Chapter 28. Location and Maps

Maps have become a crucial component of mobile applications—both for finding various landmarks that might be located in a particular area, and for navigation. Generally, maps are used in conjunction with location services that provide geographic coordinates and descriptions of landmarks, and possible navigation options.

Xamarin.Forms includes a `Map` element that derives from `View` and is rendered with platform-specific map objects. On all platforms, the map has the following features:

- The map can be displayed with streets or as an aerial view using satellite imagery.
- The map can indicate the user's location.
- A program using the map can center it on a particular location with a particular span (or zoom level).
- `Pin` objects can identify landmarks on the map.
- A user can manipulate the map with pan and pinch gestures.

In addition, Xamarin.Forms includes a `Geocoder` class that converts between text addresses and geographic coordinates.

Other types of location services—such as navigation, identifying points of interest, or geofencing—are not provided through Xamarin.Forms. If your application requires those services, you'll need to handle them yourself—possibly with platform-specific code. One feature not provided by Xamarin.Forms is a class that obtains the user's current location. Although the Xamarin.Forms map can *display* the user's current location, the application does not have direct access to that location. This is desirable information, so before exploring the `Map` and `Geocoder` classes, the first code presented in this chapter creates a dependency service that obtains the user's geographic coordinate.

But first, let's take a moment to explore the coordinate system used throughout this chapter.

The geographic coordinate system

As you know, any point on a plane can be identified with two numbers that indicate the point's location along horizontal and vertical axes. Similarly, any point on the surface of a sphere can also be identified with two numbers. These are called the *latitude* and *longitude*, and they are expressed as angles.

The Earth rotates on an axis that (conceptually speaking) passes through the north and south poles. Midway between these poles is the equator, which is a circle around the surface of the Earth, shown here in red:

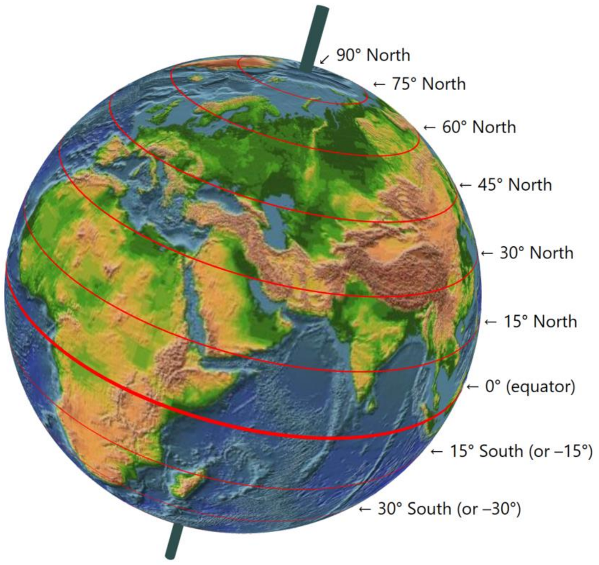
The equator is an example of a *great circle*, which is a circle on the surface of a sphere whose center is also the center of the sphere. The equator divides the Earth into northern and southern hemispheres.

The Earth is not exactly a sphere, however. Due to the centrifugal force that results from the spinning of the Earth on its axis, the Earth bulges towards the equator, creating a shape known as an *oblate spheroid* or *oblate ellipsoid*. The radius of the Earth ranges from 6,357 kilometers (3,950 miles) at the poles to 6,378 kilometers (3,963 miles) at the equator. The length of the equator is 2π times the radius at the equator, or 40,075 kilometers.

Detailed work with maps requires this non-spherical shape to be taken into account, but that's a little beyond the scope of this chapter. The variation in the radius is only about one-third of 1 percent, so it's often convenient to assume a mean radius of 6,371 kilometers (3,959 miles).

Parallels and Latitude

The angles of latitude and longitude are measured by an imaginary line with one end fixed at the center of the Earth and the other end on the surface. The latitude is an angle measured north or south relative to the equator. For any particular latitude, there is a circle on the surface of the Earth of all locations at that same latitude:



Except for the equator, these are *not* great circles. Geometrically, they are referred to as *small circles*. In the context of maps, the circles of equal latitude are known as *parallels*. The parallels get smaller as you approach the north or south poles, and then disappear into a point.

Latitudes range from 0 degrees at the equator to 90 degrees at the poles. Latitudes can be specified as north or south of the equator, or as positive and negative values. Latitudes north of the equator are positive values from 0 degrees to 90 degrees at the north pole, while latitudes south of the equator are negative values, again beginning at 0 degrees at the equator to –90 degrees at the south pole.

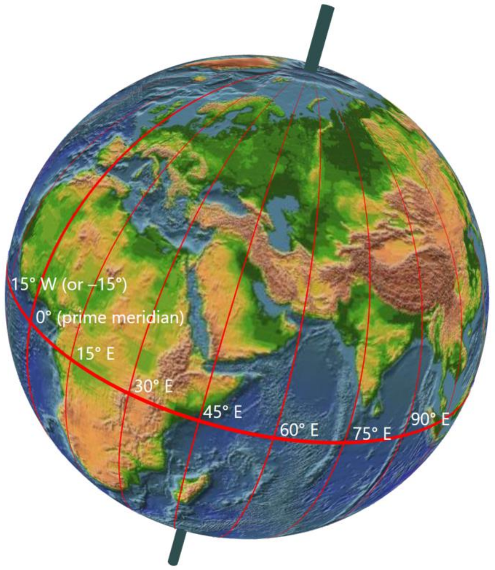
Boston has a latitude of about 42 degrees, San Francisco is a little closer to the equator with a latitude of 38 degrees, and Sydney is south of the equator with a latitude of about –34 degrees.

When determining the radius and circumference of a parallel, it's convenient to assume that the Earth is spherical, in which case the radius is the product of the radius of the Earth and the cosine of the latitude. For example, the radius of the parallel at 45 degrees is calculated by multiplying the mean radius of the Earth (6,371 kilometers) times the cosine of 45 degrees, or 0.707. That's a radius of 4,505 kilometers and a circumference of 28,306 kilometers.

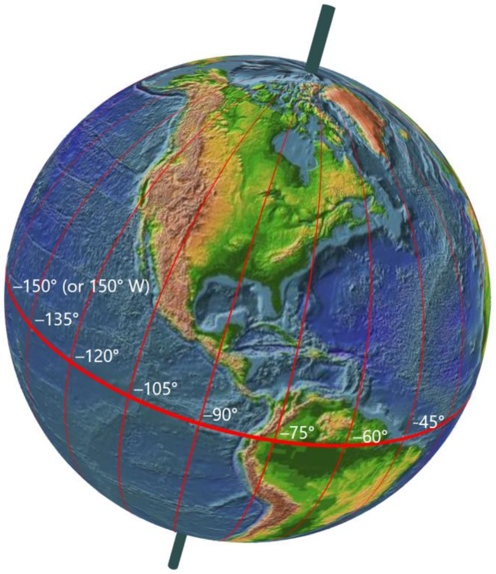
Longitude and meridians

The longitude indicates the eastern or western location of a point on the sphere. Lines of equal longitude are halves of great circles from the north pole to the south pole. These are called *meridians*. All angles of longitude are relative to a special meridian, called the *prime meridian*, which by international treaty passes through the Royal Observatory in Greenwich, London.

Longitudes can be specified as east or west of the prime meridian, or as positive or negative values. By the most common convention, positive angles of longitude are to the east of the prime meridian ranging from 0 degrees to 180 degrees to encompass the entire eastern hemisphere:



Negative angles of longitude are to the west of the prime meridian, ranging from 0 degrees to –180 degrees and encompassing the western hemisphere:



The meridian at 180 degrees is the same as the meridian at –180 degrees of longitude, and is opposite the prime meridian. This is called the *180th meridian* or *antimeridian* and roughly corresponds to the International Date Line.

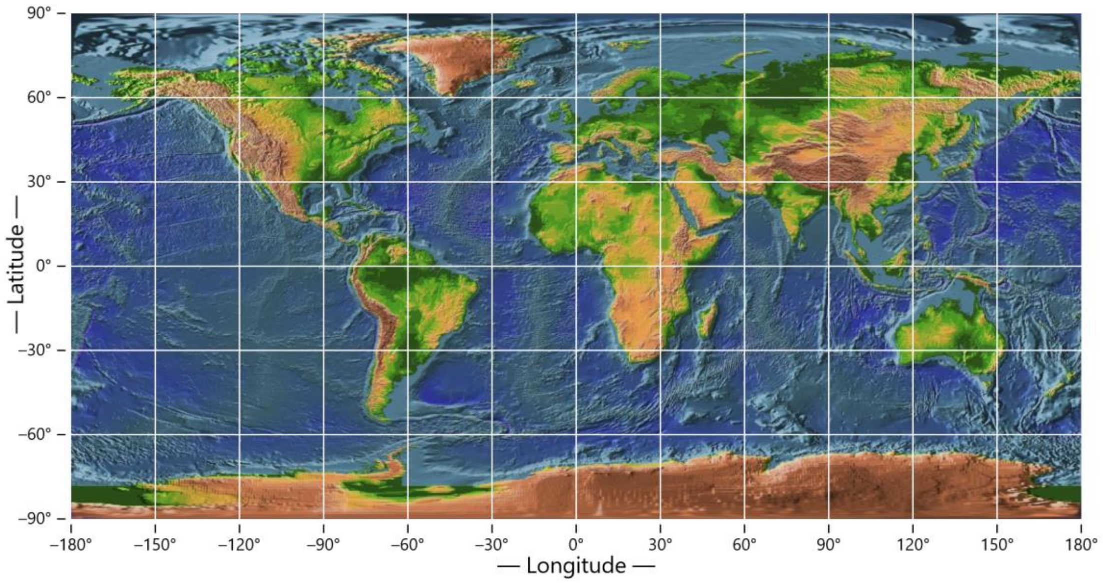
Boston has a longitude of about 71 degrees west or –71 degrees, San Francisco is –122 degrees, and Sydney is 151 degrees.

It might seem awkward that meridians are great circles and parallels are small circles, but that's the compromise that must be made to identify points on a spherical surface. The system was developed over two millennia ago by Greek mathematicians Eratosthenes and Hipparchus, and nobody since has come up with anything better.

Although meridians are commonly referred to as halves of great circles, the great circles that define the meridians are actually ellipses with a circumference of 40,008 kilometers. It is definitely *not* a coincidence that this length is very close to 40,000 kilometers. The meter was originally defined in the late 18th century to be one ten-millionth of the distance from the north pole to the equator, and it remains a handy way to remember the approximate size of the Earth.

The equirectangular projection

The bitmap that has been used in the 3D images of the Earth shown above is from the NASA/Jet Propulsion Laboratory web page http://maps.jpl.nasa.gov/earth.html, and is used courtesy of NASA/JPL-Caltech. Here is that bitmap not wrapped around a 3D sphere and overlaid with a grid of longitudes and latitudes:



This is a type of map known as an *equirectangular projection* because the meridians and parallels are equally spaced to resemble a rectangular coordinate system. The term *projection* originates from a graphical technique of first projecting the surface of a sphere to a cylinder, and then unfurling it into a flat surface.

The equirectangular projection is a highly distorted image of the Earth. The map is reasonable near the equator but as you move further north or south, areas are stretched horizontally. The worst distortions are at the poles where single points at latitudes of 90 degrees and –90 degrees are stretched to the full width of the map. When the map is used to cover a 3D sphere, the stretched-out areas are compressed so the map becomes undistorted in 3D space.

The circumference of the Earth is very close to 40,000 kilometers, so dividing that by 360 degrees results in 111 kilometers. One degree of latitude is thus approximately 111 kilometers or 69 miles, but it varies somewhat due to the bulging of the Earth at the equator, from about 110.6 km at the equator to 111.7 km at the poles.

The length of one degree of longitude isn't quite as simple because the circumference of the parallels varies by the latitude. One degree of longitude ranges from 111 kilometers at the equator, to about 79 kilometers (49 miles) at latitudes of 45 degrees or –45 degrees, to zero at the poles. In general, one degree of longitude is about 111 kilometers multiplied by the cosine of the latitude.

A degree is divided into 60 minutes, and each minute is divided into 60 seconds. At the equator, one second of longitude or latitude is about 31 meters or 101 feet.

A particular geographic location on the Earth is generally indicated with the latitude first, followed by the longitude, with positive values of degrees, minutes, and seconds, and the letters N and S to indicate north or south of the equator, and E and W for east or west of the prime meridian. This is sometimes known as DSM (degree-minute-second) format:

47°38′23″N 122°7′42″W

That's the location of Microsoft Corporation in Redmond, Washington.

In computer applications—and to facilitate mathematical calculations—the geographic location can alternatively be expressed as a pair of floating-point numbers converted to fractions of a degree and with negative values indicating southern latitudes or western longitudes:

47.639722 –122.128333

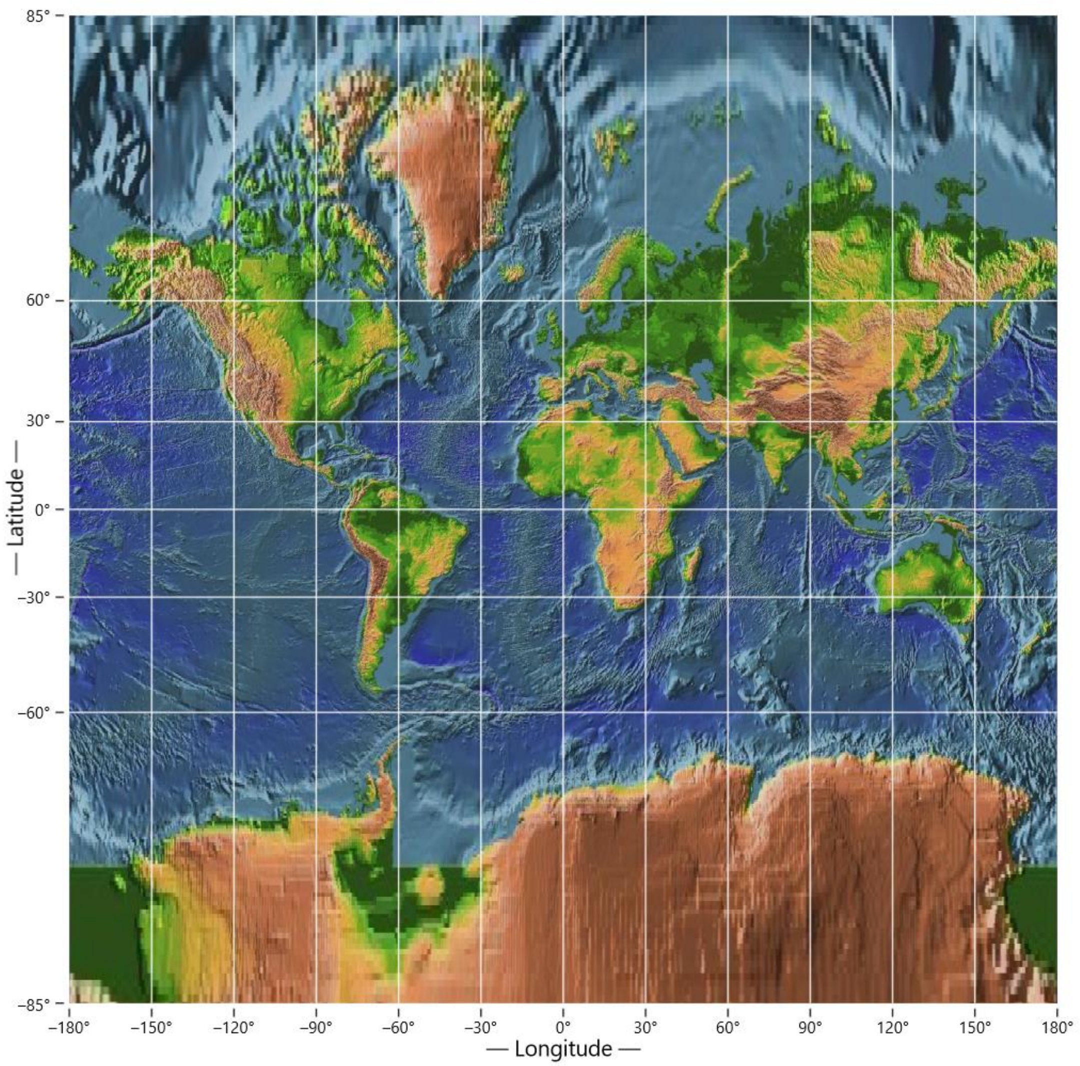
The Mercator projection

There is no way to project the surface of a sphere onto a flat map without distortion, but some types of distortions might be preferable to others. The equirectangular projection shown earlier maintains correct distances and relationships along meridians at the cost of stretching areas horizontally as you move further from the equator. The shape of Greenland is particularly distorted.

Some map projections attempt to approximate the sphere by bending meridians or parallels (or both) so they become curves rather than straight lines, but, as you might imagine, these projections would be awkward for computer mapping applications.

Historically, the most popular type of map has been the Mercator projection, developed by a Flemish geographer in the 16th century. In the Mercator projection, the meridians and parallels are straight lines, which means that areas near the poles are stretched horizontally just as in the equirectangular projection. However, the Mercator projection also stretches these areas vertically as well.

Here's the Mercator projection extending to latitudes of 85 degrees in the northern and southern hemispheres:



The Mercator projection might at first seem to introduce even more distortion than the equirectangular projection, and in one sense that's certainly true. Notice how the 15 degrees of latitude between 60 and 85 degrees occupies considerably more vertical space than the 60 degrees of latitude between 0 and 60 degrees.

However, the extent of the vertical stretching is computed to match exactly the horizontal stretching, so that all areas on the map maintain the same shape as on the globe. Directions are also consistent: For example, the direction of north-east is always a diagonal line from lower-left to upper-right, 45 degrees from the horizontal. In any local area, the projection is quite accurate. Such a projection is said to be *conformal*, and this is often a desirable characteristic of a map projection.

But obviously the Mercator projection greatly exaggerates the size of areas near the poles. In this map, Greenland has the correct shape but is has grown enormously in size. In reality, Greenland is less than 30% of the area of Australia. But the shapes are correct. Anywhere in the world, a square grid of streets would appear on the map as square rather than rectangular or curved in some way.

The Mercator projection can't actually extend all the way to the poles because the mathematics would result in infinite vertical stretching at that point. Many maps using the Mercator projection are truncated at latitudes less than 85 degrees, and some even truncate the southern hemisphere at 60 degrees.

Map services and tiles

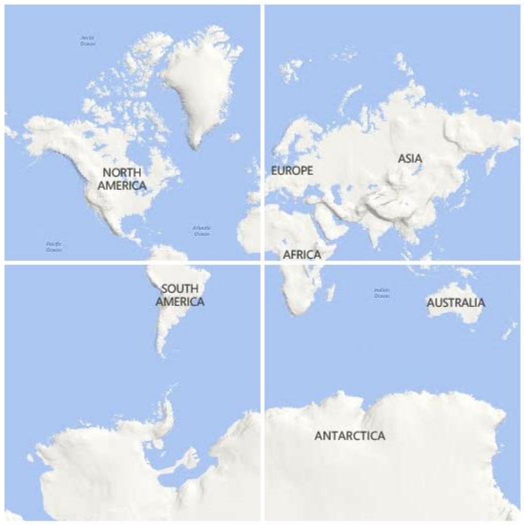
The Xamarin.Forms `Map` element is rendered by the iOS `MKMapView`, the Android `MapView`, and the Universal Windows Platform `MapControl`. These classes make use of corresponding map servers called Apple Maps, Google Maps, and Bing Maps.

These three map servers all work similarly. They all use a somewhat altered version of the Mercator projection dubbed *Web Mercator*. For performance purposes, Web Mercator fudges the distinction between spherical coordinates and ellipsoid coordinates, and consequently the results are slightly non-conformal. Despite its widespread use in these popular mapping services, the Web Mercator projection remains rather controversial among experts in map projections and geodesy.

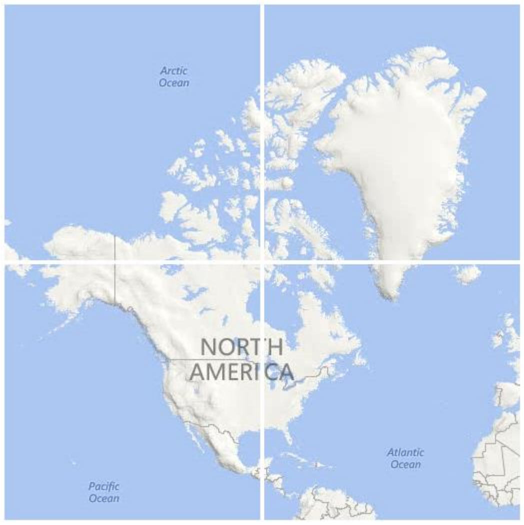
If a Mercator or Web Mercator map is extended to 85.05 degrees north and south—such as in the illustration above—then the map becomes square. (In contrast, the equirectangular projection of the whole Earth is twice as wide as it is tall.) This square shape makes it convenient for mapping services.

The various map servers deliver bitmap tiles to their clients. (Android and Windows applications that need access to these actual tiles can obtain them, but this is not the case with Apple Maps.) These bitmap tiles are 256-pixels square, and vary by longitude, latitude, and zoom level.

An unzoomed Web Mercator image of the Earth is covered with four tiles. These are images of actual tiles available from the Bing Maps service:



For the next higher zoom level, each of these areas can be rendered with four more 256-pixel square tiles:



At this zoom level, the whole Earth is covered by 16 tiles. In a level 3 zoom, the whole Earth requires 64 tiles, and so forth.

If the map server goes as high as a level 20 zoom, a trillion tiles cover the Earth—1 million horizontally and 1 million vertically, with a resolution at the equator of about 6 inches per pixel.

The job of a map control such as the native controls underlying the Xamarin.Forms `Map` view is to download and assemble these tiles to provide a seamless user experience as the user pans and zooms the map.

With this background, let's begin by obtaining the latitude and longitude of a user's phone.

Getting the user's location

Xamarin.Forms does not include a facility for an application to obtain the phone's geographic location. Such a feature is often informally called GPS, which refers to the Global Positioning System that allows phones to receive radio transmissions from dedicated satellites to determine the phone's location.

When used with mobile devices, however, GPS has some problems: it works much better outdoors than indoors, and it is processor and hardware intensive, which means it tends to drain the phone's battery. For these reasons, today's mobile platforms supplement GPS with alternative techniques to obtain the phone's location involving WiFi access points and cell phone towers.

The various platforms supported by Xamarin.Forms all support APIs for applications to obtain the geographic location of the phone. The platform APIs allow specifying various degrees of precision in the location data obtained, and how often it should be updated. In this way, the application can balance its own needs while minimizing power consumption. The dependency service presented here is fairly simple, however, and doesn't allow the application any leeway. If you need more specificity, you can build on the code shown here.

The location tracker API

A dependency service to track the geographic location of the phone is included in the **Xamarin.FormsBook.Platform** library. This library was first introduced in Chapter 20, "Async and file I/O." As you might recall, this library includes seven projects, one project for the platform-independent code and the others for the various Xamarin.Forms platforms, including a shared project for the Windows Runtime platforms.

This new dependency service begins with a simple structure in the **Xamarin.FormsBook.Platform** project to encapsulate a latitude and longitude of a geographic location:

```
namespace Xamarin.FormsBook.Platform
{
    public struct GeographicLocation
    {
        public GeographicLocation(double latitude, double longitude) : this()
        {
            // Normalize values.
            Latitude = latitude % 90;
            Longitude = longitude % 180;
        }

        public double Latitude { private set; get; }

        public double Longitude { private set; get; }

        public override string ToString()
        {
            return String.Format("{0}{1} {2}{3}", DMS(Math.Abs(Latitude)),
                                                  Latitude >= 0 ? "N" : "S",
                                                  DMS(Math.Abs(Longitude)),
                                                  Longitude >= 0 ? "E" : "W");
        }

        string DMS(double decimalDegrees)
        {
            int degrees = (int)decimalDegrees;
            decimalDegrees -= degrees;
            decimalDegrees *= 60;
            int minutes = (int)decimalDegrees;
            decimalDegrees -= minutes;
            decimalDegrees *= 60;
            int seconds = (int)Math.Round(decimalDegrees);

            // Fix rounding issues.
            if (seconds == 60)
            {
                seconds = 0;
                minutes += 1;

                if (minutes == 60)
                {
                    minutes = 0;
                    degrees += 1;
                }
            }

            return String.Format("{0}°{1}′{2}″", degrees, minutes, seconds);
        }
    }
}
```

The bulk of this structure supports a `ToString` override that uses a method named `DMS` to display the latitude and longitude with degrees, minutes, and seconds. Sometimes the `seconds` value rounds to a value of 60, which explains the code right before the `String.Format` call.

The location-tracking API implemented by this dependency service is defined by the following interface consisting of two methods and an event:

```
namespace Xamarin.FormsBook.Platform
{
    public interface ILocationTracker
    {
        event EventHandler<GeographicLocation> LocationChanged;

        void StartTracking();

        void PauseTracking();
    }
}
```

The application calls StartTracking when it wants to receive location data. Thereafter, the application receives LocationChanged events when a new GeographicLocation value is available. The application can temporarily stop these events with a call to PauseTracking. In a multipage application, it's likely you'll want to call StartTracking in the OnAppearing override for the page, and PauseTracking in the OnDisappearing override.

The ILocationTracker interface now needs to be implemented in all three platforms.

The iOS location manager

The iOS implementation of the location tracker uses the CLLocationManager class. For iOS 8 and above, a call to RequestWhenInUseAuthorization or RequestAlwaysAuthorization is required, depending on whether the application uses location services in an obvious way, or whether it uses them in a way that might not be obvious to the user. The iOS version of LocationTracker calls RequestWhenInUseAuthorization.

If you want to enhance the code shown below, you can also specify certain criteria using properties such as DesiredAccuracy and DistanceFilter. The LocationsUpdated event indicates when the location data has changed, either because the user has moved location or because more accurate information is available.

Notice the required Dependency attribute near the top of the file:

```
using System;
using CoreLocation;
using UIKit;
using Xamarin.Forms;

[assembly: Dependency(typeof(Xamarin.FormsBook.Platform.iOS.LocationTracker))]

namespace Xamarin.FormsBook.Platform.iOS
{
    public class LocationTracker : ILocationTracker
    {
        CLLocationManager locationManager;

        public event EventHandler<GeographicLocation> LocationChanged;

        public LocationTracker()
        {
            locationManager = new CLLocationManager();

            // Request authorization for iOS 8 and above.
            if (UIDevice.CurrentDevice.CheckSystemVersion(8, 0))
            {
                locationManager.RequestWhenInUseAuthorization();
            }

            locationManager.LocationsUpdated +=
                (object sender, CLLocationsUpdatedEventArgs args) =>
                {
                    CLLocationCoordinate2D coordinate = args.Locations[0].Coordinate;
                    EventHandler<GeographicLocation> handler = LocationChanged;

                    if (handler != null)
                    {
                        handler(this, new GeographicLocation(coordinate.Latitude,
                                                             coordinate.Longitude));
                    }
                };
        }
        public void StartTracking()
        {
            if (CLLocationManager.LocationServicesEnabled)
            {
                locationManager.StartUpdatingLocation();
            }
        }

        public void PauseTracking()
        {
            locationManager.StopUpdatingLocation();
        }
    }
}
```

The `StartTracking` and `PauseTracking` methods are implemented with `CLLocationManager` calls to `StartUpdatingLocation` and `StopUpdatingLocation`.

The Android location manager

Android supports two different approaches to obtaining location data: the `LocationManager` class and the `FusedLocationProviderApi` that is part of Google Play Services. To keep the code as simple as possible, the implementation here uses `LocationManager`.

The `LocationManager` constructor requires the Android `Activity` object. As you might recall, each of the platform projects in the **Xamarin.FormsBook.Platform** library contains a class named `Toolkit` with a static method named `Init`. The Android version of `Init` requires that an `Activity` object be passed as an argument, and the class makes it available as a public property.

The `LocationManager` class supports three different methods for obtaining location information, referred to as *providers*. These three providers are identified by the text strings “gps,” “network,” and “passive.” Generally, you call the `GetBestProvider` method of `LocationManager` with a `Criteria` structure to indicate the accuracy you want, and either “gps” or “network” is returned. This string is passed to the `RequestLocationUpdates` method to obtain updated location information.

The problem, however, is that this same provider is then used for the duration of the application, even though a different provider might be more appropriate, for example, if you move from indoors to outdoors. One solution (illustrated in the code below) is to call `RequestLocationUpdates` for both “gps” and “network” (which are available as constants in the `LocationManager` class) to get the best of both worlds. (For more serious location work you’ll probably want to switch to the fused location provider in Google Play Services, which switches among providers more efficiently to conserve battery life.)

The `RequestLocationUpdates` method also requires minimum update time and minimum update distance criteria. The last argument is a class that implements the `ILocationListener` interface, which defines four methods for callback purposes. For this reason, this `LocationTracker` tracker class implements both the `ILocationTracker` interface defined in **Xamarin.FormsBook.Toolkit** as well as the Android `ILocationListener` interface:

```
using System;
using System.Collections.Generic;

using Android.App;
using Android.Content;
using Android.Locations;
using Android.OS;
using Android.Runtime;

using Xamarin.Forms;

[assembly: Dependency(typeof(Xamarin.FormsBook.Platform.Android.LocationTracker))]

namespace Xamarin.FormsBook.Platform.Android
{
    public class LocationTracker : Java.Lang.Object, ILocationTracker, ILocationListener
    {
        LocationManager locationManager;

        public event EventHandler<GeographicLocation> LocationChanged;

        public LocationTracker()
        {
            Activity activity = Toolkit.Activity;

            if (activity == null)
                throw new InvalidOperationException(
                    "Must call Toolkit.Init before using LocationProvider");

            locationManager =
                activity.GetSystemService(Context.LocationService) as LocationManager;
        }

        // Two methods to implement ILocationProvider (the dependency service interface).
        public void StartTracking()
        {
            IList<string> locationProviders = locationManager.AllProviders;

            foreach (string locationProvider in locationProviders)
            {
                locationManager.RequestLocationUpdates(locationProvider, 1000, 1, this);
            }
        }
        public void PauseTracking()
        {
            locationManager.RemoveUpdates(this);
        }

        // Four methods to implement ILocationListener (the Android interface).
        public void OnLocationChanged(Location location)
        {
            EventHandler<GeographicLocation> handler = LocationChanged;

            if (handler != null)
            {
                handler(this, new GeographicLocation(location.Latitude,
                                                     location.Longitude));
            }
        }

        public void OnProviderDisabled(string provider)
        {
        }

        public void OnProviderEnabled(string provider)
        {
        }

        public void OnStatusChanged(string provider, [GeneratedEnum] Availability status,
                                    Bundle extras)
        {
        }
    }
}
```

The Windows Runtime geo locator

For the Windows Runtime platforms, you can use the `Geolocator` class. The only prerequisite is setting a `ReportInterval` property in milliseconds. You can begin tracking the location by attaching a handler for the `PositionChanged` event, and pause tracking by detaching that handler.

The `LocationTracker` class is in the shared project **Xamarin.FormsBook.Platform.WinRT** so it is used by the Windows 8.1, Windows Phone 8.1, and Universal Windows Platform projects. For the UWP, it's recommended that the application call the `Geolocator.RequestAccessAsync` method in a class that is not present in the Windows 8.1 and Windows Phone 8.1 platforms. Calling this method is not strictly necessary, but it's included in the shared code using preprocessor directives:

```
using System;
using Windows.Devices.Geolocation;

#if WINDOWS_UWP
using Windows.Foundation;
#endif

using Xamarin.Forms;
```

```
[assembly: Dependency(typeof(Xamarin.FormsBook.Platform.WinRT.LocationTracker))]

namespace Xamarin.FormsBook.Platform.WinRT
{
    public class LocationTracker : ILocationTracker
    {
        Geolocator geolocator;
#if WINDOWS_UWP
        bool isTracking;
#endif
        public LocationTracker()
        {
            geolocator = new Geolocator();
            geolocator.ReportInterval = 1000;
        }

        public event EventHandler<GeographicLocation> LocationChanged;

        public void StartTracking()
        {
#if WINDOWS_UWP
            IAsyncOperation<GeolocationAccessStatus> task = Geolocator.RequestAccessAsync();

            task.Completed = (asyncop, status) =>
            {
                GeolocationAccessStatus accessStatus = asyncop.GetResults();
                if (accessStatus == GeolocationAccessStatus.Allowed)
                {
                    geolocator.PositionChanged += OnGeolocatorPositionChanged;
                    isTracking = true;
                }
            };
#else
           geolocator.PositionChanged += OnGeolocatorPositionChanged;
#endif
        }

        public void PauseTracking()
        {
#if WINDOWS_UWP
            if (isTracking)
            {
                geolocator.PositionChanged -= OnGeolocatorPositionChanged;
            }
#else
            geolocator.PositionChanged -= OnGeolocatorPositionChanged;
#endif
        }

        void OnGeolocatorPositionChanged(Geolocator sender, PositionChangedEventArgs args)
        {
            BasicGeoposition coordinate = args.Position.Coordinate.Point.Position;

            Device.BeginInvokeOnMainThread(() =>
            {
                EventHandler<GeographicLocation> handler = LocationChanged;

                if (handler != null)
                {
                    handler(this, new GeographicLocation(coordinate.Latitude,
                                                         coordinate.Longitude));
                }
            });
        }
    }
}
```

Watch out! The `PositionChanged` handler is called in a secondary thread of execution, so any code that uses this information to access the user interface needs to switch back to the user-interface thread. For convenience, this handler performs that switch. Because this class is instantiated from the Xamarin.Forms PCL, it can make use of the `Device.BeginInvokeOnMainThread` method for this thread switch.

Displaying the phone's location

The `LocationTracker` classes are put to use in the **WhereAmI** program. The **WhereAmI** solution contains links to all the projects of the **Xamarin.FormsBook.Platform** solution, and each of the projects in **WhereAmI** contains a reference to the corresponding project in **Xamarin.FormsBook.Platform**.

WhereAmI displays your latitude and longitude in a text format, and also positions a tiny `EllipseView` element (from Chapter 27, "Custom renderers") on the NASA/JPL map of the Earth. Here's the XAML file:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:WhereAmI"
             xmlns:platform=
                 "clr-namespace:Xamarin.FormsBook.Platform;assembly=Xamarin.Formsbook.Platform"
             x:Class="WhereAmI.WhereAmIPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="0, 20, 0, 0" />
    </ContentPage.Padding>

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <Grid Grid.Row="0"
              HorizontalOptions="Center"
              VerticalOptions="Center">
            <!-- Bitmap from http://maps.jpl.nasa.gov/earth.html,
                 and is used courtesy of NASA/JPL-Caltech -->
            <Image x:Name="image"
                   Source="{local:ImageResource WhereAmI.Images.ear0xuu2.jpg}"
                   HorizontalOptions="Center"
                   VerticalOptions="Center" />

            <AbsoluteLayout x:Name="absLayout"
                            WidthRequest="{Binding Source={x:Reference image},
                                                   Path=Width}"
                            HeightRequest="{Binding Source={x:Reference image},
                                                    Path=Height}"
                            SizeChanged="OnAbsoluteLayoutSizeChanged"
                            HorizontalOptions="Center"
                            VerticalOptions="Center">

                <platform:EllipseView x:Name="ellipse"
                                      Color="White"
                                      WidthRequest="5"
                                      HeightRequest="5"
                                      TranslationX="-2.5"
                                      TranslationY="-2.5" />
            </AbsoluteLayout>
        </Grid>

        <Grid Grid.Row="1"
              Margin="10, 20"
              HorizontalOptions="Center">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>

            <Label Text="Location: "
                   HorizontalTextAlignment="End"
                   Grid.Row="0"
                   Grid.Column="0" />

            <Label x:Name="locationLabel"
                   Grid.Row="0"
                   Grid.Column="1" />

            <Label Text="Time stamp: "
                   HorizontalTextAlignment="End"
                   Grid.Row="1"
                   Grid.Column="0" />

            <Label x:Name="timestampLabel"
                   Grid.Row="1"
                   Grid.Column="1" />
        </Grid>
    </Grid>
</ContentPage>
```

The top row of the main Grid contains the Image element for the map and an overlaying AbsoluteLayout with bindings to ensure that it remains the same size as the Image. The EllipseView is a child of the AbsoluteLayout.

The bottom row of the Grid contains the Label elements used to display the latitude and longitude, and the time the information was obtained.

The code-behind file starts the location tracker going in its constructor, and updates the text display whenever new information is available. The position of the EllipseView on the map must also be updated whenever the AbsoluteLayout changes size, which happens when the phone is shifted from portrait to landscape mode:

```
public partial class WhereAmIPage : ContentPage
{
    GeographicLocation location;

    public WhereAmIPage()
    {
        InitializeComponent();

        ILocationTracker locationTracker = DependencyService.Get<ILocationTracker>();
        locationTracker.LocationChanged += OnLocationTrackerLocationChanged;
        locationTracker.StartTracking();
    }

    void OnLocationTrackerLocationChanged(object sender, GeographicLocation args)
    {
        locationLabel.Text = args.ToString();
        timestampLabel.Text = DateTime.Now.ToString("h:mm:ss.FFF");

        // Update dot on map.
        location = args;
        UpdateLocationDot(args);
    }

    void OnAbsoluteLayoutSizeChanged(object sender, EventArgs args)
    {
        // Update dot on map.
        UpdateLocationDot(location);
    }

    void UpdateLocationDot(GeographicLocation location)
    {
        double x = (location.Longitude + 180) * absLayout.Width / 360;
        double y = (90 - location.Latitude) * absLayout.Height / 180;

        AbsoluteLayout.SetLayoutBounds(ellipse,
            new Rectangle(x, y, AbsoluteLayout.AutoSize,
                                AbsoluteLayout.AutoSize));
    }
}
```

Sometimes it's hard to tell if the latitude and longitude are being updated solely from the formatted display of the values, so the displayed text also includes the update time with milliseconds.

The required overhead

As discussed earlier, the **WhereAmI** project itself contains a reference to the **Xamarin.Forms-Book.Platform** project, and each of the application projects also contains a reference to the corresponding platform library in the **Xamarin.FormsBook.Platform** solution.

In addition, each of the application projects must call the static `Toolkit.Init` method in the corresponding library. This is to ensure that the library is loaded with the application package. Each of the application programs in **WhereAmI** contains a call to `Toolkit.Init` following the standard call to `Forms.Init`. This occurs in the `AppDelegate` class in the iOS project, and in the App.xaml.cs files in the three Windows and Windows Phone projects. The Android call to `Toolkit.Init` occurs in the `MainActivity` class. The method is also responsible for saving the `Activity` object for later use:

```
public class MainActivity : global::Xamarin.Forms.Platform.Android.FormsApplicationActivity
{
    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);

        global::Xamarin.Forms.Forms.Init(this, bundle);

        Xamarin.FormsBook.Platform.Android.Toolkit.Init(this, bundle);

        LoadApplication(new App());
    }
}
```

Some additional overhead is required before using any location services. It is considered an invasion of privacy if a computer application obtains the user's location without first getting the user's permission. For this reason, all the platforms require the application to be flagged as requiring this information.

Location permission for iOS

The iOS version of the `LocationTracker` class calls the `RequestWhenInUseAuthorization` method of the `CLLocationManager` class to obtain permission from the user. This causes an alert box to be displayed with some text that you specify in the Info.plist file. You can edit that file as XML, and insert something like this between the `<dict>` tags:

```
<key>NSLocationWhenInUseUsageDescription</key>
<string>The application uses your location to display your coordinates</string>
```

The text you specify follows the question "Allow [your app] to access your location while you use the app?" so it should be an explanation (of sorts) how the application uses the location.

If your program instead calls `RequestAlwaysAuthorization` to use `CLLocationManager`, you must specify some text associated with the `NSLocationAlwaysUsageDescription` key.

Location permissions for Android

In Visual Studio, select **Properties** for the Android application project. In the properties screen, select the **Android Manifest** tab. In the **Required permissions** section check **ACCESS_FINE_LOCATION**.

In Xamarin Studio, display **Project Options** for the Android project. Choose the Android application tag, and in the **Required permissions** section, check **AccessFineLocation**.

In either case, this permission appears in the AndroidManifest.xml file (located in the **Properties** folder) as the following child of the `manifest` element:

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

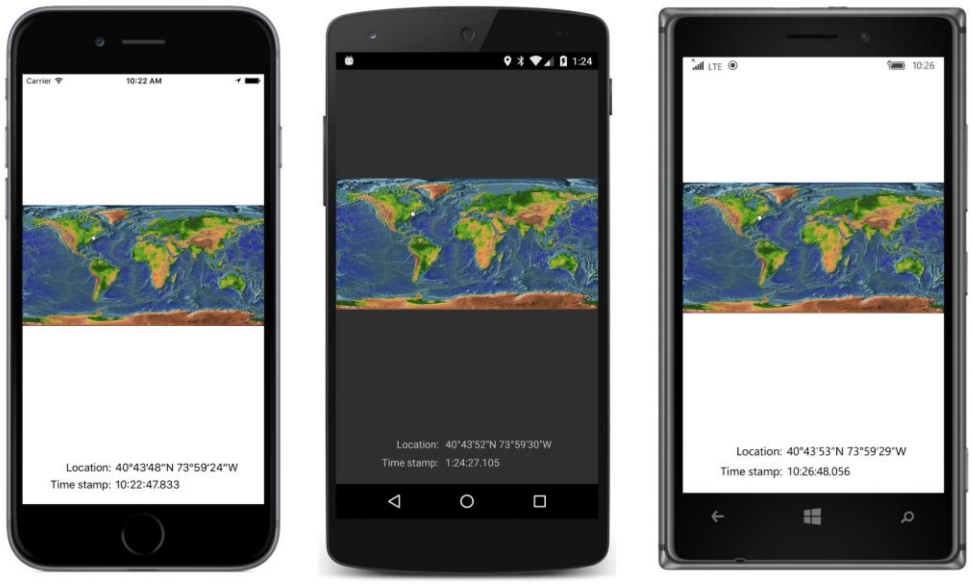
Location permissions for the Windows Runtime

In the Windows 8.1, Windows Phone 8.1, and Universal Windows Platform projects, display the project **Properties** dialog and select the **Application** tab. Click the **Package Manifest** button. This causes another dialog to be displayed with tabs across the top. Select the **Capabilities** tab. In the **Capabilities** list, make sure that **Location** is checked.

This selection shows up in the Package.appxmanifest file as a child of the `Capabilities` section of the `Package` root tag:

```
<DeviceCapability Name="location" />
```

With all that out of the way, you can run the program. Once the device obtains the phone's location, the information is displayed at the bottom in text and on the map as a small white dot:

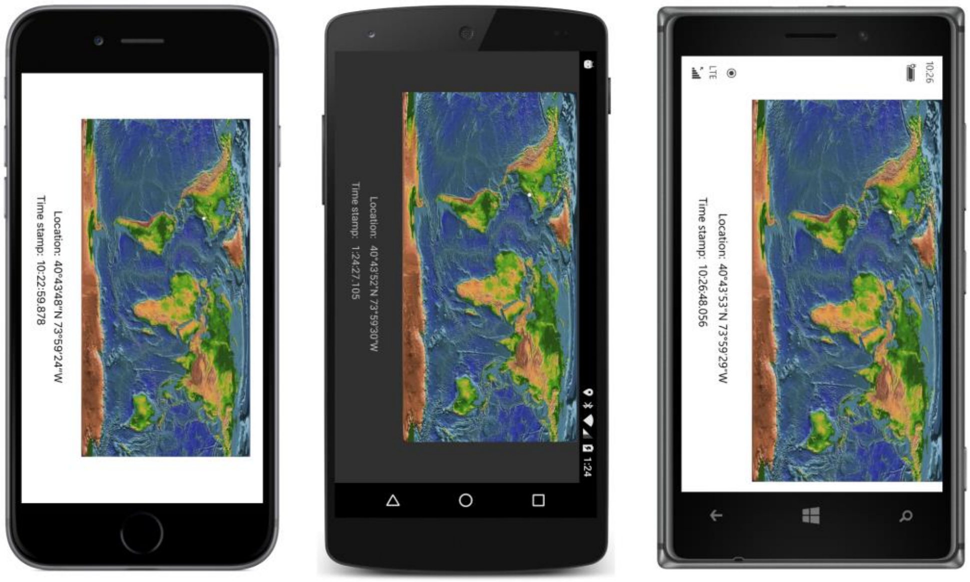


```
Location: 40°43'48"N 73°59'24"W
Time stamp: 10:22:47.833
```

```
Location: 40°43'52"N 73°59'30"W
Time stamp: 1:24:27.105
```

```
Location: 40°43'53"N 73°59'29"W
Time stamp: 10:26:48.056
```

The map is rendered larger in landscape mode:


Location: 40°43'48"N 73°59'24"W
Time stamp: 10:22:59.878

Location: 40°43'52"N 73°59'30"W
Time stamp: 1:24:27.105

Location: 40°43'53"N 73°59'29"W
Time stamp: 10:26:48.056

Of course, most users want to see a map with a little more detail! Let's do it.

Working with Xamarin.Forms.Maps

Now it's time to start working with the Xamarin.Forms `Map` view. This isn't quite as easy as working with the Xamarin.Forms `Button` and `Slider` because `Map` is implemented in an entirely different set of libraries along with several related classes, structures, and enumerations. In any application that uses the `Map` view, you'll need to install a separate NuGet package named **Xamarin.Forms.Maps**.

In addition, the Android and Windows platforms require you to enable the application for accessing the platform's map services by obtaining and specifying an authorization key.

If your application also needs to determine the user's location, you'll want to use the **LocationTracker** dependency service in your program, in which case you'll need to add the various projects of the **Xamarin.FormsBook.Platform** solution to your application solution as discussed earlier.

This means that using maps is not trivial. For that reason, the remainder of this chapter focuses on a single solution named **MapDemos** that uses navigable pages to demonstrate various aspects of working with maps.

The NuGet package

You'll need to install the **Xamarin.Forms.Maps** package in the PCL and application projects of any solution that uses the `Map` element.

In Visual Studio, right-click the solution name in the **Solution Explorer** and select **Manage NuGet Packages for Solution**. Search for **Xamarin.Forms.Maps**. Select the PCL and the application projects, and click the **Install** button.

In Xamarin Studio, for each project in the solution, select the **Add NuGet Packages** item from the **Project** menu. Search for **Xamarin.Forms.Maps**. Select it and click **Add Package**.

In either case, the assemblies **Xamarin.GooglePlayServices.Base** and **Xamarin.GooglePlay-Services.Maps** are also automatically installed in the Android project.

The version numbers of the **Xamarin.Forms** and **Xamarin.Forms.Maps** packages are synchronized, and you should ensure that the version numbers of both packages are the same. If not, update one or the other so that they match.

Initializing the Maps package

As you know, every Xamarin.Forms platform project has a call to the static `Forms.Init` method. This call is in different files in the various platforms:

- iOS: in the AppDelegate.cs file in the `FinishedLaunching` override.
- Android: in the MainActivity.cs file in the `OnCreate` override.
- Windows 8.1 and UWP: in App.xaml.cs in the `OnLaunched` override.

After the **Xamarin.Forms.Maps** package is installed, you'll need to supplement the normal call to `Forms.Init` with a call to the `FormsMaps.Init` method in the `Xamarin` namespace.

In the AppDelegate.cs file in the iOS project, it looks like this:

```
global::Xamarin.Forms.Forms.Init();
global::Xamarin.FormsMaps.Init();
```

In the Android project, both `Init` methods require parameters:

```
global::Xamarin.Forms.Forms.Init(this, bundle);
global::Xamarin.FormsMaps.Init(this, bundle);
```

In all three Windows Runtime projects, the `FormsMaps.Init` call requires an authorization key to use Bing maps. After you obtain that key, you'll need to include it as an argument:

```
Xamarin.Forms.Forms.Init(e);
Xamarin.FormsMaps.Init("MAP_KEY");
```

If you've downloaded the **MapDemos** solution from the GitHub repository, you'll need to replace that text with your own authorization key for using Bing Maps.

In addition, you'll also need an authorization key for the Android project to use Google Maps. The next section describes these requirements.

Enabling map services

You've already seen how you can enable a project for location services. Because the `Map` view can obtain and display the user's location, you should also include that same enabling if you're using the **Xamarin.Forms.Maps** assembly.

In addition, the Android and Windows platforms require that you obtain an authorization key for your application. You'll need to get keys of your own to run the **MapDemos** sample!

Enabling iOS maps

An iOS application can use the Xamarin.Forms `Map` view without any special authorization or permission. However, the `Map` can obtain the user's location, so you'll want to add the following two lines to the Info.plist file within the `<dict>` tags:

```
<key>NSLocationWhenInUseUsageDescription</key>
<string>The application uses your location to display it on a map</string>
```

This is the same as when you use iOS location services apart from the map.

Enabling Android maps

Android requires that the application contain an embedded authorization key for using the Android map or Google Map services. Go to the following website:

https://developers.google.com/maps/documentation/android-api/

Press the **Get a Key** button. You'll see a brief outline of the three steps. If you've never used this system before, you'll want to select the **Create a project** option from the dropdown and press **Continue**. This is a page labeled **Credentials**.

In the **Name** field, enter a name that identifies the application for you, or simply leave it as **Android key 1**. You'll also see a button labeled **+ Add package name and fingerprint,** but that is optional and not required for simply using the Android `MapView` in a sample application. Just click the **Create** button and you'll get a key of 39 seemingly random characters.

In the AndroidManifest.xml file, find (or add) an `application` section under the root `manifest` element, and add a `meta-data` element like so:

```
<manifest …>
    …
    <application …>
        <meta-data android:name="com.google.android.maps.v2.API_KEY"
                   android:value="MAP_KEY" />
    </application>
    …
</manifest>
```

Replace `MAP_KEY` with the key you obtained from the Google website.

It's recommended that you enable the following permissions, either in Visual Studio or Xamarin Studio by checking items in the **Required permissions** list in the **Android Manifest** section of project properties (or options), or by inserting the following within the `manifest` tags of the AndroidManifest.xml file:

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_LOCATION_EXTRA_COMMANDS" />
<uses-permission android:name="android.permission.ACCESS_MOCK_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name="android.permission.INTERNET" />
```

Enabling Windows Runtime maps

To use maps in the Windows Runtime projects, do a web search for the **Bing Maps SDK for Windows 8.1 Store apps** and install it.

To obtain an authorization key, visit the Bing Maps Dev Center at https://www.bingmapsportal.com/. Sign in with your Windows Live ID. In the tab labeled **My account**, pick **My Keys**. You'll see a link to create a new key. All that's required is the **Application name** (for example **MapDemos**), a **Key type** (choose **Basic**), and an **Application type**. If you pick **Universal Windows App**, you can use the same key with the other Windows and Windows Phone projects.

The key is over 100 characters in length. Insert it in place of `MAP_KEY` in the following calls in App.xaml.cs in the Windows App.xaml.cs files:

```
Xamarin.FormsMaps.Init("MAP_KEY");
```

You'll also want to enable **Location** capabilities as described earlier.

The unadorned map

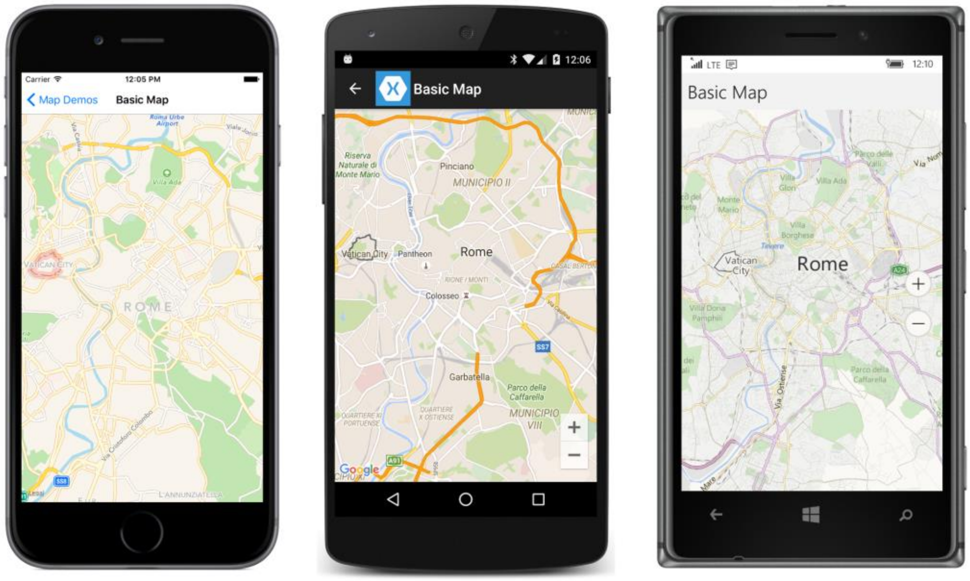
The **MapDemos** program consists of a home page named `MapDemosHomePage` that allows you to navigate to various pages that demonstrate progressively more sophisticated ways of using the `Map` view and related classes.

Once you've installed the NuGet package for **Xamarin.Forms.Maps**, obtained the necessary authorization keys, added permissions and calls to `Xamarin.FormsMaps.Init`, displaying a map is fairly easy. The only slight complication is a new XML namespace declaration for the **Xamarin.Forms.Maps** assembly and namespace. Here is the `BasicMapPage` class in the **MapDemos** project:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:maps="clr-namespace:Xamarin.Forms.Maps;assembly=Xamarin.Forms.Maps"
             x:Class="MapDemos.BasicMapPage"
             Title="Basic Map">

    <maps:Map />
```

```
</ContentPage>
```

When running **MapDemos**, select the **Basic Map** option on the home page. By default, the map displays the homes of philosophers Cicero, Lucretius, and Epictetus:



You can use your finger to move the map, or a pair of fingers to pinch the map. Pinching causes the map to zoom in or out. With a twist of two fingers you can also rotate the map. Notice that both Android and Windows 10 Mobile include a pair of buttons on the right side of the map to perform zooming operations.

To disable the horizontal and vertical movement of the map, you can set the `HasScrollEnabled` property of `Map` to `false`. This property has no effect on Windows 8.1 and Windows Phone 8.1.

To disable zooming, set `HasZoomEnabled` to `false`. This property also has no effect on Windows 8.1 and Windows Phone 8.1. On the UWP platforms, setting the property to `false` prevents scaling the map using pinch gestures but still allows the map to be scaled using the circled **+** and **–** buttons. On Android, the scaling buttons disappear when the property is set to `false`.

On iOS and Android, setting both these properties to `false` prevents the map from being scrolled or zoomed, but you can still use two fingers to rotate it.

Streets and Terrain

The `Map` has a `MapType` property of type `MapType`, an enumeration with three members:

- `Street`
- `Satellite`

- Hybrid

The default value is Street.

MapType is backed by a bindable property so it can be the target of a data binding, which means that you can use the RadioButtonManager class in the **Xamarin.FormsBook.Toolkit** library to manage three radio buttons that let you switch between these three options.

For that reason, the **MapDemos** solution includes the **Xamarin.FormsBook.Toolkit** library project, and the **MapDemos** project includes a class for rendering the three radio buttons:

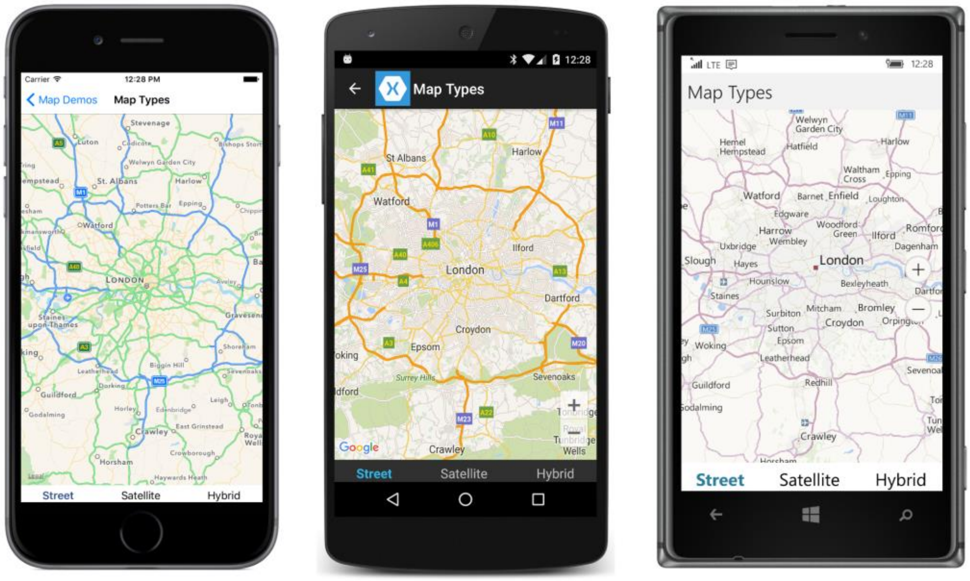
```
<ContentView xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MapDemos.MapTypeRadioButton">
    <Label Text="{Binding Name}"
       FontSize="Medium">
        <Label.GestureRecognizers>
            <TapGestureRecognizer Command="{Binding Command}"
                                  CommandParameter="{Binding Value}"/>
        </Label.GestureRecognizers>

        <Label.Triggers>
            <DataTrigger TargetType="Label"
                         Binding="{Binding IsSelected}"
                         Value="True">
                <Setter Property="TextColor" Value="Accent" />
                <Setter Property="FontAttributes" Value="Bold" />
            </DataTrigger>
        </Label.Triggers>
    </Label>
</ContentView>
```

The DataTrigger causes the selected item to be boldfaced and colored with the platform's accent color.

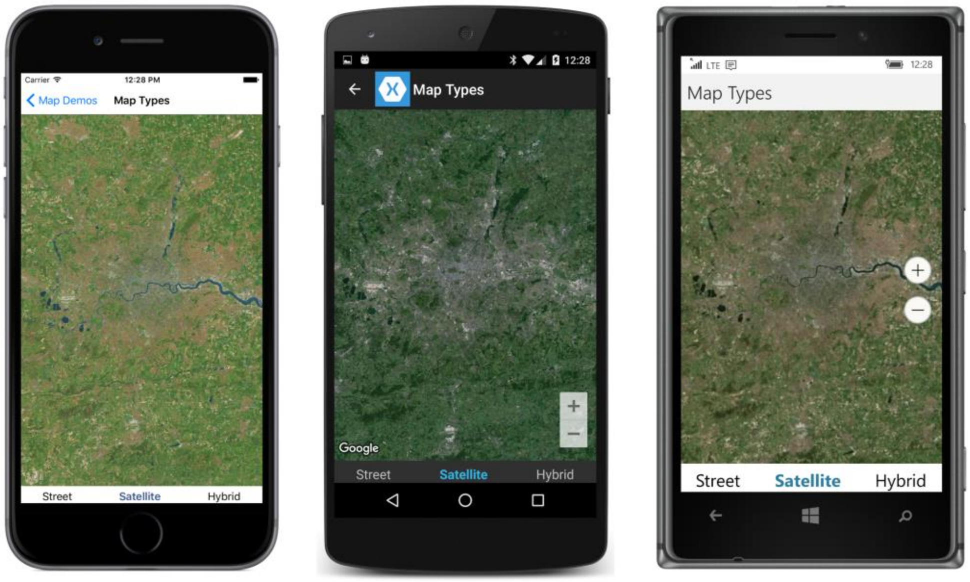
The MapTypesPage XAML file has a Map with a binding to its MapType property from the Radio-ButtonManager. Three instances of MapTypeRadioButton correspond to the three MapType options:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:MapDemos"
             xmlns:maps="clr-namespace:Xamarin.Forms.Maps;assembly=Xamarin.Forms.Maps"
             xmlns:toolkit=
                 "clr-namespace:Xamarin.FormsBook.Toolkit;assembly=Xamarin.FormsBook.Toolkit"
             x:Class="MapDemos.MapTypesPage"
             Title="Map Types">

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <maps:Map Grid.Row="0"
                  MapType="{Binding Source={x:Reference mapTypeRadios},
                                    Path=SelectedValue}" />

        <StackLayout Orientation="Horizontal"
                     Grid.Row="1">
            <StackLayout.BindingContext>
                <toolkit:RadioButtonManager x:Name="mapTypeRadios"
                                            x:TypeArguments="maps:MapType" />
            </StackLayout.BindingContext>

            <StackLayout.Resources>
                <ResourceDictionary>
                    <Style TargetType="local:MapTypeRadioButton">
                        <Setter Property="HorizontalOptions" Value="CenterAndExpand" />
                    </Style>
                </ResourceDictionary>
            </StackLayout.Resources>

            <local:MapTypeRadioButton BindingContext="{Binding Items[0]}" />
            <local:MapTypeRadioButton BindingContext="{Binding Items[1]}" />
            <local:MapTypeRadioButton BindingContext="{Binding Items[2]}" />
        </StackLayout>
    </Grid>
</ContentPage>
```

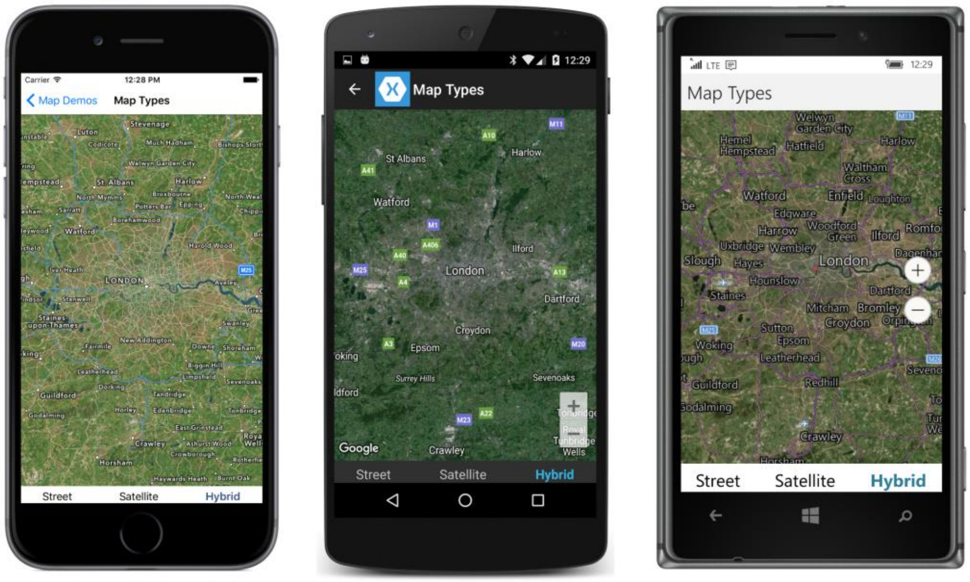
For this screenshot, all three maps have been shifted to display London surrounded by the M25 motorway:



Here's the satellite image:



And here's the `Hybrid` type—a satellite image with superimposed streets and labels on iOS and Windows 10 Mobile, but only labels on Android:



Map coordinates

After a user pans and scales a map, how does the program know what the user is looking at? The `Map` has a property named `VisibleRegion` that indicates what part of the world is visible on the screen.

The VisibleRegion property is *not* backed by a bindable property and there is *no* event indicating when the VisibleRegion has changed. If you want your program to monitor the property, you'll probably use a timer. The property is of type MapSpan, a class which indicates the latitude and longitude of the point at the center of the map, accompanied by three additional properties to describe the area encompassed by the map.

Here are all four read-only properties of MapSpan:

- Center of type Position
- LatitudeDegrees of type double
- LongitudeDegrees of type double
- Radius of type Distance

The LatitudeDegrees and LongitudeDegrees properties indicate the number of degrees spanned by the map, while Radius is a distance from the center that defines a circle entirely visible on the map. These three properties might be regarded as indicating a "zoom level" for the map.

Both Position and Distance are structures included in the **Xamarin.Forms.Maps** assembly and namespace.

Position is very much like the GeographicLocation structure shown earlier in this chapter. It has a constructor that requires latitude and longitude values, and it creates an immutable value with read-only properties named Latitude and Longitude. However, Position doesn't have the handy ToString method that GeographicLocation implements.

The Distance structure eases conversions between metric and English measurements. A constructor accepts a distance in meters, but the structure also defines static FromMeters, FromKilometers, and FromMiles creation methods, and read-only properties named Meters, Kilometers, and Miles. As with the Position structure, Distance values are immutable.

The MapSpan class is also immutable. The MapSpan constructor requires a center with a span in latitude and longitude. Alternatively, the static FromCenterAndRadius method allows specifying a radius instead. You'll see how these work shortly.

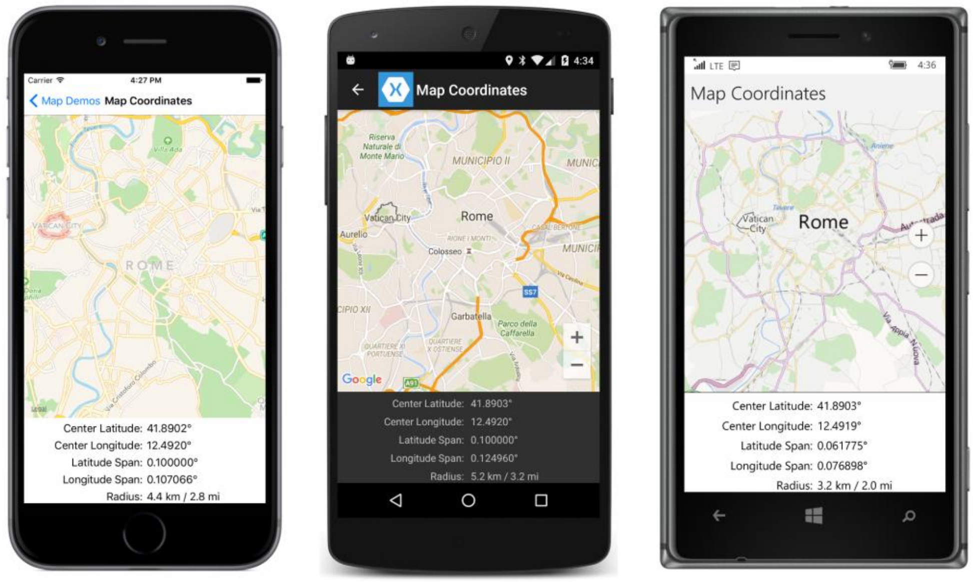
Much of the XAML file for MapCoordinatesPage is devoted to several Label elements to display the property values of the MapSpan object:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:maps="clr-namespace:Xamarin.Forms.Maps;assembly=Xamarin.Forms.Maps"
             x:Class="MapDemos.MapCoordinatesPage"
             Title="Map Coordinates">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <maps:Map x:Name="map"
                  Grid.Row="0" />

        <Grid Grid.Row="1">
            <Label Text="Center Latitude: " Grid.Row="0" Grid.Column="0"
                   HorizontalTextAlignment="End" />
            <Label x:Name="latitude" Grid.Row="0" Grid.Column="1" />

            <Label Text="Center Longitude: " Grid.Row="1" Grid.Column="0"
                   HorizontalTextAlignment="End" />
            <Label x:Name="longitude" Grid.Row="1" Grid.Column="1" />

            <Label Text="Latitude Span: " Grid.Row="2" Grid.Column="0"
                   HorizontalTextAlignment="End" />
            <Label x:Name="latitudeSpan"  Grid.Row="2" Grid.Column="1" />

            <Label Text="Longitude Span: " Grid.Row="3" Grid.Column="0"
                   HorizontalTextAlignment="End" />
            <Label x:Name="longitudeSpan"  Grid.Row="3" Grid.Column="1" />

            <Label Text="Radius: " Grid.Row="4" Grid.Column="0"
                   HorizontalTextAlignment="End" />
            <Label x:Name="radius"  Grid.Row="4" Grid.Column="1" />
        </Grid>
    </Grid>
</ContentPage>
```

The updating of the `Label` elements is handled in a timer callback in the code-behind file:

```
public partial class MapCoordinatesPage : ContentPage
{
    bool stopTimer;

    public MapCoordinatesPage()
    {
        InitializeComponent();

        Device.StartTimer(TimeSpan.FromSeconds(0.1), () =>
        {
            if (stopTimer)
                return false;

            MapSpan mapSpan = map.VisibleRegion;

            if (mapSpan != null)
            {
                latitude.Text = mapSpan.Center.Latitude.ToString("F4") + '°';
                longitude.Text = mapSpan.Center.Longitude.ToString("F4") + '°';
                latitudeSpan.Text = mapSpan.LatitudeDegrees.ToString("F6") + '°';
                longitudeSpan.Text = mapSpan.LongitudeDegrees.ToString("F6") + '°';
                radius.Text = mapSpan.Radius.Kilometers.ToString("F1") + " km / " +
                              mapSpan.Radius.Miles.ToString("F1") + " mi";
            }
            return true;
        });
    }

    protected override void OnDisappearing()
    {
        stopTimer = true;
        base.OnDisappearing();
    }
}
```

Here's that page of the **MapDemos** program when it first starts up:



```
Center Latitude: 41.8902°
Center Longitude: 12.4920°
Latitude Span: 0.100000°
Longitude Span: 0.107066°
Radius: 4.4 km / 2.8 mi
```

```
Center Latitude: 41.8903°
Center Longitude: 12.4920°
Latitude Span: 0.100000°
Longitude Span: 0.124960°
Radius: 5.2 km / 3.2 mi
```

```
Center Latitude: 41.8903°
Center Longitude: 12.4919°
Latitude Span: 0.061775°
Longitude Span: 0.076898°
Radius: 3.2 km / 2.0 mi
```

`Map` initializes itself to have both a latitude span and longitude span of 0.1 degrees. (You'll see how to programmatically do this shortly.) The resultant map is sized so that both the latitude span and the longitude span are *at least* 0.1 degrees.

The Windows 10 Mobile map adds a little bit of margin around the area specified, which is why Rome appears a little more zoomed out on that screen when compared to the others.

The version of Xamarin.Forms used for this chapter (2.3.1.114) calculates the latitude and longitude spans incorrectly on the UWP and Windows Phone 8.1 platforms. The actual values are double those shown: about 0.12 degrees of latitude and 0.15 degrees of longitude, which encompasses the requested size with a bit of margin. The `MapSpan` class calculates the `Radius` property based on the latitude and longitude spans, so that is also incorrect.

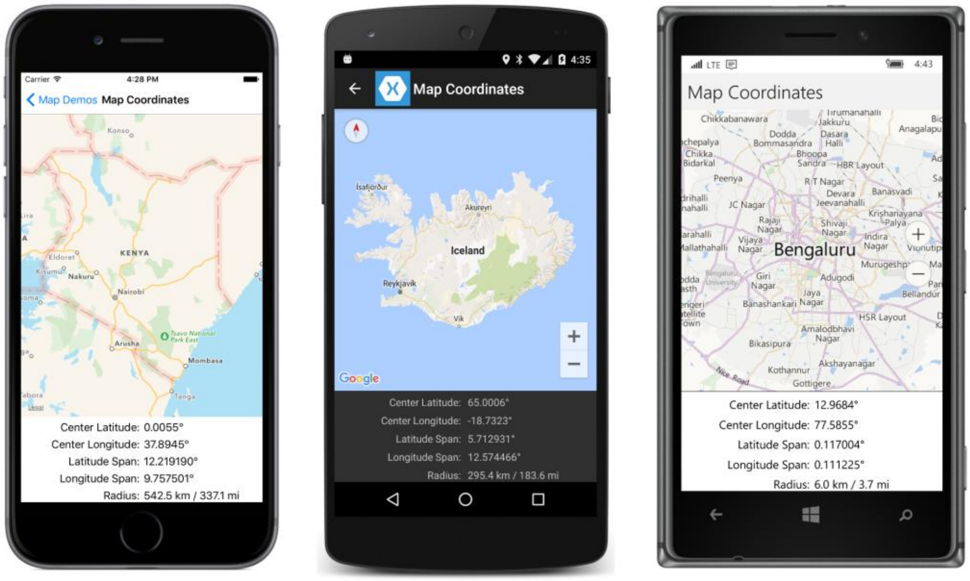
Can you reconcile the radius with the latitude and longitude spans on iOS and Android? Each degree of latitude is about 111 kilometers, so the iOS and Android screens both display a map with a height of about 11.1 kilometers.

At a latitude of 42 degrees, there are 82.5 kilometers to a degree of longitude. (That's 111 times the cosine of 42 degrees.) The iOS longitude span indicates a map width of about 8.8 kilometers, while the Android longitude span is about 10.3 kilometers. In both cases the width is less than the height, so the radius is half the width: 4.4 kilometers on the iPhone screen, and 5.2 kilometers on the Android screen.

Conceptually, the `Radius` property defines a centered circle on the map that is as large as possibly while remaining entirely visible.

If you simply pan the map without changing the zoom, you'll see the longitude span remain constant while the latitude span changes: The latitude span gets larger as you approach the equator, and smaller as you move away. This is a result of the Mercator projection, which spreads out the parallels at the northern and southern extremes. The radius also decreases as you move away from the equator because the Mercator projection is visually stretching areas so that a smaller plot is displayed.

Here are maps that have been panned to various areas of the world:



Center Latitude: 0.0055°
Center Longitude: 37.8945°
Latitude Span: 12.219190°
Longitude Span: 9.757501°
Radius: 542.5 km / 337.1 mi

Center Latitude: 65.0006°
Center Longitude: -18.7323°
Latitude Span: 5.712931°
Longitude Span: 12.574466°
Radius: 295.4 km / 183.6 mi

Center Latitude: 12.9684°
Center Longitude: 77.5855°
Latitude Span: 0.117004°
Longitude Span: 0.111225°
Radius: 6.0 km / 3.7 mi

The iPhone screen displays a map of Kenya centered almost at the equator. The difference between the latitude and longitude simply reflects the aspect ratio of the rendered map.

The map on the Android screen is very nearly a square, but the latitude and longitude spans are quite different. That's a result of the Mercator projection. Iceland is 65 degrees north of the equator, and at that latitude the Mercator projection vertically stretches by a factor of about 2.4 (the secant of 65 degrees).

Windows Phone displays the map of India's technology center Bangalore in a nearly square aspect ratio as well, but the city is much closer to the equator, so the vertical stretching is only about 2.6 percent, and the latitude and longitude spans are nearly equal. (But keep in mind that these values are half of the actual values, and the radius of the area is really 12 kilometers.)

You might think that you can overlay an `AbsoluteLayout` on the `Map`, and then use the `MapSpan` information and some mathematics to position boxes or ellipses at particular geographic locations, much like the **WhereAmI** program shown earlier. That would certainly be possible were the map not subject to two-finger rotation. That map rotation makes this otherwise intriguing idea impossible.

Position extensions

The `MapCoordinatesPage` code you just saw could have been simplified if the `Position` structure included a `ToString` method that formats the longitude and latitude values. For future applications, let's add a `ToString` method to `Position` as an extension method.

Normally a class containing extension methods developed in this book would go in the **Xamarin.FormsBook.Toolkit** library. However, a class containing extension methods for `Position` requires the libraries installed by the **Xamarin.Forms.Maps** NuGet package, and that's overkill for everything else in **Xamarin.FormsBook.Toolkit**. The class shouldn't go in the **Xamarin.FormsBook.Platform** library either because it's not platform dependent.

This means that a new library is required for `Map`-related platform-independent classes. This library is called **Xamarin.FormsBook.Toolkit.Maps**. Among the downloadable code for this book, this library is with the others in the **Libraries** folder.

Here's a versatile `ToString` method for `Position`:

```
namespace Xamarin.FormsBook.Toolkit.Maps
{
    public static class PositionExtensions
    {
        public static string ToString(this Position self, string format)
        {
            if (String.IsNullOrWhiteSpace(format))
                throw new FormatException("Missing format specifier on ToString");

            string floatFormat = "F";

            if (format.Length > 1)
                floatFormat += format.Substring(1);

            char fmt = Char.ToUpper(format[0]);

            switch (fmt)
            {
                case 'F':       // 'float': raw numbers
                    return String.Format("{0} {1}", self.Latitude.ToString(floatFormat),
                                                    self.Longitude.ToString(floatFormat));

                case 'D':       // 'degrees'
                case 'M':       // 'minutes'
                case 'S':       // 'seconds'
                    return Composite(self, fmt, floatFormat);

                default:
                    throw new FormatException("ToString format must be F, D, M, or S");
            }
        }

        static string Composite(Position position, char fmt, string floatFmt)
        {
            return String.Format("{0}{1} {2}{3}", DMS(position.Latitude, fmt, floatFmt),
                                                  position.Latitude >= 0 ? 'N' : 'S',
                                                  DMS(position.Longitude, fmt, floatFmt),
                                                  position.Longitude >= 0 ? 'E' : 'W');
        }

        static string DMS(double value, char fmt, string floatFormat)
        {
            value = Math.Abs(value);

            if (fmt == 'D')
                return String.Format("{0}\u00B0", value.ToString(floatFormat));

            int degrees = (int)value;
            value = (value - degrees) * 60;

            if (fmt == 'M')
            {
                // Check for rounding issues.
                string strMins = value.ToString(floatFormat);
                if (strMins.StartsWith("60"))
                {
                    value -= 60;
                    degrees += 1;
                }
                return String.Format("{0}\u00B0{1}\u2032", degrees,
                                     value.ToString(floatFormat));
            }

            int minutes = (int)value;
            value = (value - minutes) * 60;

            // Check for rounding issues.
            string strSecs = value.ToString(floatFormat);
            if (strSecs.StartsWith("60"))
            {
                value -= 60;
                minutes += 1;

                if (minutes == 60)
                {
                    minutes -= 60;
                    degrees += 1;
                }
            }
```

```
            return String.Format("{0}\u00B0{1}\u2032{2}\u2033", degrees, minutes,
                                 value.ToString(floatFormat));
        }
        …
    }
}
```

This `ToString` method allows formatting latitude and longitude values in one of four ways based on the following formatting specifications:

- `F#` where # is a number indicating the number of decimal points.
- `D#` to display degrees with a specified number of decimal points.
- `M#` to display degrees and minutes, where minutes have the specified number of decimal points.
- `S#` to display degrees, minutes, and seconds, where seconds have the specified number of decimal points.

The `F` specification is for simple positive and negative floating point values of the latitude and longitude. The `D`, `M`, and `S` specifications don't display negative values. Instead, they use the letters **N** and **S** for north and south latitudes, and **E** and **W** for east and west longitudes.

For the `D`, `M`, and `S` specifications, the number of decimal places only applies to the last number of the latitude and longitude. For example, if you specify `S2`, the degrees and minutes are displayed as integers but the seconds are displayed with two decimal places for an accuracy of 31 centimeters at the equator.

Because these are extension methods, they are only valid when the C# compiler encounters a `ToString` method with a parameter on a `Position` value. You cannot use these formatting specifications in the formatting string of a `String.Format` call, or in a `StringFormat` specification of a data binding.

The **MapDemos** project has a reference to the **Xamarin.FormsBook.Toolkit.Maps** project. The next program demonstrates the new `ToString` method.

Setting an initial location

The `MapSpan` class has a dual purpose: As you've seen, you use it when determining what portion of the world the `Map` displays. `Map` can take an optional constructor argument of type `MapSpan` that initializes the map to a particular location and zoom level. You specify the location of the map by the latitude and longitude of the point at the center of the map. You can indicate the zoom level in one of two ways: as a range of latitude and longitude degrees, or as a radius in meters, kilometers, or miles.

After the `Map` has been instantiated, you can later programmatically move it to a different location by calling the `MoveToRegion` method, which also requires a `MapSpan` object.

You can create a MapSpan object in one of two ways:

- The constructor requires a Position argument for the center of the map, and latitude and longitude spans.
- The static FromCenterAndRadius method requires a Position argument and a radius value of type Distance.

Once created, a MapSpan object is immutable.

Let's test this out. Very conveniently, the states of Wyoming and Colorado have borders based entirely on lines of latitude and longitude. The south and north borders of Wyoming are the parallels at 37 degrees north and 41 degrees north, for a span of 4 degrees. The east and west borders are the meridians with longitudes of 102.05 degrees and 109.05 degrees west, for a span of 7 degrees. (The fractional 0.05 degrees might seem odd until you learn that the original calculation of the east and west borders was based on the meridian at Washington D.C.) The center is a latitude of 39 degrees and a longitude of –105.55 degrees.

Here's the WyomingPage XAML file:

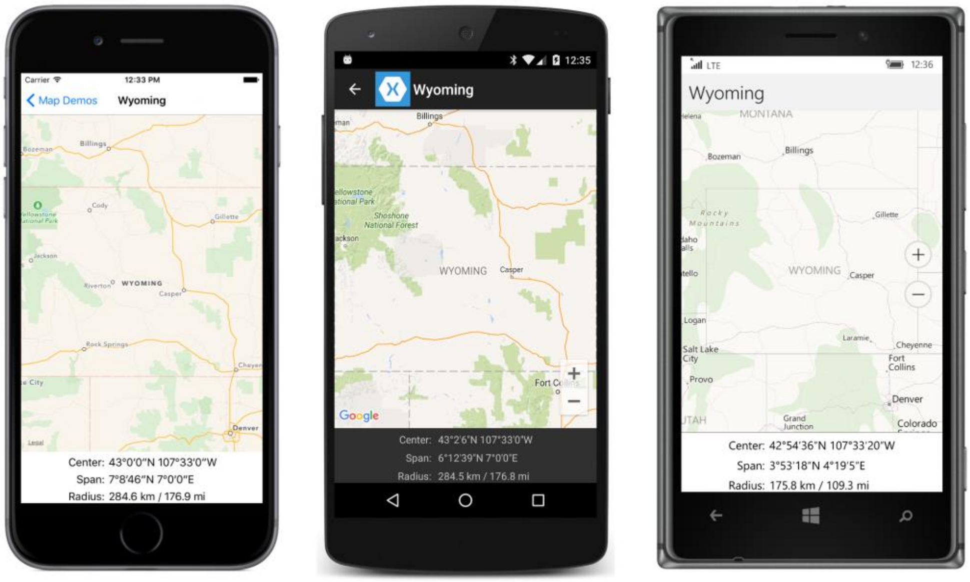
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:maps="clr-namespace:Xamarin.Forms.Maps;assembly=Xamarin.Forms.Maps"
             x:Class="MapDemos.WyomingPage"
             Title="Wyoming">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <maps:Map x:Name="map"
                  Grid.Row="0" />

        <Grid Grid.Row="1">
            <Label Text="Center: " Grid.Row="0" Grid.Column="0"
                   HorizontalTextAlignment="End" />
            <Label x:Name="center" Grid.Row="0" Grid.Column="1" />

            <Label Text="Span: " Grid.Row="1" Grid.Column="0"
                   HorizontalTextAlignment="End" />
            <Label x:Name="span"  Grid.Row="1" Grid.Column="1" />

            <Label Text="Radius: " Grid.Row="2" Grid.Column="0"
                   HorizontalTextAlignment="End" />
            <Label x:Name="radius"  Grid.Row="2" Grid.Column="1" />
        </Grid>
    </Grid>
</ContentPage>
```

As in the previous sample, most of the file is dedicated to displaying the current location, but the markup is a little shorter because the code-behind file is able to make use of the ToString method in the PositionExtensions class:

```
public partial class WyomingPage : ContentPage
{
    bool stopTimer;

    public WyomingPage()
    {
        InitializeComponent();

        map.MoveToRegion(new MapSpan(new Position(43, -107.55), 4, 7));

        Device.StartTimer(TimeSpan.FromSeconds(0.1), () =>
        {
            if (stopTimer)
                return false;

            MapSpan mapSpan = map.VisibleRegion;

            if (mapSpan != null)
            {
                center.Text = mapSpan.Center.ToString("S0");
                span.Text = new Position(mapSpan.LatitudeDegrees,
                                         mapSpan.LongitudeDegrees).ToString("S0");
                radius.Text = mapSpan.Radius.Kilometers.ToString("F1") + " km / " +
                                mapSpan.Radius.Miles.ToString("F1") + " mi";
            }
            return true;
        });
    }

    protected override void OnDisappearing()
    {
        stopTimer = true;
        base.OnDisappearing();
    }
}
```

Notice the call to MoveToRegion with a MapSpan constructor to specify the center and latitude and longitude span of Wyoming.

Here's how the map comes up on the three platforms:



```
Center: 43°0'0"N 107°33'0"W
Span: 7°8'46"N 7°0'0"E
Radius: 284.6 km / 176.9 mi
```

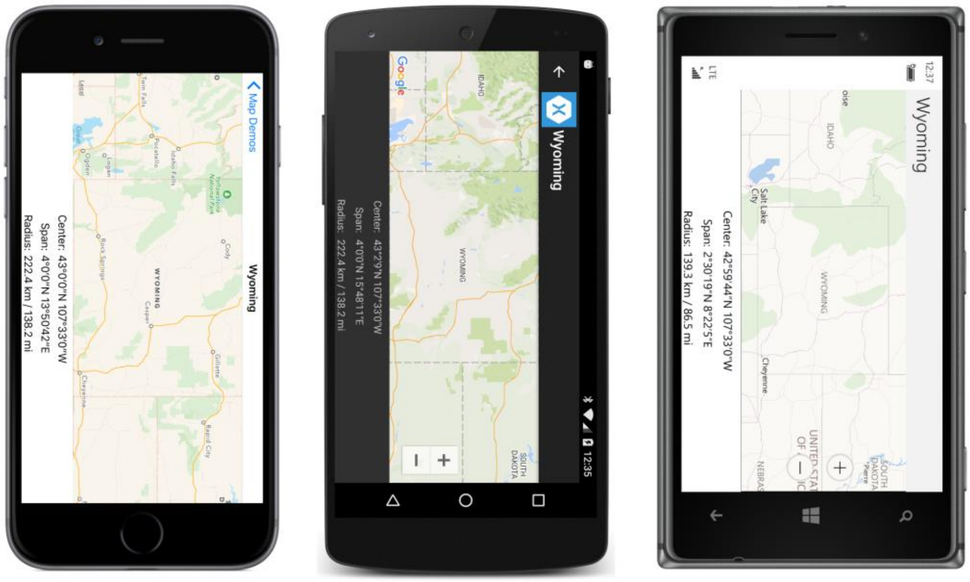
```
Center: 43°2'6"N 107°33'0"W
Span: 6°12'39"N 7°0'0"E
Radius: 284.5 km / 176.8 mi
```

```
Center: 42°54'36"N 107°33'20"W
Span: 3°53'18"N 4°19'5"E
Radius: 175.8 km / 109.3 mi
```

On iOS and Android, the 7-degree-longitude width of Wyoming is position snugly within the width of the phone. At a latitude of 43 degrees, one degree of longitude is about 81 kilometers (111 times the cosine of 43), so the width of Wyoming is about 568 kilometers. The radius is about half that, or 284 kilometers.

The Windows 10 Mobile map puts a little extra space around the specified range. This is normal and is *not* something weird that Xamarin.Forms is doing. It is unrelated to the incorrect span and radius information.

You can also try the program in landscape mode. You can turn the phone sideways while Wyoming is displayed, but doing so will maintain the same zoom level. To ensure that the map is once again zoomed based on the size of the latitude and longitude spans, first go back to the home page, turn the phone sideways, and then navigate to the Wyoming map again:

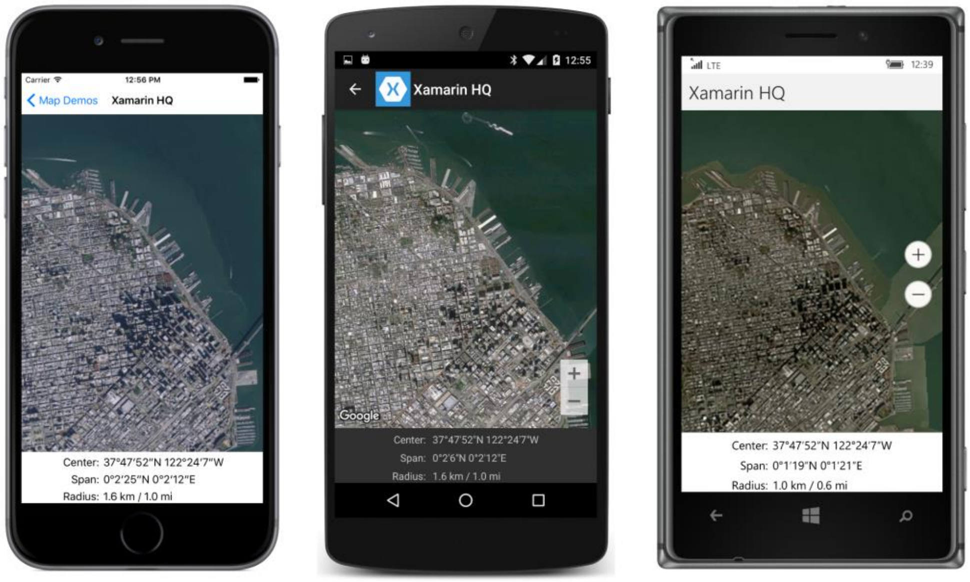

Map Demos
Wyoming
Center: 43°0'0"N 107°33'0"W
Span: 4°0'0"N 13°50'42"E
Radius: 222.4 km / 138.2 mi
Wyoming
Center: 43°2'9"N 107°33'0"W
Span: 4°0'0"N 15°48'11"E
Radius: 222.4 km / 138.2 mi
Wyoming
Center: 42°59'44"N 107°33'0"W
Span: 2°30'19"N 8°22'5"E
Radius: 139.3 km / 86.5 mi

Now the map has been sized so that the 4-degree latitude height of Wyoming fits snugly on the page (at least on the iOS and Android screens). There are 111 kilometers to each degree of latitude, so the radius is now 222 kilometers.

It's also possible to use the `x:Arguments` tag and the `x:FactoryMethod` attribute to initialize the location and the span of the map right in XAML. Here's the XAML file for `XamarinHQPage` that uses this technique to display a 1-mile radius of the Xamarin headquarters in San Francisco:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:maps="clr-namespace:Xamarin.Forms.Maps;assembly=Xamarin.Forms.Maps"
             x:Class="MapDemos.XamarinHQPage"
             Title="Xamarin HQ">

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <maps:Map x:Name="map"
                  Grid.Row="0"
                  MapType="Satellite">
            <x:Arguments>
                <maps:MapSpan x:FactoryMethod="FromCenterAndRadius">
                    <x:Arguments>
                        <maps:Position>
                            <x:Arguments>
                                <x:Double>37.797787</x:Double>
                                <x:Double>-122.401855</x:Double>
                            </x:Arguments>
                        </maps:Position>
                        <maps:Distance x:FactoryMethod="FromMiles">
                            <x:Arguments>
                                <x:Double>1</x:Double>
                            </x:Arguments>
                        </maps:Distance>
                    </x:Arguments>
                </maps:MapSpan>
            </x:Arguments>
        </maps:Map>

        <Grid Grid.Row="1"
              HorizontalOptions="Center">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>

            <Label Text="Center: " Grid.Row="0" Grid.Column="0"
                   HorizontalTextAlignment="End" />
            <Label x:Name="center" Grid.Row="0" Grid.Column="1" />

            <Label Text="Span: " Grid.Row="1" Grid.Column="0"
                   HorizontalTextAlignment="End" />
            <Label x:Name="span"  Grid.Row="1" Grid.Column="1" />

            <Label Text="Radius: " Grid.Row="2" Grid.Column="0"
                   HorizontalTextAlignment="End" />
            <Label x:Name="radius"  Grid.Row="2" Grid.Column="1" />
        </Grid>
    </Grid>
</ContentPage>
```

The code-behind file is similar to the previous page except there is no code to set the map's position. Here's the result:



Center: 37°47′52″N 122°24′7″W
Span: 0°2′25″N 0°2′12″E
Radius: 1.6 km / 1.0 mi

Center: 37°47′52″N 122°24′7″W
Span: 0°2′6″N 0°2′12″E
Radius: 1.6 km / 1.0 mi

Center: 37°47′52″N 122°24′7″W
Span: 0°1′19″N 0°1′21″E
Radius: 1.0 km / 0.6 mi

The radius is set to 1 mile, so the map has been sized to display a span of at least 2 miles both horizontally and vertically. Within the `MapSpan` object, that distance is converted to a latitude and longitude in this way:

One degree of latitude is 69 miles, so 1 mile is about 0.0145 degrees or about 0.87 minutes. The map must be zoomed to display at least double that or 1.74 minutes, or 1 minute and 44 seconds.

At a latitude of 37.8 degrees, one degree of longitude is about 54.5 miles (69 times the cosine of 37.8 degrees), so one mile is 0.183 degrees or about 1.1 minutes. The map must be zoomed to display at least double that, or 2.2 minutes, which is 2 minutes and 12 seconds.

The map is thus sized to display at least 1 minute and 44 seconds of latitude, and 2 minutes and 12 seconds of latitude. The latitude span on both the iOS and Android devices is greater than 1 minute and 44 seconds, and the longitude span is exactly 2 minutes and 12 seconds.

Internally, the `MapSpan` class does not maintain a radius value. The `FromCenterAndRadius` method simply converts the radius to a latitude and longitude span using calculations similar to what have been described throughout this chapter, and passes those values to the `MapSpan` constructor. The `get` accessor of the `Radius` property returns the minimum of the latitude span and the longitude span converted to a `Distance` value.

Dynamic zooming

The `RadiusZoomPage` allows you to zoom in and out of a particular location using the `Slider`.

Zoom levels are generally a logarithmic scale. If zoom level 0 is a very zoomed out view with a radius of 1000 miles (for example), then zoom level 1 is a radius of 500 miles, and zoom level 2 is a radius of 250 miles. Each successive zoom level effectively doubles the magnification and halves the radius. Mathematically,

$$ZoomLevel = \log_2 \frac{1000}{Radius}$$

or:

$$Radius = 1000 \times 2^{-ZoomLevel}$$

When using a `Slider` for this purpose, generally you'll want to give the `Slider` a range corresponding to the zoom levels you want, and to convert that to a radius (or longitude or latitude span) for the map.

The XAML file for the `RadiusZoomPage` class includes a `Map` and a `Slider` initialized to a range of 0 through 16. These are the zoom levels:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:maps="clr-namespace:Xamarin.Forms.Maps;assembly=Xamarin.Forms.Maps"
             xmlns:local="clr-namespace:MapDemos"
             x:Class="MapDemos.RadiusZoomPage"
             Title="Radius Zoom">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <maps:Map x:Name="map"
                  Grid.Row="0"
                  MapType="Hybrid" />

        <Slider x:Name="slider"
                Grid.Row="1"
                Maximum="16"
                Margin="10, 0" />
    </Grid>
</ContentPage>
```

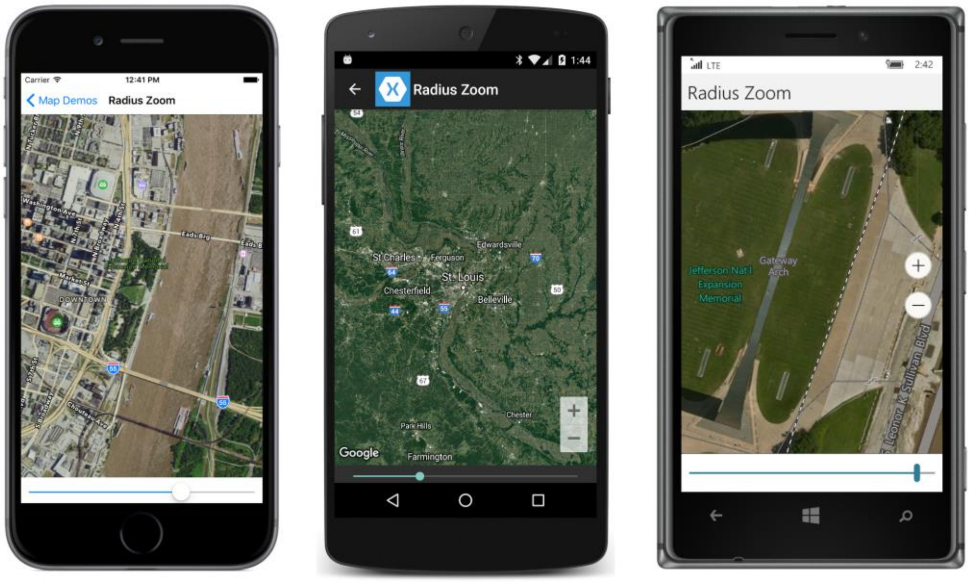
The code-behind file initializes both the `Map` position and the `Slider` value. The `Map` is centered on the famous Gateway Arch in St. Louis, Missouri, with a radius of 1 kilometer. The `Slider` is given a value of 11, implicitly associating that zoom level with a 1-kilometer radius. The constructor concludes by setting a handler for the `ValueChanged` event:

```
public partial class RadiusZoomPage : ContentPage
{
    const double InitialRadius = 1;     // kilometer
    const double InitialZoom = 11;

    public RadiusZoomPage()
    {
        InitializeComponent();
        map.MoveToRegion(MapSpan.FromCenterAndRadius(new Position(38.62452, -90.18471),
                                                     Distance.FromKilometers(InitialRadius)));
        slider.Value = InitialZoom;
        slider.ValueChanged += OnSliderValueChanged;
    }

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        // Watch out for Android event firing!
        if (map.VisibleRegion == null)
            return;

        Position center = map.VisibleRegion.Center;
        double radius = InitialRadius * Math.Pow(2, InitialZoom - args.NewValue);
        map.MoveToRegion(MapSpan.FromCenterAndRadius(center, Distance.FromKilometers(radius)));
    }
}
```

The first statement in the `ValueChanged` handler might seem odd: Although the constructor sets the `ValueChanged` handler *after* setting the `Value` property, on Android the `ValueChanged` event might be fired when the `Slider` renderer is attached during layout. The Android `SeekBar` is integer-based with a range of 1000, and the `Value` of 11 is represented by a `SeekBar` value of 687, which corresponds to a `Value` property of 10.992 (687 divided by 1000 times 16). That change causes the `Slider` to fire the event. However, at that time the `VisibleRegion` of the `Map` might be `null`.

The handler uses the existing center and calculates a new radius based on the `Slider` value. With a minimum `Slider` value of 0, the radius is 2,048 kilometers, and with a maximum `Slider` value of 16, the radius is 1/32 kilometer, or 31.25 meters.

The three screenshots here show the initial view on the iOS screen, a zoomed-out map on Android, and a zoomed-in view on the Windows 10 Mobile screen:



You'll discover a couple of issues with this program:

First, you'll probably have a strong desire to move the `Slider` very quickly to zoom in and out, but that doesn't work well. The underlying maps perform animation for the zoom, and they won't attempt to zoom the map while still getting `MoveToRegion` calls. It's best to move the `Slider` rather slowly and not continuously. (Animations that perform smooth zooms on Earth sites usually obtain their tiles directly from the map server to have more control over their display.)

On iOS, the maximum zoom is greater than that provided by the Apple Maps service, and instead of seeing images of the terrain, you'll see simple placeholders.

Perhaps most disturbing is this: On both Android and Windows 10 Mobile, if you zoom out and then zoom back in, you won't come back to the same spot!

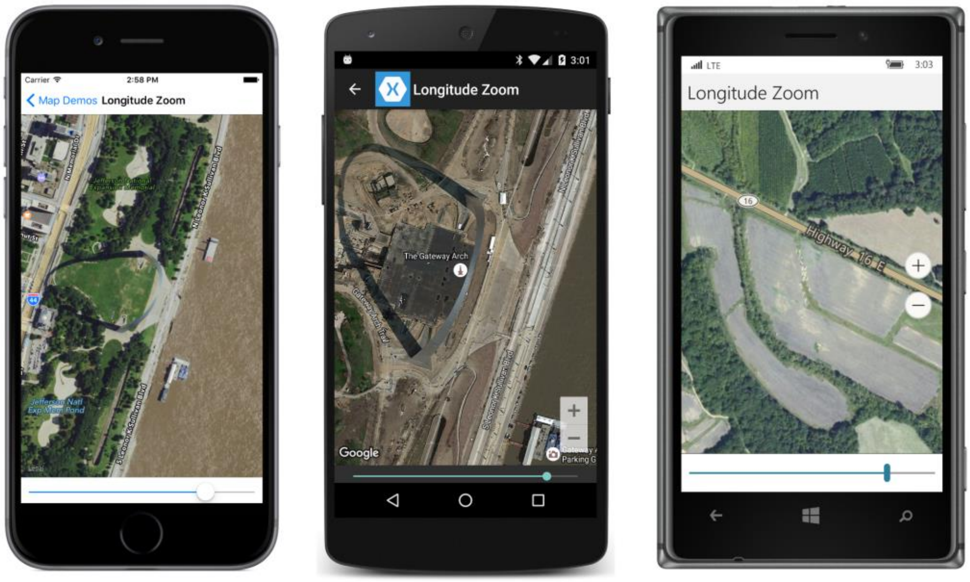
On Android, that problem evidently results from some rounding errors involving the radius. You can fix it by scaling the longitude span rather than the radius. Here's the XAML file for `LongitudeZoom-Page`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:maps="clr-namespace:Xamarin.Forms.Maps;assembly=Xamarin.Forms.Maps"
             xmlns:local="clr-namespace:MapDemos"
             x:Class="MapDemos.LongitudeZoomPage"
             Title="Longitude Zoom">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <maps:Map x:Name="map"
                  Grid.Row="0"
                  MapType="Hybrid" />

        <Slider x:Name="slider"
                Grid.Row="1"
                Maximum="16"
                Margin="10, 0"
                ValueChanged="OnSliderValueChanged" />
    </Grid>
</ContentPage>
```

The zoom range is still 0 to 16, but the constructor in the code-behind file doesn't change the `Value` property of the `Slider`. Instead, it associates a zoom level of zero with a longitude span of 48 degrees, which is sufficient to encompass much of the continental United States:

```
public partial class LongitudeZoomPage : ContentPage
{
    const double InitialLongitudeSpan = 48;
    const double InitialZoom = 0;

    public LongitudeZoomPage()
    {
        InitializeComponent();

        map.MoveToRegion(new MapSpan(new Position(38.62452, -90.18471),
                                     0, InitialLongitudeSpan));
    }

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        Position center = map.VisibleRegion.Center;
        double longitudeSpan = InitialLongitudeSpan * Math.Pow(2, InitialZoom - args.NewValue);
        map.MoveToRegion(new MapSpan(center, 0, longitudeSpan));
    }
}
```

Notice that both `MapSpan` constructors set the latitude span to 0. This is not a problem. The `Map` bases the zoom level to encompass both the latitude and longitude span, so one of these values can be zero. With zoom levels from 0 to 16, the longitude span ranges from 48 degrees to the value 48 degrees divided by 65,536, which is about 2.6 seconds. Here are all three platforms zoomed to see the arch (and its shadow) in more detail:



All except Windows 10 Mobile, unfortunately. The underlying `MapControl` doesn't seem to maintain a tight consistency between the center that it's set to, and the center it reports, which means that the `center` variable obtained in the `OnSliderValueChanged` method isn't necessarily the location of the Gateway Arch. To make this program work on Windows 10 Mobile, you would need to set the center coordinates explicitly in every calls to `MoveToRegion`.

The Phone's Location

`Map` has an `IsShowingUser` property that is backed by a bindable property, which means that it can be the target of a data binding. The `ShowLocationPage` binds that property to a `Switch`. The code-behind file contains nothing but a standard call to `InitializeComponent`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:maps="clr-namespace:Xamarin.Forms.Maps;assembly=Xamarin.Forms.Maps"
             x:Class="MapDemos.ShowLocationPage"
             Title="Show Location">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <maps:Map Grid.Row="0"
                  IsShowingUser="{Binding Source={x:Reference switch},
                                          Path=IsToggled,
                                          Mode=TwoWay}" />

        <StackLayout Grid.Row="1"
```

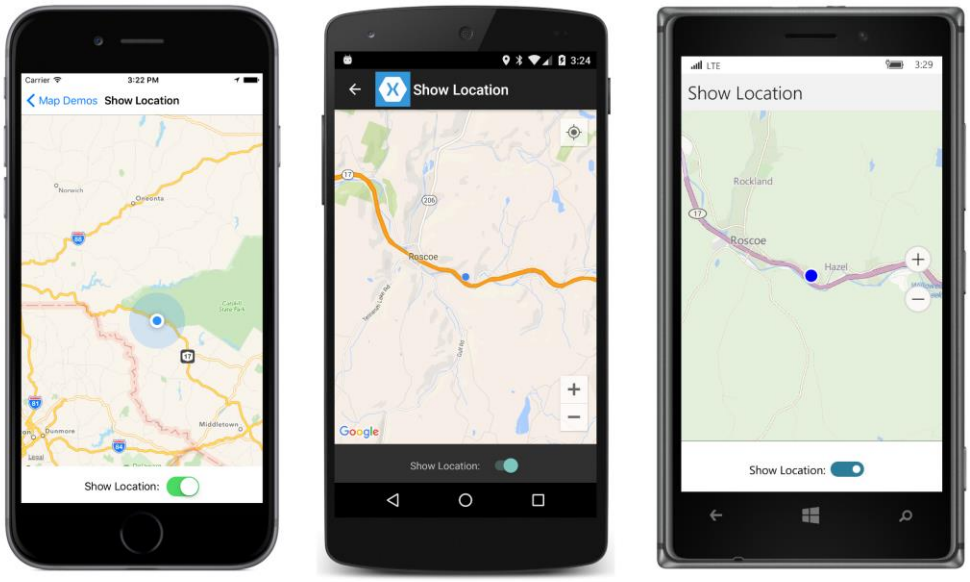
```
                     Orientation="Horizontal"
                     HorizontalOptions="Center"
                     Margin="0, 10">

            <Label Text="Show Location: "
                   VerticalOptions="Center" />

            <Switch x:Name="switch" />
        </StackLayout>
    </Grid>
</ContentPage>
```

What happens when you toggle that switch on and set `IsShowingUser` to `true`? It depends on the platform.

On iOS, you'll first be asked if **MapDemos** can use your location. This is the first time that the `Map` needs the user's location, so this is the first time you'll see the text description included with the `NSLocationWhenInuseUsageDescription` key in the Info.plist file. If you give **MapDemos** permission to get your location, nothing seems to happen. But if you then use your fingers to navigate to your location, you'll see it identified with a pulsing blue dot.

On Android, setting `IsShowingUser` to true causes a crosshair icon to appear in the upper-right corner of the map. Pressing that causes the map to shift and zoom to your current location, which is displayed with a static blue dot.

On Windows 10 Mobile, you'll be asked with a message box if the program can use your location. If you respond **Yes**, what happens (or what is supposed to happen) isn't quite clear. If at first nothing seems to happen, you might want to try using your fingers to move to your location. Sometimes when you get close, the map will jump to your location and display it with a blue dot:



Under the assumption that Android handles this option best, let's try to duplicate the Android approach on iOS and Windows 10 Mobile.

First, we'll need a button that resembles the Android cross-hair icon. Appropriate images can be found in the GitHub repository http://www.github.com/google/material-design-icons. The **MapDemos** project contains an **Images** directory with two files copied from the **maps/1x_web** folder of that repository:

- ic_my_location_black_24dp.png
- ic_my_location_black_48dp.png

These two bitmaps are 24 pixels square and 48 pixels square, respectively. Because Windows 10 Mobile and iOS size bitmaps differently when they are displayed, the first bitmap is intended for Windows 10 Mobile and the second for iOS. In a real application, you'll probably want to have several different sizes in the application projects appropriate for different device resolutions.

In accordance with the name of the bitmaps, the name of the custom element that displays this bitmap is `MyLocationButton`. Here's the XAML file for `MyLocationButton`. Notice that it derives from `ContentView`:

```
<ContentView xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:MapDemos"
             x:Class="MapDemos.MyLocationButton"
             BackgroundColor="White"
             Padding="10">
    <ContentView.GestureRecognizers>
        <TapGestureRecognizer Tapped="OnTapped" />
    </ContentView.GestureRecognizers>

    <Image InputTransparent="True">
        <Image.Source>
            <OnPlatform x:TypeArguments="ImageSource"
            iOS="{local:ImageResource MapDemos.Images.ic_my_location_black_48dp.png}"
            WinPhone="{local:ImageResource MapDemos.Images.ic_my_location_black_24dp.png}" />
        </Image.Source>
    </Image>
</ContentView>
```

The `ContentView` is colored white with `Padding` that causes it to be larger than its child, the `Image` element, by 10 units on all sizes. The `Image` element is made transparent to input, so all taps are detected by the `TapGestureRecognizer` on the `ContentView`.

The code-behind file generates a `Clicked` event from the `Tapped` event:

```
public partial class MyLocationButton : ContentView
{
    public event EventHandler Clicked;

    public MyLocationButton()
    {
        InitializeComponent();
    }

    void OnTapped(object sender, EventArgs args)
    {
        Clicked?.Invoke(sender, args);
    }
}
```

The `GoToLocationPage` sample simply overlays a `MyLocationButton` in the upper-right corner of a single-cell `Grid` also containing the `Map`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:maps="clr-namespace:Xamarin.Forms.Maps;assembly=Xamarin.Forms.Maps"
             xmlns:local="clr-namespace:MapDemos"
             x:Class="MapDemos.GoToLocationPage"
             Title="Go to Location">

    <Grid>
        <maps:Map x:Name="map"
                  IsShowingUser="True" />

        <local:MyLocationButton x:Name="myLocationButton"
                                IsVisible="False"
                                Margin="10"
                                HorizontalOptions="End"
                                VerticalOptions="Start"
                                Clicked="OnLocationButtonClicked" />
    </Grid>
</ContentPage>
```

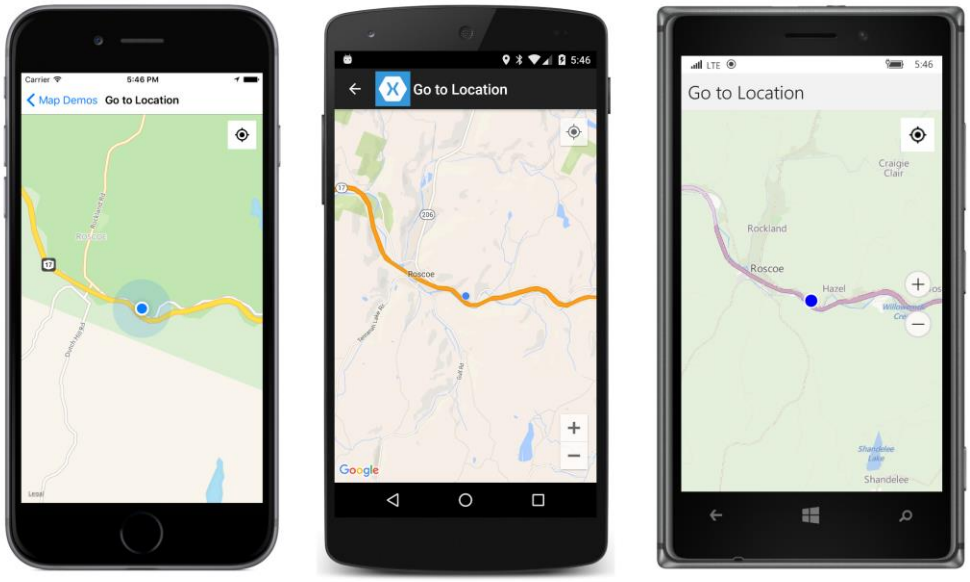
Notice that the `IsShowingUser` property is initialized to `True`. This displays a blue dot at the user's location on iOS and Windows 10 Mobile. On Android, it also displays the map's standard location button. That's one reason why the `IsVisible` property of `MyLocationButton` is set to `False`. This button shouldn't be displayed on Android devices, and it should not be displayed on the other platforms until the code-behind file has the user's location.

To obtain the user's location, the code-behind file needs access to the `LocationTracker` class developed earlier in this chapter. For that reason, the **MapDemos** solution contains the seven projects in the **Xamarin.FormsBook.Platform** solution. The **MapDemos** project has a reference to the **Xamarin.FormsBook.Platform** project itself, and all the **MapDemos** application projects have references to the corresponding platform libraries, and they all contain calls to the `Toolkit.Init` methods in those libraries.

The code-behind file doesn't do anything for Android. For iOS and Windows 10 Mobile, the constructor creates an `ILocationTracker` object and attaches a handler for the `LocationChanged` event. This handler simply saves the new position and (if this is the first call) makes the `MyLocationButton` object visible. The location tracker is also started and paused during the calls to `OnAppearing` and `OnDisappearing`:

```
public partial class GoToLocationPage : ContentPage
{
    ILocationTracker locationTracker;
    Position position;

    public GoToLocationPage()
    {
        InitializeComponent();

        if (Device.OS != TargetPlatform.Android)
        {
            locationTracker = DependencyService.Get<ILocationTracker>();
            locationTracker.LocationChanged += OnLocationTracker;
        }
    }

    void OnLocationTracker(object sender, GeographicLocation args)
    {
        position = new Position(args.Latitude, args.Longitude);
        myLocationButton.IsVisible = Device.OS != TargetPlatform.Android;
    }

    protected override void OnAppearing()
    {
        base.OnAppearing();

        if (Device.OS != TargetPlatform.Android)
        {
            locationTracker.StartTracking();
        }
    }
    protected override void OnDisappearing()
    {
        base.OnDisappearing();

        if (Device.OS != TargetPlatform.Android)
        {
            locationTracker.PauseTracking();
        }
    }

    void OnLocationButtonClicked(object sender, EventArgs args)
    {
        map.MoveToRegion(MapSpan.FromCenterAndRadius(position, Distance.FromMiles(2)));
    }
}
```

The final method in this class is executed when the user presses the `MyLocationButton`. It simply calls `MoveToRegion` with the saved position and a radius of two miles. Here's the result after pressing that button from a location in the Catskill Mountains in New York State:



As you can see, the custom `MyLocationButton` objects on the iOS and iPhone screens look fairly similar to the one on the Android map, and the two-mile radius seems consistent as well.

As the phone moves location, the blue dot tracks the location because that's handled by the underlying map control. Pressing the crosshair button again moves that new location to the center with a radius of two miles.

Pins and science museums

The Map class defines a Pins property of type IList<Pin>. This property allows you to display markers on the map indicating specific geographic locations.

The Pin class defines four properties:

- Label of type string to assign a name to the Pin.
- Position of type Position, indicating its geographic location.
- Address of type string with a human-readable street address.
- Type of type PinType.

PinType is an enumeration with four members—Generic (the default), Place, SavedPin, and SearchResult—but none of the platforms make use of this property.

The Label property must be set before a Pin is added to the Pins collection, but the Address property is optional. In a sense, the Position property is also optional, but Position is a structure, so if you don't specify it, the pin will be positioned at a latitude and longitude of zero—on the prime meridian at the equator in the Gulf of Guinea.

The Pin class also defines a Clicked event that applications generally use to provide more information to the user about that location.

The Pin class overrides its Equals method to return true if two objects have the same four property settings. This can help you avoid multiple Pin objects denoting the same location.

The Pin class has no facility to attach other information that an application might find useful, and the class is sealed so you can't derive from it and add additional properties. However, Pin derives from BindableObject, which means that you can set the BindingContext property to any convenient object of your choice. The Position, Address, and Type properties are all backed by bindable properties, so you can set them using data bindings.

However, the Pins collection itself is not backed by a bindable property, so you cannot bind a collection to the Pins property.

Let's use the Pins property of Map to display the locations and other information of science museums in the United States.

The **MapDemos** project has a **Data** folder with a file ScienceMuseums.xml that contains information about 253 museums in the United States. The source of these museums is the Wikipedia page, "List of science centers in the United States." Here are the top and bottom of this file:

```
<Locations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Title>Science Museums</Title>
  <Sites>
    <Site
```

```
      Name="A.C. Gilbert's Discovery Village"
      LongName="A.C. Gilbert's Discovery Village"
      Latitude="44.945"
      Longitude="-123.04111"
      Address="116 Marion St., NE, Salem, Oregon"
      Website="http://www.acgilbert.org/" />
    <Site
      Name="A. E. Seaman Mineral Museum"
      LongName="A. E. Seaman Mineral Museum"
      Latitude="47.1108"
      Longitude="-88.5526"
      Address="Houghton, Michigan"
      Website="http://www.museum.mtu.edu/" />
    …
    <Site
      Name="Zeum"
      LongName="Zeum: San Francisco's Children's Museum"
      Latitude="37.783273"
      Longitude="-122.401827"
      Address="San Francisco, California"
      Website="http://www.creativity.org/" />
  </Sites>
</Locations>
```

The **MapDemos** project also includes two classes for deserializing this data. The `Site` class contains all the information about a particular museum:

```
public class Site
{
    [XmlAttribute]
    public string Name { set; get; }

    [XmlAttribute]
    public string LongName { set; get; }

    [XmlAttribute]
    public double Latitude { set; get; }

    [XmlAttribute]
    public double Longitude { set; get; }

    [XmlAttribute]
    public string Address { set; get; }

    [XmlAttribute]
    public string Website { set; get; }

    [XmlIgnore]
    public double DistanceToCenter { set; get; }

    [XmlIgnore]
    public double DistanceToUser { set; get; }
}
```

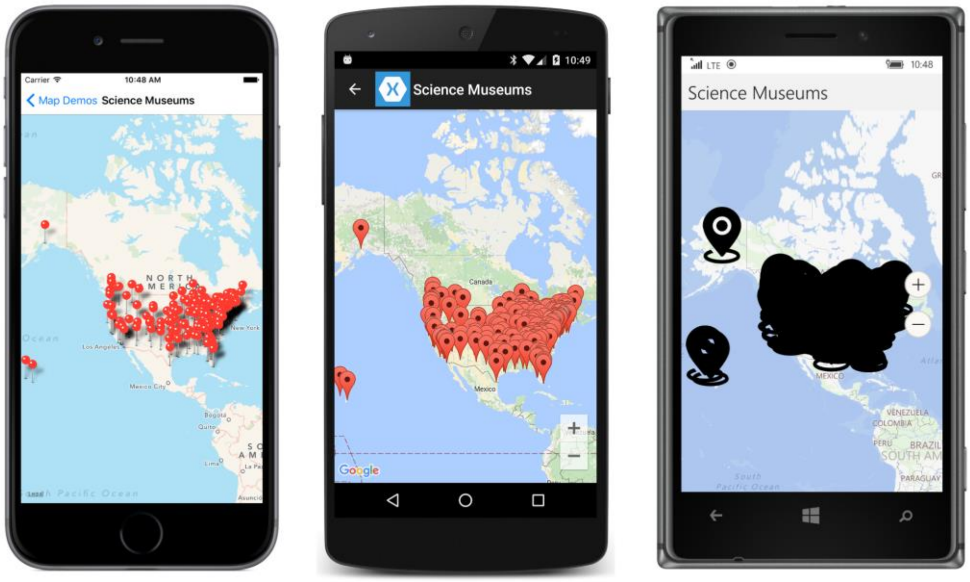
As you can see, the first six of these properties correspond to the attributes of the `Site` elements in the XML file. You'll see how the last two properties are used shortly.

The `Locations` class defines a `Sites` property that is a collection of `Site` objects:

```
public class Locations
{
    public Locations()
    {
        Sites = new List<Site>();
    }

    public string Title { set; get; }

    public List<Site> Sites { set; get; }

    public static Locations Load(string resource)
    {
        Assembly assembly = typeof(Locations).GetTypeInfo().Assembly;

        using (Stream stream = assembly.GetManifestResourceStream(resource))
        {
            using (StreamReader reader = new StreamReader(stream))
            {
                XmlSerializer xmlSerializer = new XmlSerializer(typeof(Locations));
                Locations locations = (Locations)xmlSerializer.Deserialize(stream);

                return locations;
            }
        }
    }
}
```

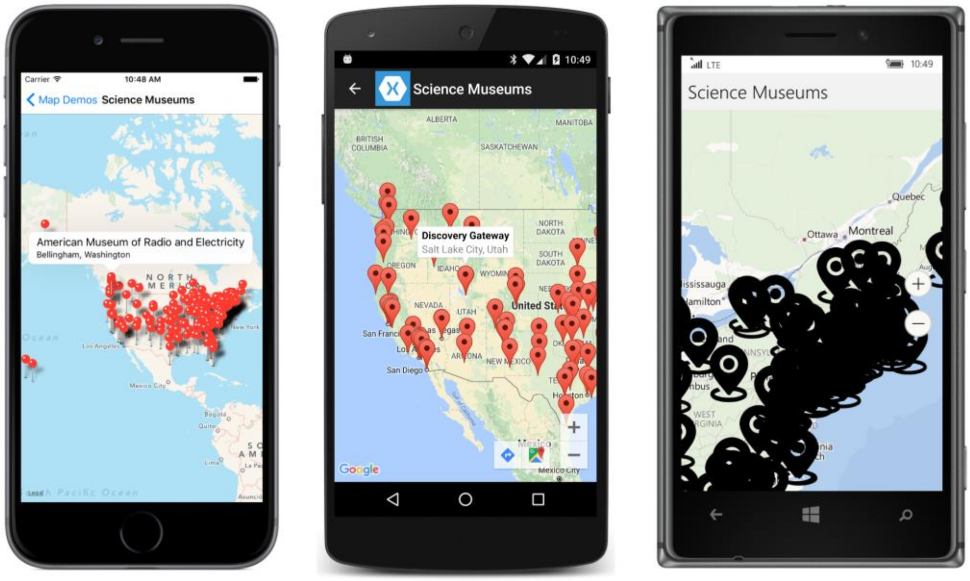
In addition, the `Locations` class defines a static `Load` method that deserializes an application resource to a `Locations` object. This resource is the ScienceMuseums.xml file.

Let's first look at the program in operation. When the `ScienceMuseumsPage` first appears, the map is zoomed out sufficiently to display all 253 locations of science museums in the United States:



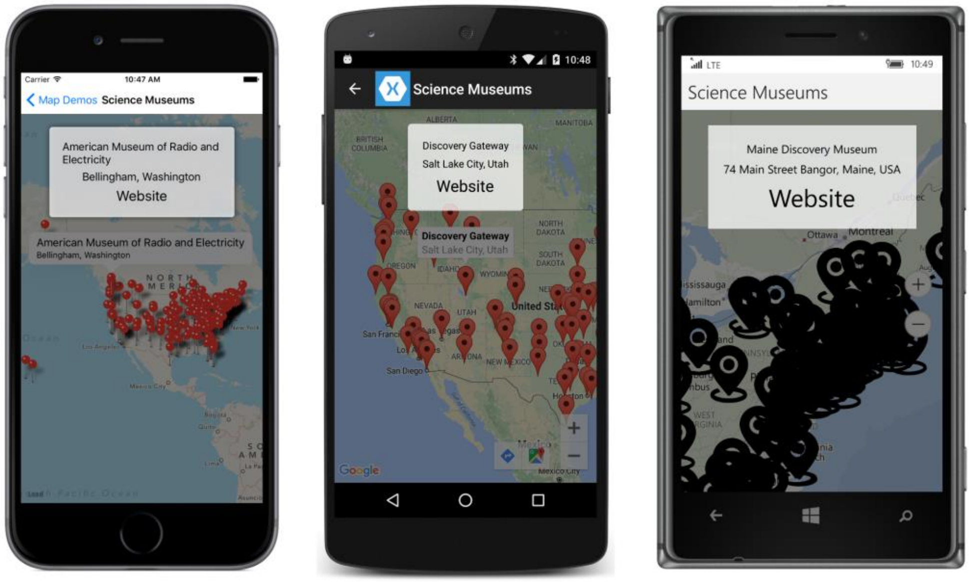
The pins look very different on the three platforms. You can use your fingers normally to pan and pinch this map.

On iOS and Android, tapping any pin causes the map to display the `Label` and `Address` properties in the form of a banner:

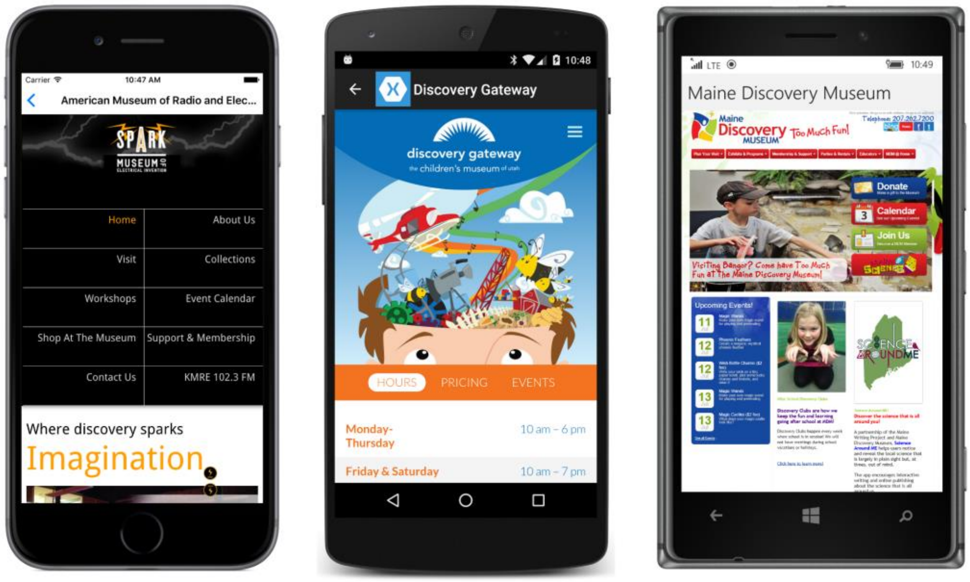


This information is displayed by the underlying native map object and you have no control over it except in specifying the `Label` and `Address` properties. Tapping anywhere on the map causes the banner to disappear.

On iOS and Android, tapping the pin or banner again (or on the Universal Windows Platform, simply tapping the pin) triggers the `Clicked` event of the `Pin`. You can handle this event in whatever way you want, including navigating to another page. The `ScienceMuseumsPage` chooses to display a popup with additional information:



You can tap anywhere on the screen to dismiss this popup. Or you can tap the word **Website** to navigate to a new page that uses a Xamarin.Forms `WebView` to display that particular science museum's website:



Because the application can navigate to any website listed in the ScienceMuseums.xml file, the iOS application must have permission for doing so. This requires the following dictionary key in the `dict` section of the Info.plist file:

```
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSAllowsArbitraryLoads</key>
    <true/>
</dict>
```

The XAML file for the `ScienceMuseumsPage` class defines a single-cell `Grid` that contains all the program's visuals—the `Map` and an overlapping `AbsoluteLayout` with a `Frame` that functions as a popup containing a `StackLayout` with `Label` elements:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:maps="clr-namespace:Xamarin.Forms.Maps;assembly=Xamarin.Forms.Maps"
             x:Class="MapDemos.ScienceMuseumsPage"
             Title="Science Museums">
    <Grid>
        <maps:Map x:Name="map" />

        <AbsoluteLayout x:Name="absLayout"
                        InputTransparent="True">

            <AbsoluteLayout.GestureRecognizers>
                <TapGestureRecognizer Tapped="OnAbsoluteLayoutTapped" />
            </AbsoluteLayout.GestureRecognizers>

            <Frame x:Name="popup"
                   BackgroundColor="#C0FFFFFF"
                   Margin="20"
                   IsVisible="False"
                   AbsoluteLayout.LayoutFlags="XProportional"
                   AbsoluteLayout.LayoutBounds="0.5, 0">

                <StackLayout>
                    <StackLayout.Resources>
                        <ResourceDictionary>
                            <Style TargetType="Label">
                                <Setter Property="TextColor" Value="Black" />
                                <Setter Property="HorizontalOptions" Value="Center" />
                            </Style>
                        </ResourceDictionary>
                    </StackLayout.Resources>

                    <Label Text="{Binding LongName}" />
                    <Label Text="{Binding Address}" />
                    <Label Text="Website"
                           FontSize="Large">
                        <Label.GestureRecognizers>
                            <TapGestureRecognizer Tapped="OnLinkLabelTapped" />
                        </Label.GestureRecognizers>
                    </Label>
                </StackLayout>
            </Frame>
        </AbsoluteLayout>
    </Grid>
</ContentPage>
```

The `InputTransparent` property of the `AbsoluteLayout` is set to `True`, so it won't interfere with the `Map`. The `IsVisible` property of the `Frame` is set to `False`, so it is initially invisible.

The code-behind file calls `MoveToRegion` to zoom the map to a central location, so all of the United States is visible including Alaska and Hawaii. A `foreach` loop creates all the `Pin` objects and adds them to the `Pins` collection of the `Map` object. Notice that the `BindingContext` of each `Pin` object is set to the `Site` object that contains all the available information about the particular museum:

```
public partial class ScienceMuseumsPage : ContentPage
{
    // Load in the data.
    Locations locations = Locations.Load("MapDemos.Data.ScienceMuseums.xml");

    public ScienceMuseumsPage()
    {
        InitializeComponent();

        // Center the U.S. in the map
        map.MoveToRegion(new MapSpan(new Position(45, -110), 50, 100));

        // Create the pins
        foreach (Site site in locations.Sites)
        {
            Pin pin = new Pin
            {
                BindingContext = site,
                Label = site.Name,
                Position = new Position(site.Latitude, site.Longitude),
                Address = site.Address
            };

            pin.Clicked += OnPinClicked;
            map.Pins.Add(pin);
        }
    }

    void OnPinClicked(object sender, EventArgs args)
    {
        absLayout.BackgroundColor = new Color(0, 0, 0, 0.5);
        absLayout.InputTransparent = false;
        popup.BindingContext = (sender as Pin).BindingContext;
        popup.IsVisible = true;
    }

    void OnAbsoluteLayoutTapped(object sender, EventArgs args)
    {
        popup.IsVisible = false;
        popup.BindingContext = null;
        absLayout.InputTransparent = true;
        absLayout.BackgroundColor = Color.Transparent;
    }

    async void OnLinkLabelTapped(object sender, EventArgs args)
    {
        Site site = ((sender as Label).BindingContext as Site);

        await Navigation.PushAsync(new ContentPage
        {
            Title = site.Name,
            Content = new WebView
            {
                Source = site.Website
            }
        });
    }
}
```

The code-behind file also contains three event handlers:

The `Clicked` handler for the `Pin` displays the popup, which is a `Frame` named `popup`. To block all input to the `Map` object, the code sets the `InputTransparent` property of the `AbsoluteLayout` to `False`, and gives it a semi-transparent black color. It then displays the popup by setting the `Bind-ingContext` of the `Frame` and setting `IsVisible` to `true`.

If the user dismisses the popup by tapping anywhere on the screen, the `OnAbsoluteLayout-Tapped` method reverses everything the `OnPinClicked` method did.

If instead the user taps the word **Website** on the popup, then the `OnLinkLabelTapped` method navigates to a new page displaying the museum's website. Notice that this handler can obtain the `Site` object associated with the popup simply by accessing the `BindingContext` of the `Label`, which is inherited from the `BindingContext` of the `Frame`.

This distance between two points

When you use pins to indicate locations of various landmarks, you probably want to restrict the number of pins displayed on the map at any one time. Perhaps you only want to display the ten museums closest to the user's location, or the ten museums closest to the center of the displayed map.

In either case, this job requires that your application be capable of computing distances between two geographic locations. In the general case, this distance should be based on how long it takes to drive, or to use public transportation, or to walk, but that information requires using a map service. A much easier calculation is sometimes called in English "as the crow flies" or "in a beeline." This is the geodesic distance that corresponds to the distance along the arc of the circumference of the great circle that passes through those two points.

The formula for the geodesic distance is rather messy if you take into account the true shape of the Earth, but if you assume the Earth is a sphere, it's not too bad. Let φ (the Greek letter phi) represent the latitude and λ (the Greek letter lambda) be the longitude. Those are the standard mathematical symbols for these two quantities. You can calculate the angular distance (*d*) between two points (φ_1, λ_1) and (φ_2, λ_2) like this:

$$\cos d = \sin \varphi_1 \sin \varphi_2 + \cos \varphi_1 \cos \varphi_2 \cos(\lambda_1 - \lambda_2)$$

Both this formula and the more complex formula for an oblate spheroid can be found in Jean Meeus's classic book *Astronomical Algorithms*, 2nd edition (Willman-Bell, 1998), on pages 84–85.

The value *d* is an angle with a vertex in the center of the Earth and the two sides passing through the two points on the surface of the Earth. It's usually converted to a distance in kilometers or miles by dividing by 360 degrees and multiplying by the circumference of the Earth. Or, if the angle is conveniently in radians, you can simply multiply the angle by the radius of the Earth.

The `PositionExtensions` class in the **Xamarin.FormsBook.Toolkit.Maps** assembly contains a `DistanceTo` method that calculates the geodesic distance between two `Position` values and returns a `Distance` value:

```
namespace Xamarin.FormsBook.Toolkit.Maps
{
    public static class PositionExtensions
    {
        …
        public static Distance DistanceTo(this Position self, Position other)
        {
            // Angle in radians along great circle.
            double radians = Math.Acos(
                Math.Sin(ToRadians(self.Latitude)) * Math.Sin(ToRadians(other.Latitude)) +
                Math.Cos(ToRadians(self.Latitude)) * Math.Cos(ToRadians(other.Latitude)) *
                    Math.Cos(ToRadians(self.Longitude - other.Longitude))
                );

            // Multiply by the Earth's radius to get the actual distance.
            return Distance.FromKilometers(6371 * radians);
        }

        static double ToRadians(double angle)
        {
            return Math.PI * angle / 180;
        }
    }
}
```

The XAML file for the `LocalMuseumsPage` is similar to `ScienceMuseumsPage`, but also includes the `MyLocationButton` developed in this chapter, and another `Label` with a binding to the `DistanceToUser` property of the `Site` class:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:maps="clr-namespace:Xamarin.Forms.Maps;assembly=Xamarin.Forms.Maps"
             xmlns:local="clr-namespace:MapDemos"
             x:Class="MapDemos.LocalMuseumsPage"
             Title="Local Museums">
    <Grid>
        <maps:Map x:Name="map"
                  IsShowingUser="True" />

        <local:MyLocationButton x:Name="myLocationButton"
                                IsVisible="False"
                                Margin="10"
                                HorizontalOptions="End"
                                VerticalOptions="Start"
                                Clicked="OnLocationButtonClicked" />

        <AbsoluteLayout x:Name="absLayout"
                        InputTransparent="True">

            <AbsoluteLayout.GestureRecognizers>
                <TapGestureRecognizer Tapped="OnAbsoluteLayoutTapped" />
            </AbsoluteLayout.GestureRecognizers>

            <Frame x:Name="popup"
                   BackgroundColor="#C0FFFFFF"
                   Margin="20"
                   IsVisible="False"
                   AbsoluteLayout.LayoutFlags="XProportional"
                   AbsoluteLayout.LayoutBounds="0.5, 0">
                <StackLayout>
                    <StackLayout.Resources>
                        <ResourceDictionary>
                            <Style TargetType="Label">
                                <Setter Property="TextColor" Value="Black" />
                                <Setter Property="HorizontalOptions" Value="Center" />
                            </Style>
                        </ResourceDictionary>
                    </StackLayout.Resources>

                    <Label Text="{Binding LongName}" />
                    <Label Text="{Binding Address}" />
                    <Label Text="{Binding DistanceToUser,
                                      StringFormat='{0:F1} miles away'}" />
                    <Label Text="Website"
                           FontSize="Large">
                        <Label.GestureRecognizers>
                            <TapGestureRecognizer Tapped="OnLinkLabelTapped" />
                        </Label.GestureRecognizers>
                    </Label>
                </StackLayout>
            </Frame>
        </AbsoluteLayout>
    </Grid>
</ContentPage>
```

The constructor in the code-behind file does *not* create the `Pin` objects. Instead, it uses the `ILocationTracker` dependency service to track the user's location, and it starts a timer to obtain the displayed center of the map:

```
public partial class LocalMuseumsPage : ContentPage
{
    const int NUM_VISIBLE = 10;
    static readonly Distance RADIUS = Distance.FromMiles(100);

    Locations locations = Locations.Load("MapDemos.Data.ScienceMuseums.xml");
    ILocationTracker locationTracker;
    Position userPosition;
    Position mapCenter;

    public LocalMuseumsPage()
    {
        InitializeComponent();

        // Center the map on the 48-states geographic center.
        map.MoveToRegion(MapSpan.FromCenterAndRadius(
            new Position(39.828175, -98.579500), RADIUS));

        // Track user's location.
        locationTracker = DependencyService.Get<ILocationTracker>();
        locationTracker.LocationChanged += OnLocationTracker;

        // Track map center.
        Device.StartTimer(TimeSpan.FromSeconds(1), OnTimerCallback);
    }
    …
}
```

The location tracker needs to be running on all the platforms. It only treats Android differently by not enabling the MyLocationButton. Whenever the location changes, the OnLocationTracker event handler stores the distance between the user's location and each museum's location in the DistanceToUser property of the Site object:

```
public partial class LocalMuseumsPage : ContentPage
{
    …
    // Location tracking and go-to location methods.
    void OnLocationTracker(object sender, GeographicLocation args)
    {
        userPosition = new Position(args.Latitude, args.Longitude);

        // Determine distances between the user and museums.
        foreach (Site site in locations.Sites)
        {
            Position sitePosition = new Position(site.Latitude, site.Longitude);
            site.DistanceToUser = sitePosition.DistanceTo(userPosition).Miles;
        }

        myLocationButton.IsVisible = Device.OS != TargetPlatform.Android;
    }

    protected override void OnAppearing()
    {
        base.OnAppearing();
        locationTracker.StartTracking();
    }

    protected override void OnDisappearing()
    {
        base.OnDisappearing();
        locationTracker.PauseTracking();
    }

    void OnLocationButtonClicked(object sender, EventArgs args)
    {
        map.MoveToRegion(MapSpan.FromCenterAndRadius(userPosition, RADIUS));
    }
    …
}
```

The OnLocationButtonClicked method is only called for iOS and the Windows platforms and re-centers and re-zooms the map to the user's location.

The timer callback is called every second to obtain the new location of the map's center. The map might have been moved by the user pressing the MyLocationButton, or by manually panning or stretching the map. The goal is to display pins of the ten museums closest to the center of the map.

The strategy is to first calculate the distance between the center of the map and each museum, and to store that distance in the DistanceToCenter property of each Site object. Then the Sites collection can be sorted by the DistanceToCenter property. The museums with the ten shortest distances become Pin objects.

To simplify the logic, there's a temptation to simply clear the Pins collection of the Map and create ten new Pin objects. However, that approach isn't optimum. If the user has displayed the banner of a Pin, and a call to OnTimerCallback occurs that clears the Pins collection, then that Pin and the banner will disappear. The Pin might quickly return, and the switch might be imperceptible, but it's a different object and the banner will no longer be displayed.

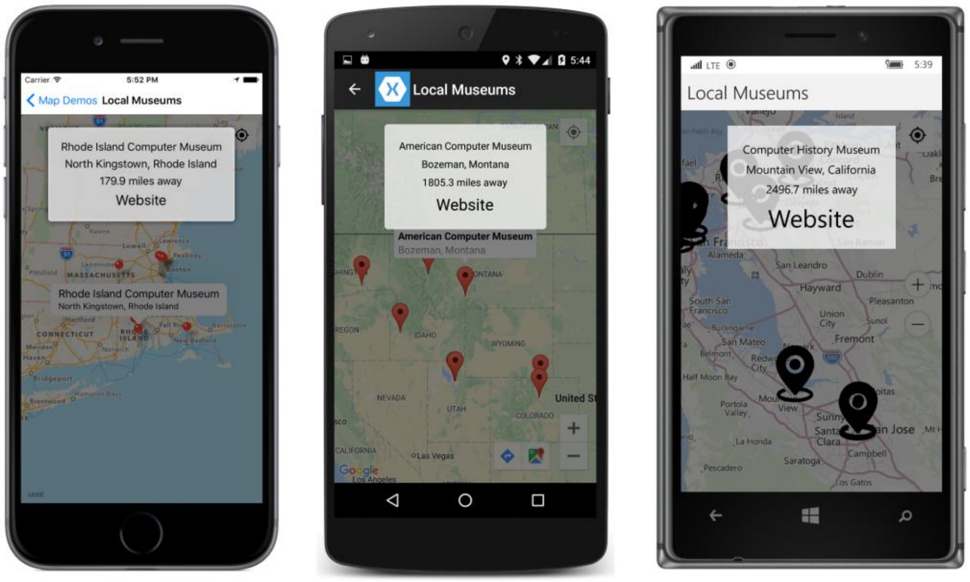
For that reason, the logic to remove and add pins attempts to preserve Pin objects that should remain in the Pins collection. It first removes Pin objects that represent museums no longer among the ten closest museums, and then creates new Pin objects for museums not in the Pins collection but which are now among the ten closest.

```
public partial class LocalMuseumsPage : ContentPage
{
    …
    // Track the map center to create and detach pins.
    bool OnTimerCallback()
    {
        if (map.VisibleRegion == null)
            return true;

        Position mapCenter = map.VisibleRegion.Center;

        if (this.mapCenter != mapCenter)
        {
            this.mapCenter = mapCenter;

            // Loop through sites and calculate distance from map center.
            foreach (Site site in locations.Sites)
            {
                Position sitePosition = new Position(site.Latitude, site.Longitude);
                site.DistanceToCenter = sitePosition.DistanceTo(mapCenter).Miles;
            }

            // Sort by distance.
            locations.Sites.Sort((site1, site2) =>
                site1.DistanceToCenter.CompareTo(site2.DistanceToCenter));

            // Remove pins not in the top 10.
            List<Pin> removeList = new List<Pin>();

            foreach (Pin pin in map.Pins)
            {
                bool match = false;

                for (int i = 0; i < NUM_VISIBLE; i++)
                {
                    match |= pin.BindingContext == locations.Sites[i];
                }

                if (!match)
                {
                    removeList.Add(pin);
                }
            }

            foreach (Pin pin in removeList)
            {
                pin.Clicked -= OnPinClicked;
                map.Pins.Remove(pin);
            }

            // Add pins from the top 10.
            for (int i = 0; i < NUM_VISIBLE; i++)
            {
                Site site = locations.Sites[i];
                bool match = false;

                foreach (Pin pin in map.Pins)
                {
                    match |= pin.BindingContext == site;
                }

                if (!match)
                {
                    Pin newPin = new Pin
                    {
                        BindingContext = site,
                        Label = site.Name,
                        Position = new Position(site.Latitude, site.Longitude),
                        Address = site.Address
                    };

                    newPin.Clicked += OnPinClicked;
                    map.Pins.Add(newPin);
                }
            }
        }
        return true;
    }
    …
}
```

The remainder of the code-behind file contains the three event handlers from the `ScienceMuseums-Page`.

There are three advantages of `LocalMuseumsPage` over `ScienceMuseumsPage`: First, `LocalMuse-umsPage` allows moving the map to the user's location and displaying that location with a blue dot. Second, the map isn't cluttered with hundreds of markers, but simply displays the ten museums closest to the map center.

The third advantage is that the little popup now includes the distance of the museum from the user's location, even if it might be hundreds or thousands of miles away:



Geocoding and back again

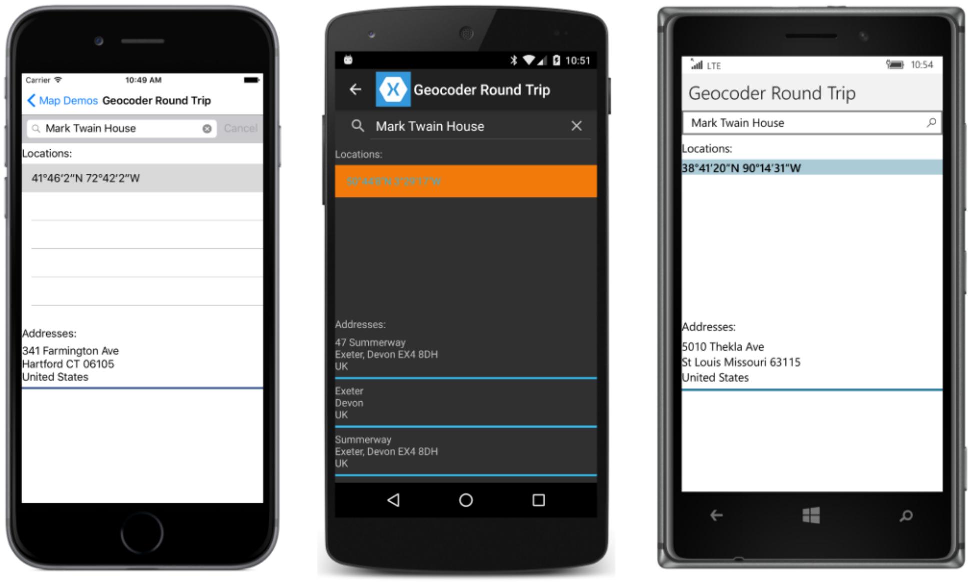
The final feature of the **Xamarin.Forms.Maps** assembly is the `Geocoder` class that converts addresses into geographic positions, and geographic positions into addresses. This class contains two methods:

- `Task<IEnumerable<Position>> GetPositionsForAddressAsync(String address)`
- `Task<IEnumerable<String>> GetAddressesForPositionAsync(Position position)`

Both methods are asynchronous, and both return `IEnumerable` collections. `GetPositionsFor-AddressAsync` returns zero or more `Position` values for a particular address expressed as a string, and `GetAddressesForPositionAsync` returns one or more addresses in the form of strings for a specified `Position` value.

This facility as implemented in Xamarin.Forms is probably not as good as similar services you've doubtlessly encountered on the web or your phone. You'll need to determine for yourself whether these two methods will be adequate for your needs or not, and whether you'll need to explore alternative web services. The final sample in this chapter is called `GeocoderRoundTripPage`, and its sole purpose is to help you make this determination.

The page contains a `SearchBar` that lets you enter an address. The zero or more `Position` values returned from the `GetPositionsForAddressAsync` method are displayed in a `ListView`. You can then select one of those `Position` values, and the zero or more addresses returned from `Get-AddressesForPositionAsync` are then displayed in a `StackLayout` in a `ScrollView`. Here are the results when you enter **Mark Twain House** and select the position in the `ListView`:



Locations:
41°46'2"N 72°42'2"W

Addresses:
341 Farmington Ave
Hartford CT 06105
United States

Locations:

Addresses:
47 Summerway
Exeter, Devon EX4 8DH
UK

Exeter
Devon
UK

Summerway
Exeter, Devon EX4 8DH
UK

Locations:
38°41'20"N 90°14'31"W

Addresses:
5010 Thekla Ave
St Louis Missouri 63115
United States

It should be noted that the web-based versions of Google Maps and Bing Maps both agree with the iPhone that the Mark Twain House is in Hartford Connecticut, so obviously the service you are using here is not what users get on the web.

The XAML file contains the `SearchBar`, `ListView`, and `ScrollView`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:toolkit=
         "clr-namespace:Xamarin.FormsBook.Toolkit.Maps;assembly=Xamarin.FormsBook.Toolkit.Maps"
             x:Class="MapDemos.GeocoderRoundTripPage"
             Title="Geocoder Round Trip">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <SearchBar Grid.Row="0"
                   Placeholder="Enter location"
                   SearchButtonPressed="OnSearchButtonPressed" />

        <Label Grid.Row="1"
               Text="Locations:" />

        <ListView x:Name="positionsListView"
                  Grid.Row="2"
                  ItemSelected="OnPositionItemSelected">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <TextCell>
                        <TextCell.Text>
                            <Binding Path=".">
                                <Binding.Converter>
                                    <toolkit:PositionFormatConverter Format="S0" />
                                </Binding.Converter>
                            </Binding>
                        </TextCell.Text>
                    </TextCell>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>

        <Label Grid.Row="3"
               Text="Addresses:" />

        <ScrollView Grid.Row="4">
            <StackLayout x:Name="addressesStack" />
        </ScrollView>
    </Grid>
</ContentPage>
```

Notice the `PositionFormatConverter` that is intended to format the `Position` objects in the `ListView`. This is necessary because the `StringFormat` property of `BindingBase` will not find the `ToString` extension method defined in the **Xamarin.FormsBook.Toolkit.Maps** library. Here's the `PositionFormatConverter` defined in that same library:

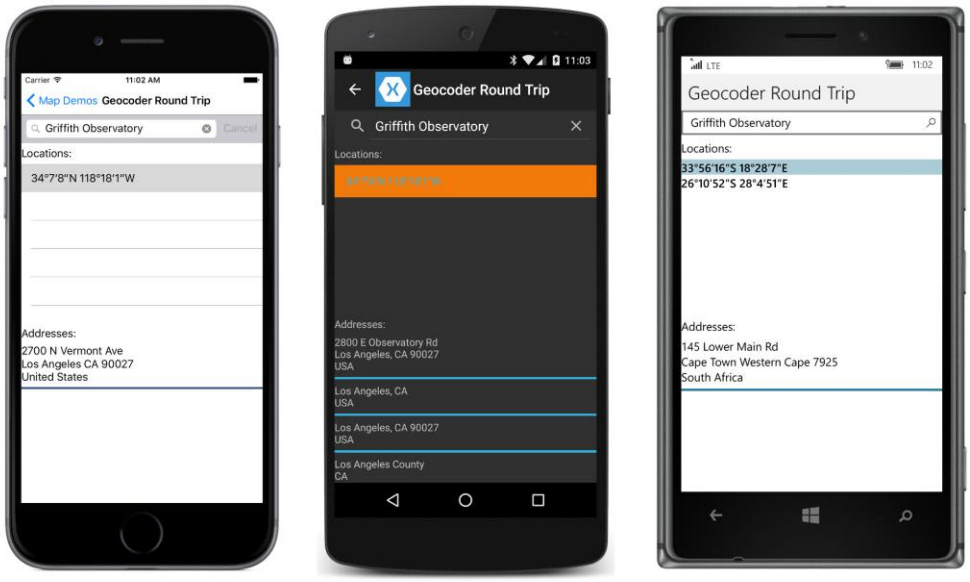
```
namespace Xamarin.FormsBook.Toolkit.Maps
{
    public class PositionFormatConverter : IValueConverter
    {
        public string Format { set; get; }

        public object Convert(object value, Type targetType,
                              object parameter, CultureInfo culture)
        {
            if (value == null)
                return "{null}";

            Position position = (Position)value;
            return position.ToString(Format ?? "S");
        }

        public object ConvertBack(object value, Type targetType,
                                  object parameter, CultureInfo culture)
        {
            return new Position();
        }
    }
}
```

The code-behind file for `GeocoderRoundTripPage` handles the `SearchButtonPressed` event from the `SearchBar` and the `ItemSelected` event from the `ListView`:

```
public partial class GeocoderRoundTripPage : ContentPage
{
    Geocoder geocoder = new Geocoder();

    public GeocoderRoundTripPage()
    {
        InitializeComponent();
    }

    async void OnSearchButtonPressed(object sender, EventArgs args)
    {
        string address = ((SearchBar)sender).Text;
        IEnumerable<Position> positions = await geocoder.GetPositionsForAddressAsync(address);
        positionsListView.ItemsSource = positions;

        // And clear out the list of addresses.
        addressesStack.Children.Clear();
    }

    async void OnPositionItemSelected(object sender, SelectedItemChangedEventArgs args)
    {
        Position position = (Position)args.SelectedItem;
        IEnumerable<string> addresses = await geocoder.GetAddressesForPositionAsync(position);

        addressesStack.Children.Clear();

        foreach (string address in addresses)
        {
            addressesStack.Children.Add(
                new Label
                {
                    Text = address
                });
            addressesStack.Children.Add(
                new BoxView
                {
                    HeightRequest = 3,
                    Color = Color.Accent
                });
        }
    }
}
```

Notice that the `OnPositionItemSelected` handler inserts a thin `BoxView` between each address. Often the addresses contain multiple lines, and the display might otherwise be a little confusing.

Let's give the `Geocoder` another chance:



Griffith Observatory

Locations:

34°7'8"N 118°18'1"W

Addresses:

2700 N Vermont Ave
Los Angeles CA 90027
United States

Griffith Observatory

Locations:

Addresses:

2800 E Observatory Rd
Los Angeles, CA 90027
USA

Los Angeles, CA
USA

Los Angeles, CA 90027
USA

Los Angeles County
CA

Griffith Observatory

Locations:

33°56'16"S 18°28'7"E
26°10'52"S 28°4'51"E

Addresses:

145 Lower Main Rd
Cape Town Western Cape 7925
South Africa

And now you can experiment with it on your own, or perhaps you've seen enough.

The **Xamarin.Forms.Maps** library provides the basics, but for any application that uses maps and loctions in a more sophisticated manner, let it be merely a springboard for more extensive exploration into the exciting world of map services.