The BioJava Tutorial

BioJava is a library of open source classes intended as a framework for applications which analyse or present biological sequence data. This tutorial illustrates the core sequence-handling interfaces available to the application programmer, and explains how BioJava differs from other sequence-handling libraries. For more comprehensive descriptions of the BioJava API, please consult the JavaDoc documentation.

1. Symbols and SymbolLists
2. Sequences and Features
3. Sequence I/O basics
4. ChangeEvent overview
5. ChangeEvent example using Distribution objects
6. Implementing Changeable
7. Blast-like parsing (NCBI Blast, WU-Blast, HMMER)
8. walkthrough of one of the dynamic programming examples
9. Installing BioSQL

The BioJava tutorial, like BioJava itself, is a work in progress, and all suggestions (and offers to write extra chapters ;) are welcome. If you see any glaring errors, or would like to contribute some documentation, please contact Thomas Down or the biojava-l mailing list.

BioJava.org

Open Bio sites

bioperl.org
biopython.org
bioxml.org
biodas.org
biocorba.org

Documentation

Overview
Getting started
Tutorial
BioJava in Anger
JavaDoc API
Demos
WikiDocs

Bug Tracking

Web Interface

Mailing List
biojava-l

Archive
Mail us
Subscribe

Mailing List
biojava-dev

Archive
Mail us
Subscribe

Forum (experimental)

Participants

Download

WebCVS

About BioJava

The BioJava Project is an open-source project dedicated to providing Java tools for processing biological data. This will include objects for manipulating sequences, file parsers, CORBA interoperability, DAS, access to ACeDB, dynamic programming, and simple statistical routines to name just a few things.

The BioJava library is useful for automating those daily and mundane bioinformatics tasks. As the library matures, the BioJava libraries will provide a foundation upon which both free software and commercial packages can be developed.

News

12th December	Take a look at BioJava in Anger, a new website with a collection of cookbook-style documentation for BioJava.
30th October	An experimental forum/blog has been set up. It may not be there always or it may migrate to another platform/host eventually. You can find it here
23th August	BioJava 1.22 released, bugfix for SequenceIO round-tripping errors.
14th March	BioJava 1.21 released, including BioSQL support, for general purpose storage of sequence data in a relational database.
13th February	BioJava 1.20 is ready, featuring improved sequence and blast parsers, faster dynamic programming routines, client libraries for DAS 1.0, and much more. Take a look at the change log or go straight to the download area
6th February 2002	There is going to be a repeat of last year's successful `BioJava boot camp' workshop, held at the Wellcome Trust Genome Campus near Cambridge. For more details, look here

Getting a copy of the project

BioJava is distributed under LGPL. This means that you can use the libraries without your software being forced under either the LGPL or GPL. LGPL is *not* GPL.

BioJava releases can be obtained by FTP from our download area. Instructions for installing the library, and building source releases, can be found on the Getting started page.

You can also maintain an up-to-date view of BioJava with CVS. We provide anonymous CVS server. If you wish to contribute your existing code or help maintain part of the BioJava code-base, then we can supply you with a read/write account.

Documentation

The overall design ethos of BioJava is probably the best place to start, as it explains how BioJava is structured.

The current JavaDoc api reference should provide up-to-date api documentation. Many of the demonstration programs have some description, both as to their purpose, and how to run them.

There is also a short tutorial on sequence handling in BioJava.

Thanks

This site could not exist without the donations of bandwith and hardware from Genetics Institute, Inc. and the Compaq Bioinformatics Solutions Center. In addition, we would like to thank Chris Dagdigian for maintaining the bio* servers.

All Classes

Packages
org.biojava.bio
org.biojava.bio.dist
org.biojava.bio.dp
org.biojava.bio.dp.onehead
org.biojava.bio.dp.twohead
org.biojava.bio.gui
org.biojava.bio.gui.sequence
org.biojava.bio.program
org.biojava.bio.program.blast2html
org.biojava.bio.program.das
org.biojava.bio.program.gff
org.biojava.bio.program.phred
org.biojava.bio.program.sax
org.biojava.bio.program.search
org.biojava.bio.program.ssbind
org.biojava.bio.program.xff
org.biojava.bio.program.xml
org.biojava.bio.proteomics
org.biojava.bio.search
org.biojava.bio.seq
org.biojava.bio.seq.db
org.biojava.bio.seq.db.biosql
org.biojava.bio.seq.db.emblcd
org.biojava.bio.seq.genomic
org.biojava.bio.seq.homol
org.biojava.bio.seq.impl
org.biojava.bio.seq.io
org.biojava.bio.seq.io.agave
org.biojava.bio.seq.io.game
org.biojava.bio.seq.projection
org.biojava.bio.seq.ragbag
org.biojava.bio.symbol
org.biojava.stats.svm
org.biojava.stats.svm.tools
org.biojava.utils
org.biojava.utils.cache
org.biojava.utils.io
org.biojava.utils.stax
org.biojava.utils.xml

All Classes

biojava

Core Packages

org.biojava.bio	The core classes that will be used throughout the bio packages.
org.biojava.bio.dist	Probability distributions over Alphabets.
org.biojava.bio.search	Interfaces and classes for representing sequence similarity search results.
org.biojava.bio.seq	Classes and interfaces for defining biological sequences and informatics objects.
org.biojava.bio.seq.db	Collections of biological sequence data.
org.biojava.bio.seq.genomic	Interfaces for representing key features of genomes.
org.biojava.bio.seq.io	Classes and interfaces for processing and producing flat-file representations of sequences.
org.biojava.bio.symbol	Representation of the Symbols that make up a sequence, and locations within them.

Sequence databases and formats

org.biojava.bio.program.das	Development client for the Distributed Annotation System.
org.biojava.bio.program.xff	Event-driven parsing system for the Extensible Feature Format (XFF).
org.biojava.bio.seq.db.biosql	General purpose Sequence storage in a relational database.
org.biojava.bio.seq.db.emblcd	Readers for the EMBL CD-ROM format binary index files used by EMBOSS and Staden packages.
org.biojava.bio.seq.io.agave	Classes for converting between AGAVE XML and BioJava objects.
org.biojava.bio.seq.io.game	Event-driven parsing system for the Gene Annotation Markup Elements (GAME).
org.biojava.bio.seq.ragbag	The Ragbag package is a set of classes for setting up a virtual sequence contig without the need of writing Biojava code.

User Interface Components

org.biojava.bio.gui	Graphical interfaces for biojava objects.
org.biojava.bio.gui.sequence	Graphical displays of biological sequences and associated annotations (highly experimental).

External Tools

org.biojava.bio.program	Java wrappers for interacting with external bioinformatics tools.
org.biojava.bio.program.blast2html	Code for generating HTML reports from blast output
org.biojava.bio.program.gff	GFF manipulation.
org.biojava.bio.program.phred	Parser for Phred output
org.biojava.bio.program.sax	Parsers which offer XML representations of the output from common bioinformatics tools.
org.biojava.bio.program.search	Interfaces and classes for parsing the results of external search programs.
org.biojava.bio.program.ssbind	Creation of SeqSimilaritySearchResult objects from SAX events using the BioJava BlastLikeDataSetCollection DTD.
org.biojava.bio.program.xml	Utility classes for the `org.biojava.bio.program.sax` package.

Dynamic Programming Packages

org.biojava.bio.dp	HMM and Dynamic Programming Algorithms.
org.biojava.bio.dp.onehead	
org.biojava.bio.dp.twohead	

Developers' Packages

org.biojava.bio.seq.impl	Standard in-memory implementations of `Sequence` and `Feature`.
org.biojava.bio.seq.projection	Code for projecting Feature objects into alternate coordinate systems.
org.biojava.utils	Miscellaneous utility classes used by other BioJava components.
org.biojava.utils.cache	A simple cache system with pluggable caching behaviours.
org.biojava.utils.io	I/O utility classes
org.biojava.utils.stax	The Stack API for XML (StAX).
org.biojava.utils.xml	This package contains a number of utilities for processing XML documents.

Experimental/New Packages

org.biojava.bio.proteomics	Utilities for analysing protein sequences.
org.biojava.bio.seq.homol	Feature interfaces for representing regions of homology between sequences.

org.biojava.stats.svm	Support Vector Machine classification and regression.
org.biojava.stats.svm.tools	Tools for use of the SVM package.

BarLogoPainter
BaseXMLWriter
BasicFeatureRenderer
BasisSymbol
BaumWelchSampler
BaumWelchTrainer
BetweenLocation
BioError
BioException
BioRuntimeException
BioSQLSequenceDB
Blast2HTMLHandler
BlastDBQueryHandler
BlastLikeHomologyBuilder
BlastLikeSAXParser
BlastLikeSearchBuilder
BlastLikeToXMLConverter
BlockPainter
BooleanElementHandlerBase
BumpedRenderer
ByteElementHandlerBase
Cache
CacheMap
CacheReference
CachingKernel
CachingSequenceDB
Cell
CellCalculator
CellCalculatorFactory
CellCalculatorFactoryMaker
Changeable
ChangeableCache
ChangeAdapter
ChangeEvent
ChangeForwarder
ChangeListener
ChangeListener.AlwaysVetoListener
ChangeListener.LoggingListener
ChangeSupport
ChangeType
ChangeVetoException
CharacterTokenization
CharElementHandlerBase
ChunkedSymbolListBuilder
CircularLocation
CircularSequence
CircularView
ClassifierExample
ClassifierExample.PointClassifier
Classify
ClustalWAlignmentSAXParser
ColourCommand
ComponentFeature
ComponentFeature.Template
ComponentFeatureHandler
CompoundLocation

ElementRecognizer.ByNSName
ElementRecognizer.ByNSName
ElementRecognizer.HasAttribute
ElementRecognizer.HasAttribute
ElementRecognizer.HasAttribute
EllipticalBeadRenderer
Embl2AgaveAnnotFilter
EmblCDROMIndexReader
EmblCDROMIndexStore
EmblCDROMRandomAccess
EmblFileFormer
EmblLikeFormat
EmblLikeLocationParser
EmblProcessor
EmblProcessor.Factory
EmissionCache
EmissionState
EntryNamIdxReader
EntryNamRandomAccess
Exon
Exon.Template
FastaDescriptionLineParser
FastaDescriptionLineParser.Factory
FastaFormat
FastaSearchBuilder
FastaSearchParser
FastaSearchSAXParser
FastaSequenceSAXParser
Feature
Feature.ByEmblOrderComparator
Feature.ByLocationComparator
Feature.Template
FeatureBlockSequenceRenderer
FeatureFilter
FeatureFilter.AcceptAllFilter
FeatureFilter.AcceptNoneFilter
FeatureFilter.And
FeatureFilter.AndNot
FeatureFilter.ByAncestor
FeatureFilter.ByAnnotation
FeatureFilter.ByClass
FeatureFilter.ByParent
FeatureFilter.BySource
FeatureFilter.ByType
FeatureFilter.ContainedByLocation
FeatureFilter.FrameFilter
FeatureFilter.HasAnnotation
FeatureFilter.Not
FeatureFilter.Or
FeatureFilter.OverlapsLocation
FeatureFilter.StrandFilter
FeatureHandler
FeatureHolder
FeatureHolder.EmptyFeatureHolder
FeatureImpl
FeatureLabelRenderer

GFFRecordFilter
GFFRecordFilter.AcceptAll
GFFRecordFilter.FeatureFilter
GFFRecordFilter.SequenceFilter
GFFRecordFilter.SourceFilter
GFFWriter
HashSequenceDB
HitDescHandler
HitIdHandler
Homology
HomologyDB
HomologyFeature
HomologyFeature.Template
HTMLRenderer
IDMaker
IDMaker.ByName
IDMaker.ByURN
IgnoreCountsTrainer
IgnoreRecordException
IllegalAlphabetException
IllegalIDException
IllegalSymbolException
IllegalTransitionException
Index
IndexedCount
IndexedSequenceDB
IndexStore
Initializable
IntegerAlphabet
IntegerAlphabet.IntegerSymbol
IntegerTokenization
IntElementHandlerBase
ItemValue
LabelRenderer
LabelRenderer.RenderNothing
LayeredRenderer
LazyFeatureHolder
LightPairDPCursor
LinearKernel
LineInfo
ListSumKernel
ListTools
ListTools.Doublet
ListTools.Triplet
ListWrapper
Location
Location.EmptyLocation
Location.LocationComparator
LocationHandlerBase
LocationTools
LogoContext
LogoPainter
LongElementHandlerBase
MagicalState
MarkovModel
MarkovModel.DistributionForwarder

MassCalc
MatrixPairDPCursor
Meme
MergeAnnotation
MergeFeatureHolder
ModelInState
ModelTrainer
MSFAlignmentFormat
MultiLineRenderer
NameTokenization
NcbiDatabaseURLGenerator
NestedError
NestedException
NestedKernel
NestedRuntimeException
NormalizingKernel
ObjectUtil
OptimizableFilter
OrderNDistribution
OrderNDistributionFactory
OverlayAnnotation
OverlayMap
OverlayMarker
OverlayRendererWrapper
PaddingRenderer
PairDistribution
PairDPCursor
PairDPMatrix
PairwiseDiagonalRenderer
PairwiseDP
PairwiseFilteringRenderer
PairwiseOverlayRenderer
PairwiseRenderContext
PairwiseSequencePanel
PairwiseSequenceRenderer
PairwiseSequenceRenderer.PairwiseRendererForwarder
ParseErrorEvent
ParseErrorListener
ParseErrorSource
ParseException
ParserException
PdbSAXParser
PdbToXMLConverter
PhredFormat
PhredSequence
PhredTools
PlainBlock
PlainStyle
PointLocation
PolynomialKernel
PrimaryTranscript
PrimaryTranscript.Template
ProfileHMM
ProjectedFeatureHolder
Projection
ProjectionContext

ProjectionEngine
ProjectionEngine.Instantiator
ProjectionEngine.InstantiatorImpl
PropDetailHandler
Protease
ProteinRefSeqFileFormer
ProteinRefSeqProcessor
ProteinRefSeqProcessor.Factory
ProteinTools
Qualitative
QueryableSequenceDB
QueryIdHandler
RadialBaseKernel
RagbagAssembly
RagbagComponentDirectory
RagbagComponentDirectory.EmptyComponentDirectory
RagbagFeatureTypeCatcher
RagbagFileParserFactory
RagbagFilteredCachedSeqFactory
RagbagFilterFactory
RagbagFixedCacheSeqFactory
RagbagHashedComponentDirectory
RagbagIdleSequenceBuilder
RagbagMap
RagbagSequenceFactory
RagbagSoftRefSeqFactory
RagbagUncachedSeqFactory
RandomAccessReader
RangeLocation
RealizingFeatureHolder
RectangularBeadRenderer
ReferenceServer
RelabeledAlignment
RemoteFeature
RemoteFeature.Region
RemoteFeature.Resolver
RemoteFeature.Template
ResourceEntityResolver
ReversibleTranslationTable
RNAFeature
RNAFeature.Template
RNATools
RoundRectangularBeadRenderer
RulerRenderer
SAX2StAXAdaptor
SAX2StAXAdaptor
ScoreType
ScoreType.NullModel
ScoreType.Odds
ScoreType.Probability
SearchBuilder
SearchContentHandler
SearchParser
SearchReader
SeqFileFormer
SeqFileFormerFactory

SeqIOAdapter
SeqIOEventEmitter
SeqIOFilter
SeqIOListener
SeqIOTools
SeqSimilarityAdapter
SeqSimilaritySearcher
SeqSimilaritySearchHit
SeqSimilaritySearchHit.ByScoreComparator
SeqSimilaritySearchHit.BySubHitCountComparator
SeqSimilaritySearchResult
SeqSimilaritySearchSubHit
SeqSimilaritySearchSubHit.ByScoreComparator
SeqSimilaritySearchSubHit.BySubjectStartComparator
Sequence
SequenceAlignmentSAXParser
SequenceAnnotator
SequenceBuilder
SequenceBuilderBase
SequenceBuilderFactory
SequenceBuilderFilter
SequenceContentHandlerBase
SequenceDB
SequenceDBInstallation
SequenceDBLite
SequenceDBSearchHit
SequenceDBSearchResult
SequenceDBSearchSubHit
SequenceDBWrapper
SequenceFactory
SequenceFormat
SequenceHandler
SequenceIterator
SequencePanel
SequencePoster
SequenceRenderContext
SequenceRenderContext.Border
SequenceRenderer
SequenceRenderer.RendererForwarder
SequenceRendererWrapper
SequencesAsGFF
SequenceViewerEvent
SequenceViewerListener
SequenceViewerMotionListener
SequenceViewerMotionSupport
SequenceViewerSupport
SigmoidKernel
SimilarityPairBuilder
SimilarityPairFeature
SimilarityPairFeature.EmptyPairwiseAlignment
SimilarityPairFeature.Template
SimpleAlignment
SimpleAlignmentStyler
SimpleAlphabet
SimpleAnnotation
SimpleAnnotFilter

SimpleAssembly
SimpleAssemblyBuilder
SimpleAtomicSymbol
SimpleDistribution
SimpleDistributionTrainer
SimpleDistributionTrainerContext
SimpleDotState
SimpleEmissionState
SimpleExon
SimpleFeature
SimpleFeatureHolder
SimpleFeatureRealizer
SimpleFramedFeature
SimpleGene
SimpleGFFRecord
SimpleHomology
SimpleHomologyFeature
SimpleIndex
SimpleItemValue
SimpleLabelRenderer
SimpleMarkovModel
SimpleModelInState
SimpleModelTrainer
SimplePrimaryTranscript
SimpleRemoteFeature
SimpleRemoteFeature.DBResolver
SimpleReversibleTranslationTable
SimpleRNAFeature
SimpleSeqSimilaritySearchHit
SimpleSeqSimilaritySearchResult
SimpleSeqSimilaritySearchSubHit
SimpleSequence
SimpleSequenceBuilder
SimpleSequenceDBInstallation
SimpleSequenceFactory
SimpleSimilarityPairFeature
SimpleSpliceVariant
SimpleStatePath
SimpleStrandedFeature
SimpleSVMClassifierModel
SimpleSVMTarget
SimpleSymbolList
SimpleSymbolPropertyTable
SimpleSymbolStyle
SimpleTranslatedRegion
SimpleTranslationTable
SimpleWeightMatrix
SimpleXMLEmitter
SingleDP
SingleDPMatrix
SingletonAlphabet
SingletonList
SixFrameRenderer
SixFrameZiggyRenderer
SmallAnnotation
SmallMap

SMORegressionTrainer
SMOTrainer
SoftReferenceCache
SparseVector
SparseVector.NormalizingKernel
SpliceVariant
SpliceVariant.Template
SSPropHandlerFactory
StackedLogoPainter
State
StatePath
StaticMemberPlaceHolder
StAXContentHandler
StAXContentHandler
StAXContentHandlerBase
StAXContentHandlerBase
StAXFeatureHandler
StAXFeatureHandler
StAXHandlerFactory
StAXHandlerFactory
StAXPropertyHandler
StAXPropertyHandler
StoppingCriteria
StopRenderer
StrandedFeature
StrandedFeature.Strand
StrandedFeature.Template
StrandedFeatureHandler
StreamParser
StreamReader
StreamWriter
StringElementHandlerBase
SubHitSummaryHandler
SubPairwiseRenderContext
SubSequence
SubSequenceDB
SubSequenceRenderContext
SuffixTree
SuffixTree.SuffixNode
SuffixTreeKernel
SuffixTreeKernel.DepthScaler
SuffixTreeKernel.MultipleScalar
SuffixTreeKernel.NullModelScaler
SuffixTreeKernel.SelectionScalar
SuffixTreeKernel.UniformScaler
SVM_Light
SVM_Light.LabelledVector
SVMClassifierModel
SVMKernel
SVMRegressionModel
SVMTarget
SwissprotFileFormer
SwissprotProcessor
SwissprotProcessor.Factory
Symbol
SymbolList

SymbolList.EmptySymbolList
SymbolListViews
SymbolPropertyTable
SymbolReader
SymbolSequenceRenderer
SymbolStyle
SymbolTokenization
SymbolTokenization.TokenType
TabIndexStore
TextBlock
TextLogoPainter
TickFeatureRenderer
Train
Trainable
TrainerTransition
TrainingAlgorithm
TrainingContext
TrainingEvent
TrainingListener
TrainRegression
Transition
TransitionTrainer
TranslatedDistribution
TranslatedRegion
TranslatedRegion.Template
TranslatedSequencePanel
TranslationTable
TypedProperties
TypesListener
UniformDistribution
URLGeneratorFactory
UtilHelper
ViewingSequenceDB
ViewSequence
WeakCacheMap
WebSequenceDB
WeightMatrix
WeightMatrixAnnotator
WMAsMM
WordTokenization
XFFFeatureSetHandler
XFFPartHandlerFactory
XMLBeans
XMLDispatcher
XmlMarkovModel
XMLPeerBuilder
XMLPeerFactory
ZiggyFeatureRenderer

BioJava: Symbols and SymbolLists

By Thomas Down

This chapter covers the fundamentals of accessing biological sequence data from BioJava, and explains how BioJava's treatment of sequences differs from other libraries. This chapter refers to Java API defined in the packages `org.biojava.bio.symbol` and `org.biojava.bio.seq`. For a complete overview of the APIs provided by these packages, please consult the JavaDoc documentation.

NOTE: this chapter has been updated for BioJava release 1.2.

Symbols and Alphabets

When biological sequence data first became available, it was necessary to find a convenient way to communicate it. A logical approach is to represent each monomer in a biological macromolecule using a single letter -- usually the initial letter of the chemical entity being described, for instance `T' for thymidine residues in DNA. When this data was entered into computers, it was logical to use the same scheme. A lot of computational biology software is based on normal string handling APIs. While the notion of a sequence as a string of ASCII characters has served us well to date, there are several issues which can present problems to the programmer:

Validation
: It is possible to pass *any* string to a routine which is expecting a biological sequence. Any validation has to be performed on an *ad hoc* basis.

Ambiguity
: The meaning of each symbol is not necessarily clear. The `T' which means thymidine in DNA is the same `T' which is a threonine residue in a protein sequence

Limited alphabet
: While there are obvious encodings for nucleic acid and sequence data as strings, the same approach does not always work well for other kinds of data generated in biological sequence analysis software

BioJava takes a rather different approach to sequence data. Instead of using a string of ASCII characters, a sequence is modelled as a list of Java objects implementing the `Symbol` interface. This class, and the others described here, are part of the Java package `org.biojava.bio.symbol`.

```
public interface Symbol {
    public String getName();
    public Annotation getAnnotation();
    public Alphabet getMatches();
}
```

All `Symbol` instances have a `name` property (for instance, Thymidine). They may optionally have extra information associated with them (for instance, information about the chemical properties of a DNA base) stored in a standard BioJava data structure called an `Annotation`. Annotations are just set of key-value data. The final method, `getMatches`, is only important for *ambiguous* symbols, which are covered at the end of this chapter.

The set of `Symbol` objects which may be found in a particular type of sequence data are defined in an `Alphabet`. It it always possible to define custom Symbols and Alphabets, but BioJava supplies a set of predefined alphabets for representing biological molecules. These are accessible through a central registry called the `AlphabetManager`, and through convenience methods.

```
FiniteAlphabet dna = DNATools.getDNA();
```

```
Iterator dnaSymbols = dna.iterator();
while (dnaSymbols.hasNext()) {
    Symbol s = (Symbol) dnaSymbols.next();
    System.out.println(s.getName());
}
```

SymbolList: the simple sequence

The basic interface for sequence data in biojava is `SymbolList`. Every SymbolList has an associated Alphabet, and may only contain Symbols from that alphabet. SymbolLists can be seen as strings which are made up of Symbol objects rather than characters. The interface specifies methods for querying the alphabet and length, and accessing the Symbols:

```
SymbolList seq = getSomeSequence();
System.out.println("Alphabet = " + seq.getAlphabet().getName());
System.out.println("Length = " + seq.length());
System.out.println("First symbol = " + seq.symbolAt(1).getName());
```

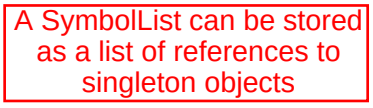
Note that numbering of Symbols within the SymbolList runs from 1 to length, *not* from 0 to length-1 as is the case with Java strings. This is consistent with the coordinate system found in files of annotated biological sequences.

There are several other standard methods in the SymbolList interface. `subList` returns a new SymbolList representing part of the sequence, just like the `substring` method of the `String` class. `seqString` returns a normal string representation of the sequence. This latter method will only work if the SymbolList uses an alphabet where all symbols have their `token` property defined. However, since this is true of the commonly used DNA and protein alphabets, this method is useful if you need interaction between BioJava and legacy sequence analysis code.

The `SymbolList` interface does not define any methods for modifying the underlying sequence data. Future versions of BioJava may also include a `MutableSymbolList` interface.

Doesn't this all waste memory?

A common concern with BioJava's Symbol/SymbolList model is that it must use much more memory than a simple string-based approach to sequence storage. It should be stressed that BioJava does *not* use a separate object to represent each nucleotide in a long DNA sequence. In fact, there are just four `singleton' `Symbol` objects which represent the symbols found in the DNA alphabet. These can be accessed at any time using static methods of the `DNATools` class. Whenever a thymidine residue is stored in a sequence, all that is really stored is a *reference* to the singleton thymidine object. Typically, this takes up four bytes of memory: more than the two bytes used by a Java `char`, but still manageable.



A SymbolList can be stored as a list of references to singleton objects

Actually, it is possible in principle to store a DNA sequence (without gaps or ambiguous residues) using only two *bits* per residue. Since the BioJava `SymbolList` is an interface, it only defines how the sequence should be accessed -- not how data is stored. If space is important, it is possible to implement a `packed' implementation of `SymbolList`. Client code need never worry about the underlying data model.

BioJava's object oriented view of sequences brings other advantages. Many programs which analyse DNA sequences need to have simultaneous access to the original sequence and that of its complementary strand. In BioJava this is easy.

```
SymbolList forward = getSequence();
SymbolList backward = DNATools.reverseComplement(forward);
System.out.println("First base: " + forward.symbolAt(1).getName());
System.out.println("Complement: " + backward.symbolAt(backward.length()).
                                        getName());
```

Since the reverse complement of a DNA sequence is a simple programmatic transformation, BioJava doesn't need to physically store the sequence in memory at all. Instead, it just creates a special implementation of the `SymbolList` interface, which computes the reverse strand sequence on the fly. This will typically cost just a few bytes of memory regardless of the sequence length, compared to megabytes for a string representation of a typical genome sequence.

How do I access my sequence data?

Each `Alphabet` object can have one or more `SymbolTokenization` implementations associated. These are two-way mappings between `Symbol` objects and textual representations of the data. They are the primary mechanism for creating new SymbolLists from existing (character-encoded) sequence data. By convention, any Alphabet which has a commonly accepted textual representation has a SymbolTokenization called `token' associated:

```
String seqString = "GATTACA";
Alphabet dna = DNATools.getDNA();
SymbolTokenization dnaToke = dna.getTokenization("token");
SymbolList seq = new SimpleSymbolList(dnaToke, seqString);
String seqString2 = dnaToke.tokenizeSymbolList(seq);
System.out.println("Strings match: " + seqString2.equalsIgnoreCase(seqString));
```

This low-level parsing mechanism is supplemented by a more sophisticated sequence Input/Output framework, defined in the package `org.biojava.bio.seq.io`. This uses pluggable file format converters, and can currently read and write in Fasta, EMBL, and Genbank formats. BioJava can also fetch data from services such as DAS, and BioCorba, and access databases such as those used by the Ensembl project (additional packages are required to BioCorba and Ensembl support).

What about the Sequence interface?

Until this point, we have concentrated on the `SymbolList` interface which, as its name suggests, is a raw list of `Symbol` references. Real entries in sequence databases are more complicated than this: sequences almost always have some kind of ID code or description associated, and many are also accompanied by tables of annotations. In BioJava, `Sequence` is a subinterface of SymbolList which adds a `name` property, plus a mechanism for querying tables of features.

The general rule is that the `Sequence` interface is normally used for sequences which have been loaded into a program from files or databases. `SymbolList` may be a more appropriate type for sequences generated internally by an analysis program.

A simple example

The following program is a very simple example, which reads one or more DNA sequences from a FASTA format data file and reports the GC content of each. This example is a (very) simple application of the BioJava Sequence I/O framework, described in later chapters. Used as below, it allows you to iterate over all the sequences in a multiple-entry file, rather than holding all of them in memory at once.

```
import java.io.*;
```

```
import org.biojava.bio.symbol.*;
import org.biojava.bio.seq.*;
import org.biojava.bio.seq.io.*;

public class GCContent {
    public static void main(String[] args)
        throws Exception
    {
        if (args.length != 1)
            throw new Exception("usage: java GCContent filename.fa");
        String fileName = args[0];

        // Set up sequence iterator

        BufferedReader br = new BufferedReader(
                                new FileReader(fileName));
        SequenceIterator stream = SeqIOTools.readFastaDNA(br);

        // Iterate over all sequences in the stream

        while (stream.hasNext()) {
            Sequence seq = stream.nextSequence();
            int gc = 0;
            for (int pos = 1; pos <= seq.length(); ++pos) {
                Symbol sym = seq.symbolAt(pos);
                if (sym == DNATools.g() || sym == DNATools.c())
                    ++gc;
            }
            System.out.println(seq.getName() + ": " +
                               ((gc * 100.0) / seq.length()) +
                               "%");
        }
    }
}
```

Ambiguous symbols

Sometimes, it is useful to represent sequences which are not perfectly defined. In such cases, it is common to use *ambiguous* symbols. A common example is the 'N' character in DNA sequences, which is used to indicate parts of a sequence where the sequencing traces were difficult to interpret. Sometimes, runs of Ns are also used to indicate gaps in assemblies. In the case of DNA, additional ambiguity symbols have been defined, covering all possible combinations of the four bases. For instance, the symbol 'W' realy means (A or T).

Within the BioJava object model, it is possible to inspect any ambiguous symbol to determine the set of atomic symbols which it matches, using the `getMatches` method. Atomic symbols can be considered to be the special case where `getMatches` returns a set whose size is exactly one. As a conveniece, atomic symbols also implement the `AtomicSymbol` interfaces.

You might want to modify the GCContent program, above, so as to ignore any ambiguous symbols in the input sequence.

BioJava sequences tutorial 0.3 by Thomas Down. Please mail any comments or suggestions to the author or to the biojava-l mailing list.

BioJava: Sequences and Features

By Thomas Down

Chapter 1 of this tutorial covered the `SymbolList` interface, BioJava's basic representation of biological sequence data. This chapter examines the `Sequence` interface. This adds extra functionality to `SymbolList`, providing a convenient way to handle annotated sequences from biological database. This chapter concentrates on classes and interfaces defined in the package `org.biojava.bio.seq`. For full descriptions of all the API used here, please consult the JavaDoc documentation.

A tour of a Sequence

`Sequence` is a sub-interface of `SymbolList`. Thus, all the standard methods for accessing sequence data in a `SymbolList` can equally be applied to a `Sequence`, and Sequences can be passed to any analysis methods which normally expect to receive a `SymbolList`. The `Sequence` interface adds two types of additional data to a `SymbolList`

- Global annotations, such as names, database identifiers, and literature references
- Location-specific annotations (features)

Two pieces of global annotation information are considered to be sufficiently important that they have dedicated accessor methods. The `name` of the Sequence is a simple string description of the Sequence: normally the name or accession number of the Sequence in the database from which it is retrieved. The `getURN` method, on the other hand, should return a more structured identifier for the sequence, represented as a Uniform Resource Identifier (URI) e.g.:

- `urn:sequence/embl:AL121903`
- `file:///home/thomas/genome.fasta|rpoN`
- `http://adzel.casseiopeia.org/seqs/myseqs.fasta|seq0001`
- `acedb://humace.sanger.ac.uk/DNA/AL121903`

URNs are a special class of URIs which represent global names for `well known' resources. Note that, despite the method name, it may not be appropriate to give an actual URN for sequences. However, for sequences from databases such as EMBL, where many sites have local installations, use of URNs is encouraged.

The exact use of the `name` and `URN` properties is currently dependent to some extent on how the sequence was loaded. As BioJava enters more common use, more formal definitions of these properties will emerge.

Other annotations

In additions to the two `identifier' properties of the Sequence, it may have other annotation data associated with it. BioJava contains an `Annotation` interface, which represents a set of key-value pairs, a little like a Java `Map` (indeed, `Annotation` has an `asMap` method).

```
Sequence seq = getSequence();
Annotation seqAn = seq.getAnnotation();
for (Iterator i = seqAn.keys().iterator(); i.hasNext(); ) {
    Object key = i.next();
    Object value = seqAn.getProperty(key);
    System.out.println(key.toString() + ": " + value.toString());
}
```

`Annotation` objects aren't just used in Sequences -- many other BioJava objects, including Features, can also have annotations associated with them.

Currently, there are no specific conventions for the kind of data which might be found in an `Annotation`. In general, the keys should be strings (although there is no requirement that this be the case). But the values may be any Java object. More guidelines for the contents of `Annotation` objects may be introduced as BioJava develops.

Features and FeatureHolders

A `Feature` represents a region of a sequence with some defined properties attached. Typically, features might represent structures such as genes and repeat elements on chromosomes, or alpha helices in proteins. As a Java interface, `Feature` has the following basic properties:

- A location within the sequence, represented by a `Location` object. This has a defined start and end (equal in the case of point locations), and may or may not be contiguous.
- A type (for instance, ``gene'' or ``helix'').
- A source (often the name of the program which discovered the feature.
- An Annotation object, which can contain any other data.

In addition, all Features have a place in a `tree' of Features, attached to a Sequence. Features cannot be created independently of a Sequence.

If a large class of features exists which have important properties over and above those represented in the `Feature` interface, a sub-interface of `Feature` may be defined. Currently, there is only one such sub-interface in the BioJava core: `StrandedFeature`. This is used for features in duplex DNA which have a defined directionality. For instance, genes would normally be represented with StrandedFeatures, while some kinds of regulatory region might be plain Features.

Sets of Features are stored in objects implementing the `FeatureHolder` interface. `Sequence` is a sub-interface of `FeatureHolder`. `Feature` itself also extends `FeatureHolder`, giving the possibility of representing `nested' features. For instance, a `Feature` representing a large genetic regulatory region might contain sub-features annotating individual transcription factor binding sites. The

recursive method below will print a simple text representation of a tree of features:

```
public void printFeatures(FeatureHolder fh,
                          PrintWriter pw,
                          String prefix)
{
    for (Iterator i = fh.features(); i.hasNext(); ) {
        Feature f = (Feature) i.next();
        pw.print(prefix);
        pw.print(f.getType());
        pw.print(" at ");
        pw.print(f.getLocation().toString());
        pw.println();
        printFeatures(f, pw, prefix + "    ");
    }
}
```

all `Feature` implementations include two methods which indicate how it fits into a feature tree. `getParent` returns the `FeatureHolder` (Sequence or Feature) which is the feature's immediate parent, while `getSequence` returns the `Sequence` object which is the root of the tree. Feature objects are *always* associated with a specific sequence, and always have exactly one parent `FeatureHolder`.

Creating new features

It is expected that there will never be any publicly visible implementations of `Feature` or its sub-interfaces. Instead, features should be produced using the `createFeature` method of a `FeatureHolder`. This ensures that there are no `orphan' features, not properly attached to a parent `Sequence`. It also gives `Sequence` implemetors the chance to control the attachment of features to their sequence class. Some sequences may only accept certain kinds of feature. Other implementations, especially those intimately coupled with database storage mechanisms, may wish to use their own special implementations of the `Feature` interface.

The `createFeature` method has the following signature:

```
public Feature createFeature(Feature.Template template);
```

there is no requirement that a particular `FeatureHolder` should include a working implementation of this method. If it is not possible to create a new child feature, `UnsupportedOperationException` will be thrown. In particular, this method is *only* implemented by `Sequence` and `Feature` objects. When `FeatureHolder` instances are used to return arbitrary `bags' of features, they will never support this method.

`Feature.Template` is a concrete nested class of the `Feature` interface. It just contains public fields corresponding to each property of `Feature`. A feature could be attached to a Sequence as follows:

```
Feature.Template template = new Feature.Template();
```

```
template.type = "TestFeature";
template.source = "Test";
template.location = new RangeLocation(100, 200);
template.annotation = Annotation.EMPTY_ANNOTATION;
mySequence.createFeature(template);
```

Every sub-interface of `Feature` should have a nested class, also named `Template`, which extends `Feature.Template` and adds any extra fields needed to construct that specialized kind of feature.

BioJava sequences tutorial 0.1 by Thomas Down. Please mail any comments or suggestions to the author or to the biojava-l mailing list.

BioJava: Sequence I/O basics

By Thomas Down

This chapter covers the BioJava support for handling biological sequence data available in the form of files. It covers API provided by the package `org.biojava.bio.seq.io`. For a complete overview of the API provided in this package, consult the JavaDoc documentation.

NOTE: this chapter has been updated for the BioJava 1.2 release.

Getting started with sequence I/O

The BioJava sequence I/O code is designed to be flexible and easy to adapt for a wide variety of purposes. However, if you don't need this flexibility, there are some convenience methods which set up the parsers for reading a variety of common formats. All these methods take a Java `BufferedReader`, and return an iterator which allows you to scan through the sequences in a file. For example:

```
BufferedReader br = new BufferedReader(new FileReader(fileName));
SequenceIterator stream = SeqIOTools.readFastaDNA(br);
while (stream.hasNext()) {
    Sequence seq = stream.nextSequence();
    // do something with the sequence.
}
```

For a full list of formats supported in this way, check the Javadoc documentation for the SeqIOTools class.

Sequence input goals

A typical biological sequence file contains three things:

- Global information about the sequence (ID, species, etc.)
- Annotations to specific regions of the sequence.
- Actual sequence data

Actual file formats need not provide all of these. For instance, FASTA files contain almost pure sequence data -- the only other information is a single description line for each sequence. At the other extreme, GFF files are simply a list of features, with no sequence data in the file.

A BioJava `Sequence` object contains the same kinds of information as a sequence file. The primary aim of the input architecture is obviously to take a stream containing sequence file data, and return one or more `Sequence` objects. In addition, we there are two other goals:

Decoupled `Sequence` creation

> BioJava represents sequence data using the `Sequence` interface. We allow multiple implementations of this interface, which might be optimized for quite specific purposes. Some implementations will be purely in-memory objects, while others might be persistant objects reflecting data in some kind of

database. We want to allow you to create any kind of `Sequence` object from a given data stream.

Pluggable filters

Not all users will wish to exactly reflect the contents of a sequence file as a `Sequence` object. Sometimes it is useful to select specific pieces of data from a file, or to change it into some other format. For instance, BioJava has a hierarchical model for features attached to a sequence, whereas many file formats (for instance, EMBL) do not. You might wish to rebuild some kind of feature hierarchy from an EMBL flatfile during the parsing process.

SequenceBuilders

The sequence input framework is based around the `SequenceBuilder` interface (this is actually a sub-interface of `SeqIOListener`, but for these purposes you will usually be using `SequenceBuilder`). The role of a SequenceBuilder is to accumulate information discovered while parsing a sequence file, and ultimately to construct a `Sequence` object.

There are two kinds of SequenceBuilder implementation.

Builders

These actually contruct new `Sequence` objects. Generally, there will just be one Builder implementation for each `Sequence` implementation. The basic BioJava library provides one Builder implementation, `SimpleSequenceBuilder`, which constructs simple in-memory representations for any kind of sequence data.

Filters

These don't construct Sequence objects themselves, but are chained to another `SequenceBuilder`. When they are notified of data, they perform some processing, then pass the information on to the next SequenceBuilder in the chain.

Whenever a `SequenceBuilder` is required, you can either simply provide a `Builder' implementation, or you can create a chain consisting of one or more `Filters', leading ultimately to a `Builder'.

A `SequenceBuilder` object should only be used once. If multiple sequences are being read from a stream, a new `SequenceBuilder` (or chain) should be constructed for each one. For convenience, we provide a `SequenceBuilderFactory` interface, whose sole purpose is to encapsulate the construction of SequenceBuilders. Each SequenceBuilder implementation should provide a suitable factory implementation as well.

For `Builder' implementations, it is usually possible to provide a `singleton' factory object. For `SimpleSequenceBuilder` this is the static field `SimpleSequenceBuilder.FACTORY`. For filters, the factory must be parameterized with another `SequenceBuilderFactory` so that a complete chain can be constructed. For instance:

```
SequenceBuilderFactory mySBF =
    new EmblProcessor.Factory(SimpleSequenceBuilder.FACTORY);
```

Authors of new `SequenceBuilder` implementations are encouraged to consider this naming style when implementing `SequenceBuilderFactory`.

Putting it together: StreamReader

The simplest way to use the BioJava sequence input code is to construct a StreamReader. The constructor takes four paramters:

- A normal Java `BufferedReader`, encapsulating the stream of data to parse.
- A `SequenceFormat` object, which is responsible for actually parsing sequence data from the stream.
- A `SymbolTokenization`, which represents a mapping from textual characters to BioJava `Symbol` objects.
- A `SequenceBuilderFactory` to support construction of `Sequence` objects.

A `StreamReader` might be constructed as follows:

```
Alphabet dna = DNATools.getDNA();
SymbolTokenization dnaParser = dna.getTokenization("token");
BufferedReader br = new BufferedReader(
                          new FileReader(fileName));
SequenceBuilderFactory sbf = new FastaDescriptionLineParser.Factory(
                                       SimpleSequenceBuilder.FACTORY);
StreamReader stream = new StreamReader(br,
                                       new FastaFormat(),
                                       dnaParser,
                                       fact);
```

(this is just a snippet from the example program in chapter 1, and you may like to refer back for more information).

The `StreamReader` class implements the `SequenceIterator` interface, so you can easily iterate over all sequences in a stream:

```
while (stream.hasNext()) {
    Sequence seq = stream.nextSequence();
    // Perform some processing on seq
}
```

Another application: IndexedSequenceDB

As biology enters the post-genomic era, it is common to need to work with databases of sequence data far too large to fit in available memory. One way to handle large amounts of sequence is to use a dedicated database system: either a specialized solution such as ACeDB or a set of tables in a standard database application, as used by the Ensembl project. If, however, you don't wish to use one of these solutions, BioJava offers a simple and efficient sequence database implementation backed by one or more sequence files on disk. These files can be in any format, so long as a suitable `SequenceFormat` class exists.

As a simple example of an IndexedSequenceDB in use, the following servlet retrieves sequences from a large database, and sends them on to the client in FASTA format. The database could be created using the CreateIndex and AddFiles programs included in the BioJava demos directory.

```
import java.io.*;
```

```
import javax.servlet.*;
import javac.servlet.http.*;

import org.biojava.bio.symbol.*;
import org.biojava.bio.seq.*;
import org.biojava.bio.seq.io.*;
import org.biojava.bio.seq.db.*;

public class SequenceServlet extends HttpServlet {
    private SequenceDB indexedDB;      // Database to serve
    private SequenceFormat seqFormat;  // Used for writing

    public void init(ServletConfig config)
        throws ServletException
    {
        super.init(config);
        String dbName = config.getInitParameter("sequence.db");
        if (dbName == null)
            throw new ServletException("Database not specified");
        try {
            TabIndexStore index = TabIndexStore.open(dbName);
            indexedDB = new IndexedSequenceDB(index);
        } catch (Exception ex) {
            log("Can't open sequence database: " + dbName, ex);
            throw new ServletException();
        }

        seqFormat = new FastaFormat();
    }

    public void doGet(HttpServletRequest req,
                      HttpServletResponse resp)
        throws ServletException, IOException
    {
        String id = req.getParameter("id");
        if (id == null) {
            resp.sendError(HttpServletResponse.SC_NOT_FOUND,
                           "No id parameter in request");
            return;
        }

        try {
            Sequence seq = indexedDB.getSequence(id);
            resp.setContentType("text/plain");
            PrintStream stream = new PrintStream(resp.getOutputStream());
            seqFormat.writeSequence(seq, stream);
```

```
        } catch (BioException ex) {
            log("Can't retrieve sequence", ex);
            resp.sendError(HttpServletResponse.SC_NOT_FOUND,
                           "Couldn't load sequence " + id);
        }
    }
}
```

BioJava sequences tutorial 0.3 by Thomas Down. Please mail any comments or suggestions to the author or to the biojava-l mailing list.

Changeability, Mutability and Events

By Matthew Pocock

BioJava contains a powerful API for communicating when objects wish to change their state, and potentialy preventing them from changing if it would invalidate the state of another object, all without violating the principals of encapsulation. The main classes are in the `org.biojava.utils` package and include `Changeable`, `ChangeEvent`, `ChangeListener`, `ChangeType` and `ChangeVetoException` . For full descriptions of all the API used here, please consult the JavaDoc documentation.

What is the difference between Changeability and Mutability?

Many Java objects are mutable. That is, you can invoke methods that change their state. The `Collections` API supplys mutable implementations of the `List` interface. There is also a method `Collections.immutableList(List l)` that returns a view of the underlying list where the mutators throw exceptions. Through this view object there is no way to edit the list. However, if the underlying list is modified then the 'imutable' view will reflect this. That is, although it is immutable, it is still changeable.

Things get even more complicated in the world of Bioinformatics. Many instances need to be mutable with respect to some clients and immutable for others. Also, some processes rely on objects remaining constant throughout. You can't perform a database search reliably if the database is being modified. However, once the search is complete there is no reason not to change the database. This transient immutability can't be modeled using the design pattern used for the `Collections`. The situation above is complicated even further because while a search is going on, every single sequence must be maintained in an uneditable state. However, a search object realy doesn't want to go through the process of modifying every single sequence object. This would be very ineficient. Something more flexible is needed, and the Changeability API is it.

What is a `ChangeEvent`?

`ChangeEvent` extends `java.util.EventObject` and adds the methods:

- `getChange` - the new value
- `getPrevious` - the old value
- `getType` - the 'type' of event
- `getChained` - an event that caused this event to be fired

In constrast to the classical Java events model, one event class is shared among all types of BioJava events. The 'type' of the event is signaled by the value of the type property. `ChangeType` is a final

class. Each interface that will fire `ChangeEvent`s will have public static final ChangeType fields with descriptive names. `ChangeEvent` objects store a descriptive name but are always compared with the `==` operator. This scheme is a type-safe extention of the Swing `PropertyChangeEvent` system but BioJava interfaces explicitly publish what types of event they may fire.

`ChangeListener`: The contract for handeling events

Objects that wish to be informed of `ChangeEvent`s must implement the `ChangeListener` interface. This has just two methods:

- `preChange(ChangeEvent ce)`
- `postChange(ChangeEvent ce)`

An object will invoke `preChange` to inform listeners that it wishes to alter its state. A `ChangeListener` may fire a `ChangeVetoException` to prevent this change from taking place. The event source *must* respect this. Once the event source has finnished updating its state, it will invoke the `postChangeEvent` method with an equivalent `ChangeEvent` (one with the same values for its properties). The `postChange` method should then take apropreate action to update the state of the listening object.

There are two `ChangeListener` implementations supplied by default. `ChangeListener.ALWAYS_VETO` always throws a `ChangeException` in `preChange`. This object is usefull if you wish to unconditionaly lock an object's property. In the exceptional circumstance when `ChangeListener.ALWAYS_VETO` is registered and a `postChange` is reached, it throws a NestedError with an assertion failure message. This should only be able to happen if the event source is incorrectly implemented.

`ChangeException.LOG_TO_OUT` prints all changes out to `System.out`. If you want to log to a different stream, construct a new instance of `ChangeListener.LoggingListener` with the stream.

Using `ChangeSupport` to implement `Changeable`

To flag that an object is a source of `ChangeEvent`s, it should implement `Changeable`. This interface has the following methods:

- `addChangeListener(ChangeListener cl)`
- `addChangeListener(ChangeListener cl, ChangeType ct)`
- `removeChangeListener(ChangeListener cl)`
- `removeChangeListener(ChangeListener cl, ChangeType ct)`

The methods with `ChangeType` arguments register the listener for that type of event only. The methods without register the listener for all events. Wherever possible, the type of event should be specified. This potentialy allows for lazy instantiation of various resources and will result in fewer events actualy being fired.

`ChangeSupport` is a utility class that handles 99% of the cases where you wish to implement the `Changeable` interface. Idealy, you should instantiate one of these objects and then delegate the listener methods to this. In addition to the methods in `Changeable` , `ChangeSupport` supplys the methods:

- `firePreChangeEvent(ChangeEvent ce)`
- `firePostChangeEvent(ChangeEvent ce)`

These methods invoke the `preChange` and `postChange` methods of the apropreate listeners. `firePreChangeEvent` will pass on any `ChangeVetoException`s that the listeners throw.

`AbstractChangeable` is an abstract implementation of `Changeable` that delegates to a `ChangeSupport`. In the cases where your class does not have to inherit from any class but must implement `Changeable`, this is a perfect base class. It will lazily instantiate the delegate only when listeners need to be registered.

In the next tutorial, we will implement an event source and add some listeners to it.

BioJava events tutorial 0.1 by Matthew Pocock. Please mail any comments or suggestions to the author or to the biojava-l mailing list.

Changeability examples

By Matthew Pocock

We are going to play with the `Changeability` code using the example of a GUI for viewing the roles on a rulet wheel. We will try to estimate the probability of the ball falling on any one of the 40 slots and of it falling on red or black.

Grab the source-code or run the applet directly. You may be unable to run the applet as it requires Java2. You should be able to view it in a Java2 appletviewer with no problems.

The imports

We will need to import some standard graphical packages to make the GUI, and java.util as it gives us stuff like iterators. From biojava, we will need org.biojava.utils for both the standard exceptions (NestedException and NestedError), and all of the Changeability api. The other biojava packages give us things like symbol objects, alphabets, annotations and probability distributions.

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.util.*;
import javax.swing.*;

import org.biojava.utils.*;
import org.biojava.bio.*;
import org.biojava.bio.symbol.*;
import org.biojava.bio.dist.*;
```

Setting up the roulet data

Firstly, we need to declare the class as extending JApplet so that we can use it inside a web-page and also rely on Swing working properly.

```
public class Roulet extends JApplet {
```

Then we can declare the static variables that will define the game.

```
  public static final FiniteAlphabet rolls;
  public static final Symbol [] allRolls;

  public static final FiniteAlphabet redBlack;
  public static final Symbol red;
  public static final Symbol black;

  // probability distribution used to sample rolls of the wheel
  public static final Distribution wheelRoler;
```

Of course, all of these items must be initialized. We will use a static initialization block.

```
  // stuff to make the roulet wheel exist.
  static {
    final int numRolls = 40;

    // make the rolls alphabet
    rolls = new SimpleAlphabet("Rolls");
    allRolls = new Symbol[numRolls];
```

Having made the rolls alphabet, we now must populate it with each possible roulet wheel outcome - 1..40 - as a symbol instance.

```
    for(int i = 1; i <= numRolls; i++) {
      Symbol s = allRolls[i-1] = AlphabetManager.createSymbol(
        (char) (i + '0'),
        i + "",
        Annotation.EMPTY_ANNOTATION
      );

      // attempt to add the symbol
      // this should work, but we still have to catch the exceptions. Since they
      // should be impossible throw, we re-throw them as assertion-failures.
      try {
        rolls.addSymbol(s);
      } catch (ChangeVetoException cve) {
        throw new NestedError(
          cve, "Assertoin Failure: Can't add symbol to the rolls alphabet"
        );
      } catch (IllegalSymbolException ise) {
        throw new NestedError(
          ise, "Assertoin Failure: Can't add symbol to the rolls alphabet"
        );
      }
    }
```

Notice that we have to catch exceptions that should be imposible to generate, but are specified in the API. Under different circumstances, these exceptions may be legitimately thrown, and we would have caught them and done something more sensible to handle the error.

```
    rolls.addChangeListener(ChangeListener.ALWAYS_VETO, Alphabet.SYMBOLS);
```

This is an example of using ALWAYS_VETO to prevent things from changing. Here we lock the SYMBOLS property of rolls so that no more symbol instances can be added or removed from the alphabet. This ensures data-integrity and makes it harder to write syntaciticaly correct buggs.

We must now make the red/black alphabet.

```
    redBlack = new SimpleAlphabet("Red/Black");

    // the "red" symbol
    red = AlphabetManager.createSymbol(
      'r', "red",
      Annotation.EMPTY_ANNOTATION
    );
```

```
    // the "black" symbol"
    black = AlphabetManager.createSymbol(
      'b', "black",
      Annotation.EMPTY_ANNOTATION
    );

    // again, add them and throw any exceptions on as assertion-failures.
    try {
      redBlack.addSymbol(red);
      redBlack.addSymbol(black);
    } catch (ChangeVetoException cve) {
      throw new BioError(
        cve, "Assertoin Failure: Can't add symbol to the red/black alphabet"
      );
    } catch (IllegalSymbolException ise) {
      throw new BioError(
        ise, "Assertoin Failure: Can't add symbol to the red/black alphabet"
      );
    }
    // and again lock the alphabet
    redBlack.addChangeListener(ChangeListener.ALWAYS_VETO, Alphabet.SYMBOLS);
```

Notice that again while the symbols are added we must check that nothing goes wrong. Also, again, we lock the red/black alphabet so that it can't be tampered with.

Now we will set up a probability distribution that can be sampled from to simulate the rolling of a roulet wheel. We will simply use an instance of UniformDistribution rather than generating a special distribution ourselves - cassinoes should have un-biassed wheels.

```
    wheelRoler = new UniformDistribution(rolls);
  }
```

And there we close the static block. Everything is set up for a game of chance.

Applet for playing the game

Let us start by setting up the state of the applet that will be used for estimating how the game is played, and for rendering the current best-guess for the outcomes of multiple roles of the wheel.

```
  private Distribution rollDist;
  private Distribution redBlackDist;
  private boolean running = false;
  private Thread countAdder;
```

rollDist will be our estimate of the probability of any one of the roles. redBlackDist is our estimate of getting one of red or black (even/odd). We will use the thread in countAdder to repeatedly sample the game, and when running is set to false, we will temporarily suspend sampling.

In the applet's init method we will set up all the state and build the GUI.

```
  public void init() {
    super.init(); // can't hurt...
```

Firstly, lets create the rollDist and redBlackDist objects.

```
    try {
      rollDist = DistributionFactory.DEFAULT.createDistribution(rolls);
    } catch (IllegalAlphabetException iae) {
      throw new NestedError(iae, "Could not create distribution");
    }

    redBlackDist = new RedBlackDist(rollDist);
```

Now we must make an object to estimate the rollDist probabilities. This is done using a DistributionTrainerContext instance called dtc. dtc will colate counts for each of the forty outcomes so that rollDist can then represent these frequencies as a probability distribution.

```
    final DistributionTrainerContext dtc =
      new SimpleDistributionTrainerContext();
    dtc.registerDistribution(rollDist);
```

Now we will create the thread that samples roles from the roulet wheel. It will synchronize upon itself so that we can suspend it as we wish.

```
    countAdder = new Thread(new Runnable() {
      public void run() {
        while(true) {
```

We will check the value of the running member variable to check if we should be sampling the wheel.

```
          boolean running;
          synchronized(countAdder) {
            running = Roulet.this.running;
          }
          if(running == true) {
```

Here we perform the sampling and inform the trainer of the role. To force rollDist to reflect the new counts, we also call tdc.train, and catch all the resulting exceptions (which should be imposible if everything is set up coorectly).

```
            Symbol s = Roulet.wheelRoler.sampleSymbol();
            try {
              dtc.addCount(rollDist, s, 1.0);
              dtc.train();
            } catch (IllegalSymbolException ise) {
              // should be impossible!
              throw new NestedError(
                ise, "Assertion Failure: Sampled symbol not in alphabet"
              );
            } catch (ChangeVetoException cve) {
              cve.printStackTrace();
            }
```

Now we will synchronize on the thread and sleep for a half seccond.

```
            synchronized(countAdder) {
              try {
                countAdder.wait(500);
              } catch (InterruptedException ie) {
```

```
            }
          }
```

This code handles the case when the sampling thread has been asked to stop running temporarily. Again, we must synchronize on the sampling thread.

```
        } else {
          synchronized(countAdder) {
            try {
              countAdder.wait();
            } catch (InterruptedException ie) {
            } catch (IllegalMonitorStateException imse) {
              throw new NestedError(imse, "Ouch");
            }
          }
        }
      }
    }
  });
```

That is the end of the sampling thread.

Now we can move onto the GUI. Let's set up buttons to start and stop the sampler thread and to clear the counts so far.

```
  final JButton start = new JButton("Start");
  final JButton stop = new JButton("Stop");
  final JButton clear = new JButton("Clear");
```

The start button must start of enabled, and should cause sampling to start.

```
  start.setEnabled(true);
  start.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
      synchronized(countAdder) {
        running = true;
        start.setEnabled(false);
        stop.setEnabled(true);
        countAdder.notify();
      }
    }
  });
```

The stop button should start off dissabled, and should cause the sampling to stop.

```
  stop.setEnabled(false);
  stop.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
      synchronized(countAdder) {
        running = false;
        start.setEnabled(true);
        stop.setEnabled(false);
        countAdder.notify();
      }
    }
```

```
    });
```

The clear button should be enabled, and should both clear the counts and susspend sampling.

```
    clear.setEnabled(true);
    clear.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        synchronized(countAdder) {
          running = false;
          start.setEnabled(true);
          stop.setEnabled(false);
          dtc.clearCounts();
          countAdder.notify();
        }
      }
    });
```

Now we should build the GUI components to render the probability distributions as pie-charts.

```
    Pie allPie;
    try {
      allPie = new Pie(rollDist, AlphabetManager.getAlphabetIndex(allRolls));
    } catch (IllegalSymbolException ise) {
      throw new NestedError(ise, "Assertion Failure: Can't make indexer");
    } catch (BioException be) {
      throw new NestedError(be, "Assertion Failure: Can't make indexer");
    }
    Pie redBlackPie = new Pie(redBlackDist);
```

Now, we add all of these components to the applet.

```
    getContentPane().setLayout(new BorderLayout());
    JPanel top = new JPanel();
    top.setLayout(new FlowLayout());
    top.add(start);
    top.add(stop);
    top.add(clear);
    getContentPane().add(top, BorderLayout.NORTH);

    JPanel center = new JPanel();
    center.setLayout(new FlowLayout());
    center.add(redBlackPie);
    center.add(allPie);
    Dimension d = new Dimension(200, 200);
    redBlackPie.setPreferredSize(d);
    allPie.setPreferredSize(d);

    getContentPane().add(center, BorderLayout.CENTER);
  }
```

This is the end of init. It has set up the state of the object, ready for it to render estimated probabilities of each wheel outcome being observed by repeatedly sampling the roulet wheel.

Starting the game off

The last bit of the applet is the command to set the sampler thread into motion. This realy fits into the applet's start method naturaly.

```
  public void start() {
    super.start();

    countAdder.start();
  }
}
```

And that is the end of the Roulet class.

The pie-chart rendering component

To render a distribution as a pie-chart, we need a custom sub-class of JComponent. It will have to respond to changes in the distribution and consistently paint itself on the screen. Here is the state it will need.

```
class Pie extends JComponent {
  private Distribution dist;
  private AlphabetIndex indexer;
  private ChangeListener repainter;
```

dist is the distribution that this pie-chart will render. indexer will be used to consistently order the states, and repainter is a ChangeListener instance that will repaint the pie whenever dist changes.

The first constructor just creates an alphabet indexer and chains onto the seccond one.

```
  public Pie(Distribution dist) {
    this(dist, AlphabetManager.getAlphabetIndex((FiniteAlphabet)
dist.getAlphabet()));
  }
```

The second constructor builds a couple of ChangeListener instances

```
  public Pie(Distribution dist, AlphabetIndex indexer) {
    this.dist = dist;
    this.indexer = indexer;

    repainter = new ChangeAdapter() {
      public void postChange(ChangeEvent ce) {
        repaint();
      }
    };

    dist.addChangeListener(repainter, Distribution.WEIGHTS);
  }
```

We must provide a way to render the pie-chart. JComponent likes us to override the paintComponent method, so this is what we shall do. The first job for the paint method is to work out some basic geometric points around which to render.

```
 protected void paintComponent(Graphics g) {
   super.paintComponent(g);
   Graphics2D g2 = (Graphics2D) g;

   double pad = 5.0;
   Rectangle2D boundingBox = new Rectangle2D.Double(
     pad, pad,
     getWidth() - 2.0 * pad, getHeight() - 2.0 * pad
   );
   double midx = getWidth() * 0.5;
   double midy = getHeight() * 0.5;
```

Now we can render each slice of the pie-chart, using a width proportional to the probability of each symbol, skipping each zero probability.

```
   double angle = 0.0;
   for(int i = 0; i < indexer.getAlphabet().size(); i++) {
     try {
       Symbol s = indexer.symbolForIndex(i);
       double p = dist.getWeight(s);
       if(p != 0.0) {
         double extent = p * 365.0;

         Arc2D slice = new Arc2D.Double(boundingBox, angle, extent, Arc2D.PIE);
         char token = s.getToken();
         if(s == Roulet.red) {
           g2.setPaint(Color.red);
         } else if(s == Roulet.black) {
           g2.setPaint(Color.black);
         } else if( ((token - '0') % 2) == 0) {
           g2.setPaint(Color.red);
         } else {
           g2.setPaint(Color.black);
         }

         g2.fill(slice);
         g2.setPaint(Color.blue);
         g2.draw(slice);

         angle += extent;
       }
     } catch (IllegalSymbolException ise) {
       ise.printStackTrace();
     }
   }
```

The last task is to render on some labels so that we know what each slice represents.

```
   angle = 0.0;
   g2.setPaint(Color.yellow);
   for(int i = 0; i < indexer.getAlphabet().size(); i++) {
     try {
       Symbol s = indexer.symbolForIndex(i);
```

```
        double p = dist.getWeight(s);
        if(p != 0.0) {
          double extent = p * 365.0;

          double a2 = Math.toRadians(angle + 0.5 * extent);
          g2.drawString(
            s.getName(),
            (float) (midx + Math.cos(a2) * midx * 0.8),
            (float) (midy - Math.sin(a2) * midy * 0.8)
          );

          angle += extent;
        }
      } catch (IllegalSymbolException ise) {
        ise.printStackTrace();
      }
    }
  }
}
```

That is the end of the pie-chart class.

RedBlackDist as a view onto the rollDist distribution

The RedBlackDist class will implement Distribution, but will need to map the 40-symbol alphabet of the entire roulet wheel into the 2-symbol alphabet of red/black. It must remain synchronized with the main wheel, updating its state whenever its parent does.

```
class RedBlackDist extends AbstractDistribution {
  private Distribution parent;
  private Distribution nullModel;
  private double red;
  private double black;

  private ChangeListener parentL;
  private ChangeListener propUpdater;
```

parent is the distribution being viewed. nullModel represents a view of the parent's null model. red and black will store the probabilities of comming up red or black in the parent. parentL will listen to the parent for when it changes and notify all interested parties that this distribution is changing in response. propUpdater will do the job of actualy calculating red and black from the parent.

Let's set up our distribution.

```
  public RedBlackDist(final Distribution parent) {
    this.parent = parent;
    generateChangeSupport(Distribution.WEIGHTS);

    parent.addChangeListener(parentL = new ChangeForwarder(
      this, changeSupport
    ) {
```

This listener will forward changes to the parent weights as changes to this distribution. It extends ChangeForwarder

that is a special instance that passes on changes to one object as knock-on events to another. By using the ChangeEvent constructor that includes a ChangeEvent, we can pass on the complete chain-of-evidence that allows listeners to work out why we are claiming to alter.

```
      protected ChangeEvent generateEvent(ChangeEvent ce) {
        return new ChangeEvent(
          getSource(), Distribution.WEIGHTS,
          null, null,
          ce
        );
      }
    }, Distribution.WEIGHTS);
```

We must also add a listener to ourselves to trap successful attempts to change (those that are not vetoed), and to update the values of red and black.

```
    addChangeListener(propUpdater = new ChangeAdapter() {
      public void postChange(ChangeEvent ce) {
        red = 0.0;
        black = 0.0;
        for(
          Iterator i = ((FiniteAlphabet) (parent.getAlphabet())).iterator();
          i.hasNext();
        ) {
          Symbol s = (Symbol) i.next();
          try {
            if( (s.getToken() - '0') % 2 == 0) { // even - red
              red += parent.getWeight(s);
            } else { // odd - black
              black += parent.getWeight(s);
            }
          } catch (IllegalSymbolException ise) {
            throw new NestedError(ise, "Assertion Failure: Can't find symbol");
          }
        }
      }
    }, Distribution.WEIGHTS);
  }
```

And that is the end of the constructor.

Now we must provide the missing methods in AbstractDistribution. These are fairly booring. Our alphabet is the same as the roulet redBlack object, and getWeightImpl will return the value of red for the red symbol and the value of black for the black symbol.

```
  public Alphabet getAlphabet() {
    return Roulet.redBlack;
  }

  protected double getWeightImpl(AtomicSymbol sym)
  throws IllegalSymbolException {
    if(sym == Roulet.red) {
      return red;
```

```
    } else if(sym == Roulet.black) {
      return black;
    } else {
      throw new IllegalSymbolException("No symbol known for " + sym);
    }
  }
```

All of these methods are just stubs. Notice that they throw ChangeVetoExceptions to indicate that they are not implemented. ChangeVetoException can either mean that the change is dissalowed because some listener explicitly stops it, or that the method is not supported. Either way, the state of the object will not be updated.

```
  protected void setWeightImpl(AtomicSymbol as, double weight)
  throws ChangeVetoException, IllegalSymbolException {
    throw new ChangeVetoException("RedBlackDist is immutable");
  }

  protected void setNullModelImpl(Distribution nullModel)
  throws ChangeVetoException, IllegalAlphabetException {
    throw new ChangeVetoException("RedBlackDist is immutable");
  }

  public Distribution getNullModel() {
    if(nullModel == null) {
      nullModel = new RedBlackDist(parent.getNullModel());
    }
    return nullModel;
  }
}
```

What you should see

If you type this in and compile, or run the applet directly, you should see a gui with a start, stop and clear button. If you click on start, the applet will start sampling the table every 1/2 seccond. You will notice that the two pie-charts reflect these roles by repainting. If you click stop, the sampling thread will stop getting new roles. If you click start again, then more counts will be collected. If you click clear, then the sampling will stop. Pressing start again will start the process off from the initial point of just one count collected.

By the end of this, you should feel comefortable with listening for events and writing custom ChangeListener implementations. You should be able to prevent a property from altering by adding an ALWAYS_VETO listener. You should have an understanding of how when one object changes, it may cause the state of another object to change, and off how to write a ChangeAdapter instance that will wire this together. I hope it was fun.

Implementing Changeability

By Matthew Pocock

We are going to implement a simple ChangeEvent source that stores a String name property and can inform other objects if this name changes. By the end of this tutorial you should be comefortable with the general issues surrounding implementing event sources and for ensuring that resources are allocated as needed.

The Nameable interface

By convention, BioJava always defines changes in an interface. This allows a range of implementations to provide a unified API to a change without mandaiting them to shair any code. We will define the Nameable interface.

```
package demos.Changeable;

import org.biojava.utils.*;

public interface Nameable extends Changeable {
```

The first thing we must do is define the ChangeType that indicates that the name has changed. By convention, it is a public static final field of the interface and is named in upper-case, with word boundaries indicated by underscores. The constructor needs a description, and also the name of the current class and the name of the field. This is so that during serialization, the ChangeType instance will resolve correctly both over time and between VMs.

```
  /**
   * The ChangeType that indicates that the name property has changed.
   */
  public static final ChangeType NAME = new ChangeType(
    "The name has changed.", // human-readable description
    "demos.Changeable",      // the current class name
    "NAME"                   // field name
  );
```

Now we have the definition of the accessor methods.

```
  /**
   * Return the name associated with this Nameable.
   *
   * @return the name property
   */
  public String getName();

  /**
   * Change the name associated with this Nameable.
   *
   * @param the new value for the name property
   * @throws ChangeVetoException if for any reason the name could not be set
   */
  public void setName(String name)
```

```
  throws ChangeVetoException;
}
```

and that's it for the Nameable interface.

The simplest implementation - extend AbstractChangeable

The simplest way to implement the Nameable interface is to inherit from AbstractChangeable. This is the aproach we will take here. Firstly we will define the class and add a couple of constructors.

```
package demos.Changeable;

import org.biojava.utils.*;

public class SimpleChangeable
extends AbstractChangeable
implements Nameable {
  private String name;

  public SimpleChangeable() {
    this(null);
  }

  public SimpleChangeable(String name) {
    this.name = name;
  }
```

The getName method can also be written in the obvious way.

```
  public String getName() {
    return name;
  }
```

The process of informing listeners requires some bagage to be present - in particular, the list of listeners. This would impose overhead on all instances of Changeable, regardless of whether listeners exist or not. The solution to this is to lazily instantiate the supporting objects. Fortunately, AbstractChangeable handles all of this for you. The two methods you need to use are hasListeners(), which will return true if there are any listeners at all and false otherwise. If there are no listeners, then the name can be set directly.

```
  public void setName(String name)
  throws ChangeVetoException {
    if(!hasListeners()) {
      this.name = name;
    } else {
```

If there are listeners, then the method getChangeSupport() is used to retrieve the ChangeSupport instance that maintains the listeners list. You should then synchronize on this to ensure that no listeners are added or removed while the name is being set.

```
      ChangeSupport cs = getChangeSupport(Nameable.NAME);
```

```
      synchronized(cs) {
```

Next, we make a new ChangeEvent to describe how the object wishes to alter, we fire a preChange notification to the listeners so that they have a chance to veto the change, we make the change and lastly we inform the listeners that the change has been made.

```
        ChangeEvent ce = new ChangeEvent(this, Nameable.NAME, name, this.name);
        cs.firePreChange(ce);
        this.name = name;
        cs.firePostChange(ce);
      }
    }
  }
}
```

That is the end of the implementation.

Using ChangeSupport directly

The previous example used ChangeSupport to store a list of listeners but via the AbstractChangeable class. Java only allows classes to inherit from one other class. This means that if you have a class that must implement Changeability but already is derived from another class, you can't use AbstractChangeable. You can, however, still use ChangeSupport. To illustrate this, we will look at the code in AbstractChangeable that wires in the ChangeSupport object.

AbstractChangeable is in the package org.biojava.utils, and implements Changeable. It is abstract as you *must* sub-class to provide code to actualy fire events.

```
package org.biojava.utils;

public abstract class AbstractChangeable implements Changeable {
```

The listener networks are not preserved during serialization. This is partly to prevent arbitrarily large networks of objects being dumped, and partly because listeners can be safely added in custom serialization/deserialization code.

```
  private transient ChangeSupport changeSupport = null;
```

The hasListeners method is implemented in the obvious way. It is protected, because it is realy a memory optimization method, and not part of the external interface of extending classes.

```
  protected boolean hasListeners() {
    return changeSupport != null;
  }
```

To retrieve the ChangeSupport delegate, we need to provide an access method. Again, this is protected and implemented in the obvious way.

```
  protected ChangeSupport getChangeSupport(ChangeType ct) {
    if(changeSupport == null) {
      changeSupport = new ChangeSupport();
    }

    return changeSupport;
```

```
  }
```

Some subclasses may wish to override this method and lazily instantiate resoruces when the first listener for a particular ChangeType is added. In this case, the overriden method should first call super.getChangeSupport and then perform any checkes it wishes.

Now that the protected methods are in place, we can provide the bodies of the listener management methods. These firstly use getChangeSupport to retrieve the delegate, and then ask it to add or remove a listener. We must synchronize on the delegate to make sure that it maintains in a consistent state.

```
  public void addChangeListener(ChangeListener cl) {
    ChangeSupport cs = getChangeSupport(null);
    synchronized(cs) {
      cs.addChangeListener(cl);
    }
  }

  public void addChangeListener(ChangeListener cl, ChangeType ct) {
    ChangeSupport cs = getChangeSupport(ct);
    synchronized(cs) {
      cs.addChangeListener(cl, ct);
    }
  }

  public void removeChangeListener(ChangeListener cl) {
    ChangeSupport cs = getChangeSupport(null);
    synchronized(cs) {
      cs.removeChangeListener(cl);
    }
  }

  public void removeChangeListener(ChangeListener cl, ChangeType ct) {
    ChangeSupport cs = getChangeSupport(ct);
    synchronized(cs) {
      cs.removeChangeListener(cl, ct);
    }
  }
}
```

And that is the end of the class. You should be able to cut-and-paste this code into your own Changeable objects to implement the basic delegate-management.

Using an abstract class to provide the event handeling

Often there are a number of implementatoins of an interface that are almost exactly the same except for the particulars of how data is stored. It is a shame to write the event code multiple times. A useful design pattern for this is to provide an Abstract class that takes care of all the synchronization issues and calles stub methods to perform the actual access to object state. Here is an example of that for the Nameable class.

The abstract class will look like this.

```
public abstract class AbstractNameable implements Nameable {
```

```
  public void setName(String name)
  throws ChangeVetoException {
    if(!hasListeners()) {
      setNameImpl(name);
    } else {
      ChangeSupport cs = getChangeSupport(Nameable.NAME);
      synchronized(cs) {
        ChangeEvent ce = new ChangeEvent(this, Nameable.NAME, name, this.name);
        cs.firePreChange(ce);
        setNameImpl(name);
        cs.firePostChange(ce);
      }
    }
  }

  protected abstract void setNameImpl(String name)
  throws ChangeVetoException;
}
```

The implementation would look something like this.

```
public class MyNameable extends AbstractNameable {
  private String name;

  public String getName() {
    return this.name;
  }

  public void setName(String name)
  throws ChangeVetoException {
    this.name = name;
  }
}
```

This split between the abstract implementation that handles all of the event guts and a realy light-weight implementation that controls access to data-storage is very useful in practice, and is used extensively in BioJava, particularly in the org.biojava.bio.dist package.

What next?

By now, you should be able to define interfaces that are Changeable, and to write implementations of these interfaces using AbstractChangeable or by delegating to ChangeSupport directly. For cases where there are many implementations that differ only in the means of data-storage, you should be able to factor the Changeablility code into an abstract class, and subclass this for each form of data-access.

BioJava: - Blast-Like SAX Parsers

Cambridge Antibody Technology

Introduction

- Getting up and running
- Example application - producing HTML from blast-like output

---

BioJava Blast2HTML tutorial 1.0 by Cambridge Antibody Technology.
Please mail any comments or suggestions to the author or to the biojava-l mailing list.

Simple HMMs with BioJava

by David Huen

We will now go through the source of one of the examples in the demos/dp directory: **Dice.java** (contributed by Samiul Hasan).

Introduction

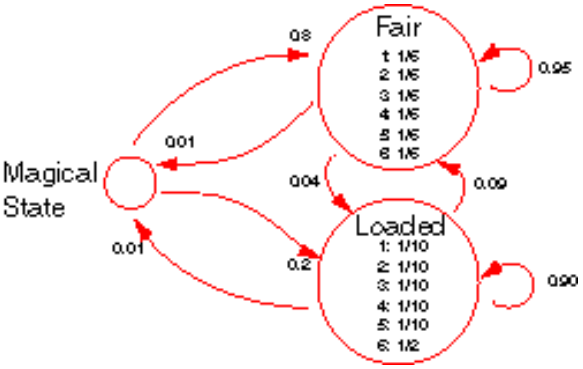
The program implements the "occasionally dishonest casino" example used in the book "Biological Sequence Analysis" by R. Durbin, S. Eddy, A. Krogh, G. Mitchison.

Basically, it conceives a casino with two dice, one fair and one loaded. The fair die lands on any of its sides equal probability while the loaded die yields "6" half the time, all the other sides being of equal probability. These probabilities represent the emission distribution of the fair die state and the loaded die states respectively.

The casino switches between using the fair die and the loaded die periodically. When on the fair die, the probability that the next throw is with the fair die too is 0.95. Similarly, when on the loaded die, the probability of continuing with it is 0.90. These probabilities yield the transition distributions of the states.

The HMM as modelled in the code is slightly modified from the above description with the inclusion of a MagicalState. This state is used to represent the start and end of the states of the model. The transition from the MagicalState to the fair die state occurs with a probability of 0.8 while the transition to the loaded die state occurs with a probability of 0.2. A termination condition was also introduced to allow transitions from the fair die and loaded die states to the Magical state with a probability of 0.01.

The resultant HMM looks like this:-



Code

The core of the program is the *createCasino()* method. This creates an instance of the **MarkovModel** class that implements the model.

```
public static MarkovModel createCasino() {
  Symbol[] rolls=new Symbol[6];

  //set up the dice alphabet
  SimpleAlphabet diceAlphabet=new SimpleAlphabet();
  diceAlphabet.setName("DiceAlphabet");

  for(int i=1;i<7;i++) {
    try {
      rolls[i-1]=
```

```
AlphabetManager.createSymbol((char)('0'+i),""+i,Annotation.EMPTY_ANNOTATION);
        diceAlphabet.addSymbol(rolls[i-1]);
      } catch (Exception e) {
        throw new NestedError(
          e, "Can't create symbols to represent dice rolls"
        );
      }
    }
```

A **Symbol** array ***rolls*** is created to hold the **Symbol**s generated by **AlphabetManager** to represent the outcomes of the dice. An Alphabet is also defined over these **Symbol**s.

Next, distributions representing the emission probabilities of the fair die and loaded die states are created (named ***fairD*** and ***loadedD*** respectively). The die states themselves are then created as **SimpleEmissionState**s, ***fairS*** and ***loadedS*** respectively.

You will observe an int array *advance* with a single value of 1. In a single-head HMM like ours, there is only one generated sequence and in our case, we progress along this sole sequence a single position per transition in the model. In multihead HMMs, there will be multiple sequences generated by the HMM and it is possible that the increment through the different sequences might be different. For example, single-stepping a protein sequence amounts to an increment of three on its corresponding DNA sequence.

```
    int [] advance = { 1 };
    Distribution fairD;
    Distribution loadedD;
    try {
      fairD = DistributionFactory.DEFAULT.createDistribution(diceAlphabet);
      loadedD = DistributionFactory.DEFAULT.createDistribution(diceAlphabet);
    } catch (Exception e) {
      throw new NestedError(e, "Can't create distributions");
    }
    EmissionState fairS = new SimpleEmissionState("fair",
Annotation.EMPTY_ANNOTATION, advance, fairD);
    EmissionState loadedS = new SimpleEmissionState("loaded",
Annotation.EMPTY_ANNOTATION, advance, loadedD);
```

The HMM is then created with these states:-

```
    SimpleMarkovModel casino = new SimpleMarkovModel(1, diceAlphabet, "Casino");
    try {
      casino.addState(fairS);
      casino.addState(loadedS);
    } catch (Exception e) {
      throw new NestedError(e, "Can't add states to model");
    }
```

Next, we need to model the transitions between the states. We do this like so:-

```
    try {
      casino.createTransition(casino.magicalState(),fairS);
      casino.createTransition(casino.magicalState(),loadedS);
      casino.createTransition(fairS,casino.magicalState());
      casino.createTransition(loadedS,casino.magicalState());
```

```
      casino.createTransition(fairS,loadedS);
      casino.createTransition(loadedS,fairS);
      casino.createTransition(fairS,fairS);
      casino.createTransition(loadedS,loadedS);
    } catch (Exception e) {
      throw new NestedError(e, "Can't create transitions");
    }
```

Note the presence of a MagicalState that is returned by *casino.magicalState()*. This is inherent to the **SimpleMarkovModel** class and does not need to be created by the user.

The emission distributions ***fairD*** and ***loadedD*** we set up earlier need to be initialised. We do that here.

```
    try {
      for(int i=0;i<rolls.length;i++)   {
        fairD.setWeight(rolls[i],1.0/6.0);
        loadedD.setWeight(rolls[i], 0.1);
      }
      loadedD.setWeight(rolls[5],0.5);
    } catch (Exception e) {
      throw new NestedError(e, "Can't set emission probabilities");
    }
```

We also need to initialise the transition distributions. Note how this is done: the transition distribution of each state is requested from the model with a *getWeights()* and then updated with the required values by calling the *getWeight()* method of that distribution. It is not necessary thereafter to call *setWeights()* to pass the Distribution for a state back to the model. This may seem strange but it is done this way because model object may use unique Distribution classes that cannot be replaced by a generic Distribution class for greater internal efficiency. Every state in the model needs to have its own transition distribution initialised appropriately.

```
    //set up transition scores.
    try {
      Distribution dist;

      dist = casino.getWeights(casino.magicalState());
      dist.setWeight(fairS, 0.8);
      dist.setWeight(loadedS, 0.2);

      dist = casino.getWeights(fairS);
      dist.setWeight(loadedS,               0.04);
      dist.setWeight(fairS,                 0.95);
      dist.setWeight(casino.magicalState(), 0.01);

      dist = casino.getWeights(loadedS);
      dist.setWeight(fairS,                 0.09);
      dist.setWeight(loadedS,               0.90);
      dist.setWeight(casino.magicalState(), 0.01);
    } catch (Exception e) {
      throw new NestedError(e, "Can't set transition probabilities");
    }
```

Having completed constructing the MarkovModel, all that remains is to return it to the caller.

```
    return casino;
```

Using the MarkovModel

Having created the **MarkovModel**, we create the corresponding dynamic programming object:-

```
DP dp=DPFactory.DEFAULT.createDP(casino);
```

Now, at last, we have something we can use! To generate a sequence of dice throws with this model, we do:-

```
StatePath obs_rolls = dp.generate(300);

SymbolList roll_sequence = obs_rolls.symbolListForLabel(StatePath.SEQUENCE);
```

The *generate()* method generates a path through the model that emits 300 symbols and we turn that path into a **SymbolList** with the second line.

At this point, it will be worthwhile digressing briefly on the StatePath object. This object embodies an **Alignment** of the sequences emitted by the **DP** object. In a multihead object, multiple aligned sequences will be emitted. In our case, only a single sequence is emitted and that is accessed with the label StatePath.SEQUENCE. That sequence turns out to a run of dice throws from our occasionally dishonest casino.

Next, we want to test one of the DP algorithms in the **DP** object. We want to process the ***roll_sequence*** **SymbolList** we have just generated and use the Viterbi method to predict which die each of the throws might have arisen from.

To do this, we create an array of **SymbolList**s with only ***roll_sequence*** in it - ours is a single-head HMM - and apply the Viterbi algorithm using the the model probabilities (ScoreType.PROBABILITY) for the computation (you could have also applied the null-model or log-odds probabilities here). This will yield the state path that has most support from the model and that state path is the model's prediction of which die a particular result came from.

```
SymbolList[] res_array = {roll_sequence};
StatePath v = dp.viterbi(res_array, ScoreType.PROBABILITY);
```

All that remains is to print out the generated sequence and the actual state path from which it came (ie. which die) and the HMM estimate of the state path, ***v***, (which is which die a particular throw came from as estimated by the HMM).

```
//print out obs_sequence, output, state symbols.
for(int i = 1; i <= obs_rolls.length()/60; i++) {
  for(int j=i*60; j
```

The first two print statements print the generated sequence and the actual state path by accessing the ***obs_rolls*** StatePath with the StatePath.SEQUENCE and StatePath.STATES respectively. The predicted state path comes from the third print block which accesses the ***v*** StatePath.

The output then looks like this:-

```
5445522135252456663636324325222535661665466666665336665432 61
fffffffffffllllllllllfffffffffffflllllllllllllllllffffff
fffffffffffffffffffffffffffffffffflllllllllllllllllffffff
```

```
3635462532525465244225552422232243444324233413654155516321 61
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

144212242323456563652263346116214136666156616666566421456123
ffffffllffffffffffffffffffffffflfllllllllllllllffffffff
fffffffffffffffffffffffffffffffflllllllllllllllffffffff

346313546514332164351242356166641344615135266642261112465663
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
```

The top line is the sequence emitted by our HMM when we made it generate 300 throws. The next is the state from which the throw came (f-fair l-loaded, these are the first letters of the labels "fair" and "loaded" we used when creating the **SimpleEmissionState** objects that represent the dice). The last is similar but this time from the **StatePath *v*** that is the result of the Viterbi algorithm. The performance is pretty on on this occasion but it can vary widely!

Installing and using BioSQL

by David Huen

Last modified: 19th November 2002.

This document describes how to install and use Biosql. BioSQL is a part of the OBDA standard and was developed as a common sequence database schema for the different language projects within the Open Bioinformatics Foundation.

While BioSQL is fairly vendor-neutral in its design, this tutorial is based on the case that I know best, that is, the installation of BioSQL on an x86 machine running RedHat 7.2.

Installing Postgresql

If not already installed, PostgreSQL can be installed from RPMs with:-

```
rpm -ivh postgresql-7.2.1-5.i386.rpm postgresql-libs-7.2.1-5.i386.rpm
postgresql-server-7.2.1-5.i386.rpm
```

Root privileges will almost certainly be required (if not your machine is seriously insecure!!!). You will also need a JDBC to permit Java to connect to your PostgreSQL database and that can be installed with postgresql-jdbc-7.1.3-2.i386.rpm. However, I would recommend downloading the latest from here. You will end up with a jar file containing the JDBC implementation which you will need to place in your CLASSPATH.

The installs will place a control script within /etc/init.d named postgresql. When this script runs for the first time, it will create a **database cluster** and initialise it. This cluster is the set of files used by the database for storage purposes.

On RH7.2 the default location for the cluster in at /var/lib/pgsql/. This is a bit of a disadvantage as /var is usually a pretty small partition. It is possible at this stage to symlink /var/lib/pgsql to a directory within another partition altogether to circumvent this problem. I would suggest doing this immediately.

At this stage, you will need to create the database you intend using and a user to use it. I would suggest **NOT** using the superuser named *postgres* for anything other than occasional essential administration.

At this point, I will digress briefly into PostgreSQL authentication as choices you make will affect what you can do. PostgreSQL has a variety of routes to achieve this. The default at installation permits connection only from local users and permits access to a database ONLY by a user of the same username. This may be quite adequate for experimentation but not so convenient if you want to set up a BioSQL database for several local users or possibly even remote users.

PostgresQL has other mechanisms which are described in their documentation. Authentication is specifically described here. You might consider password authentication but do use md5 encryption with this option, especially if you intend to authenticate remote users. In the Redhat 7.2 installation, the file you will need to edit to set these options is /var/lib/pgsql/data/pg_hba.conf. The location of this file varies with other distributions.

As initially installed in RH7.2, PostgreSQL will require root privileges to set up further. The *postgres* superuser cannot be logged into but you can invoke the necessary commands from root to execute:-

```
$ su postgres -c 'createdb <insert db name here>'
```

and a user created with:-

```
$ su postgres -c 'createuser <insert user name here>'
```

For the purposes of this tutorial, I will not change the default authentication so the database name should be chosen to correspond to your user name. The user name used in this exercise is **gadfly** and this will be reflected in the choice of database name and user name. One additional change that will be necessary is to enable TCP/IP connections as the Unix domain socket restriction of the default installation is incompatible with the PostgreSQL JDBC implementation.

To do so, you need to add the "-i" flag to the startup script. Edit /etc/init.d/postgresql and change the line:-

```
su -l postgres -s /bin/sh -c "/usr/bin/pg_ctl -D $PGDATA -p /usr/bin/postmaster start
> /dev/null 2>&1" < /dev/null
```

to:-

```
su -l postgres -s /bin/sh -c "/usr/bin/pg_ctl -o "-i" -D $PGDATA -p
/usr/bin/postmaster start  > /dev/null 2>&1" < /dev/null
```

The /var/lib/pgsql/data/pg_hba.conf file will also need to be edited to permit access via TCP/IP. This can be achieved by uncommenting:-

```
#host       all         127.0.0.1     255.255.255.255    trust
```

Both these operations require root access: seek advice as to the best option given your local security circumstances.

One additional change is that postgresql in RH7.3 does not come with the pgsql language enabled. As BioSQL uses that for acceleration, you will need to enable it. This can be done within root with:-

```
su postgres -c 'createlang plpgsql template1'
```

Installing BioSQL

The PostgreSQL server must be running to complete the BioSQL installation. You can check that it is with:-

```
$ /etc/rc.d/postgresql status
```

and doing:-

```
$ /etc/rc.d/postgresql start
```

if it is not running. You may require root privileges for this. You should have PostgreSQL started up during system startup with the SysV init system that comes with most Unixen.

You will need three scripts that serve to initialise the new database with the BioSQL schema and load accelerators for this schema. These are:-

```
biosql-accelerators-pg.sql
biosqldb-assembly-pg.sql
biosqldb-pg.sql
```

They may be obtained from here.

We now need to load the schema into the database we have created. We do so as follows (user entries in **bold**):-

```
$ psql gadfly
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

gadfly=> \i biosqldb-pg.sql
```

```
CREATE
psql:biosqldb-pg.sql:13: NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit
index 'biodatabase_pkey' for table 'biodatabase'
CREATE
<rest of output snipped>
INSERT 16862 1
psql:biosqldb-pg.sql:304: NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit
index 'cache_corba_support_pkey' for table 'cache_corba_support'
CREATE
gadfly=> \i biosqldb-assembly-pg.sql
<rest of output snipped>
gadfly=> \i biosql-accelerators-pg.sql
<rest of output snipped>
gadfly=> \q

$
```

Let's walk through the session above. **psql** is the name of the PostgreSQL interactive shell. We invoke it to connect to the PostgreSQL server and accept commands for a database named *gadfly* that we had created earlier. **psql** starts and displays its user prompt. All **psql** commands begin with a backslash (\). The **\i** instructs **psql** to take input from a file. I instruct **psql** to take input from the biosqldb-pg.sql, biosqldb-assembly-pg.sql and biosql-accelerators-pg.sql successively. **psql** reads the SQL statements within each of the files and proceeds to construct the BioSQL database schema, printing out a summary of its actions as it proceeds. Finally, I quit the **psql** interactive shell with **\q**. At this point you have a BioSQL schema installed and ready to run!!!

Do remember that if you do not explicitly load the JDBC drivers in your code, you should set a Java environment variable to tell it what to look for like so:-

```
java -Djdbc.drivers=org.postgresql.Driver <whatever your java code is>
```

BioJava: Blast-like Parsing Cook Book

By Cambridge Antibody Technology

This section of the biojava tutorial covers making use of the output from software used for sequence similarity/homology based searches of biological databases. The material is presented in a Cook Book fashion giving practical examples that should be enough to get you going. If you want to make use of the output from the following programs by using BioJava, this is a useful tutorial to work through:

- NCBI Blast (blastn, blastx, blastp, tblastn, tblastx)
- WU-Blast (blastn, blastx, blastp, tblastn, tblastx)
- or HMMER

NB Please check the JavaDocs of `BlastLikeSAXParser` *to see the extent of support for output from the various applications.*

The section of BioJava you will be making use of in the tutorial is the SAX2-compliant event-based parsing framework. After following this tutorial, you will you be able to not only to deal with output from the above pieces for bioinformatics software, but also get started with working with other types of data, such as three-dimensional macromolecular structures which are also supported by the framework.

What you need to know about the parsing framework

The framework has been designed in such a way that you don't need to understand the details of how it works in order to use it. This is achieved by providing facade classes that are simple to use. For parsing Blast-like output, the facade class you need to use is `org.biojava.bio.program.sax.BlastLikeSAXParser`. You pass streams of data to this class, and the framework will do the rest. As the name suggests, this class is actually a SAX parser, and implements the `org.xml.sax.XMLReader` interface. You are thus able to treat the output data as thought it is in an XML format.

The framework performs the magic of emitting SAX2 events from non-XML format data. Thus you don't have to do any parsing yourself. Rather you will simply be writing XML Content Handlers. The recipes for XML Content Handlers presented here will point you in the direction of populating your own (or BioJava) objects with bioinformatics data.

It is also worth noting, that the SAX events that the framework emits are consistent with a scenario where all the pieces of bioinformatics software above, actually produced identically formatted data.

Benefits of using the framework

- Allows you to focus on the objects you want to create, and forget about writing complex parsing code
- Allows you to make use of the output from more pieces of software. Because of the "concept-based" approach to the representation of data, many of the Content Handler classes you write can be re-used with the output of several different programs.

Recipes

The recipes are simple examples designed to get you up and running populating objects in the way you want. For each example recipe, two classes are provided:

- An XML Content Handler (this is the class that does the work of populating objects with data)
- A sample application class that takes blast-like program output and and sets up for parsing using the Content Handler class.

NB You will find the complete source code for all the classes described here the demos section of biojava, in the eventbasedparsing package.

After Example 1, the only classes that are described are the XML Content Hander classes, because the application classes are essentially identical for all examples.

To help you get going, in addition to the source code for the examples, there are also several example examples of raw ouput from NCBI-blast, WU-blast, and HMMER the "files" directory of the demos section of biojava.

Example 1

For all the hits from a search as detailed in the summary section of the output, prepare a list of Hit Ids. This is an example of a re-useable Content Handler. The same piece of code works equally well with the output from multiple flavours of NCBI Blast, WU-Blast, and HMMER.

Step A - Create an application that sets up the parser and does the parsing

The full source is in `eventbasedparsing.TutorialEx1`. Because there is no difference between what you do here, and what you would do to parse XML files there isn't much to do. First create a SAX Parser that deals with Blast-like output.

```
XMLReader oParser = (XMLReader) new BlastLikeSAXParser();
```

Next choose the Content Handler. In this case, we will be using the class `TutorialEx1Handler`, which takes a reference to an ArrayList in the constructor. When the SAX Parser parses the file, the ContentHandler will populate the ArrayList with Hit Ids from the summary section of the output.

```
ContentHandler oHandler =
   (ContentHandler) new TutorialEx1Handler(oDatabaseIdList);
```

The final step in the set-up is to connect the Content Handler to the SAX Parser.

```
oParser.setContentHandler(oHandler);
```

For the purposes of the tutorial applications, we will simply be reading output from files on disk. Create a `FileInputStream`, and parse it by calling the `parse` method on the SAX Parser.

```
oInputFileStream = new FileInputStream(oInput);
oParser.parse(new InputSource(oInputFileStream));
```

Finally, having populated the ArrayList with HitIds, we simply print them out.

```
System.out.println("Results of parsing");
System.out.println("==================");
for (int i = 0; i < oDatabaseIdList.size();i++) {
      System.out.println(oDatabaseIdList.get(i));
}
```

Step B - Create the logic for parsing

This is simply of matter of writing an XML Content Handler. The full source is in `eventbasedparsing.TutorialEx1Handler`. The logic here is trivial, we simply wish to identify Hit Ids that are contained within in the Summary sections of the output data, and add each Hit Id to the ArrayList.

```
if ( (oNameStack.peek().toString().equals("HitId")) &&
     (this.findInStack("Summary") != -1) ) {
   oDatabaseIdList.add(poAtts.getValue("id"));
}
```

Running the application

After compiling, if you run the application from the demos directory by typing the following:

```
java eventbasedparsing/TutorialEx1 files/ncbiblast/shortBlastn.out
```

You should see the following output:

```
Results of parsing
==================
U51677
L38477
X80457
```

BioJava: Blast2HTML

Cambridge Antibody Technology

Introduction

This tutorial covers the use of the Blast-like parsing framework to generate HTML representations of the Blast-like XML.

Here are some examples of the type of output you can generate.

- Blastp
- Blastn

Prerequisites are:-

- an upto date copy of biojava
- the programs in the demos directory

Running the demos

To generate for yourself the above example HTML files, change directory to the demos directory of biojava. The following commands will generate the HTML to standard out.

java eventbasedparsing.Blast2HTML nucleic files/ncbiblast/blastn.out
java eventbasedparsing.Blast2HTML protein files/ncbiblast/blastp.out

You can choose an output file (instead of redirecting standard out) by adding a third argument to the command -

java eventbasedparsing.Blast2HTML protein files/ncbiblast/blastp.out blastp.html

Customising the Output

The `HTMLRenderer` constructor takes several parameters which allow customisation of the HTML.

- **Style sheet**
  You can change the definition of the styles in the style sheet.
- **Alignment width**
  The alignment width simply specifies the number of bases/residues per alignment block.
- **URLGeneratorFactory**
  Returns a `List` of `DatabaseURLGenerator`s. These are used to convert database ID's to URL's and links. You can create your own. See NcbiDatabaseURLGenerator for an example.
- **`AlignmentMarker`**
  Delegates most of it's operations to the `ColourCommand` and `AlignmentStyler`.
  - `ColourCommand`

Controls whether a pair of characters in the alignment are styled or not.

- `AlignmentStyler`
  Decides what style to apply to any given pair of characters.

E.g. To markup mismatches in red you would have a `ColourCommand` that decides only mismatches are coloured, and then an `AlignmentStyler` that colours any characters passed to it as red.

There are a couple of implementations of AlignmentStyler: SimpleAlignmentStyler and BlastMatrixAlignmentStyler - see the Javadocs for details.

Of course you can also use custom handlers to only pass on a subset of the output.

BioJava Blast2HTML tutorial 1.0 by Cambridge Antibody Technology.
Please mail any comments or suggestions to the author or to the biojava-l mailing list.