Bio .Net Programmer’s Primer

# Introduction

This document is meant to serve as an introduction to programming using the Bio .Net libraries from Microsoft Research. This will likely be helpful for both programmers wishing to pursue application development in the field of bioinformatics as well as for programmers wishing to contribute back to the Bio .Net libraries project. This primer should not be viewed as introduction to either the basics of biology or .Net programming.

All code examples will be given in C#, though any other .Net language can be used.

# Working with Sequences

A Sequence object stores the data and meta-data associated with any sequence of symbol-based data. Most often the sequence will represent a DNA, RNA, or protein sequence. Sequence allows access to the core data, which are defined as ISequenceItem implementations as well as defined meta-data (like an ID property) as well as general table based meta-data that can be defined by the user.

All sequence implementations support the interface ISequence and the standard implementation for ISequence is simply called Sequence. A Sequence must have its Alphabet (DNA, RNA, Protein, etc.) defined at all times. All other data and meta-data can be added or accessed at any time after the Sequence is constructed.

### The Sequence Class

To create a sequence for storing DNA data:

Sequence mySequence = new Sequence(Alphabets.Dna);

This creates a sequence with no data or meta-data. Sequences can be set to be read-only or editable. As of this writing, Sequence always defaults to read-only (which saves memory), but this can be changed by setting a single flag.

To create a DNA sequence whose nucleotides are already known:

Sequence mySequence = new Sequence(Alphabets.Dna, "GATTCCA");

The individual characters in the string must simply map to known characters in the DNA alphabet.

Another way to do this more explicitly:

Sequence mySequence = new Sequence(Alphabets.Dna);

mySequence.IsReadOnly = false;

mySequence.Add(Alphabets.Dna.G);

mySequence.Add(Alphabets.Dna.A);

mySequence.Add(Alphabets.Dna.T);

These sequences now store sequence data. Each symbol of data is stored as an ISequenceItem and every ISequence implementation can be treated as a generic List of ISequenceItem. Since the items stored in the DNA Alphabet are of type Nucleotide, Nucleotide must be an implementation of ISequenceItem.

Accessing the items in a sequence is simple. Consider this program which counts the number of ‘G’ symbols found in an ISequence:

ISequence mySequence = new Sequence(Alphabets.Dna, "GATTAGC");

int gCount = 0;

foreach (Nucleotide n in mySequence)

{

if (n == Alphabets.Dna.G)

gCount++;

}

### Meta-data in ISequence

Each sequence object holds some commonly used meta-data fields. These include:

|  |  |
| --- | --- |
| Field Name | Description |
| ID | An identifier of the sequence, meant for internal programming usage |
| DisplayID | An identifier of the sequence that can be displayed to the end user |
| MoleculeType | One of an enumeration of known types: DNA, Protein, mRNA, tRNA, etc. |
| Statistics | Information on the total counts of various sequence items in the sequence |

Each of these is accessed as a .Net property on the sequence class. The ISequence interface definition only allows reading these properties, though individual implementations of the sequence may allow setting them as well.

A generic dictionary is also available for meta-data that is not as commonly used, but may still be important to a particular application or sequence format. Data stored in this dictionary is accessed via a string key. Using the Sequence implementation you can also set items in this dictionary.

For instance, if you know the sequence you are looking at was found from a particular paper citation you can set that using:

ISequence mySequence = new Sequence(Alphabets.Dna, "GATTAGC");

mySequence.IsReadOnly = false;

mySequence.Metadata["Citation"] = "ABCs of Bio, Vol. 3";

To access the field at any time, you can call:

string citation = (string)mySequence["Citation"];

The cast is necessary because all of the values in the Metadata dictionary are stored as type object.

The various parsers may set their own meta-data in a reliable way, so be sure to consult the documentation of those parsers to be able to retrieve the data you are interested in.

### Other ISequence Implementations (Advanced)

In addition to the standard Sequence implementation of ISequence, there are several other implementations included in the base library.

* **VirtualSequence**  
  Allows access all of the meta-data fields of a sequence but does not allow the setting or retrieval of any sequence items. This is primarily used in applications where symbol data is not relevant, but annotations are.
* **SparseSequence**  
  For use when the sequence data is quite large, but only a small number of the items in that data are relevant. The SparseSequence stores its data in an indexed fashion, rather than as a list or array.
* **SegmentedSequence**  
  A SegmentedSequence is a sequence that is composed of several smaller sequences, one following another. This implementation allows you to define a larger sequence from smaller ones without having to duplicate the memory usage.
* **BasicDerivedSequence**  
  An IDerivedSequence is a sequence whose data mainly derives from a separate origin sequence but is modified in some way. With BasicDerivedSequence you can access the complement or reversal of the source sequence without having to store the data in memory twice. Future implementations of IDerivedSequence are planned.

### Sequence Encodings (Advanced)

In addition to storing an Alphabet which determines how to interpret symbols into ISequenceItem instances, the Sequence class also stores an Encoding which is settable by the user. Internally each SequenceItem is not stored as a character from a string but as a number (usually as a single byte). Alphabets and Encodings have a many-to-many relationship.

If you want to change an encoding, it is best to consult the documentation to find out which alphabets and encodings are compatible. This compatibility can also be found in the EncodingMap class.

By default a standard encoding is set for you when you set the Alphabet of the Sequence. You can set the Encoding at construction of the Sequence as well. Some encodings are able to better compress their data than others (usually at the loss of support for some of the symbols in the Alphabet). For more information see the API documentation in the constructors for the Sequence class.

# Parsers and Formatters

Parser implementations are used to create data Bio model objects from files that have been formatted according to some well-defined file format. Formatters work in the other direction, converting model objects back in to a file format.

Parsers that parse sequences derive from the ISequenceParser interface. This interface can be found in the Bio.IO namespace. Particular implementations will be found in their own sub-namespace. For instance the GenBankParser is found in namespace Bio.IO.GenBank.

To parse a GenBank formatted file:

GenBankParser parser = new GenBankParser();

string filePath = @"C:\SomeDirectory\SomeGenBankFile.gbk";

List<ISequence> sequences = parser.Parse(filePath);

You can use similar code for the other supported file formats. If you do not know the format ahead of time, the code library can also try to discover it for you, though this may not work 100% of the time:

string filePath = @"C:\SomeDirectory\SomeGenBankFile.gbk";

ISequenceParser parser = SequenceParsers.FindParserByFile(filePath);

List<ISequence> sequences = parser.Parse(filePath);

To format your sequence from a data object:

string filePath = @"C:\SomeDirectory\SomeGenBankFile.gbk";

GenBankFormatter formatter = new GenBankFormatter();

Sequence mySequence = new Sequence(Alphabets.Dna, "GATTTAAACC");

formatter.Format(mySequence, filePath);

# Algorithms

Algorithms that work with sequences and other biological structures modeled in the Bio code library can take many shapes and support many different scenarios. In general methods for manipulating data objects (e.g. aligning two sequences) or performing analysis are found in the Bio.Algorithms namespace. This is a growing area for the Bio .Net code libraries, and many useful features will continue to be added to this space.

For details on specific algorithms, please consult the API documentation.

# Web Services

Web service calls are found in the Bio.Web namespace. Each web service implements an interface called IWebService and while making a call to each web service implementation is done in a similar way in the Bio .Net libraries, the services themselves are all a little different.

One of the most common web services in the bioinformatics space is BLAST. There are several implementations of BLAST. In these libraries we provide access to NCBI’s QBLAST as well as EBI’s Wu-BLAST. As with most other web services, to call a BLAST web service you need to follow these steps:

1. Configure your web service request with the appropriate parameters for that service.
2. Make the web service call (this can be done synchronously or asynchronously).
3. Optionally query the status of the service while it is running.
4. Wait for and retrieve the result.

These steps can be illustrated in the following example, which call’s into NCBI QBLAST:

// Configure the service request

NcbiQBlast service = new NcbiQBlast();

ConfigParameters configParams = new ConfigParameters();

configParams.UseBrowserProxy = true;

service.ConfigurationParameters = configParams;

NcbiBlastParameters searchParams = new NcbiBlastParameters();

// Fill in the BLAST settings

searchParams.Add("Program", "blastp");

searchParams.Add("Database", "swissprot");

searchParams.Add("Expect", "1e-10");

searchParams.Add("CompositionBasedStatistics", "0");

service.ServiceParameters = searchParams;

// Create the request.

Sequence mySequence = new Sequence(Alphabets.Protein, “QIKDLLVS”);

string jobID = service.SubmitSearchRequest(mySequence);

// Query the status

ServiceRequestInformation info = service.GetRequestStatus(jobID);

if (info.Status != ServiceRequestStatus.Waiting &&

info.Status != ServiceRequestStatus.Ready)

{

// Handle the unexpected error

}

// Get async results. Use polling until the results are ready

int maxAttempts = 10;

int attempt = 1;

object resultsObject = null;

while (attempt <= maxAttempts && resultsObject == null)

{

resultsObject = service.FetchResultsAsync(jobID);

System.Threading.Thread.Sleep(resultsObject == null ? 5000 : 0);

++attempt;

}

List<BlastResult> results = resultsObject as List<BlastResult>;