Umber Bellows 0.3.5 User Guide

Chapter 1 - Introduction

Managing XML data in Java can be a laborious task, whether you're dealing with event-based SAX loaders or tree-based DOM loaders. Even generated Java/XML bindings have their problems. Often you just want to load up an XML document, check a property, and close it out. With the three solutions above, you have to load the XML, iterate through the elements until you find (or don't find) the one you're looking for, and convert the results into a usable form, resulting in a pile of XML-specific application code. The Bellows XML library aims to make the "loading" and "finding" tasks quick and painless.

Bellows introduces a simple lightweight Java object, the Datum, to hold each XML element in memory, similar to a stripped down DOM Node object. Bellows then uses a user-friendly pattern-based query language, based on XPath but simplified and extended in other ways, to make it easy to grab the XML data you want. And it only takes a line or two of Java code.

Bellows is very lightweight, only 124 KB itself, with no required third-party library dependencies. Under Java 1.4, Bellows can automatically detect and make use of the bundled XML parser; under Java 1.3 and earlier, it can fall back on optional third party XML parsers.

This user guide will introduce you to the three types of Datum objects, Datum, ListDatum, and TableDatum, and will discuss the Bellows query language in depth.

Chapter 2 - The Datum

The Datum object is the atomic unit of data in the Bellows world. Each XML element it loads is converted into a Datum; XML attributes are loaded into the Datum property API. The Datum also allows for XML-style namespaces and an optional type safety mechanism to help keep data valid.

Section 2.1 - Properties

At its simplest, a Datum is a glorified HashMap. Each entry in the Datum HashMap is called a "property", and can be set and retrieved with setProperty() and getProperty(). Property names are always Strings, but property values can be any Java object. Properties loaded from an existing XML document will of course have all String values, but Datum objects you create from scratch can have non-String

properties.

```
public void setProperty (String name, Object value)
public Object getProperty (String name)
```

The Datum goes beyond a simple HashMap with the addition of the parent-child relationship. Datum objects can be assembled into a hierarchical tree with the setParent() and getChildren() methods. Any Datum object can have a parent, and more than one Datum can share the same parent; a Datum with a null parent is the top, or root, node of its hierarchy. The base Datum getChildren() method always returns an empty array. As we will see later, the ListDatum object adds support for adding child objects.

```
public void setParent (Datum parent)
public Datum getParent ()
public Datum[] getChildren ()
```

When resolving properties with the getProperty() method, the Datum will first search for a value in its local properties. If it finds a local value, it will return that. However, if the requested property does not exist locally, the Datum will look for the same property in its parent, and the parent's parent, and so on until it finds the property or traverses past the root Datum. If the property cannot be found, getProperty() will return null. In other words, Datum objects inherit properties from their parents, but can override them locally.

Sometimes, you'll want to limit your property search to the local Datum, without property inheritance. The getLocalProperty() method returns the local property without searching the parent. No setLocalProperty() method exists, because it is not possible to directly set parent properties from the child (if you needed to do that, you could acquire the parent with getParent() and set it there). The setProperty() method only affects the local property set.

```
public Object getLocalProperty (String name)
```

The getKeyNames() method retrieves a complete list of local property names in the Datum (not the parent).

```
public String[] getKeyNames ()
```

Each Datum also has a type and an id. The type is the name of the Datum's abstract type, and is roughly equivalent to its XML element name. The id is the unique text identifier for the Datum, but its use is up to the application. For example, a smart XML loader might place ID attributes (in the XML DTD sense) in

the "id" property.

```
public String getType ()
public String getId ()
```

The Datum object is very flexible, but sometimes flexibility comes at the cost of added complexity. All Datum properties are managed, regardless of type safety, as java.lang.Object references, and must be converted (usually a simple cast) to whatever data type you need. The Datum class offers many convenience methods to simplify your data access code.

The Datum property convenience methods use some simple rules to convert Datum data into a variety of common types. These accessors let you specify a default value to return if the property does not exist, and whether or not it should check parents for the properties. Two versions exist for each type of accessor, with the inheritance flag present on one, and assumed to be false (or local values only) on the other.

```
public String getStringProperty (String key, String defaultValue)
public String getStringProperty (String key,
  String namespace, String defaultValue, boolean inherited)

public boolean getBooleanProperty (String key, boolean defaultValue)
public boolean getBooleanProperty (String key,
  String namespace, boolean defaultValue, boolean inherited)

public int getIntegerProperty (String key, int defaultValue)
public int getIntegerProperty (String key,
  String namespace, int defaultValue, boolean inherited)
```

Another accessor specializes in enumerated value sets, converting text values into an indexed key:

```
public int getEnumProperty (String key, String[] enums, int defaultValue)
public int getEnumProperty (String key,
  String namespace, String[] enums, int defaultValue, boolean inherited)
```

The enum accessor takes the same parameters as the other accessors, except for an additional String[] array of enum values. The accessor will look up the requested property value in the enum array and return the index of the match, or the defaultValue if it fails to find a match. The following example demonstrates enumerated font styles. Given the XML element

```
<font style="bold"/>
```

and the code to create it as a Datum object

```
Datum datum = new Datum ("font");
datum.setProperty ("style", "bold");
```

and an enumeration that contains common font style names

```
String[] enums = new String[] { "plain", "bold", "italic" };
```

the enum accessor will return a value of 1, corresponding to the array offset of "bold" in the enums array:

```
int style = datum.getEnumProperty ("style", enums, -1);  // 1
```

Here are some other results:

```
datum.setProperty ("style", "plain");
style = datum.getEnumProperty ("style", enums, -1);  // 0

datum.setProperty ("style", "italic");
style = datum.getEnumProperty ("style", enums, -1);  // 2

datum.setProperty ("style", "nonexistent");
style = datum.getEnumProperty ("style", enums, -1);  // -1
```

Section 2.2 - Namespaces

XML introduces the concept of namespaces to data documents. XML documents can have more than one element type with the same name, yet still be distinct types. XML namespaces are most commonly used to tag content that conforms to different XML schema, but is interleaved inside the same document. For example, if you wanted to embed an SVG line drawing inside your text XML document, you could declare an "SVG" namespace and place all SVG elements in that namespace. The XML namespace precedes the element and attribute names, and is delimited by a colon (:). The SVG example might look like this:

```
<document>
    ...
  <paragraph>
    <text>The graph would look something like this:</text>
    <svg>
        ...</svg>
```

```
    <text>And furthermore...</text>
  </paragraph>
</document>
```

The <svg> element and its attributes fall into the "SVG" namespace.

Datum namespaces are similar to XML namespaces but do not match perfectly. Datum properties (and XML attributes) can exist in separate namespaces, but Datum objects themselves do not currently reside in namespaces. In the future, Datum namespaces will be brought more into line with XML namespaces to make translation between the two easier. In the short term, element-level namespacing can be simulated by assigning Datum or Datum[] objects to a property within that namespace. As we will see later, the Bellows query language provides special support for handling Datum content inside a property.

Using Datum namespaces is as simple as calling the alternate versions of the property methods with the extra namespace parameter.

```
public void setProperty (String name, Object value, String namespace)
public Object getProperty (String name, String namespace)
public Object getLocalProperty (String name, String namespace)
```

Thus, to get the "name" property from the default namespace, you might use this code:

```
String name = (String)datum.getProperty ("name");
```

To get the "name" property from the "alt" namespace, you would do something like this:

```
String altName = (String)datum.getProperty ("name", "alt");
```

Here, the "name" property refers to two separate locations, and might contain completely different values, perhaps with completely different object types. You can change the value of "name" in "alt" without affecting "name" in the default namespace. Namespaces are the only way to assign more than one value to a property in a single Datum object. The XML for the example above might look something like this:

```
<object name="default-name" alt:name="alt-name"/>
```

The set of namespaces in a Datum is a dynamic, changing thing. As you set properties in new namespaces, the Datum will implicitly create a new namespace.

When you reset all properties in a namespace to null, removing them all, the Datum will discard the namespace. You can get a list of all current namespaces with the getNamespaces() method:

```
public String[] getNamespaces ()
```

The default namespace will show up in the namespace as the static data member Datum.DEFAULT_NAMESPACE.

An alternate version of getKeyNames() exists for retrieving properties from namespaces. No version exists for grabbing every property that exists in the Datum. To obtain such a list, you must get the list of namespaces with getNamespaces() and call getKeyNames() on each namespace.

```
public getKeyNames (String namespace)
```

Another method, findKeys(), can retrieve property names for a given namespace according to a regular expression pattern.

```
public String[] findKeys (String keyFilter, String namespace)
```

The keyFilter parameter holds the regular expression; only key names that match the expression will be returned. For example, given a Datum object with the keys "key", "key01", "otherprop", and "otherkey", the keyFilter ".*key" would match "key" and "otherkey". The filter "key[0-9]?[0-9]?" would match "key" and "key01".

The implementation of the regular expression parser determines the exact format allowed by the keyFilter parameter. If Umber Core cannot find a regular expression parser, the findKeys() method will throw an UnsupportedOperationException exception.

Section 2.3 - The ListDatum

To model XML documents, a Datum must be able to maintain parent-child relationships. The getParent() method links a Datum to its parent, but the Datum base class has no methods for adding children. A simple Datum cannot have children.

The ListDatum is a class derived from Datum which introduces a container for children. It adds JavaBean-style indexed accessors for the 'children' property:

```
public void setChildren (Datum[] children)
```

```
public void setChildren (int index, Datum child)
public Datum[] getChildren ()
public Datum getChildren (int index)
```

You can access or reassign the array of child Datum objects as a whole, or singly by index. Since Java arrays are of fixed length, using the above methods to add a new child, you would have to create a new slightly larger array, copy the old array into it, and set the new child. ListDatum offers convenience methods for adding a child Datum to the end of the current array, or to an arbitrary position in the child array. An invalid index in the latter version will trigger an IndexOutOfBoundsException.

```
public void addChildren (Datum child)
public void addChildren (Datum child, int position)
```

Likewise, it is possible to remove a child from its parent using only the setChildren() methods, but the removeChildren() convenience methods will do the dirty work for you. You can remove children by index or by object reference; if the index or object does not exist in the parent, the removeChildren() methods will return without error.

```
public void removeChildren (int index)
public void removeChildren (Datum child)
```

Both the addChildren() and removeChildren() methods also take care of approriately calling setParent() on the affected children. The parent of the removed child is always set to null.

One last final convenience method provides an easy way to determine the array index of a child, given its object reference. If the Datum child object does not exist within the parent, this method returns a value of -1. This method is not recursive, so if the child object exists as a grandchild, not an immediate child, findChildIndex() will return a -1.

```
public int findChildIndex (Datum child)
```

Section 2.4 - Text Content

Another Datum helper method is extractText(), which recursively grabs all text content from a given property inside the Datum and that property in all of its children, then concatenates it together into a single String. An optional collapseWhitespace parameter strips out all unnecessary whitespace characters,

collapsing them to single spaces.

```
public String extractText (String propertyName,
  String namespace, boolean collapseWhitespace)
```

This approach is particularly useful for dealing with XML PCDATA character content, which is stored in a special Datum property by SAXLoader (see the chapter on loading XML). To further facilitate PCDATA operations, Datum provides special methods for adding, extracting, and testing for PCDATA. The addPcdata() is only available on ListDatum, since PCDATA resides in child nodes. The other methods reside in the Datum class to make them easily accessible for all nodes, without requiring a downcast to ListDatum.

```
public static void addPcdata (Object pcdata)
public static String extractPcdata (boolean collapseWhitespace)
public static boolean hasPcdataOnly ()
```

The addPcdata() method appends a chunk of PCDATA to the given Datum node. The format of PCDATA content is described in further depth in the section on the SAXLoader class, but for now it is enough to mention that all PCDATA content is wrapped inside a specially named Datum object. The addPcdata() method takes care of creating and appending that wrapper object in a style compatible with SAXLoader.

Thus, given a Datum node roughly corresponding to the XML "<name/>":

```
ListDatum name = new ListDatum ("name");
```

the following invocation would change the XML to read "<name>Joe</name>":

```
name.addPcdata ("Joe");
```

A further invocation would simply append more text to the existing PCDATA content, resulting in the XML "<name>Joe Smith</name>":

```
name.addPcdata (" Smith");
```

Technically, the name object would contain two separate Datum child objects, one with the text "Joe" and the other with the text " Smith". You might also want to interweave PCDATA with element content:

```
ListDatum content = new ListDatum ("content");
content.addPcdata ("line one");
content.addChildren (new Datum ("br"));
```

```
content.addPcdata ("line two");
```

The code above would produce XML like this:

```
<content>line one
  <br/>line two
</content>
```

The extractPcdata() method performs the reverse operation, by recursively searching a Datum node for all PCDATA-style content, converting it to String data as necessary, and concatenating it all into a single returned String object. An optional collapseWhitespace parameter provides the same functionality as in the extractText() method.

The hasPcdataOnly() method evaluates a Datum object to see if it has only PCDATA content. Any non-PCDATA children will cause it to return false (like in the "content" example above), as will a non-ListDatum object, or a completely empty ListDatum object.

Section 2.5 - The TableDatum

The ListDatum expresses its children as a one-dimensional array; the TableDatum extends ListDatum to cover two dimensions, as a table grid. Each cell in the table is one Datum and can be accessed through getCell() and setCell():

```
public void setCell (int col, int row, Datum datum)
public Datum getCell (int col, int row)
```

Cells that haven't been set yet will yield a null from getCell(), even nonexistent or out of bounds cell coordinates. The table will expand and contract to match the largest current row and column. You can get the current width and height with the following methods:

```
public int getTableWidth ()
public int getTableHeight ()
```

TableDatum also supports an optional row of headers which can represent, for example, column labels. You can treat the headers just like ListDatum children:

```
public void setHeaders (Datum[] headers)
public void setHeaders (int index, Datum headers)
public Datum[] getHeaders ()
public Datum getHeaders (int index)
```

The headers do not affect the table height, but they do affect the width.

```
TableDatum table = new TableDatum (); // width = 0; height = 0

// no header
table.setCell (2, 4, new Datum ()); // width = 3; height = 5

// create header inside previous width
table.setHeaders (2, new Datum ()); // width = 3; height = 5

// create header outside previous width
table.setHeaders (5, new Datum ()); // width = 6; height = 5
```

The Bellows SAXLoader does not yet support loading XML into TableDatum objects, but a mapping for it may be implemented in the future.

Chapter 3 - Loading XML

Bellows implements its own class, SAXLoader, to load XML content into Datum trees, and wraps it in Reader and Writer classes to better integrate it with the java.io API. This chapter will explore the process of getting XML in and out of a Datum tree.

Section 3.1 - SAX Loading

The Datum is analogous to an XML element; this makes it easy to translate between XML documents and Datum trees. It is a simple matter to write a SAX XML parser to perform that mapping. The SAXLoader class has a load() method which takes a Reader pointing to the XML document:

```
public Datum load (java.io.Reader reader)
  throws UmberClassException, BellowsIOException, BellowsParseException
```

The load() method will throw one of three checked exceptions if it is unable to complete its SAX parsing run. It throws UmberClassException if it can't find a SAX parser implementation (this should never happen in JVM 1.4 and later), or BellowsIOException if a fatal error prevents it from reading from the XML document, or BellowsParseException if it encounters questionable XML. All three exceptions derive from UmberException, and can be caught with a single catch() block if desired. If no exceptions are thrown, the returned

Usage is simple:

```
SAXLoader loader = new SAXLoader ();

Reader reader = new FileReader (...);
try
{
  Datum xmlRoot = loader.load (reader);
}
catch (UmberException e)
{
  // report error...
}
```

The root XML element is returned directly from the load() method, and the rest of the XML document is available through the getChildren() methods of the top level Datum. In general, all XML elements are instantiated as ListDatum objects, even if they have no content, and PCDATA text content is wrapped in Datum objects, and accessible through the PCDATA_PROPERTY property defined in SAXLoader, or with the extractPcdata() method in Datum.

For example, given the following XML fragment:

```
<text style="bold">Text content</text>
```

we could load it with code like this:

```
SAXLoader loader = new SAXLoader ();

String xml = "<text style='bold'>Text content</text>";
StringReader reader = new StringReader (xml);
try
{
  Datum root = loader.load (reader);
}
catch (UmberException e)
{
  // report error...
}
```

This would place the <text> element in the root ListDatum, and the contents "Text content" in a single child Datum object. You can extract it like this (ignoring proper error checking for simplicity):

```
Datum child = root.getChildren (0);
Object pcdata = child.getLocalProperty (SAXLoader.PCDATA_PROPERTY);
```

Or, with the extractPcdata() method on the root object (passing in a true to collapse whitespace):

```
String pcdata2 = root.extractPcdata (true);
```

The difference in the whitespace is that the XML

```
<text>Text content</text>
```

would return the String "Text content" for both methods, while the multiline XML

```
<text>
   Text content</text>
```

would return something like " Text content " for the more literal getLocalProperty() method, and "Text content" for the collapsing extractPcdata() method.

If an XML document contains references to external files, you will need to supply a directory path to give the SAXLoader a point of reference for finding those files. This reference should take the form of a fully qualified URI for the original XML file, for example "file:///home/jsmith/stuff/data.xml". Simply pass this URI into the alternate load() method.

```
public Datum load (Reader reader, String baseURI)
  throws UmberClassException, BellowsIOException, BellowsParseException
```

If you have a File reference to the XML document you are loading, you can get the URI from the File object like this:

```
File xmlFile = ...;
String baseURI = xmlFile.toURL ().toString ();

SAXLoader loader = new SAXLoader ();

Reader reader = new FileReader (xmlFile);
try
{
  Datum xmlRoot = loader.load (reader, baseURI);
}
catch (UmberException e)
```

```
{
  // report error...
}
```

A third load() method accepts the XML document as a String file path. This method will set the baseURI for you, based on the absolute path of the XML file it is loading.

```
public Datum load (String fileName)
  throws UmberClassException, BellowsIOException, BellowsParseException
```

The SAXLoader stores XML attributes as Datum properties. Given the earlier <text> XML example, we can retrieve the "style" attribute as the "style" Datum property:

```
String style = (String)datum.getLocalProperty ("style");
// style now contains the string "bold"
```

All properties loaded by SAXLoader will be String objects; however, Datum properties are allowed to contain java.lang.Object values, so any Datum properties filled in by the application may or may not be Strings. When in doubt, call toString() on the property value or use Datum.getStringProperty() to extract String values.

To make life easier for the developer, the SAXLoader class does its best to find and load an appropriate XML parser. First, it attempts to use the XMLReader class specified by the user, if any. This can be passed into the alternate constructor as a fully qualified class String:

```
SAXLoader loader = new SAXLoader (
  "org.apache.crimson.parser.XMLReaderImpl");
```

The SAXLoader provides three common SAX parser class names as static properties:

```
SAXLoader.CRIMSON_READER
SAXLoader.XERCES_READER
SAXLoader.NANOXML_PARSER
```

If the given class is unavailable on the CLASSPATH or is not a proper XMLReader implementation, the loader will attempt to load a parser with the SAXParserFactory class. As long as a jar file for Xerces or Crimson is on the classpath (or bundled with the JVM in the case of Java 1.4), SAXLoader will find it for you.

Section 3.2 - The DatumReader

The SAXLoader class is the fundamental mechanism for loading XML documents into Datum trees. However, to better interface with the Java I/O API, Bellows offers some simple wrapper classes for Reader and Writer.

The DatumReader constructors each wrap an existing Reader object. You can optionally specify a SAX parser class, which the DatumReader will pass directly to the SAXLoader. The third constructor below provides a way to pass in a base URI (see the URI discussion in the SAXLoader section).

```
public DatumReader (Reader in)
public DatumReader (Reader in, String parserClass)
public DatumReader (Reader in, String parserClass, String baseURI)
```

The readXml() method pulls one XML document at a time from the Reader, throwing the same exceptions as SAXLoader.load(). However, unlike SAXLoader, it can handle more than one XML document from the same input stream. As long as the XML is valid and no I/O errors occur, you can keep calling readXml() until it returns null, creating a separate Datum tree for each invocation.

```
public Datum readXml ()
  throws UmberClassException, BellowsIOException, BellowsParseException
```

The following code reads the XML content "<root1/><root2/>" as two successive Datum objects. The third call to readXml() returns null and breaks out of the while() loop.

```
StringReader stringReader = new StringReader ("<root1/><root2/>");
DatumReader reader = new DatumReader (stringReader);

try
{
  Datum root = reader.readXml ();
  while (root != null)
  {
    // Process XML...

    root = reader.readXml ();
  }
}
catch (UmberException e)
```

```
{
  // report error...
}
```

Instantiating a Datum tree from a String is such a common task that DatumReader supplies a static convenience method to do just that:

```
public static Datum fromXml (String xml)
  throws UmberClassException, BellowsIOException, BellowsParseException
```

Like the SAXLoader.load() method it ultimately wraps, fromXml() will either throw an exception or return a Datum object; it will never return null. Also, if the input String contains more than one full XML document, fromXml() will only return the first one.

```
try
{
  Datum root = DatumReader.fromXml ("<root/>");
}
catch (UmberException e)
{
  // report error...
}
```

Incidentally, to work around a common assumption by SAX parsers that an input stream always consists of exactly one XML document, the DatumReader watches the data that passes through its read() methods and artificially mocks up an end-of-stream condition at the end of each document to fool the SAX parser into thinking the stream is completed. It then resets its internal counter for the next call to readXml() and continues from there. This introduces a small extra overhead for DatumReader, but opens up the possibility of such tasks as handling a series of XML documents over a socket, or reading multiple XML documents from the same file.

Section 3.3 - The DatumWriter

The DatumWriter class handles the reverse process, converting a Datum tree into an output stream. The usage is similar to loading with DatumReader. Pass the Datum tree into the wrapped Writer instance by calling the writeXml() method.

```
Datum root = new ListDatum ("root");
```

```
StringWriter stringWriter = new StringWriter ();
DatumWriter writer = new DatumWriter(stringWriter);

try
{
  writer.writeXml (root);
}
catch (IOException e)
{
  // report error...
}

String xml = stringWriter.toString ();
```

By default, DatumWriter writes all output as a single line, with no line feeds, using empty elements wherever it can, and converting special XML characters in PCDATA into the character entities <, >, &, and ". (NOTE: To better support the W3C XML conformance tests, it does not use the ' entity.)

The XML output format of a Datum tree can be customized with an XmlFormat object, which is assigned and altered with the get/setFormat() methods. This XmlFormat object will never be null. The setFormat() method will throw a NullPointerException to prevent that.

```
public XmlFormat getFormat ()
public void setFormat (XmlFormat format)
```

XmlFormat is a simple JavaBean with properties for all the public formatting options. The default constructor creates an object with the common defaults. The alternate constructor explicitly exposes all properties.

```
public XmlFormat ()
public XmlFormat (int indent, boolean collapse, boolean encodePcdata)
public XmlFormat (int indent, boolean collapse, boolean encodePcdata,
  boolean includePI, boolean includeNotations, boolean canonical)
```

The setIndent() method changes the indent level. A positive indent sets the number of spaces per nesting level; a zero indent aligns all content to the left, but leaves each element on its own line; a negative indent tells DatumWriter to wrap the entire document on a single line, with no line feeds.

```
public void setIndent (int indent)
```

You can also control whether or not the writer collapses elements without element or PCDATA content. By default, DatumWriter collapses these empty elements, e.g., "<child/>". If you pass a value of false to setCollapse(), the writer will always create explicit start and end elements, e.g., "<child></child>".

```
public void setCollapse (boolean collapse)
```

By default, DatumWriter encodes all special characters that occur in PCDATA content with the XML entities, <, >, &, and ". This protects the XML document from questionable markup characters that might sneak in via PCDATA nodes. To disable this feature, for example if you want to allow XML markup inside PCDATA Datum nodes, call the setEncodePcdata() method with a false.

```
public void setEncodePcdata (boolean encodePcdata)
```

Processing instructions are not included in the output by default, even if they are present in the Datum tree. To output them, you must pass a true to setIncludePI(). The SAXLoader must also be configured to load them.

```
public void setIncludePI (boolean includePI)
```

Notations, like PIs, are ignored by default, but can be output by enabling them in the writer's XmlFormat object.

```
public void setIncludeNotations (boolean includeNotations)
```

The canonical setting is not meant for common use, and is mainly included for better compliance in the W3C XML conformance tests. These tests expect XML output to take a very specific format, which is not likely to be the form you want your application to create. By default, canonical output is disabled.

```
public void setCanonical (boolean canonical)
```

The DatumWriter class offers a set of static convenience methods to directly convert a Datum tree into an XML String or send it to an arbitrary Writer:

```
public static String toXml (Datum datum)
public static String toXml (Datum datum, XmlFormat format)
public static void toXml (Datum datum, XmlFormat format, Writer out)
```

The DatumWriter example above simplifies to:

```
Datum root = new ListDatum ("root");
String xml = DatumWriter.toXml (root);
```

Finally, DatumWriter supplies a debugging method to make it easy to send XML-ized Datum content to debug output.

```
public static void dumpDatum (Datum datum)
```

Section 3.4 - The TextReader

A third Java I/O helper class collates ASCII text files into simple Datum trees. The TextReader places each page, delimited by the " " character, into a <page> element, and each line of text, delimited by the " " character, into a <line> element inside the <page> element. It places the entire contents inside a single <document> element.

For example, given the following example code:

```
StringReader stringReader =
  new StringReader("first line second line third line");
TextReader reader = new TextReader(stringReader);
Datum root = null;

try
{
  root = reader.load ();
}
catch (IOException e)
{
  // report error...
}
```

The TextReader would create this XML tree:

```
<document>
  <page>
    <line>first line</line>
    <line>second line</line>
  </page>
  <page>
    <line>third line</line>
  </page>
</document>
```

You can change the names of each element, or the page-feed character, with these

methods:

```
public void setDocumentName(String documentName)
public void setPageName(String pageName)
public void setLineName(String lineName)
public void setPageBreak (String pageBreak)
```

Chapter 4 - Loading HTML

Bellows also supports parsing and loading HTML content with the third-party TagSoup library. Through TagSoup, the Bellows HtmlReader class can handle many poorly formed HTML documents. This includes problems which would freak out a normal XML parser, such as dangling tags, overlapping (non-nested) markup, and even non-HTML documents.

The HtmlReader works similarly to DatumReader, except that it throws a BellowsLoaderException if it can't find the TagSoup library on the CLASSPATH. You can fall back on DatumReader in your application code in this case.

```
public HtmlReader(Reader in)
  throws BellowsLoaderException
public HtmlReader(Reader in, String baseURI)
  throws BellowsLoaderException
```

To read an HTML document from the input, call the readHtml() method. Like DatumReader.readXml(), this method will return null if parse or I/O errors occur while reading.

```
public Datum readHtml()
```

HtmlReader also has a convenience method for directly loading HTML from a String. Like the constructor, this method will throw an exception if it can't find the TagSoup library.

```
public static Datum fromHtml(String html)
  throws BellowsLoaderException
```

The reader will wrap plain text inside an <html> document for you. Thus, passing "first paragraph<p>second paragraph" to it:

```
Datum html = HtmlReader.fromHtml("first paragraph<p>second paragraph"");
```

will create a Datum tree that looks like this, with a properly closed <p> tag

after the second paragraph:

```
<html version="-//W3C//DTD HTML 4.01 Transitional//EN">
  <body>
      first paragraph
    <p>
       second paragraph</p>
  </body>
</html>
```

Since HtmlReader returns a normal Datum tree, you can use any of the other Bellows tools on it, including DatumWriter and DatumQuery.

Chapter 5 - The Bellows Query Language

The Bellows query engine helps manage complex Datum object trees by providing a simple query language loosely based on XML's XPath specification.

Section 5.1 - Simple Paths

The basic XML query looks like a filesystem path, consisting of element names separated by slash marks, e.g., "layout/document/section". However, while a filesystem path points to exactly one file or directory, a Bellows query path can point to multiple XML elements. The query is a pattern to match, not a unique location. Thus, given the following XML fragment, the query "layout/document/section" will match the first two <section> elements, but will ignore the unmatched <notsection> element and the out-of-place third <section> element:

```
<layout>
  <document>
    <section/>
    <section/>
    <notsection/>
  </document>
  <section/>
</layout>
```

The query language provides wildcards to match arbitrary children at one or more levels. The '*' wildcard matches all nodes in a single tree level; the '**' wildcard (referred to in this guide as the "multicard") matches all nodes in an

arbitrary number of tree levels. The wildcard can occur at the end of a path, while the multicard may only occur in the beginning or middle of a path. Thus, the multicard needs at least one trailing element to establish context in its search.

(NOTE: A trailing multicard may someday be implemented to return all leaf nodes of the current tree position, but since those results could be any element at any depth, the usefulness of such a result set is dubious and potentially dangerous.)

Thus, given the following XML sample:

```
<document>
  <preface>
    <text/>
  </preface>
  <body>
    <text/>
    <text/>
  </body>
  <text/>
</document>
```

the query "document/*/text" will return the first three <text> elements, but ignore the fourth <text> at a different level. To get all four <text> elements, you must use the '**' wildcard, e.g., "document/**/text". As this example illustrates, the '**' wildcard matches zero or more tree levels, while the '*' wildcard matches exactly one (i.e., not zero) tree level.

Each XML element above corresponds to a Datum or ListDatum object with the element name as its Datum type. Thus, the getType() method on the Datum corresponding to the <preface> element above would return the text "preface". The nested elements correspond to child Datum objects and could be accessed through the parent's getChildren() method. Queries simplify the process of "tunnelling" through Datum trees with a bunch of getChildren() and equals() methods.

Section 5.2 - Attribute Paths

Query paths may also contain an attribute value, for queries that resolve to text content or to Datum content inside a property. The attribute path looks like a regular element node, except it is prefixed with an ampersand character. In the following example, the query "document/&description" resolves to the text "First

Draft":

```
<document description="First Draft">
  <body>
    <text/>
    <text/>
  </body>
</document>
```

Because the query API must return results as Datum objects, the query wraps each matched attribute value inside a new Datum object, as PCDATA. Multiple attribute path matches are loaded into separate Datum objects. The attribute content can be accessed through the Datum's extractPcdata() method.

Although Datum trees loaded from XML documents (e.g., with SAXLoader, DatumReader, or TextReader) will always contain String data in their properties, other Datum trees created directly by Java code might have non-String data in their properties, including Datum and Datum[] instances. These last two cases are handled differently. Since these property contents are already in Datum form, they don't need to be wrapped. Thus, they can be returned directly. For example, the following code places a Datum object into the "prop" property, and to contrast, some String data into the "prop2" property.

```
ListDatum root = new ListDatum ("root");
ListDatum child = new ListDatum ("child");
root.setProperty ("prop", child);
root.setProperty ("prop2", "otherchild");
```

The query "root/&prop" would return the child Datum instance directly; conversely, the query "root/&prop2" would return the String "otherchild" as PCDATA wrapped inside a new Datum instance.

The attribute path query does not necessarily have to be the last node in the query. If the returned property is a Datum tree with further children, the query might continue into the Datum object(s) it finds in the property. For example, the code below sets up a child tree which contains two grandchild nodes. The child nodes are attached to a property on the root node, and are not part of the root's normal children.

```
ListDatum root = new ListDatum ("root");
ListDatum child = new ListDatum ("child");
ListDatum grandchild1 = new Datum ("grandchild");
```

```
ListDatum grandchild2 = new Datum ("grandchild");

child.addChildren (grandchild1);
child.addChildren (grandchild2);

root.setProperty ("prop", child);
```

In this case, the query "root/&prop/grandchild" would return both grandchild Datum objects. Since the root does not have any normal children, the query "root/*" would return zero results.

When an attribute path finds a Datum[] value, each non-null Datum in the array is added directly into the results. Thus, with the code below, the query "root/&prop" would return both child nodes as separate matches.

```
ListDatum root = new ListDatum ("root");
ListDatum child1 = new ListDatum ("child");
ListDatum child2 = new ListDatum ("child");
Datum[] nodes = new Datum[] { child1, child2 };
root.setProperty ("prop", nodes);
```

Note that the names of the Datum nodes stored directly in the property are never referenced in the query. Even though the queries above are traversing through the "child" nodes, the query accesses them through their property name, not their Datum type name.

Section 5.3 - Index Filters

The simple queries in the previous section returned all matching nodes. However, sometimes you only want one specific node out of the group.

Any node in a query can be qualified with a filter delimited by square brackets. The simplest form is the index filter, consisting of a number which specifies which element to match. Index filters are zero-based, so the first element is zero, the second is one, and so on, similar to Java arrays. An out-of-bounds array index fails to match anything and is quietly ignored.

To get the second text node inside the first <body> element in the example below, you could use the query "document/body/text[1]".

```
<document>
  <body>
    <text/>
```

```
    <text/>
  </body>
  <body>
    <text/>
  </body>
</document>
```

This example illustrates a potential problem with overly promiscuous query paths. At first sight, the query "document/body/text[2]" might seem to fail, since the "[2]" index asks for the third element, and the first <body> element only has two. However, the "document/body" part of the query matches both <body> elements. The "document/body/text" part resolves from the two body nodes into all three <text> nodes. Those results are then passed through the "[2]" filter, which returns the <text> element in the second <body> element. To restrict your results to the first <body> element, you would query "document/body[0]/text[2]" instead. This query will return no results, since "body[0]" only matches the first <body> element.

Bellows implements a special index filter extension that mimics arrays in the Python programming language. Negative array indices start counting from the end of the array, towards the front of the array. The filter "[-1]" always accesses the last element, "[-2]" matches the second to last, and so on. Thus, the query "document/body/text[2]" in the previous example is equivalent to the query "document/body/text[-1]". Both return the third <text> element.

Section 5.4 - Range Filters

Another Python-style filtering extension not available in the XPath standard is the colon-separated range. The simple indexes described in the previous section will always match exactly one element. With a range, however, you can specify more than one adjacent element.

A range looks like "[m:n]" where 'm' is the beginning index and 'n' is the ending index. The range match will include the element at 'm', but will stop just short of the 'n' element. For example, given a collection of ten <text> elements, the query "text[2:5]" will match elements 2, 3, and 4.

Both 'm' and 'n' are optional, and will default to the beginning and end of the array, respectively. Thus, the query "text[3:]" will match 7 elements: all elements from the fourth (index 3) to the end of the ten-element array (index 9). The query "text[:3]" will match the first three elements, i.e., up to but not including index 3. A range value of "[:]" is the same as the entire array, so

the queries "text[:]" and "text" are identical.

Negative array values also work with ranges, where -1 is the last element in the array, wherever that might be. Positive and negative indices can be mixed freely, although invalid ranges will fail to match anything. For example, the query "text[2:-4]" would return indices 2 through 5 from our ten-element example, and the query "text[-4:-2]" would return indices 6 and 7. The table of indexes below will make this clearer. The top row shows the positive indices of the ten <text> elements, and the bottom row shows the negative indices of those same elements.

```
Pos:   0  1  2  3  4  5  6  7  8  9
Neg: -10 -9 -8 -7 -6 -5 -4 -3 -2 -1
```

Section 5.5 - Attribute Filters

Query filters can also select matches according to the name or content of XML attributes (which correspond to Datum properties). Attribute names are declared with a prefix of '@'. Given the XML sample:

```
<document>
  <block type="paragraph"/>
  <block id="block1" type="paragraph"/>
  <block id="block2" type="quote"/>
</document>
```

We can test for the existence of an attribute with the form "[@name]". The query "document/block[@id]" would match the second and third <block> elements, all nodes that contain an "id" attribute; the query "document/block[@type]" would match all <block> elements, since all of them happen to have a "type" attribute.

Queries can also filter according to the value of the attributes. The query "document/block[@id='block1']" would return only the second <block> element. Attribute filters will return all matches, not just the first. The query "document/block[@type='paragraph']" would return the first two <block> elements, but not the third.

The single quotes around the attribute value are optional (by contrast, the official XPath specification requires them), so the query "document/block[@id=block1]" is also valid.

Section 5.6 - Chained filters

In another departure from the XPath notation style, the Bellows Query Language allows for more than one filter against the same set of nodes. The query applies each filter in the "chain" cumulatively to the node set in the order they occur in the query. Thus, the query "document/block[@id][@type=paragraph]" would first filter out all <block> nodes without an "id" attribute, then filter out all remaining <block> nodes that lack a "type" attribute that equals "paragraph". In the example from the previous section, this query would return the second <block> node:

```
<document>
  <block type="paragraph"/>
  <block id="block1" type="paragraph"/>
  <block id="block2" type="quote"/>
</document>
```

Changing the order of the filters in the query will change the order they are applied to the node list. Shuffling the order in the query above would not happen to affect the outcome, since those chained filters only test for the existence of static data. However, once you add index or range filters into the mix, the order in the chain can make a big difference in the results. For example, the query "document/block[0][@id]" when run against the XML above would return zero matches, since it will first narrow to the first <block> node. That node does not contain an "id" attribute, so the query fails to match anything. On the other hand, reversing the filters to read "document/block[@id][0]" will first grab the second and third <block> nodes, then match the first (index 0) node in that sub-set. That query will match against the second <block> node in the XML document.

Section 5.7 - Running Queries

Now that we know how to construct a query with the Bellows Query Language, we can introduce the DatumQuery class, which is the front end to the Bellows query engine. Usage is very simple. You create an (immutable) query object with a Datum tree and a query expression:

```
SAXLoader loader = new SAXLoader ();
ListDatum root = loader.load (xmlReader);
DatumQuery query = new DatumQuery (root, "root/child[1]");
```

Before running the query, you can make sure it's a valid expression with the isValid() method. Invalid queries will always return an empty array (never

null).

```
public boolean isValid()
```

Invoke the query with the run() method to get an array of all Datum objects that match the query pattern:

```
Datum[] results = query.run ();
```

Once you create a query object, you cannot change its query expression. If you need a new query expression, you'll have to create a new DatumQuery object. If the Datum tree does not change, run() will always return the same results, but if the tree does change, the run() will return the very latest results.

For queries you will only run once, you can use the static run() method, which does not require a DatumQuery instance:

```
public static Datum[] run (Datum datum, String query)
```

The QueryCache class can also act as wrapper to run multiple queries on a single Datum tree. The query results are cached in the cache object instance for better performance of repetitive queries. First, create a QueryCache object with your Datum tree:

```
QueryCache cache = new QueryCache (xmlRoot);
```

You can then run a series of queries against the xmlRoot tree with the runQuery() method:

```
Datum[] results1 = cache.runQuery ("root/child");
Datum[] results2 = cache.runQuery ("root/child[1]");
Datum[] results3 = cache.runQuery ("root/*");
Datum[] results4 = cache.runQuery ("/root/child[1]");
```

The results of these queries are cached in the QueryCache object, and will be returned the next time you call runQuery() with the exact same query. In the preceding code sample, results4 is the same object reference as results2, i.e., the Java expression (results2 == results4) evaluates to 'true'.

The results4 query does not go through the engine, but is retrieved from a data cache inside the QueryCache. Thus, the second call to the same query only takes as long as a cache lookup. The tradeoff to the improved performance is that if the Datum content changes in between cached queries, the runQuery() method will continue to return stale results. To clear out the cache and force fresh

queries, you must call the emptyCache() method.

```
public void emptyCache ()
```

Chapter 6 - Bellows Utilities

Section 6.1 - JavaBean To XML Conversions

Another helpful utility feature of Bellows is its support for translating between live JavaBean object data and Datum XML content. Bellows offers two utility classes for converting between JavaBeans and Datum trees: BeanToXml and XmlToBean.

The BeanToXml class takes an arbitrary Java object and recursively scans it for JavaBean properties. Each "simple" property (e.g., int, Integer, String) becomes a child element or attribute value inside a Datum tree created by BeanToXml. Non-simple JavaBean properties become nested elements which are in turn scanned for their properties.

The top level element for each JavaBean is the class name of that object, in whichever naming style the bean was converted with. The convertBean() method takes the JavaBean instance, the style for converting element names, and the collapse mode.

```
public Datum convertBean (Object bean,
    PropertyName.PropertyStyle style, boolean collapse)
```

Consider the following example class:

```
public class SimpleBean
{
  private String _str;
  private int _int;

public String getStringProp() { return _str; };
public void setStringProp(String strProp) { _str = strProp; }

  public int getIntegerProp() { return _int; };
  public void setIntegerProp(int intProp) { _int = intProp; }
}
```

The code below creates a SimpleBean object and loads it into a Datum tree:

```
SimpleBean bean = new SimpleBean();
bean.setStringProp("string value");
bean.setIntegerProp(14);

Datum xml = new BeanToXml().convertBean(bean,
  PropertyName.DEFAULT_STYLE, true);
```

DEFAULT_STYLE tells BeanToXml to create elements and attribute names in hyphen-separated lower case. The collapse of 'true' results in simple properties like String and int being created as attributes. The generated XML would look like this:

```
<simple-bean integer-prop="14" string-prop="string value"/>
```

Changing the style and collapse parameters will affect the XML output. To get uppercase underscored names and all properties as child elements, you would invoke convertBean() like this instead:

```
Datum xml = new BeanToXml().convertBean(bean,
  PropertyName.UPPER_UNDERSCORE_STYLE, false);
```

The XML would then be:

```
<SIMPLE_BEAN>
  <STRING_PROP>string value</STRING_PROP>
  <INTEGER_PROP>14</INTEGER_PROP>
</SIMPLE_BEAN>
```

Nested JavaBeans simply reside in child elements. Suppose the SimpleBean above is a property of a more complex NestedBean.

```
public class NestedBean
{
  private Boolean _bool;
  private SimpleBean _bean;

public Boolean getBooleanProp() { return _bool; };
public void setBooleanProp(Boolean boolProp) { _bool = boolProp; }

  public int getBean() { return _bean; };
  public void setBean(SimpleBean bean) { _bean = bean; }
}
```

Given the following code:

```
SimpleBean bean1 = new SimpleBean();
bean1.setStringProp("string value");
bean1.setIntegerProp(14);

NestedBean bean2 = new NestedBean();
bean2.setBooleanProp(Boolean.TRUE);
bean2.setBean(bean1);

Datum xml = new BeanToXml().convertBean(bean2,
  PropertyName.DEFAULT_STYLE, false);
```

If we created XML (with collapse set to false), we would get something like the example below. Note that the immediate child elements of <nested-bean> are property names, and the complex property values are expressed in further-nested child nodes with the property class as the element name, in this case <simple-bean>.

```
<nested-bean>
  <boolean-prop>true</boolean-prop>
  <bean>
    <simple-bean>
      <string-prop>string-value</string-prop>
      <integer-prop>14</integer-prop>
    </simple-bean>
  </bean>
</nested-bean>
```

If we instead run with collapse set to true, BeanToXml puts all simple properties, in this case booleanProp, as attributes in the bean element, and all "complex" properties in immediate child elements. Note that the complex bean property is named <bean> after the property name, not <simple-bean> after the class name.

```
<nested-bean boolean-prop="true">
  <bean integer-prop="14" string-prop="string value"/>
</nested-bean>
```

BeanToXml also handles array properties. For example, if SimpleBean instead had an array of int objects:

```
public class SimpleBean
{
```

```
  private String _str;
  private int[] _ints;

public String getStringProp() { return _str; };
public void setStringProp(String strProp) { _str = strProp; }

  public int[] getIntegerProps() { return _ints; };
  public void setIntegerProps(int[] intProps) { _ints = intProps; }
}
```

and given the following code:

```
SimpleBean bean = new SimpleBean();
bean.setStringProp("string value");
bean.setIntegerProps(new int[] { 10, 20, 30 });

Datum xml = new BeanToXml().convertBean(bean,
  PropertyName.DEFAULT_STYLE, true);
```

we would get this uncollapsed XML:

```
<simple-bean>
  <string-prop>string value</string-prop>
  <integer-props>
    <int>10</int>
    <int>20</int>
    <int>30</int>
  </integer-props>
</simple-bean>
```

The collapsed XML version would look like this:

```
<simple-bean string-prop="string value">
  <integer-props>10</integer-props>
  <integer-props>20</integer-props>
  <integer-props>30</integer-props>
</simple-bean>
```

In cases where the XML you want to create does not match the default output of BeanToXml, you can customize it on a per-class basis, by implementing the IXmlExporter interface on a given JavaBean class. The IXmlExporter interface contains a single method:

```
Datum toXml ()
```

As BeanToXml iterates through your JavaBean model, it first checks each node to see if it implements IXmlExporter. If so, it calls toXml() and inserts that result into the XML tree; if not, it falls back to its normal reflective XML creation.

Section 6.2 - XML to JavaBean Conversions

To go the opposite direction, from XML to JavaBean, use the XmlToBean class, which contains two convertXml() methods. Each method takes a loaded Datum tree and the Java class to use as the top level object instance. The first method takes a Class object, and the second method takes the fully qualified Java class as a String.

```
public Object convertXml (Datum xml, Class beanClass)
public Object convertXml (Datum xml, String beanClassName)
```

XmlToBean creates the top level object, scans its JavaBean properties, and attempts to pull data from the XML for each property it finds. It recurses into any non-simple properties it finds, instantiating objects of the proper class to match what it sees in the XML. Essentially, XmlToBean performs the same conversion that BeanToXml does, except in reverse. Given a JavaBean, you should be able to call BeanToXml.convertBean(), then call XmlToBean.convertXml() on the resulting Datum tree, and get an identical copy of your original bean -- assuming all of its data is stored in proper JavaBean properties.

Thus, given our original example:

```
SimpleBean bean = new SimpleBean();
bean.setStringProp("string value");
bean.setIntegerProp(14);

Datum xml = new BeanToXml().convertBean(bean,
  PropertyName.DEFAULT_STYLE, true);
```

We should be able to round trip like this:

```
SimpleBean beanCopy = (SimpleBean)new XmlToBean().convertXml(
  xml, SimpleBean.class);

String strProp = beanCopy.getStringProp(); // "string value"
int intProp = beanCopy.getIntegerProp();   // 14
```

Given that SimpleBean is defined in the my.bean.model package, the other convertXml() method would look like this:

```
SimpleBean beanCopy = (SimpleBean)new XmlToBean().convertXml(
  xml, "my.bean.model.SimpleBean");
```

XmlToBean will ignore all content that does not map to JavaBean properties in the provided Class. Because of this, the same XML document can polymorphically be loaded into more than one type of JavaBean instance by passing in different Java classes to convertXml().

Furthermore, XmlToBean is not particular which naming style the input XML uses. It can even handle a mixed style:

```
<simple_bean STRING-PROP="string value">
  <integerProp>14</integerProp>
</simple_bean>
```

If the bean class is null, or if it can't find or instantiate the given bean class (for example if the bean is not an accessible public class, or if the bean lacks a default constructor), the convertXml() method will return a null.

Section 6.3 - JavaBean To XML Property Filtering

By default, BeanToXml loads all official JavaBean properties in the bean and ignores all non-JavaBean methods. The BeanToXml converter allows more flexible customization of this with the BeanToXmlFilter helper class. This filter provides optional support for loading Collection properties and for excluding specific properties from the XML output, and offers a couple hook methods for applications to extend the filtering algorithm.

First, we'll look at collection properties. Normally, BeanToXml ignores JavaBean-style properties of type Collection, since aggregate JavaBean properties are supposed to use array semantics. However, for data models that do use Collections, BeanToXml has optional support for it, through the filter's collectionAllowed property:

```
public BeanToXmlFilter(boolean collectionAllowed)
public final boolean isCollectionAllowed()
public final void setCollectionAllowed(boolean collectionAllowed)
```

To turn this on, create a BeanToXmlFilter object with collections turned on:

```
BeanToXmlFilter filter = new BeanToXmlFilter (true);
```

Pass it to the alternate BeanToXml.convertBean() method:

```
public Datum convertBean(Object bean, PropertyStyle style,
    boolean collapse, BeanToXmlFilter filter)
```

BeanToXml loads the collection just like an array. So a MyObject.getMyCollection() method that returns a List of String and Integer objects might result in XML like this:

```
<my-object>
  <my-collection>
    <integer>123</integer>
    <integer>456</integer>
    <string>hello</string>
    <integer>789</integer>
  </my-collection>
</my-object>
```

If the collectionAllowed property is not explicitly set to true in the filter, BeanToXml will completely ignore all Collection properties.

Unfortunately, when Collection properties are turned on, BeanToXml and XmlToBean will no longer reliably work with round tripping. Since Collection properties give no clue as to which object types they might contain, and the generated XML does not contain fully qualified object classes, XmlToBean has no way to reconstitute Collection contents. With the <my-object> example above, XmlToBean could only assume the contents were java.lang.Integer and java.lang.String, but what if an application uses its own my.app.util.Integer class?

The base implementation of BeanToXmlFilter also contains support for ignoring (filtering out) JavaBean properties based on their Java class types or property names. The filter's skipTypes property controls the former and skipPropNames the latter.

```
public BeanToXmlFilter (boolean collectionAllowed,
    Class[] skipTypes, String[] skipPropNames)
public final Class[] getSkipTypes ()
public final void setSkipTypes (Class[] skipTypes)
public final String[] getSkipPropNames ()
public final void setSkipPropNames (String[] skipPropNames)
```

The skipTypes Class array tells the filter to ignore all properties that return a type castable to any skipTypes Class. This includes interfaces and parent classes, so for example including java.lang.Number in skipTypes will exclude all properties that return Integer, Double, etc.

The skipTypes array also affects Collection and array contents. A skipTypes of java.lang.String will exclude all String properties, plus all String[] properties, plus all String elements in any Collection properties (if they are enabled). The converter excludes the entire array property because all such arrays would always be empty, since the String contents would be excluded too. Note that since Collection properties can contain any object type, the filter will not exclude the entire Collection, even if all of its elements have been individually excluded.

The skipPropNames array contains all JavaBean property names you want to exclude from the XML output. These names should be the generic property name, not a method name. For example, the getMyProperty() method would translate to a property name of "myProperty". The converter currently does not take the object class into consideration for property name filtering, so "myProperty" will exclude that property from all JavaBeans it finds, regardless of the class. (A future version of Bellows may add that feature.)

Finally, BeanToXmlFilter provides two hooks, called by BeanToXml, where the application can install custom type and property name filtering:

```
public boolean allowPropertyType (Class propertyClass)
public boolean allowPropertyName (String propertyName)
```

To alter the filter's behavior, subclass BeanToXmlFilter and override one or both of these methods. If you want the skipTypes and skipPropNames arrays to work in your filter extension, don't forget to call the base class versions of the methods.

```
public boolean allowPropertyType (Class propertyClass)
{
  // Put overrides of skipTypes here...

// If base implementation want to skip a type, let it.
if (!super.allowPropertyType (propertyClass))
{
  return false;
```

```
}

  // Put additional skip checks here...
}
```

Section 6.4 - JavaBean Conversion Injection

As a further optional convenience, Bellows offers a simple base class to "inject" BeanToXml-style conversions into your object models. Simply derive your objects from the Bellows XmlAware class, and all objects will inherit a toXml() method which will generate a Datum tree for that object and its children. Call toXml() on the root node and get the entire object model in XML form; call it on a child node and get that branch of the model as XML.

By default, XmlAware generates XML with the DEFAULT_STYLE naming convention (lowercase hyphenated), with collapse set at false. You can change this for each node by calling an alternate constructor in your objects.

```
public XmlAware()
public XmlAware(PropertyName.PropertyStyle style, boolean collapse)
public XmlAware (PropertyName.PropertyStyle style, boolean collapse,
    BeanToXmlFilter filter)  public Datum toXml ()
```

Thus, to create uppercase hyphenated XML with collapsed attributes, you could declare your object's constructor like this:

```
public class MyObject extends XmlAware
{
  public MyObject()
  {
    super(PropertyName.UPPER_HYPHEN_STYLE, true);
  }
}
```

Turning on support for Collection accessors is as simple as:

```
public class MyObject extends XmlAware
{
  public MyObject()
  {
    super(PropertyName.UPPER_HYPHEN_STYLE, true,
      new BeanToXmlFilter (true));
```

```
    }
  }
```

You can change the format for all the objects in your model by overriding the properties in a common base class (e.g., derive all model classes from MyObject). Obviously this is only useful if have total control over your object model. Pre-existing object models can use BeanToXml directly to implement their own toXml() method, without forcing a change in the inheritance tree.

Section 6.5 - JavaBean Persistence

Bellows also provides special classes for persisting JavaBean instances with the official new XML JavaBean persistence format. The BeanLoader and BeanUnloader classes follow the same XML format as the XML JavaBean persistence engine in Java 1.4's java.beans.XMLEncoder class.

The BeanLoader API to convert Datum trees into JavaBean objects is simply:

```
public Object loadBean (Datum datum)
  throws BeanLoaderException
```

If the BeanLoader cannot instantiate the JavaBean class or any class it uses, it will throw a BeanLoaderException.

The BeanUnloader API does the reverse:

```
public Datum unloadBean (Object bean)
```

The XMLEncoder-style XML format embeds the fully qualified JavaBean classes into the XML content, which makes the XML completely self-contained. Unlike XmlToBean, you do not have to pass in the bean class as a separate parameter.

For more information about the XML format, see http://java.sun.com/products/jfc/tsc/articles/persistence3/.

Chapter 7 - Advanced Topics

Section 7.1 - The Traversal API

Bellows provides a generic traversal API for running arbitrary transformations and/or filtering against a collection of data objects. The traversal API was originally created to provide a unified front end to the variety of filtering operations in the Bellows query engine, but it is applicable to many operations

outside of Bellows. For example, it has been used extensively in the Catalan data transformation library.

The traversal API consists of an interface, NodeProcessor, and a concrete class, Traversal. The Traversal class implements an immutable object which manages the transformation of a List of Object references as it passes through a series of NodeProcessor implementations. The Traversal API is very simple:

```
public Traversal (List nodes)
public List traverse (NodeProcessor processor)
public List getNodes ()
```

The constructor accepts the input data. The traverse() method sends the entire data set through the supplied NodeProcessor; the processor can decide which nodes to process, which to reject, and which to pass through untouched. The getNodes() method returns a copy of the current processed collection of data nodes, at any time between traversals.

The Traversal is typically used like this:

```
List input = getInputData ();  // create input data
Traversal traversal = new Traversal (input);

NodeProcessor proc1 = ...
NodeProcessor proc2 = ...

traversal.traverse (proc1);
traversal.traverse (proc2);

List output = traversal.getNodes ();
```

The NodeProcessor interface consists of three methods, one to process a single node of data, and two to perform pre-processing initialization and post-processing cleanup:

```
List processNode (Object node)
void start (List nodes)
void end (List nodes)
```

The processNode() method is the heart of the processor. It acts as a pipeline for filtering, transforming, counting, or otherwise processing the input data, one node at a time. Each input data node is processed separately, although the processor implementation can manage state information throughout the entire

traversal of all input nodes.

The start() and end() methods provide hooks for that sort of thing. The driving Traversal always calls the start() method before it sends any data to the NodeProcessor, and always calls the end() method after it has processed the final input node. The Traversal passes in the full input data List into both start() and end(). The start() method can store a copy of it, or extract any global information it needs for processing, for example, the total node count. The end() method receives the actual output data node List just before the Traversal ends. Any changes it makes will directly affect the data returned to the Traversal.

For example, a search-and-replace processor might perform a simple text modification on all String nodes it sees, not needing to do anything in the start() or end() methods. It would pass the altered nodes out through the List return value of processNode(). On the other hand, a sorting processor might collect each input data node into an internal List and defer sorting until the end() method, when it can be sure it has all the nodes. The sorter would likely return an empty List from each processNode() method call, then add the sorted List as a whole in the end() method:

```
public void end (List nodes)
{
   nodes.clear ();
   nodes.addAll (_sortedNodes);
}
```

Section 7.2 - Document Validation

Aside from what support the XML parser implementations provide, Bellows does not currently support structural and content validation through DTDs, XML Schema, or other custom means. Future releases of Bellows will add these features. But for now, your best bet is to either rely on the XML parser to reject invalid documents, or use the TypesafeDatum wrapper to implement a custom per-Datum validation framework.

To wrap an existing Datum object with a typesafe front end, use the static wrapDatum() method. This allocates a new instance of TypesafeDatum which is itself a subclass of Datum. The typesafe wrapper transparently forwards all method calls into the original wrapped Datum, rejecting attempts to violate the wrapper's constraints. Thus, after wrapping a Datum, you can treat the wrapper as if it were the original.

```
public static TypesafeDatum wrapDatum (Datum inner)
```

The TypesafeDatum offers some basic support for declaring valid property names and value types. Any type violation will result in an IllegalArgumentException with a message describing the problem.

All type safety constraints must be set up explicitly, with the declareProperty() method:

```
public declareProperty (String propertyName, Class propertyType,
  String namespace)
```

The propertyName parameter is the name of the property accessed through get/setProperty(). The propertyType parameter restricts the property to a specific Java class or derivative. If you pass in null, the type will be set to java.lang.Object, which effectively declares a typeless property. The namespace can be the name of the namespace, or null, which maps to DEFAULT_NAMESPACE.

To demonstrate how this works, let's take the example of a Datum which can only have a String "name" property, an Integer "id" property, and a "value" property that can hold any value, all in the default namespace. We would create the Datum like this:

```
Datum datum = new Datum ();
TypesafeDatum wrapper = TypesafeDatum.wrapDatum (datum);
wrapper.declareProperty ("name", String.class, null);
wrapper.declareProperty ("id", Integer.class, null);
wrapper.declareProperty ("value", null, null);
```

These declarations are equivalent to the following:

```
wrapper.declareProperty ("name", String.class, Datum.DEFAULT_NAMESPACE);
wrapper.declareProperty ("id", Integer.class, Datum.DEFAULT_NAMESPACE);
wrapper.declareProperty ("value", Object.class, Datum.DEFAULT_NAMESPACE);
```

Any attempt to set a nonexistent or wrongly typed property will result in an exception:

```
// All of these throw IllegalArgumentException:
wrapper.setProperty ("nonexistent", "doesn't matter");
wrapper.setProperty ("name", new Float (2.4f));
wrapper.setProperty ("id", "423");
```

If the inner Datum contains data that violates the type safety, any attempts to retrieve the bad values will also trigger an exception.

Although Bellows doesn't yet support loading type constraints from DTDs or XML Schema, it does provide a copyTypes() method to transfer type info from one wrapper to another.

```
public copyTypes (TypesafeDatum datum)
```

You'll have to load the first one with declareProperty(), but you can stamp other TypesafeDatum objects with the same type. To load a new TypesafeDatum with the properties we declared above, we would do this:

```
// Wrap the new Datum just like before.
Datum datum2 = new Datum ();
TypesafeDatum wrapper2 = TypesafeDatum.wrapDatum (datum2);

// Load new wrapper type info from old wrapper.
wrapper2.copyTypes (wrapper);
```

If you're not sure which type a property is expecting, you can query with the getPropertyType() method:

```
public Class getPropertyType (String propertyName, String namespace)
```

TypesafeDatum does not yet understand the difference between required and optional properties. It assumes all properties are optional, and will not complain if any properties are missing, only when illegal properties and values are accessed. This will probably not be fixed in TypesafeDatum, but rather worked into the new Validator API in later releases of Bellows.

Section 7.3 - Customizing the SAXLoader

The default configuration of the SAXLoader class is targeted for the average user. By default, SAXLoader will load important whitespace, but will ignore typically unused whitespace, questionable Unicode content, XML processing instructions, and NOTATION declarations. However, sometimes an application needs access to these XML features. SAXLoader gives you control over them through four JavaBean properties.

The exact amount of whitespace saved by the SAXLoader depends on how strict its whitespace rules currently are. By default, the SAXLoader will discard whitespace in elements that contain no non-whitespace PCDATA. This extraneous

whitespace is typically found in indented XML documents. In normal usage, the indentation should be ignored, and if loaded would bloat the Datum tree with many unwanted PCDATA nodes. However, sometimes these blocks of whitespace are important. You can force the SAXLoader to include all of them by passing a value of true to the setStrictWhitespace() method:

```
public boolean isStrictWhitespace ()
public void setStrictWhitespace (boolean strictWhitespace)
```

Most XML documents will contain proper Unicode content, and thus will not need extra checking for it. Because this is the common case and since the process of running Unicode checks for an entire XML document is a potential performance drain, Unicode validation is turned off by default. If an XML document contains invalid Unicode, the questionable content will be loaded anyway, likely as garbage characters. Conversely, if Unicode checking is enabled, an invalid document will be rejected; the load() method will return null, even if the rest of the document is fine.

```
public boolean isUnicodeCheckEnabled ()
public void setUnicodeCheckEnabled (boolean unicodeCheckEnabled)
```

XML processing instructions (PIs) provide a way to embed application-specific instructions inside an XML document. These instructions can be completely ignored by other applications, and do not change the actual XML content. They supply domain-specific processing hints and semantics. Since PIs are not commonly used, they are ignored by default. To instruct the SAXLoader to include them in the Datum trees it creates, use the following methods:

```
public boolean isProcessingInstructionsEnabled ()
public void setProcessingInstructionsEnabled (boolean piEnabled)
```

Processing Instructions take the following form:

Each PI can have its own target name ("my-pi" in the example above) and data content ("pi-data" above). Since PIs can occur virtually anywhere in an XML document, alongside regular XML elements, and because position is important, the SAXLoader treats PIs like special Datum nodes (similar to the way it loads PCDATA). The Datum type and the properties used to store the PI name and value are each stored with a static SAXLoader field:

```
SAXLoader.PI_TYPE
SAXLoader.PI_TARGET_PROPERTY
```

```
SAXLoader.PI_DATA_PROPERTY
```

PIs can occur before and after the root XML node. Since this lies outside the root node, the SAXLoader must attach these special PIs as Datum properties on the root node:

```
SAXLoader.PI_BEFORE_ROOT
SAXLoader.PI_AFTER_ROOT
```

The PI_BEFORE_ROOT property contains a Datum[] array with all PIs that occur before the root node, or null if none are found there; similarly, PI_AFTER_ROOT contains a Datum[] array with all PIs after the root node, or null if none.

The XML NOTATION is used to declare a special format for unparsed content, and is often used in conjunction with PIs. They can be safely ignored, and are by default.

```
public boolean isNotationsEnabled ()
public void setNotationsEnabled (boolean notationsEnabled)
```

A NOTATION has a name and a SYSTEM or PUBLIC identifier. A notation can also declare both SYSTEM and PUBLIC identifiers, in which case it uses the PUBLIC keyword for both:

```
<!NOTATION my-note1 SYSTEM 'http://system-id.com'>
<!NOTATION my-note2 PUBLIC 'public-id'>
<!NOTATION my-note3 PUBLIC 'public-id' 'http://system-id.com'>
```

NOTATIONs can only occur inside the DOCTYPE section at the head of an XML document.

Please see the XML specification at http://www.w3.org/XML/ for more detailed information about the four properties described above.