

Chapter 34. GPS Mapping and Geotagged Images

GPS chips are becoming standard in smart phones, but what about adding similar functionality to laptops, netbooks, and other mobile PCs? GPS receivers that connect to devices via a USB port aren't particularly expensive (US\$30 for a decent one), and are surprisingly easy to utilize from Java.

The first part of this chapter describes my experience using the Canmore GT-730F USB GPS receiver (<http://www.canmore.com.tw/>) for obtaining latitude and longitude information to display on a map downloaded from OpenStreetMap (<http://www.openstreetmap.org/>). Figure 1 shows a map of the road outside my office building, created as I walked around with the GPS receiver plugged into my netbook.



Figure 1. My Current Position on a Map.

My position is shown by a yellow circle in the center of the map. As I moved about, the map was updated dynamically so the yellow circle stayed centered in the window. Figure 2 shows my location a little later, after I've walked down the street.

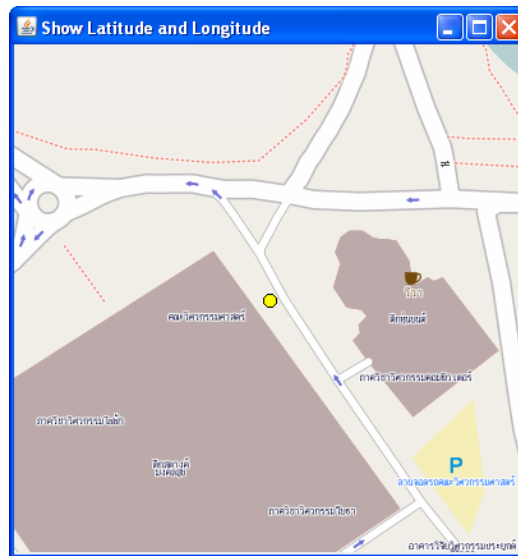


Figure 2. Updated Map Position.

The map rendering, including dynamic updating, is easy; all the heavy lifting is done by the JMapView Swing component (<http://wiki.openstreetmap.org/wiki/JMapView>) which manages the integration between Java and OpenStreetMap. JMapView also supports zoom controls, mouse dragging of the map, client-side map caches, and different map sources.

The application's GPS processing is handled by the gpsinput package from GPSylon (<http://www.tegmento.org/gpsylon/>), although the GPS data is so simple that it can be processed directly with the Java Communications API (<http://www.oracle.com/technetwork/java/index-jsp-141752.html>). I'll start by showing how to use the RXTX API (a near-compatible alternative to the Java Communications API, available at <http://www.rxtx.org/>) to treat the GPS receiver as a serial port input stream.

This chapter includes three other GPS-related applications:

- a 'tile sticher' which downloads OpenStreetMap map tiles and combines them into a single PNG image;
- a program that queries Panoramio's Web service (<http://www.panoramio.com>) for geotagged images. The image closest to the user's current latitude and longitude is downloaded;
- examples using exiftool (<http://www.sno.phy.queensu.ca/~phil/exiftool/>) and the Sanselan API (<http://commons.apache.org/sanselan/>) to read and write GPS information to the Exif metadata of JPG images.

1. Setting up the GPS Receiver

Figure 3 shows a picture of the Canmore GT-730F USB GPS receiver plugged into my netbook.



Figure 3. GPS Receiver and Netbook.

The Canmore receiver doesn't come with much documentation or software, but all that's really needed is the correct driver for the PC. The driver makes the receiver appear to be a serial port. This transformation can be checked in Windows XP by looking at the ports listed by the Device Manager in the System control panel. The receiver on my netbook appears as COM3 in Figure 4.



Figure 4. The GPS Receiver as a Serial Port (COM3).

In the case of the Canmore, I found it best to visit their Website (<http://www.canmore.com.tw/>) to download the latest USB-to-serial driver, manuals, and support software.

The Canmore receiver processes GPS information using the NMEA 0183 communication standard, which is based on simple ASCII 'sentences'; see <http://www.gpsinformation.org/dale/nmea.htm> for details on the NMEA format.

Each NMEA sentence is self-contained, independent of the other sentences sent to the receiver. Each sentence starts with a "\$" and a label which consists of a two letter prefix that defines the device (GP for GPS receivers) and a three letter sequence that specifies the sentence's contents. Each sentence ends with a carriage return/line feed

sequence (e.g. "\r\n" in Java) and sentences are no longer than 80 characters. Data items in a sentence are separated by commas.

The important sentence is "\$GPGGA" which contains location information. For example, the underlined parts of the "\$GPGGA" sentence:

```
$GPGGA,1,23519,4807.038,N,01131.000,E,1,08,0.9,545.4,M,46.9,M,,*47
```

state that the current fix was taken at 12:35:19 (the second argument of the sentence) at latitude 48 degrees 07.038' N (the third and fourth arguments), longitude 11 degrees 31.000' E (the fifth and sixth arguments).

Checking the Receiver

It's a good idea to check the GPS device using its own software before starting to code. In the case of the Canmore receiver, there's a simple "AGPS Tool" which shows the stream of NMEA sentences arriving at the device, and lists the serial port name and baud rate of the device (see Figure 5).

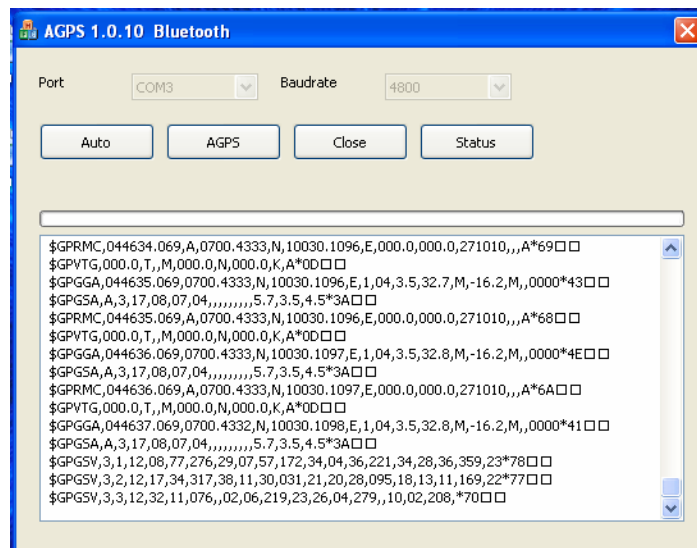


Figure 5. The Canmore AGPS Tool.

For the best results, the receiver should be outside, away from tall buildings and trees blocking the skyline. Initially, the receiver may take a minute or two to start producing data since a 'cold start' requires a scan of the sky for satellites.

A more visually interesting tool for the Canmore is the "GPS Viewer", shown in Figure 6, which displays satellite details, latitude and longitude, and other information.

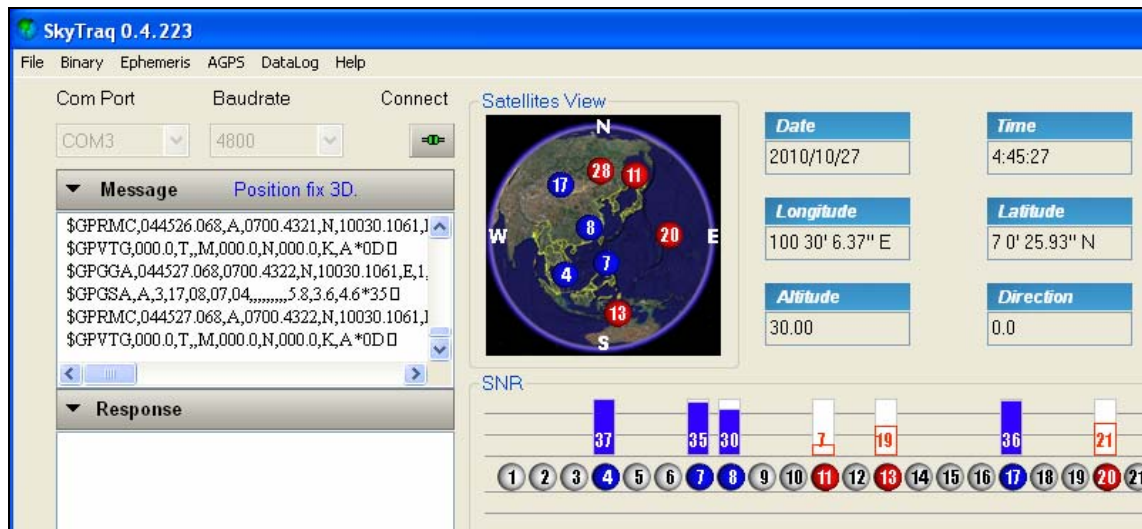


Figure 6. Part of the Canmore GPS Viewer.

The AGPS tool and GPS viewer are in the "AGPS Manual & Tool.zip" file available at <http://www.canmore.com.tw/download.php>. Other tools and recent drivers can be found on that page.

Both tools show that the receiver is working, and also give its serial port name (COM3) and baud rate (4800), which are required in the Java code that follows.

2. Accessing the GPS Receiver with Java

Since NMEA sentences are delivered by the Canmore receiver acting as a serial port, it's possible to process the input using the Java Communications API (<http://www.oracle.com/technetwork/java/index-jsp-141752.html>). Unfortunately, there are a few issues. One is that the Java Communications API isn't part of the standard JDK, and so must be downloaded and installed separately. More seriously, at least for Windows users like myself, is that the current version of the API no longer supports Windows! However, there is a near-compatible alternative library called RXTX (<http://www.rxtx.org/>) which does offer serial and parallel communication across all major platforms.

For most tasks, code using the Java Communications API can be compiled without change by RXTX, except that package names must be changed – the Java Communications API classes are mostly in a `javax.comm` package, while RXTX places the same-named classes in a `gui.io` package.

The RXTX home page looks a little disorganized, but the RXTX Wiki, which is accessible from that page, is clearer. An easy way of learning RXTX is to look for tutorials on the older Java Communications API, and remember to change the package names when compiling the code with RXTX.

The best descriptions of the Java Communications API can be found in chapter 22 of *Java I/O* by Elliotte Rusty Harold, O'Reilly 2006 (2nd edition), and chapter 12 of *Java Cookbook* by Ian Darwin, O'Reilly 2004 (2nd edition).

I downloaded rxtx 2.1-7r2 for Windows (rxtx-2.1-7-bins-r2.zip), unzipped it and created a folder c:\rxtx containing RXTXcomm.jar and the two Windows DLLs rxtxSerial.dll and rxtxParallel.dll (see Figure 7).

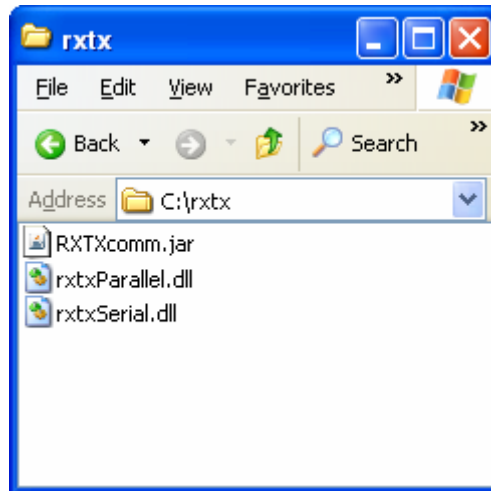


Figure 7. The RXTX Folder.

The location of these files is important when writing Java compilation and execution command lines later.

2.1. Listing the Ports

A simple, but useful, RXTX program lists serial and parallel port details:

```
import gnu.io.*;    // since using RXTX
import java.util.*;

public class PortList
{
    public static void main (String [] args)
    {
        Enumeration portList = CommPortIdentifier.getPortIdentifiers();
        while (portList.hasMoreElements()) {
            CommPortIdentifier portId =
                (CommPortIdentifier)portList.nextElement();

            if (portId.getPortType() == CommPortIdentifier.PORT_SERIAL) {
                System.out.println("Serial port: " + portId.getName());
                try {
                    SerialPort port =
                        (SerialPort) portId.open("portList", 1000); //1 sec wait
                    System.out.println("  baud: " + port.getBaudRate() +
                        "; data bits: " + port.getDataBits() +
                        "; flow control: " + port.getFlowControlMode() +
                        "; parity: " + port.getParity() +
                        "; stop bits: " + port.getStopBits() );
                    if (port != null)
                        port.close();
                }
                catch (PortInUseException e)
                { System.out.println("  Port already in use" ); }
            }
        }
    }
}
```

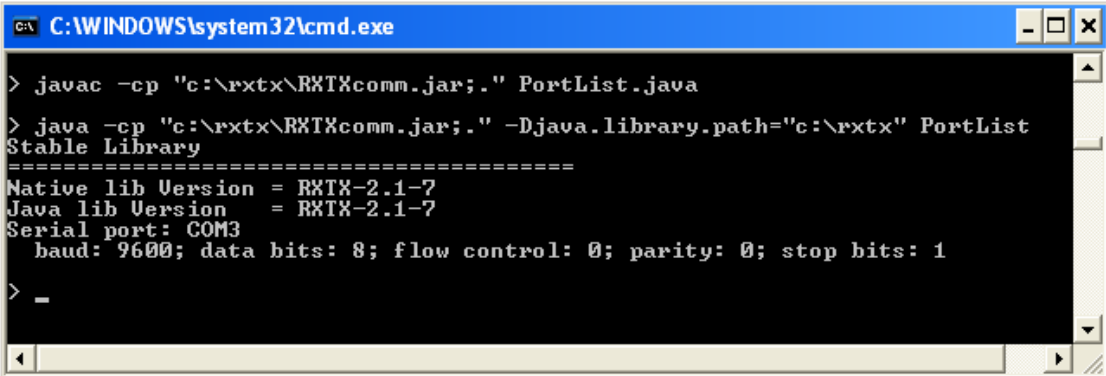
```

    }
    else if (portId.getPortType() ==
              CommPortIdentifier.PORT_PARALLEL) {
        System.out.println("Parallel port: " + portId.getName());
        try {
            ParallelPort port =
                (ParallelPort) portId.open("portList", 1000);
            System.out.println("    " + port);
            if (port != null)
                port.close();
        }
        catch (PortInUseException e)
        { System.out.println("    Port already in use" ); }
    }
    else
        System.out.println("Other port type: " + portId.getName());
}
} // end of main()
} // class PortList

```

CommPortIdentifier.getPortIdentifiers() returns an enumeration of the ports, which is listed out by a loop. The loop makes a distinction between serial, parallel, and other ports, and prints more details about the serial ports (which I'll be using with the GPS receiver). Each port is opened to check that it's available. If the opening hasn't succeeded within 1000 ms, then it's aborted and an exception raised.

On my netbook, the program is compiled and executed as shown in Figure 8.



```

C:\WINDOWS\system32\cmd.exe
> javac -cp "c:\rxtx\RXTXcomm.jar;." PortList.java
> java -cp "c:\rxtx\RXTXcomm.jar;." -Djava.library.path="c:\rxtx" PortList
Stable Library
=====
Native lib Version = RXTX-2.1-7
Java lib Version   = RXTX-2.1-7
Serial port: COM3
    baud: 9600; data bits: 8; flow control: 0; parity: 0; stop bits: 1
> _

```

Figure 8. Compiling and Running PortList.

The javac compilation line requires a classpath declaration for the RXTXcomm.jar file. The java execution line must also specify the location of the RXTX DLLs using -Djava.library.path.

The only port listed in Figure 8 is COM3, a serial port with a baud rate of 9600. This last piece of information is misleading, since it's necessary to reduce the baud rate to 4800 to cleanly receive NMEA data. This is confirmed by the AGPS Tool and GPS Viewer applications shown in Figures 5 and 6.

2.2. Reading the GPS Data

There are five main tasks required for reading GPS data:

1. Open the correct serial port, checking for a variety of error cases.
2. Layer an input stream over the port, suitable for reading lines of ASCII text.
3. Set up a port event listener which waits for data (the NMEA sentences) to arrive.
4. Process an incoming sentence, which in my case involves printing out the latitude and longitude details in the "\$GPGGA" sentences.
5. Close the port when the input has finished.

It's possible to layer a `BufferedReader` over the port in task 2 since NMEA data is structured as ASCII text lines ending with `\n`. Then `BufferedReader.readLine()` can return each sentence as a string.

It's straightforward to choose which sentences to process because each one starts with a unique label (e.g. "\$GPGGA"). Also, a sentence can be pulled apart easily (with `String.split()`) since its arguments are separated by `,`'s.

The `SerialBufReader()` constructor performs the first three tasks – opening the port, setting up the input stream, and registering a listener.

```
// globals
private static final int BAUD_RATE = 4800;

private BufferedReader br;

public SerialBufReader(String comStr)
{
    // convert the serial port name into an ID
    CommPortIdentifier portId = null;
    try {
        portId = CommPortIdentifier.getPortIdentifier(comStr);
    }
    catch(NoSuchPortException e) {
        System.out.println("Could not find port: " + comStr);
        System.exit(0);
    }

    // check that it's a serial port
    if (portId.getPortType() != CommPortIdentifier.PORT_SERIAL) {
        System.out.println(comStr + " not a serial port");
        System.exit(0);
    }

    // open the port
    SerialPort serialPort = null;
    try {
        serialPort = (SerialPort) portId.open("SerialBufReader", 2000);
    }
    catch (PortInUseException e) {
        System.out.println(comStr + " already in use");
        System.exit(0);
    }

    // use this class as the serial port listener;
    // serialEvent() will be called
    try {
        serialPort.addEventListener(this);
    }
```



```

    }
    catch (TooManyListenersException e) {
        System.out.println(comStr + " has listener already");
        System.exit(0);
    }

    serialPort.notifyOnDataAvailable(true);
        // listen for data availability

    try { // set up port properties
        serialPort.enableReceiveTimeout(1000); // wait for 1 sec
        serialPort.enableReceiveThreshold(0); // no lower limit
        serialPort.setSerialPortParams(BAUD_RATE, SerialPort.DATABITS_8,
            SerialPort.STOPBITS_1, SerialPort.PARITY_NONE);
    }
    catch (UnsupportedCommOperationException e) {
        System.out.println(comStr + ": error while setting parameters");
        System.exit(0);
    }

    // layer a BufferedReader on top of the port
    try {
        br = new BufferedReader( new InputStreamReader(
            serialPort.getInputStream() ));
    }
    catch (IOException e) {
        System.out.println("Could not create reader for: " + comStr);
        System.exit(0);
    }

    System.out.println("Opened port " + comStr + " for input");
} // end of SerialBufReader()

```

The `SerialBufReader` class implements `SerialPortEventListener`, so is registered with the serial port:

```
serialPort.addEventListener(this);
```

It's possible to listen for multiple control events, such as parity errors and carrier detection. However, detecting data arrival is most useful for a GPS device. The event types of interest are added to the listener by 'notify' methods. For example:

```
serialPort.notifyOnDataAvailable(true);
```

makes the listener wake up when data is available for reading.

Various port properties are set, the most important being the baud rate. After some experimentation, I decided to make this 4800 rather than the 9600 reported by the `PortList` application. A reduced rate meant that data was cleanly received without non-printable noise characters.

The receive timeout and threshold were also decided upon by experimentation. The timeout specifies the number of milliseconds that must pass before a call to `read()` returns, while the threshold gives a lower bound for the number of bytes that `read()` should return.

The creation of a `BufferedReader` requires three layers on top of the port:

```
br = new BufferedReader( new InputStreamReader(
```

```
serialPort.getInputStream() );
```

For less well structured data (without newlines, for instance) it may be necessary for the programmer to read the serial port data byte-by-byte, and build up useable text himself. I include a short example of that style of programming later in this section.

When the NMEA sentence data arrives, the listener calls `serialEvent()`. After a quick check of the event type (which should be `SerialPortEvent.DATA_AVAILABLE`), the `BufferedReader` text is retrieved:

```
// global
private BufferedReader br;

public void serialEvent(SerialPortEvent event)
{
    if (event.getEventType() == SerialPortEvent.DATA_AVAILABLE) {
        try {
            String msg = br.readLine();
            if (msg == null) {
                System.out.println("input terminated");
                br.close();
                System.exit(0);
            }
            else {
                System.out.println("Msg len: " + msg.length() +
                                   " ; [" + msg + " ]");
                showPosition(msg);
            }
        }
        catch(IOException e) {
            System.out.println(e);
            System.exit(0);
        }
    }
    else
        System.out.println("Port event received: " + event);
} // end of serialEvent()
```

If `BufferedReader.readLine()` returns null then there's a problem, and the port is closed. Otherwise, the NMEA sentence is printed, and further processed in `showPosition()`.

The important sentences begin with "\$GPGGA" because they include location details. I want to extract the latitude and longitude arguments, which are underlined in the following example:

```
$GPGGA,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M,46.9,M,,*47
```

`showPosition()` uses `String.split()` to pull the sentence apart at the commas, and prints the third, fourth, fifth, and sixth arguments:

```
private void showPosition(String msg)
{
    String[] nmeaArgs = msg.split(",");
    if (nmeaArgs.length >= 6) {
```

```

    if (nmeaArgs[0].equals("$GPGGA")) // position fix NMEA sentence
        System.out.println("Lat: " + nmeaArgs[2] + " " + nmeaArgs[3] +
            "; Long: " + nmeaArgs[4] + " " + nmeaArgs[5]);
    }
} // end of showPosition()

```

A typical execution of SerialBufReader is shown in Figure 9.

```

C:\WINDOWS\system32\cmd.exe
> javac -cp "c:\rxtx\RXTXcomm.jar;." SerialBufReader.java
> java -cp "c:\rxtx\RXTXcomm.jar;." -Djava.library.path="c:\rxtx" SerialBufRe
r COM3
Stable Library
=====
Native lib Version = RXTX-2.1-7
Java lib Version = RXTX-2.1-7
Opened port COM3 for input
Msg len: 96; [746...N*77]
Msg len: 38; [$GPUTG,141.9,T.,M.000.6,N.001.1,K,N*09]
Msg len: 75; [$GPGGA,074611.079,0700.4342,N,10030.1132,E,0,00,3.4,34.3,M,-16.
,0000*48]
Lat: 0700.4342 N; Long: 10030.1132 E
Msg len: 37; [$GPGSA,0,1,,,,,5.6,3.4,4.4*34]
Msg len: 70; [$GPRMC,074611.079,0,0700.4342,N,10030.1132,E,000.6,141.9,271010
N*76]
Msg len: 38; [$GPUTG,141.9,T.,M.000.6,N.001.1,K,N*09]
Msg len: 75; [$GPGGA,074612.079,0700.4342,N,10030.1132,E,0,00,3.4,34.3,M,-16.
,0000*4B]
Lat: 0700.4342 N; Long: 10030.1132 E
Msg len: 37; [$GPGSA,0,1,,,,,5.6,3.4,4.4*34]
Msg len: 70; [$GPRMC,074612.079,0,0700.4342,N,10030.1132,E,000.6,141.9,271010
N*75]
Msg len: 38; [$GPUTG,141.9,T.,M.000.6,N.001.1,K,N*09]
Msg len: 75; [$GPGGA,074613.079,0700.4342,N,10030.1132,E,0,00,3.4,34.3,M,-16.
,0000*4A]
Lat: 0700.4342 N; Long: 10030.1132 E
Msg len: 37; [$GPGSA,0,1,,,,,5.6,3.4,4.4*34]
Msg len: 60; [$GPGSU,3,1,12,10,73,166,,28,48,141,,04,41,350,,02,41,300,*7D]
Msg len: 60; [$GPGSU,3,2,12,17,34,047,,05,31,207,,27,17,270,,09,12,282,*7F]
Msg len: 60; [$GPGSU,3,3,12,08,10,165,,13,08,096,,26,04,207,,12,01,326,*74]
> -

```

Figure 9. Executing SerialBufReader.

The reported \$GPGGA values are 0700.4342 N and 10030.1132 E, which are latitude and longitude 7 degrees, 0.4342' N and 100 degrees 30.1132' E. They can be converted to decimals by dividing the minutes parts by 60 to get latitude and longitude 7.007237 and 100.501887, which is outside my office. (A quick way to visualize such values is to type them (separated by a comma) into Google Maps.)

2.3. More Complex Input Processing

Sometimes it's not possible to use a `BufferedReader` to read the input. For instance, the device may be receiving non-ASCII data, or the messages don't have newline boundaries. In those cases, a single input stream can be layered over the port to allow the data to be read as bytes. It's up to the programmer to combine the bytes into more structured information.

The following code fragments show how this functionality can be implemented for the serial port reader. In the constructor, an input stream is connected to the port:

```
// global
private InputStream is;

// in the constructor
is = serialPort.getInputStream();
```

The listener method, `serialEvent()`, builds a text message byte-by-byte by calling `readMsg()`:

```
public void serialEvent(SerialPortEvent event)
{
    if (event.getEventType() == SerialPortEvent.DATA_AVAILABLE) {
        String msg = readMsg();
        System.out.println("Msg len: " + msg.length() +
                           "; [" + msg + "]);
    }
    else
        System.out.println("Port event received: " + event);
} // end of serialEvent()
```

```
private String readMsg()
{
    StringBuffer buf = new StringBuffer();
    int ch;
    while(true) {
        try {
            ch = is.read();
            if (ch == -1)
                break;
            if ((char)ch == '\n') // "end of message" char
                break;
            if (isAsciiPrintable(ch)) // is it printable?
                buf.append((char)ch);
            else
                buf.append('?'); // use '?' instead
        }
        catch(IOException e)
        { System.out.println(e); }
    }
    return buf.toString().trim(); // trim whitespace (\r, \n)
} // end of readMsg()
```

```
private boolean isAsciiPrintable(int ch)
{ return ch >= 32 && ch < 127; }
```

`readMsg()` gets to see every byte, so it's possible to use any byte as a message delimiter, including non-printable characters; this version looks for newlines to separate the messages.

The bytes are combined into a string, after changing non-printable characters to '?'s.

3. Using a GPS Library

Although it's possible to access the Canmore GPS receiver with a serial communications API, it makes more sense to employ one tailored for GPS input. The API will hide the low-level details of port setup and NMEA data extraction. In addition, the API should be able to support other types of GPS devices (e.g. those using Bluetooth), and other data formats (e.g. the Garmin protocols (<http://developer.garmin.com/schemas/>)).

I'll utilize the `gpsinput` package that's part of GPSylon (<http://www.tegmento.org/gpsylon/>). GPSylon can plot GPS data on maps downloaded from MapBlast or Expedia, but I won't be using those capabilities. I'll restrict myself to its `gpsinput` library, which can read NMEA and Garmin data from serial and Bluetooth GPS devices, from files, and even from GPS daemons running across a network.

I downloaded the latest GPSylon binary and source code from <http://sourceforge.net/projects/gpsmap/files/> (the source code includes the API documentation). I extracted the `gpsinput` JAR file (`gpsinput-0.5.3.jar`) from the binary code file, and I also needed the `log4j` JAR (a logging service useful for debugging). `Gpsinput` utilizes RXTX for its serial port communication, but I already had that installed from my earlier programming.

There's not a great deal of information about the `gpsinput` package aside from its API documentation. However, `gpsinput` is used by GPSylon's `GPSTool` command line utility, whose source can be examined. Also, the download includes a software design document for `GPSTool` (in `SW_Design.pdf`) which describes code fragments using `gpsinput`.

The following example, called `ReadGPS.java`, employs the `gpsinput` package to connect to the Canmore GPS receiver. It reports all the incoming messages, and prints latitude and longitude details when `LOCATION` GPS events occur.

```
public class ReadGPS
{
    private static final int BAUD_RATE = 4800;

    public static void main(String[] argc)
    {
        if (argc.length != 1) {
            System.out.println("Usage: run ReadGPS <com>");
            System.exit(0);
        }

        try {
            // create a processor suitable for NMEA data
            final GPSTDataProcessor gpsDataProc =
                new GPSTNmeaDataProcessor();

            // create a GPS device for a serial port
            Hashtable<String, Object> env =
                new Hashtable<String, Object>();
            env.put(GPSTSerialDevice.PORT_NAME_KEY, argc[0]);
            env.put(GPSTSerialDevice.PORT_SPEED_KEY,
                new Integer(BAUD_RATE));
            GPSTDevice gpsDev = new GPSTSerialDevice();
```

```

gpsDev.init(env);

// connect processor to device and open it
gpsDataProc.setGPSDevice(gpsDev);
gpsDataProc.open();
System.out.println("Opened GPS device");

// print out device info
System.out.println("GPS Info:");
String[] info = gpsDataProc.getGPSInfo();
for(int i=0; i < info.length; i++)
    System.out.println("  " + info[i]);

// create a raw data listener, to print all messages
gpsDataProc.addGPSRawDataListener( new GPSRawDataListener() {
    public void gpsRawDataReceived(char[] data, int offset,
                                   int length)
    { String msg = new String(data,offset,length).trim();
      System.out.println("Raw data (" + msg.length() +
                          "): [" + msg + "]);
    }
});

// create a listener for GPS location events
PropertyChangeListener listener =
    new PropertyChangeListener () {
    public void propertyChange(PropertyChangeEvent event)
    {
        Object value = event.getNewValue();
        String name = event.getPropertyName();
        System.out.println("Property Name: " + name);
        if (name.equals(GPSDataProcessor.LOCATION)) {
            GPSPosition gpspos = (GPSPosition)value;
            System.out.println("Lat | long: "+
                               gpspos.getLatitude() + " | " +
                               gpspos.getLongitude() );
        }
    }
};
gpsDataProc.addGPSDataChangeListener(
    gpsDataProc.LOCATION, listener);
}
catch(GPSException e)
{ e.printStackTrace(); }
catch(Exception e)
{ System.err.println(e); }
} // end of main()
} // end of ReadGPS class

```

The constructor creates a GPS processor for NMEA data (an instance of `GPSSNmeaDataProcessor`) which is connected to a serial port GPS device representing the Canmore receiver. Two listeners are created to monitor incoming data.

`GPSRawDataListener` is triggered whenever any kind of message arrives, which is useful during testing.

A `PropertyChangeListener` instance is associated with `LOCATION` GPS events, which correspond to the arrival of a NMEA "\$GPGGA" sentences. A sentence's latitude and longitude details are accessed using the `GPSPosition` class.

ReadGPS is passed the serial port name (COM3, as previously), and Figure 10 shows some typical output.

```
Raw data (59): [$GPGSA,A,3,24.32,20.11,08.17,13.23,04.07,50,,1.8,0.9,1.5*38]
Raw data (70): [$GPRMC,030522.511,A,0700.0686,N,10029.5222,E,000.0,086.7,041110,
,A*63]
Raw data (38): [$GPUTG,086.7,T,,M,000.0,N,000.0,K,A*04]
Raw data (75): [$GPGGA,030523.511,0700.0686,N,10029.5222,E,1,11,0.9,31.3,M,-16.3
,M,,0000*4C]
Property Name: location
Lat : long: 7.001143333333333 : 100.49203833333334
Raw data (59): [$GPGSA,A,3,24.32,20.11,08.17,13.23,04.07,50,,1.8,0.9,1.5*38]
Raw data (68): [$GPGSU,3,1,12,07,86,017,29,11,60,038,35,50,55,098,37,08,48,335,2
4*70]
```

Figure 10. ReadGPS Output.

The reported decimal latitude and longitude are 7.001143 and 100.492038 degrees, the location of my home.

4. Maps in Application

The previous example does a good job of printing the latitude and longitude, but I want to draw that information on a dynamic map, so it moves as the user travels around (as in Figures 1 and 2). One way of doing this is to have the necessary map 'tiles' stored locally on the PC, but I prefer to download them as needed from over the Web. This latter approach has the drawback that it requires the PC to maintain an Internet connection. Later on, I'll look at how to program with locally stored map tiles.

Before I start mapping my GPS coordinates, I want to explain how to render map tiles inside an application. "Application" is a key word here, since I don't want to execute the code as an applet inside a browser. This makes things surprisingly tricky since well-known map sources, such as Google Maps and Yahoo! Maps, are geared up for browser-based access. For instance, Google Maps prohibits the dynamic downloading of map information to applications.

A good answer is to move to an open source solution which doesn't impose silly terms of use. As a consequence, I'll be utilizing OpenStreetMap (OSM for short), a free, editable world map (<http://www.openstreetmap.org/>).

The OSM wiki describes JMapView (<http://wiki.openstreetmap.org/wiki/JMapView>), a subclass of JPanel, that adds an OSM map view. All the details of converting coordinates to pixels, caching tiles, and stitching them together on screen are hidden. JMapView features include zoom controls, mouse dragging, different map tile sources (e.g. Mapnik, Tiles@Home, Cyclemap), and map markers.

A popular alternative to JMapView is the similarly-named JXMapView (<https://swingx-ws.dev.java.net/>), part of the SwingX project. It also has similar functionality to JMapView, but can use a wider variety of image servers aside from OSM. There are two excellent articles on JXMapView programming by Josh Marinacci: "Building Maps into Your Swing Application with the JXMapView" (<http://today.java.net/pub/a/today/2007/10/30/building-maps-into-swing-app-with-jxmapviewer.html>) and "Mapping Mashups with the JXMapView" (<http://today.java.net/article/2007/11/09/mapping-mashups-jxmapviewer>).

I chose JMapView since it's simpler, while still doing what I need; in particular it only communicates with OSM.

The developers of JMapView don't make it easy for less experienced programmers to start using it. There's no pre-compiled binary available; instead you have to download the source code from a SVN repository (SVN is a version-control system) located at <http://svn.openstreetmap.org/applications/viewer/jmapviewer>. There's an ANT build script to compile the source, but it's missing a command for generating the API documentation. There's no other documentation, except for a good example, Demo.java by Jan Peter Stotz, in the JMapView download. The following code is a simplified version of Demo.java, called ShowMap.java.

The program is called from the command line with a user supplied latitude and longitude. For example:

```
java -cp "c:\JMapView\JMapView.jar; ." ShowMap 7.006835 100.502
```

The (7.006835 100.502) coordinate is displayed as a yellow dot in the center of the relevant OSM map, as shown in Figure 11.



Figure 11. ShowMap in Action.

The map can be dragged around by the mouse, and the zoom controls work as expected. ShowMap isn't connected to any GPS source, so the marker doesn't change its position relative to the map as I walk about.

ShowMap is defined as follows:

```
public class ShowMap extends JFrame
{
    // Robot building is the default position
    private static final double LAT_DEFAULT = 7.006835;
    private static final double LONG_DEFAULT = 100.5020;
```



```

public ShowMap(double lat, double lon)
{
    super("Show Map");

    Container c = getContentPane();
    c.setLayout( new BorderLayout() );

    JMapView map = new JMapView();
        // map panel, default size is 400x400
    c.add(map, BorderLayout.CENTER);

    JLabel helpLabel =
        new JLabel("<html>Use right mouse button to move,<br> " +
            "left double click or mouse wheel to zoom.",
                JLabel.CENTER);
    helpLabel.setFont(new Font("Serif", Font.PLAIN, 18));
    c.add(helpLabel, BorderLayout.SOUTH);

    // use a position marker to center the map
    map.addMapMarker(new MapMarkerDot(lat, lon));
    map.setDisplayToFitMapMarkers();
    map.setZoom(17); // so roads are visible

    setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    pack();
    setVisible(true);
} // end of ShowMap()

// -----

public static void main(String[] args)
{
    double lat = LAT_DEFAULT;
    double lon = LONG_DEFAULT;
    if (args.length !=2)
        System.out.println(
            "Usage: run ShowMap [ latitude longitude ]");
    else {
        lat = Double.parseDouble(args[0]);
        lon = Double.parseDouble(args[1]);
    }
    System.out.printf("Using (lat, long): (%.6f, %.6f)\n", lat, lon);
    new ShowMap(lat, lon);
} // end of main()

} // end of ShowMap class

```

JMapView is a subclass of JPanel, so can be added to a JFrame in the normal way. The only map-specific parts of the code are the three lines:

```

map.addMapMarker(new MapMarkerDot(lat, lon));
map.setDisplayToFitMapMarkers();
map.setZoom(17); // so roads are visible

```

MapMarkerDot is an implementation of the MapMarker interface, which can be subclassed to define different kinds of markers by redefining paint(). There's also a MapMarkerRectangle interface which differs from MapMarker in containing two latitude and longitudes, for the top-left and bottom-right coordinates of the rectangle.

Missing from my code are many complicated features supported by JMapView, including tile downloading and caching, tile combination and rendering, and the management of the zoom controls and mouse.

5. Combining GPS and Mapping

Having coded examples for GPS reading (ReadGPS.java) and map rendering (ShowMap.java), it's time to combine them.

ShowGPS is shown in operation in Figures 1 and 2. The arrival of GPS location data triggers the replacement of the current MapMarkerDot on the map with one in the new GPS position. This causes the map to be redrawn, which may require JMapView to download some new map tiles. Note from the figures, that I've switched off the zoom slider.

The ShowGPS() constructor looks a lot like the ShowMap code since it's utilizing the JMapView panel in the same way:

```
// globals
// default marker position (the Robot building)
private static final double LAT_DEFAULT = 7.006835;
private static final double LONG_DEFAULT = 100.5020;

private JMapView map; // the map display panel

private MapMarkerDot currPos; // current location on the map
private double currLat, currLong;

public ShowGPS(String comStr)
{
    super("Show GPS");
    Container c = getContentPane();
    c.setLayout( new BorderLayout() );

    map = new JMapView(); // map panel, default size is 400x400
    c.add(map, BorderLayout.CENTER);

    // start with a default location
    currLat = LAT_DEFAULT;
    currLong = LONG_DEFAULT;
    System.out.printf("Starting Position: (%.6f, %.6f)\n",
                      currLat, currLong);

    currPos = new MapMarkerDot(currLat, currLong);
    map.addMapMarker(currPos);
    map.setDisplayToFitMapMarkers(); // center the map on the marker
    map.setZoom(17); // so roads are visible
    map.setZoomContolsVisible(false);

    monitorGPS(comStr); // start monitoring the GPS receiver

    setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    pack();
    setVisible(true);
}
```

```
} // end of ShowGPS()
```

The important addition is the call to `monitorGPS()` which starts monitoring the GPS receiver for messages.

```
// globals
// hardwired baud rate
private static final int BAUD_RATE = 4800;

private void monitorGPS(String comStr)
{
    try {
        // create processor for NMEA data
        final GPSTDataProcessor gpsDataProc = new GPSTNmeaDataProcessor();

        // create GPS device for serial port
        Hashtable<String, Object> env =
            new Hashtable<String, Object>();
        env.put(GPSTSerialDevice.PORT_NAME_KEY, comStr);
        env.put(GPSTSerialDevice.PORT_SPEED_KEY,
            new Integer(BAUD_RATE));
        GPSTDevice gpsDev = new GPSTSerialDevice();
        gpsDev.init(env);

        // connect processor to device and open it
        gpsDataProc.setGPSTDevice(gpsDev);
        gpsDataProc.open();
        System.out.println ("Opened GPS device");

        /* create a listener for GPS location events;
           propertyChange() is called when an event arrives */
        gpsDataProc.addGPSTDataChangeListener(
            gpsDataProc.LOCATION, this);
    }
    catch(Exception e)
    {
        System.err.println(e);
        System.exit(0);
    }
} // end of monitorGPS()
```

`monitorGPS()` sets up a connection to the GPS receiver and a listener for `LOCATION` events, and so its code is quite similar to `ReadGPS.java`.

The arrival of a `LOCATION` event triggers a call to `propertyChange()`, which is defined by the `ShowGPS` class itself.

`propertyChange()` extracts the latitude and longitude values from the GPS data, and uses them to update the map display.

```
public void propertyChange(PropertyChangeEvent event)
// called when the GPS receiver gets a location event
{
    Object value = event.getNewValue();
    String name = event.getPropertyName();

    if (name.equals(GPSTDataProcessor.LOCATION)) {
        GPSTPosition gpspos = (GPSTPosition)value;
    }
}
```

```

double newLat = gpspos.getLatitude();
double newLong = gpspos.getLongitude();

if ((newLat == 0) && (newLong == 0)) // ignore (0,0) pos
    return;

// update the map position
currLat = newLat; currLong = newLong;
map.removeMapMarker(currPos);
currPos = new MapMarkerDot(newLat, newLong);

map.addMapMarker(currPos);
map.setDisplayToFitMapMarkers();
map.repaint();
}
} // end of propertyChange()

```

The current map marker is removed, and a new one added at the new position.

Updating the Map Less Often

One way of improving ShowGPS is to reduce the number of map redraws, which are currently carried out after every GPS LOCATION event, even if the user hasn't moved. This improvement can be achieved by comparing the new user position with the current one, and only changing the marker (and so redrawing the map) if the two positions are sufficiently far apart. The following code fragment would need to be added to propertyChange():

```

double distMoved = GeoMath.distanceApart(newLat, newLong,
                                           currLat, currLong);
if (distMoved < MIN_MOVE_DIST) // ignore a small move
    return;

```

`GeoMath.distanceApart()` calculates the surface distance between two coordinates, which is a little difficult because of the (near) spherical nature of the Earth. For those interested, the code uses the Haversine formula, which is explained at http://en.wikipedia.org/wiki/Haversine_formula and <http://mathforum.org/library/drmath/view/51879.html>.

```

// in the GeoMath class
public static double distanceApart(double lat1, double long1,
                                   double lat2, double long2)
{ // convert latitude and longitudes to radians
  double diffLat = Math.toRadians(lat2-lat1);
  double diffLong = Math.toRadians(long2-long1);
  double h = haversin(diffLat) +
            ( Math.cos(Math.toRadians(lat1)) *
              Math.cos(Math.toRadians(lat2)) *
              haversin(diffLong) );
  double dist = 2.0 * EARTH_RADIUS * Math.asin( Math.sqrt(h) );
  return dist;
} // end of distanceApart()

private static double haversin(double angle)

```

```
// angle is in radians
{ return Math.sin(angle/2) * Math.sin(angle/2); }
```

6. Dealing with Network Problems

A problem with ShowGPS is its reliance on two networks – GPS satellites and wireless connection to the Internet so that OSM tiles can be downloaded.

JMapViewer deals with network connectivity problems by rendering unavailable map tiles as red "X"s, as in Figure 12.

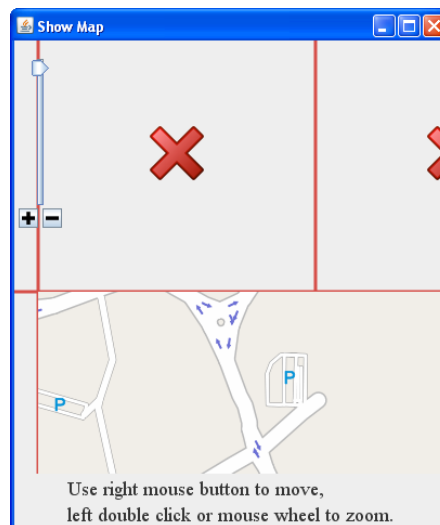


Figure 12. A Map with Missing Tiles.

Once a tile has been downloaded, it's cached in memory so it won't need to be downloaded again. JMapViewer implements this behavior by delegating the task of accessing a tile to a `TileController` object. `TileController` relies on three classes: `TileLoader`, `TileSource`, and `TileCache`.

`TileController` doesn't use `TileLoader` directly (it's an interface), but a subclass, `OsmTileLoader`, to download tiles from OSM. OSM offers several different map tile sources (e.g. Mapnik, Tiles@Home, and Cyclemap) controlled by `TileSource`. After a map tile has been downloaded, it's stored in an in-memory `TileCache` object, implemented as a hash table by `MemoryTileCache`.

JMapViewer implements a policy for map tile downloads in its `paintComponent()` method. The tile containing the current latitude and longitude position is rendered, as well as tiles spreading out in a spiral from that tile until the edges of the viewing panel are reached. Access to a tile is mediated by `TileController`, which determines whether a cached in-memory tile is used or the tile has to be downloaded.

JMapViewer (through `TileController`) employs `OsmTileLoader` for downloading tiles. However, the JMapViewer package also includes a second loader, called `OsmFileCacheTileLoader`. It loads tiles from OSM via HTTP *and* saves all the loaded tiles as files in a local directory on the PC. If a tile is present in this file cache then it won't be loaded from OSM again.

This file cache is different from the in-memory TileCache object in that it lives on after the application has terminated. When the application is restarted, it won't have to download tiles again if it is using OsmFileCacheTileLoader as its tile loader.

I realize all of this is a bit confusing, but I'm really only talking about three classes: OsmFileCacheTileLoader (a tile loader *and* file cache), TileSource (a Web source for map tiles), and MemoryTileCache (an in-memory tile cache). An example using these three classes will help to clarify matters.

7. Stitching Tiles Together

StitchTiles.java downloads map tiles from OSM without rendering them. It retrieves all the tiles neighboring a latitude and longitude supplied by the user, utilizing OsmFileCacheTileLoader so the tiles are stored in a file cache. They're also loaded into memory inside a MemoryTileCache object, so their images can be 'stitched together' to form a single large image which is stored in its own file.

The file cache means that when StitchTiles is run again, it will utilize the local tiles rather than download them again, with a visible increase in speed. Also, the file cache can be employed even when the network is disconnected.

StitchTiles is called like so:

```
java -cp "c:\JMapView\JMapView.jar;." StitchTiles
      7.006835 100.502
```

This downloads the tile containing latitude 7.006835, longitude 100.502 (my office building), and the tiles immediately neighboring it. The resulting *nine* tiles are stitched together and the image stored in tilesIm.png, which is shown in Figure 13.



Figure 13. The Stitched Map (at quarter scale).

The PC's file cache for the tiles is shown in Figure 14.

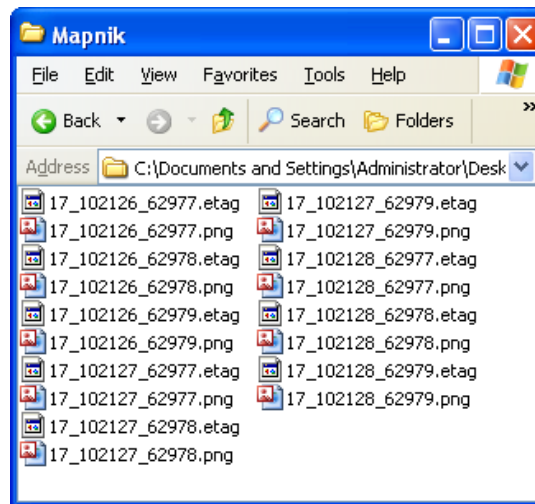


Figure 14. The Tile Files Cache.

Each tile is represented by a PNG and ETAG file, whose names are generated from the tiles' IDs.

If the same call is made again to `StitchTiles`, there's a noticeable speed increase since none of the tiles need to be downloaded. Also, if the location argument is changed then `OsmFileCacheTileLoader` will only retrieve tiles which have not yet been cached. This means that subsequent calls to `StitchTiles` are somewhat faster depending on their proximity to earlier locations.

The `StitchTiles()` constructor creates instances of `OsmTileSource`, `OsmFileCacheTileLoader`, and `MemoryTileCache`, then converts the latitude and longitude into its corresponding tile ID. The ID is employed to load the tile and its neighbors, which are combined to form a single image.

```
// globals
private static final int ZOOM = 17;    // so roads are visible

private static final String TILES_DIR = "/tiles";
    // the local directory for storing the downloaded tiles

private OsmTileSource.Mapnik tileSource;
    // Mapnik is the Web-based source for the maps
private OsmFileCacheTileLoader tileLoader;
    // loader which uses the Web or a file cache
private MemoryTileCache tileCache;
    // in-memory data structure for tiles

public StitchTiles(double lat, double lon)
{
    tileSource = new OsmTileSource.Mapnik();

    tileLoader = new OsmFileCacheTileLoader(this,
        new File( System.getProperty("user.dir") + TILES_DIR));

    tileCache = new MemoryTileCache();

    // Get tile ID (x,y) for the user's specified location
    int tileSize = tileSource.getTileSize();
```

```

int tilex = OsmMercator.LonToX(lon, ZOOM)/tileSize;
int tiley = OsmMercator.LatToY(lat, ZOOM)/tileSize;

loadTiles(tilex, tiley);
reportCache(tilex, tiley);
saveTilesImage(tilex, tiley, tileSize);
} // end of StitchTiles()

```

The first argument of the `OsmFileCacheTileLoader` instance is the object that will be notified whenever a tile download finishes. The second argument is the directory where tiles are cached.

The `OsmMercator.LonToX()` and `OsmMercator.LatToY()` methods together generate a tile ID (a (x, y) pair) from the latitude and longitude. This (tilex, tiley) ID represents the center tile, and is used in later stages of the code.

7.1. Loading the Tiles

Once the (tilex, tiley) ID for the center tile is known, the neighboring tiles are easy to identify since each is 1 tile unit away along the x and/or y- axes. Each tile is loaded by calling `loadTile()` from `loadTiles()`.

```

// globals
private static final int TILES_RADIUS = 1;
    // tiles radius around the center tile for downloading
    // '1' means 8 tiles around center, so 9 are downloaded altogether

private AtomicInteger jobCount = new AtomicInteger(0);

private void loadTiles(int tilex, int tiley)
// load tiles around this tile into the tiles cache
{
    for (int x = (tilex - TILES_RADIUS);
        x <= (tilex + TILES_RADIUS); x++)
        for (int y = (tiley - TILES_RADIUS);
            y <= (tiley + TILES_RADIUS); y++)
            loadTile(x,y);

    // wait for all the downloads to finish
    while (jobCount.get() != 0) {
        try {
            Thread.sleep(500);
        }
        catch (InterruptedException ex){}
    }
} // end of loadTiles()

```

`JMapView` can download several tiles concurrently, by creating a thread for each one, managed by an instance of `JobDispatcher`. A drawback of `JobDispatcher` is that it doesn't offer a way to see how many threads are currently running, pending, or finished. My solution is an atomic integer called `jobCount`: when it reaches 0, all the threads have finished, and the waiting `loadTiles()` call can finally return.

`loadTile()` creates a `JobDispatcher` thread for downloading a tile, and increments the atomic integer.


```

//globals
// for managing the concurrent download tasks
private JobDispatcher jobDispatcher = JobDispatcher.getInstance();
private AtomicInteger jobCount = new AtomicInteger(0);

private void loadTile(int tx, int ty)
{
    Tile tile = new Tile(tileSource, tx, ty, ZOOM);
    tileCache.addTile(tile);
    System.out.println("Added " + tile + " to cache");

    if (!tile.isLoaded()) {
        System.out.println("Starting loading " + tile);
        jobDispatcher.addJob(
            tileLoader.createTileLoaderJob(tileSource, tx, ty, ZOOM) );
            // do a local file read or web download
        jobCount.incrementAndGet();
    }
    else
        System.out.println("Cached tile already loaded");
} // end of loadTile()

```

loadTile() creates a reference to the *in-memory* cache for the tile, and checks if the tile's details have previously been loaded into that cache. If they haven't then the tile loader is called. The loader is an instance of OsmFileCacheTileLoader, and so will first try to load the tile from the *file* cache before resorting to network access.

When the tile loader has finished loading a particular tile, it calls tileLoadingFinished():

```

public void tileLoadingFinished(Tile tile, boolean success)
{ System.out.println(" Finished loading " + tile + ": " + success);
  jobCount.decrementAndGet();
}

```

tileLoadingFinished() is called by each threaded download, and so it's possible for multiple decrements to be applied to jobCount at the same time. Even worse, decrements may occur at the same time that jobCount is being incremented by tileLoader(). It's to avoid these race conditions that jobCount is declared as an atomic integer rather than just an int. It's not possible for an atomic variable to have multiple operations, such as incrementing and decrementing, applied to it at the same time.

7.2. Reporting on the In-memory Tile Cache

reportCache() is called after loadTiles() is finished, which means that the in-memory cache has been completely filled with tiles (either downloaded from the Web or from the file cache). reportCache() shows how tiles can be examined inside the cache.

```

private void reportCache(int tilex, int tiley)
{
    System.out.println("\nSize of cache: " + tileCache.getTileCount());
    Tile t;
    for (int x = (tilex - TILES_RADIUS);

```

```

        x <= (tilex + TILES_RADIUS); x++)
    for (int y = (tiley - TILES_RADIUS);
        y <= (tiley + TILES_RADIUS); y++) {
        t = tileCache.getTile(tileSource, x, y, ZOOM);
        System.out.println(" " + t + " load status: " + t.isLoaded());
        System.out.println(" " + t.getUrl());
    }
} // end of reportCache()

```

TileCache.getTile() identifies a tile by its tile source, its (x, y) ID, and the zoom factor.

7.3. Creating the Stitched Image

Building a combined image is a matter of obtaining each tile's image (with Tile.getImage()) and drawing them into a graphics context linked to an empty BufferedImage. The result is written out to a PNG file.

```

// globals
private static final String IMG_FNM = "tilesIm.png";
    // file for storing the combined tiles image

private void saveTilesImage(int tilex, int tiley, int tileSize)
// construct and save a single image made from all the tiles
{
    System.out.println();
    BufferedImage tilesIm = combineTiles(tilex, tiley, tileSize);
    try { // Write image to file
        ImageIO.write(tilesIm, "png", new File(IMG_FNM));
        System.out.println("Saved tiles image to " + IMG_FNM);
    }
    catch (IOException e)
    { System.out.println("Could not save image to " + IMG_FNM); }
} // end of saveTilesImage()

private BufferedImage combineTiles(int tilex, int tiley,
    int tileSize)
// combine all the tile images into a single image
{
    int widthTiles = (TILES_RADIUS*2)+1;

    // create empty image
    BufferedImage im =
        new BufferedImage(widthTiles*tileSize, widthTiles*tileSize,
            BufferedImage.TYPE_INT_ARGB);
    Graphics2D g2d = im.createGraphics();

    Tile t;
    int xPos = 0;
    int yPos;
    // stitch tiles together into image column-by column, left-to-right
    for (int x = (tilex - TILES_RADIUS);
        x <= (tilex + TILES_RADIUS); x++) {
        yPos = 0;
        for (int y = (tiley - TILES_RADIUS);

```

```
        y <= (tiley + TILES_RADIUS); y++) {
    t = tileCache.getTile(tileSource, x, y, ZOOM);
    g2d.drawImage(t.getImage(), xPos, yPos, null);
    yPos += tileSize;
}
xPos += tileSize;
}
g2d.dispose();
return im;
} // end of combineTiles()
```

combineTiles() calls TileCache.getTile() in a similar manner to reportCache(), and draws each tile's image into the graphics context at a specific location. The nested loops create the combined image a column at a time, moving left-to-right.

8. Nearby Pictures

Photo sharing sites, such as Flickr, allow images to include geographical metadata, such as latitude, longitude, and altitude (it's called geotagging or geocoding). One popular site is Panoramio (<http://www.panoramio.com/>), whose pictures often appear inside Google Maps and Google Earth.

Panoramio offers a Web service which returns JSON-formatted details about photos located within a specified geographical area. Its API is described at <http://www.panoramio.com/api/data/api.html>.

My SearchPanoramio.java application looks for Panoramio images located close to a supplied latitude and longitude. At most 20 matches are listed (descriptions and URLs), and the picture nearest to the location is downloaded and saved in a local PNG file.

For example, the following search looks for images within 200 meters of my office:

```
java -cp "json.jar;." SearchPanoramio 7.006835 100.502 200
```

The search returns eight matches, and the closest (at 7.006704, 100.501127) was saved (see Figure 15).



Figure 15. The Panoramio Photo Nearest to my Office.

Panoramio has detailed display requirements for its photos (see the "Requirements" section at <http://www.panoramio.com/api/data/api.html>), which include showing the Panoramio logo and creating links to the user and photo pages. These assume that the picture will be displayed in a browser, where hyperlinks can be supported, but that's not possible in SearchPanoramio.java. Instead, I've modified the downloaded image: the logo appears at the top-left corner, photo details at the top-right, and author information at the bottom-left. Nevertheless, this is probably insufficient for a commercial application involving Panoramio images. Panoramio's legal page (http://www.panoramio.com/help/policies_legalities#policy_qa) states that you must obtain explicit permission from the photographer to use, copy, print, or *download* a photo.

The basic structure of SearchPanoramio.java is the same as the Web services examples I described in Chapter 33 (<http://fivedots.coe.psu.ac.th/~ad/jg/ch33/>). It utilizes the JSON library and my WebUtils class to extract information from the JSON data that Panoramio returns. I won't explain those details again, so please have a look at that chapter if you're not familiar with JSON.

SearchPanoramio starts by extracting a latitude, longitude, and radius from the command line. After searching for an image, it saves it in a PNG file.

```
// globals
private static final int NEAR_DIST = 500; // default radius (ms)
private static final String IMG_FNM = "nearest.png";

public static void main(String args[])
{
    if ((args.length < 2) || (args.length > 3)) {
        System.err.println("Usage: run SearchPanoramio latitude
                           longitude [distance radius]");
        System.exit(1);
    }

    double lat = Double.parseDouble(args[0]);
```

```

double lon = Double.parseDouble(args[1]);
int radius = NEAR_DIST;
if (args.length == 3)
    radius = Integer.parseInt(args[2]);

BufferedImage im = imagesSearch(lat, lon, radius);
if (im == null)
    System.out.println("No image to save");
else {
    try {
        ImageIO.write(im, "png", new File(IMG_FNM));
        System.out.println("Saved image to " + IMG_FNM);
    }
    catch (IOException e)
    { System.out.println("Could not save image to " + IMG_FNM); }
}
} // end of main()

```

A typical query to the Panoramio Web service has the form:

```

http://www.panoramio.com/map/get_panoramas.php?
    set=public&
    from=0&to=20&
    miny=7.005&minx=100.500&maxy=7.009&maxx=100.504&
    size=medium"

```

"from" and "to" extract a range of photos based on how recently they were added to Panoramio. "minx", "miny", "maxx", "maxy" delimit the search rectangle (minimum longitude and latitude, maximum longitude and latitude).

The JSON result consists of an array of tuples, one tuple for each image. For instance:

```

"count": 82,
"has_more": false,
"photos": [
  {
    "height": 375,
    "latitude": 7.005122,
    "longitude": 100.501492,
    "owner_id": 1656012,
    "owner_name": "nurholis",
    "owner_url": "http://www.panoramio.com/user/1656012",
    "photo_file_url": "http://mw2.google.com/
      mw-panoramio/photos/medium/10085216.jpg",
    "photo_id": 10085216,
    "photo_title": "Faculty of Agro-Industry",
    "photo_url": "http://www.panoramio.com/photo/10085216",
    "upload_date": "10 May 2008",
    "width": 500
  },
  {
    "height": 333,
    "latitude": 7.008333,
    "longitude": 100.502619,
    "owner_id": 1656012,
    "owner_name": "nurholis",
    "owner_url": "http://www.panoramio.com/user/1656012",
    "photo_file_url": "http://mw2.google.com/

```

```

        mw-panoramio/photos/medium/11210175.jpg",
        "photo_id": 11210175,
        "photo_title": "Bangku pinggir danau",
        "photo_url": "http://www.panoramio.com/photo/11210175",
        "upload_date": "14 June 2008",
        "width": 500
    },
    :
    : // more tuples
    ]
}

```

In my tests, the initial count field doesn't seem to bear much relationship to the actual number of tuples returned.

The hardest part about coding against the Panoramio interface is the calculation of the latitude and longitude search rectangle. My solution (in GeoMath. `boundingBoxCoords()`) is based on code by Philip Matuschek at <http://janmatuschek.de/LatitudeLongitudeBoundingCoordinates>. It generates a rectangle with the supplied location at the center, and edges the specified distance away from that center. The tricky aspect is that the minimum and maximum longitudes are not on the same circle of latitude as the one supplied by the user (see Figure 16).

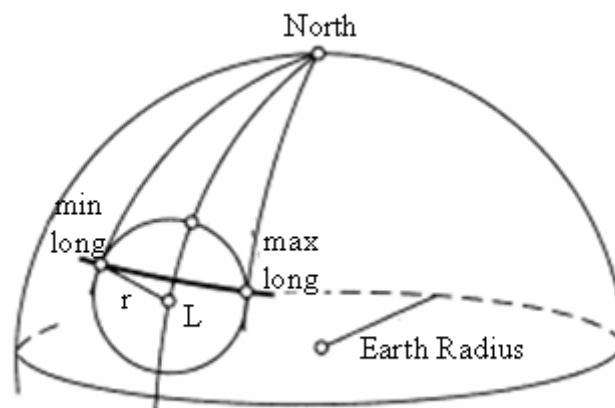


Figure 16. Minimum and Maximum Longitudes for L.

In Figure 16, L is the supplied location, and r the distance from that point to the edges of the bounding box. Due to the curvature of the earth, the maximum longitudinal distances are not on the same circle of latitude as L, but are located further North.

The `imagesSearch()` code in `SearchPanoramio`:

```

private static BufferedImage imagesSearch(double lat,
                                         double lon, int radius)
/* query Panoramio for images close to (lat,lon)
   and return the closest. */
{
    System.out.printf("Searching Panoramio close to
                      (%.6f, %.6f)\n", lat, lon);
    System.out.println("Radius (m): " + radius);
}

```

```

double[] bbox = GeoMath.boundingBoxCoords(lat, lon, radius);
System.out.printf("Bounding box: Min: (%.6f, %.6f);
                  Max:(%.6f, %.6f)\n\n",
                  bbox[0] , bbox[1], bbox[2], bbox[3]);
try {
    String urlStr = "http://www.panoramio.com/map/get_panoramas.php?"
        "set=public&from=0&to=20&" + // get first 20 images
        "miny=" + bbox[0] + "&minx=" + bbox[1] + "&" +
        "maxy=" + bbox[2] + "&maxx=" + bbox[3] + "&" +
        "size=medium";
    String jsonStr = WebUtils.webGetString(urlStr);

    JSONObject json = new JSONObject(jsonStr);
    WebUtils.saveString("temp.json", json.toString(2) );

    PanoramioItem[] items = extractMatches(json);
    if (items != null) {
        int itemIdx = selectClosestItem(items, lat, lon);
        System.out.println("Closest Photo: " +
                           items[itemIdx].getPhotoTitle());
        System.out.printf("(lat,long): (%.6f,%.6f)\n",
                           items[itemIdx].getLatitude(),
                           items[itemIdx].getLongitude() );
        return items[itemIdx].getImage();
    }
}
catch (Exception e)
{ System.out.println(e); }
return null;
} // end of imagesSearch()

```

extractMatches() extracts the data tuples from the JSON result, creating a **PanoramioItem** object for each one; the items are returned in an array.

```

private static PanoramioItem[] extractMatches(JSONObject json)
{
    try {
        int count = Integer.parseInt( json.getString("count"));
        if (count > 0) {
            JSONArray ja = json.getJSONArray("photos");
            int numMatches = ja.length();
            System.out.println("\nNo. of matches: " + numMatches + "\n");
            if (numMatches == 0)
                return null;
            PanoramioItem[] items = new PanoramioItem[numMatches];

            for (int i = 0; i < numMatches; i++) {
                items[i] = new PanoramioItem( ja.getJSONObject(i) );

                System.out.print((i+1) + ". ");
                System.out.println("Photo Title: " +
                                    items[i].getPhotoTitle());
                System.out.printf("    (lat,long): (%.6f,%.6f)\n",
                                    items[i].getLatitude(), items[i].getLongitude() );
                System.out.println("    URL: " +
                                    items[i].getPhotoFileURL() + "\n");
            }
            return items;
        }
    }
    else

```

```

        System.out.println("\nNo matches found");
    }
    catch (Exception e)
    { System.out.println(e);
      System.exit(1);
    }

    return null;
} // end of extractMatches()

```

extractMatches() also lists out each item's title, latitude, longitude, and URL.

A PanoramioItem object holds all the Panoramio data about an image. The class has numerous get methods, one for each image attribute, as can be seen in PanoramioItem's class diagram in Figure 17.

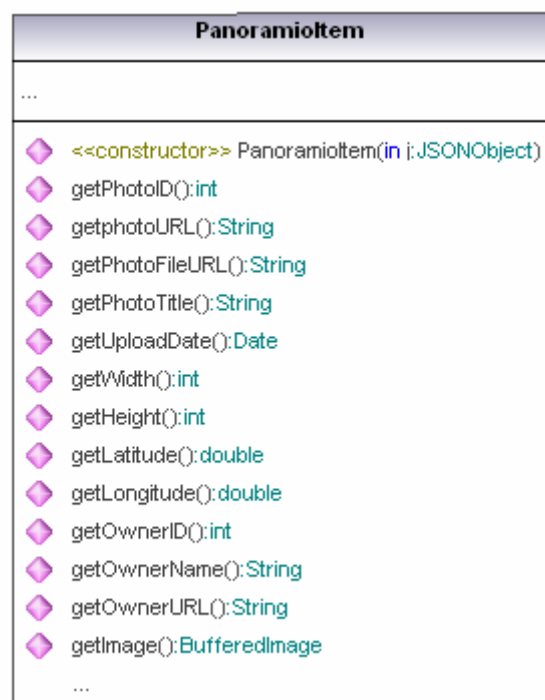


Figure 17. PanoramioItem Class Diagram.

getImage() is the most interesting PanoramioItem method: it downloads the image stored at the photo URL address, and adds the Panoramio display information (see Figure 15 for an example).

```

// globals
private String photoURL, photoFileURL

public BufferedImage getImage()
{
    System.out.println("Downloading image at:\n\t" + photoFileURL);
    URL url = null;
    BufferedImage photoIm = null;
    try {
        url = new URL(photoFileURL);
    }
}

```



```

    photoIm = ImageIO.read(url);
}
catch (IOException e)
{ System.out.println("Problem downloading"); }

if (photoIm == null)
    return null;
else
    return addRequirements(photoIm);
} // return getImage()

```

addRequirements() creates a new **BufferedImage** with its own graphics context and writes the picture into it, followed by the logo and various bits of text. I employ a **FontMetrics** object to calculate the width of the text so it can be positioned close to the image's edges.

```

// globals
private static final String LOGO_FNM = "panoramioLogo.png";

// image details
private int photoID;
private String photoURL, photoTitle;
private int width, height; // of image

private BufferedImage addRequirements(BufferedImage photoIm)
{
    // load the logo
    BufferedImage logoIm = null;
    try{
        logoIm = ImageIO.read(new File(LOGO_FNM));
    }
    catch(IOException e){
        System.out.println("Could not load logo from " + LOGO_FNM);
    }

    // build the new image
    BufferedImage im = new BufferedImage(width, height,
        BufferedImage.TYPE_INT_ARGB);
    Graphics2D g2d = im.createGraphics();

    FontMetrics fm = g2d.getFontMetrics();

    g2d.drawImage(photoIm, 0, 0, null); // original image

    // add logo to image
    g2d.drawImage(logoIm, 5, 5, null); // logo at top-left

    // transparent white for highlighting the text
    Color bgColor = new Color(1, 1, 1, 0.5f);

    // add top-right text
    int textWidth = fm.stringWidth(photoURL);
    g2d.setPaint(bgColor); // draw background box
    g2d.fillRect(width-7-textWidth, 3,
        textWidth+6, 2*(4+fm.getHeight()+2));
    g2d.setPaint(Color.BLACK);

    g2d.drawString(photoURL,

```

```

        width-5-textWidth, 2*(4+fm.getHeight())); // 2nd line
textWidth = fm.stringWidth(photoTitle);
g2d.drawString(photoTitle,
        width-5-textWidth, 4+fm.getHeight()); // 1st line

// add bottom-left text
// more calls to drawString() ...

g2d.dispose();
return im;
} // end of addRequirements()

```

Back in SearchPanoramio, the array of PanoramioItem objects is searched for the item closest to the user's location.

```

private static int selectClosestItem(PanoramioItem[] items,
                                     double lat, double lon)
// search the Panoramio results for the image closest to (lat,lon)
{
    int itemIdx = 0;
    double minDist = GeoMath.distanceApart(lat, lon,
                                           items[itemIdx].getLatitude(),
                                           items[itemIdx].getLongitude());
    for (int i=1; i < items.length; i++) {
        double distApart = GeoMath.distanceApart(lat, lon,
                                                  items[i].getLatitude(), items[i].getLongitude());
        if (distApart < minDist) {
            minDist = distApart;
            itemIdx = i;
        }
    }
    return itemIdx;
} // end of selectClosestItem()

```

The distance between the user's position and the image is calculated with `GeoMath.distanceApart()`, which was utilized earlier in `ShowGPS.java`.

9. Geotagging Images

The images at Panoramio are already tagged with geographical information, but how can we create our own geotagged photos suitable for uploading to Panoramio or Flickr? One solution is to use the sites' own editing tools. Another is to invest in a camera or mobile phone that adds location details automatically when a picture is taken. What about a programming solution? How can I write code that adds latitude and longitude information to an image?

The most common approach is to modify the Exchangeable image file format (Exif) metadata of a JPG (http://en.wikipedia.org/wiki/Exchangeable_image_file_format). This metadata isn't visible in the picture itself, but is stored in the form of tags (key/value pairs) in reserved areas inside the image's binary.

There are Exif tags for a wide range of topics including: date and time information, camera settings such as orientation and shutter speed, copyright details and, most importantly for my needs, GPS data such as latitude, longitude, and altitude. A drawback of Exif is that it isn't supported by the JPEG 2000, PNG, or GIF graphics formats.

I'll utilize two approaches for reading/writing GPS data: a command-line tool called ExifTool by Phil Harvey (<http://www.sno.phy.queensu.ca/~phil/exiftool/>), and the Sanselan API (<http://commons.apache.org/sanselan/>) which can read and write a variety of image formats, not only Exif metadata.

9.1. Using ExifTool

ExifTool can read and write several forms of metadata, including Exif, IPTC, XMP, JFIF, and GeoTIFF. It's implemented as a Perl library, but the Windows and Mac downloads are standalone executables (<http://www.sno.phy.queensu.ca/~phil/exiftool/>).

I retrieved the Windows standalone, unzipped the executable, and renamed it to `exiftool.exe` so it could be used from the command line.

Listing a file's metadata is a simple `exiftool` call:

```
exiftool engSign.jpg
```

The output is shown in Figure 18.

```

C:\Documents and Settings\Administrator\Desktop\Geotagging
> exiftool engSign.jpg
ExifTool Version Number      : 8.37
File Name                    : engSign.jpg
Directory                   : .
File Size                    : 55 kB
File Modification Date/Time  : 2010:11:03 15:45:52+07:00
File Permissions            : rw-rw-rw-
File Type                   : JPEG
MIME Type                   : image/jpeg
Exif Byte Order              : Little-endian (Intel, II)
Compression                 : Uncompressed
Photometric Interpretation   : RGB
Planar Configuration        : Chunky
Modify Date                 : 2010:11:03 15:45:50
Y Cb Cr Positioning         : Centered
Orientation                 : Horizontal (normal)
X Resolution                 : 72
Y Resolution                 : 72
Resolution Unit              : inches
Thumbnail Offset            : 358
Thumbnail Length            : 7120
JFIF Version                : 1.01
XMP Toolkit                 : XMP Core 4.1.1
Image Width                 : 500
Image Height                : 375
Encoding Process            : Baseline DCT, Huffman coding
Bits Per Sample             : 8
Color Components            : 3
Y Cb Cr Sub Sampling        : YCbCr4:2:2 (2 1)
GPS Latitude                : 7 deg 0' 28.18" N
GPS Latitude Ref            : North
GPS Longitude               : 100 deg 30' 2.64" E
GPS Longitude Ref           : East
GPS Position                : 7 deg 0' 28.18" N, 100 deg 30' 2.
Image Size                  : 500x375
Thumbnail Image             : <Binary data 7120 bytes, use -b o
act>

```

Figure 18. Reading Exif Metadata with ExifTool.

The GPS-related information appears towards the end of the list in Figure 18.

For people averse to command-line tools, most graphics viewers support the reading of metadata. For example, the same image's Exif information displayed by XnView (<http://www.xnview.com/>) is shown in Figure 19.

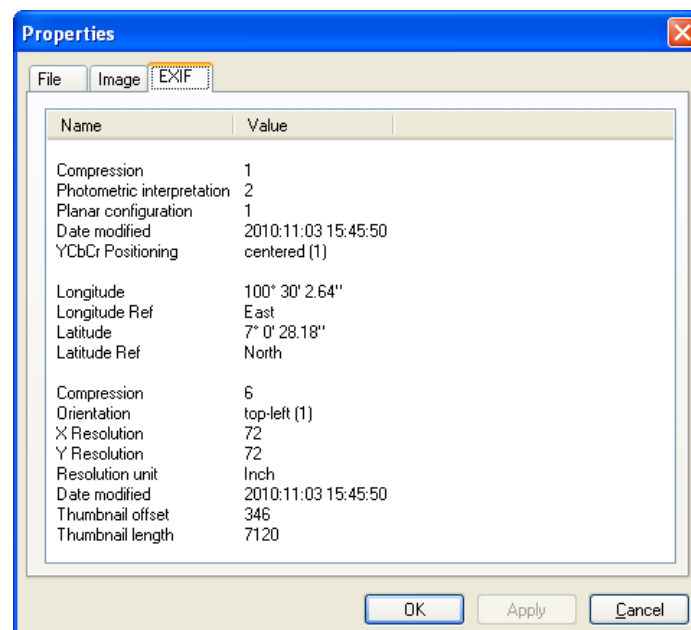


Figure 19. Reading Exif Metadata with XnView.

A nice feature of XnView is its ability to pass the Exif location information to GeoHack, and from there to Google Maps or OpenStreetMap (OSM). Figure 20 shows the OSM map highlighting the image's position.



Figure 20. The OSM Map Loaded by XnView.

A drawback of XnView (and most other image viewers) is that they don't support the *writing* of metadata. In fact, XnView may support Exif writing in the near future since it already supports the editing of IPTC metadata (http://en.wikipedia.org/wiki/IPTC_Information_Interchange_Model).

Where ExifTool really shines is for modifying Exif tags. From the command line, a `-exif:<tag name>=<value>` argument sets the value for a particular tag. The GPS tags are listed at <http://www.sno.phy.queensu.ca/~phil/exiftool/TagNames/GPS.html>, and the ExifTool FAQ has some examples of their use under the "What format do I use for writing GPS coordinates?" heading (<http://www.sno.phy.queensu.ca/~phil/exiftool/faq.html>).

For example, to add the latitude and longitude (53.314401 -2.255305) to the `library.jpg` file would require:

```
exiftool -exif:gpslatitude=53.314401 -exif:gpslatituderef=N
        -exif:gpslongitude=-2.255305 -exif:gpslongituderef=W
        -n library.jpg
```

The numbers are in decimal degrees, but degrees, minutes, and seconds can be employed instead. Compass directions must be specified with the `GPSPLatitudeRef` and `GPSPLongitudeRef` tags, *ignoring* negative latitude and longitude values. For instance, in the example the longitude is negative (-2.255305) but the `GPSPLongitudeRef` is still W (West). The "-n" argument allows short, non-descriptive tag values to be used.

The determination of the compass directions is easily automated in a batch script. stampGPS.bat uses two if-tests to generate the necessary values for GPSPLatitudeRef and GPSPLongitudeRef:

```
@echo off
IF [%1]==[] goto Error
IF [%2]==[] goto Error
IF [%3]==[] goto Error

rem get integer parts of latitude and longitude
set _lat=%1
set _long=%2
set /A _latInt=_lat
set /A _longInt=_long

if %_latInt% LSS 0 (
  set _latRef=-exif:gpslatituderef=S
) else (
  set _latRef=-exif:gpslatituderef=N
)

if %_longInt% LSS 0 (
  set _longRef=-exif:GPSLongitudeRef=W
) else (
  set _longRef=-exif:GPSLongitudeRef=E
)

echo Adding (%_lat% %_long%) to %3 with exiftool...

exiftool -exif:gpslatitude=%_lat% -exif:gpslongitude=%_long% ^
  %_latRef% %_longRef% ^
  -exif:GPSAltitude=0 -exif:GPSAltitudeRef=0 -n %3

echo Finished.
exit /b

:Error
echo Incorrect number of arguments
exit /b
```

The batch file's exiftool call includes GPSAltitude and GPSAltitudeRef tag settings, although they aren't mandatory.

A call to stampGPS.bat requires three arguments: the latitude, the longitude, and file name, as in the following example:

```
stampGPS 53.314401 -2.255305 library.jpg
```

The resulting GPS details for the file are shown in Figure 21.

```

> exiftool library.jpg
ExifTool Version Number      : 8.37
File Name                    : library.jpg
Directory                   :
File Size                    : 25 kB
File Modification Date/Time  : 2010:11:04 15:27:33+07:00
File Permissions             : rw-rw-rw-
File Type                    : JPEG
MIME Type                    : image/jpeg
JFIF Version                 : 1.01
Exif Byte Order              : Big-endian (Motorola, MM)
X Resolution                  : 300
Y Resolution                  : 300
Resolution Unit               : inches
Y Cb Cr Positioning          : Centered
GPS Version ID                : 2.3.0.0
GPS Latitude Ref              : North
GPS Longitude Ref             : West
GPS Altitude Ref              : Above Sea Level
Image Width                   : 330
Image Height                  : 239
Encoding Process              : Baseline DCT, Huffman coding
Bits Per Sample               : 8
Color Components              : 3
Y Cb Cr Sub Sampling         : YCbCr4:2:0 (2 2)
GPS Altitude                  : 0 m Above Sea Level
GPS Latitude                  : 53 deg 18' 51.84" N
GPS Longitude                  : 2 deg 15' 19.10" W
GPS Position                  : 53 deg 18' 51.84" N, 2 deg 15' 19
Image Size                    : 330x239

```

Figure 21. GPS Information in library.jpg

The modified GPS data appears towards the end of the list in Figure 21.

9.2. Using Apache Commons Sanselan

The Sanselan API (<http://commons.apache.org/sanselan/>) can read and write images in numerous formats (JPG, PNG, GIF, BMP, PSD, etc.), including various types of metadata (Exif, ICPT, and XMP).

My MDViewer.java example shows some typical Sanselan ways of listing Exif tags in a JPG file. The file's metadata is accessed, then the values for specified tag name constants are printed. Another approach is to iterate through all the tags.

The follow call looks at the metadata inside library.jpg:

```

java -cp "c:\sanselan\sanselan-0.97-incubator.jar;."
      MDViewer library.jpg

```

The output is shown in Figure 22.

```

Various EXIF tags:
XResolution: 300
Date Time: not found
Date Time Original: not found
Create Date: not found
ISO: not found
Shutter Speed Value: not found
Aperture Value: not found
Brightness Value: not found

GPS-related EXIF tags:
GPS Latitude Ref: 'N'
GPS Latitude: 53, 18, 129609/2500 <51.844>
GPS Longitude Ref: 'W'
GPS Longitude: 2, 15, 9549/500 <19.098>

Metadata Items:
XResolution: 300
YResolution: 300
Resolution Unit: 2
YCbCr Positioning: 1
GPSInfo: 90
Interop Index: 'N'
Interop Version: 53, 18, 129609/2500 <51.844>
Unknown Tag <0x3>: 'W'
Unknown Tag <0x4>: 2, 15, 9549/500 <19.098>
Unknown Tag <0x5>: 0
Unknown Tag <0x6>: 0

```

Figure 22. Metadata inside engSign.png.

The code for MDViewer.java:

```

public static void main(String[] args)
{
    if (args.length != 1) {
        System.out.println("Usage: run MDViewer <jpg file>");
        return;
    }

    JpegImageMetadata jpegMD = MetaDataUtils.getJpegMD(args[0]);
    if (jpegMD == null)
        return;

    System.out.println("\nVarious EXIF tags:");
    printTag(jpegMD, TiffConstants.TIFF_TAG_XRESOLUTION);
    printTag(jpegMD, TiffConstants.TIFF_TAG_DATE_TIME);
    printTag(jpegMD, TiffConstants.EXIF_TAG_DATE_TIME_ORIGINAL);
    printTag(jpegMD, TiffConstants.EXIF_TAG_CREATE_DATE);
    printTag(jpegMD, TiffConstants.EXIF_TAG_ISO);
    printTag(jpegMD, TiffConstants.EXIF_TAG_SHUTTER_SPEED_VALUE);
    printTag(jpegMD, TiffConstants.EXIF_TAG_APERTURE_VALUE);
    printTag(jpegMD, TiffConstants.EXIF_TAG_BRIGHTNESS_VALUE);

    System.out.println("\nGPS-related EXIF tags:");
    printTag(jpegMD, TiffConstants.GPS_TAG_GPS_LATITUDE_REF);
    printTag(jpegMD, TiffConstants.GPS_TAG_GPS_LATITUDE);
    printTag(jpegMD, TiffConstants.GPS_TAG_GPS_LONGITUDE_REF);
    printTag(jpegMD, TiffConstants.GPS_TAG_GPS_LONGITUDE);
    System.out.println();

    showItems(jpegMD);
} // end of main()

```


The JPG's metadata is returned as a `JpegImageMetadata` object by `MetaDataUtils.getJpegMD()`. The object is queried for particular tags by `printTag()`, and all the tags are listed by `showItems()`.

The Exif tag structure was first used in TIFF files before being taken up as a metadata format for JPGs, and the JPG version retains the TIFF-based directory structure and naming scheme. For that reason, Sanselan's Exif tag names are defined in its `TiffConstants` class.

`getJpegMD()` is a support function stored in the my `MetaDataUtils` class. It accesses the metadata part of the file, checking for errors, including whether the metadata is for a JPG:

```
// in the MetaDataUtils class
public static JpegImageMetadata getJpegMD(String fnm)
{
    System.out.println("Looking for JPEG metadata in " + fnm);
    IImageMetadata md = null;
    try {
        md = Sanselan.getMetadata( new File(fnm));
    }
    catch(Exception e)
    { System.out.println(e); }

    if (md == null) {
        System.out.println("No metadata found");
        return null;
    }

    if (!(md instanceof JpegImageMetadata)) {
        System.out.println("Metadata is not for JPEG");
        return null;
    }
    return (JpegImageMetadata) md;
} // end of getJpegMD()
```

A typical call to `printTag()` in `MDViewer.java` supplies the JPG metadata object and a tag name constant:

```
printTag(jpegMD, TiffConstants.EXIF_TAG_SHUTTER_SPEED_VALUE);
```

`printTag()` extracts the associated value from the metadata, and prints it:

```
private static void printTag(JpegImageMetadata jpegMD,
                             TagInfo tagInfo)
{
    TiffField field = jpegMD.findEXIFValue(tagInfo);
    if (field == null)
        System.out.println(tagInfo.name + ": not found");
    else
        System.out.println(tagInfo.name + ": " +
                           field.getValueDescription());
} // end of printTag()
```

`showItems()` takes a different approach to printing tags – it extracts all the tags from the metadata, and prints their values.

```
private static void showItems(JpegImageMetadata jpegMD)
{
    System.out.println("Metadata Items:");
    ArrayList items = jpegMD.getItems();
    for (int i = 0; i < items.size(); i++)
        System.out.println("  " + items.get(i));
} // end of showItems()
```

Reading Exif GPS Metadata

MDViewer can print out any tag, including GPS-related tags, but the Sanselan API has some features especially aimed at GPS data. The following code comes from my GPSTagTests.java example, and shows how to read *and write* Exif latitude and longitude details for a JPG. I'll explain the reading parts here, and writing in the next subsection.

In GPSTagTests.java , showGPS() prints GPS values:

```
private static void showGPS(String fnm)
// print the latitude and longitude info for the file
{
    double[] coord = MetaDataUtils.getGPSCoord(fnm);
    if (coord != null) {
        System.out.println("GPS data for " + fnm + ":");

        System.out.printf("  Latitude (Degrees North): %.6f\n",
                           coord[0]);

        int[] dms = GeoMath.DDtoDMS(coord[0]);
        System.out.println("    in DMS form: " + dms[0] + " deg " +
                           dms[1] + "' " +
                           dms[2] + "'");

        System.out.printf("  Longitude (Degrees East): %.6f\n",
                           coord[1]);

        dms = GeoMath.DDtoDMS(coord[1]);
        System.out.println("    in DMS form: " + dms[0] + " deg " +
                           dms[1] + "' " +
                           dms[2] + "'");

        System.out.println();
    }
    else
        System.out.println("No data found");
} // end of showGPS()
```

Its output, when applied to home.jpg:

```
GPS data for home.jpg:
  Latitude (Degrees North): 53.314401
    in DMS form: 53 deg 18' 51''
  Longitude (Degrees East): -2.255305
    in DMS form: -2 deg 15' 19''
```

showGPS() uses MetaDataUtils.getGPSCoord() to return the latitude and longitude values inside an array.

```
// in the MetaDataUtils class
public static double[] getGPSCoord(String fnm)
// return the [lat, long] GPS data from the file in an array
{
    TiffImageMetadata.GPSInfo gpsInfo = MetaDataUtils.getGPSInfo(fnm);
    if (gpsInfo == null)
        return null;

    try {
        double longitude = gpsInfo.getLongitudeAsDegreesEast();
        double latitude = gpsInfo.getLatitudeAsDegreesNorth();
        return new double[] { latitude, longitude };
    }
    catch(ImageReadException e) {
        System.out.println("Could not read latitude and
                           longitude from " + fnm);
    }
    return null;
} // end of getGPSCoord()
```

`MetaDataUtils.getGPSInfo()` returns a collection of GPS information in a `TiffImageMetadata.GPSInfo` object, and the longitude and latitude are extracted from it. `getGPSInfo()` is defined as:

```
// in the MetaDataUtils class
private static TiffImageMetadata.GPSInfo getGPSInfo(String fnm)
// get the GPS data from the file
{
    // access the file's metadata
    JpegImageMetadata jpegMD = MetaDataUtils.getJpegMD(fnm);
    if (jpegMD == null)
        return null;

    TiffImageMetadata exifMD = jpegMD.getExif();
    if (exifMD == null) {
        System.out.println("No EXIF metadata found");
        return null;
    }

    TiffImageMetadata.GPSInfo gpsInfo = null;
    try {
        gpsInfo = exifMD.getGPS();
    }
    catch(ImageReadException e)
    { System.out.println("Could not read GPS Information"); }

    return gpsInfo;
} // end of getGPSInfo()
```

It utilizes `JpegImageMetadata.getExif()` to access the Exif metadata (if it exists), and then `TiffImageMetadata.getGPS()` for its GPS-related data.

`showGPS()` obtains the latitude and longitude in decimal degree format (e.g. 53.314401), but an alternative, and popular, notation is degrees, minutes and seconds (e.g. 53° 18' 51"). The `GeoMath.DDtoDMS()` carries out the conversion:

```
// in the GeoMath class
```

```

public static int[] DDtoDMS(double decDegrees)
{
    // chop degrees at the decimal
    int degrees = (int) Math.floor(decDegrees);
    if (degrees < 0)
        degrees = degrees + 1;

    // get fraction after the decimal
    double frac = Math.abs(decDegrees - degrees);

    // convert this fraction to seconds (without minutes)
    double dfSec = frac * 3600;

    // get number of whole minutes in the fraction
    int minutes = (int) Math.floor(dfSec / 60);

    // put the remainder in seconds
    int seconds = (int) Math.floor(dfSec - minutes*60);

    // fix round-off errors
    if (seconds == 60) {
        minutes++;
        seconds = 0;
    }

    if (minutes == 60) {
        if (degrees < 0)
            degrees--;
        else // degrees => 0
            degrees++;
        minutes = 0;
    }

    return new int[] { degrees, minutes, seconds };
} // end of DDtoDMS()

```

There's also a `DMStoDD()` function that does the opposite translation. The functions borrow ideas from "Coordinate conversions made easy" by Sami Salkosuo (<http://www.ibm.com/developerworks/java/library/j-coordconvert/>) and code by John Tauxe at <http://www.neptuneandco.com/~jtauxe/bits/LatLonConvert.java>.

Writing Exif GPS Metadata

In `GPSTagTests.java`, the writing of GPS information into a JPG file is illustrated by the `writeGPS()` method. For example, a call to `writeGPS()` takes an input file and location values.:

```
writeGPS("library.jpg", 7.006835, 100.5020);
```

The hard work in `writeGPS()` is managed by `MetaDataUtils.putGPSInfo()` which writes the modified JPG into a new file:

```

private static void writeGPS(String fnm, double lat, double lon)
{
    System.out.printf("\nWriting (lat, lon): (%.6f, %.6f)\n",
                    lat, lon);
}

```

```

String outFnm = extendFilename(fnm, "NEW");
System.out.println("Saving changes to " + outFnm);
boolean hasSucceeded =
    MetaDataUtils.putGPSInfo(fnm, lat, lon, outFnm);
if (hasSucceeded) {
    System.out.println("\nGPS data saved\n");
    showGPS(outFnm);
}
else
    System.out.println("\nGPS data not saved\n");
} // end of writeGPS()

```

The following output from writeGPS() shows how library.jpg has its position changed to (7.006835, 100.502), but the changes are saved in libraryNEW.jpg:

```

Writing (lat, lon): (7.006835, 100.502000)
Saving changes to libraryNEW.jpg
Looking for JPEG metadata in library.jpg

```

```

GPS data saved

```

```

Looking for JPEG metadata in libraryNEW.jpg
GPS data for libraryNEW.jpg:
  Latitude (Degrees North): 7.006835
    in DMS form: 7 deg 0' 24''
  Longitude (Degrees East): 100.502000
    in DMS form: 100 deg 30' 7''

```

MetaDataUtils.putGPSInfo() takes an existing file, a new latitude and longitude, and writes the changes into a new file. At the heart of putGPSInfo() is a call to updateExifMetadataLossless() in Sanselan's ExifRewriter class, which holds many methods for Exif writing, updating, and removal.

There's quite a lot of preparatory work required before ExifRewriter.updateExifMetadataLossless() can be called: the Exif metadata for the existing file must be accessed, a stream opened into the new file, and a TiffOutputSet created for holding the new GPS data.

```

// in the MetaDataUtils class

public static boolean putGPSInfo(String fnm,
                                double lat, double lon, String outFnm)
// modify (lat,lon) GPS data in fnm, saving it in outFnm
{
    File inFile = null;
    try {
        inFile = new File(fnm);
    }
    catch(Exception e)
    { System.out.println(e);
      return false;
    }
}

// open output stream to a new file
BufferedOutputStream bout = null;
try {
    bout = new BufferedOutputStream(

```

```

        new FileOutputStream(outFnm));
    }
    catch (Exception e)
    { System.out.println(e);
      return false;
    }

    // create TiffOutputSet for holding new GPS data
    TiffOutputSet outSet = null;
    try {
        // access the input file's meta data (may be null)
        JpegImageMetadata jpegMD = MetaDataUtils.getJpegMD(fnm);
        if (jpegMD != null) { // copy Exif metadata
            TiffImageMetadata exif = jpegMD.getExif();
            if (exif != null)
                outSet = exif.getOutputSet();
        }
        // if file has no metadata, create empty TiffOutputSet
        if (outSet == null)
            outSet = new TiffOutputSet();

        outSet.setGPSInDegrees(lon, lat);
    }
    catch (Exception e)
    { System.out.println(e);
      return false;
    }

    // write fnm contents and modified metadata to new file
    try {
        new ExifRewriter().updateExifMetadataLossless(
            inFile, bout, outSet);

        bout.flush();
        bout.close();
        return true;
    }
    catch (Exception e)
    { System.out.println(e);}

    return false;
} // end of putGPSInfo()

```

putGPSInfo() takes some care over the creation of the TiffOutputSet; it may be initialized with the Exif data from the existing file, or may be created from nothing if the file doesn't have any metadata. TiffOutputSet.setGPSInDegrees() sets the latitude and longitude values.