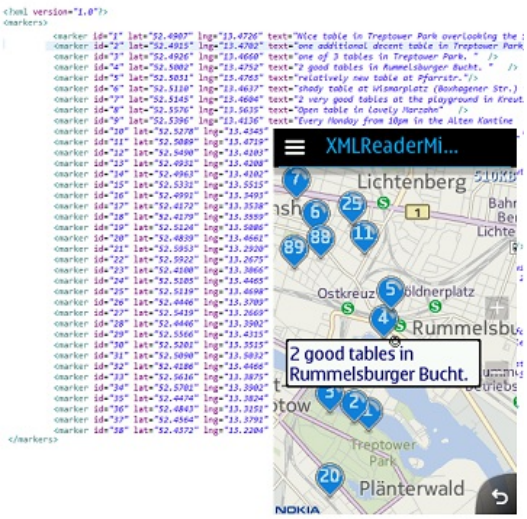
# Reading XML Map Marker data with the Maps API for Java ME

This article explains how to read in geographic XML data from a file (or URL) and transform it into `MapStandardMarkers` to be displayed on a map

**Tip:** The code behind this example has been integrated into the latest 1.3 release of the HERE Maps API for Java ME - see HERE Map Code Examples. The code has been designed to be fully backwards compatible for older phones . The API has been integrated as a plug-in into the Asha SDK 1.0.




-

## Introduction

In order to keep a Java ME app and relevant and up-to-date, it will be necessary to be able to retrieve some information from a hosted web service via a data connection. **XML** is a popular data format in use for many web service end points. This example shows how to process a specific **XML** data format including both geographic and non-geographic information and offers a design pattern to reduce the work required to process other arbitrary **XML** data files. A short discussion of the pros and cons of using the popular **KML** data format (which itself is a subset of **XML**) is also included.

The Code Example processes an XML file containing the locations of several public table tennis tables within the city of Berlin. The XML data file for this information is held in the following format:

```
<?xml version="1.0"?>
    <markers>
         <marker id="1" lat="52.4907" lng="13.4726" text="Nice table in Treptower Park overlooking the Spree"  />
         <marker id="2" lat="52.4915" lng="13.4702" text="one additional decent table in Treptower Park, overlooking the Spree"  />
...etc
    </markers>
```

## Creating an Asynchronous XML Parser

The crux of this example is the `DelegatingAsynchSAXParser` class. As the name suggests this is an asynchronous **SAX** parser which delegates the file processing to a helper class. Details of the code can be found below:

### Creating a simple SAXParser

One reason behind the popularity of the XML data format is that all devices which support JSR-172 come with a pre-installed **SAX** parser. **SAX** (Simple API for **XML**) is an event-based sequential access parser for reading **XML** files. It make sense to start the parser implementation using the SAX model. A simple skeleton is shown below:

```
public class DelegatingAsynchSAXParser extends org.xml.sax.helpers.DefaultHandler{
    public void startElement(String uri, String localName, String qName, org.xml.sax.Attributes attributes) throws org.xml.sax.SAXEx
            // SAX Parser has found the start of an XML element
    }
    public void endElement(String uri, String localName, String qName) throws SAXException {
        // SAX Parser has found the end of an XML element
    }
    public void characters(char[] ch, int start, int length) throws org.xml.sax.SAXException {
       // SAX Parser has found characters between two XML elements
    }
}
```

## Delegating the XML file processing

The '**X**' in **XML** stands for e**X**tensible. This means that provided a file follows the standard syntax for **XML**, the names of the elements within the file could be anything. In order to make this SAX Parser re-usable, the parser itself will not process the file, it will **delegate** the process instead. Firstly a "well-known" interface needs to be created:

```
public interface SAXParserDelegate {
     void onStartElement(String uri, String localName, String qName, org.xml.sax.Attributes attributes);
     void onEndElement(String uri, String localName, String qName);
     void onCharacters(char[] ch, int start, int length);
}
```

Then the job of actually processing the file can be **delegated** to a specific handler. The `DelegatingAsynchSAXParser` class is merely handling the SAXParser events.

```
public class DelegatingAsynchSAXParser extends org.xml.sax.helpers.DefaultHandler implements Runnable {

...
    private SAXParserDelegate delegate;
...
    public void startElement(String uri, String localName, String qName, org.xml.sax.Attributes attributes) throws org.xml.sax.SAXEx
         // SAX Parser has found the start of an XML element
        delegate.onStartElement(uri, localName, qName, attributes);
    }
    public void endElement(String uri, String localName, String qName) throws SAXException {
        // SAX Parser has found the end of an XML element
        delegate.onEndElement(uri, localName, qName);
    }
    public void characters(char[] ch, int start, int length) throws org.xml.sax.SAXException {
         // SAX Parser has found characters between two XML elements
        delegate.onCharacters(ch, start, length);
    }
}
```

## Making the XML processing Asynchronous

Since the length of the file to process is potentially very large, it may take time to process and leave the app unresponsive. It is therefore necessary to make the parser asynchronous. In other words, it needs to run on its own thread and make a callback when the data processing has been completed.

```
public class DelegatingAsynchSAXParser extends org.xml.sax.helpers.DefaultHandler implements Runnable {
...
    private AsynchSAXParserListener listener;
    private InputStream in;
...
public void parse(InputStream in, SAXParserDelegate delegate, AsynchSAXParserListener listener) {
        this.listener = listener;
        this.in = in;
        this.delegate = delegate;
        new Thread(this).start();
    }

    public void run() {
        try {
            javax.xml.parsers.SAXParser parser = javax.xml.parsers.SAXParserFactory.newInstance().newSAXParser();
            parser.parse(this.in, this);
            listener.onParseComplete();
        } catch (Throwable t) {
            listener.onParseError(t);
        }
    }
```

where the `AsynchSAXParserListener` interface represents the necessary callback functions for success and failure:

```
public interface AsynchSAXParserListener {
    void onParseComplete();
    void onParseError(Throwable error);
}
```

## How to call the AsynchSAXParser

The calling class needs to supply three things:

- An `InputStream` holding the **XML** data to process - this could either come from an `HttpRequest` to a web service or an on-board `resource`.
- A `SAXParserDelegate` to do the actual processing of the **XML**
- A `AsynchSAXParserListener` callback function to continue the thread of execution and display the processed data.

**Tip:** It would make sense to display an *indeterminate* progress bar whist the data is being loaded.

In order to display **map** data, the `XMLToMapMarkersDelegate` which implements the `SAXParserDelegate` interface is also supplied with a `MapFactory`. The details of `XMLToMapMarkersDelegate` are described below, but it can be seen that is supplies both a `MapComponent` and a `MapContainer` which can be added to the current `MapDisplay`. The call to `repaint()` is necessary to make sure that all the contents of the `MapContainer` are correctly displayed on the map.

```
public class XMLMapDataDemo extends MapCanvas implements CommandListener,
  AsynchSAXParserListener {

...
      private final XMLToMapMarkersDelegate mapDataProcessor = new XMLToMapMarkersDelegate(
   this.mapFactory);
...
      private void readMarkerXMLData() {
      DelegatingAsynchSAXParser parser = new DelegatingAsynchSAXParser();
            parser.parse(getClass().getResourceAsStream("/markers.xml"),  mapDataProcessor,this);
 }

      /** AsynchSAXParserListener callback function when XML parsing was successful.
        Display the processed data */
 public void onParseComplete() {
  map.addMapObject(mapDataProcessor.getContainer());
  map.addMapComponent(mapDataProcessor.getComponent());
  map.zoomTo(mapDataProcessor.getContainer().getBoundingBox(), false);
  repaint();
 }

       /** AsynchSAXParserListener callback function when XML parsing failed
        Display an error message */
 public void onParseError(Throwable error) {
  System.out.println(error);
  Alert alertView = new Alert("Map error: ", error.getMessage(), null, AlertType.ERROR);
  display.setCurrent(alertView);
 }
```

## Creating a SAX Parser Delegate

The actual processing of the **XML** will take place in a custom SAX Parser Delegate, the details of which will vary from implementation to implementation. Therefore a new version of this class will need to be written for each specific **XML** data file to be parsed, and the processing will depend on the details of the syntax of the **XML** input and the required output format. In the Code Example, two example delegates are given, one to process **XML** to be displayed on a `MapCanvas` and the other to display **XML** data as `Strings` to be displayed on a `Form`. For map data which is the main concern here, the **XML** data to process can be split into geographic elements and non geographic elements. The former are processed first (e.g. obtaining a latitude and longitude and creating a `MapStandardMarker`), and then the latter can be added to a custom `MapComponent` (associating textual data to a location) - the example used here is based on the marker tooltip component

For the **XML** of the following format:

```
<marker id="1" lat="52.4907" lng="13.4726" text="Nice table in Treptower Park overlooking the Spree"  />
```

It is possible to know that we are processing a **marker** based on the qualified name `qName` of the element, and then retrieve the data from the `attributes` as shown:

```
   public void onStartElement(String uri, String localName, String qName, org.xml.sax.Attributes attributes) {
         if ("marker".equals(qName)) {
             MapObject marker = factory.createStandardMarker(new GeoCoordinate(Double.parseDouble(attributes.getValue("lat")),
                     Double.parseDouble(attributes.getValue("lng")), Float.NaN), -1, attributes.getValue("id"), MapStandardMarker.BAL
             container.addMapObject(marker);
             tooltipComponent.setTooltip(marker, attributes.getValue("text"));
         }
     }
```

## Comparison between XML and KML data files.

**KML** is a standard **XML** notation for geographic applications. The Maps API for Java already contains a library to process KML files. Since **KML** is a well defined subset of **XML**, the code required to display **KML** data is much shorter than the equivalent proprietary **XML**. However designing your own data format can result in a less verbose data file and therefore reduce data traffic and speed up the application. Compare the following XML snippets:

**KML Syntax**

```
<Placemark>
    <description>Nice table in Treptower Park overlooking the Spree</description>
    <Point>
        <coordinates>13.4726,52.4907,0</coordinates>
    </Point>
</Placemark>
```

**Proprietary XML Syntax**

```
 <marker id="1" lat="52.4907" lng="13.4726" text="Nice table in Treptower Park overlooking the Spree"  />
```

However it may also be necessary to consider the extensibility, readability and maintainability of the data in which case **KML** would always be preferred. It should be possible to convert any proprietary **XML** format to standard **KML** using an XSLT transform on the web service endpoint.

## Summary

The Code Example provided offers an *extensible* method for processing arbitrary **XML** data. The example includes a pair of sample delegates for processing both geographic and non-geographic data sets. In the example, the data is loaded directly from a resource file, but the data parsing method could be used against any `InputStream` (e.g. from a web service using an `HttpConnection`).