Part 2 - Maps API

This document refers to the deprecated Google Maps for Android v1 . Maps v1 is not compatible with the newer Google Maps Android API v2 . There is a Xamarin.Android sample using Google Maps for Android v2 at Github.

Using the Maps application is great, but sometimes you want to include maps directly in your application. In addition to the built-in maps application, Google also offers a native mapping API for Android. The Maps API is suitable for cases where you want to maintain more control over the mapping experience. Things that are possible with the Maps API include:

- Programmatically zooming and panning a map.
- Displaying traffic information.
- Displaying a satellite view.
- Adding built-in zoom controls.
- Annotating a map with overlays.

Before examining each of these features in detail, you must first set up a few things in the Google Maps API.

Configuring Maps API Prerequisites

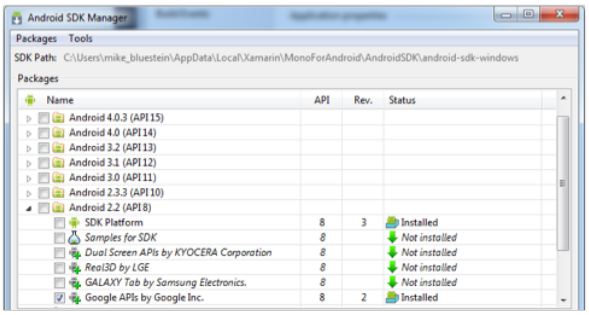
Several items need to be configured before you can use the Maps API, including:

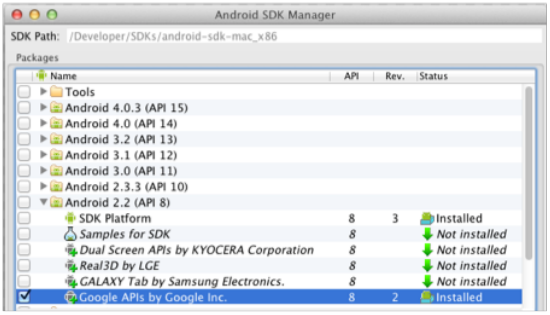
- Installing the Google APIs Add-On.
- Creating an emulator with the Google APIs.
- Obtaining a Maps API key.
- Adding a Google Maps assembly reference.

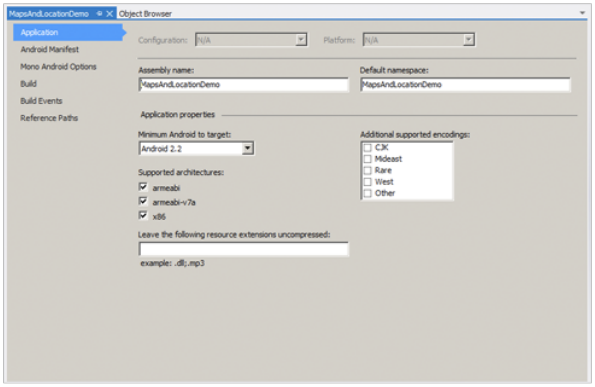
Google APIs Add-On

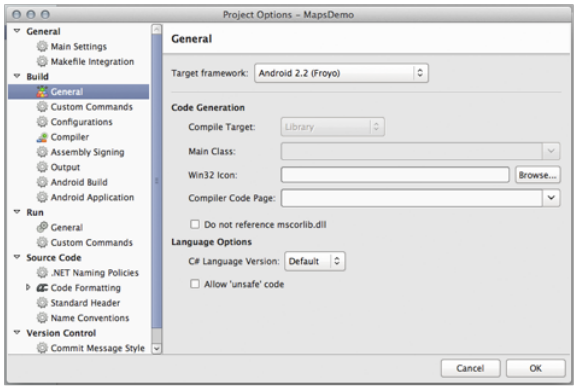
Google ships the Maps API as part of the Google APIs Add-On , which is separate from the core Android SDK. You can use the Android SDK Manager to install the Google APIs as follows:

1. Launch the Android SDK Manager by selecting Tools > Open AVD Manager in Xamarin Studio, or Tools > Start Android Emulator Manager in Visual Studio.
2. Select the Show: Updates/New check box.
3. Select Google APIs by Google Inc. corresponding to the target framework you are using. The target framework is available in Xamarin Studio Project Options under the Build > General section and in Visual Studio under the Project Properties > Application tab. By default, this will be set to Android 2.2, as shown in the screenshots below for both Windows and OSX.









4. Click Install Packages.

Creating an Emulator with Google APIs

On a device, Google APIs should be available. To work with maps in an emulator, the Google APIs need to be selected when you create the virtual device. Follow these steps to create a virtual device with Google APIs:

1. From the Android SDK Manager select Tools > Manage AVDs.
2. Click New.
3. Enter a name for the virtual device.
4. In the Target dropdown, select Google APIs for the API level you installed above.
5. Click Create AVD.

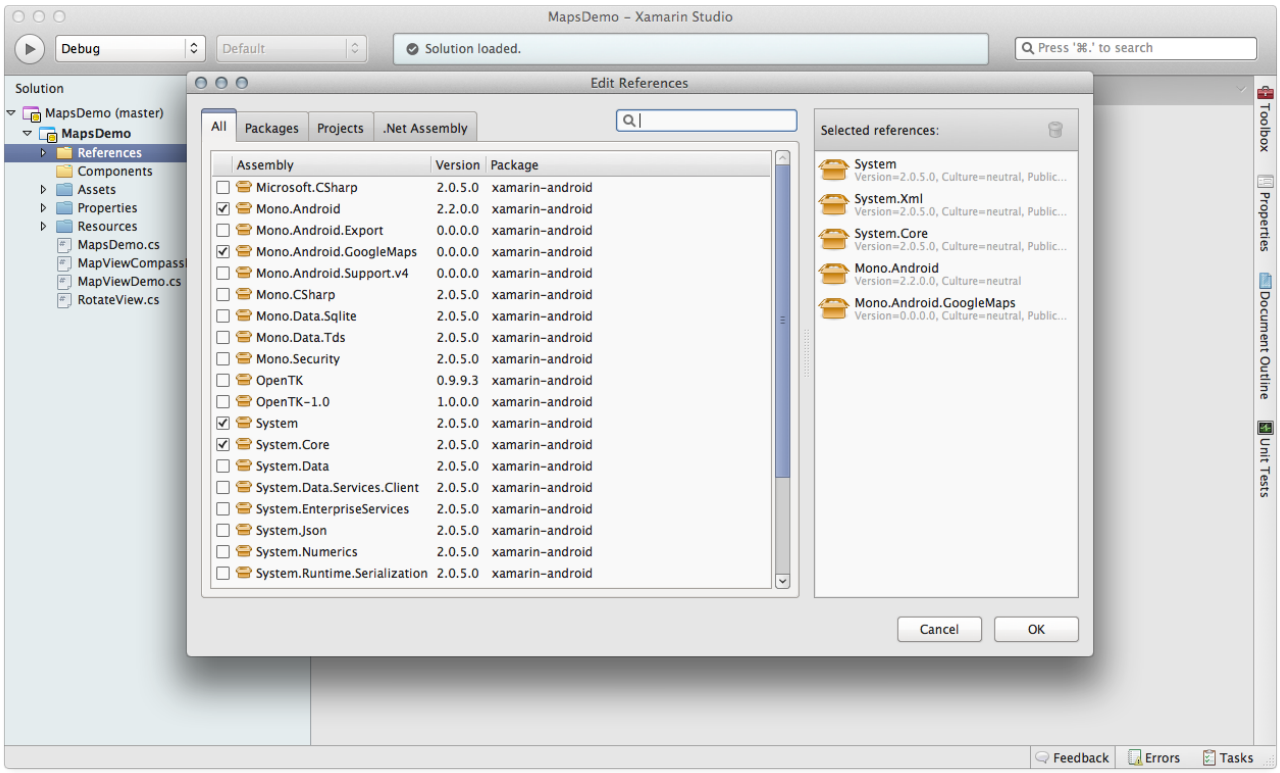
> **Note:** Google does not currently ship an x86 emulator image with Google APIs.

Map API Key

You need to obtain an API key from Google before you can use the Maps API. For information about how to obtain and use an API key with Xamarin.Android, see the document Obtaining a Google Maps API Key .

Adding Google Maps Reference

The Maps API is available in Xamarin.Android in the Mono.Android.GoogleMap assembly, which needs to be added as a project reference, as shown below:



Once all these prerequisites are in place, we can use the MapView control to incorporate maps in our application, as we'll now discuss.

The MapView Control

The MapView class encapsulates Google maps in a reusable control. As it is a subclass of the ViewGroup class, it can be added in XML or code, just like any other view. It can also encapsulate other views, as shown later in this article when we'll add overlays to a map. For now, consider a basic example of adding a MapView in an application.

Simple MapView Example

The first thing we need is the XML markup that includes the MapView. The following XML specifies a MapView within a LinearLayout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:orientation="vertical"
              android:layout_width="fill_parent"
              android:layout_height="fill_parent">
       <com.google.android.maps.MapView
              android:id="@+id/map"
              android:layout_width="fill_parent"
              android:layout_height="wrap_content"
              android:apiKey="<em>YOUR_API_KEY</em>" />
</LinearLayout>
```

We declare the MapView in XML like this in the same way that we would for any other view, except there is an added requirement to include the apiKey. With the XML in place, you can now use

Xamarin.Android code to see how a MapView is added within an Activity.

Subclassing MapActivity

The Activity used to include the MapView must make the MapActivity class into a subclass and override IsRouteDisplayed. Attempting to add a MapView to an Activity that does not subclass MapActivity will throw an exception. Since this example doesn't display routing information, the IsRouteDisplayed implementation returns false, as shown below:

```
[Activity (Label = "SampleMapActivity")]
public class SampleMapActivity : MapActivity
{
	protected override void OnCreate (Bundle bundle)
	{
		base.OnCreate (bundle);
		SetContentView (Resource.Layout.MapLayout);
	}

	protected override bool IsRouteDisplayed {
		get {
			return false;		}
	}
}
```

If we run the code now, the user cannot interact with the static map that is presented. To make the map interactive, you must set a few properties on the MapView.

MapView Properties

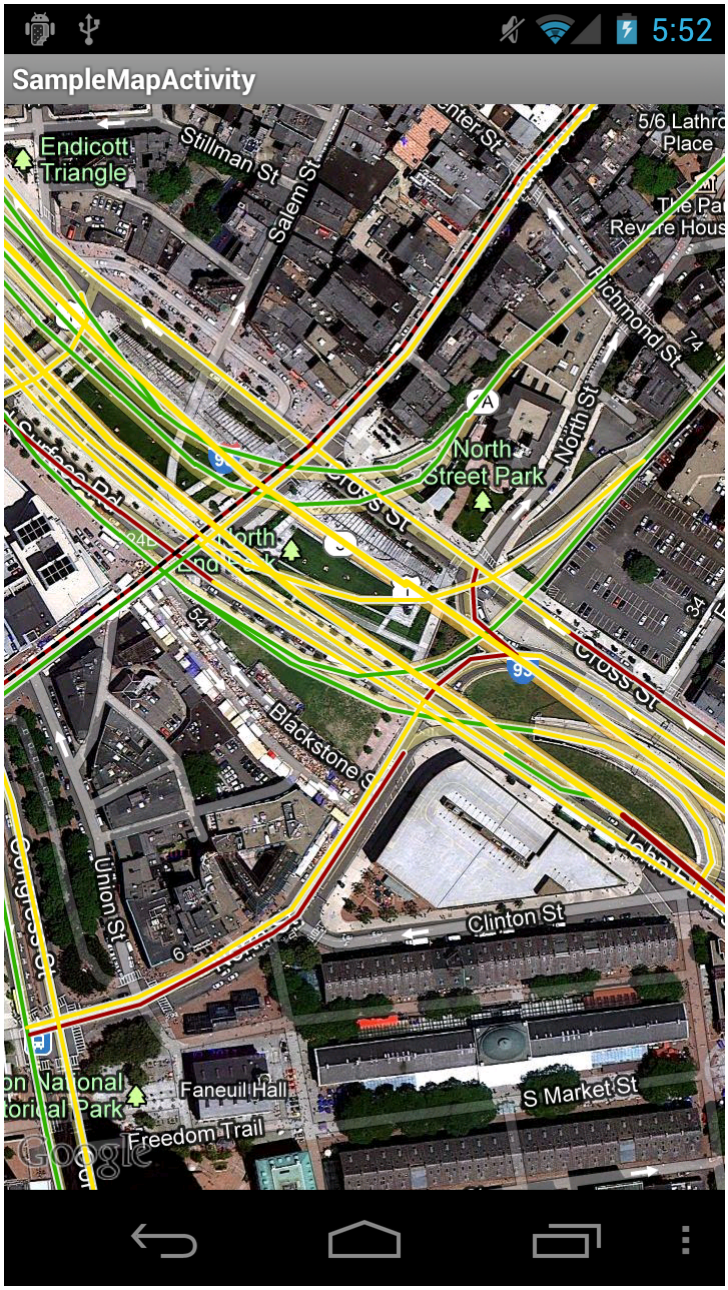
The MapView class defines several properties that control the functionality of the map. The most basic is the Clickable property, which must be set to true for panning and pinch-to-zoom to work.

```
var map = FindViewById<MapView>(Resource.Id.map);
map.Clickable = true;
```

Additionally, there is out-of-the-box support for showing traffic information and satellite imagery through the Traffic and Satellite properties, which are set as shown below:

```
map.Traffic = true;
map.Satellite = true;
```

With these properties set, the resulting map can now be panned and zoomed. It also displays the satellite imagery (mentioned above) and traffic information as color-coded annotations over streets. Red denotes areas of high traffic, yellow means moderate flow, and green is light, as shown in the following screenshot:

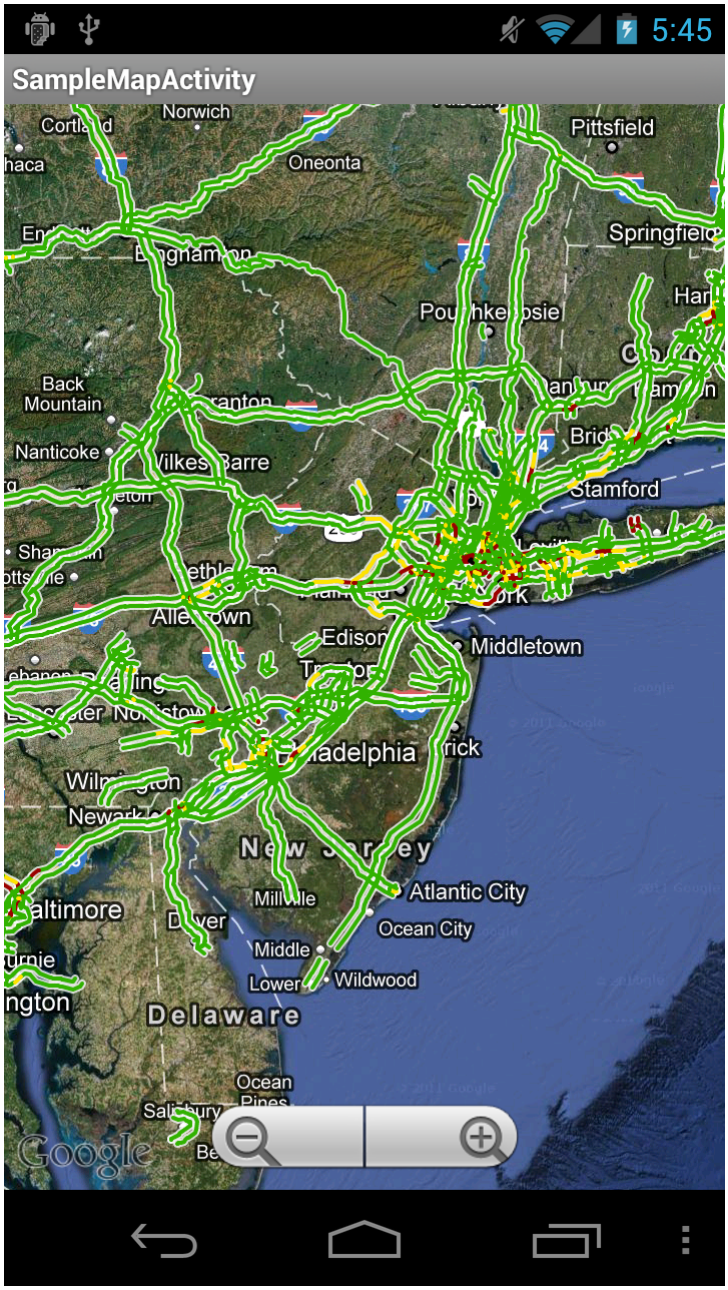


Using Built-in Zoom Controls

The screenshot above has been zoomed in (using touch gestures) to clearly show the traffic levels. The MapView also supports built-in zoom controls, which can be turned on by calling the SetBuiltInZoomControls method like this:

```
map.SetBuiltInZoomControls (true);
```

The resulting map now shows zoom controls at the bottom of the screen. These controls appear when you tap on the map, as shown:



In addition to using touch gestures and zoom controls, the MapView control can also be zoomed and panned programmatically. We'll cover this functionality next.

Controlling Maps with the MapController

The MapController class contains a variety of helper functions that allow an application to zoom and position a MapView from code. For zooming, it enables functionality such as:

- Setting the zoom level.
- Zooming in and out incrementally.
- Zooming in and out incrementally while maintaining a fixed point on the screen.

The MapController also supports programmatic positioning of a MapView, enabling such features as:

- Setting the location of the map's center point directly.
- Animating the map to a new location.

Zoom level

The zoom level of a MapView defines its scale, ranging from 1 to 21, where 1 is fully zoomed out and 21 is fully zoomed in. At a zoom level of 1, the equator is scaled to 256 pixels. Zoom increases by a factor of 2 for each successive zoom level.

To set the zoom level, call the SetZoom method as shown:

```
map.Controller.SetZoom (10);
```

If a MapView is already displayed, calling SetZoom will change its display to the new zoom level without any animated transition. To animate below zoom levels, several methods are available.

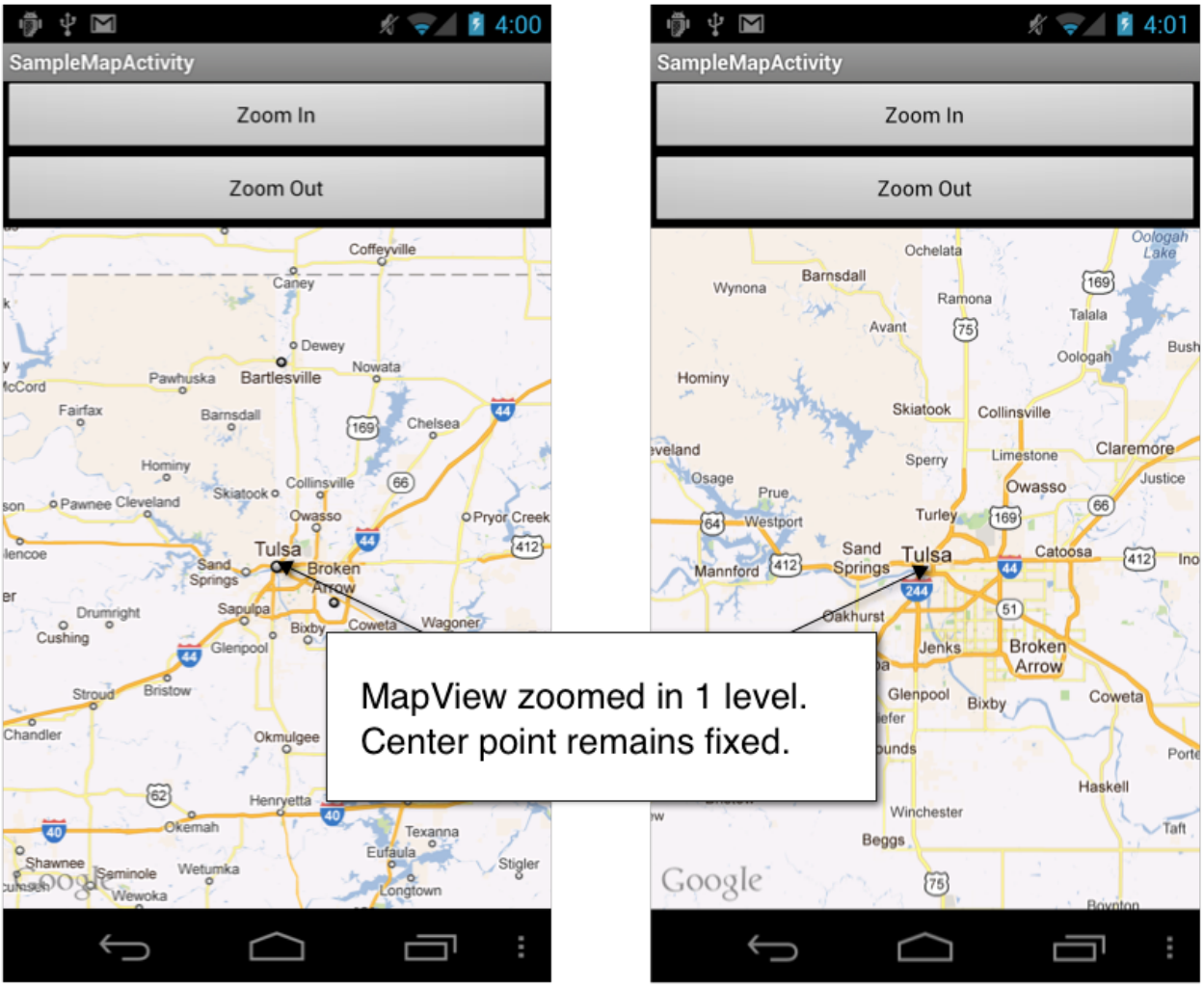
Zoom Methods

The MapController class has several other methods for zooming with animated transitions. For example, the ZoomIn and ZoomOut methods will cause the map to zoom in and out, respectively, just like the built-in zoom controls.

Consider an Activity with two buttons that call ZoomIn and ZoomOut in their click event handlers as shown here:

```
zoomInButton.Click += (sender, e) => {
       map.Controller.ZoomIn ();
};

zoomOutButton.Click += (sender, e) => {
       map.Controller.ZoomOut ();
};
```

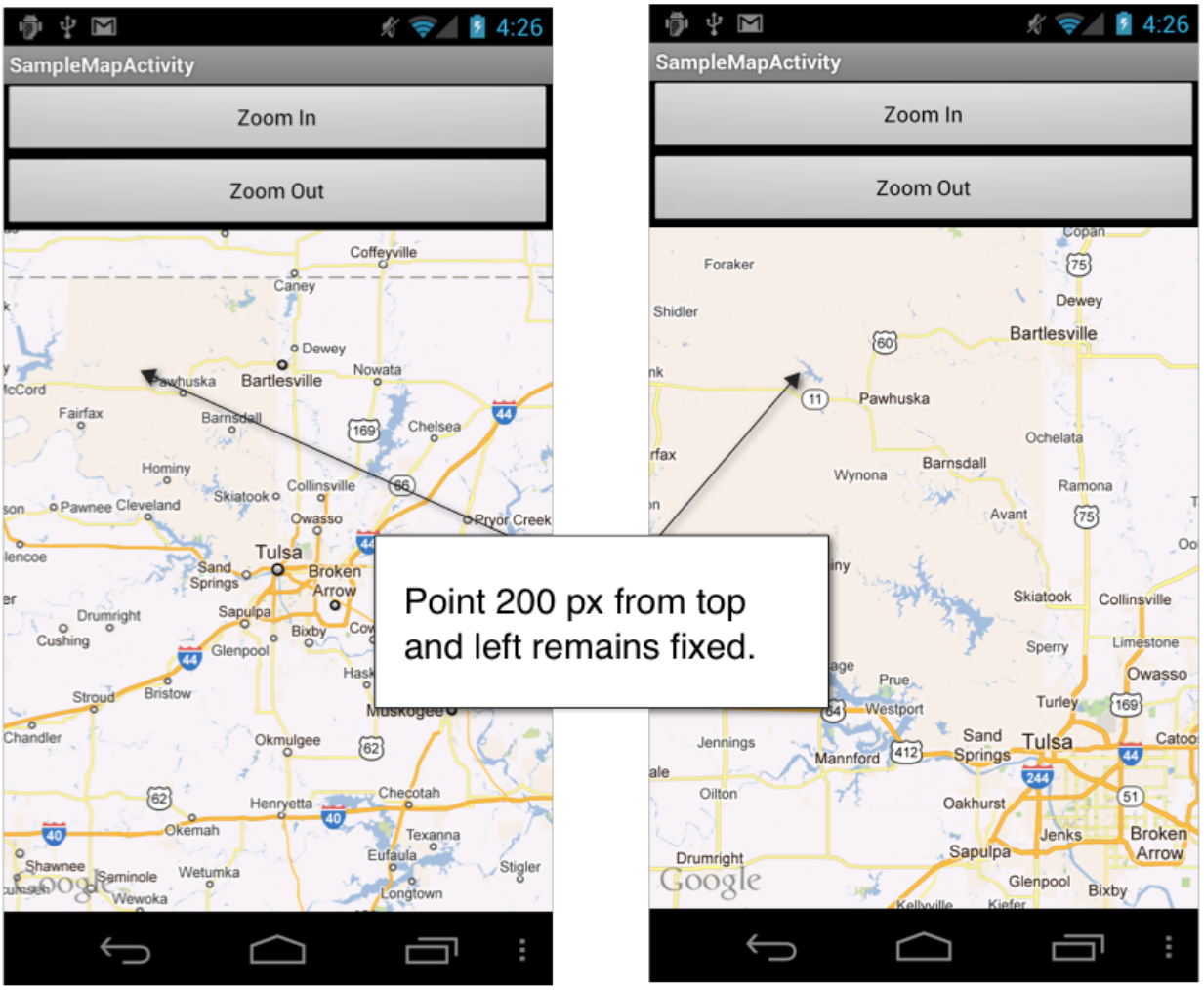
This code will zoom the map in and out one level at a time with an animation between each transition, keeping the center point of the map fixed, as shown below:



Additionally, the MapController contains the ZoomInFixing and ZoomOutFixing methods, which will keep a specified point fixed on the screen while zooming. To illustrate, let's change the earlier code to call these methods:

```
zoomInButton.Click += (sender, e) => {
       map.Controller.ZoomInFixing (200, 200);
};

zoomOutButton.Click += (sender, e) => {
       map.Controller.ZoomOutFixing (200, 200);
};
```

Instead of the center remaining fixed, the map now stays fixed on the point we specify, measured in pixels from the upper left of the MapView, as shown below:



Zooming is one way to control a MapView's display. The MapController also supports positioning a map to show different geographic locations.

Positioning a Map

The MapController's SetCenter method sets a map's center to a specified geographic location. SetCenter takes an instance of a GeoPoint class, which encapsulates the latitude and longitude as integers measured in micro degrees, as shown below:

```
map.Controller.SetCenter (
       new GeoPoint ((int)42.374260E6, (int)-71.120824E6));
```

Calling SetCenter will cause the map to display the new location at its center without any transitional animation. However, the MapController also includes support for animating to locations as well.

Animating A Map Change

To animate a MapView's center to a new location, you can call the MapController's AnimateTo method. The simplest form of AnimateTo takes a GeoPoint for the new location as shown below:

```
map.Controller.AnimateTo(
```

```
       new GeoPoint ((int)40.741773E6, (int)-74.004986E6);
```

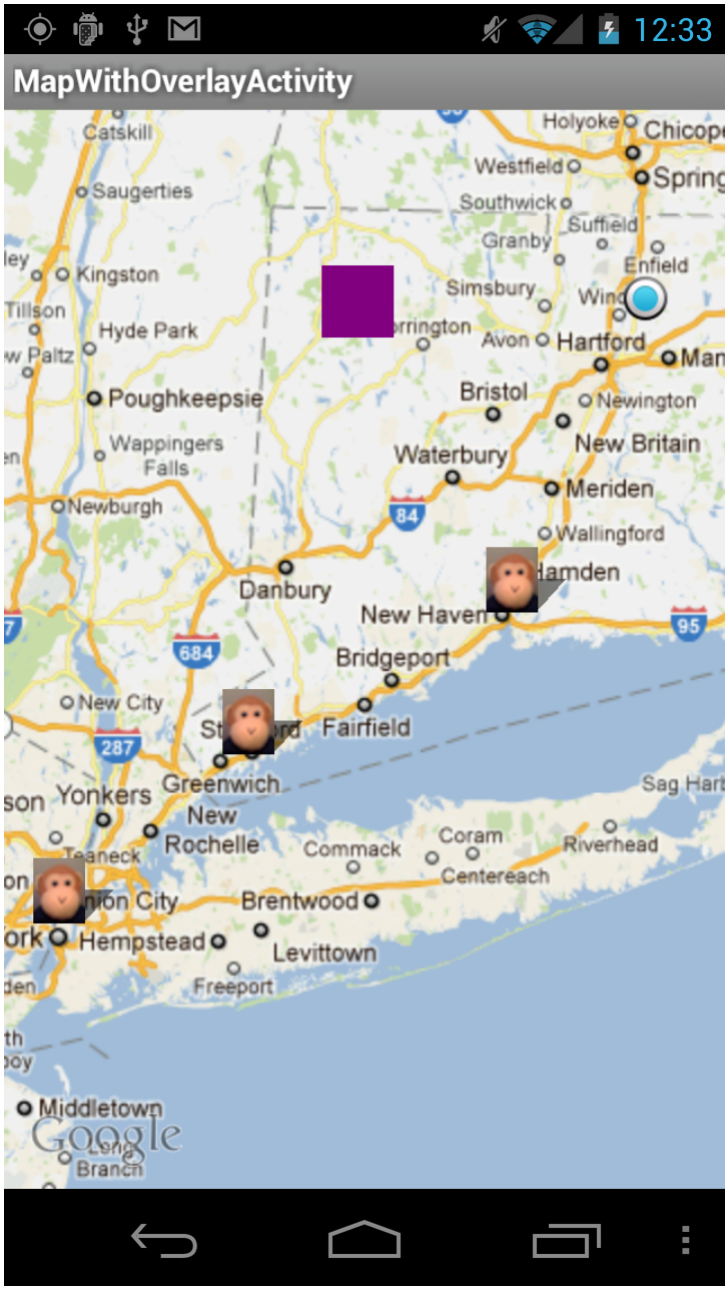
Like the SetCenter method, this code will center the map on the new location, only this time the map will pan rather than jump directly to the new point.

MapController also supports running code when a map animation has completed. For instance, if we call the overloaded version of AnimateTo that takes an Action delegate, we can pass a lambda expression that will execute when the animation completes. For example, the following code displays a toast message upon map change completion:

```
map.Controller.AnimateTo (
       new GeoPoint ((int)40.741773E6, (int)-74.004986E6), () => {
       var toast = Toast.MakeText (this, "Welcome to NY", ToastLength.Short);
       toast.Show ();
});
```

Adding Overlays to a Map

The MapView class includes several mechanisms for displaying overlay graphics on a map, as shown below:



With the MapView you can:

- Add an overlay for the current location of a device.
- Add multiple overlays that use the ItemizedOverlay class.
- Add custom drawn overlays.
- Add overlays comprised of other views.

Each of these techniques is examined below.

Adding an overlay for the current location

Android's built-in overlay class, called MyLocationOverlay, automatically displays the device's current location; it does not require you to manually determine the device location yourself.

The following screenshot shows a MapView with the device's location appearing as a blue dot:

To use the MyLocationOverlay follow these steps:

1. Create an instance of MyLocationOverlay.
2. Add the overlay to the MapView's Overlays collection.
3. Call the overlay's EnableMyLocation method in OnResume.
4. Call the overlay's DisableMyLocation method in OnPause.
5. Set the required permission to ACCESS_FINE_LOCATION and/or ACCESS_COARSE_LOCATION.

To create a MyLocationOverlay instance, pass its constructor a context and a MapView instance that will be used to display the overlay. Then, add it to the Overlays collection as shown below:

```
_myLocationOverlay = new MyLocationOverlay (this, map);
map.Overlays.Add (_myLocationOverlay);
```

The above code can be called in the Activity's OnCreate method. However, before the overlay displays itself on the map, it needs to call EnableMyLocation. As determining location is expensive in terms of battery life, it's best to call EnableMyLocation in the Activity's OnResume method, and then subsequently call DisableMyLocation in OnPause as shown below:

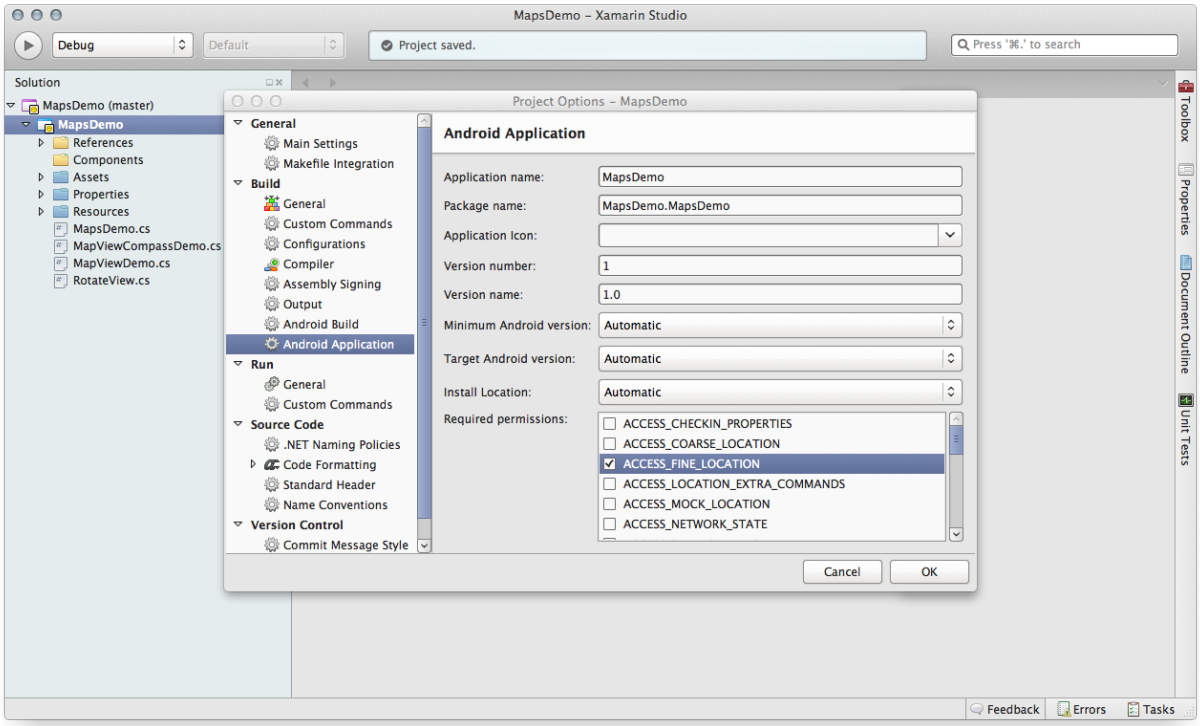
```
protected override void OnResume ()
{
	base.OnResume ();
	_myLocationOverlay.EnableMyLocation ();
}

protected override void OnPause ()
{
	base.OnPause ();
	_myLocationOverlay.DisableMyLocation ();
}
```

If you call DisableMyLocation in OnPause, this will ensure that location services won't consume the battery when the Activity is not on the screen. For a full discussion of Android's activity lifecycle, see the Activity Lifecycle article.

Before the MyLocationOverlay instance will display itself on the MapView, we must set the permission (in the manifest file) that is required before we can access the device's location. To set the ACCESS_FINE_LOCATION permission, follow these steps:

1. Double-click on the project in the Solution Explorer.
2. Select Xamarin.Android Application under the Build section.
3. Click Add Android Manifest (if one doesn't exist).

Select ACCESS_FINE_LOCATION from the Required permissions list, as shown below:



Handling Location Updates

When MyLocationOverlay receives its first location update from the system, it will call the Action delegate we pass to the RunOnFirstFix method, after which the location will be available in overlay's MyLocation property. For example, the following code moves the map to the device's location when it

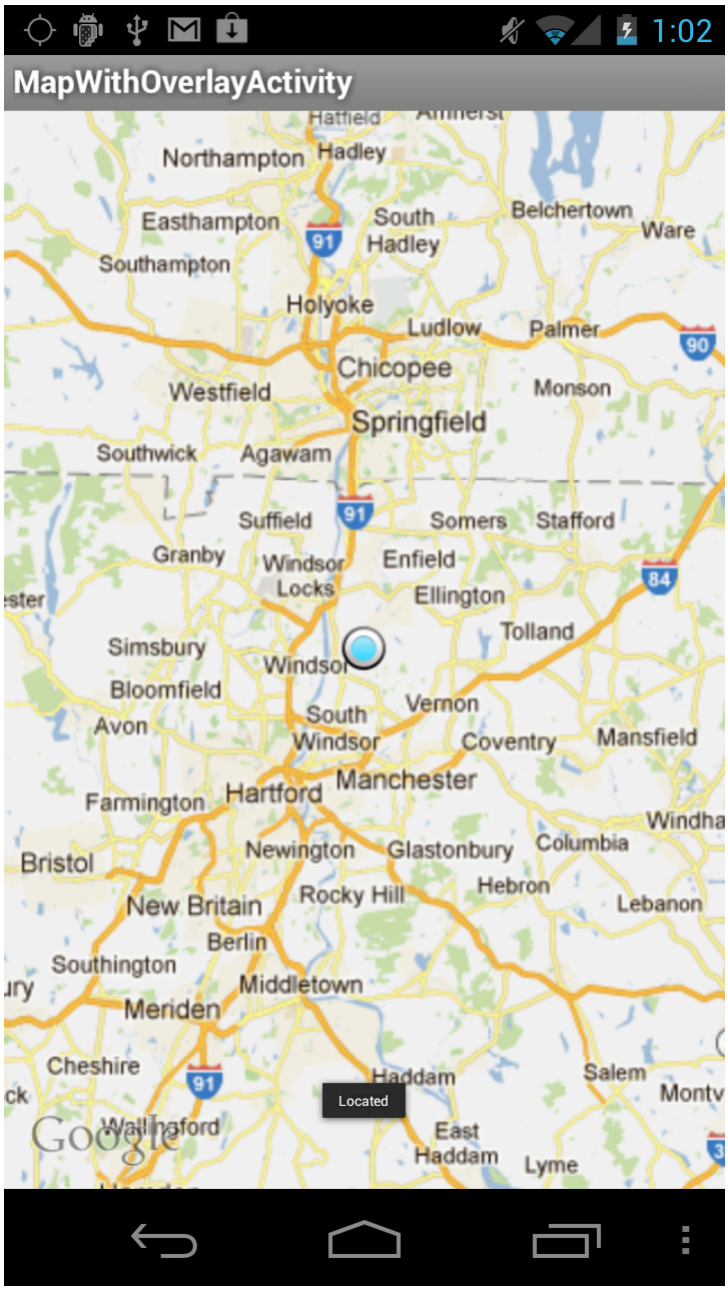
is first determined:

```
_myLocationOverlay.RunOnFirstFix(() => {
	map.Controller.AnimateTo(_myLocationOverlay.MyLocation);
});
```

This code will run in its own thread, so in order to update the UI, we need to wrap the code in a call to RunOnUIThread, as shown below:

```
_myLocationOverlay.RunOnFirstFix (() => {
	map.Controller.AnimateTo (_myLocationOverlay.MyLocation);

	RunOnUiThread (() => {
		var toast = Toast.MakeText (this, "Located", ToastLength.Short);
		toast.Show ();
	});
});
```

Running the code above will display a toast, in addition to the overlay showing the current location:

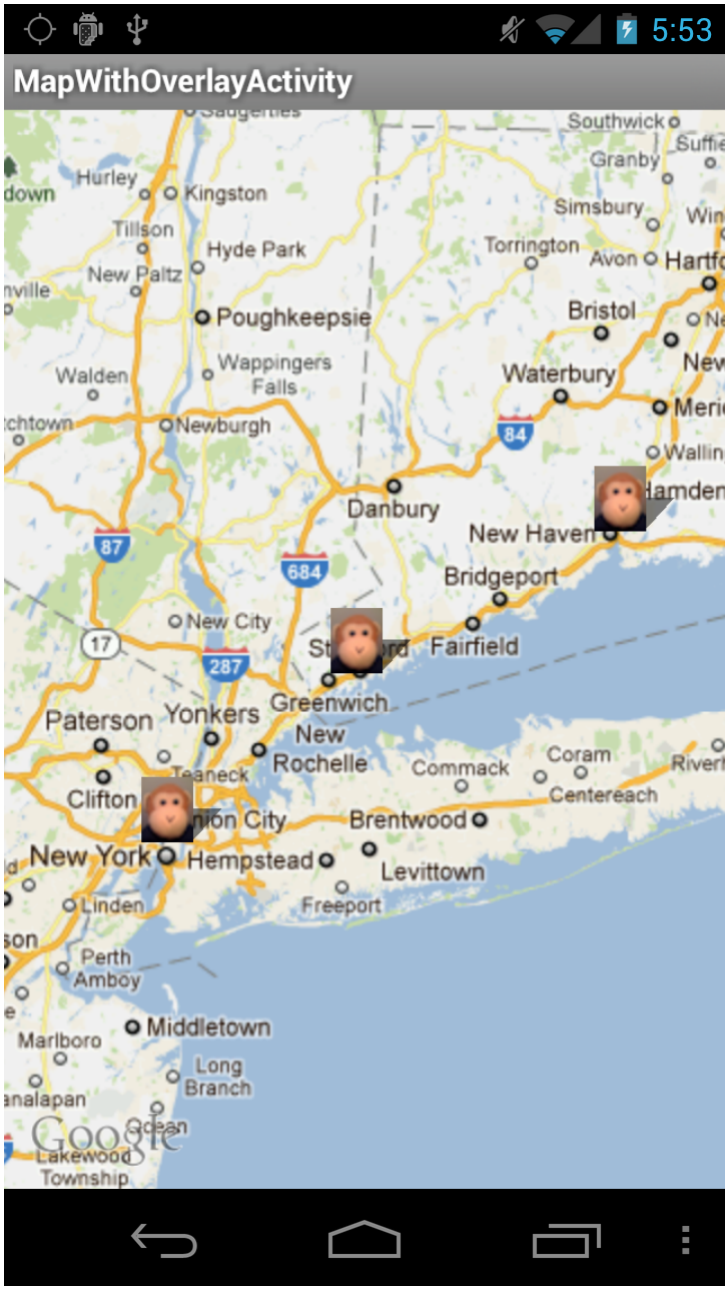


Besides showing the current location, Android uses the ItemizedOverlay class to support displaying

other types of overlays.

Adding Itemized Overlays

By subclassing the ItemizedOverlay class, you can add images to a map at the locations of your choosing. The class is designed to make it easy to add several items to a map. For instance, the following example adds images to the map at various locations, as shown below:



First, for the overlay in this example, an image is added to the Resources > drawable folder in the solution explorer, setting the build action to AndroidResource. This example uses a file named *monkey.png*, which is available in the source code that accompanies the article.

The purpose of the ItemizedOverlay class is to hold a list of *OverlayItem* objects. Each OverlayItem encapsulates the location to place an overlay—and optionally title and snippet strings—that can be used to display a message when the item is tapped. The following code shows several sample

OverlayItems created in the constructor of our ItemizedOverlay subclass:

```
class MonkeyItemizedOverlay: ItemizedOverlay
{
	List<OverlayItem> _items;

	public MonkeyItemizedOverlay (Drawable monkey) : base(monkey)
	{
		// populate some sample location data for the overlay items
		_items = new List<OverlayItem>{
			new OverlayItem (new GeoPoint ((int)40.741773E6,
				(int)-74.004986E6), null, null),
			new OverlayItem (new GeoPoint ((int)41.051696E6,
				(int)-73.545667E6), null, null),
			new OverlayItem (new GeoPoint ((int)41.311197E6,
				(int)-72.902646E6), null, null)
		};

		BoundCenterBottom(monkey);
		Populate();
	}
	â¦
}
```

As you can see, the constructor for the ItemizedOverlay subclass takes a Drawable containing the overlay's image. This Drawable is used for the UI of every OverlayItem contained in the ItemizedOverlay.

This constructor implementation also calls the methods BoundCenterBottom and Populate:

- BoundCenterBottom – Puts the origin at the bottom center of the drawable, so its shadow comes off the bottom edge.
- Populate - Calls the Size and CreateItem methods internally to return the OverlayItems to the MapView.

Since the code shown earlier adds OverlayItem data in the constructor, you should subsequently call Populate to force calls to CreateItem. Call Populate the number of times specified by Size. The CreateItem and Size implementations are shown below:

```
protected override Java.Lang.Object CreateItem (int i)
{
	var item = _items[i];
	return item;
}

public override int Size ()
{
	return _items.Count();
}
```

This code returns the number of OverlayItems in Size and returns each one in CreateItem for the given index.

To use the MonkeyItemizedOverlay class, simply create it by passing a drawable to the constructor, and then adding it to the MapView's Overlays collection, as shown below:

```
var monkey = Resources.GetDrawable (Resource.Drawable.monkey);
var monkeyOverlay = new MonkeyItemizedOverlay (monkey);
map.Overlays.Add (monkeyOverlay);
```

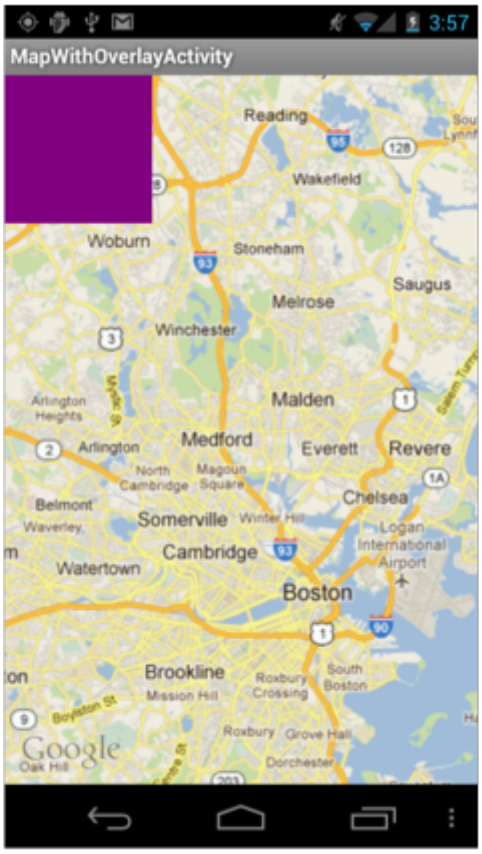
When running this code, the custom images will be added to the map at the proper locations, as shown in the earlier screenshot.

Custom Drawn Overlays

For more control over the overlay's graphics, you can also subclass the Overlay class and override the Draw method. For example, the following code will create a rectangle on top of the map when an instance of CustomMapOverlay is added to the MapView's Overlays collection:

```
class CustomMapOverlay : Overlay
{
	public override void Draw (Android.Graphics.Canvas canvas,
		MapView mapView, bool shadow)
	{
		base.Draw (canvas, mapView, shadow);

		var paint = new Paint ();
		paint.AntiAlias = true;
		paint.Color = Color.Purple;

		canvas.DrawRect (0, 0, 100, 100, paint);
	}
}
```

As you can see below, the pixel coordinates in the upper left of the screen show where the rectangle is placed. However, when the map is moved or scaled, the graphics for the overlay remain fixed:

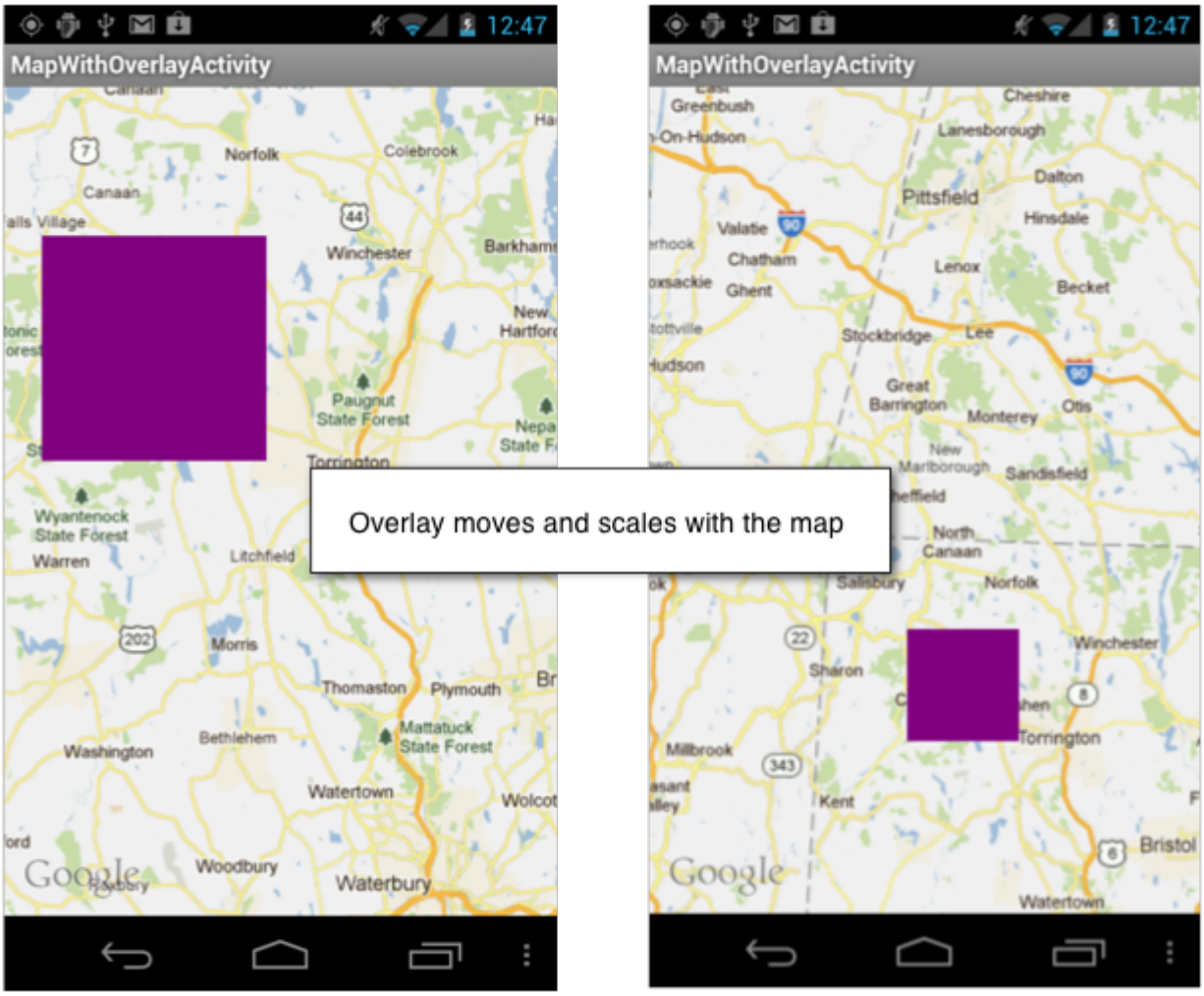


If you want the graphics to move and scale with the map, you need to create them based upon GeoPoints. The MapView class contains a Projection property that will return a Projection instance with methods that convert between geographic and screen coordinates. You can use these methods to position and size the overlay as shown in the following code, which creates a 20,000m square overlay:

```
var gp = new GeoPoint ((int)41.940542E6, (int)-73.363447E6);
var pt = mapView.Projection.ToPixels (gp, null);
```

```
float distance = mapView.Projection.MetersToEquatorPixels (20000);

canvas.DrawRect (pt.X, pt.Y, pt.X + distance, pt.Y + distance, paint);
```

The ToPixelsmethod converts a GeoPoint to a pixel location, which allows you to specify the overlay's screen position based upon a location. Also, you should use the MetersToEquatorPixels to set the overlay's size based upon the current zoom level. Now when we zoom or pan the map, the overlay adjusts accordingly, as shown below:



MapView as a view container

Since the MapView class is a ViewGroup, it can contain other views. Therefore, you can use the AddView method to add controls directly to a map. For instance, the following example adds a button with a click event handler that sets the map's center to a different location:

```
var mapButton = new Button (this){Text = "Go to Maui"};

var layoutParams = new MapView.LayoutParams (100, 50,
       new GeoPoint ((int)42.374260E6, (int)-71.120824E6),
       MapView.LayoutParams.TopLeft);

mapButton.Click += (sender, e) => {
       map.Controller.SetCenter (
              new GeoPoint ((int)20.866667E6, (int)-156.500556E6));
};
```

```
map.AddView (mapButton, layoutParams);
```

Use the MapView.LayoutParams to pass a GeoPoint, which anchors the button to a specific geographic location as shown below:

As you've seen, Android offers rich mapping support. Along with mapping, Android also provides access to location services, which is discussed next.

Next: Part 3 - Location Services

Source URL: http://docs.xamarin.com/guides/android/platform_features/maps_and_location/part_2_-_maps_api