controls should move around properly.
  2. You should be able to select **File | Exit** and have the application close.

If everything looks good then this step is complete; move onto the next task to begin adding .NET Bio functionality!

Task 2: Reading Sequences with .NET Bio

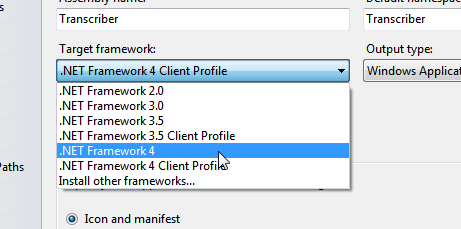
* 1. In this task you will add a reference to .NET Bio to the Windows Forms project and use it to load sequences from files and display the data in the **TextBox** and **ComboBox** controls you’ve created.

1. To start off with, either continue from Task 1, or open the starter project located at [Task2\Before\Lab2.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp3_MBF.V2.zip\MBF.V2\Module%2002\Lab\Task2\Before\Lab2.sln).
2. The first step in using .NET Bio is to add a reference to the **Bio.dll** assembly.
   1. Right-click on the **References** folder and select “Add Reference”.
   2. In the dialog, select the “**Browse**” tab.
   3. 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”**



We have one minor project property setting to set to in order to use .NET Bio. There are actually two versions of .NET 4.0 available – the full desktop edition, and the smaller client-profile edition. The Client Profile version is intended specifically for desktop-based Windows applications – it is a reduced feature set version that leaves out thing such as web server (ASP.NET) support, WCF service support, etc. By default, Visual Studio sets WPF and Windows Forms applications to use this version and normally it would be fine. Unfortunately, at this time .NET Bio is not compiled to target this reduced set, so we need to convert the project to use the full desktop .NET version.

1. Double-click on the **Properties** folder in the **Solution Explorer** (or right-click on the solution and select Properties).
2. With the **Application** tab selected, locate the **Target framework** option – it should say “.NET Framework 4.0 Client Profile”. Pull this down and select “.NET Framework 4.0”. Visual Studio will prompt you to reload the project. Reload the project.



1. Once it is done reloading, open the **MainForm** designer view.
2. Click on the **File** menu item – and then double-click on the “**Open**” child item.

Visual Studio should change to the code-behind file for this form and it should have created a new method called **openToolStripMenuItem\_Click**. This is the method that is called when the user selects the **File | Open** menu option. Here, we will prompt the user for a file and then load it using .NET Bio.

* 1. **Prompting for files:** Windows ships with a set of standard dialogs – **Open**, **Save**, and **Print** being the three most commonly used. .NET exposes these dialogs through three classes: **OpenFileDialog**, **SaveFileDialog** and **PrintFileDialog**. Each of these classes has a set of properties to control things like the displayed title, what directory to start in, what files are acceptable, etc.

1. In the method, create a new **OpenFileDialog** instance and assign it the name “ofd”. We want to make sure the file exists and to always reset the directory after the user selects the file so set the following properties on the instance variable:
   1. Set the **Title** property to “Select Sequence File”
   2. Set the **Filter** property to “All Files|\*.\*”
   3. Set the **RestoreDirectory** property to true.
   4. Set the **CheckFileExists** property to true.
2. Call **ShowDialog** on the **OpenFileDialog** instance to display the prompt. It returns a **DialogResult** type. This is an enumeration (values like **OK**, **Cancel**, **Yes**, **No**, etc.).
3. If the return result is **DialogResult.OK**, then
   1. Get the selected filename from the **FileName** property of the **OpenFileDialog**.
   2. Use .NET Bio’s **SequenceParsers** class to determine the parser type. Recall that this class has properties exposing each of the known parser types, as well as a static method to identify the proper parser from a filename using the extension of the file.
      1. You will need to add the **Bio.IO** namespace to use this class.
      2. Make sure to test for error conditions.
   3. If a valid parser can be found, use it to parse the file.
      * + 1. You do not need to open the parser as it was created with the **FindParserByFileName** method which passed in the filename to the constructor. If you *do* call **Open**, it will likely throw an exception as the filename is already filled in!

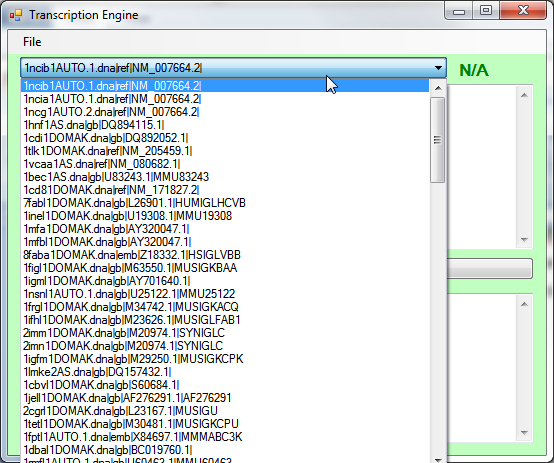
You should get back an **IEnumerable <ISequence>**. Go ahead and use **ToList()** on the returning enumerable to get a list of the sequences. In order to do this, you need a **using System.Linq** at the top of the source file – the **LINQ** namespace is where the **ToList** extension method is located.

* 1. Save the resulting list of sequences in a private field of the class (the lab will name it \_**sequences**).
  2. Since you have loaded the sequences, you can close the parser once the **Parse** is complete.
  3. Make sure to catch any exceptions that occur during the parse operation – report all errors using **MessageBox.Show**. It pops up a message box and displays whatever text you pass it.

1. For a little additional guidance compare your code to how the code should look as shown below:
   * 1. IList<ISequence> \_sequences;
     2. private void openToolStripMenuItem\_Click(object sender, EventArgs e)
     3. {
     4. OpenFileDialog ofd = new OpenFileDialog();
     5. ofd.Title = "Select Sequence File";
     6. ofd.Filter = "All Files|\*.\*";
     7. ofd.RestoreDirectory = true;
     8. ofd.CheckFileExists = true;
     9. if (ofd.ShowDialog() == DialogResult.OK)
     10. {
     11. var filename = ofd.FileName;
     12. var parser = SequenceParsers.FindParserByFileName(filename);
     13. if (parser == null)
     14. {
     15. MessageBox.Show("No parser available for file.",
     16. "Unable to load file");
     17. return;
     18. }
     19. try
     20. {
     21. \_sequences = parser.Parse().ToList();
     22. \_parser.Close();
     23. }
     24. catch (Exception ex)
     25. {
     26. \_sequences = null;
     27. MessageBox.Show(ex.Message, "Failed to parse " +
     28. Path.GetFileName(filename));
     29. }
     30. }
     31. }
2. Compile and run the application. It should now allow you to select a file, even though it does not do anything with it yet.
3. Close the program and switch back to the code view. If you are in the designer view, you can press F7 to switch to the code view.
4. Create a new method in the code behind. Its purpose will be to populate the UI with the loaded sequence data.
   1. Name the method **LoadSequences**. It does not need any parameters since we have the sequence list stored in a field.
   2. Place a call to this new method in your **File|Open** handler – at the end is fine.
5. In the new **LoadSequences** method; first reset the **Sequence ComboBox** by clearing its **Items** collection and setting the selected index property to “-1”. This will remove all the existing items (if there are any from a previous load) and ensure there are no selected items.
   * 1. private void LoadSequences()
     2. {
     3. cbSequences.Items.Clear();
     4. cbSequences.SelectedIndex = -1;

}

1. Next, if we have a set of loaded sequences (\_**sequences** is not null), we want to populate the ComboBox with them. Using a **foreach** loop add each sequence into the ComboBox – use the **ISequence.ID** property to get a printable text name. After it is populated, set the selected index to the first item (0).
2. If you need a little help, here is the lab example:
   * 1. private void LoadSequences()
     2. {
     3. cbSequences.Items.Clear();
     4. cbSequences.SelectedIndex = -1;
     5. if (\_sequences != null)
     6. {
     7. foreach (var seq in \_sequences)
     8. cbSequences.Items.Add(seq.ID);
     9. cbSequences.SelectedIndex = 0;
     10. }
     11. }
3. Run the program and see how well it works. There are some sample data files in the [Data](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp3_MBF.V2.zip\MBF.V2\Module%2002\Lab\Data) folder in both FASTA and GenBank format. If you select one of them and pull down the ComboBox drop down, it should display one or more sequences.



1. Close the program and switch back to the **MainForm** designer view by double-clicking on the MainForm.cs entry in the solution explorer.
2. Double-click on the **ComboBox** – this will wire up the default event handler for the **ComboBox** which occurs when you select a new item (or a new item is programmatically set). Visual Studio should switch you into the code-behind view in a new method it has created called **cbSequences\_SelectedIndexChanged**.
3. In this method we want to do several things:
   1. First, determine if a new sequence is selected or not. We can determine this by looking at the **cbSequences.SelectedIndex** property – it will be >= 0 if an item is selected and “-1” if not selected. This will be the index into our \_**sequences** collection of the selected sequence.
   2. Next, if a sequence is selected, we want to populate the sequence type **Label** [**sequenceTypeText**] that sits right next to the ComboBox (it has the default text “N/A” in it now), with the type of sequence we have selected – this should be DNA, RNA, or Protein. The easiest way to tell is to look at the assigned **Alphabet** of the sequence – it should be assigned to the alphabet type allowed for sequence items contained within the sequence. We can set the name of the alphabet on the **Text** property of the **Label**.
   3. Also, we want to take the sequence items from the sequence (the symbols) and put them into the first TextBox [**sourceSequenceText**] (just below the ComboBox) so you can see the data. This is a bit tricky as there is no conversion method to go from a sequence to a string, and not only that – but the data might be very large so rendering it all can be prohibitive.
      1. Use a LINQ expression to select the first 255 values from the sequence, cast the byte to a char and create a new string from it. If you need help, see the code fragment below.
   4. If no sequence is selected, then we want to reset the **Label** and **TextBox** text.
4. If you need a little guidance for the above tasks, you can use the following code:
   * 1. private void cbSequences\_SelectedIndexChanged(object sender, EventArgs e)
     2. {
     3. if (cbSequences.SelectedIndex >= 0)
     4. {
     5. ISequence seq = \_sequences[cbSequences.SelectedIndex];
     6. sequenceTypeText.Text = seq.Alphabet.Name;
     7. sourceSequenceText.Text = new string(
     8. seq.Select(b => (char) b)
     9. .Take(255)
     10. .ToArray());
     11. }
     12. else
     13. {
     14. sequenceTypeText.Text = "N/A";
     15. sourceSequenceText.Text = string.Empty;
     16. }

}

That completes our reading of the sequences and displaying the data. If you are up for a challenge, here is an additional step for you:

See if you can add support to select the parser to use if the auto-detection fails. This would involve displaying an additional dialog, populating a list of parsers and allowing the user to select one of the parsers to return. Here are some Windows Forms tips:

* To display a new form, add a new Form to the project (right-click on the project and select “**Add New Item – select Windows Form**”. Then instantiate a copy of that new Form type in your parser selection code and call **ShowDialog** on it to display it as a window on the screen. It will return when the dialog is closed – presumably when the user clicks **OK** or **Cancel**.
* You can retrieve all the available parsers using the **SequenceParsers.All** static property.
* Add a public property to your dialog to return the selected parser in – when the **ShowDialog** call returns, the instance will have whatever value the dialog has set.
* To force a failure, just change the extension on one of the existing data files – that will cause .NET Bio to not recognize the format and your new code should kick in to allow manual selection!

You can examine the after project at [Task2\After\Lab2.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp3_MBF.V2.zip\MBF.V2\Module%2002\Lab\Task2\After\Lab2.sln) to see an example of this dialog if you would just like to see it in action.

Task 3: Running Algorithms

* 1. In this task you will add a reference to .NET Bio to the Windows Forms project and use it to load sequences from files and display the data in the **TextBox** and **ComboBox** controls you’ve created.

1. To start off with, either continue from Task 2, or open the starter project located at [Task3\Before\Lab2.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp3_MBF.V2.zip\MBF.V2\Module%2002\Lab\Task3\Before\Lab2.sln).
2. Open the **MainForm** designer view by double-clicking on the **MainForm.cs** item in the Solution Explorer.
3. Double-click on the “**Transcribe**” button – this will generate an event handler for the default event, which is the **Click** event. Visual Studio should switch you to the code behind handler it has generated (**button1\_Click**).
4. In the click handler, we want to:
   1. Identify the selected sequence (if there is one).
   2. Identify what type of sequence it is (**DNA**, **RNA**, or **Protein**).
   3. If it is **DNA**, we want to transcribe it to **RNA**. If it is **RNA**, we want to reverse-transcribe it to **DNA**. If it is Protein we will ignore it.
   4. Finally, we will place the results into the last TextBox on the Form (**targetSequenceText**). You have the code necessary to display the first 255 characters of the sequence as a string. You might consider moving that to a new static method that you can reuse when needed!
5. You already know how to do most of the above – here is the final bit of information you need: The **Transcription** algorithm is contained in the **Bio.Algorithm.Translation.Transcription** class. It has two static methods: **Transcribe** and **ReverseTranscribe** that each takes an **ISequence** and return an **ISequence**.

See if you can use all the above to complete this step. If you need help, check the **after** project solution located at [Task3\After\Lab2.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp3_MBF.V2.zip\MBF.V2\Module%2002\Lab\Task3\After\Lab2.sln).

If this was easy, and you have plenty of time, try adding an additional bit of functionality. Take the transcribed RNA and translate it into protein strands using the **ProteinTranslation** algorithm. See the **after** project solution for an example of this.

Task 4: Saving Sequences

* 1. In this final task you will add support to save the resulting transcribed sequence to disk.

1. To begin, either continue from Task 3, or open the starter project located at [Task4\Before\Lab2.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp3_MBF.V2.zip\MBF.V2\Module%2002\Lab\Task4\Before\Lab2.sln).
2. Open the **MainForm** designer view by double-clicking on the **MainForm.cs** element in the **Solution Explorer**.
3. Click on the **File** menu item to expand it in the designer, and then double-click on the **Save** child menu item. This will generate the **Click** handler for the menu item and should transition you to the code behind.
4. We need a copy of the transcribed sequence – locate the Transcribe Buttons **Click** handler (it should be called **button1\_Click**). Towards the end of the method, where it is setting the result text, save the sequence results into a field in the class. The lab will call it “\_transcribedSequence”.
5. Scroll back to your **File | Save** menu handler. Here we want to use the **SaveFileDialog** to prompt the user for a filename to save the sequence to. Since we have a limited number of formatters, we will tell the dialog the proper file extensions to use by setting the **Filter** property.
6. As a first step, make sure you have a transcribed sequence – exit the method if you do not.
7. Next, create an instance of the **SaveFileDialog** type, name it “sfd” in the method.
   1. Set the **Title** property to “Select Filename for Transcribed Sequence”.
   2. Set the **OverwritePrompt** property to **true**.
   3. Set the **RestoreDirectory** property to **true**.
   4. For fun, set the **Filter** property to the following LINQ expression:
   5. string.Join("|", SequenceFormatters.All.Select(sf => string.Format("{0}|{1}", sf.Name, sf.SupportedFileTypes.Replace(',', ';')))),

This will take all the formatters, and join the extensions together in a semi-colon delimited list, with each formatter type being separated with a pipe “|” character.

We should end up with a string that looks like:

“FastA Files|\*.fa;\*.fasta;|GenBank|\*.gbb;\*.genbank … “

1. Display the dialog. If it is dismissed with an **OK** result, take the file name and locate the proper formatter using the **SequenceFormatters** class. With the selected formatter, call **Format** to save the sequence to a file.
2. Here is the code if you need some hints.
   1. **Note:** this uses a slightly different (but still valid) syntax to initialize the **SaveFileDialog**. This syntax is referred to as “inline property assignments”. It is a new feature of C# 3.0 where you can assign property values as part of the constructor syntax. Under the covers, the compiler is just expanding it for you. If you prefer to break it out to individual assignments that’s fine too –
   2. **sfd.**Title = “…”;  
      **sfd.**Filter = string.Join (…);
      1. private ISequence \_transcribedSequence;
      2. private void saveToolStripMenuItem\_Click(object sender, EventArgs e)
      3. {
      4. if (\_transcribedSequence != null)
      5. {
      6. SaveFileDialog sfd = new SaveFileDialog()
      7. {
      8. Title = "Select Filename for Transcribed Sequence",
      9. Filter = string.Join("|", SequenceFormatters.All
      10. .Select(sf => string.Format("{0}|{1}",
      11. sf.Name, sf.SupportedFileTypes.Replace(',', ';')))),
      12. OverwritePrompt = true,
      13. RestoreDirectory = true
      14. };
      15. if (sfd.ShowDialog() == DialogResult.OK)
      16. {
      17. var formatter = SequenceFormatters
      18. .FindFormatterByFileName(sfd.FileName);
      19. if (formatter != null)
      20. {
      21. formatter.Write(\_transcribedSequence);
      22. MessageBox.Show("Sequence has been saved to " + sfd.FileName,
      23. "Sequence Saved");
      24. formatter.Close();
      25. }
      26. }
      27. }
      28. }
3. Compile and run the application. Open a file and transcribe a sequence. Save the file to a new file, then open that new file to make sure it is correct.

The final solution for this lab is available at [Task4\After\Lab2.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp3_MBF.V2.zip\MBF.V2\Module%2002\Lab\Task4\After\Lab2.sln).