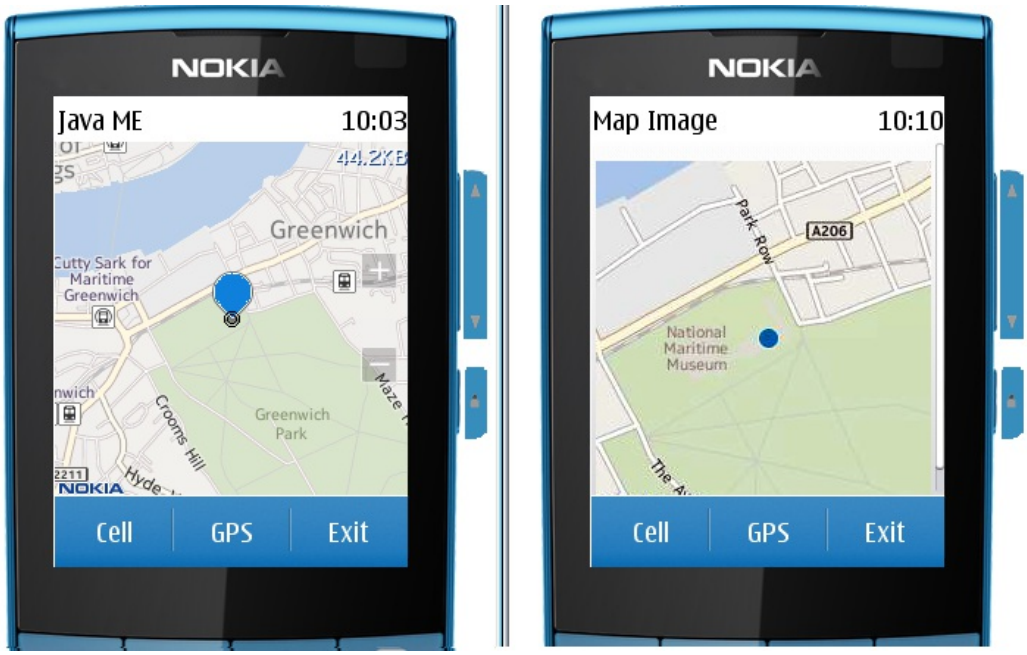
Using a JSR-179 Location Provider and displaying the result on a Map

There are several pre-existing articles on how to initialize a `LocationProvider` to find out the location of a device. This article extends the concept by displaying the location on a map using two HERE Mapping APIs the RESTful Map API and the HERE Maps API for Java ME



Picture showing Maps generated from the HERE Maps API for Java ME (left) and the RESTful Map API on an emulated device.

Introduction

The following articles give an overview on how to create a `LocationListener`, and retrieve the longitude and latitude locating a device.

- How to get Location Using Location API JSR 179
- Finding position in Java ME
- Determining Current Location via Cell ID
- Best practises for listening to location updates with Java ME

Most people do not think in terms of longitude and latitude, and would prefer to have their location displayed on a map. Fortunately Nokia provide access to their mapping services through a variety of public APIs. The article creates two examples combining the location listener with two of the HERE mapping APIs, and discusses the merits of each approach.

- A static map location using the RESTful Map API on Java ME
- A pannable map location using Map Markers with the HERE Maps API for Java ME

Location Provider Factory

It should be obvious that any application needing to discover the location of the device, will only require a single instance of a location provider. This can be ensured by the use of a factory pattern encapsulating a singleton. The class below offers two alternative methods to locate a device, either using GPS or CELL ID. The consuming application doesn't need to know the details, it merely needs to start and stop the location provider and offer the standard `LocationListener` interface to interact with the results.

Note Requests for location updates have been set up asynchronously at the **default** rate for the device, rather than querying directly with `provider.getLocation()`. This has been done for two reasons:

- It allows the app to respond to other events (such as map panning or zoom) whilst waiting for a response.
- It avoids setting up a thread to loop and continuously poll for locations and update a map; such a thread could easily overwhelm the rendering capabilities of a low level device.

```
public class LocationProviderFactory {
```

```
    public static final int CELL_ID = 0;
    public static final int GPS = 1;
    private static LocationProvider provider;
    private static String title;
    private static final LocationProviderFactory INSTANCE =
            new LocationProviderFactory();

    /**
     *  I'm a singleton.
     */
    private LocationProviderFactory() {
    }

    /**
     * Start locating at a default interval, timeout and maxAge;
     * @param listener - an listener who will receive location updates.
     */
    public static void startLocating(LocationListener listener) {
        if (provider != null) {
            provider.setLocationListener(listener, -1, -1, -1);
        }
    }

    /**
     * Stop sending location updates to the currently registered listener
     */
    public static void stopLocating() {
        if (provider != null) {
            provider.setLocationListener(null, -1, -1, -1);
        }
    }

    /**
     *
     * @return The name of the current provider.
     */
    public static String getTitle() {
        return title;
    }

    /**
     * Obtains a provider to be
     * @param type
     * @return The current location provider, or NULL if unable to set it up.
     * @throws LocationException
     */
    public static LocationProvider setProvider(int type) {

        try {
            if (type == CELL_ID) {
                provider = INSTANCE.getCellIdProviderInstance();
                title = "Obtaining Cell Id...";
            } else if (type == GPS) {
                provider = INSTANCE.getGPSProviderInstance();
                title = "Obtaining GPS...";
            }

        } catch (LocationException lex) {
             provider = null;
```

```
            title = "Provider not found...";
        } catch (NoClassDefFoundError ncdfex) {
            provider = null;
            title = "Provider not supported...";
        }

        return provider;
    }

    /**
     *
     * @return A Cell Id Location Provider
     * @throws LocationException if the provider could not be found.
     */
    private LocationProvider getCellIdProviderInstance() throws LocationException {
        int[] methods = {Location.MTA_ASSISTED | Location.MTE_CELLID |
Location.MTY_NETWORKBASED};
        return LocationUtil.getLocationProvider(methods, null);
    }

    /**
     *
     * @return A GPS Location Provider
     * @throws LocationException if the provider could not be found.
     */
    private LocationProvider getGPSProviderInstance() throws LocationException {
        Criteria criteria = new Criteria();

        criteria.setCostAllowed(true);
        criteria.setPreferredPowerConsumption(Criteria.NO_REQUIREMENT);
        criteria.setSpeedAndCourseRequired(false);
        criteria.setAltitudeRequired(false);
        criteria.setAddressInfoRequired(false);
        return LocationProvider.getInstance(criteria);
    }
}
```

Displaying A Map Image based on a Location

Warning: Typically the RESTful Map API,should only be used if you need a **single map image**, for any dynamic mapping use cases you will be better off using the HERE Maps API for Java ME at the cost of including an extra library jar file (approx 151 KB) as part of your app download. There is a significant reduction in required network traffic within the first couple of 200x200 pixel images due to the use of map tiling and caching. see here for more details

The form below consists of three command buttons to select a location provider and start locating. A static map can be retrieved by making an *http* request to the RESTful Map API. The details of how to do this can be found in the Using the RESTful Map API with Java ME article. Once an image is received, it is appended as an `ImageItem` to the `Form`

Note - the app id and token will need to be replaced with your own app id and token for the example to work correctly.

```
public class LocatorMapImageForm extends Form implements CommandListener,
LocationListener {

    // For the map
    private LocateMapImageMIDlet midlet;
    private QualifiedCoordinates lastKnownCoords;
    private static final float THRESHOLD_DISTANCE = 100f;
    // For the exit command, and the choice of location provider.
    private static final Command EXIT_COMMAND = new Command("Exit", Command.EXIT, 1);
```

```
    private static final Command LOCATE_BY_GPS_COMMAND = new Command("GPS",
Command.SCREEN, 2);
    private static final Command LOCATE_BY_CELL_ID_COMMAND = new Command("Cell",
Command.SCREEN, 3);
    private final MapImage mapImage;
    private int count = 0;

    /**
     * Set Up, initialise the Static Map and display it on the screen.
     * @param midlet
     */
    public LocatorMapImageForm(LocateMapImageMIDlet midlet) {
        super("Where Am I?");


        this.midlet = midlet;

        addCommand(EXIT_COMMAND);
        addCommand(LOCATE_BY_GPS_COMMAND);
        addCommand(LOCATE_BY_CELL_ID_COMMAND);


        lastKnownCoords = new QualifiedCoordinates(0d, 0d, 0f, 0f, 0f);

        // Ensure that we can request URLs from the RESTful Map URL.
        mapImage = new MapImage(this.getHeight(), this.getWidth(),
                // Start the application centered over the Royal Observatory in London.
                51.4813491d, -0.0031516d, 15);
        // You must get your own app_id and token by registering at
        // https://api.developer.nokia.com/ovi-api/ui/registration
        // Insert your own AppId and Token, as obtained from the above
        // URL into the two methods below.
        mapImage.setAppID("Your App ID goes here ...");
        mapImage.setToken("Your Token goes here ...");
        // Set language and image quality
        mapImage.setMapLabelLanguage(MapLanguage.ENGLISH);
        mapImage.setImageQuality(50);

        // Set up a default initial location.
        this.append(mapImage.getImage());

    }

    /**
     * Respond to Command Events, there are three.
     * <ul>
     *   <li>Locate by GPS</li>
     *   <li>Locate by Cell ID Provider</li>
     *   <ll>Exit App</li>
     * </ul>
     * @param command
     * @param displayable
     */
    public void commandAction(Command command, Displayable displayable) {
        if (displayable == this) {
            if (command == EXIT_COMMAND) {
                midlet.exitMIDlet();
            } else {
                LocationProviderFactory.stopLocating();
```



```
                if (command == LOCATE_BY_GPS_COMMAND) {
                    LocationProviderFactory.setProvider(LocationProviderFactory.GPS);

                } else if (command == LOCATE_BY_CELL_ID_COMMAND) {

LocationProviderFactory.setProvider(LocationProviderFactory.CELL_ID);

                }
                setTitle(LocationProviderFactory.getTitle());
                LocationProviderFactory.startLocating(this);
            }
        }
    }
```

Filtering Location Updated events

The most important method is the implementation of `locationUpdated()`, which is fired whenever a location request response is received. Since the app is requesting location updates at a default rate, these will be received **regardless** of whether the location has changed. Rendering a map will involve generating network traffic, so in order not to request the same or a very similar location each time, the previous location is checked, and a new map only requested if the location has changed.

Note that `providerStateChanged()` must also be implemented as part of `LocationListener`, but is not used.

```
    /**
     * Standard Event Listener for all JSR-179 implementations.
     * Just get hold of the location and <b>if it has changed</b>
     * significantly update the map.
     * @param provider
     * @param location
     */
    public void locationUpdated(LocationProvider provider, Location location) {

        // When the location is updated, request a Map
        QualifiedCoordinates coords = location.getQualifiedCoordinates();
        if (coords.distance(lastKnownCoords) > THRESHOLD_DISTANCE) {
            updateMap(coords);
            lastKnownCoords = new QualifiedCoordinates(
                    coords.getLatitude(), coords.getLongitude(), coords.getAltitude(),
                    coords.getHorizontalAccuracy(), coords.getVerticalAccuracy());
        } else {
            updateTitleOnly();
        }
    }


    /**
     * Standard Event Listener for all JSR-179 implementations.
     *
     * @param provider
     * @param newState
     */
    public void providerStateChanged(LocationProvider provider, int newState) {
        // We don't need to do anything here, but could use this to switch
        // to a backup provider if our current provicder is no longer
        // available.
    }
```


Visual Feedback when location is updated

In order to show that the location provider is working, some sort of visual feedback is required. In the case that the device has moved, this is simply updating the map. An additional method has been added to update the current title if a location request response has occurred and the map has not been updated. This may not be necessary in a real-life example.

```
    /**
     *    If the location has not moved outside of the Threshhold, do not update
     * the map. Just show that we've done something by altering the map title.
     */
    private void updateTitleOnly() {

        if (count < 5) {
            count++;
            setTitle("." + getTitle());
        } else {
            setTitle(getTitle().substring(5));
            count = 0;
        }
    }

    /**
     * Request a new Image and display Item on Form
     * @param coords
     */
    private synchronized void updateMap(QualifiedCoordinates coords) {
        mapImage.setLatitude(coords.getLatitude());
        mapImage.setLongitude(coords.getLongitude());
        this.deleteAll();
        this.append(mapImage.getImage());
        setTitle(coords.getLatitude() + " " + coords.getLongitude());
    }
```

Displaying a Map Location with the HERE Maps API for Java ME

The set up for this example is similar to the RESTful Map example, except that instead of using a `Form`, we use a `MapCanvas`. The `MapCanvas` class is part of the `maps-core.jar` binary. The HERE Maps API for Java ME has been integrated as a plug-in into the Asha SDK 1.0

The `MapCanvas` needs the same three `Command` buttons to select a location provider and start locating. A map will be displayed as soon as the `MapCanvas` is initialized. The start location and zoom level have been set programmatically, and a `StandardMapMarker` added to highlight the current location.

Note - the app id and token will need to be replaced with your own app id and token for the example to work correctly.

```
/**
 * A MapCanvas which displays a pannable map using HERE Maps API for Java ME, the map
displays the
 * location of the device as retrieved using JSR 179.
 */
public class LocatorMapCanvas extends MapCanvas implements CommandListener,
LocationListener {

    // For the map
    private LocateOnAMapMIDlet midlet;
    private QualifiedCoordinates lastKnownCoords;
    private static final float THRESHOLD_DISTANCE = 100f;
    // For the exit command, and the choice of location provider.
```

```
    private static final Command EXIT_COMMAND = new Command("Exit", Command.EXIT, 1);
    private static final Command LOCATE_BY_GPS_COMMAND = new Command("GPS",
Command.SCREEN, 2);
    private static final Command LOCATE_BY_CELL_ID_COMMAND = new Command("Cell",
Command.SCREEN, 3);
    private MapStandardMarker marker;

    private int count = 0;

    public LocatorMapCanvas(Display display, LocateOnAMapMIDlet midlet) {
        super(display);
        addCommand(EXIT_COMMAND);
        addCommand(LOCATE_BY_GPS_COMMAND);
        addCommand(LOCATE_BY_CELL_ID_COMMAND);

        this.midlet = midlet;
        lastKnownCoords = new QualifiedCoordinates(0d, 0d, 0f, 0f, 0f);

        map.setZoomLevel(15, 0, 0);

        // Start the application centered over the Royal Observatory in London.
        map.setCenter(new GeoCoordinate(51.4813491d, -0.0031516d, 0));

        marker = mapFactory.createStandardMarker(map.getCenter()l);
        map.addMapObject(marker);

    }

  public void onMapUpdateError(String description, Throwable detail, boolean critical) {
  }

  public void onMapContentComplete() {
  }
```

Filtering Location Updated events

The implementations of `locationUpdated()` and `providerStateChanged()` are **exact duplicates** of the previous example for the reasons given above, the code is not duplicated below. Ideally, a new map should not be requested unless the position has changed. The HERE Maps API for Java ME is more intelligent than the code using RESTful Map API web-service, since it is able to cache underlying map tile requests and will only request new Map tiles if necessary. However limiting the `MapCanvas` updates is still good practice to avoid the screen flickering on each location request response.

Visual Feedback when location is updated

The `updateTitleOnly()` method is an exact duplicate of the RESTful Map example and is purely to show some change each time the location is requested. The code has not been duplicated. The `updateMap()` method re-centers the `MapDisplay` and moves the marker. Either one of these operations will cause the `MapCanvas` to *repaint* automatically

```
    /**
     * Re-center the map and move the marker.
     * @param coords
     */
    private synchronized void updateMap(QualifiedCoordinates coords) {
        // When the location is updated, request a Map
        map.setCenter(new GeoCoordinate(coords.getLatitude(),
                coords.getLongitude(), coords.getAltitude()));

        marker.setCoordinate(new GeoCoordinate(coords.getLatitude(),
```

```
                coords.getLongitude(), coords.getAltitude()));

        setTitle(coords.getLatitude() + " " + coords.getLongitude());

    }
```

Which Mapping Service should be used?

Both examples work, and are able to display a map on the screen. The choice of which method to use is down to the developer, but the main consideration should be the end user.

The RESTful Map API example sends a **single** http request, and retrieves a **single** image in response. The size of the image received will depend on the image quality parameter, but at the standard 85% level of compression the image will be as follows:

RESTful Map API

This Map is 12.4Kb

The HERE Maps API for Java ME example sends **multiple** http requests each receiving a small map tile, and then stitches the final image out of the relevant parts of the responses. Each tile will be about 4Kb, and for the first map displayed, typically nine map tiles will be required .

HERE Maps API for Java ME


... plus ...

... plus ...

etc...

Each tile is about 4Kb, typically nine tiles = total 36 Kb

Therefore if a **single** static Map is required, using the RESTful Map API will probably involve less traffic. **However**, the HERE Maps API for Java ME is able to cache the map tiles received whereas the RESTful Map API implementation is unable to re-use the map images requested, so it must request new images **every** time. Hence, if a **series** of maps are required, the HERE Maps API for Java ME is much more economical in the medium term. This results in maps which are refreshed more quickly, and there is no traffic overhead.

When using a `LocationProvider` as in the example above, the Map is likely to change as the device moves, therefore, the preferred implementation would be to use the HERE Maps API for Java ME. A saving in network traffic should occur after **three** maps have been displayed.

Summary

The RESTful Map API may provide a simpler alternative to the HERE Maps API for Java ME for simple **static** mapping use cases and provides a smaller app size to download. However if the map needs to be **refreshed** at all, the tiling and caching capabilities of the HERE Maps API for Java ME result in less network traffic for the user, and this benefit swiftly outweighs the benefits of a smaller download to the end user.

In the case of **tracking** the location of a device, it should be self evident, that the HERE Maps API for Java ME approach should be preferred.

