Geometry and CRS 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. Please set up your development environment prior to starting this tutorial. We will list the maven dependencies required at the start of the workbook.

This work book covers the dirty raw underbelly of the GIS world; yes I am afraid we are talking about... math. However please do not be afraid – all the math amounts to is shapes drawn on the earth.

This workbook is constructed in a code first manner; allowing you to work through the code example and read on if you have any questions.

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 CRS Lab

This example is an updated version of Quickstart. It adds a dialog allowing you to change the projection that the map is drawn in. The data will be “reprojected” into the correct coordinate reference system for display.

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

```
<dependencies>
  <dependency>
    <groupId>org.geotools</groupId>
    <artifactId>gt-swing</artifactId>
    <version>${geotools.version}</version>
    <!-- For this module we explicitly exclude some of its own -->
    <!-- dependencies from being downloaded because they are   -->
    <!-- big and we don't need them                            -->
    <exclusions>
      <exclusion>
        <groupId>org.apache.xmlgraphics</groupId>
        <artifactId>batik-transcoder</artifactId>
        </exclusion>
      </exclusions>
  </dependency>
  <dependency>
    <groupId>org.geotools</groupId>
    <artifactId>gt-shapefile</artifactId>
    <version>${geotools.version}</version>
  </dependency>
  <dependency>
    <groupId>org.geotools</groupId>
    <artifactId>gt-epsg-hsql</artifactId>
    <version>${geotools.version}</version>
  </dependency>
</dependencies>
```

2. Create the CRSLab.java file and copy and paste the following code.

```
package org.geotools.demo;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import javax.swing.JButton;
import javax.swing.JToolBar;
import org.geotools.data.FeatureSource;
import org.geotools.data.FileDataStore;
import org.geotools.data.FileDataStoreFinder;
import org.geotools.map.DefaultMapContext;
import org.geotools.map.MapContext;
import org.geotools.swing.JCRSChooser;
import org.geotools.swing.JMapFrame;
import org.geotools.swing.data.JFileDataStoreChooser;
import org.opengis.referencing.crs.CoordinateReferenceSystem;

/**
 * This is a visual example of changing the coordinate reference
 * system of a feature layer.
 */
public class CRSLab {

}
```

3. We can now create a main method that connects to a shapefile and uses a JMapFrame to display it.

   This should look familiar to you from the Quickstart example.

```
public static void main(String[] args) throws Exception {
    File file = JFileDataStoreChooser.showOpenFile("shp", null);
    if (file == null) {
        return;
    }

    FileDataStore store = FileDataStoreFinder.getDataStore(file);
    FeatureSource featureSource = store.getFeatureSource();

    // Create a map context and add our shapefile to it
    final MapContext map = new DefaultMapContext();
    map.addLayer(featureSource, null);

    JMapFrame mapFrame = new JMapFrame(map);
    mapFrame.enableTool(JMapFrame.Tool.NONE);
    mapFrame.enableStatusBar(true);

    JToolBar toolbar = mapFrame.getToolBar();
    JButton btn = new JButton("Change CRS");
    toolbar.add( new AbstractAction("Change CRS") {
        public void actionPerformed(ActionEvent arg0) {
            try {
                CoordinateReferenceSystem crs = JCRSChooser.showDialog(
                        null, "Coordinate Reference System", "Choose a new projection:", null);
                if( crs != null){
                    map.setCoordinateReferenceSystem(crs);
                }

            } catch (Exception ex) {
                System.out.println("Could not use crs " + ex);
            }
        }
    });
    mapFrame.setSize(800, 600);
    mapFrame.setVisible(true);
}
```

4. Here is how we have customized the map frame

- Firstly, mapFrame.enableTool(JMapFrame.Tool.NONE) requests that an empty toolbar be created
- Next we create a JButton and add it to the toolbar
- Finally we set an action for the button so that when it is clicked a chooser dialog will be displayed to select a coordinate reference system which will be set as the new CRS of the map.

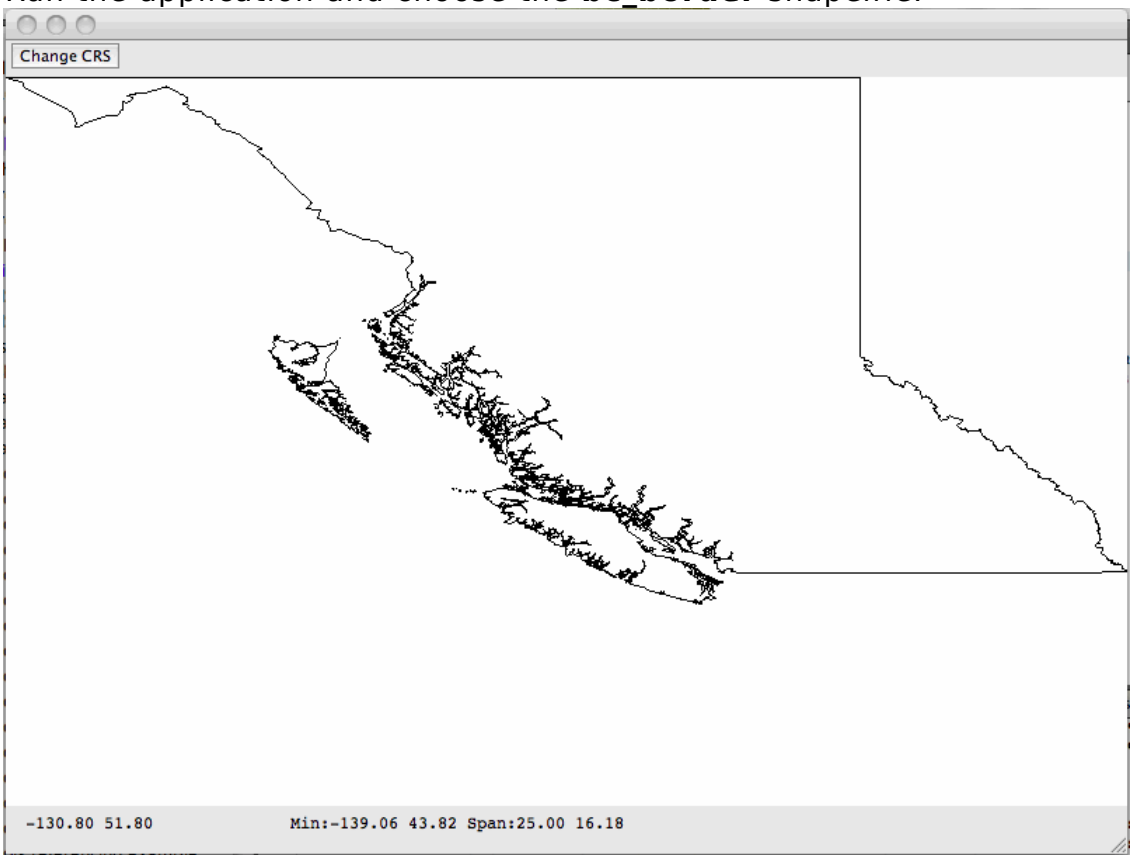
2.1 Running the Application

1. Grab the sample data from the Quickstart.

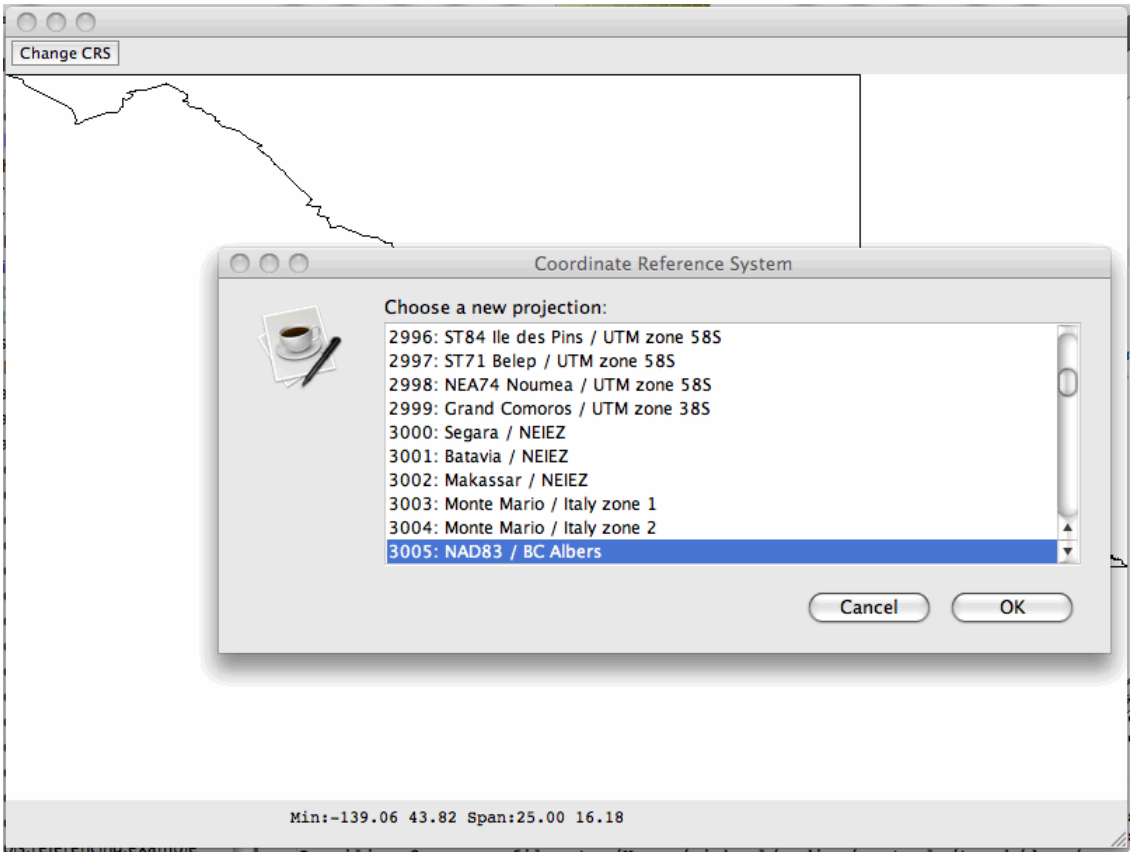
   http://udig.refractions.net/docs/data-v1_2.zip

   Please unzip this data directory to a location you can find easily like your desktop.

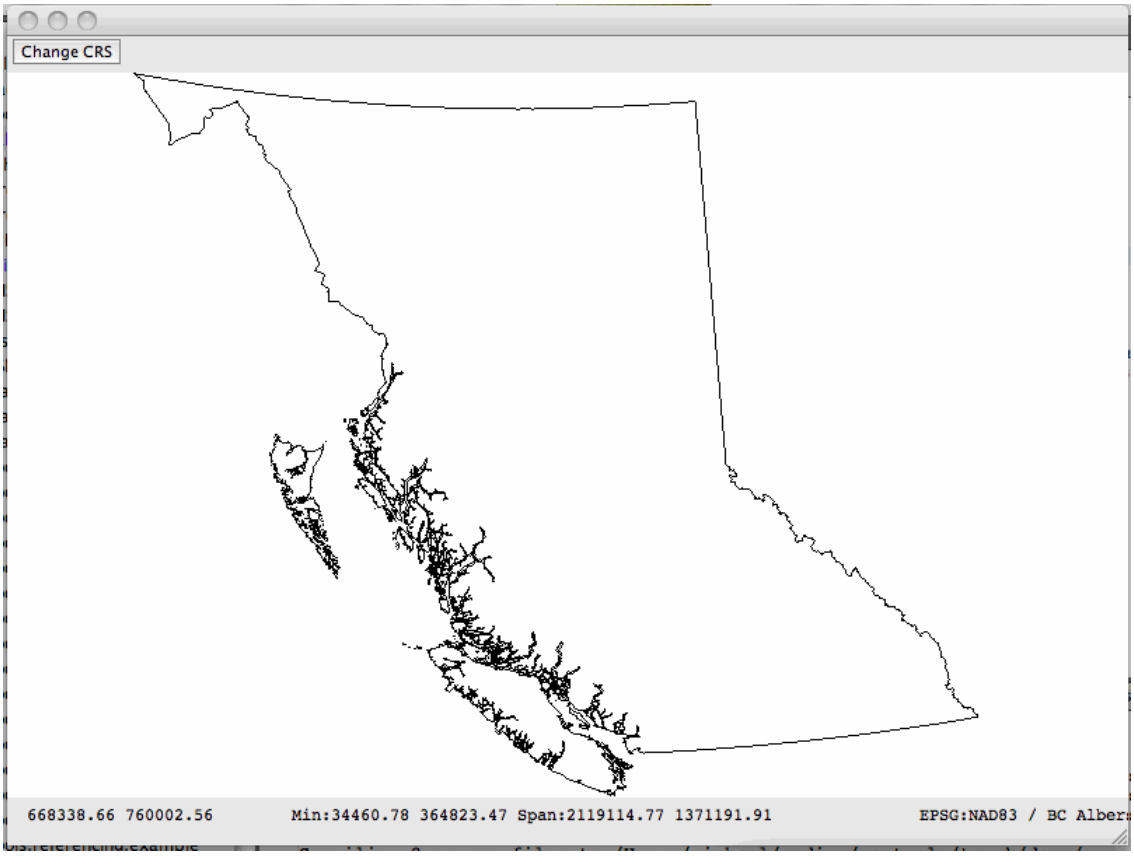
2. Run the application and choose the **bc_border** shapefile.



3. Now click the 'Change CRS' button and select the EPSG:3005 BC Albers projection. Hint: you can type 3005 rather than scrolling through the very long list.



4. When you click OK the map will be re-displayed in this new map projection. As well as the change in shape of the border, notice that the units in the status bar have changed from degrees to meters.



5. If you want to return to the original map projection, choose EPSG:4326.

3 Reproject a Shapefile

We can now put what we know together into a utility that will read in a shapefile and write out a shapefile in a different coordinate reference system.

One important thing to pick up from this lab is how easy it is to create a MathTransform between two CoordinateReferenceSystems.

You can use the MathTransform to transform points one at a time; or use the JTS utility class to create a copy of a Geometry with the points modified.

Let us use these two ideas to write out a new shapefile in a new projection.

1. Start by adding an export button.

```
        toolbar.add( new SafeAction("Export") {
            public void action(ActionEvent e) throws Throwable {
                export( featureSource, map.getCoordinateReferenceSystem(), file );
            }
        });
```

2. We will now create an export method to prompt the user for a filename.

```
    static void export(FeatureSource featureSource,
            CoordinateReferenceSystem crs, File origional) throws Exception {
       SimpleFeatureType schema = featureSource.getSchema();
        JFileDataStoreChooser chooser = new JFileDataStoreChooser("shp");
        chooser.setDialogTitle("Save reprojected shapefile");
        chooser.setSaveFile(sourceFile);
        int returnVal = chooser.showSaveDialog(null);
        if (returnVal != JFileDataStoreChooser.APPROVE_OPTION) {
            return;
        }
        File file = chooser.getSelectedFile();
        if (file.equals(sourceFile)) {
            JOptionPane.showMessageDialog(
                    null, "Cannot replace " + file,
                    "File warning", JOptionPane.WARNING_MESSAGE);
            return;
        }
```

3. We can now set up the MathTransform between the two coordinate reference systems.

```
        CoordinateReferenceSystem dataCRS = schema.getCoordinateReferenceSystem();
        CoordinateReferenceSystem worldCRS = map.getCoordinateReferenceSystem();
        boolean lenient = true; // allow for some error due to different datums
        MathTransform transform = CRS.findMathTransform( dataCRS, worldCRS, lenient );
```

4. And then grab all the features.

```
        FeatureCollection<SimpleFeatureType, SimpleFeature> featureCollection = featureSource.getFeatures();
```

5. To create a new shapefile we will need to produce a FeatureType that is similar to our original – the only difference will be the CoordinateReferenceSystem of the geometry descriptor.

```
        DataStoreFactorySpi factory = new ShapefileDataStoreFactory();
        Map<String, Serializable> create = new HashMap<String, Serializable>();
        create.put("url", file.toURI().toURL());
        create.put("create spatial index", Boolean.TRUE);
        DataStore newDataStore = factory.createNewDataStore(create);
        SimpleFeatureType featureType = SimpleFeatureTypeBuilder.retype( schema, worldCRS );
        newDataStore.createSchema( featureType );
```

6. We can now carefully open an iterator to go through the contents, and a writer to write out the new Shapefile.

```
// carefully open an iterator and writer to process the results
        Transaction transaction = new DefaultTransaction("Reproject");
        FeatureWriter<SimpleFeatureType, SimpleFeature>
              writer = newDataStore.getFeatureWriterAppend( featureType.getTypeName(), transaction);
        FeatureIterator<SimpleFeature> iterator = featureCollection.features();
        try {
            while( iterator.hasNext() ){
                // copy the contents of each feature and transform the geometry
                SimpleFeature feature = iterator.next();
                SimpleFeature copy = writer.next();
                copy.setAttributes( feature.getAttributes() );

                Geometry geometry = (Geometry) feature.getDefaultGeometry();
                Geometry geometry2 = JTS.transform(geometry, transform);

                copy.setDefaultGeometry( geometry2 );
                writer.write();
            }
            transaction.commit();
            JOptionPane.showMessageDialog(
                null, "Export complete", "Export", JOptionPane.INFORMATION_MESSAGE);

        } catch (Exception problem) {
            problem.printStackTrace();
            transaction.rollback();
            JOptionPane.showMessageDialog(null, "Export to shapefile failed", "Export",
JOptionPane.ERROR_MESSAGE);
        } finally {
            writer.close();
            iterator.close();
            transaction.close();
        }
```

3.1 Things to Try

Here are a couple things to try with the above application.

- Have a look at the coordinates displayed at the bottom of the screen in EPSG:4326 and in EPSG:3005. You should be able to see that one is measured in degrees and the other measured in meters.
- Make a button to print out the map coordinate reference system as human readable “Well Known Text”. This is the same text format used by a shapefile's “prj” side car file!
- Visit the JTS website and look up how to simplify geometry. Modify the example to simplify the geometry before writing it out - there are several techniques to try (the TopologyPreservingSimplifier and DouglasPeuckerSimplifier classes are recommended).

  This exercise is a common form of data preparation.
- One thing that can be dangerous about geometry – especially ones you make yourself – is that they can be invalid. The geoemtry.isValid() method allows you to test for invalid geometry – such as a polygon that forms a figure eight; or a LineString with only one point. Add code to test for invalid geometry.

- There are many tricks to fixing an invalid geometry. An easy place to start is to use geometry.buffer(0). Use this tip to build your own shapefile cleaner.

4 Geometry

Geometry is literally the shape of GIS.

Usually there is one geometry for a feature; and the attributes provide further description or measurement. It is sometimes hard to think of the geometry as being another attribute. It helps if you consider situations where there are several representations of the same thing.

We can represent the city of Sydney:

- as a single location, ie. a point
- as the city limits (so you can tell when you are inside Sydney), ie. a polygon

Point

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 );
```

Line

Here is an example of a WKT LineString.

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

WKTReader reader = new WKTReader( geometryFactory );
LineString line = (LineString) reader.read("LINESTRING(0 2, 2 0, 8 6)");
```

A LineString is a sequence of segments in the same manner as a java String is a sequence of characters.

Here is an example using the Geometry Factory.

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

Coordinate[] coords  =
```

```
    new Coordinate[] {new Coordinate(0, 2), new Coordinate(2, 0), new Coordinate(8, 6) };

LineString line = geometryFactory.createLineString(coordinates);
```

Polygon

A polygon is formed in WKT by constructing an outer ring, and then a series of holes.

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

WKTReader reader = new WKTReader( geometryFactory );
Polygon polygon = (Polygon) reader.read("POLYGON((20 10, 30 0, 40 10, 30 20, 20 10))");
```

Why not use Java Shape!

Java Shape is actually very useful and covers ideas mentioned above – it is however very focused on drawing.

Geometry allows us to handle the “information” part of Geographic Information System – we can use it to create new geometry and test the relationships between geometries.

```
        // Create Geometry using other Geometry
        Geometry smoke = fire.buffer( 10 );
        Geometry evacuate = cities.intersection( smoke );

        // test important relationships
        boolean onFire = me.intersects( fire );
        boolean thatIntoYou = me.disjoint( you );
        boolean run = you.isWithinDistance( fire, 2 );

        // relationships actually captured as a fancy
        // String called an intersection matrix
        //
        IntersectionMatrix matrix = he.relate( you );
        thatIntoYou = matrix.isDisjoint();

        // it really is a fancy string; you can do
        // pattern matching to sort out what the geometries
        // being compared are up to
        boolean disjoint = matrix.matches("FF*FF****");
```

You are encouraged to read the javadocs for JTS which contains helpful definitions.

The disjoint predicate has the following equivalent definitions:

- The two geometries have no point in common
- The DE-9IM Intersection Matrix for the two geometries is FF*FF****
- !g.intersects(this) (disjoint is the inverse of intersects)

5 Coordinate Reference System

Earlier we talked about the JTS library which provides our data model for Geometry. This is the real rocket science for map making – the idea of a shape and enough math to do something fun with it.

But there is one question we have not answered yet – what does a geometry mean?

You may think I am joking but the question is serious. A Geometry is just a bunch of math (a set of points in the mathematical sense). They have no meaning on their own.

An easier example is the number 3. The number 3 has no meaning on its own. It is only when you attach a “unit” that the number 3 can represent a concept. 3 metres. 3 feet. 3 score years.

In order to provide a Geometry with meaning we need to know what those individual points mean. We need to know where they are located – and the data structure that tells us this is called the Coordinate Reference System.

The Coordinate Reference System defines a couple of concepts for us:

- It defines the axis used – along with the units of measure.

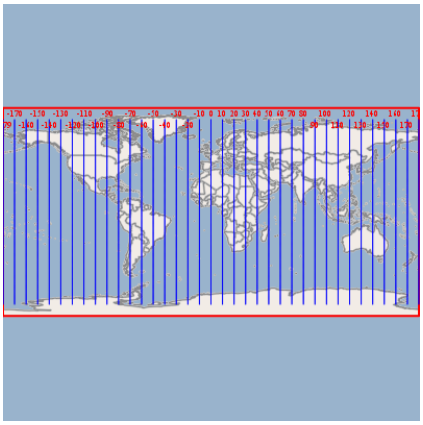
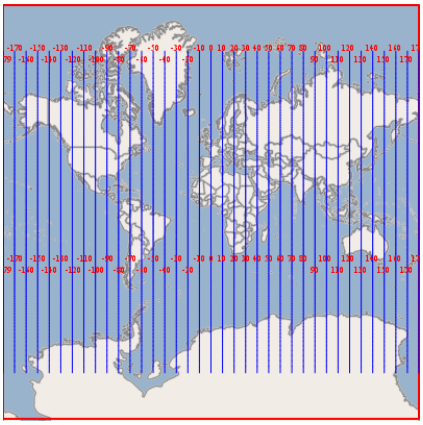
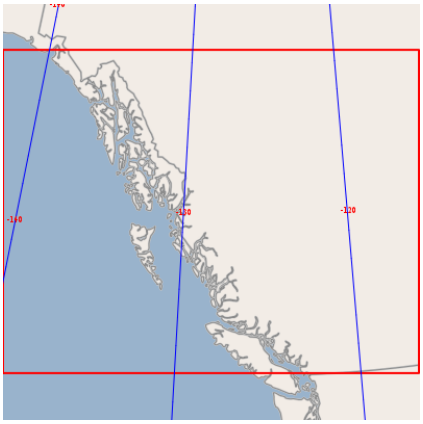
  So you can have lat measured in degrees , and lon measured in degrees from the equator.

  Or you can have x measured in metres, and y measured in metres which is very handy for calculating distances or areas.
- It defines the shape of the world. No really it does – not all coordinate reference systems imagine the same shape of the world. The CRS used by Google pretends the world is a perfect sphere, while the CRS used by “EPSG:4326” has a different shape in mind – so if you mix them up your data will be drawn in the wrong place (by up to 20 km).

As a programmer I view the coordinate reference system idea as a neat hack. Everyone is really talking about points in 3D space here – and rather than having to record x,y,z all the time we are cheating and only recording two points (lat,lon) and using a model of the shape of the earth in order to calculate z.

5.1.1 EPSG Codes

The first group that cared about this stuff enough to write it down in database form was the European Petroleum Standards Group (EPSG). The database is distributed in Microsoft Access and is ported into all kinds of other forms including the gt-hsql jar included with GeoTools.

	**EPSG:4326** **EPSG Projection 4326 - WGS 84** This is the big one – information measured by lat/lon using decimal degrees. CRS.decode("EPSG:4326"); DefaultGeographicCRS.WGS84;
	**EPSG: 3785** **Popular Visualisation CRS / Mercator** The official code for the "Google map" projection used by a lot of web mapping applications. It is nice to pretend the world is a sphere (it makes your math very fast) – but it looks really odd if you draw a square in Oslo. CRS.decode("EPSG:3785");
	**EPSG:3005** **NAD83 / BC Albers** Example of an "equal area" projection for the west coast of Canada. The axes are measured in metres which is handy for calculating distance or area. CRS.decode("EPSG:3005");

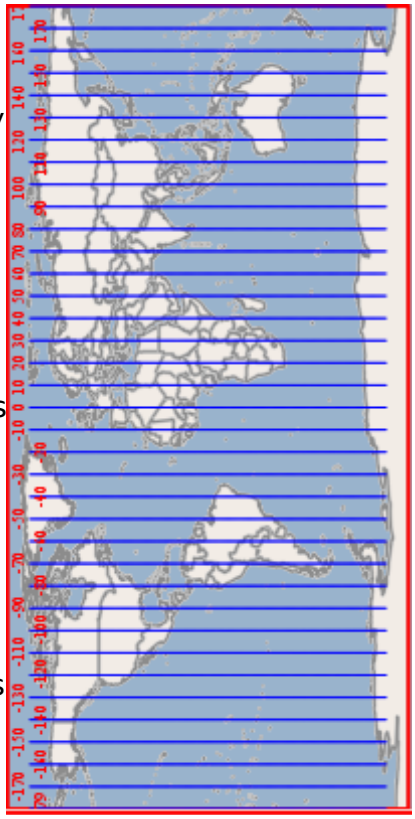
*The pictures here were provided by the **Spatial Reference Website***

Note that both EPSG:4326 and EPSG:3785 are using lat/lon – but arrive at a very different shape for their map.

5.1.2 Axis Order

When navigating by stars you can figure out latitude by the angle to the north star – but you need to guess for longitude based on how many days you have been traveling.

Hence y/x order.



This is also where I need to make a public apology. As computer scientists we occasionally get fed up when we work in a domain where "they are doing it wrong". Map making is an example of this. When we arrived on the scene maps were always recording position in latitude, followed by longitude; that is, with the north-south axis first and the east-west access second. When you draw that on the screen quickly it looks like the world is sideways as the coordinates are in"y/x" to my way of thinking and you need to swap them before drawing.

We are so used to working in x/y order that we would end up assuming it was supposed to be this way – and have been fighting with map makers ever since.

So if you see some data in "EPSG:4326" you have **no** idea if it is in x/y order or in y/x order.

We have finally sorted out an alternative; rather then EPSG:4326 we are supposed to use "*urn*:ogc:def:crs:*EPSG*:6.6:*4326*". If you ever see that you can be sure that a) someone really knows what they are doing and b) the data is recorded in exactly the order defined by the EPSG database.