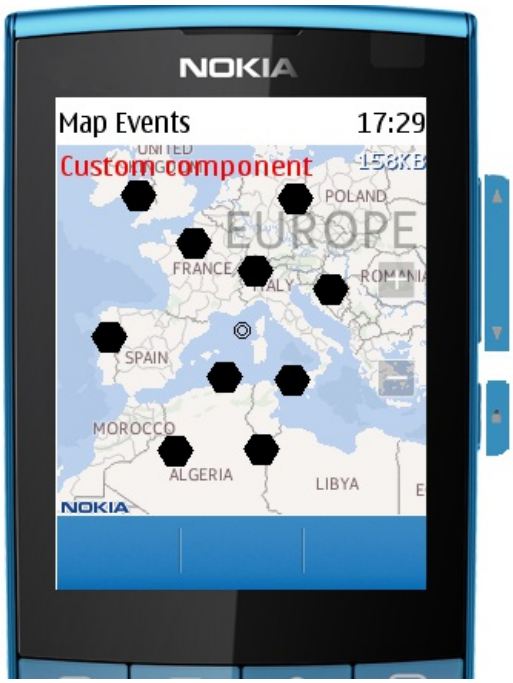
Handling map events with Maps API for Java ME

This article explains how to handle key and pointer events related to a `MapDisplay` with the HERE Maps API for Java ME

Introduction



The HERE Maps API for Java ME offers an in-built mechanism to intercept and handle events generated by the user when interacting on a map, and more specifically with a `MapDisplay` object.

This mechanism is based on two main interfaces:

- `MapComponent` - the `MapComponent` is intended to extend the map default functionality with new features
- `EventListener` - provides all the necessary callbacks to intercept key and pointer events

MapComponent and EventListener

A `MapComponent` can add, replace or remove features of a map. When implementing the `MapComponent` interface, the main methods that need to be implemented are:

- `attach()` and `detach()` - called when the `MapComponent` object is attached or detached from a `MapDisplay`
- `mapUpdated()` - called when the attached `MapDisplay` object gets updated
- `paint()` - can be used to paint the `MapComponent`'s graphics (if any) over the map
- `getEventListener()` - returns the `EventListener` object used to handle the generated key and pointer events

The `EventListener` interface defines methods to handle both key and pointer events, and specifically:

- `keyPressed()`, `keyReleased()` and `keyRepeated()` Events - for key events
- `pointerPressed()`, `pointerDragger()`, `pointerReleased()` - for pointer events

Sample implementation: adding markers dynamically

This code sample shows a possible implementation of the MapComponent/EventListener model, that will allow the user to dynamically place markers on a map by touching the point of interest.

The base MIDlet

The base MIDlet used in this sample simply shows a map, as shown in the code snippet below:

```
public class MapEventsMIDlet extends MIDlet {

    public void startApp() {
```


(C) Copyright Nokia 2014. All rights reserved.

```
        ApplicationContext ctx = ApplicationContext.getInstance();

        ctx.setAppID("MyAppId");
        ctx.setToken("MyToken");

        Display display = Display.getDisplay(this);
        MapCanvas mapCanvas = new MyMapCanvas(display);
        display.setCurrent(mapCanvas);

        mapCanvas.getMapDisplay().addMapComponent(new
CustomMapComponent(mapCanvas.getMapFactory()));
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }

    private class MyMapCanvas extends MapCanvas {

        public MyMapCanvas(Display display) {
            super(display);
        }

        public void onMapUpdateError(String description, Throwable detail, boolean
critical) {
            Alert alertView = new Alert("Map error: " + detail.getMessage());
            display.setCurrent(alertView);
        }
        public void onMapContentComplete() {
        }
    }
}
```

The CustomEventListener

The sample MIDlet must allow the user to add markers by touching the map: the relevant event callback so is the `pointerPressed()`.

In order to build a `MapMarker` object, a `MapFactory` instance is necessary, so let's define a constructor that passes a reference to an instance:

```
public class CustomEventListener implements EventListener
{
    MapFactory mapFactory = null;

    public CustomEventListener(MapFactory factory)
    {
        this.mapFactory = factory;
    }
}
```

Also, since markers must be added to a `MapDisplay`, a reference to that is necessary, and so a variable and its setter are defined:

```
public class CustomEventListener implements EventListener
```


(C) Copyright Nokia 2014. All rights reserved.

```
{
    MapDisplay mapDisplay = null;

    [...]

    public void setMapDisplay(MapDisplay display) {
        this.mapDisplay = display;
    }
}
```

The `pointerPressed()` behaviour should be the following:

- when the user clicks on the map, check the **x and y coordinates** of the event
- translate the x and y into **geographical coordinates**: this can be done by using the `MapDisplay.pixelToGeo()` method
- use the calculated geographical coordinates to create a `MapMarker`
- add the marker to the map

These operations can be implemented as shown in the following code snippet:

```
public boolean pointerPressed(int x, int y) {

        Point clickPoint = new Point(x, y);

        GeoCoordinate clickCoordinate = mapDisplay.pixelToGeo(clickPoint);

        MapStandardMarker marker = mapFactory.createStandardMarker(clickCoordinate, 24,
null, MapStandardMarker.HEXAGON);
        mapDisplay.addMapObject(marker);

        return true;
    }
```

Since the `CustomEventListener` is not interested in handling other events, an empty implementation is added for them all:

```
public boolean keyPressed(int keyCode, int gameAction)  {
    return false;
}
public boolean keyReleased(int keyCode, int gameAction)  {
    return false;
}
public boolean keyRepeated(int keyCode, int gameAction, int repeatCount)  {
    return false;
}
public boolean pointerDragged(int x, int y) {
    return false;
}
public boolean pointerReleased(int x, int y) {
    return false;
}
```

The CustomMapComponent

The `CustomMapComponent` class must build an instance of the `CustomEventListener` class defined above, and return it through its `getEventListener()` method. Since the `CustomEventListener` constructor requires a `MapFactory` argument, a similar constructor is defined for the `CustomMapComponent` class, so that the `MapFactory` can be externally provided.

(C) Copyright Nokia 2014. All rights reserved.

```
public class CustomMapComponent implements MapComponent
{
    CustomEventListener eventListener = null;

    public CustomMapComponent(MapFactory factory)
    {
        eventListener = new CustomEventListener(factory);
    }
    public EventListener getEventListener()
    {
        return eventListener;
    }
}
```

The `attach()` and `detach()` methods are used to notify the `CustomEventListener` of the newly set `MapDisplay`, thus appropriately setting its related variable.

```
public class CustomMapComponent implements MapComponent
{
    [...]

    public void attach(MapDisplay mapDisplay) {
        eventListener.setMapDisplay(mapDisplay);
    }
    public void detach(MapDisplay arg0) {
        eventListener.setMapDisplay(null);
    }
}
```

The `paint()` event is used to paint sample text over the map: more useful and complicated graphics can be added by using the Graphics object passed as an argument.

```
public class CustomMapComponent implements MapComponent
{
    [...]

    public void paint(Graphics g)
    {
        g.setColor(0xff0000);
        g.drawString("Custom component", 2, 2, Graphics.TOP | Graphics.LEFT);
    }
}
```

The other interface methods are trivially implemented as shown below.

```
public class CustomMapComponent implements MapComponent
{
    [...]

    public String getId()
    {
        return "MyCustomId";
```


(C) Copyright Nokia 2014. All rights reserved.

```
    }
    public String getVersion()
    {
        return "1.0.0";
    }
    public void mapUpdated(boolean zoomChanged)
    {
        // the map state is updated
    }
}
```

Using the CustomMapComponent

In order to use the `CustomEventListener` and `CustomMapComponent` object, it is necessary to add the component to the MIDlet main `MapDisplay` by using its `addMapComponent()` method. The base MIDlet's `startApp()` method can be modified as follows:

```
protected void startApp() throws MIDletStateChangeException
{
    [...]

    Display display = Display.getDisplay(this);
    MapCanvas mapCanvas = new MyMapCanvas(display);
    display.setCurrent(mapCanvas);

    mapCanvas.getMapDisplay().addMapComponent(new
CustomMapComponent(mapCanvas.getMapFactory()));
}
```

Resources

The sample MIDlet described in this article is available for download here: File:JavaMELocationAPI MapEventsMIDlet.zip

Source code of the `CustomMapComponent` and `CustomEventListener` implementation together with a sample `MapEventsMIDlet` can be downloaded from here: File:MapEventsMIDlet.zip

Summary

The MIDlet built in this article shows how map event handling can be implemented by using the MapComponent/EventListener paradigm. By using those interfaces, it is possible to add or drastically modify the functionality of a map, by appropriately handling the relevant key and pointer events.

(C) Copyright Nokia 2014. All rights reserved.

(C) Copyright Nokia 2014. All rights reserved.