

# Sasha Maps

Alexander Maryanovsky

May 16, 2009

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Hello, World (literally)</b>	<b>3</b>
<b>3</b>	<b>Custom Maps</b>	<b>6</b>
<b>4</b>	<b>Custom Actions</b>	<b>8</b>
<b>5</b>	<b>Overlays</b>	<b>10</b>
<b>6</b>	<b>Widgets on the Map</b>	<b>11</b>
6.1	Overview Map Widget . . . . .	11
6.2	Zoom Widget . . . . .	12
6.3	Scale Widget . . . . .	12
<b>7</b>	<b>Map Widget Events</b>	<b>13</b>
<b>8</b>	<b>Web Map Service</b>	<b>14</b>
<b>9</b>	<b>Disclaimers</b>	<b>16</b>
<b>10</b>	<b>Resources</b>	<b>17</b>

# 1 Introduction

Sasha Maps is a GWT library similar in capabilities to the client-side of Google Maps. The core functionality is displaying and interacting with a map. A map consists of a set of huge images, each image represents a “zoom” at which the user may view the map. The larger the zoom, the larger and more detailed is the image representing it. Each such image is cut by the server into fixed-size images (tiles), and served this way to the client. This allows the client to download only the tiles it needs, instead of the entire image, as normally only a small portion of it is visible to the user.

## An overview of the currently implemented major features

- Displaying a tiled map. The library has built-in support for Google Maps and tiled WMS maps. To display your own maps, you need to implement a few Java interfaces, as described in the following chapters.
- The user may interact with the map using any of the standard ways, such as dragging, zooming in and out via the mouse scroll wheel, recentering on a clicked spot, zooming into a clicked spot and others.
- A developer may implement his own, custom ways for the user to interact with the map.
- Overlays may be added to (and removed from) the map. Overlays are widgets which are anchored at a specified location on the map. These may be used to mark certain locations on the map, such as points of interest.
- Widgets may be added to the map widget. These, unlike overlays, are positioned relative to the map widget, rather than relative to the map. These may be used to add buttons, labels, zoom controls etc. to the map widget.
- Several useful map widgets are provided:
  - An overview map widget which displays the map several zooms out, and highlights the portion visible on the main map.
  - A zoom widget which shows the current zoom and lets the user to quickly zoom in and out, or jump to a specific zoom.
  - A scale widget which displays the scale of view (i.e. how many real-world distance units, such as kilometers, there are in a certain amount of pixels).
- Locations on the map are described by their latitude and longitude, similarly to Google Maps.

## 2 Hello, World (literally)

Below is a small GWT application, demonstrating how to set up Sasha Maps.

```
package com.maryanovsky.mapdemo.client;

// Imports removed for brevity

/**
 * The entry point of the Map demo.
 */
public class TutorialDemo implements EntryPoint{

    /**
     * The Google Maps API key.
     */
    private static final String GOOGLE_MAPS_API_KEY = "Paste your key here";

    /**
     * Starts the map demo.
     */
    public void onModuleLoad(){
        GoogleMaps googleMaps = GoogleMaps.getInstance(GOOGLE_MAPS_API_KEY);
        TileLayer tileLayer = googleMaps.getNormalTileLayer();
        Map map = new Map(GoogleMaps.MERCATOR_PROJECTION, tileLayer, 0, 17);

        LatLng initialLocation = new LatLng(60.050317, 30.350161);
        int initialZoom = 13;

        MapWidget mapWidget = new MapWidget(
            new MapLocationModel(0, 17, initialLocation, initialZoom));
        mapWidget.setMap(map);

        UserEventManager eventManager = mapWidget.getUserEventManager();
        eventManager.setDragAction(MapWidget.DRAG_MAP_ACTION);
        eventManager.setDoubleClickAction(MapWidget.ANIMATED_ZOOM_IN_ACTION);
        eventManager.setWheelAction(MapWidget.ANIMATED_ZOOM_ACTION);
        eventManager.setRightClickAction(MapWidget.ANIMATED_PAN_MAP_ACTION);
        UiUtils.disableContextMenu(mapWidget.getElement());

        UiUtils.addFullSize(RootPanel.get(), mapWidget);
    }
}
```

I will assume the reader is already familiar with Java and GWT and will avoid explaining the boilerplate. Let's start, then:

```
private static final String GOOGLE_MAPS_API_KEY = "Paste your key here";
```

This defines the Google Maps API key you will be using to access Google Maps. After [registering for a key](#), replace the "Paste your key here" string with it.

```
GoogleMaps googleMaps = GoogleMaps.getInstance(GOOGLE_MAPS_API_KEY);
```

Create a `GoogleMaps` instance through which we will access Google Maps' various services.

```
TileLayer tileLayer = googleMaps.getNormalTileLayer();
```

The tile layer we will display. Here we will use a single tile layer - Google Maps' normal map. You can also display several stacked tile layers on the same map, but for simplicity, we will use just one here.

```
Map map = new Map(Google.MERCATOR_PROJECTION, tileLayer, 0, 17);
```

Create the map. Note that, unlike in Google Maps, the `Map` object is not the widget seen by the user - it is merely a description of the map. It consists of the projection, the list of tile layers, and the range of supported zooms. In our example, we use Google's implementation of the Mercator projection, the tile layer we obtained in the previous line, and a zoom range of 0 to 17 (inclusive).

```
LatLng initialLocation = new LatLng(60.050317, 30.350161);  
int initialZoom = 13;
```

Here we define the initial location of the center of the map, specified via its latitude and longitude coordinates, and the initially displayed zoom.

```
MapWidget mapWidget = new MapWidget(  
    new MapLocationModel(0, 17, initialLocation, initialZoom));
```

We create a new `MapWidget`, the widget which will display the map. We initialize it with a location model bound to the zoom range 0 to 17, and with the initial position at our chosen location and zoom. Note that although we specified the 0 to 17 range again, we may have chosen a different, smaller range, which would make the widget display only these zooms (even though the map itself has more available).

```
mapWidget.setMap(map);
```

The map widget is set to display the map we created earlier.

```
UserEventManager eventManager = mapWidget.getUserEventManager();
```

We obtain from the map widget its `UserEventManager`, which is the object responsible for the widget's interaction with the user.

```
eventManager.setDragAction(MapWidget.DRAG_MAP_ACTION);  
eventManager.setDoubleClickAction(MapWidget.ANIMATED_ZOOM_IN_ACTION);  
eventManager.setWheelAction(MapWidget.ANIMATED_ZOOM_ACTION);  
eventManager.setRightClickAction(MapWidget.ANIMATED_PAN_MAP_ACTION);
```

We set all the standard user actions, dragging, double-clicking, scroll-wheeling and right-clicking, to perform one of the actions that the library already implements. Of course you may choose to bind different actions to these user-actions, such as non-animated versions of the ones used here, or implement your own.

```
UiUtils.disableContextMenu(mapWidget.getElement());
```

This disables the browser's context menu for the map widget. This is necessary (although not supported by all browsers) because we have registered a right-click action with the map widget, and a context menu would get in the way.

```
UiUtils.addFullSize(RootPanel.get(), mapWidget);
```

Finally, we add the map widget to the root panel of the containing HTML page, at 100% of its size.

### 3 Custom Maps

In the demo app above, we used `GoogleMaps` twice - once with `GoogleMaps.getNormalTileLayer()` and once with `Google.MERCATOR_PROJECTION`. These are ready implementations of interfaces which, if you wish to display your own maps, you need to implement yourself. These two interfaces are `TileLayer` and `Projection`. The `TileLayer` interface defines how the map is cut into tiles and the URL at which each tile may be found. The `Projection` interface defines how latitude-longitude coordinates map to pixel coordinates within the large map image for each zoom.

#### TileLayer

```
public interface TileLayer{
    SizeView getTileSize();
    String getTileUrl(int x, int y, int zoom);
}
```

As you can see, the `TileLayer` interface defines two methods - `getTileSize`, which specifies the size of each tile (in pixels) and `getTileUrl`, which returns the URL of the tile at a specified location (in pixels, relative to the top-left of the full map image for the specified zoom). An implementation might look like this:

```
import com.maryanovsky.map.client.*;
import com.maryanovsky.gwtutils.client.*;

public class ExampleTileLayer implements TileLayer{

    private static final SizeView TILE_SIZE = new Size(256, 256);

    public SizeView getTileSize(){
        return TILE_SIZE;
    }

    public String getTileUrl(int x, int y, int zoom){
        if ((zoom < 0) || (zoom > 5)) // Outside the supported zoom range
            return null;

        x /= TILE_SIZE.getWidth();
        y /= TILE_SIZE.getHeight();

        return "http://tiles.example-map.com/tile?" +
            "x=" + x + "&y=" + y + "&zoom=" + zoom;
    }
}
```

## Projection

```
public interface Projection{
    Point fromLatLngToPixel(LatLng latLng, int zoom);
    LatLng fromPixelToLatLng(PointView pixel, int zoom);
    double getZoomMagnification(int startZoom, int endZoom);
    SizeView getWrapSize(int zoom);
}
```

The two methods `fromLatLngToPixel` and `fromPixelToLatLng` convert between latitude-longitude and pixel coordinates. Pixel coordinates are relative to the top-left of the large image for each zoom, and they increase down and to the right, just like normal images do.

The `getZoomMagnification` method returns (approximately) how much larger each pixel is at `startZoom` than at `endZoom`, in terms of the real-world distance (meters) covered by it. With Google Maps, for example, each successive zoom increases the pixels per meter scale by two, so the implementation for Google Maps simply returns `Math.pow(2.0, endZoom - startZoom)`.

The `getWrapSize` method is needed for displaying maps that wrap around on either the X or the Y axis. For example, most interactive maps of the Earth will seamlessly wrap at the  $\pm 180^\circ$  longitude line. This method should return the number of pixels, at the given zoom, after which the map wraps. The width property of the returned `SizeView` object corresponds to the wrap on the X axis; height to the Y axis. A value of 0 in one of the properties indicates that the map does not wrap on the corresponding axis. Most earth maps, for example, do not wrap on the Y (longitude) axis.

## AbstractProjection

Many projections are implemented in a similar way, and to avoid repeating this code in each one, Sasha Maps provides a convenient partial (abstract) implementation - `AbstractProjection` in the `com.maryanovsky.maps.client.projections` package. `AbstractProjection` divides the responsibility of handling the projection itself and the zoom magnification between two separate objects. Projecting is delegated to two abstract methods - `fromLatLngToPixelImpl(LatLng)` and `fromPixelToLatLngImpl(double, double)`, which need only to project at some predefined “native” zoom. The definition of zoom magnification is delegated to an implementation of a new interface, `ZoomStrategy`, which has a single method, `getZoomMagnification`, with the same semantics as `Projection.getZoomMagnification`.

The `com.maryanovsky.maps.client.projections` also provides some ready implementations of commonly used projections:

- `ScaleProjection`, which simply scales the  $[-180, 180] \times [-90, 90]$  range of legal latitude-longitude values to the size of the map at each zoom.
- `MercatorProjection`, which implements the commonly used mercator projection.

Additionally, two commonly used implementations of `ZoomStrategy` are provided:

- In `FixedCoefZoomStrategy`, each zoom magnifies by a constant factor more than the previous one.
- `ResolutionListZoomStrategy` takes a list of resolutions (amount of real-world units per pixel), one for each zoom, and uses them to calculate the magnifications between the zooms.

## 4 Custom Actions

The demo application mapped user-actions to the standard actions, such as dragging a map and zooming it in and out. Now we will see how you can implement your own actions. The relevant class here is `UserEventManager`, whose job is to capture the low-level events a browser sends, and convert them into high-level events by invoking an “action” you tell it to. As shown in the demo application, you obtain the `MapWidget`’s `UserEventManager` by invoking `MapWidget.getUserEventManager()`. Here are its methods, which allow you to have your action invoked when a certain event occurs:

```
public class UserEventManager{
    public void setClickAction(ClickAction clickAction);
    public void setDoubleClickAction(ClickAction clickAction);
    public void setTripleClickAction(ClickAction clickAction);
    public void setRightClickAction(ClickAction clickAction);
    public void setMiddleClickAction(ClickAction clickAction);
    public void setDragAction(DragAction dragAction);
    public void setWheelAction(WheelAction wheelAction);
}
```

To write an action, you must implement the relevant action interface - `ClickAction`, `DragAction` or `WheelAction` (support for `KeyAction` to be added soon). Let’s look at them.

```
public interface ClickAction{
    void mouseClicked(Widget target, PointView point);
}
```

`ClickAction` is the simplest - it defines a single method, `mouseClicked`, which is invoked when a mouse button is clicked (which button, and how many times it is to be clicked depends on the `UserEventManager` method you pass it to). It is given the widget and the coordinate the user clicked on (the coordinate is relative to the widget’s top left).

```
public interface DragAction{
    void mouseDragged(Widget target, PointView mousePoint, PointView dragOffset);
    void mouseDragEnded(Widget target, PointView releasePoint, PointView dragOffset);
    void mousePressed(Widget target, PointView pressed);
    void mouseReleased(Widget target, PointView point);
}
```

`DragAction` defines four methods, two of them auxiliary. `mouseDragged` is invoked whenever a mouse is moved while the left mouse button is pressed. It is passed the widget on which the drag occurred, the coordinate where the mouse-down event occurred and the offset from that point to the current location of the mouse. `mouseDragEnded` is invoked when the mouse is released following a drag gesture. It is passed the same arguments as `mouseDragged`.

The `mousePressed` and `mouseReleased` methods are auxiliary because they are not always needed for drag actions. They are only useful if you need something to occur (at least programmatically, if not visually) not when the mouse is dragged, but immediately when it is pressed (before being moved). The `mousePressed` method is invoked on exactly this event. Then, if the user releases the mouse button



before moving the mouse, the `mouseReleased` method is invoked. Otherwise, the `mouseDragged` method is invoked, and the `mouseReleased` method is *not* invoked when the drag gesture ends.

```
public interface WheelAction{
    void mouseWheelScrolledUp(Widget target, PointView point);
    void mouseWheelScrolledDown(Widget target, PointView point);
    void mouseWheelScrolled(Widget target, PointView point,
        MouseWheelVelocity velocity);
}
```

The `WheelAction` interface defines methods for two high-level events - `mouseWheelScrolledUp` and `mouseWheelScrolledDown`, and one method for a low-level event which is simply passed on, as received from the browser - `mouseWheelScrolled`. The reason for the necessity of `mouseWheelScrolled` is due to browser incompatibilities, and is outside the scope of this tutorial. For most things, the two other methods should be sufficient. They are invoked whenever the mouse wheel is scrolled in the corresponding direction, and they are also given the current location of the mouse pointer.

You may have noticed that the actions above are all given the user-action's "target" widget. You may also think that for a `MapWidget`'s `UserEventManager`, the target widget is the map widget itself. You may even be correct, but that is an implementation detail, and may change in the future. To obtain the map widget from the target widget, the `MapWidget` class provides a special method:

```
public class MapWidget{
    public static MapWidget getMapWidgetFromActionTarget(Widget target);
}
```

For convenience, `MapWidget` defines, for each action interface, a "map action" abstract class, which implements that action interface. Each implemented method obtains the map widget from the target widget, and invokes a new, abstract method, passing it the `MapWidget`. Let us take a look at `MapClickAction`, for example:

```
public abstract static class MapClickAction implements ClickAction{
    public void mouseClicked(Widget target, PointView point){
        MapWidget mapWidget = MapWidget.getMapWidgetFromActionTarget(target);
        if (mapWidget.getMap() != null)
            mapClicked(mapWidget, point);
    }

    protected abstract void mapClicked(MapWidget mapWidget, PointView point);
}
```

These abstract classes allow you to receive a `MapWidget` instead of obtaining it from the target widget yourself. The downside is that you have to subclass instead of implementing an interface.

## 5 Overlays

Overlays are entities which are notified whenever the location displayed by a map widget changes (even during a temporary changes, such as a drag gesture). Normally, an overlay will add some sort of widget to the map widget's overlay panel and use these notifications to update its location, so that the widget appears to be attached to the map. In fact, the `WidgetOverlay` class, which implements the `Overlay` interface, does just that with any widget you care to give it a reference to. Let's take a look at these classes.

```
public interface Overlay{
    void added(MapWidget mapWidget);
    void updated(MapWidget mapWidget, boolean isTemporary);
    void removed(MapWidget mapWidget);
}
```

All of `Overlay`'s method are notification methods. `added` and `removed` are invoked when the overlay is added and removed from the map respectively, while `updated` is invoked when the map widget's location changes (the `isTemporary` parameter specifies whether the change is temporary, such as during a drag-map gesture, or permanent, such as when the drag gesture ends). When the `added` and `removed` methods are invoked, the overlay is expected to add/remove whatever the widgets are that the overlay places on the map (an image, for example) to/from `mapWidget.getOverlayPanel()`. When `updated` is invoked, the overlay should adjust the location of its widgets, or remove them altogether, if they strolled outside the visible portion of the map (and add them back again, if they strolled back in).

Mostly, however, you will want to use the convenient `WidgetOverlay` class, which does all the work for you. It just needs the widget you would like to place on the map, its location (in latitude-longitude) and the offset of the widget's top-left point, in pixels, from that point. The offset is needed, if for example, you want to put an image of an arrow on the map, and you want the arrow tip, rather than the top-left of the image, pointing at your sweetheart's house. Additionally, `WidgetOverlay` provides convenient methods for making the widget draggable. `WidgetOverlay.makeOffsetDraggable`, for example, will put the overlay into a mode when its offset changes when the widget is dragged, while `WidgetOverlay.makeAnchorDraggable` makes it adjust its anchor instead.

## 6 Widgets on the Map

`MapWidget` provides you with a panel overlaying the map widget, placed exactly at its origin and sized exactly to its dimensions. This panel, obtainable via `mapWidget.getWidgetPanel()` is useful for adding widgets to the map widget which do not move when the map's location changes. To place a widget in one of the map widget's corners, take a look at the `UiUtils.addInCorner` method in the `GwtUtils` library.

The library provides some widgets commonly used in interactive maps. You can find these widgets in various subpackages of the `com.maryanovsky.map.client.widgets` package.

Many of these widgets can be completely customized with regards to how they look. They do, however, provide a default look, and to properly use that, you will need to import their CSS into your page's HTML. You can do this by importing either the individual widget's css file or the "widgets/widgets.css" file, which in turn imports all of the widgets' css files. The individual widget's css file is named after the widget's subpackage and is found in the directory corresponding to its subpackage relative to the `com.maryanovsky.map.client` package. For example, the scale widget's package is `com.maryanovsky.map.client.widgets.scale` and its css file is therefore "widgets/scale/scale.css".

### 6.1 Overview Map Widget

`com.maryanovsky.map.client.widgets.overview.OverviewMapWidget` is a subclass of `MapWidget` which follows around the location of another, "main", `MapWidget`. To create an `OverviewMapWidget`, you must give its constructor the main `MapWidget`, and an object defining the meaning of "follow" - a `Synchronization` (an inner static interface of `OverviewMapWidget`). I will not go into the details of what `Synchronization` does, as you will rarely want to implement it yourself. Instead, use its friendly implementation, the `StandardSynchronization` (also an inner static class of `OverviewMapWidget`).

`StandardSynchronization`'s constructor takes two parameters - the first determines how far zoomed out the overview map will be, relative to the main map, and the second, whether the overview map widget will follow its main counterpart using animated or non-animated panning. The second parameter is a straightforward boolean, but the first one requires a little explaining. How far the overview map is zoomed out is determined dynamically, by comparing the sizes of the two map widgets and making sure that a certain portion of the area (of the world) displayed by the overview map is covered by the area displayed by the main map widget. The exact (relative) size of that portion is the first argument to `StandardSynchronization`'s constructor. If you pass 0.5, for example, the overview map widget will display at the largest zoom such that the area displayed by the main map widget will occupy at most half of the overview widget. If you don't understand the above, or just don't care to understand, the recommended value for the first parameter of `StandardSynchronization` is 0.4.

To conclude, let's see an example of creating an `OverviewMapWidget` and adding it to the bottom right corner of a map widget. Assume we already have a `MapWidget`, aptly named `mapWidget`.

```
MapWidget overviewMapWidget = new OverviewMapWidget(mapWidget,
    new OverviewMapWidget.StandardSynchronization(0.4, true));
overviewMapWidget.setMap(mapWidget.getMap());
overviewMapWidget.setSize("150px", "150px");
overviewMapWidget.getUserEventManager().setDragAction(
    MapWidget.DRAG_MAP_ACTION);
UiUtils.addInCorner(mapWidget.getWidgetPanel(), overviewMapWidget,
    Alignment.BOTTOM_RIGHT, new Size(0, 0));
```

## 6.2 Zoom Widget

`com.maryanovsky.map.client.widgets.zoom.LargeZoomWidget` shows the user what zoom the map is currently at, and lets him zoom in or out and to quickly jump to a desired zoom. By default it looks similarly to GoogleMaps' zoom widget, but it can be customized to look completely different. To use the default look, you need to import the "widgets/zoom/zoom.css" file into your page's CSS. Other than that, using the zoom widget is straightforward:

```
LargeZoomWidget zoomWidget = new LargeZoomWidget(mapWidget);
UiUtils.addInCorner(mapWidget.getWidgetPanel(), zoomWidget,
    Alignment.TOP_LEFT, new Size(0, 0));
```

To customize the looks of the `LargeZoomWidget`, you must implement the `LargeZoomWidget.Images` interface and pass an instance of the implementing class to the appropriate `LargeZoomWidget` constructor. To implement the interface, you need 6 images:

- The zoom in and zoom out button images.
- The top and bottom parts of the zoom "ladder".
- The "step" of the zoom "ladder". This is the part that is repeated for every zoom level.
- The "mark" image. This is the image indicating the current zoom.

## 6.3 Scale Widget

`com.maryanovsky.map.client.widgets.scale.ScaleWidget` is a non-interactive widget which shows the user the scale of the map, that is, how many real-world distance units there are in a certain amount of pixels currently visible on the map. `ScaleWidget`'s constructor takes two arguments: the map widget and a `ScaleWidget.Scale`. The 2nd argument is an interface which defines the size of the scale widget in pixels and the text it displays. If your application is showing a map of the Earth, however, you can use an existing implementation - `EarthScale`. You can either instantiate `EarthScale` yourself (it takes the preferred size of the widget, a list of sizes in meters it may display and a list of size names to display) or use one of the static method which give you an instance. There are currently two such methods: `getEnglishMetricInstance(int)` for metric units and `getEnglishImperialInstance(int)`, for imperial units.

Here's an example which creates and adds a scale widget which uses metric units and a preferred size of 100 pixels:

```
ScaleWidget scaleWidget =
    new ScaleWidget(mapWidget, EarthScale.getEnglishMetricInstance(100));
UiUtils.addInCorner(mapWidget.getWidgetPanel(), scaleWidget,
    Alignment.BOTTOM_LEFT, new Size(16, 16));
```

## 7 Map Widget Events

`MapWidget` provides fairly standard (in the Java world) methods for listening to changes in its properties. Properties such as the displayed map, the size of the widget and the temporary offset of the map (the one used during a drag gesture) can be tracked by registering a `PropertyChangeListener` (although not of the `java.beans` variety, as that package did not make it into GWT, but from the `com.maryanovsky.gwtutils.client` package). The location of the map, including the zoom, is encapsulated by the `MapLocationModel` class, which lets you register a `ChangeListener` (again, from the `com.maryanovsky.gwtutils.client` package) with it.

User events should be processed via the `UserEventManager`, as described in a previous section.

## 8 Web Map Service

In addition to Google Maps, the library supports the widely used Web Map Service protocol (the tiled variant defined by the WMS Tiling Client Recommendation). Support for WMS maps can be found in the `com.maryanovsky.maps.client.opengis.wms` package. WMS maps are a bit more complicated to use than Google Maps, but fortunately the library hides most of this complexity from you.

To use a WMS map you first need to obtain the `Capabilities` object describing the service's capabilities. If you are using your own WMS server, this is not a problem - you just send an HTTP request and let the library parse the response and pass you back the `Capabilities` object. If you are using someone else's WMS server or if your WMS server is at a different host than your maps client, the same-host security restriction kicks in, and you will be unable to send a request to the WMS server without a proxy. Fortunately, there are many free proxies available on the web, and it is even not very difficult to write your own. Of course you can also simply grab the XML document that is sent in response to a WMS Capabilities request and put it as a regular XML file on your web server.

Assuming you will be querying a real WMS server for its capabilities, you will need to use the `Capabilities.makeGetRequestUrl` method to build the full URL of the request from the base URL of the service. Upon receiving the response, you can create a `Capabilities` object by passing the response XML document to the `Capabilities` constructor. Alternatively, you can use the `XmlResponseHandler` implementation, `Capabilities.ResponseHandler`, to do all the work for you, including error handling.

Altogether, here is how it would look:

```
try{
    ResultCallback<Capabilities> callback = new ResultCallback<Capabilities>(){
        public void requestFinished(){
            public void resultReceived(Capabilities capabilities){
                showMap(capabilities);
            }
        }
    };

    String url = Capabilities.makeGetRequestUrl("http://mymaps.com/wms-c/");
    RequestBuilder requestBuilder = new RequestBuilder(RequestBuilder.GET, url);
    requestBuilder.sendRequest(null,
        new Capabilities.ResponseHandler(url, callback));
} catch (RequestException e){
    Window.alert(e.getMessage());
}
```

With a `Capabilities` object in hand, you need to query it for the `TileSet` you want to display. There are two ways to do this. The `Capabilities.getTileSets` method simply returns the list of `TileSets` defined by the service. If you know the name of the layer you want to display, however, there is a more convenient method - `Capabilities.getTileSetsByLayerName(String)`. This method will find and return all the `TileSets` which display the specified layer (normally there is just one), sorted by the number of layers displayed in the `TileSet`, in increasing order. This lets you pick the `TileSet` with the least number of unwanted layers:

```
TileSet tileSet = capabilities.getTileSetsByLayerName("basic").get(0);
```

Armed with the `Capabilities` object and the right `TileSet`, you can now create the `Map`:

```
Map map = Wms.createMap(capabilities, tileSet, true);
```

The last argument to `Wms.createMap` specifies whether the map will wrap at the maximum longitude. Now that you have a `Map` object, you create a `MapLocationModel` and a `MapWidget` the same way you normally would (query the map for the supported zoom range).

Note that `Wms.createMap` only works if the SRS of the `TileSet` is supported (currently, EPSG:4326 and OSGEO:41001 are supported). If it isn't, you have more work ahead of you in implementing `Projection`. You can then, however, use `TileSetLayer` for your `TileLayer`, so at least that part has been taken care of.

## 9 Disclaimers

Although now reasonably mature, and, in my opinion, production ready, Sasha Maps is still very much a work in progress. It lacks some functionality, but should be relatively bug-free on the latest versions of popular modern browsers (Firefox 3, Internet Explorer 7, Safari 3.1 and Opera 9.5). If you encounter any bugs, please email me at [msasha@gmail.com](mailto:msasha@gmail.com).

Sasha Maps is inspired by, but does not use any code from Google Maps. It is written entirely from scratch, by me.

The maps it is capable of displaying are not my property, but the property of whoever owns them. To actually display them on your website, you normally need permission from the owner. To use the maps provided by Google, you need to [sign up for the Google Maps API](#), although you will not need their actual API.



## 10 Resources

- [GWT \(Google Web Toolkit\)](#)
- [Sasha Maps](#)
- [Sasha Maps demo application](#)
- [Sasha Maps library javadoc](#)
- [GwtUtils library javadoc](#)