FORUM NOKIA

MIDP: Location API Developer's Guide

Version 2.0; October 31st, 2006

Java™

Change history

October 31st, 2006	Version 2.0	This document is based on MIDP: Location API Developer's Guide, version 1.0, earlier published on Forum Nokia at http://www.forum.nokia.com/. The structure of the document has been harmonized and other minor updates have been made. In addition, the actual example Java code and some code comments have been added.

Disclaimer

The information in this document is provided "as is," with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. Furthermore, information provided in this document is preliminary, and may be changed substantially prior to final release. This document is provided for informational purposes only.

Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights.

Nokia Corporation retains the right to make changes to this specification at any time, without notice.

License

A license is hereby granted to download and print a copy of this specification for personal use only. No other license to any other intellectual property rights is granted herein.

Contents

1 About this document

This document is an introduction to the Location API for Java™ Platform, Micro Edition (Java ME™) (JSR-179 at http://www.jcp.org/aboutJava/communityprocess/final/jsr179/), which is an optional package that can be used with many Java ME profiles.

In addition, section Example: Tourist Route MIDlet on page 17, describes and illustrates the architecture of MIDP: Location API Example - Tourist Route at http://www.forum.nokia.com/info/sw.nokia.com/id/f7e8ad78-7898-4053-ab83-74c147923866/MIDP_Location_API_Example_Tourist_Route_v1_0.zip.html.

Note: The example application does not work in Series 40.

Intended audience

This document is intended for MIDP developers, as well as Java™ EE and Java SE developers wishing to use the S60 platform for developing mobile Java applications or services. The reader should be familiar with MIDP 2.0 before attempting to understand this example.

Scope

This document assumes a good knowledge of application and service development and the Java programming language. Furthermore, it assumes familiarity with enterprise application development. However, previous knowledge of developing for the mobile environment is not necessary.

This document focuses on the Location API and therefore explaining the Java technology is out of the scope of this documentation. For information on Java technology, see Java Technology web site at http://java.sun.com/ and Java Mobility Development Center at http://developers.sun.com/techtopics/mobility/. For additional information on MIDP tools and documentation, see Forum Nokia, Mobile Java section at http://www.forum.nokia.com/main/resources/technologies/java/.

2 Location API overview

The Location API for Java™ 2 Platform, Micro Edition (J2ME™) (JSR-179 at http://www.jcp.org/aboutJava/communityprocess/final/jsr179/), first introduced in S60 and Series 40 3rd Edition, is an optional package that can be used with many J2ME profiles. The minimum platform for JSR-179 use is Connected Limited Device Configuration (CLDC) 1.1, because the API requires floating point math support.

The MM API is supported in S60 and Series 40 with clarifications detailed in the Location API for J2ME™ (JSR-179): Implementation Notes.

The purpose of the Location API is to enable the development of location-based mobile applications. Considering the nature of mobile devices, the Location API provides a natural way to utilize location-based information. Moreover, the Location API is a compact package of classes and interfaces that are easy to use. The three main features that the Location API brings to mobile programming are:

- Obtaining information about the location of a device
- The possibility to create, edit, store, and retrieve landmarks
- The possibility to obtain the orientation of a device

The Location API needs a connection to a location-providing method, which generates the locations. Location-providing methods differ from each other in many ways. For example, the use of some methods may cost more than others, and the accuracies supported by individual location-providing methods vary. The most common methods are device-based (for example, GPS module - a method based on a Global Positioning System's satellites), network-based (for example, cell of origin - a method in which the network determines a user's place), or hybrid methods (for example, A-GPS - a GPS method which also uses network-based information to speed up location determination).

The figure below shows a generalized structure of a Location API MIDlet that uses a GPS module as a location-providing method. After the MIDlet is found to be working properly in the SDK environment, it should also be tested under real-world conditions. Typically, real-world testing means the outdoor use of the MIDlet in a device that supports the Location API.

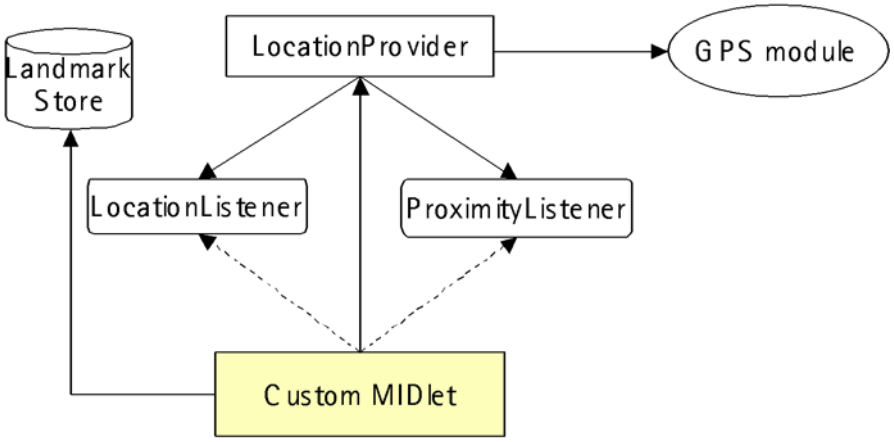

Figure 1: General Location API MIDlet model

2.1 Main use cases

Location-Based Services (LBS) are applications that utilize a user's current physical location to provide enhanced services. You can use location information for a variety of purposes. The use cases include:

- Mapping, navigation, and directions applications, which can also be combined with directory services
- Workforce-tracking and management applications
- Interactive gaming and services that complement sporting events, concerts, and more
- "Finder" applications can apply users' location to help them locate people or places nearby

- Weather applications can provide local forecasts and warnings about bad weather.

2.2 Geographic Coordinate System

A basic knowledge of geographic coordinates is required before starting to code with JSR-179. The Location API uses coordinates that are given using the World Geodetic System (WGS 84) datum at http://www.wgs84.com. It is also currently used as a reference system by the Global Positioning System (GPS).

The coordinates in the API are constructed from latitude, longitude, and altitude values. (In this case, altitude is the elevation above the sea level.) In the below Figure the horizontal lines measure latitude. These lines represent the north-south position between the poles. The North Pole is 90 degrees North (+90 degrees) and the South Pole is 90 degrees South (-90 degrees). The largest circle is called the *Equator* and is defined as 0 degrees. Locations above the Equator have positive latitudes (0 to +90 degrees) and locations below the Equator have negative ones (0 to -90 degrees).

However, the definition of the North Pole is not unambiguous. Location API implementations may use either the Magnetic North Pole or the True North Pole (also known as Geographical North Pole). In practice, any application using a compass should check how the API defines the North Pole.

Magnetic North Pole is the point to which magnetic compasses point. The position of the Magnetic North Pole is not static, since its position moves several kilometers a year. Poles are not at directly opposite positions of the globe, because they move independently of each other. The definition of the True North Pole defines latitude as +90 degrees. The distance from the North Pole to the Equator is approximately equal to the distance from the South Pole to the Equator.

The lines drawn from north to south in the below Figure are meridians, which are constant longitudinal values. There is no natural starting position for longitude, which is why a reference meridian had to be chosen. Traditionally the Prime Meridian is the meridian that passes through the Greenwich Observatory (located in Greenwich, England). The value of the Prime Meridian (Greenwich Meridian) is 0 degrees. WGS84, which the Location API uses, defines its zero meridian about 100 meters east of the traditional one. Locations east of the prime meridian have positive longitudinal values (0 to +180 degrees). Locations west of the prime meridian have negative longitudinal values (0 to -180 degrees).

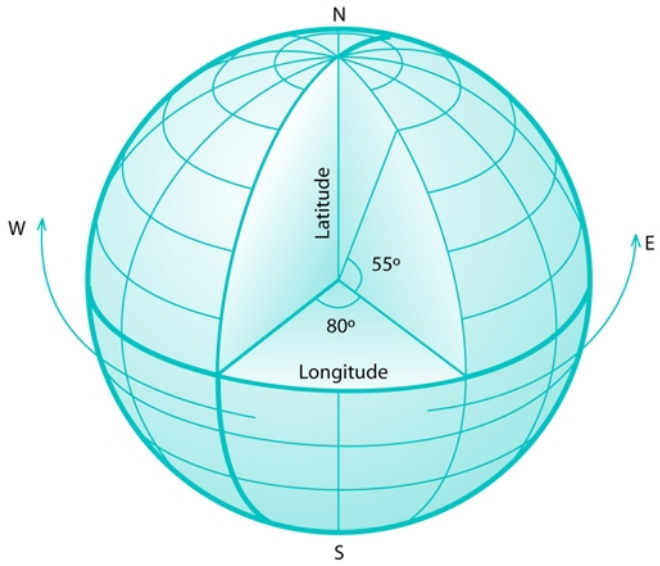

Figure 2: A globe showing longitudinal and latitudinal lines. The Zero Meridian and Equator are drawn in bold.

As seen in the figure below, the latitude lines become smaller and smaller near the poles. At the equator, one degree of longitude is roughly 111.3 km, whereas at 60 degrees of latitude one degree of longitude is only 55.8 km. That makes it difficult to see the lengths of latitudinal lines.

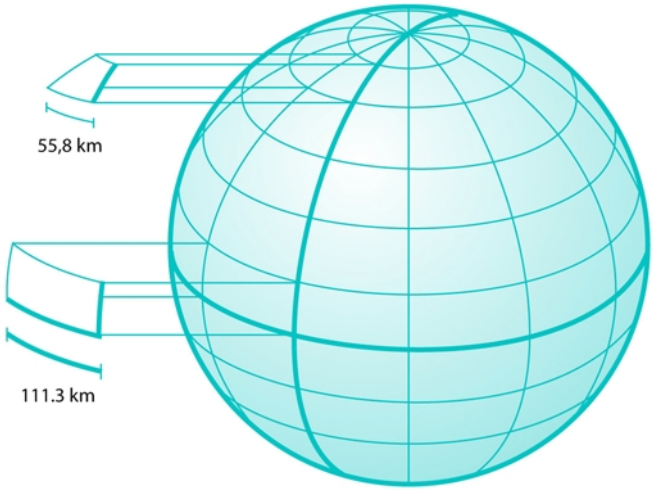

Figure 3: Length difference between latitude lines on the equator and at 60 degrees of latitude

2.3 Development Environment

Location API-based applications can be developed and tested in a simulated environment. For example, the S60 3rd Edition, FP 1 SDK can be used to simulate a route from a location to another in a Location API MIDlet. Application testing in the simulated environment requires location data. Test data can be created in a number of ways. One way is to use the Route tool that is provided, for example, with Series 60 3rd Edition, Feature Pack 1, for MIDP (available through `prefs.exe`). In the Route tool you can paint your route and save it on NMEA 0183 format for your MIDlet's usage. The NMEA (National Marine Electronics Association) 0183 format is a standard that is commonly used in GPS data transmission.

To change the coordinates of a route area, enter the new coordinates (either using the "N", "S" notation or by using negative and positive values) to the top, left, bottom, and right edit fields and click the **Load scale** button. The new coordinates are displayed above the drawing area. Loading a new scale deletes all existing coordinates in the route, since scaling them would easily cause confusion. When the route is finished you can save it by clicking the **Save** Button. Clicking the **Apply** button allows the SDKs to use the new route you have painted. The figure below shows an example screen shot of the Route tool in use.

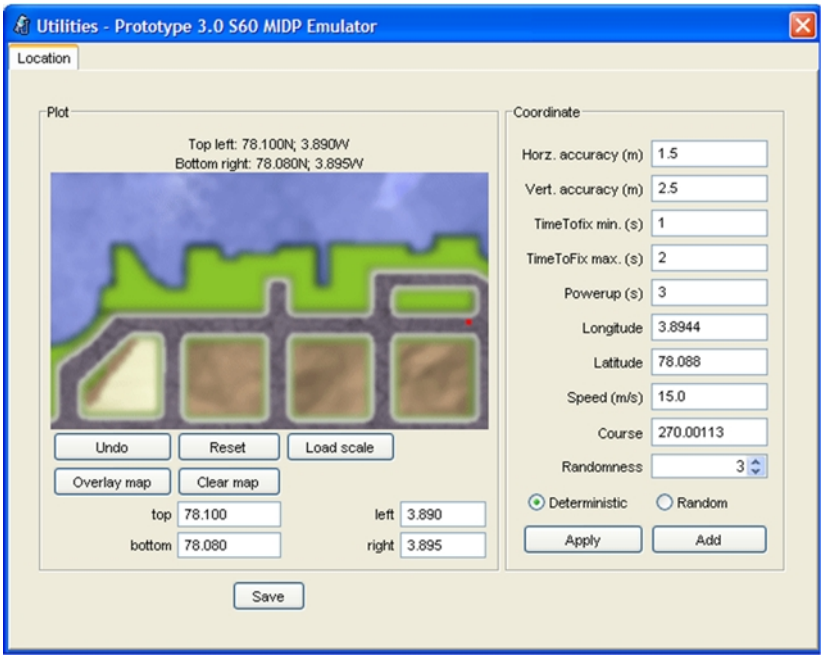

Figure 4: Location route tool

3 Using the Location API

The Location API is defined in the `javax.microedition.location` package. The API's presence can be discovered by using the system property `microedition.location.version`. The value of this system property is the version of the implemented API specification, for example, "1.0". This section of the developer's guide will demonstrate the usage of the Location API with code samples from an example MIDlet called Tourist Route available at Forum Nokia at http://www.forum.nokia.com/info/sw.nokia.com/id/f7e8ad78-7898-4053-ab83-74c147923866/MIDP_Location_API_Example_Tourist_Route_v1_0.zip.html and also included in this developer's guide. Each example code snippet shows only the essential part of the Location API usage. While reading this document it might also be useful to take a look at the TouristRoute MIDlet's source code in the IDE.

All the Location API classes are within the same package. This section introduces the API classes in small functionality groups, which should help to get the pieces together. The first operation in the API use is choosing the location provider; therefore, those classes are introduced first in the table below.

Table 3.1: Classes needed in choosing the location provider

Class	Short definition
`Criteria`	Used for the selection of the location provider.
`LocationProvider`	Represents a source of the location information (e.g., GPS module).

Usually an application that uses the Location API examines the device's position updates. The Location API uses listeners to examine these updates. The table below introduces these listeners.

Table 3.2: Listeners in the Location API

Class	Short definition
`LocationListener`	Listener that receives events associated with a particular `LocationProvider`.
`ProximityListener`	Listener to events associated with detecting proximity to registered coordinates.

The table below briefly introduces a set of classes for the measurement of a device's location.

Table 3.3: Classes needed in the device's location measurement

Class	Short definition
`AddressInfo`	Holds textual address information about a location.
`Coordinates`	Represents coordinates as latitude-longitude-altitude values.
`Location`	Represents the standard set of location information (time-stamped coordinates, speed, accuracy, course, etc.)
`LocationException`	Is thrown when a location API-specific error has occurred.
`QualifiedCoordinates`	Represents coordinates as latitude-longitude-altitude values that are associated with an accuracy value.

Location information can be stored for later use in the landmark store. The table below introduces the Location API classes for this purpose.

Table 3.4: Landmark and landmark store classes

Class	Short definition
`Landmark`	Represents a known location with a name.
`LandmarkException`	Is thrown when an error related to handling landmarks has occurred.
`LandmarkStore`	Methods for persistent landmark store management (store, delete, and retrieve landmarks).

If the device's sensors support the obtaining of orientation information, the information is available through the Location API's `Orientation` class (see the table below).

Table 3.5: Orientation information class in the Location API

Class	Short definition
`Orientation`	Represents the physical orientation of the device.

3.1 Device Location

Devices may not necessarily need a built-in location-providing method. For example, devices implementing the Location API may require an accessory that implements a location method. Such an accessory may be a GPS receiver with a Bluetooth connection or a GPS receiver in a functional cover. The accuracy of the location information depends on the location-providing method. Some devices implementing the Location API may obtain location information from the mobile phone network stations. Local short-range positioning beacons can also be used to obtain the location information. In addition, some devices may support a hybrid approach called assisted-GPS (A-GPS), in which GPS and the network are used to provide the location. The figure below demonstrates the positioning of a device by using the network's mobile stations and/or satellites.

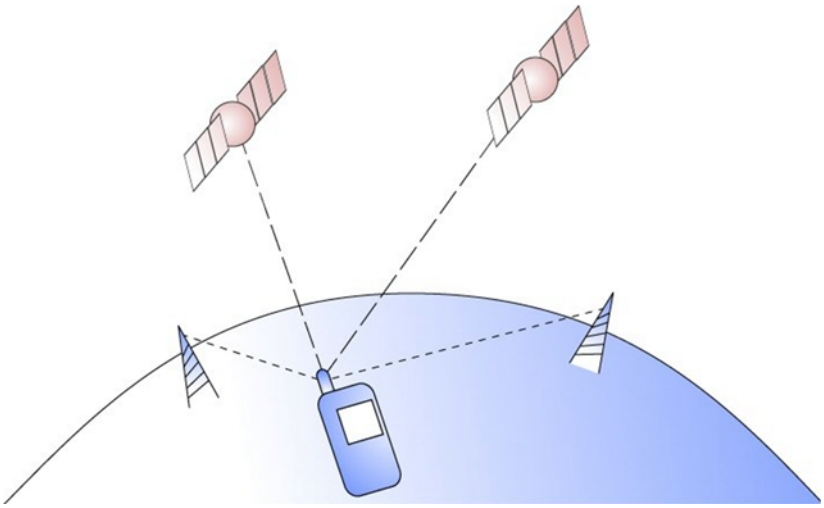

Figure 5: The positioning of a device by using mobile network stations and satellites

In general, location data from GPS receivers is more accurate than network-based data. In a downtown area GPS receivers may suffer from "canyon effects," where satellite visibility is intermittent. That may cause service breaks on MIDlets using the Location API. Depending on the location-providing method, the use

may or may not be free. The API includes the possibility to define certain criteria for which location-providing method should be used. The decision whether to pay for the service can be made using the criteria.

3.1.1 Choosing the location provider

The starting point of the API use is the `LocationProvider` class, which represents the location provider module. An instance of the `LocationProvider` class can be created by using the factory method `LocationProvider.getInstance(Criteria criteria)`. Criteria parameters can be used to define what kind of location provider is accepted. The Location API implementation chooses the location provider that best matches the defined criteria. The TouristRoute example defines a set of cost criteria that are used in a location provider search process.

Code sample 1: Creating criteria (TouristRoute's ConfigurationProvider.java class)

```
Criteria crit1 = new Criteria();
crit1.setHorizontalAccuracy(25); // 25m
crit1.setVerticalAccuracy(25); // 25m
crit1.setPreferredResponseTime(Criteria.NO_REQUIREMENT);
crit1.setPreferredPowerConsumption(Criteria.NO_REQUIREMENT);
crit1.setCostAllowed(false);
crit1.setSpeedAndCourseRequired(true);
crit1.setAltitudeRequired(true);
crit1.setAddressInfoRequired(true);
```

Code sample 2: Choosing the location provider (TouristRoute's ConfigurationProvider.java class)

A code snippet from the location provider search process can be seen in the Code sample 2 below. See the `ConfigurationProvider` class code in the TouristRoute MIDlet for a complete code that includes more logic.

```
try

    provider = LocationProvider.getInstance(criteria);
    if (provider != null)
    {
        // provider found!
        // .. see the complete source code for the real action.
    }

catch(LocationException le)

    // LocationProviders are currently out of service
```

If the device does not have any built-in location methods available or any external methods connected to it, the `LocationProvider.getInstance()` method throws `LocationException`, and the TouristRoute MIDlet gives a notification that says all providers are out of service. The location provider is found when the result from `LocationProvider.getInstance()` is not `null`. The TouristRoute MIDlet continues the search in a loop until one location-providing method is found.

3.1.2 Creating listeners

When `LocationProvider` has been successfully created, it can be used to register the MIDlet to listen for regular location updates and to detect certain coordinate proximity. Current location update and location provider state change events can be listened to through the `LocationListener` interface (see Code sample 3). Through this interface a JSR-179 application may track a device's route.

Code sample 3: LocationListener interface's skeleton (TouristRoute's TouristData.java class)

```
public void locationUpdated(LocationProvider provider, final Location location)

public void providerStateChanged(LocationProvider provider, final int newState)
```

Note: The standard `LocationListener` interface does not set the `location` or `newState` parameters as final. In the example, the `final` keyword is used on parameters to enable the anonymous thread implementation to access the reference. In the TouristRoute example, both implemented methods start a new thread, because `LocationListener` methods should return quickly without performing any extensive processing.

Through `ProximityListener`, the interface MIDlet obtains notifications when a registered coordinate is detected (see Code sample 4). Proximity event listening makes it possible to monitor when the device arrives in a certain area, for example close to a hotel or restaurant.

Code sample 4: Proximity listener's skeleton (TouristRoute's TouristData class)

```
public void proximityEvent(Coordinates coordinates, Location location)

public void monitoringStateChanged(boolean isMonitoringActive)
```

3.1.3 Registering listeners

LocationListener implementation can be registered to a particular location provider with LocationProvider's `setLocationProvider` method. With this it is possible to define the following parameters: the `interval` (in seconds) at which the application wants to receive events; the `timeout` value, which indicates how late (in seconds) an update is allowed to be in comparison to the defined interval; the `maxage` value, which defines how old the location update value is allowed to be. Code sample 5 demonstrates how the TouristRoute MIDlet registers the `LocationListener`

Code sample 5: Registering the location listener (TouristRoute's TouristData class)

```
if (provider != null)

  int interval = -1; // default interval of this provider
  int timeout = 0; // parameter has no effect.
  int maxage = 0; // parameter has no effect.

  provider.setLocationListener(this,interval,timeout,maxage);
```

Code sample 6 shows the registration of `ProximityListener` by using `LocationProvider`'s static `addProximityListener` method. With this method, for each listened coordinate one can also set a radius (in meters) that is a threshold value for event launching. In case the registration fails due to a lack of resources or support, `LocationException` is thrown. In the TouristRoute MIDlet, `ProximityListener` is registered to each landmark found from the `LandmarkStore`. Proximity listener registration method takes the `Coordinates` parameter, which can be obtained, for example, from the `Landmark` instance by using the `getQualifiedCoordinates()` method, same as in the TouristRoute example.

Code sample 6: Registering proximity listeners (TouristRoute's TouristData class)

```
try

    LocationProvider.addProximityListener(this,
                                          coordinates,
                                          PROXIMITY_RADIUS)

catch (LocationException e)

    // Platform does not have resources to add a new listener
    // and coordinates to be monitored or does not support
    // proximity monitoring at all
```

Note: All devices and emulators might not support proximity event sending through the `ProximityListener` interface. If the target device has no support, listeners have no effect.

3.2 Landmark Store

Landmarks can be stored for later use to a specific database called landmark store. All landmark stores are shared between all J2ME™ applications. Landmarks in the stores have a name identifier and may be placed into one or more categories. Grouping landmarks to categories makes it easier to find landmarks from the store, because especially in urban areas many different types of landmarks may be located close to each other. Devices contain one default landmark store. In addition, it may be allowed to create new landmark stores. The Location API enables applications to add new categories and remove existing ones within landmark stores.

3.2.1 Accessing the landmark store

All landmark store access methods are available through the `LandmarkStore` class. The first step in the landmark store use is to create an instance of `LandmarkStore` with the `getInstance(String storeName)` method. A landmark store instance will be returned if a named store can be found. If no landmark store with a specified name exists, the `getInstance` method returns `null`. The default landmark store can be obtained with a `null` parameter. Getting a landmark store instance is demonstrated in Code sample 7.

Code sample 7: Getting a landmark store instance (TouristRoute's ControlPoints.java class)

```
try

    store = LandmarkStore.getInstance(STORENAME);

catch(NullPointerException npe)


if (store == null)

   // code ...
```

A new landmark store can be created with the `createlandmarkStore(String StoreName)` method. Code sample 8 demonstrates the landmark store creation. Successfully created landmark stores are accessible through the `getInstance()` method. If landmark store creation failed due to an exception, a default landmark store can be accessed by using a `null` parameter in the `getInstance()` method.

Code sample 8: Creating a new landmark store (TouristRoute's ControlPoints.java class)

```
try

    LandmarkStore.createLandmarkStore(STORENAME);

catch(IllegalArgumentException iae)

    // Name is too long or landmark store with the specified name
    // already exists.

catch (IOException e)

    // Landmark store couldn't be created due to an I/O error

catch (LandmarkException e)

   // Implementation does not support creating new landmark stores.
```

3.2.2 Landmark store management

The `LandmarkStore` class contains a set of landmark management methods. `Enumeration` of `Landmark` objects can be obtained with one of the three `getLandmarks()` methods. The first method does not use parameters and provides a list of all stored landmarks in a specific landmark store. The other two methods can be used to fetch only the landmarks that match the set parameters. The second method can be seen in Code sample 9. In this example, one can obtain a list of landmarks from a specified category that are within a defined area and bounded by the min and max latitude and longitude positions. The third `getLandmarks()` method can be used to list landmarks within a specific category.

Code sample 9: Listing landmarks from a specific area and category

```
getLandmarks(String category, double minLatitude, double maxLatitude,
                              double minLongitude, double maxLongitude)
  throws java.io.IOException
```

New landmarks can be added to the landmark store with the `addLandmark(Landmark landmark, String category)` method. An added landmark will be associated with the specified category. A `null` category parameter indicates that a landmark does not belong to any category. If a landmark object passed as a parameter already belongs to the landmark store, the same landmark instance will be added to the specified category or categories it already belongs to. The sequence diagram in below figure shows how a new `Landmark` object is generated and added to the landmark store in the TouristRoute MIDlet.

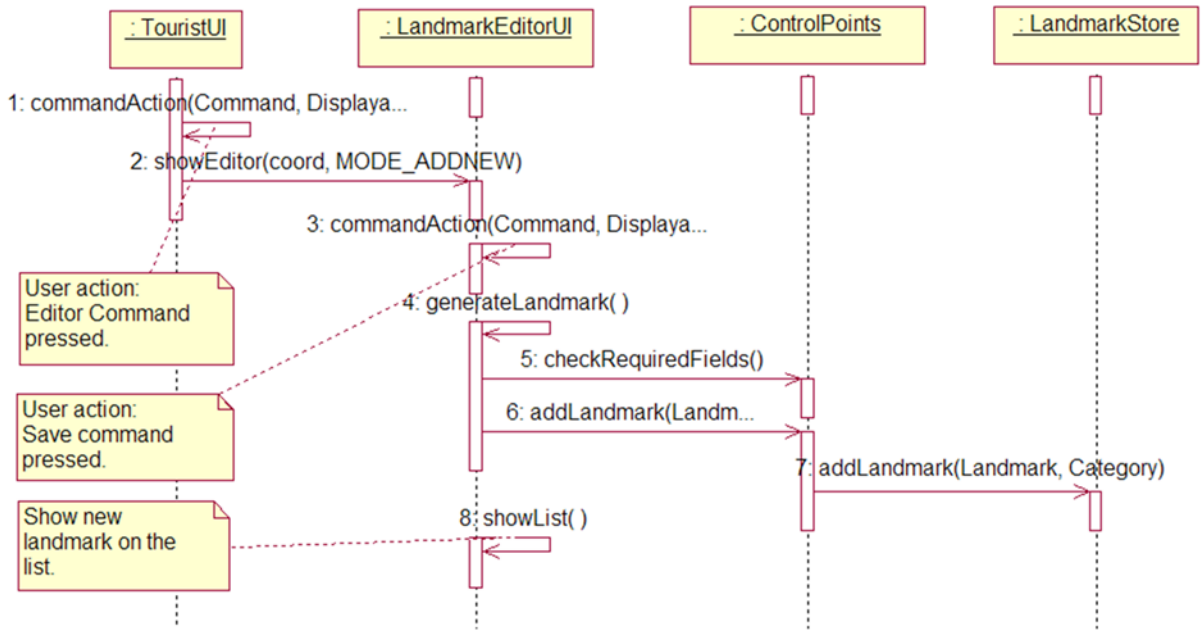

Figure 6: Sequence diagram of adding a new landmark in the TouristRoute example MIDlet

An existing landmark in the landmark store can be updated by using the `updateLandmark(Landmark landmark)` method. This method throws `LandmarkException` if the landmark passed as a parameter does not belong to the landmark store. In addition, landmark removal from the landmark store can be done with the `deleteLandmark(Landmark lm)` method. Also this method throws `LandmarkException` if the landmark passed as a parameter does not belong to the store. Furthermore, an existing `Landmark` object can also be removed only from the specific category with the `removeLandmarkFromCategory (Landmark lm, String category)` method.

Before using these three landmark store management methods, an instance of the `Landmark` object must exist. The constructor of the `Landmark` class requires the name, description, qualified coordinates, and address info parameters. `QualifiedCoordinates` for the landmark can be obtained, for example, from the `LocationListner.locationUpdated()` method's location parameter.

Code sample 10 shows how the TouristRoute example MIDlet creates a new `AddressInfo` instance and passes it to `Landmark`'s constructor. `AddressInfo` is a container class that holds textual address information about the location. It has only two methods: `getField(int field)` for getting field values, and `setField(int field, String value)` for setting the values. The `AddressInfo` class contains a set of constant `int` field values that should be used in the parameters passed to those two methods (see Code sample 10).

Code sample 10: Creating a landmark instance in the TouristRoute MIDlet (TouristRoute's LandmarkEditorUI.java class)

```
AddressInfo info = new AddressInfo();
info.setField(AddressInfo.COUNTRY, countryField.getString());
info.setField(AddressInfo.STATE, stateField.getString());
info.setField(AddressInfo.CITY, cityField.getString());
info.setField(AddressInfo.STREET, streetField.getString());
info.setField(AddressInfo.BUILDING_NAME, buildingNameField.getString());

Landmark lm = new Landmark(nameField.getString(),
                           descField.getString(),
                           coord,
                           info);
```

3.2.3 Category handling

The `LandmarkStore` class also includes category management functionality. `Enumeration` of landmark stores categories can be fetched with the `getCategories()` method. New categories can be added to an existing landmark store with the `addCategory` method and existing ones can be deleted with `deleteCategory`

Default categories may vary between device implementations. For example, a device may contain the following set of predefined categories: "Accommodation", "Business", "Communication", "Educational institute", "Entertainment", "Food & Beverage", "Geographical Area", "Outdoor activities", "People", "Public service", "Religious places", "Shopping", "Sightseeing", "Sports", "Transport".

3.3 Device Orientation

Some devices may include a hardware sensor that determines the orientation of the device. If the sensors are magnetic, the orientation is given as magnetic angles to the application. In practice, this means that if the application needs to use True North instead of Magnetic North (see section, "Geographic Coordinate System"), the application must perform the conversion task. The sensor type can be checked by using the `isOrientationMagnetic()` method of the `Orientation` class. Orientation information can be obtained from the API's `Orientation` class. Code sample 11 demonstrates how to obtain the orientation support information of a device.

Code sample 11: Obtaining orientation support information (TouristRoute's ConfigurationProvider.java class)

```
try

    orientation = Orientation.getOrientation();
    return true;

catch (LocationException e)

    return false;
```

If the access of `Orientation.getOrientation()` does not cause `LocationException`, the device supports at least the obtaining of compass azimuth.

3.3.1 Orientation sensor differences

If the device's sensor provides only a horizontal compass azimuth, the `getPitch()` and `getRoll()` methods of the `Orientation` class return `Float.NaN` as a sign that information is not available. If the device contains a 3D-orientation sensor, these two methods provide numeric information. The range of pitch values is [-90.0, 90.0] degrees. A negative value means that the top of the device screen is pointing towards the ground. The range of roll values is [-180.0, 180.0]. If roll values are negative, the device is orientated counter-clockwise from its default orientation.

4 Example: Location and orientation detection in mobile application development

The purpose of the MIDP: Location API Example - Tourist Route MIDlet, available for download at Forum Nokia at http://www.forum.nokia.com/info/sw.nokia.com/id/f7e8ad78-7898-4053-ab83-74c147923866/MIDP_Location_API_Example_Tourist_Route_v1_0.zip.html, and described in this tutorial is to demonstrate the main features of the Location API from a technical point of view. The MIDlet receives updated location information from a location provider (for example, GPS module). The information is displayed to users and they may attach address information to the position data on a specific editor. In practice, the MIDlet models a tourist's sight-seeing journey. However, the example MIDlet is not intended to be a real-world tourist guide. If you are not interested in the MIDlet's user interface, you might want to skip the next section and continue reading from section Implementation of the example on page 18.

Note: The example application does not work in Series 40.

4.1 MIDlet's User Interface

The example MIDlet covers API features that might not be supported in some devices. For example, devices not supporting orientation detection will have the compass feature disabled on the TouristRoute MIDlet.

The figure below demonstrates the flow of typical MIDlet use. On startup, the MIDlet tries to connect to a location provider. At this point, real devices might display a list of discovered Bluetooth devices. Emulators such as the Prototype SDK usually skip Bluetooth device listing, and will immediately start looking for location data. When a connection to a location provider is available, the TouristRoute MIDlet will switch to the *Route view* (middle screen shot in the top row in below figure) and begin to update location information.

By pressing the **Editor** command the user may switch to the Landmark Editor view. In this view, the user may edit address information for the current device location. By selecting the **Save** command, address information will be saved to the device's landmark store. The List command selection would switch the current view to the Landmarks list (left screen shot in the bottom row in below figure) without saving the landmark. The Landmarks list view contains all the saved landmarks. Also, the **Save** command selection on the landmark editor switches the current view to the Landmarks list view

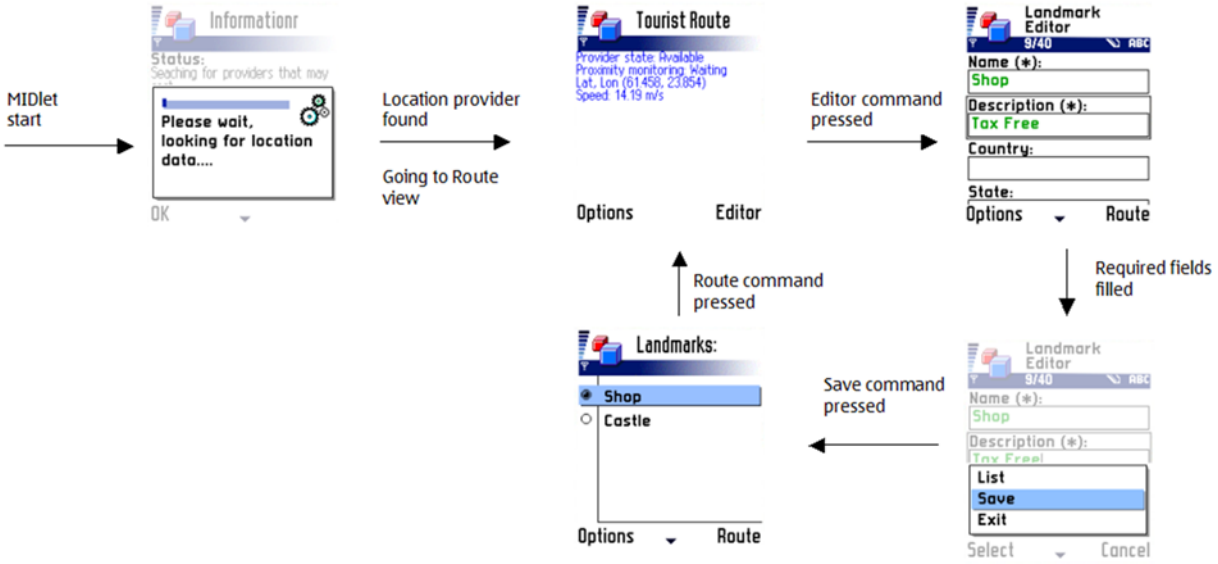

Figure 7: Screen flow from MIDlet start to landmark addition

Devices and emulators supporting orientation will have an extra **Compass** command available on the Route view. Pressing the **Compass** command switches the view to a Compass view, which shows compass information. Pressing the Pitch and Roll command on the Compass view allows switching to the Pitch and Roll view. This view displays device's pitch (tilt) and roll (rotation in degrees around device's own longitudinal axis). The Location API example in this guide provides orientation support.

From a code point of view, the use of the TouristRoute MIDlet begins from the `TouristUI` Canvas class, which displays the current position of the device. From `TouristUI`, the user may switch either to the `LandMarkEditorUI` or `CompassUI` class. However, switching to `CompassUI` is possible only if the device or emulator supports orientation. The `CompassUI` class enables switching to the `PitchRollUI` class. In addition, all the UI classes allow a possibility to switch back to the previous display. The `Displayable` class switches are shown in below figure.

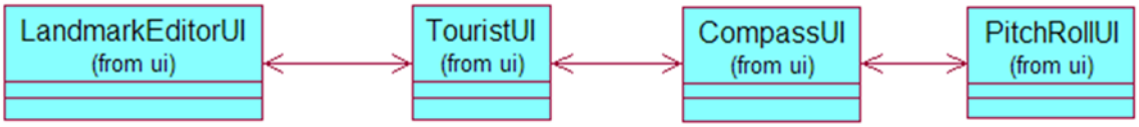

Figure 8: TouristRoute example MIDlet's UI view changes

In brief, the tasks of the UI classes are the following:

- `LandmarkEditorUI` lists stored landmarks. It also makes it possible to create new ones and remove existing ones.
- `TouristUI` displays information about the device's current position, for example, latitude and longitude coordinates, moving speed, and distance.
- `CompassUI` draws a compass. In addition, information on the compass type is displayed.
- `PitchRollUI` renders pitch and roll information of the device's orientation if the device has a 3D compass sensor supported in the API.

4.2 Prerequisites

You will need an S60 SDK that supports the packages used in this example.

Note: The example application does not work in Series 40.

4.3 Implementation of the Tourist Route MIDlet

Application Architecture

In the TouristRoute example MIDlet, all the UI classes are included in the `com.nokia.example.location.tourist.ui` package. Model classes can be found from the `com.nokia.example.location.tourist.model` package. In this example, only the classes in the model package use Location API functionality. When using the TouristRoute MIDlet to get acquainted with the Location API, the model layer classes might be the most interesting ones to take a look at. The UI layer classes only contain the MIDlet-specific UI implementation and might not help much in learning how to use the Location API. In addition, command controller logic is coded in the UI classes. The figure below shows TouristRoute example's classes and relationships.

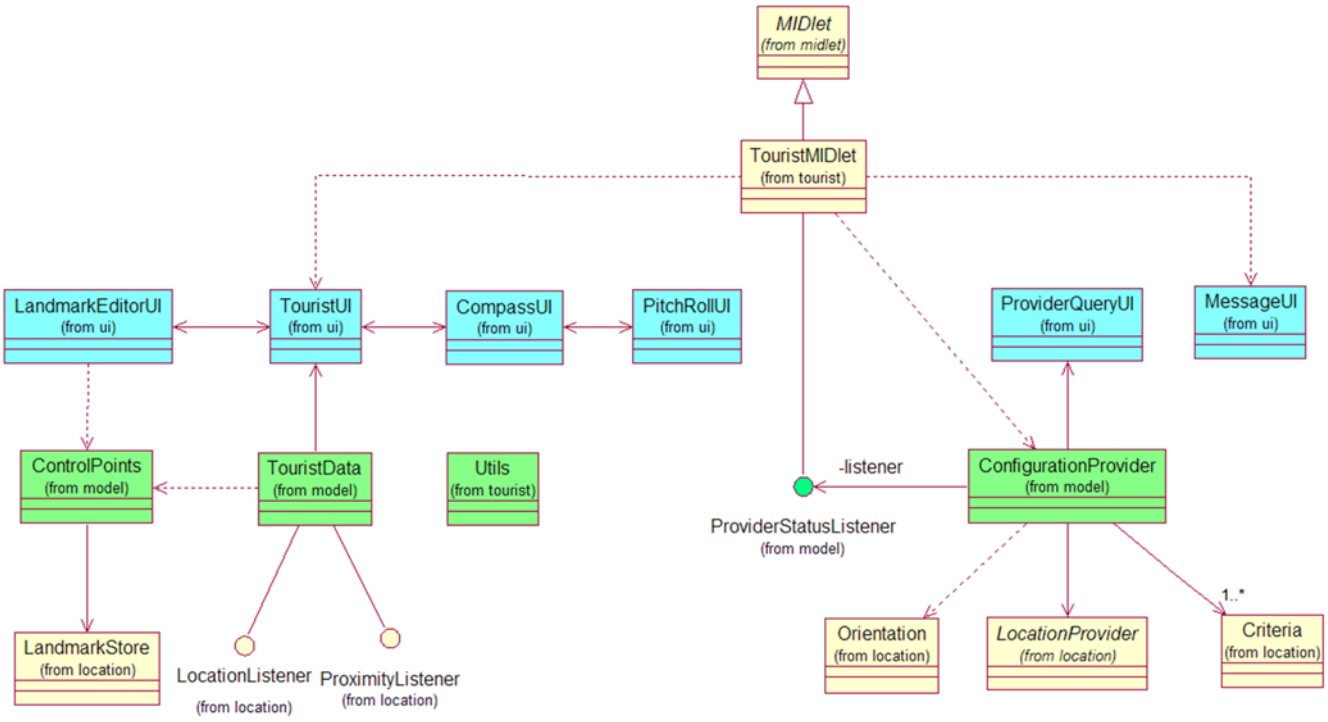

Figure 9: TouristRoute MIDlet's class diagram

The model layer classes and interfaces include the following tasks:

- `ControlPoints` handles all landmark store-related operations.
- `TouristData` listens for events through `LocationListener` and `ProximityListener` and handles data update on the viewer class `TouristUI`
- `ProviderStatusListener` is an interface that is used to deliver the location provider selection notification. In addition, this interface is used to deliver the information that `LocationProvider` has successfully started to receive location updates (for example, from a GPS module). The TouristRoute MIDlet listens for events through this interface.
- The `ConfigurationProvider` class handles `LocationProvider` selection in the application startup stage. After a location provider is selected, a notification event is sent back to MIDlet through the `ProviderSelectedListener` interface. In addition, orientation support checking and `Orientation` class access in the Location API belongs to this class.

Startup Sequence

TouristRoute example MIDlet startup performs an automatic location provider search. The MIDlet use can begin after the location provider is found and the MIDlet is capable of receiving a location update event. The purpose of this section is to clarify the TouristRoute example MIDlet's startup sequence.

The use of a network-based location provider may require the user to pay a fee. TouristRoute first tries to find a free location provider from a set of predefined location providers. If no free location providers are available, TouristRoute confirms with the end user whether (s)he wants to use a non-free service. The TouristRoute MIDlet uses predefined cost `Criteria` in searching for location providers.

The sequence diagram in below figure demonstrates the idea of how the TouristRoute MIDlet searches location providers. At first, the MIDlet looks for free location providers. If none are found, the MIDlet asks the user for permission to continue the search with non-free providers. If a cost-free location provider (for example, a GPS module over a Bluetooth connection) is found, the location provider search will end. When a location provider is found and selected, the example MIDlet receives `providerSelectedEvent()`.

At this point, the MIDlet registers to listen for events from the selected location provider. Especially when using a GPS module as a location provider, the first position calculation may take a while. That is why the first `locationUpdated()` event arrival on the `TouristData` class may take a while. When the first `locationUpdated()` event arrives, the `firstLocationUpdateEvent()` event is sent to the `TouristMIDlet` class. At this stage the application UI (`TouristUI`) is set to visible.

Note: Some location modules may take longer to start than others and the states of the module might change between module manufacturers. This means that you should take into account that the first location fix can take some time to retrieve due to the module startup time and satellite signal strengths.

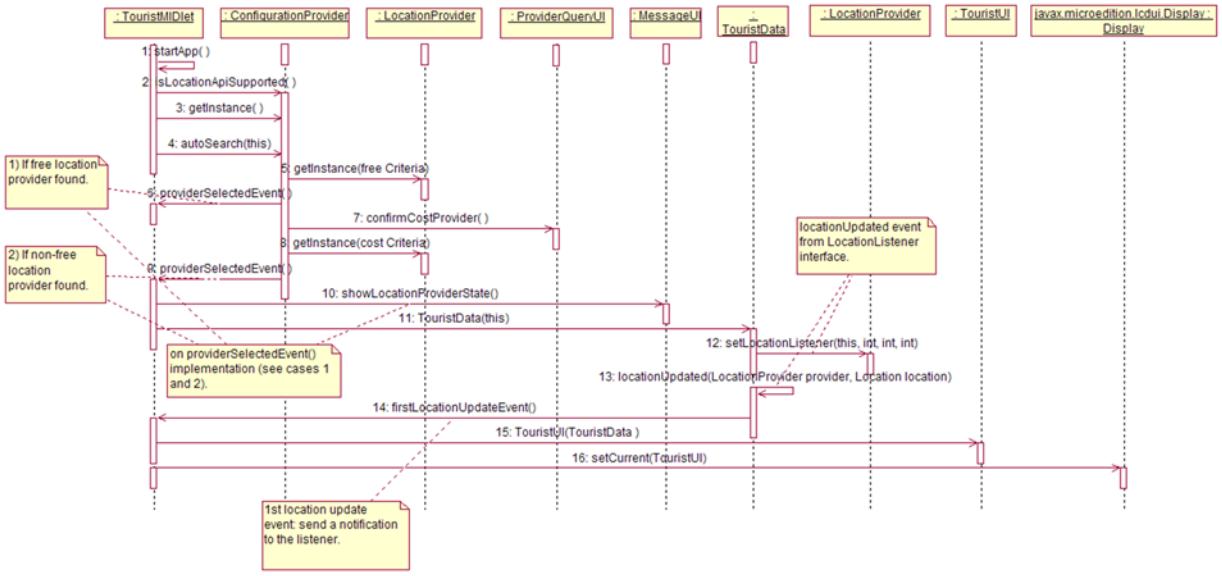

Figure 10: Sequence diagram of MIDlet startup and location provider search

4.4 Developing the Tourist Route MIDlet

In this section you create the example application using the Location API. Note that this section only presents one possible way of creating the required functionalities. You can use these examples as a basis for your own application or use the knowledge gained from this tutorial to create a different approach.

4.4.1 Creating the project environment

The project environment is constructed using the instructions for building MIDlets in IDEs or using command line, that come with the S60 or Series 40 SDKs.

4.4.2 Implementing the TouristMIDlet class

1 Create the `TouristMIDlet` class file.

2 Import the required classes.

```
package com.nokia.example.location.tourist;

import javax.microedition.lcdui.Display;
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;

import com.nokia.example.location.tourist.model.ConfigurationProvider;
import com.nokia.example.location.tourist.model.ProviderStatusListener;
import com.nokia.example.location.tourist.model.TouristData;
import com.nokia.example.location.tourist.ui.MessageUI;
import com.nokia.example.location.tourist.ui.TouristUI;
```

3 Set `TouristMIDlet` to extend MIDlet. Make and implement `ProviderStatusListener`. Create the constructor.

```
/**
 * Tourist Route MIDlet class.
 */
public class TouristMIDlet extends MIDlet implements ProviderStatusListener
{
    /** A static reference to Display object. */
    private static Display display = null;

    /** A Reference to TouristData. */
    private TouristData data = null;

    /** Lock object */
    private Object mutex = new Object();

    public TouristMIDlet()
    {
        super();
    }
```

4 Define the three required MIDlet methods for this application.

```
    protected void startApp() throws MIDletStateChangeException
    {
        display = Display.getDisplay(this);

        if (ConfigurationProvider.isLocationApiSupported())
        {
            ConfigurationProvider.getInstance().autoSearch(this);
        }
        else
        {
            MessageUI.showApiNotSupported();
        }
    }

    protected void pauseApp()
    {
    }

    protected void destroyApp(boolean unconditional)
            throws MIDletStateChangeException
    {
    }
```

5 Create a method for getting a reference to the Display object.

```
    /**
     * Getter method for Display reference.
     *
     * @return reference to Display object.
     */
    public static Display getDisplay()
    {
        return display;
    }
```

6 Create a method for indicating the selection of the location provider.

```
    /**
     * Event indicating location provider is selected. MIDlet use may therefore
     * begin.
     *
     * @see
com.nokia.example.location.tourist.model.ProviderSelectedListener#providerSelected
Event()
     */
    public void providerSelectedEvent()
    {
        // Attempt to acquire the mutex
```

```
            synchronized (mutex)
            {
                // Start scanning location updates. Also set the TouristData
                // reference data.
                MessageUI.showLocationProviderState();

                // Inform the user that MIDlet is looking for location data.
                data = new TouristData((ProviderStatusListener) this);
            }
        }
```

7 Create a method for indicating that the first location update has occurred.

```
    /**
     * Event indication about the first location update. This method sets
     * Tourist UI visible. By using mutex object, we ensure TouristaData (data)
     * is created on providerSelectedEvent.
     *
     * @see
com.nokia.example.location.tourist.model.ProviderStatusListener#firstLocationUpdat
eEvent()
     */
    public void firstLocationUpdateEvent()
    {
        // Attempt to acquire the mutex
        synchronized (mutex)
        {
            TouristUI ui = new TouristUI(data);

            data.setTouristUI(ui);
            display.setCurrent(ui);
        }
    }
}
```

4.4.3 Implementing the Utils class

1 Create the `Utils` class file.

2 Assign the class into the `com.nokia.example.location.tourist` package.

```
package com.nokia.example.location.tourist;
```

3 Create the `Utils` class. Write the code for creating a String presentation that is formatted for a specified amount of decimals. Don't use rounding rules.

```
/**
 * A container class for general purpose utility method(s).
 */
public class Utils
{
    /**
     * Creates a String presentation of double value that is formatted to have a
     * certain number of decimals. Formatted output value does not include any
     * rounding rules. Output value is just truncated on the place that is
     * defined by received decimals parameter.
     *
     * @param value -
     *            double value to be converted.
     * @param decimals -
     *            number of decimals in the String presentation.
     * @return a string representation of the argument.
     */
    public static String formatDouble(double value, int decimals)
    {
        String doubleStr = "" + value;
        int index = doubleStr.indexOf(".") != -1 ? doubleStr.indexOf(".")
                : doubleStr.indexOf(",");
        // Decimal point can not be found...
        if (index == -1) return doubleStr;
        // Truncate all decimals
        if (decimals == 0)
```

```
            {
                return doubleStr.substring(0, index);
            }

            int len = index + decimals + 1;
            if (len >= doubleStr.length()) len = doubleStr.length();

            double d = Double.parseDouble(doubleStr.substring(0, len));
            return String.valueOf(d);
        }
    }
```

4.4.4 Implementing the ConfigurationProvider class

1 Create the `ConfigurationProvider` class file.

2 Import the required classes.

```
package com.nokia.example.location.tourist.model;

import javax.microedition.location.Criteria;
import javax.microedition.location.LocationException;
import javax.microedition.location.LocationProvider;
import javax.microedition.location.Orientation;

import com.nokia.example.location.tourist.ui.ProviderQueryUI;
```

3 Create a class for handling location provider searches.

```
/**
 * Model class that handles location providers search.
 */
public class ConfigurationProvider
{
    private static ConfigurationProvider INSTANCE = null;

    /** Array of free Criterias. */
    private static Criteria[] freeCriterias = null;

    /** String array of free criteria names. */
    private static String[] freeCriteriaNames = null;

    /** Array of Criterias that may cause costs */
    private static Criteria[] costCriterias = null;

    /** String array of non-free criteria names. */
    private static String[] costCriteriaNames = null;

    /** Reference to ProviderQueryUI viewer class. */
    private ProviderQueryUI queryUI = null;

    /** Selected criteria */
    private Criteria criteria = null;

    /** Selected location provider */
    private LocationProvider provider = null;
```

4 Write a static block for constructing free criteria. The `Criteria` class is used for the selection of the location provider is defined by the values in this class.

`setHorizontalAccuracy` and `setVerticalAccuracy` set the desired horizontal and vertical accuracy preferences. `setPreferredResponseTime` sets the desired maximum response time preference. `setPreferredPowerConsumption` sets the preferred maximum level of power consumption. `setCostAllowed` sets the preferred cost setting. `setSpeedAndCourseRequired` sets whether the location provider should be able to determine speed and course. `setAltitudeRequired` sets whether the location provider should be able to determine altitude. `setAddressInfoRequired` sets whether the location provider should be able to determine textual address information.

For more information about this class and methods, see the Location API specification

```
    /*
     * Static block that constructs the content of free and non-free Criterias.
     * This code block is performed before the constructor.
     */
    static
    {
        // 1. Create pre-defined free criterias

        freeCriterias = new Criteria[2];
        freeCriteriaNames = new String[2];

        Criteria crit1 = new Criteria();
        crit1.setHorizontalAccuracy(25); // 25m
        crit1.setVerticalAccuracy(25); // 25m
        crit1.setPreferredResponseTime(Criteria.NO_REQUIREMENT);
        crit1.setPreferredPowerConsumption(Criteria.POWER_USAGE_HIGH);
        crit1.setCostAllowed(false);
        crit1.setSpeedAndCourseRequired(true);
        crit1.setAltitudeRequired(true);
        crit1.setAddressInfoRequired(true);

        freeCriterias[0] = crit1;
        freeCriteriaNames[0] = "High details, cost not allowed";

        Criteria crit2 = new Criteria();
        crit2.setHorizontalAccuracy(Criteria.NO_REQUIREMENT);
        crit2.setVerticalAccuracy(Criteria.NO_REQUIREMENT);
        crit2.setPreferredResponseTime(Criteria.NO_REQUIREMENT);
        crit2.setPreferredPowerConsumption(Criteria.POWER_USAGE_HIGH);
        crit2.setCostAllowed(false); // allowed to cost
        crit2.setSpeedAndCourseRequired(false);
        crit2.setAltitudeRequired(false);
        crit2.setAddressInfoRequired(false);

        freeCriterias[1] = crit2;
        freeCriteriaNames[1] = "Low details and power consumption, cost not
allowed";
```

Write a static block for constructing non-free criteria.

```
        // 2. Create pre-defined cost criterias

        costCriterias = new Criteria[3];
        costCriteriaNames = new String[3];

        Criteria crit3 = new Criteria();
        crit3.setHorizontalAccuracy(25); // 25m
        crit3.setVerticalAccuracy(25); // 25m
        crit3.setPreferredResponseTime(Criteria.NO_REQUIREMENT);
        crit3.setPreferredPowerConsumption(Criteria.NO_REQUIREMENT);
        crit3.setCostAllowed(true);
        crit3.setSpeedAndCourseRequired(true);
        crit3.setAltitudeRequired(true);
        crit3.setAddressInfoRequired(true);

        costCriterias[0] = crit3;
        costCriteriaNames[0] = "High details, cost allowed";

        Criteria crit4 = new Criteria();
        crit4.setHorizontalAccuracy(500); // 500m
        crit4.setVerticalAccuracy(Criteria.NO_REQUIREMENT);
        crit4.setPreferredResponseTime(Criteria.NO_REQUIREMENT);
        crit4.setPreferredPowerConsumption(Criteria.NO_REQUIREMENT);
        crit4.setCostAllowed(true);
        crit4.setSpeedAndCourseRequired(true);
        crit4.setAltitudeRequired(true);
        crit4.setAddressInfoRequired(false);

        costCriterias[1] = crit4;
        costCriteriaNames[1] = "Medium details, cost allowed";
```

```
        // Least restrictive criteria (with default values)
        Criteria crit5 = null;

        costCriterias[2] = crit5;
        costCriteriaNames[2] = "Least restrictive criteria";
    }
```

5 Create the constructor for this class.

```
    /**
     * Private constructor to force using getInstance() method.
     */
    private ConfigurationProvider()
    {
        queryUI = new ProviderQueryUI();
    }
```

6 Create a method for providing a single instance of the class.

```
    /**
     * Provide singleton instance of this class.
     *
     * @return static instance of this class.
     */
    public static ConfigurationProvider getInstance()
    {
        if (INSTANCE == null)
        {
            // Enable use of this class when Location API is supported.
            if (isLocationApiSupported())
            {
                INSTANCE = new ConfigurationProvider();
            }
            else
            {
                INSTANCE = null;
            }
        }

        return INSTANCE;
    }
```

7 Create a method for checking whether the Location API is supported.

```
    /**
     * Checks whether Location API is supported.
     *
     * @return a boolean indicating is Location API supported.
     */
    public static boolean isLocationApiSupported()
    {
        String version = System.getProperty("microedition.location.version");
        return (version != null && !version.equals("")) ? true : false;
    }

    public LocationProvider getSelectedProvider()
    {
        return provider;
    }
```

8 Create a method for searching the location provider with the use of the pre-defined criteria. The `getInstance` is a factory method used to get an actual `LocationProvider` implementation based on the defined criteria. `LocationException` is thrown when a location API specific error has occurred. The `getOrientation` method returns the terminal's current orientation.

For more information about this class and methods, see the Location API specification.

```
    /**
     * Search location provider by using pre-defined free and cost criterias.
     *
```

```
     * @param listener -
     *          a event listener that listens ProviderStatusLisneter events.
     */
    public void autoSearch(ProviderStatusListener listener)
    {
        try
        {
            for (int i = 0; i < freeCriterias.length; i++)
            {
                criteria = freeCriterias[i];

                provider = LocationProvider.getInstance(criteria);
                if (provider != null)
                {
                    // Location provider found, send a selection event.
                    listener.providerSelectedEvent();
                    return;
                }
            }

            if (queryUI.confirmCostProvider())
            {
                for (int i = 0; i < costCriterias.length; i++)
                {
                    criteria = costCriterias[i];

                    provider = LocationProvider.getInstance(criteria);
                    if (provider != null)
                    {
                        // Location provider found, send a selection event.
                        listener.providerSelectedEvent();
                        return;
                    }
                }
            }
            else
            {
                queryUI.showNoFreeServiceFound();
            }
        }
        catch (LocationException le)
        {
            queryUI.showOutOfService();
        }
    }

    public Orientation getOrientation()
    {
        try
        {
            return Orientation.getOrientation();
        }
        catch (LocationException e)
        {
            return null;
        }
    }
```

9 Create a method for determining whether orientation is supported.

```
    /**
     * Tells whether orientation is supported.
     *
     * @return a boolean indicating is orientation supported.
     */
    public boolean isOrientationSupported()
    {
        try
        {
            // Test whether Orientation instance can be obtained.
            Orientation.getOrientation();
            return true;
        }
```

```
            catch (LocationException e)
            {
                return false;
            }
        }
    }
```

4.4.5 Implementing ControlPoints class

1 Create the `ControlPoints` class file.

2 Import the required classes.

```
package com.nokia.example.location.tourist.model;

import java.io.IOException;
import java.util.Enumeration;

import javax.microedition.location.Coordinates;
import javax.microedition.location.Landmark;
import javax.microedition.location.LandmarkException;
import javax.microedition.location.LandmarkStore;
```

3 Create the `ControlPoints` class and define constants used by the class. `LandmarkStore` class provides methods to store, delete and retrieve landmarks from a persistent landmark store.

For more information about this class, see the Location API specification.

```
/**
 * Model class that handles landmark store actions.
 */
public class ControlPoints
{
    private LandmarkStore store = null;

    private static final String STORENAME = "TOURIST_DEMO";

    private static final String DEFAULT_CATEGORY = null;

    private static ControlPoints INSTANCE = null;
```

4 Create the constructor. The `createLandmarkStore` method creates a new landmark store with a specified name. The `LandmarkException` is thrown when an error related to handling landmarks has occurred.

For more information about this class and method, see the Location API specification.

```
    /**
     * Constructs instance of this class. Makes sure that landmark store
     * instance exists.
     */
    private ControlPoints()
    {
        String name = null;
        // Try to find landmark store called "TOURIST_DEMO".
        try
        {
            store = LandmarkStore.getInstance(STORENAME);
        }
        catch (NullPointerException npe)
        {
            // This should never occur.
        }

        // Check whether landmark store exists.
        if (store == null)
        {
            // Landmark store does not exist.

            try
            {
```

```
                // Try to create landmark store named "TOURIST_DEMO".
                LandmarkStore.createLandmarkStore(STORENAME);
                name = STORENAME;
            }
            catch (IllegalArgumentException iae)
            {
                // Name is too long or landmark store with the specified
                // name already exists. This Exception should not occur,
                // because we have earlier tryed to created instance of
                // this landmark store.
            }
            catch (IOException e)
            {
                // Landmark store couldn't be created due to an I/O error
            }
            catch (LandmarkException e)
            {
                // Implementation does not support creating new landmark
                // stores.
            }

            store = LandmarkStore.getInstance(name);
        }
    }
```

5 Create a singleton patterns `getInstance` method.

```
/**
 * Singleton patterns getInstance method. Makes sure that only one instance
 * of this class is alive at once.
 *
 * @return instance of this class.
 */
public static ControlPoints getInstance()
{
    if (INSTANCE == null)
    {
        INSTANCE = new ControlPoints();
    }

    return INSTANCE;
}
```

6 Create a method for finding a landmark from the landmark store by the use of an index. The `Landmark` class represents a landmark, i.e. a known location with a name.

```
/**
 * Find a Landmark from the landmark store using a index.
 *
 * @param index -
 *            the landmark in the store.
 * @return Landmark from landmark store.
 */
public Landmark findLandmark(int index)
{
    Landmark lm = null;

    Enumeration cps = ControlPoints.getInstance().getLandMarks();
    int counter = 0;

    while (cps.hasMoreElements())
    {
        lm = (Landmark) cps.nextElement();

        if (counter == index)
        {
            break;
        }
        counter++;
    }
```

```
        return lm;
    }
```

7 `getLandmarks` method lists all landmarks stored in the store.

The `getLatitude` and `getLongtitude` return the latitude/longtitude component of this coordinate. The `getQualifiedCoordinates` gets the `QualifiedCoordinates` of the landmark.

For more information about these methods, see the Location API specification.

```
/**
 * Find nearest landmark to coordinate parameter from the landmarkstore.
 */
public Landmark findNearestLandMark(Coordinates coord)
{
    Landmark landmark = null;

    double latRadius = 0.1;
    double lonRadius = 0.1;

    float dist = Float.MAX_VALUE;

    try
    {
        // Generate enumeration of Landmarks that are close to coord.
        Enumeration enu = store.getLandmarks(null, coord.getLatitude()
                - latRadius, coord.getLatitude() + latRadius, coord
                .getLongitude() - lonRadius, coord.getLongitude()
                + lonRadius);
        float tmpDist;
        Landmark tmpLandmark = null;

        while (enu.hasMoreElements())
        {
            tmpLandmark = (Landmark) enu.nextElement();
            tmpDist = tmpLandmark.getQualifiedCoordinates().distance(coord);

            if (tmpDist < dist)
            {
                landmark = tmpLandmark;
            }
        }
    }
    catch (IOException ioe)
    {
        // I/O error happened when accessing the landmark store.
        return null;
    }

    return landmark;
}

public Enumeration getLandMarks()
{
    Enumeration enu = null;

    try
    {
        enu = store.getLandmarks();
    }
    catch (IOException ioe)
    {
        // I/O error happened when accessing the landmark store.
    }

    return enu;
}
```

8 Create the various adding, updating and removing methods used by the class.

```
    public void addLandmark(Landmark landmark) throws IOException
    {
        store.addLandmark(landmark, DEFAULT_CATEGORY);
    }

    public void updateLandmark(Landmark landmark) throws IOException,
            LandmarkException
    {
        store.updateLandmark(landmark);
    }

    public void removeLandmark(Landmark landmark) throws IOException,
            LandmarkException
    {
        store.deleteLandmark(landmark);
    }

}
```

4.4.6 Implementing the ProviderStatusListener class

1. Create the `ProviderStatusListener` class file.
2. Include the class to the `com.nokia.example.location.tourist.model` package.

```
package com.nokia.example.location.tourist.model;
```

3. Create the class and add the two notification events described below to it.

```
/**
 * Listener interface for location providers status information.
 */
public interface ProviderStatusListener
{
    /**
     * A Notification event that location provider has been selected.
     */
    public void providerSelectedEvent();

    /**
     * A Notification event about the first location update.
     */
    public void firstLocationUpdateEvent();
}
```

4.4.7 Implementing the TouristData class

1. Create the `TouristData` class file.
2. Import the required classes.

```
package com.nokia.example.location.tourist.model;

import java.util.Enumeration;

import javax.microedition.location.AddressInfo;
import javax.microedition.location.Coordinates;
import javax.microedition.location.Landmark;
import javax.microedition.location.Location;
import javax.microedition.location.LocationException;
import javax.microedition.location.LocationListener;
import javax.microedition.location.LocationProvider;
import javax.microedition.location.ProximityListener;
import javax.microedition.location.QualifiedCoordinates;

import com.nokia.example.location.tourist.ui.TouristUI;
```

3 Create the `TouristData` class and set it to implement `LocationListener` and `ProximityListener`. The `LocationListener` represents a listener that receives events associated with a particular `LocationProvider`. The `ProximityListener` is an interface representing a listener to events associated with detecting proximity to some registered coordinates. The `Coordinates` class represents coordinates as latitude-longitude-altitude values.

For more information about these classes, see the Location API specification.

```
/**
 * Model class that handles LocationListeners and ProximityListeners events.
 */
public class TouristData implements LocationListener, ProximityListener
{
    /** A Reference to Tourist UI Canvas */
    private TouristUI touristUI = null;

    /** Coordinate detection threshold radius in meters */
    public static final float PROXIMITY_RADIUS = 100.0f;

    /** Previous coordinates outside the distance threshold area (20m) */
    private Coordinates prevCoordinates = null;

    /** The first location update done. */
    private boolean firstLocationUpdate = false;

    private ProviderStatusListener statusListener = null;
```

4 Create a constructor for the class. `setLocationListener` adds a `LocationListener` for updates at the defined interval.

For more information about this method, see the Location API specification.

```
    /**
     * Construct instance of this model class.
     */
    public TouristData(ProviderStatusListener listener)
    {
        statusListener = listener;

        ConfigurationProvider config = ConfigurationProvider.getInstance();

        // 1. Register LocationListener
        LocationProvider provider = config.getSelectedProvider();
        if (provider != null)
        {
            int interval = -1; // default interval of this provider
            int timeout = 0; // parameter has no effect.
            int maxage = 0; // parameter has no effect.

            provider.setLocationListener(this, interval, timeout, maxage);
        }

        // 2. Register ProxymityListener to each Landmark found from the
        // Landmark store.
        ControlPoints cp = ControlPoints.getInstance();

        Enumeration enumeration = cp.getLandMarks();
        if (enumeration != null)
        {
            while (enumeration.hasMoreElements())
            {
                Landmark lm = (Landmark) enumeration.nextElement();
                createProximityListener(lm.getQualifiedCoordinates());
            }
        }
    }
```

5 Create a setter method to `TouristUI` reference.

```
/**
 * Setter method to TouristUI reference.
 *
 * @param ui -
 *            Reference to TouristUI.
 */
public void setTouristUI(TouristUI ui)
{
    touristUI = ui;
}
```

6 Create a method for adding a new `ProximityListener` to a location provider. `addProximityListener` adds a `ProximityListener` for updates when proximity to the specified coordinates is detected.

For more information about this method, see the Location API specification.

```
/**
 * Adds new ProximityListener to the location provider. This method is
 * called when constructing instance of this class and when a new landmark
 * is saved by using LandmarkEditorUI.
 *
 * @param coordinates
 */
public void createProximityListener(Coordinates coordinates)
{
    try
    {
        LocationProvider.addProximityListener(this, coordinates,
                PROXIMITY_RADIUS);
    }
    catch (LocationException e)
    {
        // Platform does not have resources to add a new listener
        // and coordinates to be monitored or does not support
        // proximity monitoring at all
    }
}
```

7 Create a update event from `LocationListener` to start a new thread in order to prevent blocking. The `Location` class represents the standard set of basic location information. The `isValid` method returns whether this `Location` instance represents a valid location with coordinates or an invalid one where all the data, especially the latitude and longitude coordinates, may not be present. The `AddressInfo` class holds textual address information about a location and the `getAddress` method returns the `AddressInfo` associated with this Location object. The `getSpeed` method returns the terminal's current ground speed in meters per second (m/s) at the time of measurement.

For more information about these classes and methods, see the Location API specification.

```
/**
 * locationUpdated event from LocationListener interface. This method starts
 * a new thread to prevent blocking, because listener method MUST return
 * quickly and should not perform any extensive processing.
 *
 * Location parameter is set final, so that the anonymous Thread class can
 * access the value.
 */
public void locationUpdated(LocationProvider provider,
        final Location location)
{
    // First location update arrived, so we may show the UI (TouristUI)
    if (!firstLocationUpdate)
    {
        firstLocationUpdate = true;
        statusListener.firstLocationUpdateEvent();
    }

    if (touristUI != null)
    {
        new Thread()
```

```
            {
                public void run()
                {
                    if (location != null && location.isValid())
                    {
                        AddressInfo address = location.getAddressInfo();
                      QualifiedCoordinates coord = location.getQualifiedCoordinates
();
                        float speed = location.getSpeed();

                        touristUI.setInfo(address, coord, speed);
                        touristUI.setProviderState("Available");
                        touristUI.repaint();
                    }
                    else
                    {
                        touristUI.setProviderState("Not valid location data");
                        touristUI.repaint();
                    }
                }
            }.start();
        }
    }
```

8 Create a state changed event from `LocationListener` to start a new thread in order to prevent blocking.

```
    /**
     * providerStateChanged event from LocationListener interface. This method
     * starts a new thread to prevent blocking, because listener method MUST
     * return quickly and should not perform any extensive processing.
     *
     * newState parameter is set final, so that the anonymous Thread class can
     * access the value.
     */
    public void providerStateChanged(LocationProvider provider,
            final int newState)
    {
        if (touristUI != null)
        {
            new Thread()
            {
                public void run()
                {
                    switch (newState) {
                        case LocationProvider.AVAILABLE:
                            touristUI.setProviderState("Available");
                            break;
                        case LocationProvider.OUT_OF_SERVICE:
                            touristUI.setProviderState("Out of service");
                            break;
                        case LocationProvider.TEMPORARILY_UNAVAILABLE:
                            touristUI
                                    .setProviderState("Temporarily unavailable");
                            break;
                        default:
                            touristUI.setProviderState("Unknown");
                            break;
                    }

                    touristUI.repaint();
                }
            }.start();
        }
    }
```

9 Create a proximity event for `ProximityListener`, to be called only once when the terminal enters the proximity of the registered coordinates. The `getAddressInfo` method returns the `AddressInfo` associated with this `Location` object.

For more information about this method, see the Location API specification.

```
/**
 * proximity event from ProximityListener interface. The listener is called
 * only once when the terminal enters the proximity of the registered
 * coordinates. That's why no this method should not need to start a new
 * thread.
 */
public void proximityEvent(Coordinates coordinates, Location location)
{
    if (touristUI != null)
    {
        touristUI.setProviderState("Control point found!");

        Landmark lm = ControlPoints.getInstance().findNearestLandMark(
                coordinates);

        // landmark found from landmark store
        if (lm != null)
        {
            touristUI.setInfo(lm.getAddressInfo(), lm
                    .getQualifiedCoordinates(), location.getSpeed());
        }
        // landmark not found from the landmark store, this should not never
        // happen!
        else
        {
            touristUI.setInfo(location.getAddressInfo(), location
                    .getQualifiedCoordinates(), location.getSpeed());
        }

        touristUI.repaint();
    }
}
```

10 Create state changed event from `ProximityListener` to notify that the state of the monitoring has changed.

```
/**
 * monitoringStateChanged event from ProximityListener interface. Notifies
 * that the state of the proximity monitoring has changed. That's why this
 * method should not need to start a new thread.
 */
public void monitoringStateChanged(boolean isMonitoringActive)
{
    if (touristUI != null)
    {
        if (isMonitoringActive)
        {
            // proximity monitoring is active
            touristUI.setProximityState("Active");
        }
        else
        {
            // proximity monitoring can't be done currently.
            touristUI.setProximityState("Off");
        }

        touristUI.repaint();
    }
}
}
```

4.4.8 Implementing the CompassUI class

1 Create the `CompassUI` class file.

2 Import the required classes.

```
package com.nokia.example.location.tourist.ui;

import java.io.IOException;
```

```
import javax.microedition.lcdui.Canvas;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Graphics;
import javax.microedition.lcdui.Image;
import javax.microedition.location.Orientation;

import com.nokia.example.location.tourist.TouristMIDlet;
import com.nokia.example.location.tourist.model.ConfigurationProvider;
```

3 Create the `CompassUI` class and set it to extend `Canvas` and implement `CommandListener` and `Runnable`. Define the constants and commands used by this class.

```
/**
 * Viewer class representing a compass.
 */
public class CompassUI extends Canvas implements CommandListener, Runnable
{
    /** Constant value for X coordinate. Used in arrays. */
    private final int X = 0;

    /** Constant value for Y coordinate. Used in arrays. */
    private final int Y = 1;

    /** Compass center X point */
    private int centerX;

    /** Compass center Y point */
    private int centerY;

    /** Constant value representing one degree. */
    private final float degree = (float) (2 * Math.PI / 360.0);

    /** Current compass azimuth. */
    private float azimuth = 0.0f;

    /**
     * Is orientation relative to the magnetic field of the Earth or true north
     * and gravity.
     */
    private boolean isMagnetic;

    /** Flag telling is the compass update thread active */
    private boolean threadActive = false;

    /** Sleep 1000ms during each compass update */
    private final long SLEEPTIME = 1000;

    /** A reference to Route UI */
    private Displayable route = null;

    /** A reference to Pitch and Roll UI */
    private Displayable pitchrollUI = null;

    /** Command that swithes current displayable to Route UI */
    private Command routeCmd = new Command("Route", Command.BACK, 1);

    /** Command that swithes current displayable to Pitch and Roll UI */
    private Command prCmd = new Command("Pitch and Roll", Command.OK, 1);

    /** Compss background image. */
    private Image compassImage = null;
```

4 Create a constructor for the class.

```
    /**
     * Construct instance of this displayable.
     *
     * @param route
     *            is a reference to Route UI.
     */
```

```
public CompassUI(Displayable route)
{
    this.route = route;
    pitchrollUI = new PitchRollUI(this);

    loadCompassImage();

    centerX = getWidth() / 2;
    centerY = getHeight() / 2;

    addCommand(routeCmd);
    addCommand(prCmd);

    setCommandListener(this);
}
```

5 Create a method for loading a background image for the compass.

```
/**
 * Load compass backgound image from JAR file.
 */
private void loadCompassImage()
{
    String imageName = "/compass_small.gif";

    if (getWidth() > 160)
    {
        imageName = "/compass_large.gif";
    }

    try
    {
        compassImage = Image.createImage(getClass().getResourceAsStream(
                imageName));
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}
```

6 Create the two below methods the VM will call, depending on the status of the Canvas' visibility.

```
/**
 * VM calls this method immediately prior to this Canvas being made visible
 * on the display.
 */
protected void showNotify()
{
    // Actives compass update thread.
    threadActive = true;
    new Thread(this).start();
}

/**
 * VM calls this method shortly after the Canvas has been removed from the
 * display.
 */
protected void hideNotify()
{
    // Stops the thread.
    threadActive = false;
}
```

7 Create a method to render the canvas.

```
/**
 * Renders the canvas.
 *
 * @param g -
 *            the Graphics object to be used for rendering the Canvas
 */
```

```
protected void paint(Graphics g)
{
    // clean up canvas
    g.setColor(255, 255, 255);
    g.fillRect(0, 0, getWidth(), getHeight());
    int spikeLen = 5;
    int len = (compassImage.getWidth() / 2) - spikeLen;

    // draw compass background
    g.drawImage(compassImage, (getWidth() - compassImage.getWidth()) / 2,
            centerY, Graphics.LEFT | Graphics.VCENTER);

    // draw compass arrow
    g.setColor(0, 0, 255);
    drawArrow(g, degree * azimuth, len, spikeLen);

    // draw orientation type
    g.setColor(0, 0, 255);
    String otext = "True North";
    if (!isMagnetic)
    {
        otext = "Magnetic field";
    }
    g.drawString("Orientation: " + otext, 0, getHeight(), Graphics.BOTTOM
            | Graphics.LEFT);
}
```

8 Create a method to draw the compass arrow.

```
/**
 * Draw a compass arrow rotated to a certain angle.
 *
 * @param g
 *            is a reference to Graphics object.
 * @param angle
 *            in degrees [0.0,360.0)
 * @param len
 *            is arrows length.
 * @param spikeLen
 *            is length of arrows spike.
 */
private void drawArrow(Graphics g, float angle, int len, int spikeLen)
{
    int a[] = rotate(angle, 0, -(len - spikeLen));
    int b[] = rotate(angle, -spikeLen, -(len - spikeLen));
    int c[] = rotate(angle, 0, -len);
    int d[] = rotate(angle, spikeLen, -(len - spikeLen));
    int e[] = rotate(angle + (degree * 180.0), 0, -len);

    // use red foreground color
    g.setColor(255, 0, 0);
    g.drawLine(centerX, centerY, centerX + a[X], centerY + a[Y]);
    g.drawLine(centerX + b[X], centerY + b[Y], centerX + d[X], centerY
            + d[Y]);
    g.drawLine(centerX + b[X], centerY + b[Y], centerX + c[X], centerY
            + c[Y]);
    g.drawLine(centerX + c[X], centerY + c[Y], centerX + d[X], centerY
            + d[Y]);

    // use black foreground color
    g.setColor(0, 0, 0);
    g.drawLine(centerX, centerY, centerX + e[X], centerY + e[Y]);
}
```

9 Create the method for rotating the arrow.

```
/**
 * Rotate point (x,y) by degrees that angle parameter defines. The new
 * coordinate calculation is performed with a 2x2 rotate matrix.
 *
 * @param angle
 *            to be rotated
 * @param x
```

```
     *            coordinate
     * @param y
     *            coordinate
     * @return new coordinate pair in int array format [x,y]
     */
    private int[] rotate(double angle, int x, int y)
    {
        int rotated[] = new int[2];
        rotated[X] = (int) (Math.cos(angle) * x + Math.sin(angle) * y);
        rotated[Y] = (int) (-Math.sin(angle) * x + Math.cos(angle) * y);
        return rotated;
    }
```

10 Create a run method from `Runnable`. Its purpose is to update the azimuth, pitch and roll and to repaint the canvas. The `isOrientationMagnetic` method returns a boolean value that indicates whether this `Orientation` is relative to the magnetic field of the Earth or relative to true north and gravity. The `getCompassAzimuth` method returns the terminal's horizontal compass azimuth in degrees relative to either magnetic or true north.

For more information about these methods, see the Location API specification.

```
    /**
     * run method from Runnable interface. Updates azimuth, pitch and roll
     * values and repaints the canvas.
     *
     * If Orientation is supported on the terminal, compass sensor is either 2D
     * or 3D. If the terminals compass sensor providers only compass azimuth,
     * pitch and roll values are Float.NaN.
     *
     * @see HideNotify() method.
     */
    public void run()
    {
        // Keep the thread running until another displayable is set visible.
        // See also hideNotify() method.
        while (threadActive)
        {
            Orientation orientation = ConfigurationProvider.getInstance()
                    .getOrientation();

            if (orientation != null)
            {
                isMagnetic = orientation.isOrientationMagnetic();
                azimuth = orientation.getCompassAzimuth();
            }

            repaint();

            try
            {
                // Pause this thread for a secord before next update.
                Thread.sleep(SLEEPTIME);
            }
            catch (InterruptedException e)
            {
            }
        }
    }
```

11 Create a method for detecting command button presses.

```
    /**
     * Event indicating when a command button is pressed.
     *
     * @see javax.microedition.lcdui.CommandListener#commandAction
(javax.microedition.lcdui.Command,
     *      javax.microedition.lcdui.Displayable)
     */
    public void commandAction(Command command, Displayable d)
    {
        if (command == routeCmd)
```

```
            {
                TouristMIDlet.getDisplay().setCurrent(route);
            }
            else if (command == prCmd)
            {
                TouristMIDlet.getDisplay().setCurrent(pitchrollUI);
            }
        }

    }
```

4.4.9 Implementing the LandmarkEditorUI class

1 Create the `LandmarkEditorUI` class file.

2 Import the required classes.

```
package com.nokia.example.location.tourist.ui;

import java.io.IOException;
import java.util.Enumeration;

import javax.microedition.lcdui.Alert;
import javax.microedition.lcdui.AlertType;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.List;
import javax.microedition.lcdui.StringItem;
import javax.microedition.lcdui.TextField;
import javax.microedition.location.AddressInfo;
import javax.microedition.location.Landmark;
import javax.microedition.location.LandmarkException;
import javax.microedition.location.QualifiedCoordinates;

import com.nokia.example.location.tourist.TouristMIDlet;
import com.nokia.example.location.tourist.Utils;
import com.nokia.example.location.tourist.model.ControlPoints;
import com.nokia.example.location.tourist.model.TouristData;
```

3 Create the `LandmarkEditorUI` class and set it to extend `Form` and implement `CommandListener`. Define the `TextFields` used by the form.

```
/**
 * Viewer class enabling landmarks address info monifications.
 */
class LandmarkEditorUI extends Form implements CommandListener
{
    // Landmark fields, (*) = required field
    private TextField nameField = new TextField("Name (*):", "", 20,
            TextField.ANY);

    private TextField descField = new TextField("Description (*):", "", 40,
            TextField.ANY);

    // AddressInfo fields
    private TextField countryField = new TextField("Country:", "", 20,
            TextField.ANY);

    private TextField stateField = new TextField("State:", "", 20,
            TextField.ANY);

    private TextField cityField = new TextField("City:", "", 20, TextField.ANY);

    private TextField streetField = new TextField("Street:", "", 20,
            TextField.ANY);

    private TextField buildingNameField = new TextField("Building name (*):",
            "", 20, TextField.ANY);
```

Define the coordinate information `StringItems` and the commands and constants used by the Landmark selector list. The `QualifiedCoordinates` class represents coordinates as latitude-longitude-altitude values that are associated with an accuracy value.

For more information about this method, see the Location API specification.

```
// Coordinate information
private StringItem lat = new StringItem("Lat:", "");

private StringItem lon = new StringItem("Lon:", "");

private StringItem alt = new StringItem("Alt:", "");

/** Landmark selector list */
private List list = new List("Landmarks:", List.EXCLUSIVE);

private Command saveNewCmd = new Command("Save", Command.OK, 1);

private Command saveUpdatedCmd = new Command("Save", Command.OK, 1);

private Command updateCmd = new Command("Update", Command.OK, 1);

private Command removeCmd = new Command("Remove", Command.OK, 1);

private Command listCmd = new Command("List", Command.OK, 1);

private Command routeCmd = new Command("Route", Command.BACK, 1);

public static final int MODE_UPDATE = 0;

public static final int MODE_ADDNEW = 1;

private QualifiedCoordinates coord = null;

private Displayable route = null;

private TouristData data = null;
```

4 Create the constructor for the class.

```
/**
 * Construct Landmark Editor UI Form.
 *
 * @param route -
 *            a reference of Route UI.
 * @param data -
 *            a reference to TouristData model layer class.
 */
public LandmarkEditorUI(Displayable route, TouristData data)
{
    super("Landmark Editor");

    this.route = route;
    this.data = data;

    // route and list commands always available on landmark editor
    addCommand(routeCmd);
    addCommand(listCmd);
    setCommandListener(this);

    // route command always available on landmarks list
    list.addCommand(routeCmd);
    list.setCommandListener(this);
}
```

5 Create a method for initializing the UI components and showing the landmark editor. The `getAltitude` method returns the altitude component of this coordinate.

```
/**
 * Initialize UI components and show the landmark editor.
 */
```

```
public void showEditor(QualifiedCoordinates newCoord, int mode)
{
    this.coord = newCoord;

    // initialize coordinate values
    lat.setText(Utils.formatDouble(newCoord.getLatitude(), 3));
    lon.setText(Utils.formatDouble(newCoord.getLongitude(), 3));
    alt.setText(Utils.formatDouble(newCoord.getAltitude(), 1));

    // initialize editable test field values
    nameField.setString("");
    descField.setString("");
    countryField.setString("");
    stateField.setString("");
    cityField.setString("");
    streetField.setString("");
    buildingNameField.setString("");

    deleteAll();
    append(nameField);
    append(descField);
    append(countryField);
    append(stateField);
    append(cityField);
    append(streetField);
    append(buildingNameField);

    append(lat);
    append(lon);
    append(alt);


    // Update existing landmark.
    if (mode == MODE_UPDATE)
    {
        Landmark lm = ControlPoints.getInstance().findNearestLandMark(
                newCoord);

        if (lm != null)
        {
            nameField.setString(lm.getName());
            descField.setString(lm.getDescription());

            AddressInfo info = lm.getAddressInfo();
            if (info != null)
            {
                countryField.setString(info.getField(AddressInfo.COUNTRY));
                stateField.setString(info.getField(AddressInfo.STATE));
                cityField.setString(info.getField(AddressInfo.CITY));
                streetField.setString(info.getField(AddressInfo.STREET));
                buildingNameField.setString(info
                        .getField(AddressInfo.BUILDING_NAME));
            }
        }

        removeCommand(updateCmd);
        removeCommand(saveNewCmd);
        addCommand(saveUpdatedCmd);
        addCommand(listCmd);
    }
    // Add new landmark to the landmark store.
    else if (mode == MODE_ADDNEW)
    {
        removeCommand(updateCmd);
        removeCommand(saveUpdatedCmd);
        addCommand(saveNewCmd);
        addCommand(listCmd);
    }

    TouristMIDlet.getDisplay().setCurrent(this);
}
```

6 Create a method for showing the landmark selector list.

```
/**
 * Show landmark selector List. Content of the list is refreshed each time
 * this method is called.
 */
public void showList()
{
    list.deleteAll();

    Landmark lm = null;
    Enumeration landmarks = ControlPoints.getInstance().getLandMarks();

    // Check whether landmarks can be founds from the Landmark store.
    if (landmarks != null)
    {
        while (landmarks.hasMoreElements())
        {
            lm = ((Landmark) landmarks.nextElement());
            list.append(lm.getName(), null);
        }

        list.addCommand(updateCmd);
        list.addCommand(removeCmd);
    }
    // No landmarks found (list is empty)
    else
    {
        list.removeCommand(updateCmd);
        list.removeCommand(removeCmd);
    }

    TouristMIDlet.getDisplay().setCurrent(list);
}
```

7 Create a method for checking that the required fields are not missing.

```
/**
 * Check whether any required fields are missing. (*) on the TextFields label
 * indicates that field is mandatory.
 *
 * @return Name of the missing field name or null indicating no required
 *         fields are missing.
 */
private String checkRequiredFields()
{
    if (nameField.getString().equals(""))
    {
        return "Name";
    }
    else if (nameField.getString().equals(""))
    {
        return "Description";
    }
    else if (buildingNameField.getString().equals(""))
    {
        return "Building name";
    }

    return null;
}
```

8 Create a method for generating a landmark from UI component values.

```
/**
 * Generate landmark from UI component values.
 *
 * @return Genereated Landmark.
 */
private Landmark generateLandmark()
{
    String field = checkRequiredFields();
    if (field != null)
```

```
            {
                Alert alert = new Alert("Error", "Value for required field "
                        + field + " is missing.", null, AlertType.ERROR);
                alert.setTimeout(3000); // 3 secs

                TouristMIDlet.getDisplay().setCurrent(alert, this);
                return null;
            }

            AddressInfo info = new AddressInfo();
            info.setField(AddressInfo.COUNTRY, countryField.getString());
            info.setField(AddressInfo.STATE, stateField.getString());
            info.setField(AddressInfo.CITY, cityField.getString());
            info.setField(AddressInfo.STREET, streetField.getString());
            info.setField(AddressInfo.BUILDING_NAME, buildingNameField.getString());

            Landmark lm = new Landmark(nameField.getString(),
                    descField.getString(), coord, info);

            return lm;
        }
```

9 Write code for detecting key presses.

```
        /**
         * Event indicating when a command button is pressed.
         *
         * @see javax.microedition.lcdui.CommandListener#commandAction
    (javax.microedition.lcdui.Command,
         *      javax.microedition.lcdui.Displayable)
         */
        public void commandAction(Command command, Displayable displayable)
        {
            Landmark landmark = null;

            // Add new Landmark to the LandmarkStore
            if (command == saveNewCmd)
            {
                landmark = generateLandmark();
                if (landmark != null)
                {
                    try
                    {
                        ControlPoints.getInstance().addLandmark(landmark);
                        data.createProximityListener(coord);
                    }
                    catch (IOException e)
                    {
                        Alert alert = new Alert("Error",
                                "I/O Error during add operation", null,
                                AlertType.ERROR);
                        alert.setTimeout(3000); // 3 secs
                        TouristMIDlet.getDisplay().setCurrent(alert);
                    }

                    // New landmark is available on the list
                    showList();
                }
            }
            // Update existing landmark
            else if (command == saveUpdatedCmd)
            {
                landmark = generateLandmark();
                if (landmark != null)
                {
                    try
                    {
                        ControlPoints.getInstance().updateLandmark(landmark);
                        data.createProximityListener(coord);
                    }
                    catch (IOException e)
                    {
                        Alert alert = new Alert("Error",
```

```
                        "I/O Error during update operation", null,
                        AlertType.ERROR);
                alert.setTimeout(3000); // 3 secs
                TouristMIDlet.getDisplay().setCurrent(alert);
            }
            catch (LandmarkException e)
            {
                // Landmark instance passed as the parameter does not
                // belong to this LandmarkStore or does not exist in the
                // store any more.
                Alert alert = new Alert("Error",
                        "Unexpected error: can not find landmark from "
                        + "the landmark store.", null,
                        AlertType.ERROR);
                alert.setTimeout(3000); // 3 secs
                TouristMIDlet.getDisplay().setCurrent(alert);
            }

            // Updated landmark is available on the list
            showList();
        }
    }
    // Go back to the previous screen
    else if (command == routeCmd)
    {
        TouristMIDlet.getDisplay().setCurrent(route);
    }
    // Show landmark editor for the selected landmark.
    else if (command == updateCmd)
    {
        int index = list.getSelectedIndex();
        landmark = ControlPoints.getInstance().findLandmark(index);

        showEditor(landmark.getQualifiedCoordinates(), MODE_UPDATE);
    }
    // Remove selected Landmark from LandmarkStore
    else if (command == removeCmd)
    {
        try
        {
            int index = list.getSelectedIndex();
            landmark = ControlPoints.getInstance().findLandmark(index);
            ControlPoints.getInstance().removeLandmark(landmark);

            Alert alert = new Alert("Information",
                    "Landmark removed successfully.", null, AlertType.INFO);
            alert.setTimeout(3000); // 3 secs
            TouristMIDlet.getDisplay().setCurrent(alert);
        }
        catch (IOException e)
        {
            Alert alert = new Alert("Error", "I/O Error during operation",
                    null, AlertType.ERROR);
            alert.setTimeout(3000); // 3 secs
            TouristMIDlet.getDisplay().setCurrent(alert);
        }
        catch (LandmarkException le)
        {
            Alert alert = new Alert("Error",
                    "landmark can not be found from the landmark store.",
                    null, AlertType.ERROR);
            alert.setTimeout(3000); // 3 secs
            TouristMIDlet.getDisplay().setCurrent(alert);
        }

        showList();
    }
    // Show the list of Landmarks
    else if (command == listCmd)
    {
        showList();
```

```
        }
    }
}
```

4.4.10 Implementing the MessageUI class

1 Create the `MessageUI` class file.

2 Import the required classes.

```
package com.nokia.example.location.tourist.ui;

import javax.microedition.lcdui.Alert;
import javax.microedition.lcdui.AlertType;
import javax.microedition.lcdui.Gauge;

import com.nokia.example.location.tourist.TouristMIDlet;
```

3 Create the `MessageUI` class.

```
/**
 * Viewer class containing a general suppose Alert notification(s).
 */
public class MessageUI
{
```

4 Create a method for showing an alert that the Location API is not supported.

```
    /**
     * Shows a alert that Location API is not supported.
     */
    public static void showApiNotSupported()
    {
        Alert alert = new Alert("Information",
                "Device do not support Location API", null, AlertType.INFO);
        TouristMIDlet.getDisplay().setCurrent(alert);
    }
```

5 Create a method for showing an alert that location data is unavailable.

```
    /**
     * Shows a alert that Location data is not available.
     */
    public static void showLocationDataNotAvailable()
    {
        Alert alert = new Alert("Information",
                "Location data is not yet available.", null, AlertType.INFO);
        TouristMIDlet.getDisplay().setCurrent(alert);
    }
```

6 Create a status dialog about the state of the location provider.

```
    /**
     * Show a status dialog informing about state of location provider.
     */
    public static void showLocationProviderState()
    {
        Gauge indicator = new Gauge(null, false, 50, 1);
        indicator.setValue(Gauge.CONTINUOUS_RUNNING);

        Alert alert = new Alert("Information",
                "Please wait, looking for location data....", null,
                AlertType.INFO);
        alert.setIndicator(indicator);

        TouristMIDlet.getDisplay().setCurrent(alert);
    }
}
```

4.4.11 Implementing the PitchRollUI class

1 Create the `PitchRollUI` class file.

2 Import the required classes.

```
package com.nokia.example.location.tourist.ui;

import javax.microedition.lcdui.Canvas;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Graphics;
import javax.microedition.location.Orientation;

import com.nokia.example.location.tourist.TouristMIDlet;
import com.nokia.example.location.tourist.model.ConfigurationProvider;
```

3 Create the `PitchRollUI` class and set it to extend `Canvas` and implement `CommandListener` and `Runnable`. Define the constants, classes commands used by the class. Also create the constructor.

```
/**
 * Viewer class that renders orientations pitch and roll values to meters.
 */
public class PitchRollUI extends Canvas implements CommandListener, Runnable
{
    private CompassUI compass;

    private float pitch = Float.NaN;

    private float roll = Float.NaN;

    /** Flag telling is the compass update thread active */
    private boolean threadActive = false;

    /** Sleep 1000ms during each compass update */
    private final long SLEEPTIME = 1000;

    private Command compassCmd = new Command("Compass", Command.BACK, 1);

    public PitchRollUI(CompassUI compass)
    {
        this.compass = compass;

        addCommand(compassCmd);
        setCommandListener(this);
    }
```

4 Create the methods that control the compass' update thread.

```
    /**
     * VM calls this method immediately prior to this Canvas being made visible
     * on the display.
     */
    protected void showNotify()
    {
        // Actives compass update thread.
        threadActive = true;
        new Thread(this).start();
    }

    /**
     * VM calls this method shortly after the Canvas has been removed from the
     * display.
     */
    protected void hideNotify()
    {
        // Stops the thread.
```

```
        threadActive = false;
    }
```

5 Create a `run` method for the class. It needs to update the azimuth, pitch and roll values and repaint the canvas. The `getPitch` method returns the terminal's tilt in degrees defined as an angle in the vertical plane orthogonal to the ground, and through the longitudinal axis of the terminal, whereas the `getRoll` method returns the terminal's rotation in degrees around its own longitudinal axis.

For more information about these methods, see the Location API specification.

```
/**
 * run method from Runnable interface. Updates azimuth, pitch and roll
 * values and repaints the canvas.
 *
 * If Orientation is supported on the terminal, support level may be either
 * 2D or 3D depending on the compass sensor. If the sensor providers only
 * compass azimuth, pitch and roll values are Float.NaN.
 */
public void run()
{
    // Run this thread until this displayable is not visiable.
    // See also hideNotify() method.
    while (threadActive)
    {
        Orientation orientation = ConfigurationProvider.getInstance()
                .getOrientation();

        if (orientation != null)
        {
            pitch = orientation.getPitch();
            roll = orientation.getRoll();
        }

        repaint();

        try
        {
            // Pause this thread for a secord before next update.
            Thread.sleep(SLEEPTIME);
        }
        catch (InterruptedException e)
        {
        }
    }
}
```

6 Create a method for rendering the canvas.

```
/**
 * Renders the canvas.
 *
 * @param g -
 *            the Graphics object to be used for rendering the Canvas
 */
protected void paint(Graphics g)
{
    // clean up canvas
    g.setColor(255, 255, 255);
    g.fillRect(0, 0, getWidth(), getHeight());

    paintPitch(g);
    paintRoll(g);
}
```

7 Create a method for drawing a meter with a scale.

```
/**
 * Draws a customized meter with a scale.
 */
protected void drawMeter(Graphics g, int min, int max, int smallStep,
        int largeStep, int baseline, int margin)
```

```
    {
        double position;
        double scale = 100.0; // scale value to minimize rounding error
        int x;

        g.setColor(0x0000ff);
        g.drawLine(margin, baseline, getWidth(), baseline);

        for (int i = min; i <= max; i += smallStep)
        {
            position = (((i + max) * scale * (getWidth() - margin)) / (2 * max))
                    / scale;
            x = margin + (int) position;
            g.drawLine(x, baseline - 3, x, baseline + 3);
        }

        for (int i = min; i <= max; i += largeStep)
        {
            position = (((i + max) * scale * (getWidth() - margin)) / (2 * max))
                    / scale;
            x = margin + (int) position;
            g.drawLine(x, baseline - 5, x, baseline + 5);
        }
    }
```

8 Create a method for painting the pitch meter and placing the current pitch value to it.

```
    /**
     * Paint pitch meter and place the current pitch value to the meter.
     *
     * @param g -
     *            the Graphics object to be used for rendering the Canvas
     */
    protected void paintPitch(Graphics g)
    {
        int baseline = 10;
        int textAreaWidth = 50;
        double position;

        g.setColor(0x000000);
        g.drawString("Pitch", 0, baseline - 5, Graphics.TOP | Graphics.LEFT);
        g.drawString("-90", textAreaWidth + 1, baseline, Graphics.TOP
                | Graphics.LEFT);
        g.drawString("0", textAreaWidth + (getWidth() - textAreaWidth) / 2,
                baseline, Graphics.TOP | Graphics.HCENTER);
        g.drawString("+90", getWidth(), baseline, Graphics.TOP
                | Graphics.RIGHT);

        drawMeter(g, -90, 90, 10, 30, baseline, textAreaWidth);

        // Draw the marker only if terminals pitch is available.
        if (pitch != Float.NaN)
        {
            position = (((pitch + 90.0) * 100 * (getWidth() - textAreaWidth)) / (2
* 90)) / 100;
            g.setColor(0x000000);
            g.fillRect(textAreaWidth + (int) position - 3, baseline - 2, 5, 5);
        }
    }
```

9 Create a method for painting the roll meter and placing the current roll value to it.

```
    /**
     * Paint roll meter and place the current pitch value to the meter.
     *
     * @param g -
     *            the Graphics object to be used for rendering the Canvas
     */
    protected void paintRoll(Graphics g)
    {
        int baseline = 40;
        int textAreaWidth = 50;
        double position;
```

```
            g.setColor(0x000000);
            g.drawString("Roll", 0, baseline - 5, Graphics.TOP | Graphics.LEFT);
            g.drawString("-180", textAreaWidth + 1, baseline, Graphics.TOP
                    | Graphics.LEFT);
            g.drawString("0", textAreaWidth + (getWidth() - textAreaWidth) / 2,
                    baseline, Graphics.TOP | Graphics.HCENTER);
            g.drawString("+180", getWidth(), baseline, Graphics.TOP
                    | Graphics.RIGHT);

            drawMeter(g, -180, 180, 15, 60, baseline, textAreaWidth);

            // Draw the marker only if terminals roll is available.
            if (roll != Float.NaN)
            {
                position = (((roll + 180.0) * 100 * (getWidth() - textAreaWidth)) / (2
* 180)) / 100;
                g.setColor(0x000000);
                g.fillRect(textAreaWidth + (int) position - 3, baseline - 2, 5, 5);
            }
        }
```

10 Create an event for monitoring command button key presses.

```
        /**
         * Event indicating when a command button is pressed.
         *
         * @see javax.microedition.lcdui.CommandListener#commandAction
(javax.microedition.lcdui.Command,
         *      javax.microedition.lcdui.Displayable)
         */
        public void commandAction(Command command, Displayable d)
        {
            if (command == compassCmd)
            {
                TouristMIDlet.getDisplay().setCurrent(compass);
            }
        }

    }
```

4.4.12 Implementing the ProviderQueryUI class

1 Create the `ProviderQueryUI` class file.

2 Import the required classes.

```
package com.nokia.example.location.tourist.ui;

import javax.microedition.lcdui.Alert;
import javax.microedition.lcdui.AlertType;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.StringItem;

import com.nokia.example.location.tourist.TouristMIDlet;
```

3 Create the `ProviderQueryUI` class. Define the classes, constants and commands used by the class.

```
/**
 * Viewer class that is responsible for all the UI actions when the application
 * is seaching for the location provider.
 */
public class ProviderQueryUI
{
    /** Status information Form */
    private Form searchForm = new Form("Seaching location provider...");

    /** StringItem showing the current status. */
    private StringItem infoItem = new StringItem("Status:", "");
```

```
    /** Provider cost selection command - Yes. */
    private Command yesCmd = new Command("Yes", Command.OK, 1);

    /** Provider cost selection command - No. */
    private Command noCmd = new Command("No", Command.STOP, 1);

    /** A boolean indicating may user allow location provider cost. */
    private boolean result = false;

    private static final String COST_QUERY_MESSAGE = "Cost free location providers
can not be found. Do you with continue "
            + "search with providers that cost?";

    private static final String OUT_OF_SERVICE_MESSAGE = "All Location providers
are currently out of service. Please unsure "
            + "that location-providing module is properly connected.";

    private static final String SEACHING_FREE_PROVIDERS = "Seaching for free
location providers.";

    private static final String SEACHING_COST_PROVIDERS = "Seaching for providers
that may cost.";

    private static final String NOT_FOUND_MESSAGE = "Try again after location-
providing module is properly connected.";

    private static final String NO_FREE_SERVICE_SERVICE_MESSAGE = "Cost free
location providers can not be found. Please ensure "
            + "that location-providing module is properly connected.";
```

4 Create the constructor for the class.

```
    /**
     * Construct the UI with default values.
     */
    public ProviderQueryUI()
    {
        infoItem.setText(SEACHING_FREE_PROVIDERS);
        searchForm.append(infoItem);
    }
```

5 Create methods for showing "out of service" and "no cost free location provider found" error messages.

```
    /**
     * Show out of service error message.
     */
    public void showOutOfService()
    {
        Alert alert = new Alert("Error", OUT_OF_SERVICE_MESSAGE, null,
                AlertType.ERROR);
        alert.setTimeout(Alert.FOREVER);
        TouristMIDlet.getDisplay().setCurrent(alert, searchForm);
        infoItem.setText(NOT_FOUND_MESSAGE);
    }

    /**
     * Show no cost free location provider found error message.
     */
    public void showNoFreeServiceFound()
    {
        Alert alert = new Alert("Error", NO_FREE_SERVICE_SERVICE_MESSAGE, null,
                AlertType.ERROR);
        alert.setTimeout(Alert.FOREVER);
        TouristMIDlet.getDisplay().setCurrent(alert, searchForm);
        infoItem.setText(NOT_FOUND_MESSAGE);
    }
```

6 Create a method for querying the user about the cost of using the location provider.

```
    /**
     * Query the user whether the use of location provider may cost. The use of
     * this method is locked with synchronized keyword, so only one thread can
```

```
         * access this method at once.
         *
         * @return a boolean indicating may user allow location provider cost.
         */
        public synchronized boolean confirmCostProvider()
        {
            Alert alert = new Alert("Confimnation", COST_QUERY_MESSAGE, null,
                    AlertType.CONFIRMATION);
            alert.addCommand(yesCmd);
            alert.addCommand(noCmd);
            alert.setTimeout(Alert.FOREVER);

            // Set the monitoring object to be this instance.
            final ProviderQueryUI hinstance = this;

            // Add a CommandLister as anomynous inner class
            alert.setCommandListener(new CommandListener()
            {
                /*
                 * Event indicating when a command button is pressed.
                 *
                 * @see javax.microedition.lcdui.CommandListener#commandAction
(javax.microedition.lcdui.Command,
                 *      javax.microedition.lcdui.Displayable)
                 */
                public void commandAction(Command command, Displayable d)
                {
                    if (command == yesCmd)
                    {
                        infoItem.setText(SEACHING_COST_PROVIDERS);
                        result = true;
                        synchronized (hinstance)
                        {
                            // Wake up the monitoring object
                            hinstance.notifyAll();
                        }
                    }
                    else if (command == noCmd)
                    {
                        result = false;
                        infoItem.setText(NOT_FOUND_MESSAGE);
                        synchronized (hinstance)
                        {
                            // Wake up the monitoring object
                            hinstance.notifyAll();
                        }
                    }

                }
            });

            TouristMIDlet.getDisplay().setCurrent(alert, searchForm);

            // Wait indefinitely for notification.
            try
            {
                wait();
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }

            TouristMIDlet.getDisplay().setCurrent(searchForm);

            return result;
        }

    }
```

4.4.13 Implementing the TouristUI class

1 Create the `TouristUI` class file.

2 Import the required classes.

```
package com.nokia.example.location.tourist.ui;

import javax.microedition.lcdui.Canvas;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Font;
import javax.microedition.lcdui.Graphics;

import javax.microedition.location.AddressInfo;
import javax.microedition.location.QualifiedCoordinates;

import com.nokia.example.location.tourist.TouristMIDlet;
import com.nokia.example.location.tourist.Utils;
import com.nokia.example.location.tourist.model.ConfigurationProvider;
import com.nokia.example.location.tourist.model.TouristData;
```

3 Create the `TouristUI` class and set it to extend `Canvas` and implement `CommandListener`. Define the constants and commands used by the class and create the constructor.

```
/**
 * Viewer class that renders current location updates.
 */
public class TouristUI extends Canvas implements CommandListener
{
    /** The current state of the location provider as a String */
    private String providerState = "Unknown";

    /** Proximity monitoring state. */
    private String proximityState = "Waiting";

    private AddressInfo info;

    private QualifiedCoordinates coord;

    private float speed;

    /** Command that shows compass canvas */
    private Command compassCmd = new Command("Compass", Command.OK, 1);

    /** Command that shows Landmark editor UI */
    private Command editorCmd = new Command("Editor", Command.STOP, 1);

    /** Rerefence to the Landmark editor UI */
    private LandmarkEditorUI editorUI = null;

    /** Rerefence to the Compass UI */
    private CompassUI compassUI = null;

    public TouristUI(TouristData data)
    {
        editorUI = new LandmarkEditorUI(this, data);
        compassUI = new CompassUI(this);

        checkSupportedFeatures();

        addCommand(editorCmd);
        setCommandListener(this);
    }
```

4 Create a method for enabling the supported and disabling the unsupported features of the Location API on the UI.

```
/**
 * Enable supported Location API features on the UI and disable unsupported
 * features.
 */
protected void checkSupportedFeatures()
{
    if (ConfigurationProvider.getInstance().isOrientationSupported())
    {
        addCommand(compassCmd);
    }
    else
    {
        removeCommand(compassCmd);
    }
}
```

5 Create methods for setting the provider and proximity states.

```
public void setProviderState(String state)
{
    providerState = state;
}

public void setProximityState(String state)
{
    proximityState = state;
}

public void setInfo(AddressInfo info, QualifiedCoordinates coord,
        float speed)
{
    this.info = info;
    this.coord = coord;
    this.speed = speed;
}
```

6 Create a method for rendering the canvas.

```
/**
 * Renders the canvas.
 *
 * @param g -
 *            the Graphics object to be used for rendering the Canvas
 */
protected void paint(Graphics g)
{
    Font f = Font.getFont(Font.FACE_SYSTEM, Font.STYLE_PLAIN,
            Font.SIZE_SMALL);
    g.setFont(f);

    // use font height as a line height
    int lineHeight = f.getHeight();
    // current line counter
    int line = 0;

    // clean the backround
    g.setColor(0xffffff);
    g.fillRect(0, 0, getWidth(), getHeight());

    g.setColor(0x0000ff);
    g.drawString("Provider state: " + providerState, 0, lineHeight
            * (line++), Graphics.LEFT | Graphics.TOP);
    g.drawString("Proximity monitoring: " + proximityState, 0, lineHeight
            * (line++), Graphics.LEFT | Graphics.TOP);

    if (coord != null)
    {
        double lat = coord.getLatitude();
        double lon = coord.getLongitude();

        g.drawString("Lat, Lon (" + Utils.formatDouble(lat, 3) + ", "
                + Utils.formatDouble(lon, 3) + ")", 0, lineHeight
                * (line++), Graphics.TOP | Graphics.LEFT);
```

```
                g.drawString("Speed: " + Utils.formatDouble(speed, 2) + " m/s", 0,
                        lineHeight * (line++), Graphics.TOP | Graphics.LEFT);
            }

            // Check if AddressInfo is available
            if (info != null)
            {
                String country = info.getField(AddressInfo.COUNTRY);
                String state = info.getField(AddressInfo.STATE);
                String city = info.getField(AddressInfo.CITY);
                String street = info.getField(AddressInfo.STREET);
                String buildingName = info.getField(AddressInfo.BUILDING_NAME);

                g.setColor(0x000000);

                if (country != null)
                    g.drawString("Country: " + country, 0, lineHeight * (line++),
                            Graphics.TOP | Graphics.LEFT);
                if (state != null)
                    g.drawString("State: " + state, 0, lineHeight * (line++),
                            Graphics.TOP | Graphics.LEFT);
                if (city != null)
                    g.drawString("City: " + city, 0, lineHeight * (line++),
                            Graphics.TOP | Graphics.LEFT);
                if (street != null)
                    g.drawString("Street: " + street, 0, lineHeight * (line++),
                            Graphics.TOP | Graphics.LEFT);
                if (buildingName != null)
                    g.drawString("Building name: " + buildingName, 0, lineHeight
                            * (line++), Graphics.TOP | Graphics.LEFT);
            }
        }
```

7 Create an event for detecting command key presses.

```
        /**
         * Event indicating when a command button is pressed.
         *
         * @see javax.microedition.lcdui.CommandListener#commandAction
    (javax.microedition.lcdui.Command,
         *      javax.microedition.lcdui.Displayable)
         */
        public void commandAction(Command command, Displayable d)
        {
            if (command == editorCmd)
            {
                if (coord != null)
                {
                    editorUI.showEditor(coord, LandmarkEditorUI.MODE_ADDNEW);
                }
                else
                {
                    MessageUI.showLocationDataNotAvailable();
                }
            }
            else if (command == compassCmd)
            {
                TouristMIDlet.getDisplay().setCurrent(compassUI);
            }
        }

    }
```

4.4.14 Building and running in an emulator

After you have created all the necessary files for the project, you can build the project and run the application in an emulator.

Note: The MIDlet used in this example was built using the Eclipse IDE. Please see the instructions for building MIDlet projects for Eclipse that come with the Getting Started with Mobile Java section of the Java™ ME Developer's Library at http://www.forum.nokia.com/main/resources/technologies/java/documentation/Java_ME_developers_library.htm.

You can also construct them using command line or other IDEs of your choice.

Note: The example application does not work in Series 40.

4.4.15 Deploying to a device

Deploy test packages onto your mobile device, using a USB cable, Bluetooth connection, infrared, or other hardware compatible connection. Alternatively, deploy the application using OTA technology.

For instructions on how to do this, see the Getting Started with Mobile Java section of the Java™ ME Developer's Library at http://www.forum.nokia.com/main/resources/technologies/java/documentation/Java_ME_developers_library.htm.