GeoTools

Feature Workbook

FOSS4G 2009 Geospatial for Java Tutorials

27 September 2009

Jody Garnett
Michael Bedward

Table of Contents

1 Welcome

Welcome to Geospatial for Java -this workbook is aimed at Java developers who are new to geospatial and would like to get started.

Welcome to FOSS4G! Please grab a buddy to go through this workbook.

You should of completed either the GeoTools NetBeans Quickstart or the GeoTools Eclipse Quickstart prior to running through this workbench. We need to be sure that you have an environment to work in with GeoTools jars and all their dependencies. For those using maven we will start off each section with the dependencies required.

This workbook features a new “code first” approach – we have made every effort to make these examples both visual and code centered. We have included some background materials explaining the concepts and ideas in case you are interested.

This workbook is part of the FOSS4G 2009 conference proceedings.

Jody Garnett

Jody Garnett is the lead architect for the uDig project; and on the steering committee for GeoTools; GeoServer and uDig. Taking the roll of geospatial consultant a bit too literally Jody has presented workshops and training courses in every continent (except Antarctica). Jody Garnett is an employee of LISAsoft.

Michael Bedward

Michael Bedward is a researcher with the NSW Department of Environment and Climate Change and an active contributor to the GeoTools users' list. He has a particularly wide grasp of all the possible mistakes one can make using GeoTools.

2 CSV2SHP

We are trying a new track for introducing features this year; rather then reading through a shapefile and ripping things apart in an artificial exercise, we are going to start by building a shapefile from scratch so you get to see every last thing that goes into creating features.

The tutorial covers the following:

- Creating a FeatureType, FeatureCollection and Features
- Using a GeometryFactory to build Points
- Writing out a Shapefile
- Forcing a Projection

At the end of the tutorial you will be able to create your own custom shapefiles!

1. To start with you will need a CSV file. Create a text file **location.csv** and copy and paste the following locations into it:

```
"Longitude", "Latitude", "Name"
-33.84,  151.26, Sydney
0, 52, London
-123.31,  48.4,  Victoria
```

   Feel free to add other locations to the file.

2. Please ensure your pom.xml includes the following:

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.geotools</groupId>
    <artifactId>gt-shapefile</artifactId>
    <version>${geotools.version}</version>
  </dependency>
  <dependency>
    <groupId>org.geotools</groupId>
    <artifactId>gt-epsg-hsql</artifactId>
    <version>${geotools.version}</version>
  </dependency>
  <dependency>
    <groupId>org.geotools</groupId>
    <artifactId>gt-swing</artifactId>
    <version>${geotools.version}</version>
  </dependency>
</dependencies>
```

   Note that the jars mentioned above will pull in a host of other dependencies (such as the hsql database driver).

3. Create **Csv2Shape.java** and copy and paste in the following code in order to get started.

```
package org.geotools.demo;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.Serializable;
import java.util.HashMap;
import java.util.Map;

import org.geotools.data.DataStoreFactorySpi;
import org.geotools.data.DataUtilities;
import org.geotools.data.DefaultTransaction;
import org.geotools.data.FeatureStore;
import org.geotools.data.Transaction;
import org.geotools.data.shapefile.ShapefileDataStore;
import org.geotools.data.shapefile.ShapefileDataStoreFactory;
import org.geotools.feature.FeatureCollection;
import org.geotools.feature.FeatureCollections;
import org.geotools.feature.simple.SimpleFeatureBuilder;
import org.geotools.feature.simple.SimpleFeatureTypeBuilder;
import org.geotools.geometry.jts.JTSFactoryFinder;
import org.geotools.referencing.crs.DefaultGeographicCRS;
import org.geotools.swing.data.JFileDataStoreChooser;
import org.opengis.feature.simple.SimpleFeature;
import org.opengis.feature.simple.SimpleFeatureType;

import com.vividsolutions.jts.geom.Coordinate;
import com.vividsolutions.jts.geom.GeometryFactory;
import com.vividsolutions.jts.geom.Point;

/**
 * This example reads data for point locations and associated attributes from
 * a comma separated text (CSV) file and exports them as a new shapefile. It
 * illustrates how to build a feature type.
 * <p>
 * Note: to keep things simple in the code below the input file should not have
 * additional spaces or tabs between fields.
 */
public class Csv2Shape {

}
```

4. We can now create our main method – to start with we will ask the user to provide a CSV file.

```
public static void main(String[] args) throws Exception {

    File file = JFileDataStoreChooser.showOpenFile("csv", null);
    if (file == null) {
        return;
    }
    // insert step 5.
}
```

Now we look at the rest of the main method in sections...

5. We create a FeatureType to describe the data the we are importing from the CSV file. Here we use the DataUtilities convenience class:

```
    /*
     * We use the DataUtilities class to create a FeatureType that
     * will describe the data in our shapefile.
     *
     * See also the createFeatureType method below for another,
     * more flexible approach.
     */
    final SimpleFeatureType TYPE = DataUtilities.createType(
            "Location",                     // <- the name for our feature type
            "location:Point:srid=4326," +   // <- the geometry attribute: Point type
            "name:String"                   // <- a String attribute
            );
```

6. We can now read the CSV File into a FeatureCollection; please note the following:

- Use of FeatureCollections.newCollection() to create a FeatureCollection
- Use of GeometryFactory to create new Points
- Creation of features (SimpleFeature objects) using SimpleFeatureBuilder

```
/*
 * We create a FeatureCollection into which we will put
 * each Feature created from a record in the input csv data file
 */
FeatureCollection<SimpleFeatureType, SimpleFeature> collection =
        FeatureCollections.newCollection();

/*
 * GeometryFactory will be used to create the geometry attribute
 * of each feature (a Point object for the location)
 */
GeometryFactory geometryFactory = JTSFactoryFinder.getGeometryFactory(null);

SimpleFeatureBuilder featureBuilder = new SimpleFeatureBuilder(TYPE);

BufferedReader reader = new BufferedReader(new FileReader(file));
try {
    /* First line of the data file is the header */
    String line = reader.readLine();
    System.out.println("Header: " + line);

    for (line = reader.readLine(); line != null; line = reader.readLine()) {
        String tokens[] = line.split("\\,");

        double longitude = Double.parseDouble(tokens[0]);
        double latitude = Double.parseDouble(tokens[1]);
        String name = tokens[2].trim();

        /* Longitude (= x coord) first ! */
        Point point = geometryFactory.createPoint(new Coordinate(longitude, latitude));

        featureBuilder.add(point);
        featureBuilder.add(name);
        SimpleFeature feature = featureBuilder.buildFeature(null);
        collection.add(feature);
    }

} finally {
    reader.close();
}
```

7. Next we are going to create a new shapefile to hold the features that we've just built. Things to note:

- Use of DataStoreFactory with a parameter to indicate that we want a spatial index
- The createSchema(SimpleFeatureType) method to set up the shapefile
- Our SimpleFeatureType did not include a map projection (coordinate reference system) which is needed to make a .prj file, so we call the forceSchemaCRS method to do this

```
    /*
     * Get an output file name and create the new shapefile
     */
    File newFile = getNewShapeFile(file);

    DataStoreFactorySpi dataStoreFactory = new ShapefileDataStoreFactory();

    Map<String, Serializable> params = new HashMap<String, Serializable>();
    params.put("url", newFile.toURI().toURL());
    params.put("create spatial index", Boolean.TRUE);

    ShapefileDataStore newDataStore = (ShapefileDataStore) dataStoreFactory.createNewDataStore(params);
    newDataStore.createSchema(TYPE);
    newDataStore.forceSchemaCRS(DefaultGeographicCRS.WGS84);
```

8. Here we use a Transaction to safely add the FeatureCollection to the shapefile in one go:

```
    /*
     * Write the features to the shapefile
     */
    Transaction transaction = new DefaultTransaction("create");

    String typeName = newDataStore.getTypeNames()[0];
    FeatureStore<SimpleFeatureType, SimpleFeature> featureStore =
            (FeatureStore<SimpleFeatureType, SimpleFeature>) newDataStore.getFeatureSource(typeName);

    featureStore.setTransaction(transaction);
    try {
        featureStore.addFeatures(collection);
        transaction.commit();

    } catch (Exception problem) {
        problem.printStackTrace();
        transaction.rollback();

    } finally {
        transaction.close();
    }

    System.exit(0);
}
```

This completes the main method; we have one more method to go.

9. This method prompts the user for a name for the output shapefile. The original CSV file is used to determine a good default name.

```
private static File getNewShapeFile(File csvFile) {
    String path = csvFile.getAbsolutePath();
    String newPath = path.substring(0, path.length() - 4) + ".shp";

    JFileDataStoreChooser chooser = new JFileDataStoreChooser("shp");
    chooser.setDialogTitle("Save shapefile");
    chooser.setSelectedFile(new File(newPath));

    int returnVal = chooser.showSaveDialog(null);

    if (returnVal != JFileDataStoreChooser.APPROVE_OPTION) {
        // the user cancelled the dialog
        System.exit(0);
    }

    File newFile = chooser.getSelectedFile();
    if (newFile.equals(csvFile)) {
        System.out.println("Error: cannot replace " + csvFile);
        System.exit(0);
    }

    return newFile;
}
```

10. That's it – the only thing left to do is run the application.

You might like to see if you can view the new shapefile using the Quickstart application !

2.1 Running the Application

Run the application using your IDE:

1. When you run this application it will prompt you for the location of a CSV file to read.
2. And then the name of a shapefile to create.

You can use the Quickstart you made in earlier workbook to display the shapefile that you just created. You could also try modifying the Quickstart to show your cities on top of another layer such as countries.

2.2 FeatureTypeBuilder

The DataUtilities class used above provided a quick and easy method to build a SimpleFeatureType, for most applications you will want to take advantage of the more flexible SimpleFeatureTypeBuilder.

1. Here is how to use SimpleFeatureTypeBuilder to accomplish the same result:

```
/**
 * Here is how you can use a SimpleFeatureType builder to create the schema
 * for your shapefile dynamically.
 * <p>
 * This method is an improvement on the code used in the main method above
 * (where we used DataUtilities.createFeatureType) because we can set a
 * Coordinate Reference System for the FeatureType and a a maximum field
 * length for the 'name' field
 * dddd
 */
private static SimpleFeatureType createFeatureType() {

    SimpleFeatureTypeBuilder builder = new SimpleFeatureTypeBuilder();
    builder.setName("Location");
    builder.setCRS(DefaultGeographicCRS.WGS84); // <- Coordinate reference system

    // add attributes in order
    builder.add("Location", Point.class);
    builder.length(15).add("Name", String.class); // <- 15 chars width for name field

    // build the type
    final SimpleFeatureType LOCATION = builder.buildFeatureType();

    return LOCATION;
}
```

Now our SimpleFeatureType contains a CoordinateReferenceSystem so there's no need to call forceSchemaCRS to generate the ".prj" file. Also, we are now limiting the Name field to 15 characters.

3 Things to Try

Here are a couple ideas to try out:

- Modify the CSV2SHP code to read the feature attribute names from the data file header rather than hard-coding them in the application.

```
"Longitude", "Latitude", "Name"
```

  You should be able to use SimpleFeatureTypeBuilder.

- Use the Geometry "buffer" method to create circles based on the population size of the each city.
- It is easy to write a quick CSVReader as we have done here; but harder to write a good one that can handle quotation marks correctly. JavaCSV is an open source library to read CSV files with a variety of options.
- To quickly find dependencies you can use the website http://mvnrepository.com/.

  Sites like this will directly provide you a maven dependency that you can cut and paste into your pom.xml.

```
<dependency>
    <groupId>net.sourceforge.javacsv</groupId>
    <artifactId>javacsv</artifactId>
    <version>2.0</version>
</dependency>
```

  For a working example of how to use this library visit the http://www.csvreader.com/ website.

- The earth has just passed through a meteor storm – generate 100 circles of different sizes across the globe. Was your town hit?

  Generating a shapefile from a model or analysis is a common use.

- Read up about the other Geometry classes supported by shapefiles: MultiLineString for linear features and MultiPolygon for areal features and modify this example to work with these.

4 Feature

You can also draw ideas like urban growth or predicted rain fall.

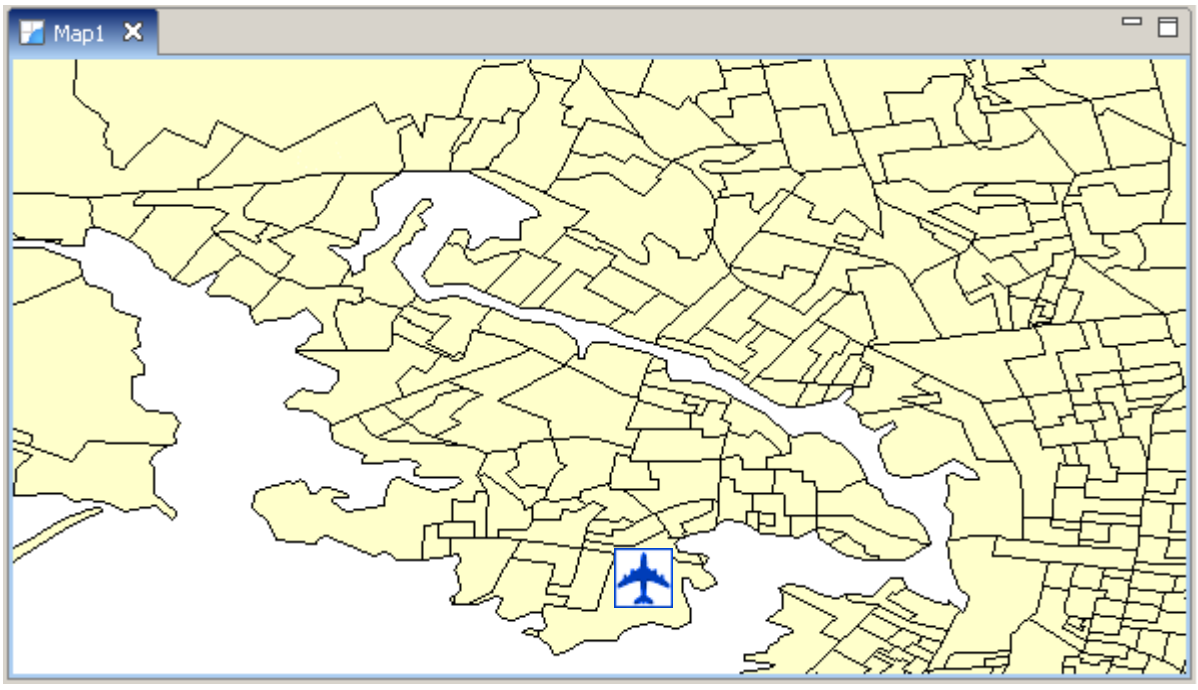
A feature is quite simply something that can be drawn on a map. The strict definition is that a feature is something in the real world – a feature of the landscape - Mt Everest, the Eiffel Tower, or even your great aunt Alice.

Explaining the concept to Java developers is easy - a feature is an Object.

Like a java object features can contain some information about the real world thing that they represent. This information is organized into attributes just as in Java information is slotted into fields.

You can see the "feature types" for a map listed in the map key.

Occasionally you have two features that have a lot in common. You may have the LAX airport in Los Angeles and the SYD airport in Sydney. Because these two features have a couple of things in common it is nice to group them together - in Java we would create a Class called Airport. On a map we will create a Feature Type called Airport.



Although it is not a capability supported by Java early programming languages made use of a prototype system (rather than a class system) that supported lots of "one off" Objects. You will find this situation is fairly common when making maps – since how many Eiffel towers are there? You

will also occasionally find the same real world thing represented a couple of different ways (the Eiffel tower can be a landmark or a tower depending on what you are talking about).

Here is a handy cheat sheet:

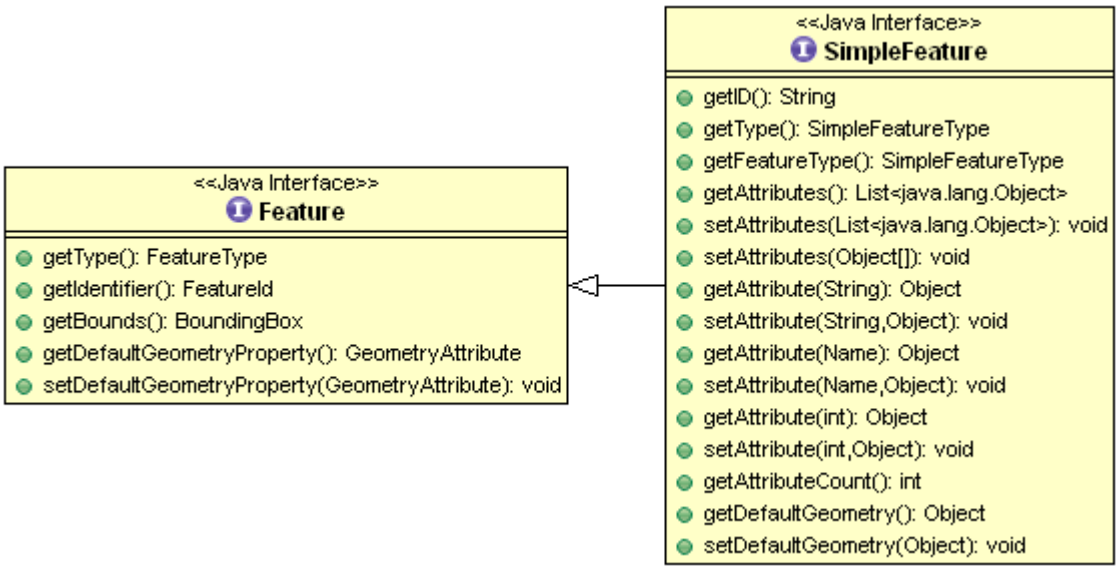
Java	**Geospatial**
Object	Feature
Class	FeatureType
Field	Attribute
Method	Operation

The Feature model is actually a little bit more crazy then us Java programmers are used to since it considers both attribute and operation to be “properties” of a Feature. Perhaps when Java gets closures we can catch up.

The really interesting thing for me is that map makers were sorting out all this stuff back in the 1400s and got every bit as geeky as programmers do now. So although we would love to teach them about object oriented programing they already have a rich set of ideas to describe the world. On the bright side, map makers are starting to use UML diagrams.

4.1 Feature Class

In **GeoTools** we have an interface for Feature, FeatureType and Attribute provided by the **GeoAPI** project. In general GeoAPI provides a very strict interface and GeoTools will provide a class.

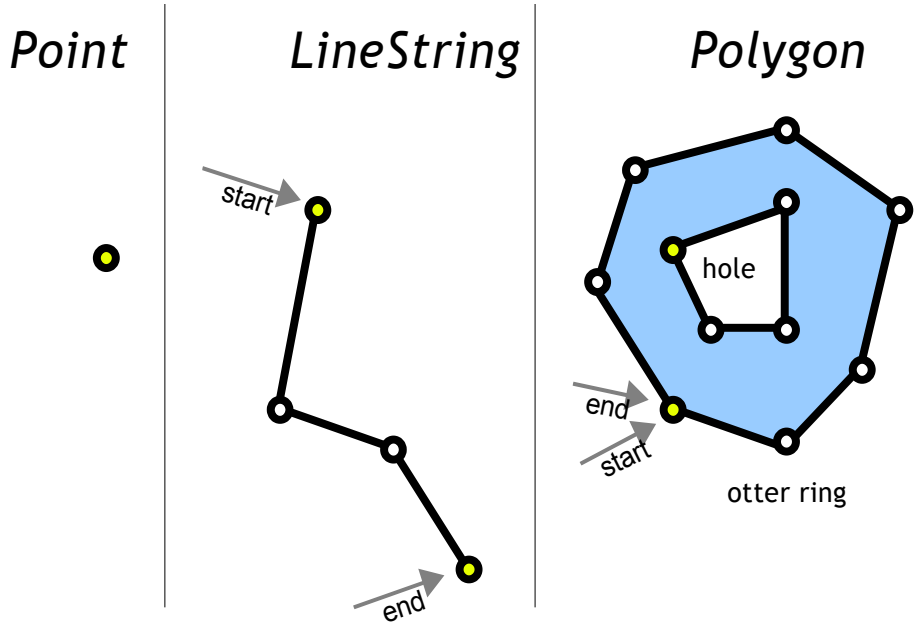


It is very common for a Feature to have only simple Attributes (String, Integer, Date and so on) rather then references to other Features, or data structures such as List<Date>. Features that meet this requirement are so common we have broken out a sub-class to represent them called SimpleFeature.

At the Java level the Feature API provided by GeoTools is similar to how java.util.Map is used – it is simply a data structure used to hold information. You can look up attribute values by key; and the list of keys is provided by the FeatureType.

4.2 Geometry

The other difference between an Object and a Feature is that a Feature has some form of location information (if not we would not be able to draw it on a map). The location information is going to be captured by a “Geometry” (or shape) that is stored in an attribute.



We make use of the **JTS Topology Suite (JTS)** to represent Geometry. The JTS library provides an excellent implementation of Geometry – and gets geeky points for having a recursive acronym ! JTS is an amazing library and does all the hard graph theory to let you work with geometry in a productive fashion.

Here is an example of creating a Point using the Well-Known-Text (WKT) format.

```
GeometryFactory geometryFactory = JTSFactoryFinder.getGeometryFactory( null );

WKTReader reader = new WKTReader( geometryFactory );
Point point = (Point) reader.read("POINT (1 1)");
```

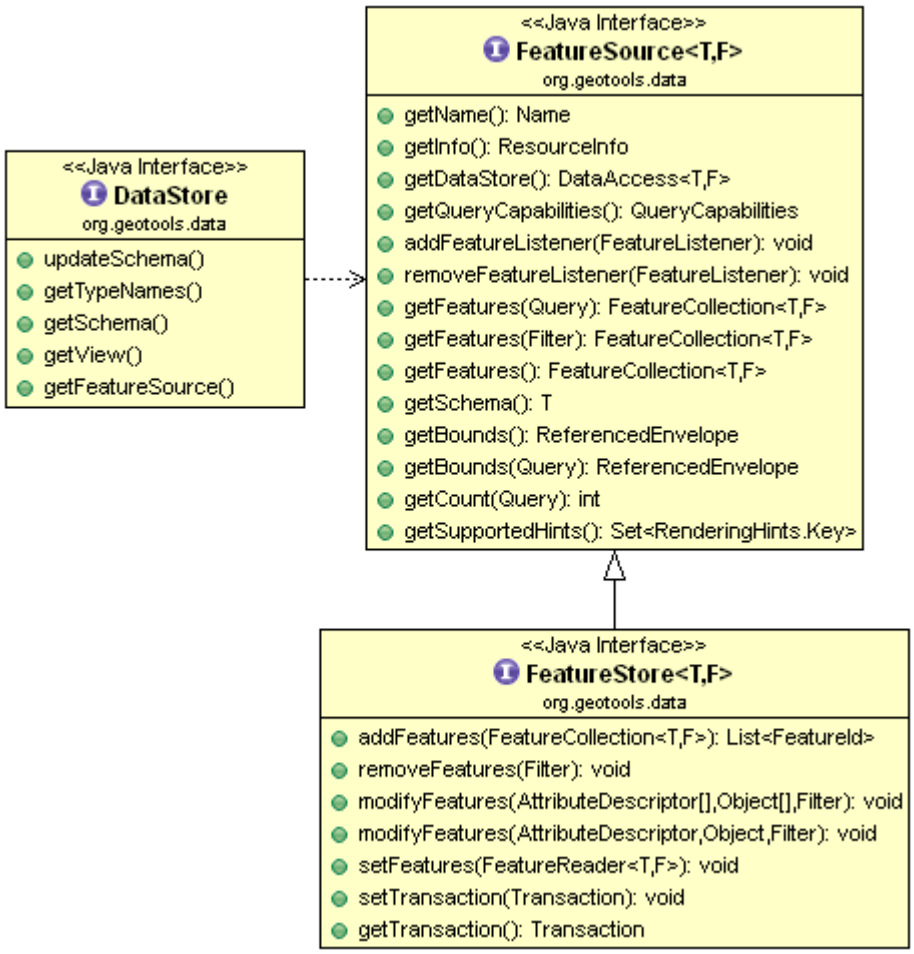
You can also create a Point by hand using the GeometryFactory directly.

```
GeometryFactory geometryFactory = JTSFactoryFinder.getGeometryFactory( null );

Coordinate coord = new Coordinate( 1, 1 );
Point point = geometryFactory.createPoint( coord );
```

4.3 DataStore

We ran into DataStore already in our Quickstart. The DataStore api is used to represent a File, Database or Service that has spatial data in it. The API has a couple of moving parts as shown below.



The FeatureSource is used to read features, the subclass FeatureStore is used for read/write access.

The way to tell if a File can be written to in GeoTools is to use an *instanceof* check.

```
String typeNames = dataStore.getTypeNames()[0];
FeatureSource source = store.getfeatureSource( typeName );
if( source instanceof FeatureStore){
   FeatureStore store = (FeatureStore) source; // write access!
   store.addFeatures( featureCollection );
   store.removeFeatures( filter ); // filter is like SQL WHERE
   store.modifyFeature( attribute, value, filter );
}
```

We decided to handle write access as a sub-class (rather then an isWritable method) in order to keep methods out of the way unless they could be used.