8 Chapter

MapJ API

This chapter is the first of six chapters that cover the MapJ API, beginning with the MapJ object and instructions on programming your first map.

MapJ Object

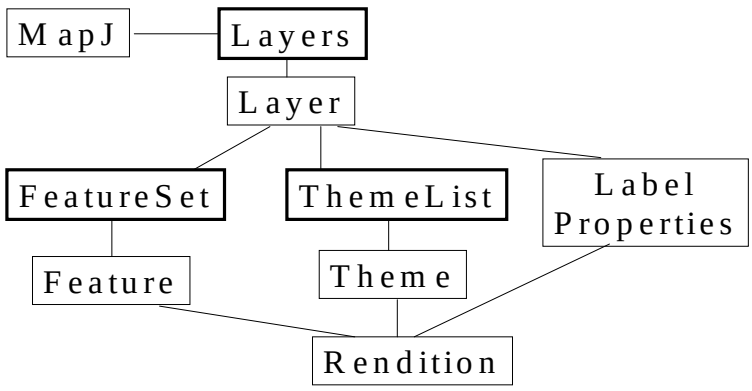
MapJ is a small light-weight component that provides an interface for the creation of maps by MapXtremeServlet or by itself. MapJ can make two types of requests: a request for data in vector form, called features, or a request for a map image file. MapJ's job is to maintain the state of the map, including keeping track of the layers, coordinate system, distance units and map bounds.

MapJ objects can be configured to work with different types of Renderers and DataProviders. In the most typical configuration MapJ is a client of MapXtremeServlet. MapJ sends requests to a MapXtremeServlet instance and as part of the request provides the servlet with its current state. MapJ obtains map images and data from the servlet.

MapJ can also work stand-alone to directly obtain map data and produce map images. A strength of MapXtreme's component-based design is that MapJ can be configured with other variations. For instance, MapJ can be configured to access map data via one or more instances of MapXtremeServlet, but still be responsible for displaying the map image. See Chapter 4: PlanningYour Application, for more information on configuration options.

The MapXtreme Java object model poster included with the product shows that almost every object, property, and method is derived from the MapJ object. Every method shown underneath the MapJ object will contribute to building the overall MapJ object. Primarily, the Data Provider, Layers, and Feature objects define each MapJ object. The other objects on the MapXtreme diagram contribute to the creation and rendering of the MapJ object; these are the Data Provider objects and the Renderer objects.

The diagram below shows the relationship among the objects that contribute to making a map. A MapJ object is made up of a Layers collection, which consists of individual layers. Related objects include themes, renditions, and label properties.



This chapter focuses on the steps to build a map and carry out map-level manipulations. In the following chapters, we will go into more detail about working with individual layers, features, themes, renditions, and labeling.

To begin creating a mapping application, you must start by creating a MapJ object. The following section describes the process necessary to generate a map.

Creating Your First Map

The following outlines the general process of creating a map using the MapJ API. In this casee, MapJ is communicating its request to MapXtremeServlet.

1. Create a MapJ Object.
2. Load map data.
3. Set map device bounds.
4. Render the map to an image file.

1. Initialize a MapJ Object

Before you can use a MapJ object or its methods in your application, you must first initialize MapJ. This is done simply with the following Java code:

```
myMap = new MapJ();
```

2. Load Map Data

Once the MapJ object has been created, you must load map data.You can load a geoset or Map Definition.

A geoset is a collection of MapInfo .tab format map layers and their settings, similar to a workspace. Map layers are saved to a geoset with the extension .gst. The MapXtreme Java sample data contains a number of geosets covering world geography.

A Map Definition describes a collection of map layers and their settings, and can be either data stored in a file or as a record in a remote database. Information in a Map Definition (.mdf) is stored as XML.

Map Definitions are strongly preferred over geosets because you can access a wide variety of data providers. Geosets are specific to .tab files. You can, however, save a geoset as a Map Definition, using the Map Definition Manager (see Chapter 14: Managing Your Data).

When map data is loaded, it clears all loaded layers, and then loads the new data. There is no default map definition setting in MapXtreme Java Edition. Therefore, as part of your initialization process, you must set a default map definition.

The following is an example of loading a geoset file:

```
myMap.loadGeoset(geosetName, dataDir, servletURL);
```

where

geosetName is the full path to the geoset

dataDir is the location on the server machine of the .tab files referenced in the geoset (may not be the same machine at MapXtremeServlet)

servletURL is the path to the MapXtremeServlet when MapJ is using a remote DataProviderRef (if using LocalDataProviderRef parameter is NULL).

For example:

```
myMap.loadGeoset("c:\\mapxtreme\\maps\\world.gst",
   "c:\\mapxtreme\\maps", "http://localhost:8080/
   mapxtreme/servlet/mapxtreme");
```

To load a map definition, you must first create a MapDefContainer which is an abstraction that represents where the map definition is stored.

Create a FileMapDefContainer if the map definition is stored in a file:

```
MapDefContainer mdc = new FileMapDefContainer(dir)
```

where `dir` = full path to the map definition file

For example:

```
MapDefContainer mdc = new
   FileMapDefContainer("c:\\mapxtreme\maps\")
```

Create a JDBCMapDefContainer if the map definition is stored in a record in an RDBMS:

```
MapDefContainer mdc = new JDBCMapDefContainer(driver,
   url, user, password)
```

where `drive` , `url`, `user` and `password` are database connection parameters.

The following example creates a Oracle Spatial MapDefContainer where the map definition is stored in a table in the database:

```
OraSoMapDefContainer mdc = new
   OraSoMapDefContainer("oracle.jdbc.driver.OracleDriver"
   , "jdbc:oracle:thin:@machinename:1521:dbSid",
   "username", "password", "tableName", "Name",
   "Map_Definition");
```

To load the map definition:

```
myMap.loadMapDefinition(mapDefContainer, name)
```

where **mapDefContainer** = the above defined container class

name = the map to load from the container (the name used in the saveMapDefinition command).

For example:

```
myMap.loadMapDefinition(mdc, "Asia");
```

3. Set Map Device Bounds

Set the size of the rendered map image using **MapJ.setDeviceBounds()**. This is set before the map is rendered. The Device Bounds set the dimensions, in pixels, of the image that will be returned from the Renderer. For example, you may want to return a map that is 800x600. The default image size is 640x480.

To set the Device Bounds, use the setDeviceBounds method of the MapJ object.

```
myMap.setDeviceBounds(new DoubleRect(0, 0, 800, 600));
```

4. Render the Map

To render the map you must instantiate a renderer object, assign it to MapJ, render the map, and dispose of the renderer.The following example uses a MapXtremeImageRenderer and renders the image as a GIF file.

Instantiate a renderer object:

```
MapXtremeImageRenderer renderer = new
   MapXtremeImageRenderer(mapxtremeServletUrl);
```

where **mapxtremeServletUrl** = URL to MapXtremeServlet, such as http://localhost:8080/mapxtreme/servlet/mapxtreme.

Assign this object to MapJ object:

```
myMap.render(renderer);
```

Render the map to file:

```
renderer.toFile("myMap.gif");
```

Dispose of the renderer:

```
renderer.dispose();
```

The MapXtremeImageRenderer returns an image of all of the specified layers. When a MapXtremeImageRenderer's render method is called, a request is made to MapXtremeServlet, which then produces an image on the server. The image is returned to the user only when a toFile, toStream, or toImage method is invoked.

Alternatively, if you have configured MapJ to work as stand-alone to directly obtain map data and produce images, instead of MapXtremeImageRenderer, you would use LocalRenderer to render the image locally.

Interaction can occur between the MapJ client and the MapXtremeServlet without using the Renderer. However, the Renderer is the only way that a map image will be returned to the user. You could create and initialize a MapJ object and execute several methods that manipulate the object or query a map, but in order to see the current map, you must use the **render** method. This is useful if you would rather create the map in several steps, and then display it.

Map Rendering Considerations

The above example renders a map as a GIF image, one of a number of supported raster output formats. Output formats of raster images are specified in the MapXtremeImageRenderer constructor by MIME type. MIME is a format standard for non-textual data such as images. The following guidelines can help you decide which type is appropriate for your needs:

- image/jpeg – JPEG – good for layers with more than 256 colors.
- image/gif – GIF – good for vector layers or layers with up to 256 colors.
- image/png – PNG – a replacement for GIF format; more than 256 colors.

For example, to output a JPEG, use the constructor of MapXtremeImageRenderer that takes the MIME type "image/jpeg," as shown here:

```
MapXtremeImageRenderer(URL, "image/jpeg");
```

Note: The MapXtremeImageRenderer constructor on the previous page did not take a MIME type for image/gif as GIF is the default output format.

When using raster files, we suggest you use JPEG output. GIF output is limited to a maximum of 256 colors and raster files generally have at least 256 RGB or gray scale colors. Adding a vector layer may bring the total number of colors to greater than 256. If this happens, the colors must be reduced, which is a time intensive operation. It's faster saving to JPEG or PNG.

Setting the Quality of a JPEG Image

You can control the quality of the JPEG output by setting the parameter **jpegQuality** in your servlet container. For example, in Tomcat, edit the web.xml file under the \mapxtreme\WEB-inf directory to contain a value for JPEG quality.

```
<init-param>
   <param-name>
      jpegQuality
   </param-name>
   <param-value>
      85
   </param-value>
</init-param>
```

The jpegQuality value ranges from 0-100, with the default at 75. A lower number means the image quality is reduced, but results in a smaller image size.

Controlling the Map View

Once your map is displayed, you will likely want to change its view to see map detail closer up, or to gain a wider view.

MapJ has several methods for controlling the map view: setZoom(), setCenter(), and setZoomAndCenter().

Setting the Zoom Level

The zoom level is the distance across the map. You may change the zoom level to any distance. The units used will be the current distance units. The zoom level is first set when the geoset or map definition file is loaded.To change the zoom level of the map, use the **setZoom** method. The following example sets the zoom level:

```
// Assuming that the current distance units are
   kilometers,

// this command will set the map zoom to 500 kilometers.

myMap.setZoom(500);
```

Recentering the Map

Part of controlling the map view is setting the center of the map. You may want to center on a found location or a particular coordinate. The **setCenter** method accomplishes this. You must pass a DoublePoint to the **setCenter** method. A DoublePoint is defined by a pair of XY double precision points.

The point location, if it is the result of a user clicking on the map at a certain location, is typically returned in pixels. MapJ requires the location to be in numeric coordinates, so a conversion method, **transformScreenToNumeric**, is necessary.

The following example creates a screen point, converts it to a "real world" point and sets the center of the map. DoublePoint is a point defined by double precision coordinates (x,y).

```
// Create the screen point
screenpoint = new DoublePoint(event.getX(),event.getY());
// Create the real world point
worldpoint = myMap.transformScreenToNumeric(screenpoint);
// Set the center of the map
myMap.setCenter(worldpoint);
```

Setting the Map Bounds

Use **setBounds** to set the bounding rectangle for the map. The method takes a DoubleRect, which is defined by coordinates that represent either the two opposing corners, or its center point, width, and height. Both ways are illustrated below.

This example uses opposing corners to set the bounds to the entire world:

```
DoubleRect bounds = new DoubleRect(-180,-90,180,90);
```

This example uses opposing corners to set the bounds to a zoomed-in map area:

```
DoubleRect bounds = new DoubleRect(-1.969272, 50.560410,
   1.443363, 52.315529);
```

This example uses the center point, width, and height to set the bounds for the world:

```
DoubleRect bounds = new DoubleRect(new
   DoublePoint(0,0),360,180);

myMap.setBounds(bounds);
```

Setting the Coordinate System

Coordinate system data is stored in a projection file called mapinfow.prj that can be found in the server directory where MapXtreme Java is installed. The PRJ file is the same used in MapInfo Professional and lists hundreds of supported coordinate systems and the parameters that define them.

Coordinate systems are set through the MapJ method **setDisplayCoordSys.**

```
String csProj = new String("\"Azimuthal Equidistant
   (North Pole)\", 5, 62, 7, 0, 90, 90");

CoordSys ts = csf.createFromPRJ(csProj);

myMap.setDisplayCoordSys(ts);
```

Additionally, you can set the coordinate system using **createFromMapBasic** to read MapBasic strings and through some pre-defined constants.

For more information see the CoordSys class in the HTML API Reference.

Setting the Map Distance Units

Units are set through the MapJ method **setDistanceUnits**

```
distUnit = LinearUnit.INDEX_KILOMETER;

myMap.setDistanceUnits(distUnit);
```

Adding a Layer

One of the more used methods of the MapJ API is the **add** method of the Layers object which allows you to bring additional data into your map. While the add method is simple enough to call, there are a number of steps that must precede it to describe what data to add, where to get it from and how to get it. Data can come from local or remote data sources, in the form of files or records from a database. In order to manage this operation, MapXtreme Java uses Data Providers, which are fully explained in Chapter 9, Mapping in Layers.

To access the Layers collection, use MapJ's **getLayers** method.

Beyond the Basic Map

Now that you've had a chance to display a basic map and manipulate the view, you will want create a more sophisticated map that represents the information you want to impart to your viewers. MapXtreme Java provides the API that allows you to control every aspect of the map. This section introduces you to some of them.

Features

Anyone who has worked with databases is familiar with the idea of a record. A record is a set of related columns of information. For example, a database of customers will have a record for each customer that includes columns for name, address, interest, etc. A feature is simply a record that combines tabular data and geometric information.

For example, the fileWorld.tab from the MapXtreme sample data is a MapInfo format database. For each country, there is a record. Each record includes several columns of tabular data as well as a reference to the geometric information that describes the shape and location of each country. This allows it to be displayed on the map. The tabular data is referred to as attribute data, and the geometric data is referred to as the geometry. These two types of data make a feature.

Features are not directly connected to the MapJ object, but are important for several reasons. As explained earlier, MapJ is the base for all of the map functions in your program. It sits at the top of the object diagram.

A Feature object sits at the lowest level of your program and deals with specific information. It is one of the most specific objects in the object model and relates to record level information. It is at the Feature level that the graphic objects can be given

different display characteristics. The characteristics that specify the appearance of a graphic object are set by the Rendition object.

Information on Feature objects can be found in the Chapter 11: Features and Searches.

Renditions

Every feature has a rendition associated with it that describes how it is to look on a map. Rendition properties can be grouped into three general categories: fill, stroke, and symbol. The fill properties control how a region is filled. The stroke properties control how a line (either a line geometry or the edge of a region) will be drawn. The symbol properties control how symbols are drawn for either point geometries, line markers, or symbol fills.

The portion of the MapJ API that controls renditions is the Rendition class. The combinations of renditions that you can achieve are practically unlimited due to the variety of methods available. Renditions can be assigned to Features, Layers, Labels, and Themes and can be used to override symbology. For more on renditions, see Chapter 12: Labeling and Renditions.

Themes

Whether layers are added by the Layers' **add** method, MapJ's **loadGeoset** method or **loadMapDefinition** method, each layer will have its own characteristics such as a line's color, its width, etc. These characteristics are based on the information in the geoset or in the data source. Usually these settings are consistent for an entire layer. For example, if you load World.tab from the sample data, each country displays with a solid, green fill pattern. Every feature in the layer appears the same way.

Themes allow you to programmatically change the appearance of some or all of the features in a layer based on some criteria. For example, if you wanted to change the color of all of the world countries that have a population over fifty million, you could accomplish that with a theme. There are four Theme classes available:

- OverrideTheme – for changing the rendition of an entire layer
- RangedTheme – for grouping data into ranges and shading based on range value
- IndividualValueTheme – for shading groups of features which share a specific attribute value
- SelectionTheme – applies a rendition to a user-defined list of selected features

Note: IDSelectionTheme, a type of theme available in previous releases, has been deprecated. The IndividualValueTheme and SelectionTheme provide all the IDSelectionTheme functionality and more.

The features in each theme all have a rendition associated with them. The Rendition object encapsulates the style properties for both graphic and text displays.

For more on theme mapping, see Chapter 13.