GISCore User Guide

© The MITRE Corporation 2009.

The GIScore package provides the capability to perform streaming input and output of data from different file formats with an emphasis on GIS file formats such as ESRI Shapefiles or geo-databases (GDB) and Google Earth KML/KMZ. As time went on it was extended to include other record oriented formats that included GIS information such as GeoRSS and GeoAtom. Additionally it has proven useful to support some non-GIS formats such as Dbf and CSV. GIScore also provides the mediation between these file formats by converting each format to an internal normalized form.

GIScore was created to overcome the perceived problems in prior projects related to the use of in-memory representations of data. In-memory models have many advantages in terms of ease of use and speed, but lack the ability to deal with large data sets. GIScore tries to straddle both of these worlds by providing good performance with modest data set sizes while addressing the ability to deal with arbitrary data set sizes.

The ideal for GIScore was that the object model would be agnostic of the underlying file formats. This is at best unrealistic and factually seemed impossible. GIScore chooses to make the library representative of the richest set of the underlying libraries and ignore features of the object model when a library cannot represent a given feature for a specific implementation. This is one of a number of choices, but one that seemed best to the original author, and better than choosing only the common features of all.

# Getting Started

GIScore is built around a factory pattern. The user of a particular stream is meant to be ignorant of the implementation of the stream class that one is using. The caller unfortunately does need to know about extra arguments each document type requires, but not extra arguments based on the underlying class per se.

## Reading a Shapefile

Here is the code to instantiate a shapefile input stream taken from some actual code:

IGISInputStream sis = GISFactory.getInputStream(DocumentType.Shapefile,  
 ngr.getShapefile().getInputStream());

Some things to note: The user passes an input stream that contains a zip input stream holding a directory of shapefiles (which may have just a single shapefile). To switch to a FileGDB the type changes the contents of the zip input stream changes to a GDB directory. KMZ is a zip holding a KML file. There are some other factory methods available as well.

Next is a simple processing loop, taken from the same code. This particular loop looks for rings and polygons to pull out the rings to determine an area of interest to query.

IGISObject obj = sis.read();

while(obj != null)

{

if (obj instanceof Feature) {

Geometry geo = ((Feature) obj).getGeometry();

if (geo instanceof LinearRing) {

processRing(ngr, ((LinearRing) geo).getPoints());

} else if (geo instanceof Polygon) {

processRing(ngr, ((Polygon) geo).getOuterRing().getPoints());

} else if (geo instanceof MultiLinearRings) {

MultiLinearRings mlr = (MultiLinearRings) geo;

for(LinearRing ring : mlr.getLinearRings()) {

processRing(ngr, ring.getPoints());

}

} else if (geo instanceof MultiPolygons) {

MultiPolygons mlp = (MultiPolygons) geo;

for(Polygon poly : mlp.getPolygons()) {

processRing(ngr, poly.getOuterRing().getPoints());

}

} else if (geo instanceof Line) {

processRing(ngr, ((Line) geo).getPoints());

}

}

obj = sis.read();

}

The loop terminates when the read method returns a null value. This particular loop ignores the schema, but yours may want to look at data associated with the features. that's up to you.

Finally, the processing should terminate by closing the input stream, which cleans everything up:

sis.close();

### Writing a Shapefile

Writing any GIS output file is going to be more deterministic then reading in one. Why? Because when you're reading one in you don't really need to be concerned with required elements and structure. When you're writing one out you do.

You will generally be safe writing a GIS output file if you remember to first write the following elements:

* Schema
* DocumentStart(documenttype)
* ContainerStart("Folder")
* One or more features
* ContainerEnd

To start writing a shapefile you use the output factory:

IGISOutputStream shpos = GISFactory.getOutputStream(DocumentType.Shapefile, zos, outDir);

In addition to the zip output stream we have an output directory specified. This output directory is a scratch directory used to create the actual shapefiles before writing them out to the stream. Ideally we would be able to write the files directly as entries to the zip, but that doesn't work in practice, so we need to write them in the file system first in order to create the zip stream.

Writing to the stream is simple enough; we create the various events and write them to the stream.

Here's a test method that writes a simple point geometry:

public void testWriteReferencePointOutput(File shapeOutputDir)

throws Exception {

FileOutputStream zip = new FileOutputStream(new File(shapeOutputDir,

"reference.zip"));

ZipOutputStream zos = new ZipOutputStream(zip);

File outDir = new File("testOutput/shptest/buf");

outDir.mkdirs();

IGISOutputStream shpos =

GISFactory.getOutputStream(DocumentType.Shapefile, zos, outDir);

Schema schema = new Schema(new URI("urn:test"));

SimpleField id = new SimpleField("testid");

id.setLength(10);

schema.put(id);

DocumentStart ds = new DocumentStart(DocumentType.Shapefile);

shpos.write(ds);

ContainerStart cs = new ContainerStart("Folder");

cs.setName("aaa");

shpos.write(cs);

shpos.write(schema);

for(int i = 0; i < 5; i++) {

Feature f = new Feature();

f.putData(id, "id " + i);

f.setSchema(schema.getId());

double lat = 40.0 + (5.0 \* RandomUtils.nextDouble());

double lon = 40.0 + (5.0 \* RandomUtils.nextDouble());

Point point = new Point(lat, lon);

f.setGeometry(point);

shpos.write(f);

}

shpos.close();

zos.flush();

zos.close();

}

## KML

Google Earth data also known as KML data can likewise be created with *GISFactory* for most basic KML needs. Elements such as *Placemark*, *GroundOverlay*, *NetworkLink*, *Point*, *LineString*, *Polygon*, *IconStyle*, *ListStyle*, *Schema*, etc. are supported in addition to basic support for Google's gx: KML extensions.

IGISInputStream kis = GISFactory.getInputStream(DocumentType.KML, is)

IGISOutputStream kos = GISFactory.getOutputStream(DocumentType.KML, os)

Example to create a KML file with a single Placemark:

OutputStream fos = new FileOutputStream(“placemark.kml”);

IGISOutputStream os = GISFactory.getOutputStream(DocumentType.KML, fs);

// same as calling: new KmlOutputStream(fs)

Feature f = new Feature();

f.setGeometry(new Point(37.422069, -122.087461));

os.write(f);

os.close();

Support for creating KMZ output streams need to create *KmzOutputStream* object explicitly and use addEntry() to add files as entries to the KMZ (ZIP) output stream.

KmzOutputStream kmzos = new KmzOutputStream(new FileOutputStream(file));

// write out KML content which gets written to doc.kml as first entry of KMZ

GroundOverlay g = new GroundOverlay();

TaggedMap icon = new TaggedMap("Icon");

icon.put("href", "images/etna.jpg");

g.setIcon(icon);

kmzos.write(g);

// add image entry to KMZ file

File file = new File("data/kml/GroundOverlay/etna.jpg");

kmzos.addEntry(new FileInputStream(file), "images/etna.jpg");

kmzos.close();

The KML input stream reader supports the most commonly used features of the various KML specifications and schemas especially the core geospatial and temporal data, which is preserved in a “common” normalized representation. The normalized object representation, which can be created programmatically or imported from existing sources, can be exported to standard formats (or user-defined ones).

While most deprecated and deleted features from older KML specs (i.e., KML 2.0 and KML 2.1) are implemented to support the importing of legacy KML data, all KML output conforms to the KML 2.2 schema so some deprecated/deleted features may not be preserved on output without some manual intervention.

For example: a 'parent' element or attribute appearing in the Schema element is such a legacy feature. This was a non-XML Schema compliant mechanism introduced in KML 2.0 to alias KML features with user-defined element names with user-defined child elements. This is handled in a limited fashion in import where aliased elements are converted to Placemarks to be valid KML but the metadata are not auto-converted to *ExtendedData* fields. Such KML files are typically examples.

If more than basic KML or KMZ handling is needed then use the KmlReader and KmlWriter classes, which are wrappers for the KmlInputStream and KmlOutputStream classes respectively, and add a lot of special handling. The KmlReader class transparently handles reading KML and compressed KMZ files by file or URL along with fetching all NetworkLinks. Likewise, the KmlWriter class handles the creation of KML or KMZ files and optionally allows adding other files as entries in the KMZ (ZIP) file. Most importantly the KmlReader class rewrites the relative URLs so they can be traced back to the correct URL and the corresponding resource fetched that would normally be tricky for nested KML/KMZ files and resources referenced within a KMZ file. All versions of KML specifications are imported to avoid the need to convert older 2.0 and 2.1 KML documents to comply with the latest OGC KML 2.2 specification. The API takes care of the dirty details and conversions required.

### Writing KML with gx: extensions

Here is KML with elements in the Google extension namespace marked with the *gx:* prefix. In this example is a gx:Track element representing a single entity with multiple time-tagged locations each with a <when> element and a corresponding <gx:coord> element.

**<kml** xmlns="http://www.opengis.net/kml/2.2" xmlns:gx="http://www.google.com/kml/ext/2.2"**>**

**<Placemark>**

**<gx:Track>**

**<when>**2010-05-28T02:02:09Z**</when>**

**<when>**2010-05-28T02:02:56Z**</when>**

**<gx:coord>**-122.207881 37.371915 156.0**</gx:coord>**

**<gx:coord>**-122.203207 37.374857 140.2**</gx:coord>**

**</gx:Track>**

**</Placemark>**

**</kml>**

### Here are the few lines of Java code using GIScore library to generate the above KML:

ByteArrayOutputStream bos = new ByteArrayOutputStream();

KmlOutputStream kos = new KmlOutputStream(bos);

DocumentStart ds = new DocumentStart(DocumentType.KML);

Namespace gxNs = Namespace.getNamespace("gx", IKml.NS\_GOOGLE\_KML\_EXT);

ds.getNamespaces().add(gxNs);

kos.write(ds);

Feature f = new Feature();

f.setName("track");

Element gxElt = new Element(gxNs, "Track");

List<Element> elts = gxElt.getChildren();

elts.add(new Element("when").withText("2010-05-28T02:02:09Z"));

elts.add(new Element("when").withText("2010-05-28T02:02:56Z"));

elts.add(new Element(gxNs, "coord").withText("-122.207881 37.371915 156.0"));

elts.add(new Element(gxNs, "coord").withText("-122.203207 37.374857 140.2"));

f.addElement(gxElt);

kos.write(f);

kos.close();

### Reading a KML file

The following snippet of code uses convenience methods to fetch all features from a given KML resource then recursively load features from NetworkLinks. This is fine in most cases when it is known that the number of features and network links is relatively small and fits into memory. However, if the number of elements may be very large, it might be necessary to process items one at a time with a strategy to abort on user input or some other mechanism to allow the process to be interrupted such as shown in the second example.

File file = new File(" placemarks.kmz");

KmlReader reader = new KmlReader(file);

List<IGISObject> features = reader.readAll();

List<IGISObject> linkedFeatures = reader.importFromNetworkLinks();

List<URI> networkLinks = reader.getNetworkLinks();

This second example loads a KML resource which includes NetworkLinks. This example uses a callback *ImportEventHandler* to handle each of the features of the imported features. If the callback *handleEvent()* method returns false then recursion is aborted and no more *NetworkLink* features are processed.

URL url = new URL(

“http://kml-samples.googlecode.com/svn/trunk/kml/NetworkLink/visibility.kml”);

KmlReader reader = new KmlReader(url);

for (IGISObject gisObj; (gisObj = reader.read()) != null; ) {

if (gisObj instanceOf Feature) {

checkFeature(gisObj);

}

}

// now import features from any network links

reader.importFromNetworkLinks(

new KmlReader.ImportEventHandler() {

public boolean handleEvent(UrlRef ref, IGISObject gisObj)

{

checkFeature(gisObj);

// return false to abort the recursive network link parsing

return true;

}

});

## Reference

For better or worse the giscore factory methods uses variable argument lists. This makes it hard to understand the calling sequences. The following documentation explains the calling sequence for each document type. For each of the following, the call is assumed to be of the form:

GISFactory.getInputStream(DocumentType.<TYPE>,<inputStream>,arguments...)

for the Input cases below or

GISFactory.getOutputStream(DocumentType.<TYPE>,<outputStream>,arguments...)

for the Output cases below

Where <TYPE> is one the types in the list below, and <inputStream> is an appropriate input stream as documented or <outputStream> is an appropriate outputStream as documented.

The FileGDB implementation uses the ESRI library for support and only works on Windows at this point. The other implementations are written in Java and will work on any platform as far as we know.

Arguments are marked (\*) for required, or unmarked if not required.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Document Type | Input Stream | Input Arguments | Output Stream | Output Arguments |
| Shapefile | ZipInputStream | IAcceptSchema | ZipOutputStream | File(\*) IContainerNameStrategy PointShapeMapper |
| FileGDB | ZipInputStream | IAcceptSchema | ZipOutputStream | File(\*) IContainerNameStrategy |
| CSV | InputStream | Schema  String Character  Character | OutputStream | String  Character  Character  Boolean |
| KML/KMZ | InputStream | n/a | OutputStream | String encoding |
| XMLGDB | n/a | n/a | OutputStream | n/a |
| GeoAtom | InputStream | n/a | OutputStream | n/a |

There are also input factory methods to read some document types as files for both convenience and efficiency reasons. It hardly makes sense to zip up a shapefile, filegdb or other GIS file just to read it in. The general outline looks like this:

IGISInputStream stream = GISFactory.getInputStream(DocumentType.<TYPE>,   
 File <dirOrFile>, <varargs>);

The following table describes the file argument and *varargs* taken for each supported type:

|  |  |  |
| --- | --- | --- |
| DocumentType | File | Arguments |
| KML | KML File | n/a |
| Shapefile | Directory holding shapefiles | IAcceptSchema |
| FileGDB | Directory holding GDB | IAcceptSchema |
| CSV | CSV File | Schema String Character Character |

Note *KmlReader* and *KmlWriter* are advanced helper classes that are not accessible via the GISFactory and must explicitly be instantiated. Likewise, GeoRSS output writer is not yet added to the Factory method described above.

For the most part, GISCore abstracts most implementation details of the specific GIS file formats and user only need to work with simple object model of a Feature.