Coordinate conversions made easy

Your primer for converting between varying geolocation systems

Sami Salkosuo August 28, 2007

Applications offering location-based services are all the rage, but how can a computer identify a location in the real world? There are several ways to answer that question, but they all involve a geographic coordinate system, and there are several such systems in use. In this article, application architect Sami Salkosuo offers Java™ code that converts location data between two popular systems: the familiar system of longitude and latitude and the Universal Transverse Mercator system.

Location services -- including GPS-based navigation systems and map sites such as Google Maps and Yahoo! Maps -- have become popular among consumers. Many organizations are already making use of location-aware services, and many more will do so as they realize the benefits and potential that these services hold. In 2006, Gartner noted that "location-aware applications will hit mainstream adoption in the next two to five years" and that already an "increasing number of organizations have deployed location-aware mobile business applications."

When an organization decides to implement a location-aware application, it is eventually a developer's task to write such an application. Building a location-aware service involves many tasks, big and small, and one (relatively small) task may be to convert coordinates from one system to another. This article presents code that performs such conversions, which could save you many hours of work.

Two different coordinate systems

Before we dive into this article's code, we need to discuss the coordinate systems that the code is designed to handle: the familiar system of longitude and latitude and the Universal Transverse Mercator (UTM) system. We'll also touch on the Military Grid Reference System (MGRS), which is based on UTM.

Latitude and longitude

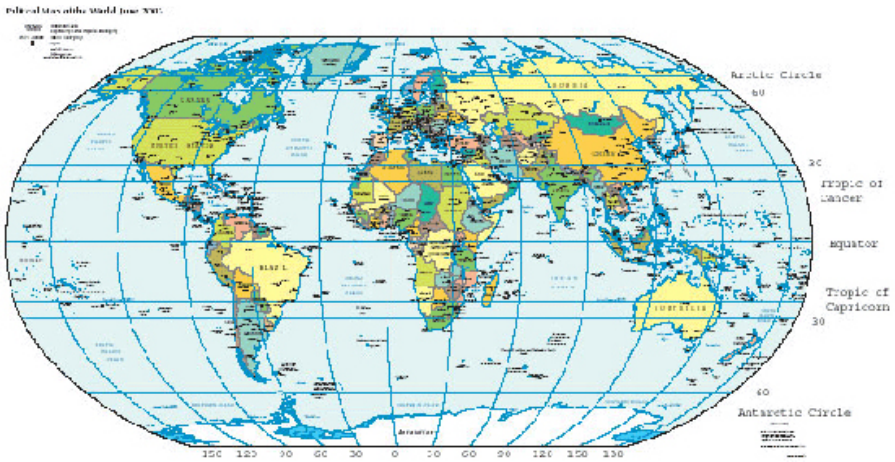
The latitude and longitude system is probably the best known way to designate geographic coordinates. It presents a location as two numbers. *Latitude* is the angle from the center of the Earth to some east-west line on the Earth's surface. *Longitude* is the angle from the center of the Earth to a north-south line on the Earth's surface. Latitude and longitude may be expressed as

decimal degrees (DD) or as degrees, minutes, and seconds (DMS); the latter gives numbers in a format such as 49Â°30'00" S 12Â°30'00" E. This is the format typically used in GPS devices.

Earth is divided by the equator (0Â° latitude) into Northern and Southern Hemispheres and by 0Â° longitude (an imaginary line from the North Pole to the South Pole that goes through the city of Greenwich in the UK) into Eastern and Western Hemispheres. The Northern Hemisphere has latitudes between 0 and 90 degrees, and the Southern Hemisphere has latitudes between 0 and -90 degrees. The Eastern Hemisphere is between 0 and 180 degrees, and the Western Hemisphere between 0 and -180 degrees.

For example, the coordinates 61.44, 25.40 (in DD) or 61Â°26'24"N, 25Â°23'60"E (in DMS) are located in southern Finland. The coordinates -47.04, -73.48 (in DD) or 47Â°02'24"S, 73Â°28'48"W (in DMS) are located in southern Chile. Figure 1 shows the Earth overlaid with latitude and longitude lines:

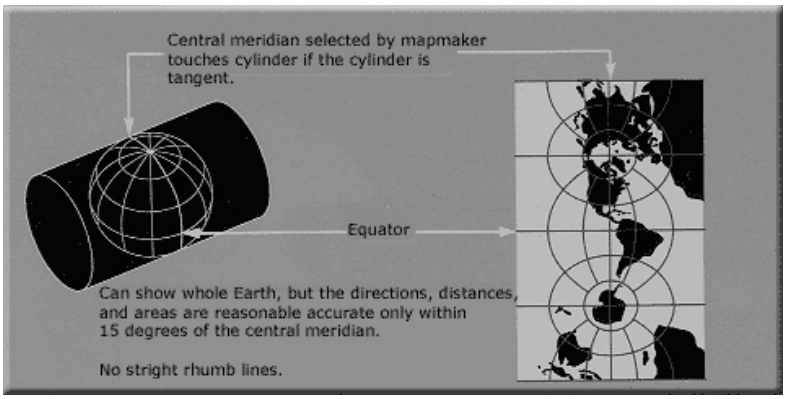
Figure 1. The Earth, with latitude and longitude lines displayed



Universal Transverse Mercator

The UTM coordinate system is a grid-based method for specifying coordinates. The UTM system divides the Earth into 60 zones, each based on the Transverse Mercator projection. Map projection in cartography is a way to present a two-dimensional curved surface on a plane, such as a normal map. Figure 2 shows a Traverse Mercator projection:

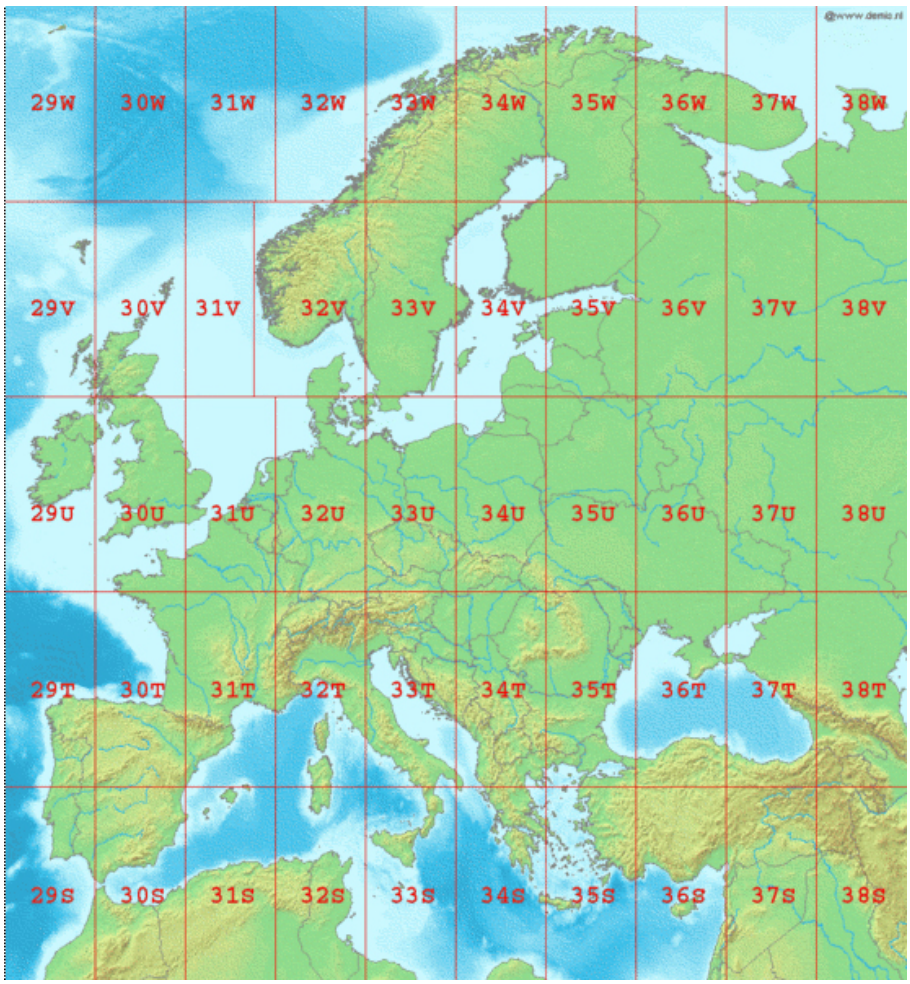
Figure 2. A Transverse Mercator projection



The UTM longitude zones are numbered 1 through 60; all but two zones (more on which in a moment) are 6Â° wide from to east to west. The longitude zones cover the whole surface of the Earth between latitudes 80Â°S and 84Â°N.

There are 20 UTM latitude zones, each 8Â° high; the zones are lettered from C to X (with letters I and O omitted). Zones A, B, Y, and Z exist outside of this system; they cover the Antarctic and Arctic regions. Figure 3 shows UTM zones in Europe. The two nonstandard longitude zones are visible in Figure 3: zone 32V is extended to cover all of southern Norway, while zone 31V is shrunk so that it covers only open water.

Figure 3. UTM zones in Europe



UTM coordinates are presented in the format *longitude zone latitude zone easting northing,* where *easting* is the projected distance from longitude zone's central meridian and *northing* is the projected distance from the equator. The values of both easting and northing are given in meters. For example, the latitude/longitude coordinates 61.44, 25.40 are presented in UTM as 35 V 414668 6812844; the latitude/longitude coordinates -47.04, -73.48 are 18 G 615471 4789269 in UTM.

Military Grid Reference System

The MGRS is the standard used by NATO militaries. MGRS is based on UTM and further divides each zone to 100 km by 100 km squares. These squares are identified by two-letter digraphs: the first letter is the east-west position within the longitude zone and the second letter is the north-south position.

For example, the UTM point 35 V 414668 6812844 is equivalent to the MGRS point 35VMJ1466812844. This MGRS point is accurate within one meter and is presented using 15 characters, where the last 10 characters are the easting and northing values within the specified grid. MGRS may be presented using 15 characters (as in previous example), or 13, 11, 9, or 7

characters; the values so expressed would be accurate within 1, 10, 100, 1,000, or 10,000 meters, respectively.

Converting coordinates

To define latitude and longitude coordinates for a location on the Earth, at minimum, you must be able to see the stars or the Sun and have a sextant and clock that displays time in GMT. You can determine latitude from the angle between a celestial object and the horizon, and you can calculate longitude from the Earth's rotation. This article does not go into these details. Instead, we'll assume that we already have coordinates in DD, DMS, or UTM format.

Converting decimal degrees to degrees/minutes/seconds, and vice versa

It is easy to convert coordinates between DD and DMS. Here's the formula for converting from DD to DMS:

```
DD: dd.ff
DMS: dd mm ss

dd=dd
mm.gg=60*ff
ss=60*gg
```

Here, `gg` is the fractional part of the calculation. Negative latitude denotes a location in the Southern Hemisphere (S) and negative longitude is a location in the Western Hemisphere (W). For example, imagine that you have the coordinates (in DD format) of 61.44, 25.40. You'd convert them as follows:

```
lat dd=61
lat mm.gg=60*0.44=26.4
lat ss=60*0.4=24
```

And:

```
lon dd=25
lon mm.gg=60*0.40=24.0
lon ss=60*0.0=0
```

Thus, in DMS format, the coordinates are 61Â°26'24"N 25Â°24'00"E.

The formula to go from DMS to DD is as follows:

```
DD: dd.ff
DMS: dd mm ss

dd.ff=dd + mm/60 + ss/3600
```

Remember, locations in the Southern Hemisphere (S) are at negative latitudes, and locations in the Western Hemisphere (W) are at negative longitudes.

Let's convert the DMS coordinates 47Â°02'24"S, 73Â°28'48"W to DD notation:

```
lat dd.ff= - (47 + 2/60 + 24/3600 )=-47.04
lon dd.ff= - (73 + 28/60 + 48/3600)=-73.48
```

Thus, the coordinates are -47.04, -73.48 in DD.

Converting from latitude/longitude to UTM and vice versa

Unlike decimal coordinates, which you can determine using a sextant and a chronometer, you cannot determine UTM coordinates without calculations. Although these calculations are nothing more than basic trigonometry and algebra, the formulas are very complicated. If you look at "The Universal Grids: Universal Transverse Mercator (UTM) and Universal Polar Stereographic (UPS)", you'll see what I mean.

The UTM conversion formulas are not presented here, but the source code in the following section provides some light.

Converting coordinates with Java code

This section introduces the source code of a library class that performs coordinate conversion between decimal degrees and UTM. This Java class is named `com.ibm.util.CoordinateConversion`; my idea was to make a single class that has methods for conversions. The class includes inner classes that actually do the conversion; if necessary, the inner classes can be refactored out from the `CoordinateConversion` class to create a library package or add classes to existing package. The conversions that the class performs are accurate to within 1 meter.

The source for `CoordinateConversion` has about 750 lines of code, so it is not presented in this text of this article in its entirety. The relevant methods are described in the following sections, and the full source code is included in the Download section.

CoordinateConversion

`CoordinateConversion` is the main class and it is instantiated to perform coordinate conversion when required. Listing 1 presents the relevant public methods, along with private inner classes that are included in the `CoordinateConversion` class:

Listing 1. CoordinateConversion

```
public class CoordinateConversion
{

  public CoordinateConversion()
  {

  }

  public double[] utm2LatLon(String UTM)
  {
    UTM2LatLon c = new UTM2LatLon();
    return c.convertUTMToLatLong(UTM);
  }

  public String latLon2UTM(double latitude, double longitude)
```

```
{
  LatLon2UTM c = new LatLon2UTM();
  return c.convertLatLonToUTM(latitude, longitude);

}

//..implementation omitted

private class LatLon2UTM
{
  public String convertLatLonToUTM(double latitude, double longitude)
  {
    //..implementation omitted
  }
  //..implementation omitted
}

private class LatLon2MGRUTM extends LatLon2UTM
{
  public String convertLatLonToMGRUTM(double latitude, double longitude)
  {
    //..implementation omitted
  }
  //..implementation omitted
}


private class MGRUTM2LatLon extends UTM2LatLon
{
  public double[] convertMGRUTMToLatLong(String mgrutm)
  {
    //..implementation omitted
  }
  //..implementation omitted
}


private class UTM2LatLon
{
  public double[] convertUTMToLatLong(String UTM)
  {
    //..implementation omitted
  }
  //..implementation omitted
}

private class Digraphs
{
  //used to get digraphs when doing conversion between
  //lat/long and MGRS
  //..implementation omitted
}

private class LatZones
{
  //include methods to determine latitude zones
  //..implementation omitted
}
```

The next section looks at the latitude/longitude and UTM conversions in more detail.

Conversion from latitude/longitude to UTM

Coordinates are converted from latitude/longitude to UTM with the `String latLon2UTM(double latitude, double longitude)` method. The method implementation creates a new instance of the

inner class `LatLon2UTM c = new LatLon2UTM();` and returns UTM coordinates as a 15-character string (that is, to 1-meter precision). The implementation of the `LatLon2UTM` methods is shown in Listing 2:

Listing 2. public String convertLatLonToUTM(double latitude, double longitude)

```
public String convertLatLonToUTM(double latitude, double longitude)
{
  validate(latitude, longitude);
  String UTM = "";

  setVariables(latitude, longitude);

  String longZone = getLongZone(longitude);
  LatZones latZones = new LatZones();
  String latZone = latZones.getLatZone(latitude);

  double _easting = getEasting();
  double _northing = getNorthing(latitude);

  UTM = longZone + " " + latZone + " " + ((int) _easting) + " "+ ((int) _northing);

  return UTM;
}
```

This method performs its conversion by calling various methods to get the latitude and longitude zone and calculating the easting and northing and so on. Input is validated using the `validate()` method; if the clause `(latitude < -90.0 || latitude > 90.0 || longitude < -180.0 || longitude >= 180.0)` is true, it throws an `IllegalArgumentException`.

The `setVariables()` method in Listing 3 sets various variables required for calculating conversions (take a look at "The Universal Grids" for more information:

Listing 3. protected void setVariables(double latitude, double longitude)

```
protected void setVariables(double latitude, double longitude)
{
  latitude = degreeToRadian(latitude);
  rho = equatorialRadius * (1 - e * e) / POW(1 - POW(e * SIN(latitude), 2), 3 / 2.0);

  nu = equatorialRadius / POW(1 - POW(e * SIN(latitude), 2), (1 / 2.0));

  double var1;
  if (longitude < 0.0)
  {
    var1 = ((int) ((180 + longitude) / 6.0)) + 1;
  }
  else
  {
    var1 = ((int) (longitude / 6)) + 31;
  }
  double var2 = (6 * var1) - 183;
  double var3 = longitude - var2;
  p = var3 * 3600 / 10000;

  S = A0 * latitude - B0 * SIN(2 * latitude) + C0 * SIN(4 * latitude) - D0
      * SIN(6 * latitude) + E0 * SIN(8 * latitude);
```

```
  K1 = S * k0;
  K2 = nu * SIN(latitude) * COS(latitude) * POW(sin1, 2) * k0 * (100000000) / 2;
  K3 = ((POW(sin1, 4) * nu * SIN(latitude) * Math.pow(COS(latitude), 3)) / 24)
      * (5 - POW(TAN(latitude), 2) + 9 * e1sq * POW(COS(latitude), 2) + 4
      * POW(e1sq, 2) * POW(COS(latitude), 4))
      * k0
      * (10000000000000000L);

  K4 = nu * COS(latitude) * sin1 * k0 * 10000;

  K5 = POW(sin1 * COS(latitude), 3) * (nu / 6)
      * (1 - POW(TAN(latitude), 2) + e1sq * POW(COS(latitude), 2)) * k0
      * 1000000000000L;

  A6 = (POW(p * sin1, 6) * nu * SIN(latitude) * POW(COS(latitude), 5) / 720)
      * (61 - 58 * POW(TAN(latitude), 2) + POW(TAN(latitude), 4) + 270
      * e1sq * POW(COS(latitude), 2) - 330 * e1sq
      * POW(SIN(latitude), 2)) * k0 * (1E+24);

}
```

The `getLongZone()` method in Listing 4 and the `LatZones` class (available in the source code) are used to get longitude and latitude zones. The longitude zone is calculated from the longitude parameter, and latitude zones are basically hard coded using an array in the `LatZones` class.

Listing 4. protected String getLongZone(double longitude)

```
protected String getLongZone(double longitude)
{
  double longZone = 0;
  if (longitude < 0.0)
  {
    longZone = ((180.0 + longitude) / 6) + 1;
  }
  else
  {
    longZone = (longitude / 6) + 31;
  }
  String val = String.valueOf((int) longZone);
  if (val.length() == 1)
  {
    val = "0" + val;
  }
  return val;
}
```

The `getNorthing()` (in Listing 5) and `getEasting()` (in Listing 6) methods calculate the northing and easting values. Both methods use variables that were set in the `setVariables()` method in Listing 3.

Listing 5. protected double getNorthing(double latitude)

```
protected double getNorthing(double latitude)
{
  double northing = K1 + K2 * p * p + K3 * POW(p, 4);
  if (latitude < 0.0)
  {
    northing = 10000000 + northing;
  }
  return northing;
}
```

Listing 6. protected double getEasting()

```
protected double getEasting()
{
  return 500000 + (K4 * p + K5 * POW(p, 3));
}
```

Listing 7 includes some sample output, including some latitude/longitude coordinates and the corresponding UTM coordinates:

Listing 7. Latitude/longitude-to-UTM test values

```
( 0.0000    0.0000  )     "31 N 166021 0"
( 0.1300   -0.2324  )     "30 N 808084 14385"
(-45.6456   23.3545 )     "34 G 683473 4942631"
(-12.7650  -33.8765 )     "25 L 404859 8588690"
(-80.5434  -170.6540)     "02 C 506346 1057742"
( 90.0000   177.0000)     "60 Z 500000 9997964"
(-90.0000  -177.0000)     "01 A 500000 2035"
( 90.0000    3.0000 )     "31 Z 500000 9997964"
( 23.4578  -135.4545)     "08 Q 453580 2594272"
( 77.3450   156.9876)     "57 X 450793 8586116"
(-89.3454  -48.9306 )     "22 A 502639 75072"
```

Conversion from UTM to latitude/longitude

Converting from UTM coordinates to latitude and longitude is a little easier than the reverse process. Again, "The Universal Grids" includes the formulas for the conversions. Listing 8 shows the code for the `convertUTMToLatLong()` method. This returns a double array, where the first element (array index `[0]`) is latitude and second element (array index `[1]`) is longitude. Because the UTM string parameter has an accuracy of 1 meter, the latitude/longitude coordinates have the same accuracy.

Listing 8. public double[] convertUTMToLatLong(String UTM)

```
public double[] convertUTMToLatLong(String UTM)
{
  double[] latlon = { 0.0, 0.0 };
  String[] utm = UTM.split(" ");
  zone = Integer.parseInt(utm[0]);
  String latZone = utm[1];
  easting = Double.parseDouble(utm[2]);
  northing = Double.parseDouble(utm[3]);
  String hemisphere = getHemisphere(latZone);
  double latitude = 0.0;
  double longitude = 0.0;

  if (hemisphere.equals("S"))
  {
    northing = 10000000 - northing;
  }
  setVariables();
  latitude = 180 * (phi1 - fact1 * (fact2 + fact3 + fact4)) / Math.PI;

  if (zone > 0)
  {
    zoneCM = 6 * zone - 183.0;
```

```
  }
  else
  {
    zoneCM = 3.0;
  }

  longitude = zoneCM - _a3;
  if (hemisphere.equals("S"))
  {
    latitude = -latitude;
  }

  latlon[0] = latitude;
  latlon[1] = longitude;
  return latlon;

}
```

The `convertUTMToLatLong()` method splits the incoming UTM string (which is in the format *34 G 683473 4942631*) and uses the `getHemisphere()` method to determine the hemisphere that the location denoted by the string is in. Determining the hemisphere is easy: latitude zones `A`, `C`, `D`, `E`, `F`, `G`, `H`, `J`, `K`, `L`, and `M` are in the Southern Hemisphere, and the rest are in the Northern Hemisphere.

The `setVariables()` method, shown in Listing 9, sets the variables required for calculation and then immediately calculates latitude. Longitude is calculated using the longitude zone.

Listing 9. protected void setVariables()

```
protected void setVariables()
{
  arc = northing / k0;
  mu = arc
      / (a * (1 - POW(e, 2) / 4.0 - 3 * POW(e, 4) / 64.0 - 5 * POW(e, 6) / 256.0));

  ei = (1 - POW((1 - e * e), (1 / 2.0)))
      / (1 + POW((1 - e * e), (1 / 2.0)));

  ca = 3 * ei / 2 - 27 * POW(ei, 3) / 32.0;

  cb = 21 * POW(ei, 2) / 16 - 55 * POW(ei, 4) / 32;
  cc = 151 * POW(ei, 3) / 96;
  cd = 1097 * POW(ei, 4) / 512;
  phi1 = mu + ca * SIN(2 * mu) + cb * SIN(4 * mu) + cc * SIN(6 * mu) + cd
      * SIN(8 * mu);

  n0 = a / POW((1 - POW((e * SIN(phi1)), 2)), (1 / 2.0));

  r0 = a * (1 - e * e) / POW((1 - POW((e * SIN(phi1)), 2)), (3 / 2.0));
  fact1 = n0 * TAN(phi1) / r0;

  _a1 = 500000 - easting;
  dd0 = _a1 / (n0 * k0);
  fact2 = dd0 * dd0 / 2;

  t0 = POW(TAN(phi1), 2);
  Q0 = e1sq * POW(COS(phi1), 2);
  fact3 = (5 + 3 * t0 + 10 * Q0 - 4 * Q0 * Q0 - 9 * e1sq) * POW(dd0, 4) / 24;

  fact4 = (61 + 90 * t0 + 298 * Q0 + 45 * t0 * t0 - 252 * e1sq - 3 * Q0
          * Q0)
          * POW(dd0, 6) / 720;

  lof1 = _a1 / (n0 * k0);
```

```
  lof2 = (1 + 2 * t0 + Q0) * POW(dd0, 3) / 6.0;
  lof3 = (5 - 2 * Q0 + 28 * t0 - 3 * POW(Q0, 2) + 8 * e1sq + 24 * POW(t0, 2))
          * POW(dd0, 5) / 120;
  _a2 = (lof1 - lof2 + lof3) / COS(phi1);
  _a3 = _a2 * 180 / Math.PI;

}
```

`setVariables()` uses the easting and northing values to set required variables. These are both class variables and are set in the `convertUTMToLatLong(String UTM)` method (see Listing 8).

Other methods

The source code also includes other public and private methods and classes. For example, methods and classes to convert coordinates between latitude/longitude and some helper methods to perform degree-to-radian conversions (and vice versa) and for various mathematical operations (such as POW, SIN, COS, and TAN).

Summary

This article introduced a little bit of the theory of world coordinate systems, along with a Java class to perform coordinate transformation. Typically, this theory is not needed in everyday development work -- except in rare cases where there is no good way to do otherwise, as I recently discovered when I encountered a coordinate transformation task.

I needed to perform conversion between latitude and longitude, UTM, and MGRS, and so I did basic research and implemented the transformations in a Java class. For me, the development work took several hours; I hope that you will save those hours for other tasks and that you find `CoordinateConversion` useful in your own work.

Downloadable resources

Description	Name	Size
Coordinate conversion source code[1]	j-coordconvert.zip	4KB

Note

1. The download also includes MGRS methods, but conversion from MGRS is incorrect.

Related topics

- Gartner's 2006 emerging technologies hype cycle highlights key technology themes
- The Universal Grids: Universal Transverse Mercator (UTM) and Universal Polar Stereographic (UPS)
- Put yourself on the map with Google Maps API, DB2/Informix, and PHP on Linux
- Creating an automatically maintained spatial table from latitude-longitude column data
- Geographic coordinate systems