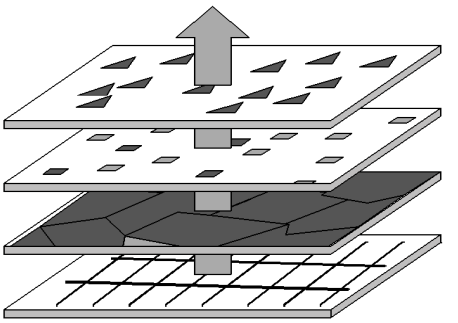
Chapter 9

Mapping In Layers

This chapter presents the relationship between tables and maps and how they are layered to create the level of detail you want.

Maps as Layers

You have already been introduced to the concept of computer maps as a collection of layers. Each table that contains graphic objects can be displayed as a layer in a map image. For example, you can display a layer of customers, a layer of streets, and a layer of county boundaries.



Think of these layers as transparencies where each layer contains a different part of the map. The layers are stacked one on top of the other and allow you to see all aspects of the map at once. For example, one layer may contain country boundaries, a second layer may have symbols that represent capitals, and a third might consist of highways. Laying these transparencies on top of each other builds a map.

Layers are made up of geographic features and associated data. For example, a layer of country boundaries has regions that define each country's boundary and it might have attributes that represent the population of each country, literacy rate, or average household income. By creating a map of layers that have information attached, you can go beyond the pretty map and query the layer for information that you can then analyze and display. That kind of map is much more effective in showing relationships among map data.

This chapter will focus on how to handle layers, such as defining a layer, adding a new layer to a map, and the types of layers you can create and display.

The Layers Collection: Building Blocks of Your Ma

The Layers collection is accessible from MapJ and contains Layer objects. These Layer objects, which are built from tables, make up your map. Each layer contains different map features, such as regions, points, or lines. The Layers collection has methods used to perform operations such as adding and removing Layer objects from the collection. Layer objects have search methods that allow you to locate specific information on a layer.

The Layer object represents data made up of map features that usually have a predominant feature type, such as regions, lines, or symbols. Typically, a Layer object corresponds to the geographic objects from one table. Each of the Layer objects in a Layer collection behave independently of each other. Their styles may be changed, zoom layering altered, etc., on an individual basis, without affecting any of the other layers.

The Layer object makes use of several related classes such as ThemeList, LabelProperties, FeatureSets, ColumnStatistics, and TableInfo. Besides methods for accessing these objects, the Layer object also has search methods that allow you to locate specific information on a layer. Through the Layer object, you can take advantage of most of the MapXtreme functionality.

How to Build a Layers Collection

To build your map, you begin by adding layers to a Layers collection. In the previous chapter we walked through the code that loaded the map data using **MapJ.loadMapDefinition** and **MapJ.loadGeoset**. The Layers collection is defined by the layers of this map definition. When a map definition is loaded, the numeric and display coordinate systems are updated and any previous layers are removed.

Once you have created a Layers collection, you can add more layers. When you use the **Layers.addMapDefinition** method, the layers are added to the current map and the existing coordinate system settings are maintained. Layers can also be added individually with the **Layers.add** method. The add method puts the Layer at the end of the collection. Use insert to control the position.

How Layers Are Drawn

Map layers in a Layers collection display in increasing index order: Layer(0) is the top layer, Layer(1) is the layer underneath Layer(0), etc., with the bottom layer drawn first and the top layer drawn last. It is important to order your layers correctly.

For example, you have a layer of customer points and a layer of census tracts. If the layers are incorrectly ordered in the Layers collection, MapXtreme will draw the customer points first and then display the census tract layer second. Your customer points would be obscured by the census tract layer. In the section "Methods of the Layers Collection," we provide a code example for positioning layers.

Defining a Layer with a Data Provider

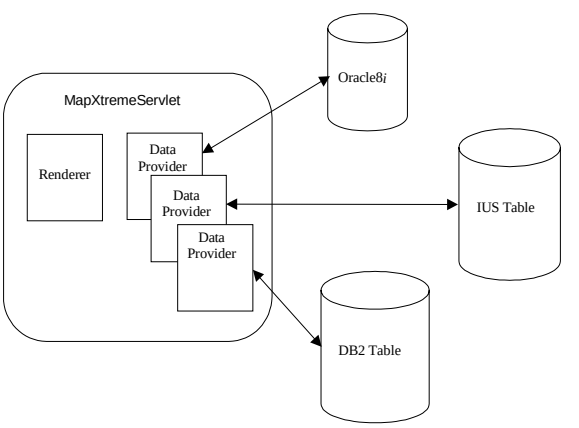
To add a layer to a Layers collection, you must first define it. The key to defining a layer is the Data Provider

Each layer has an internal object, a Data Provider, that is responsible for data access. Data Providers are not created directly by users, but their description defines a layer. The following three interfaces are used to describe a Data Provider (and thus a layer):

- TableDescHelper – describes the data
- DataProviderHelper – defines the source of the data
- DataProviderRef – describes how to get the data

MapXtreme Java has a number of DataProviders that allows you to create map layers for the following data sources:

- MapInfo tab format (.tab)
- Oracle8*i* with Spatial Option
- Oracle 7.x and 8.x with SpatialWare
- Informix Universal Server SpatialWare DataBlade
- DB2 SpatialWare Extender
- JDBC compatible tables containing longitude and latitude columns
- GeoTIFF and MIGrid Raster
- ESRI Shapefiles
- Annotation [1]



1. Not a typical Data Provider as the information is not stored in the database, but held in memory. See page 146 for more information.

TableDescHelpers

The TableDescHelper is an interface that helps to describe the data that you are accessing. There is aTableDescHelper for each different type of data source that MapXtreme can access. Each one has constructor parameters specific to the data source.

For example, the TABTableDescHelper that is used to describe a MapInfo table such as world.tab needs only the table name to describe it. The OraSoTableDescHelper, used to describe Oracle8*i* data, is defined by either a table name or SQL query. Code examples are presented later in this chapter. Further details on eachTableDescHelper can be found in the HTML Reference.

DataProviderHelpers

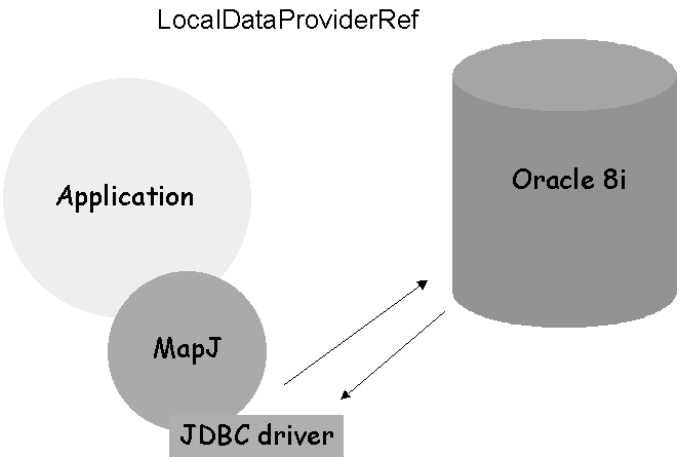
DataProviderHelpers define the data source. In the case of MapInfo TAB files, the directory containing a TAB file is the data source for that file. Therefore, the DataProviderHelper for tab files,TABDataProviderHelper, takes a directory as its only parameter.

Consider the example where a map is consists of several tables, all of which are stored within an instance of Oracle8*i* database. This database is the data source for each of the tables. The OraSoDataProviderHelper takes parameters that describe the data source, e.g., server host name, server port number, user name, and password. The same DataProviderHelper can be used for different tables from the same data source. Further details on each DataProviderHelper can be found in the HTML Reference.

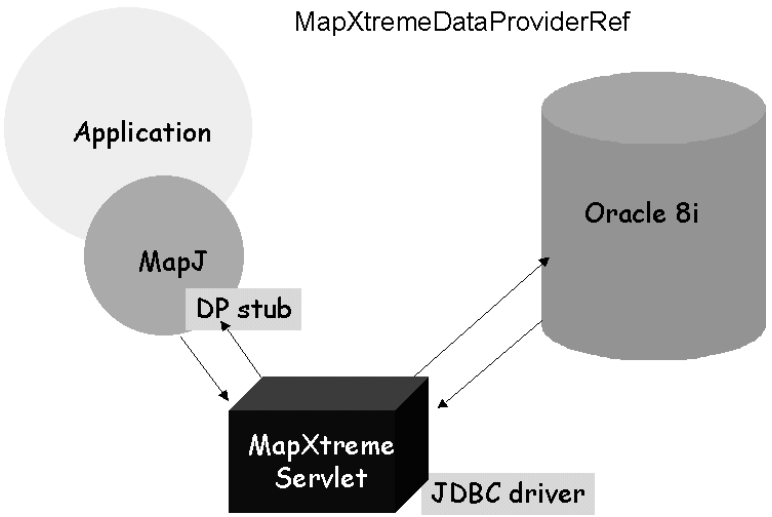
DataProviderRef

Data Providers have a built-in remoting capability. The DataProviderRef describes who is responsible for accessing the data source. There are two possibilities: 1) the application (process) that contains MapJ and this Layer can directly access the data source, or 2) the application can defer to an instance of MapXtremeServlet to access the data source, and then have MapXtremeServlet transport the data back to the application.

Using a **LocalDataProviderRef** signifies that the application will directly communicate with the data source. This means that any resources needed to access the data source must be available to the application. For instance, a JDBC driver must be on the classpath of the application in order for it to directly access data within an RDBMS.



A **MapXtremeDataProviderRef** is used when one wants to defer the data access to MapXtremeServlet. This is the typical case when using three-tier deployments. In this case only a "stub" data provider will exist in the client and the real data provider will be created by MapXtremeServlet, which then accesses data from the data source. With this deployment the resources needed to access the data source are only put in the middle tier. This avoids the need to have JDBC drivers deployed in the client, but still allows MapXtremeServlet to access data within an RDBMS.



When is Data Accessed?

The aforementioned interfaces are used to define a DataProvider. When a Layer is first defined and created, no data access occurs. DataProviders only access their underlying data source in response to a specific request such as a call to Layer's getTableInfo, a Layer's search method, or a render request.

Caveats for Defining JDBC Layers

Each JDBC data source has several corresponding DataProviderHelper constructors. While some constructors are easier to use than others, we strongly recommend setting up your JDBC Layers to use connection pooling. This requires using the most generic form of the DataProviderHelper constructor. Connection pooling is discussed in Chapter 10.

The TableDescHelper objects for JDBC Layers all share some common parameters. When a JDBC Layer is defined as a table, rather than as a SQL statement, several of these parameters are optional. These include the spatial column, coordinate system of the geometry, dimension of the geometry, and the table level Rendition. The optional parameters, if not present, are searched for within the MapInfo.MAPINFO_MAPCATALOG. If you know the information for these values we strongly recommend that you supply it when constructing the object. This will eliminate extra queries to the MAPCATALOG and increase the performance of your application.

When the primary key column(s) is not specified, MapXtreme interrogates the table schema definition in the database to find a column or columns suitable for use as a key. It will attempt to find the column(s) formally defined as the table primary key or, if not present, it will choose a column that the database guarantees is unique to each row, that is not a pseudo column and whose type is character or numeric. Identifying the primary key column(s) in the TableDescHelper eliminates this overhead and guarantees expected behavior when the primary key is used (e.g. in the Selection class). Additionally, the primary key column(s) also receive special treatment when adding features to a JDBC layer (see page 183), so specifying it in this case is also important.

JDBC Layers now support per-Feature Renditions where each Feature stored in the database can be given a possibly unique Rendition. This Rendition is stored as a column called RENDITIONCOLUMN. The MapInfo EasyLoader v 6.0 supports this column, however it does not include the column when it creates or updates the MAPCATALOG. Future versions of the MapInfo EasyLoader will include this column. Check the Web site for updates.

See Appendices B and C for more on EasyLoader and the MAPCATALOG.

Adding a Layer to a MapJ Layers Collection

This is the general procedure to add a layer:

1. Create the TableDescHelper
2. Create the DataProviderHelper
3. Create the DataProviderRef (requires the DataProviderHelper as input)
4. Use Layers.**add** method (takes DataProviderRef and TableDescHelper as input). This puts the layer at the bottom of the collection, by default. You can also use **Layers.insert**.

TableDescHelper and DataProviderHelper implementations exist for each type of data source MapXtreme Java supports. The following table is a summary. See the HTML Reference (Javadocs) for more information.

Data Source	TableDescHelper	DataProviderHelper
MapInfo Tables	TABTableDescHelper	TABDataProviderHelper
Oracle8*i* with Spatial Option	OraSoTableDescHelper	OraSoDataProviderHelper
SpatialWare for Oracle 7.3.x or 8.0.x	OraSpwTableDescHelper	OraSpwDataProviderHelper
Informix Universal Server SpatialWare DataBlade	IusSpwTableDescHelper	IusSpwDataProviderHelper
DB2 SpatialWare Extender	Db2SpwTableDescHelper	Db2SpwDataProviderHelper
JDBC compatible tables containing longitude and latitude columns	XYTableDescHelper	XYDataProviderHelper
Annotation Layers	AnnotationTableDescHelper	AnnotationDataProviderHelper
GeoTIFF Raster	GeoTIFFTableDescHelper	GeoTIFFDataProviderHelper
ESRI Shapefiles	ShapeTableDescHelper	ShapeDataProviderHelper

Code examples for TAB, Oracle8*i*, and JDBC compatiable data sources follow. Others are provided in the HTML Reference.

TAB Data Provider Example

The following is an example of creating a TABDataProvider and assigning the layer to MapJ.

```
// specify the url to the MapXtreme servlet which
   remotely

// connects us to the map engine

String MapXtremeURL = "http://localhost:8080/mapxtreme/
   servlet/mapxtreme";

// create the tab Table Desc

TABTableDescHelper TabDesc = new TABTableDescHelper(new
   File("mytab.tab").getName());

// create the tab Data Provider

TABDataProviderHelper DPHelper = new
   TABDataProviderHelper("d:\\maps");

// Create the Remote Dataprovider needed to access the
   Data

MapXtremeDataProviderRef MXDPRef = new
   MapXtremeDataProviderRef(DPHelper, MapXtremeURL);

// assign it to MapJ

map.getLayers().add(MXDPRef, TabDesc, "tabLayer");
```

Oracle8*i* Data Provider Example

The following is an example of creating a Oracle8*i*DataProvider and assigning the layer to MapJ. Be sure that the JDBC driver is in your classpath.

```
// specify the url to the MapXtreme servlet which
   remotely connects us to the map engine

String mapXtremeURL = "http://localhost:8080/mapxtreme/
   servlet/mapxtreme";

// Create the Remote Dataprovider needed to access the
   Data

   // Using pooled connections (Recommended). The
   connection url must be of the form
   “jdbc:mipool:resource_name”.

   OraSoDataProviderHelper oraDPH = new
   OraSoDataProviderHelper("jdbc:mipool:ProjectMaps",
   null, null);

   // Using Database specific DataProviderHelper

   OraSoDataProviderHelper oraSoDPHelper=new
   OraSoDataProviderHelper("databasename",1000, "DBsid",
   "tester", "tester", DriverType.thin);

// Create a String array with the name(s) of the column(s)
   to use as a unique key for records in the TableName

String[] idColumn = {"S_MEMBER"};

// Now create a Table Desc helper

   // This code uses the Construtor required when using a
   tablename

   OraSoTableDescHelper oraSoTDHelper = new
   OraSoTableDescHelper("STATES", false, idColumn,
   "S_GEOMETRY",
   null,RenditionType.none,CoordSys.longLatWGS84, 2,
   "TESTER");

   // This code uses the Construtor required when using a
   Query

   OraSoTableDescHelper oraSoTDHelper = new
   OraSoTableDescHelper("Select S_MEMBER, S_GEOMETRY,
   POP_1990 From STATES Where STATE = 'NY'", idColumn,
```

```
    "S_GEOMETRY",
    null,RenditionType.none,CoordSys.longLatWGS84, 2);

// Create the Remote Dataprovider needed to access the
    Data

MapXtremeDataProviderRef mxtDPRef = new
    MapXtremeDataProviderRef(oraSoDPHelper, mapXtremeURL);

//assign it to MapJ - note getLayers()

m_myMap.getLayers().add(mxtDPRef, oraSoTDHelper,
    "OraSoLayer");
```

XY Data Provider Example

This code sample creates a Data Provider for a JDBC data source where the spatial information is stored in X, Y columns. Be sure your JDBC driver is in your classpath.

```
// specify the url to the MapXtreme servlet which
   remotely

// connects us to the map engine

String MapXtremeURL = "http://localhost:8080/mapxtreme/
   servlet/mapxtreme";

// create the XY data provider

XYDataProviderHelper XYDPH = new
   XYDataProviderHelper("sun.jdbc.odbc.JdbcOdbcDriver","j
   dbc:odbc:Test","tester", "tester");

// Create a String array with the name(s) of the column(s)
   to

// use as a unique key for records in the TableName

String[] idColumn = {"rowid"};

// Now create a Table Desc helper

   // This code uses the Constructor required when using
   a tablename

   XYTableDescHelper XYTDH = new
   XYTableDescHelper("myTable", "tester", false,
   "longitude","latitude", renditionColumn,
   RenditionType, idColumn, CoordSys.longLatWGS84);

   // This code uses the Constructor required when using
   a Query

   XYTableDescHelper XYTDH = new
   XYTableDescHelper("select longitude, latitude, rowid
   from myTable", idColumn, "longitude","latitude",
   renditionColumn, RenditionType,
   CoordSys.longLatWGS84);

// Create the Remote Dataprovider needed to access the
   Data

MapXtremeDataProviderRef MXDPR = new
   MapXtremeDataProviderRef(XYDPH, MapXtremeURL);

//assign it to MapJ - note getLayers()

mapJ.getLayers().add(MXDPR, XYTDH, "xyLayer");
```

Annotation Layers

Annotation layers are special map layers that contain features which are used to mark or place emphasis on certain areas of the map. For example, to select and highlight features within a certain radius of a point, use **Layer.searchWithinRadius** method which returns a circular feature at the point.To display the search radius use the **createCircularRegion** method of the FeatureFactory. Once the feature has been created, use the **addFeature** method to add the new feature to the Annotation layer. You can create the Annotation layer before or after the search.

You may have more than one Annotation layer. The table for an annotation layer resides in memory. It is created using the AnnotationDataProviderHelper, the Annotation TableDescHelper and a LocalDataProviderRef. Once created, it can be treated like any other layer. Here's an example of creating an Annotation Data Provider and adding it to the MapJ Layers collection:

```
// create the annotation table desc helper

AnnotationTableDescHelper annTDHelper = new
   AnnotationTableDescHelper("annLayer");

// create the annotation data provider

AnnotationDataProviderHelper annDPHelper = new
   AnnotationDataProviderHelper();

// An Annotation layer requires the use of local memory
   space, so we create a Local DataProvider Ref

LocalDataProviderRef localDPRef = new
   LocalDataProviderRef(annDPHelper);

//assign it to MapJ - note getLayers()

mapJ.getLayers().add(localDPRef, annTDHelper,
   "AnnLayer");
```

Methods of the Layers Collection

Now that you've added some layers, you will likely need to make some changes to the Layers collection. This section describes several methods to help. In your applications you will be frequently referencing objects and methods through the Layers collection.

Get the Name of Layers in a Collection

This example tells you the number of items, in this case the number of layers, in a collection. This is used if you want to cycle through each item in the collection, for example, getting the names of each item:

```
// The following example loops through all the layers and
   gets each layer's name

Layers layers = myMap.getLayers();

Layer layer;

String layerName;

for (int i=0; i < layers.size(); i++)

{

layer = layers.elementAt(i);

layerName = layer.getName();

}
```

Get a Layer from the Collection

The **getLayer** method gets a specific Layer object from the collection. The **getLayer** method returns one of the layers as an object. You can reference layers by their names, such as Highways or Cities. You may also reference a layer by its position. The **elementAt** method returns the layer at a given position in the collection, such as 0, 1, 2, and so on. The index is zero-based. The following examples demonstrate both uses:

```
Layer myLayer;

myLayer = myMap.getLayers().getLayer("highways");

myLayer = myMap.getLayers().elementAt(5); //gets the 6th
   layer
```

Insert a Layer

The **insert** method adds a layer to the Layers collection given DataProvider information and the position at which to place the layer. Similar to adding a layer, when inserting a layer you must provide a DataProviderRef and TableDescHelper. Any layers in the collection that come after the inserted layer are shifted down one position.

```
// inserting a layer at position 5

layers.insert(dataProviderRef, tableDescHelper, 5,
   "newLayer");
```

Move a Layer

The **move** method repositions a layer in the Layers collection. The first parameter is *From* position (the top layer = 0) and the second parameter is the *To* position.

```
// moving a layer from the bottom to the top

layers.move(layers.size() - 1, 0);
```

Remove a Layer

The **remove** method removes a specified layer from the map.

```
//removing a layer by position (top layer)

layers.remove(0);


//removing a layer by name

layers.remove("highways");
```

Remove All Layers

The **removeAll** method removes all layers from the map.

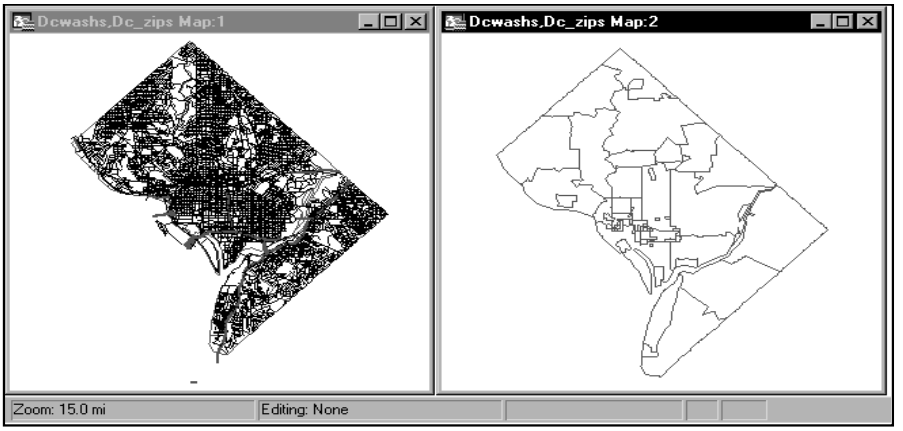
```
//removing all layers

layers.removeAll();
```

The MapJ HTML Reference has a complete listing of Layers collection methods and properties.

Zoom Layering

Sometimes you want a map layer to display only at certain zoom levels. Zoom layering controls the display of a map layer only when the map's zoom level falls within a preset distance. You can set a different zoom layering level for each layer.

For example, if your map includes a street layer, you may find that the streets become illegible when the user zooms out too far. Using Zoom Layering, set up your map so that MapXtreme does not display the streets whenever the user zooms out beyond a certain distance, for example, five miles.



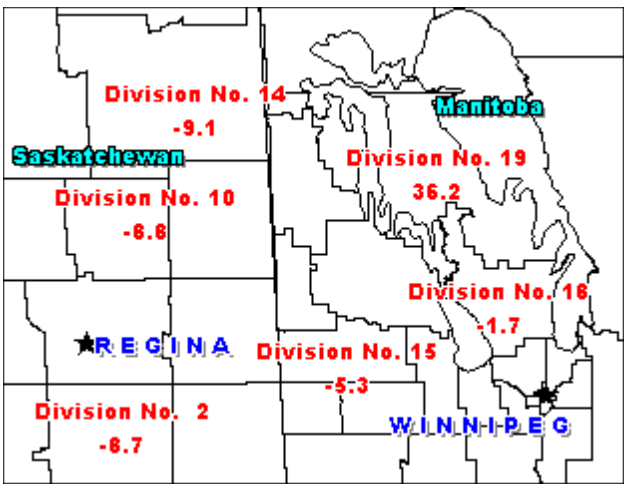
The following sample code sets up Zoom Layering by modifying the Layer object's properties so that the layer only displays if the map's zoom is between 10 and 30 km.

```
// set layer for zoom layering from 10 to 30 kilometers
layer.setZoomLayer(true);
layer.setZoomMin(10.0, LinearUnit.kilometer);
layer.setZoomMax(30.0, LinearUnit.kilometer);
```

You can set a different zoom level for every layer in your map. For example, you have a layer of streets, a layer of county boundaries, and a layer of state boundaries. You want the streets layer to be visible only when the zoom level is less than eight miles. You want the county boundary layer to display when the zoom level falls between 20 miles and 200 miles.You want the states boundary layer to be visible only when the zoom level is greater than 100 miles.

Generating Labels For a Layer

MapXtreme provides many ways to label attributes of geographic objects in a map layer. The location where they are drawn is based on the location of the geographic object's label point. This approximates, but is not necessarily the object's centroid.



As attributes, labels are dynamically connected to their map objects. If the data or geographic information changes, the labels change. The content of the label is determined by the data associated with the geographic object.

Layers can be set to be automatically labeled using Layer.**setAutoLabel** method. The method **isAutoLabel** returns True or False if the layer will be autolabeled.

In addition to label content, you control the display and style of automatic labels by using methods of the LabelProperties class.You can set conditions for displaying labels, in what style they will display,and what priority they have over all the objects in the layer.

Labeling has been greatly enhanced in this release of MapXtreme, enough so that it warrants a separate discussion. See Chapter 12: Labeling and Renditions.

Raster Images

Raster images are another type of layer you can include in your map. Rasters are computerized pictures consisting of row after row of tiny dots (pixels). These are sometimes known as bitmaps. Aerial photographs and satellite imagery are common types of raster data found in GIS.

You can display raster images in your MapXtreme Java application as backdrops to the maps you create. You then can overlay additional data, such as street maps and customer locations, on top of the image.

To display a raster image as a map layer, the image must contain geographic registration information, which are coordinates that correspond to earth locations. This will define the proper placement of the image in a map.

MapXtreme Java supports two types of raster images:

- Images that use an associated .tab file containing the geographic registration information. Raster images of this type include TIFF,JPEG, GIF,BMP,PNG, XBM, and MIG (MapInfo Grid).
- Images that have registration information contained in special tags in the image file. Formats of this type include GeoTIFF and MIG[2].

2. Note: While it is not necessary for a MIG file to have an associated .tab file, you cannot open a MI Grid image with Map Definition Manager directly. Open the associated tab file instead.

To register an image as a geographically correct image, you can bring the image into MapInfo Professional and register it there. Many USGS map images come with an associated .tab file.

Adding Raster Layers to MapJ

Raster images are brought into a map in the same way other map layers are added — by creating a Data Provider that describes the image and its location. In the case of rasters with associated .tab files, you would create a TABDataProvider. For GeoTIFF images, you would create a GeoTIFFDataProvider.

MapXtreme Java implements a flexible raster handling scheme to allow data providers to be created dynamically. In the case of the TABDataProvider, MapXtreme Java reads property information from the **rasterhandlerfactory.properties** file, which contains specific information about what raster handlers are available and in which order they should be tried. The items in the list are placed in a specific order to facilitate better raster handling. When the Data Provider is being created, the list is traversed. If the raster handlers that you will most likely be using are at the top of the list, performance will improve. You can reorder the list, if necessary.

The following raster Data Providers are available for TABDataProviders:

- JDKRasterDataProvider: Handles all JDK supported raster formats, currently JPEG and GIF.
- TIFFRasterDataProvider: Handles uncompressed, palette TIFF images. This is a very specific type of TIFF file, and provides speed and scalability performance.
- JimiRasterDataProvider: Will attempt to handle all other raster formats.

Note: If an image cannot be handled by one of the above raster Data Providers, then an exception is thrown indicating the specific file that cannot be handled.

The GeoTIFFDataProvider works similarly to the TABDataProvider but uses the **geotiffdataprovider.properties** file instead. The GeoTIFF handler cannot read tab files, so it must be added to a layer by itself.

The following code illustrates the how to create a GeoTIFFDataProvider and add the GeoTIFF image to a map. In this case, the image is stored on the local system and retrieved by the LocalDataProviderRef.

```
// Create a TableDescHelper that points to the Tiff image

GeoTIFFTableDescHelper geoTiffTDHelper = new
   GeoTIFFTableDescHelper("e:\\image\\geotiff.tif");

// Create DataProviderHelper (**note this constructor
   takes

// no parameters)

GeoTIFFDataProviderHelper geoTiffDPHelper = new
   GeoTIFFDataProviderHelper();

// If the data is local, use a LocalDataProviderRef

LocalDataProviderRef localDPRef= new
   LocalDataProviderRef(geoTiffDPHelper);

// Insert the layer into the map layer collection

map.getLayers().add(localDPRef, geoTiffTDHelper, "GeoTIFF
   Layer");
```

Considerations for Raster

The following sections offer things to keep in mind when using raster images.

Set Display to Raster Coordinate System

When adding a raster image to your map, make sure to set MapJ's display and numeric coordinate system to the raster layer's coordinate system since MapXtreme Java does not reproject raster images.

This code example shows how to determine the coordinate system for a raster layer and set the display and numeric coordinate systems accordingly:

```
TableInfo ti= rasterLayer.getTableInfo();

CoordSys cSys= ti.getCoordSys();

myMapJ.setDisplayCoordSys(cSys);

myMapJ.setNumericCoordSys(cSys);
```

Rasters and Performance

Due to the added demands of raster imagery, we recommend that you start your server or application with an expanded maximum heap size of 64 MB or more depending on your application and the types of raster files you use.

For example, to increase the maximum heap size when you are loading layers using the Map Definition Manager, from the command line, type:

```
java -mx64M com.mapinfo.mapdefman.MapDefManager
```

TIFF Images

Any handling of TIFF images must be done locally. The Renderer object and the TIFF file must exist on the same machine because the TIFF files need to be read as a random access file.

MI Grid Rasters

One of the raster image types that MapXtreme Java can display is a MI Grid raster (file extension .MIG). These are thematically shaded maps that are created in MapInfo Professional that show the data as continuous color gradations across the map. This type of thematic mapping is produced by an interpolation of point data from the source table. MapInfo Professional generates a grid file from the data interpolation and displays it as a raster image.

MI Grid images can be used like any other raster image. The registration information for the image is contained in an associated .tab file, so you must use the TABDataProvider to add it to a MapXtreme Java map.

Grid files for US rainfall, temperature and elevation are included in the MapXtreme Java sample data. To display a Grid file using Map Definition Manager, open the associated .tab file. You cannot open the .mig file directly with MDM. For more information on creating MI Grid files, see the MapInfo Professional documentation.

GIF Transparency

MapXtreme Java supports the ability to make a portion of the map image transparent to allow the layer below to show through. This is only available if the image is in GIF format of type 89a or later (type 87a does not support transparency.)

A GIF file is a palette-based raster format, where a given pixel value is determined from a lookup in a 256-color palette. Any of these colors can be transparent; however, MapXtreme Java only supports transparency for one palette entry per image.

To allow a certain palette entry to be made transparent, edit the file with an image editor, such as Paint Shop Pro™. See your image editor documentation for using transparent colors.

The illustration below shows an image on the left with a white border that, after setting the transparency in an image editor, is now transparent.