Exploring the Location API for J2ME

(pre-print of article in Dr Dobbs Journal, January 2006, 53-8)

David Parsons
Massey University, Auckland, New Zealand
d.p.parsons@massey.ac.nz

1. Introduction

The Java Location API for J2ME™ is intended to run on small client devices such as mobile phones. It integrates generic positioning and orientation data with persistent storage of point of interest (POI) objects known as Landmarks. Although there are now a few phones and software tools, including models and emulators from Ericsson and Nokia, which implement this API, developers wishing to work with it have been constrained by a limited set of options for simulating location data. Both the current Nokia and Ericsson tools limit input to GPS data (e.g. from a log file). To provide a somewhat more interactive approach, this article provides an introduction to the API using a map based simulation environment designed to work easily with the Sun J2ME Wireless Toolkit.

Although there are some other Java location APIs in the public domain, these are mostly based on the Mobile Location Protocol published by the Location Interoperability Forum (LIF). This works on the assumption that the system is a device aware cellular network, and that location information is pulled in XML format from a server-hosted API. In contrast, the Location API for J2ME is a client side generic location interface that is intended to work with many different positioning methods. Generic interfaces enable us to implement systems that span multiple sources of location information at the same time. This will become increasingly important as the devices and channels for tracking locations increase, enabling us to aggregate and prioritise different information sets that relate to the same target, as well as being able to choose between multiple methods of determining the location of a single device. This provides a number of advantages including fail-over, indoor/outdoor transparency and a choice between the speed and accuracy trade-offs that could be made between GPS, cellular or other positioning mechanisms.

In this article we introduce the object model used in the Location API and look at some simple code examples that you can explore using the map based simulator, which can be freely downloaded. The locations used in the examples described here are based on a map of Massey University's Auckland campus in New Zealand.

2. The Location API Object Model

The Location API object model consists of eleven classes and two listener interfaces (LocationListener and ProximityListener), all in the javax.microedition.location package*. Their design approach uses several standard patterns, including Façade, Factory Method, Singleton and Value Object, and standard JavaBeans–style accessors. Of the eleven classes, two are Exception classes (LocationException and LandmarkException) and another four (AddressInfo, Criteria, Orientation, and QualifiedCoordinates) are primarily value objects. Many of the properties of these objects may in practice be unavailable, depending on the location finding technology that is implementing the API, but these properties anticipate likely future developments in mobile networks and devices and the level of location and context detail that they will be able to provide.

2.1 Location related classes

Location objects are immutable aggregates of AddressInfo and QualifiedCoordinates objects. Location instances are acquired from a LocationProvider, a façade to the mobile device's underlying location information that consists of a factory method (parameterised by a Criteria object) to retrieve a LocationProvider instance, methods to return current or last-known Location objects and methods to register listeners for location and proximity events. The Coordinates class (the superclass of QualifiedCoordinates) encapsulates geometric methods such as calculating the azimuth (angle) and distance between locations.

---

* *Because the J2ME Wireless Toolkit does not allow new classes to be created in system packages, this implementation uses the alternative package name massey.microedition.location*

The Orientation class is completely separate from the rest of the object model, having no association or dependency relationships with any other classes. This is presumably because not all devices will be able to support orientation information. At a minimum, the device must be able to provide a compass azimuth value to support Orientation objects, with optional support for pitch and roll values. Orientation objects can be derived either from a factory method or from a parameterised constructor.

The classes discussed so far encapsulate the subset of the API that is directly related to the acquisition of location information from whatever underlying technology is available to the device. These are summarised in Figure 1.

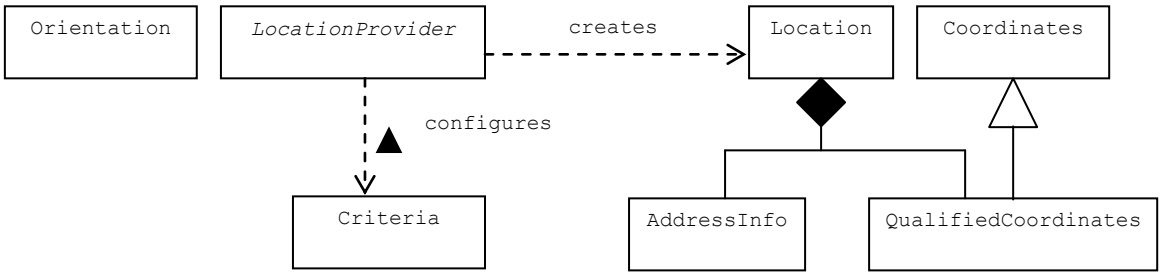


Figure 1: The Location related subset of the API

2.2 A simple location based program

Having introduced those classes that relate to location finding, we will look at a simple MIDlet that displays the current location of the device on the screen (the complete MIDlet is shown in listing 1). The interaction with the Location API in this example is very simple. First, we access the LocationProvider using the static factory method `getInstance`. The parameter is a Criteria object that can be used to set quality of service parameters, but in this example we will simply pass a null reference.

```
try
{
  locationProvider = LocationProvider.getInstance(null);
}
catch(LocationException e)
{
  // handle exception
}
```

The LocationProvider can now be used to get the current location. The parameter to the `getLocation` method is the timeout period.

```
try
 {
   location = locationProvider.getLocation(20);
 }
 catch(LocationException e)
 {
   // handle exception
 }
 catch(InterruptedException e)
 {
   // handle exception
 }
```

Once we have a Location object we can use it to find out our current latitude, longitude, altitude, direction and speed by accessing the QualifiedCoordinates object that is aggregated inside it.

```
coordinates = location.getQualifiedCoordinates();
```

A location may also contain an AddressInfo object, with details such as the location's postal address, phone number, country, URL etc. However depending on the implementation and context there may not actually be an

AddressInfo associated with a given location. For example, if you are standing in the middle of a field there will be no AddressInfo data. In the simulator implementation described here, the AddressInfo is always null.

Our current course, speed and altitude are returned from the QualifiedCoordinates object as float values, while latitude and longitude are returned as doubles. To assist the display of location data, the Coordinates class includes a method to convert a coordinate object into a String representation in either of two formats:

Format 1: Degrees, Minutes, and decimal fractions of a minute.
For example, for the double value of the coordinate 61.51d, the formatted string is 61:30:36

Format 2: Degrees, Minutes, Seconds and decimal fractions of a second.
For example, for the double value of the coordinate 61.51d, the formatted string is 61:30.6.

In the example MIDlet we use the second format, selected using the constant field `Coordinates.DD_MM_SS`. Alternatively, format 1 can be selected using `Coordinates.DD_MM`. Here is the fragment of the MIDlet where the location information is displayed on the screen, using a series of StringItems.

```
latitude = new StringItem("Latitude: " +
  Coordinates.convert(coordinates.getLatitude(), Coordinates.DD_MM_SS), "");
longitude = new StringItem("Longitude: " +
 Coordinates.convert(coordinates.getLongitude(), Coordinates.DD_MM_SS), "");
altitude = new StringItem("Altitude: " + coordinates.getAltitude(), "");
direction = new StringItem("Course: " + location.getCourse(), "");
speed = new StringItem("Speed: " + location.getSpeed(), "");
locationForm.append(latitude);
locationForm.append(longitude);
locationForm.append(altitude);
locationForm.append(direction);
locationForm.append(speed);
display.setCurrent(locationForm);
```

Figure 2 shows a screen dump from the J2ME Wireless Toolkit (version 2.2) with the current location details being displayed. Instructions for installing and running the simulation environment and example MIDlets are included in the code download.

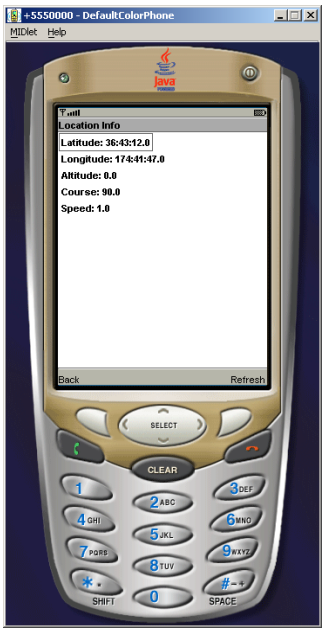


Figure 2: Location details displayed by the MIDlet

3. Using the Location and Proximity Listeners

The LocationProvider can register two types of listener; a single LocationListener and/or multiple ProximityListeners (Figure 3). These listeners help us to create more dynamic location based services that can be triggered by location related events.

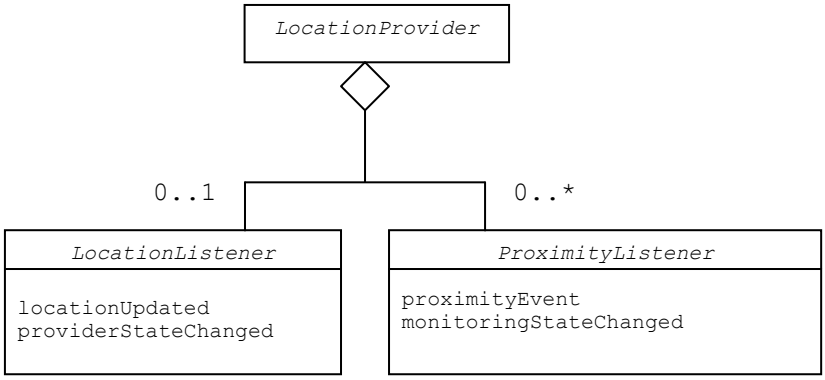


Figure 3. The listener interfaces

3.1 The LocationListener

In our previous example, we used the LocationProvider to find out the current location whenever the 'refresh' button was pressed on the phone. This is not a very flexible approach and it would be better if we could automatically trigger an event whenever the current location changed, without having to make an explicit call to `getLocation`. We can achieve this behaviour by using the LocationListener interface, which we can use to listen for updates to the current location. Only one LocationListener can be registered with the LocationProvider at any one time, so this listener can therefore act as a kind of Singleton for all components that need location information. The LocationListener is updated with current location information at specified intervals and it can expose and process that information in an application specific context to be accessed by other components on demand. In the next code example, to keep things simple, we make the MIDlet itself the LocationListener (listing 2). In order to listen to location events, our MIDlet has to implement the LocationListener interface:

```
public class LocationListenerMIDlet extends MIDlet
  implements CommandListener, LocationListener
```

Once the MIDlet implements the LocationListener interface, it can be registered with the LocationProvider as the current LocationListener via the `setLocationListener` method. The three parameters to this method are the listener, the interval between updates, the timeout period and the maximum acceptable age for location information:

```
try
{
  locationProvider = LocationProvider.getInstance(null);
  locationProvider.setLocationListener(this, 20, 10, 10);
}
catch(LocationException e)
{
  // handle exception
}
```

The listener interface contains two methods; `locationUpdated` and `providerStateChanged`. The LocationProvider triggers the `locationUpdated` method whenever the location changes. The `providerStateChanged` method is informed when the state of the LocationProvider changes between its three possible states, `LocationProvider.AVAILABLE`, `LocationProvider.TEMPORARILY_UNAVAILABLE` and `LocationProvider.OUT_OF_SERVICE`. In the example MIDlet only the `locationUpdated` method is implemented, simply calling the method that refreshes the screen form:

```
public void locationUpdated(LocationProvider provider, Location location)
{
       this.locationInfo();
```

```
    }
```

With these few changes to the MIDlet, we can now automatically update the location display whenever the location has changed, without having to press the 'refresh' button and do a manual update.

3.2 ProximityListeners

In the previous example we saw how a mobile application can use the LocationListener interface to listen for updates to the current location. In addition, we can implement the ProximityListener interface, which enables us to trigger events when we approach specified locations. Unlike the LocationListener, which can have only one registered instance, the LocationProvider allows us to register multiple ProximityListeners. Our third example MIDlet (listing 3) adds proximity monitoring to the existing location monitoring:

```
public class ProximityListenerMIDlet extends MIDlet
  implements CommandListener, LocationListener, ProximityListener
```

This time, when the MIDlet starts up, it registers itself three times so that we can listen for proximity events for three different locations (in this case three of the buildings on the map used in the simulator, the Study Centre, the Quadrangle and the Atrium). Each registration requires the listener, the coordinates and the required proximity radius from these coordinates as parameters.

```
try
{
  locationProvider = LocationProvider.getInstance(null);
  locationProvider.setLocationListener(this, 20, 10, 10);
// radius used to trigger proximity alerts
  float radius = 100.0F;
// coordinates of the Study Centre
  locationProvider.addProximityListener(this, studyCentreCoordinates, radius);
// coordinates of the Quadrangle building
  locationProvider.addProximityListener(this, quadrangleCoordinates, radius);
// coordinates of the Atrium
  locationProvider.addProximityListener(this, atriumCoordinates, radius);
}
```

The ProximityListener interface has two methods, `monitoringStateChanged` and `proximityEvent`. The `monitoringStateChanged` method simply informs listeners whether or not proximity notification is currently active. The `proximityEvent` method provides two parameters, the coordinates originally registered with the LocationProvider and the current location. In our implementation we compare the given coordinates with the ones we are listening for and select the matching location.

```
public void proximityEvent(Coordinates coordinates, Location location)
{
   if(coordinates.equals(studyCentreCoordinates))
   {
     showStudyCentreProximityAlert();
   }
   if(coordinates.equals(quadrangleCoordinates))
   {
     showQuadrangleProximityAlert();
   }
   if(coordinates.equals(atriumCoordinates))
   {
     showAtriumProximityAlert();
   }
}
```

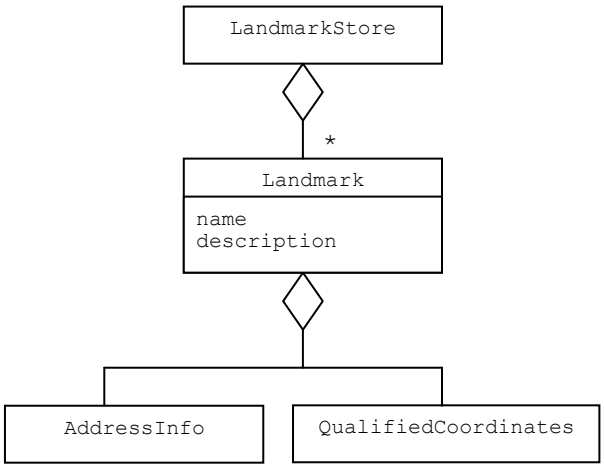
The various alerts simply display an image of the appropriate building. Figure 4 shows the proximity alert for the Study Centre being displayed on the mobile device. If you find you get excessive thrashing between the location and proximity listeners both trying to trigger the screen, you can remove the LocationListener code from this example.

Figure 4: An alert triggered by a ProximityListener

4. Landmarks and the LandmarkStore

What particularly marks out this API from other location based class libraries is the use of local storage to provide a persistent database of landmarks. This shifts the emphasis very much onto the mobile client in terms of location aware applications, enabling a local mapping from physical positions to points of interest. This approach means that at least part of a location aware application can be installed on the mobile device rather than on the server. One key advantage of this is that applications are more likely to have useful functionality even where network connectivity is unreliable.

The API includes two classes to support the persistent storage of location related data, namely the Landmark and the LandmarkStore (Figure 5). Landmark objects, like Location objects, are partial aggregates of AddressInfo and QualifiedCoordinates objects, but Landmarks and Locations have different roles in the architecture. Location objects are immutable and transitory, reflecting the dynamic movement of a mobile device. In contrast, Landmark objects are intended to be persisted in the mobile datastore and are mutable, so might be updated over time.



*

Figure 5. The classes related to persistent storage

The LandmarkStore acts as a facade to the underlying data store on the device, and is simply a collection of landmarks. There can be many LandmarkStores on a device, shared by multiple applications. Landmarks may optionally be stored under a category name, and may be added to multiple stores and multiple categories. The only restriction is that a Landmark cannot be added to the same category in the same LandmarkStore more than once. Be aware, however that the local data store will have restricted capacity, with storage limited to no more than a few hundred landmarks.

An important feature of the API is that a Landmark object can be populated from the coordinate and address data of a Location generated by the LocationProvider (provided of course that the LocationProvider is able to include an AddressInfo along with the QualifiedCoordinates.) This means that Landmarks can be dynamically added to the store.

In addition, Landmark objects already in the LandmarkStore can be linked with Location objects in terms of listener behaviour (Figure 6). A mobile application can register multiple ProximityListeners that can be triggered when the current location is within a specific range of a given Landmark. When a proximity event occurs the appropriate Landmark can be retrieved from the store.

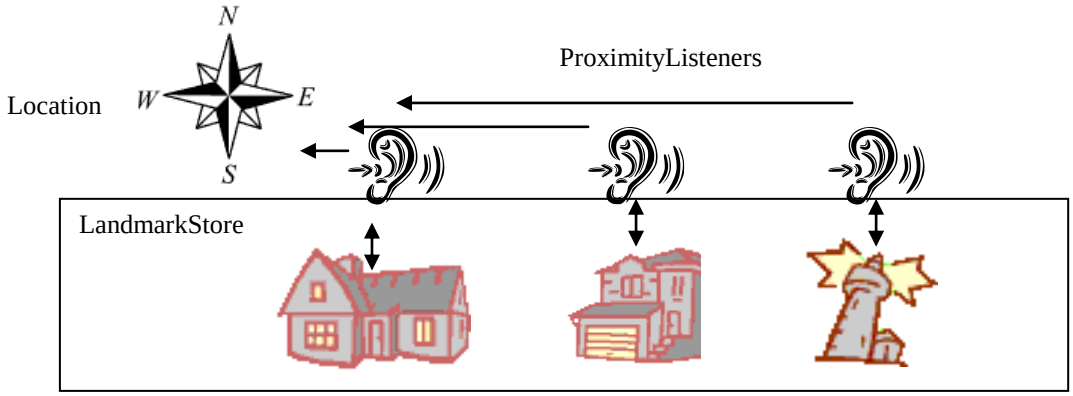


Figure 6: Using ProximityListeners with Landmarks in the LandmarkStore

There are two basic scenarios for an application that utilises the LandmarkStore. First, the store can be a small static collection of application specific Landmarks that rarely needs updating. ProximityListeners could be permanently registered to trigger proximity responses to the fixed landmarks. Systems that must deal with a larger set of Landmarks would require more dynamic provision of landmark information. In this context, the Landmark store would not contain preloaded objects but would add and remove them dynamically, using suitable push and/or pull mechanisms. A hybrid approach might also be used, with batch replacements of data based on movement between larger areas. ProximityListener registration and de-registration would also need to be dynamic, or alternatively, instead of using Proximity Listeners, the API supports searching the LandmarkStore for Landmarks that fall within a specified area.

To keep things simple, our final example (listing 4) assumes a pre-populated LandmarkStore that we use to display information about Landmarks when triggered by a proximity listener. In the previous example, we used a ProximityListener to display alerts when we came close to certain coordinates. In this MIDlet we display information from the LandmarkStore instead. The first step is to get an instance of the LandmarkStore (the null parameter indicates the default store).

```
landmarkStore = LandmarkStore.getInstance(null);
```

In our example we create the Landmark objects at start-up and write them to the LandmarkStore. The Landmark constructor requires a landmark name, a description, a set of coordinates and an AddressInfo:

```
Landmark landmark1 = new Landmark
  ("Study Centre", "Building on campus", studyCentreCoordinates, info);
```

Each landmark is added to the LandmarkStore using a category name. In this case the category is “campus”

```
landmarkStore.addLandmark(landmark1, "campus");
```

Later in the MIDlet we retrieve the landmark that we need using the `getLandmarks` method:

```
Enumeration e = landmarkStore.getLandmarks("campus", "Study Centre");
Landmark landmark = (Landmark)e.nextElement();
AddressInfo info = landmark.getAddressInfo();
```

Once the AddressInfo object is retrieved it can be displayed on the screen (Figure 7).

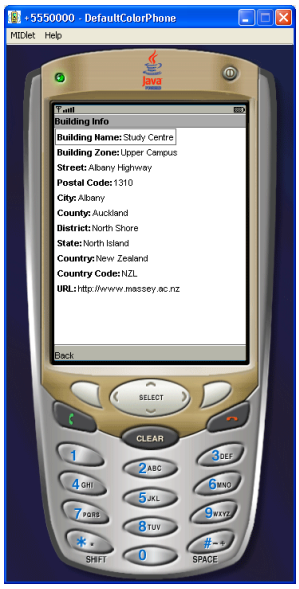


Figure 7: Displaying the AddressInfo from the Landmark

5. Using the Location API for J2ME simulator

To test the code in this article you will need to install and run the map based simulator. The simulator consists of three components:

1. A Swing application that lets a user control the direction and speed of a virtual mobile device moving across a map in a desktop environment. The virtual mobile device makes itself available to clients via the RMI registry
2. A simple web application that reads position data from the Swing application and publishes it to http clients via a JavaBean embedded in a JavaServer Page
3. An implementation of the Location API for J2ME that can be deployed into the Sun J2ME Wireless Toolkit as a jar file. This implementation acquires position data by connecting over http to the web application.

Although this three layer design may seem a little complex, it enables loose coupling between the Wireless Toolkit and the simulator. Since the LocationProvider implementation could only work with libraries available on the CLDC/MIDP platform (or we would not be able to also use the implementation on a mobile device) it is not possible to use RMI direct from the Wireless Toolkit. One advantage of this architecture is that both simulated and actual wireless devices can easily access the same data from the web server. Figure 8 summarises the various layers and components in the simulation system. Detailed instructions on how to configure and run the simulator can be found in the 'location readme' file in the downloadable archive.

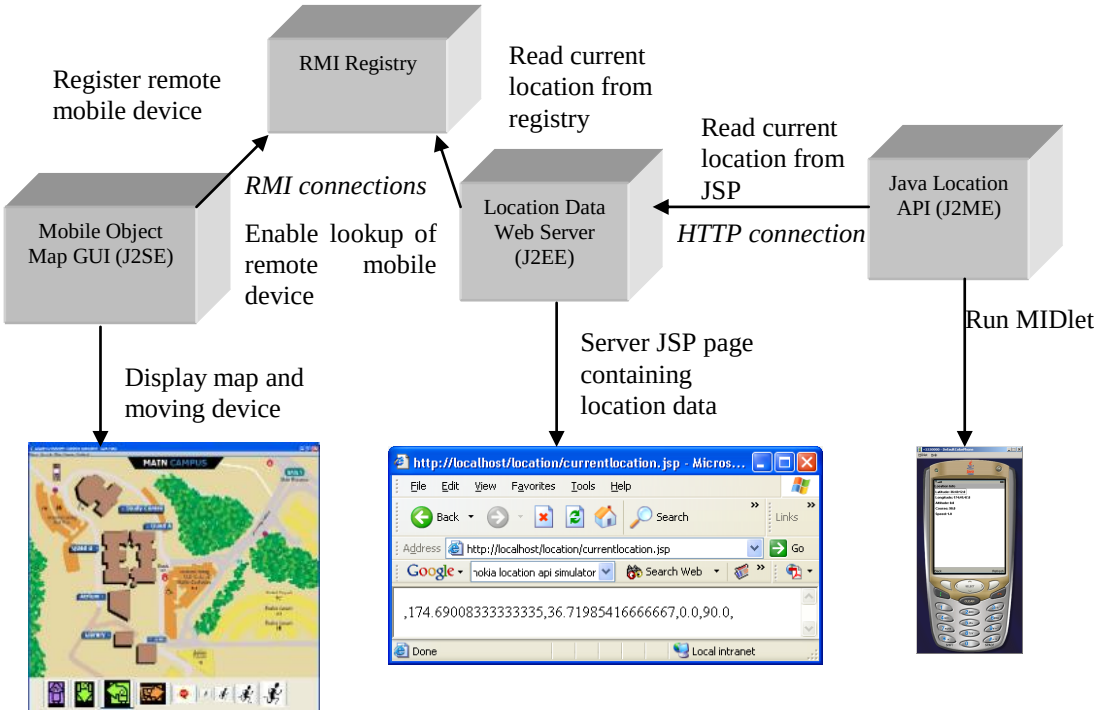


Figure 8. The layers and components of the simulation system

6. What's not in the API?

Before we conclude, it is worth briefly looking at aspects that are not included in this API. One feature that is present in some APIs, but not this one, is topological data, which is usually derived from the shapes of cell site coverage. Regardless of the underling technology, 'area of interest' data can be just as useful as point of interest data. The best we can do in this API is to define an area by its radius from a point but the facility to acquire topological data when available might be useful in some applications. There is also no support for location based queries, though these are fundamental to the use of server hosted spatial databases (such as Oracle's spatial database) that support LBS. Perhaps a client side query language could be a useful feature, enabling the client to build queries to be executed on the server. Other aspects of spatial databases such as yellow pages, routing and mapping are also key components of location based service but could not reasonably be expected to be hosted on a client device. However it might be useful if the API provided some facilities to interact with such systems. Perhaps in future versions of the specification we will see extensions to the API to enable more functionality on the mobile client.

References

Ericsson, *Mobile Positioning Protocol Specification Version 5.0*, Ericsson, 2003, http://www.ericsson.com/mobilityworld/developerszonedown/downloads/docs/mobile_positioning/mpp50_spec.pdf, last accessed June 3rd, 2005

Ericsson, *MPC Map Tool*. 2003, Ericsson. http://www.ericsson.com/mobilityworld/sub/open/technologies/mobile_positioning, last accessed June 3rd, 2005

Henson, N., 2004. Float11: Class for float-point calculations in J2ME applications CLDC 1.1 (Version 0.5) http://henson.newmail.ru/j2me/Float11.htm, last accessed June 3rd, 2005

LIF, *Mobile Location Protocol Specification Version 3.0.0*, Location Interoperability Forum, 2002, http://www.openmobilealliance.org/tech/affiliates/lif/lifindex.html, last accessed June 3rd, 2005

Loytana, K., *JSR-000179 Location API for J2ME™ (Final Release)*, Java Community Process, 2003, http://jcp.org/aboutJava/communityprocess/final/jsr179/index.html, last accessed June 3rd, 2005

McGovern, A., 2004. *Geographic Distance and Azimuth Calculations*. Retrieved 14th June, 2005, from http://www.codeguru.com/Cpp/Cpp/algorithms/general/article.php/c5115/ last accessed June 3rd, 2005

Nokia, *RI Binary for JSR-179 Location API for J2ME™*. 2004, Nokia. http://forum.nokia.com/info/sw.nokia.com/id/e7c4ed92-a6d7-4cbc-8a20-a31b17616ad9/jsr-179-ri-1_0a-bin(was1_1).zip.html, last accessed June 3rd, 2005

Oracle, *Introduction to Location APIs*, Oracle Corporation, 2004, http://www.oracle.com/technology/sample_code/products/iaswe/iASWE-LocationSample/doc/LocationAPI.html, last accessed June 3rd, 2005

Orange, *Orange UK Location API*. 2004. http://www.orangepartner.com/site/enuk/tools/orange_network_apis/orangelocationapi/p_orange_uk_location_api.jsp, last accessed June 3rd, 2005

Redknee, *Synaxis-2200™: ELS Release 2.0 Client Interface Specification Document*, Redknee Inc., 2002, http://www.sourceo2.com/O2_Developers/Tools/Location_API.htm, last accessed June 3rd, 2005

Sun, *J2ME Wireless Toolkit*. 2004, Sun Microsystems. http://java.sun.com/products/j2mewtoolkit/index.html, last accessed June 3rd, 2005