ADVANCED UWP PART 1

Interacting with the real world

By Matteo Pagani

Syncfusion®

Table of Contents

In this white paper you will understand how you can interact with the real world in a Windows 10 application developed with the Universal Windows Platform. You will learn how to use the location services to identify the position of the user, display it on a map, and provide advanced interactions like displaying a point of interest or a route on a map; how to understand when a user enters or exits in a specific location; how to work with the motion sensors that many devices (especially the portable ones) have to offer, like accelerometer, gyroscope, and compass; and more.

Geolocation services

The geolocation services offered by the Universal Windows Platform can be used to track a user's position in the world, no matter which device the app is running on. Windows can determine the current position using different approaches: by using a GPS sensor, by getting the position from the nearest mobile cell (if the device has a SIM), or based on the IP returned by the Wi-Fi connection. The approach leveraged by the system is transparent to the user. The Universal Windows Platform will take care of choosing the best one for us according to the requirements and to the hardware configuration of the device.

Note: To use the geolocation service, you need to enable the Location capability in the manifest file.

The main class we're going to use to interact with the geolocation services is called **`Geolocator`**, which is part of the **`Windows.Devices.Geolocation`** namespace.

Retrieving the status of the services

The availability of geolocations services can't be taken for granted. The user can decide not to give permission to our application to access to the user's location.

This scenario is automatically handled by Windows. The first time we use the geolocation APIs, Windows will ask the user if they want to give the app access to location services. However, as developers, the best approach is to manually check if we have received the proper permission by leveraging a new static method, introduced in Windows 10 and offered by the **`Geolocator`** class, called **`RequestAccessAsync()`**:

```
private async void Button_Click(object sender, RoutedEventArgs e)
{
    GeolocationAccessStatus status = await Geolocator.RequestAccessAsync();
    if (status == GeolocationAccessStatus.Allowed)
    {
        //use the geolocation APIs
    }
}
```

When you call it, the user will be prompted with the following message:

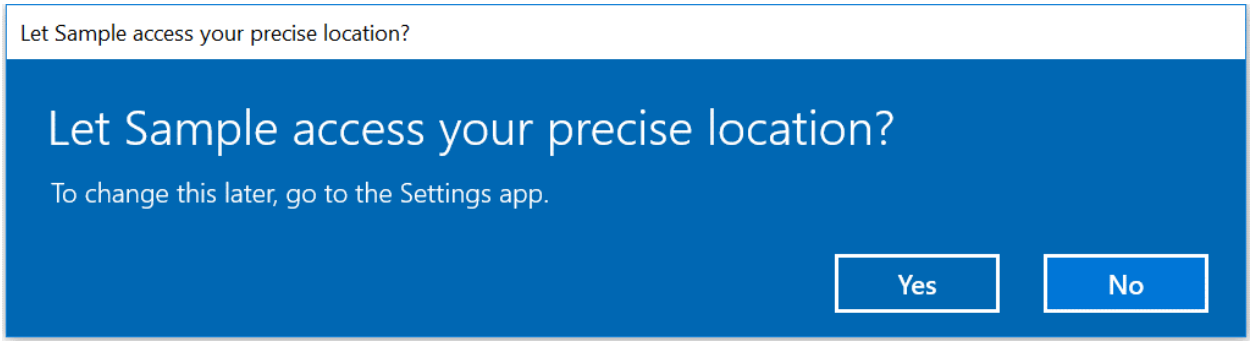

Figure 1: The app request to access to the user's location.

Only if the user has chosen **Yes** will you receive in return the value **Allowed** of the enumerator **GeolocationAccessStatus**. It's important to highlight that this prompt will appear only the first time. If you call the **RequestAccessAsync()** method again, but the user has already granted the permission, you will automatically get **Allowed** and Windows won't show the pop-up again. However, it's good practice to always check the status of the request before using the location APIs. The users can, at any time, go into Windows settings and, in the **Privacy** section, disable the location access to the app, even if they have previously granted it. Then you'll start to get **Denied** as the value of the **GeolocationAccessStatus** enumerator.

Retrieving the user's position

The simplest way to determine the user's position is to use the **GetGeopositionAsync()** method exposed by the **Geolocator** class, which performs a single acquisition. The method returns a **Geoposition** object containing all the information about the current location inside the **Point** property. For example, this property exposes an object called **Position**, which you can use to retrieve information like **Latitude**, **Longitude**, and **Altitude**.

```
private async void OnGetPositionClicked(object sender, RoutedEventArgs e)
{
    Geolocator geolocator = new Geolocator();
    Geoposition position = await geolocator.GetGeopositionAsync();
    Longitude.Text =
position.Coordinate.Point.Position.Longitude.ToString();
    Latitude.Text = position.Coordinate.Point.Position.Latitude.ToString();
}
```

Another approach is to subscribe to an event called **PositionChanged**. This way, we can track the user's position and be notified every time they change position. We can customize the frequency of the notification with two parameters: **MovementThreshold** (the distance, in meters, that should be travelled from the previous point) and **ReportInterval** (the timeframe, in milliseconds, that should pass between one notification and the other). Here is a sample code to continuously track the user's position:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    Geolocator geolocator = new Geolocator();
    geolocator.MovementThreshold = 100;
    geolocator.ReportInterval = 1000;
    geolocator.PositionChanged += geolocator_PositionChanged;
}

async void geolocator_PositionChanged(Geolocator sender,
PositionChangedEventArgs args)
{
    await Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
    {
        Geoposition currentPosition = args.Position;
        Latitude.Text =
currentPosition.Coordinate.Point.Position.Latitude.ToString();
        Longitude.Text =
currentPosition.Coordinate.Point.Position.Longitude.ToString();
    });
}
```

The event handler we created to handle the **PositionChanged** event contains a parameter with a property called **Position**. It's a **Geoposition** object with the coordinates of the current user's position. Please note that we're using the **Dispatcher** to display this data in the page: it's required, since the **PositionChanged** event is managed in a background thread, while the controls placed in the page are handled by the UI thread.

Checking the status of the location services

The **Geolocator** class also offers an event called **StatusChanged**, which is triggered when the status of the services changes. This is another reason not to take for granted the availability of location services. The users can find themselves in areas with no GPS coverage or no Internet connection and, consequently, the device wouldn't be able to retrieve the current location.

```
private async void Button_Click(object sender, RoutedEventArgs e)
{
    GeolocationAccessStatus status = await Geolocator.RequestAccessAsync();
    if (status == GeolocationAccessStatus.Allowed)
    {
        Geolocator locator = new Geolocator();
        locator.StatusChanged += Locator_StatusChanged;
    }
}

private void Locator_StatusChanged(Geolocator sender,
StatusChangedEventArgs args)
{
    switch (args.Status)
    {
        //handle the various statuses
    }
}
```

By subscribing to this event and by checking the value of the **LocationStatus** property, you'll get in return an enumerator of type **PositionStatus**, which contains different values based on the location services' status.

The supported values are:

- **Ready**: the location platform is active and it's returning valid data.
- **Initializing**: the location platform is attempting to localize the user.
- **NoData**: the location platform can't retrieve the current position.
- **Disabled**: access to the location platform is disabled.
- **NotInitialized**: the location platform hasn't tried to localize the user yet.
- **NotAvailable**: the location platform isn't available on the current device.

Geofencing

Geofencing is a feature related to the geolocation services that gives developers the opportunity to define one or more circular areas on the map (called geofences). When the user enters or exits the area, your application will be notified even if it's not running at that moment, thanks to background tasks. There are many scenarios where geofencing can be useful. For example, an application connected to a store brand could notify the user that a special sale is running when they walk near one of their shops.

An application can create up to 20,000 geofences, even if it's suggested that you not exceed the 1000 quota, or else they could become hard to manage for the operating system. Every geofence is identified by the **Geofence** class, which is part of the **Windows.Devices.Geolocation.Geofencing** namespace. When you create a new **Geofence** object, you need to set the following parameters:

- The geofence ID, which is a string that univocally identifies the geofence. An application can't create two geofences with the same ID.
- The geofence area, by using a **Geocircle** object, represented by a pair of coordinates (latitude and longitude) and the radius of the area. It's best not to create geofences with an area radius smaller than 100 meters, since they could be hard to track for the device.
- Which events should trigger the notification, by using one of the values of the **MonitoredGeofenceStates** enumerator. The possible states are **Entered** (the user entered the area), **Exited** (the user left the area), and **Removed** (the geofence has been removed from the system).
- A **bool** value, used to define if it's a one-shot geofence. If the value is **true**, the geofence will be automatically removed from the system once the user has entered or exited from the area.
- A time interval (called dwell time), which must be spent inside or outside the area for the notification to be triggered.
- A start date and time. If they're not specified, the geofence will be immediately activated.
- A time interval after the start date. If it's specified, the geofence will be automatically removed from the system when this interval has ended.

When the **Geofence** object has been properly configured, you can add it to the collection by using the **GeofenceMonitor** class, which has a unique instance (stored in the **Current** property) that is shared among all the applications. Here is a complete geofencing sample:

```
private void OnSetGeofenceClicked(object sender, RoutedEventArgs e)
{
    string id = "geofenceId";
    BasicGeoposition geoposition = new BasicGeoposition();
    geoposition.Latitude = 45.8040;
    geoposition.Longitude = 009.0838;
    double radius = 500;
    Geocircle geocircle = new Geocircle(geoposition, radius);
    MonitoredGeofenceStates mask = 0;
    mask |= MonitoredGeofenceStates.Entered;
    mask |= MonitoredGeofenceStates.Exited;
    mask |= MonitoredGeofenceStates.Removed;
    TimeSpan span = TimeSpan.FromSeconds(2);
    Geofence geofence = new Geofence(id, geocircle, mask, true, span);
    GeofenceMonitor.Current.Geofences.Add(geofence);
}
```

In the previous sample, we've defined a geofence with the following properties:

- It's placed in a specific position on the map (defined with a **Geoposition** object) and it has a radius of 500 meters.
- It's registered to monitor all the available states, so we will be notified every time the user enters or exits the area or the geofence is removed from the system.
- It's a one-shot geofence.
- It has a dwell time set to 2 seconds. The notification will be triggered only if the user stays in or leaves the area for more than 2 seconds.

To add the **Geofence** to the system, we call the **Add()** method on the **Geofences** collection, offered by the **GeofenceMonitor** instance.

Managing the geofence activation

When the application is running in the foreground and a geofence is triggered, the system invokes an event called **GeofenceStateChanged** exposed by the **GeofenceMonitor** class.

In the event handler, you have access to a **GeofenceMonitor** object, which is able to return the list of geofences that have been triggered by the **ReadReports()** method. This method will return a collection of **GeofenceStateChangeReport** objects. Each of them contains a property called **NewState**, which will tell you the event that triggered the geofence. Look at the following sample:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    GeofenceMonitor.Current.GeofenceStateChanged +=
Current_GeofenceStateChanged;
}

private async void Current_GeofenceStateChanged(GeofenceMonitor sender,
object args)
{
    var reports = sender.ReadReports();
    foreach (GeofenceStateChangeReport report in reports)
    {
        GeofenceState state = report.NewState;
        switch (state)
        {
            case GeofenceState.Entered:
                await Dispatcher.RunAsync(CoreDispatcherPriority.Normal,
async () =>
                {
                    MessageDialog dialog = new MessageDialog("The user
entered into the area");
                    await dialog.ShowAsync();
                });
                break;
            case GeofenceState.Exited:
                await Dispatcher.RunAsync(CoreDispatcherPriority.Normal,
async () =>
                {
                    MessageDialog dialog = new MessageDialog("The user has
left the area");
                    await dialog.ShowAsync();
                });
                break;
            case GeofenceState.Removed:
```

```
                await Dispatcher.RunAsync(CoreDispatcherPriority.Normal,
async () =>
                {
                    MessageDialog dialog = new MessageDialog("The geofence
has been removed");
                    await dialog.ShowAsync();
                });
                break;
            }
        }
    }
```

For each report stored in the collection returned by the **ReadReports()** method, we check the value of the **NewState** property. In the previous sample, we used a **switch** statement to manage all the possible states, represented by the **GeofenceState** enumerator: **Entered**, **Exited**, or **Removed**. It's up to you to perform the operation that best fits your scenario. For example, you could send a notification or display a specific page to the user. In this sample, we just display a pop-up message to the user with a description of what's happened. Please note that the **GeofenceStateChanged** event is managed in a background thread. We need to use the **Dispatcher** if we want to interact with the controls that are placed in the page.

The code we just saw can also be used in a background task, which is a special class containing some code that can also be performed when the application is not running. This way, we'll always be able to detect the geofences activation, even if the user is not using our application.

Interacting with a map

One of the most common scenarios when you add support to the geolocation services in your application is to display the user's position on a map. The Universal Windows Platform offers a control called **MapControl**, which also supports offline maps. No matter what device the user has, they have the option in the Windows settings to download maps for a specific country/area/region. This way, both the official, built-in Maps app and every other third-party application that uses the **MapControl** will be able to leverage the map data, even if the device isn't connected to Internet.

Using the control

The **MapControl** control isn't part of the default Windows set, so you'll need to add the namespace it belongs to (**Windows.UI.Xaml.Maps)** in the XAML **Page** definition before using it. So, you'll need to declare it like in the following sample:

```
<Page
x:Class="MapDemo.MainPage"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:local="using:MapDemo"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:maps="using:Windows.UI.Xaml.Controls.Maps"
mc:Ignorable="d"
Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">

    <Grid>
        <maps:MapControl />
    </Grid>

</Page>
```

Publishing an application that uses the MapControl

If you try to add the previous code to your application and you launch it on your PC or phone, you'll notice that a warning message is displayed over the map, claiming that you haven't declared a valid map service token. This token is required to publish an application that uses the map control in Windows, and it's provided by the Bing Maps Dev Center. To get the token, you must follow these steps:

1. Connect to the Bing Maps Dev Center at https://www.bingmapsportal.com/.
2. Sign in using your Microsoft Account (typically, it's the same you used to register your Dev Center account, which is required to publish Universal Windows Platform apps on the Store).
3. In the top menu, you will find the item **My account**, with a subsection called **My Keys**. Choose the option **Click here to create a new key.**
4. You will be prompted with the following form:

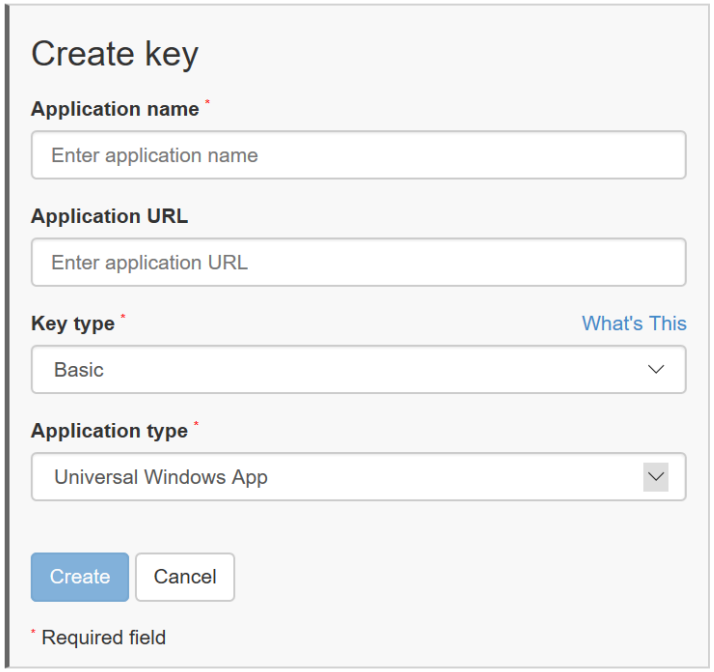

Figure 2: The form to create a new key for the MapControl.

Fill it in with the required information:

- **Application name**: the name of your application.
- **Application URL**: a website connected to your application (optional).
- **Key type**: by default, it's set to **Basic**. This is the only option available with the free plan, which gives you access to 50,000 calls per day to the Bing Maps services; more than enough both for testing and for many production scenarios. In case you think it may not be enough for your app, you can visit the Licensing page to learn more about the different paid plans.
- **Application type**: for our scenario, you must keep the default option, which is **Universal Windows App**.

5. Click **Create** and you'll get a key that you can use in your application.

The key can be connected to the **`MapControl`** control button thanks to the **`MapServiceToken`** property, like in the following sample:

```
<maps:MapControl MapServiceToken="K38JSFqmSBMVu0iTXR4aEg" />
```

Additionally, the Universal Windows Platform contains a set of APIs (that belong to the **`Windows.Services.Maps`** namespace) that can be used to interact with the services provided by Bing Maps services, like geocoding. For this scenario, you'll also have to assign the token to the **`ServiceToken`** property of the **`MapService`** class when the page is loaded.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    MapService.ServiceToken = "K38JSFqmSBMVu0iTXR4aEg";
}
```

Interacting with the map

The basic way to interact with the map control is by setting the **Center** and **ZoomLevel** properties, which define the center position of the map and the zoom level. Both properties can be set in XAML. However, the most common scenario is setting them in code, for example, to display the user's position on the map. For this purpose, the **MapControl** class offers a method called **TrySetSceneAsync()**, which automatically centers the map on a specific position with an assigned zoom level, based on the radius in meters you want to display. Optionally, the same method also accepts specifying the heading and the pitch (both in degrees), if you want deeper control over the scene. The following sample code shows how to center the map on the current user's position detected by the device:

```
private async void OnDisplayPosition(object sender, RoutedEventArgs e)
{
    GeolocationAccessStatus status = await Geolocator.RequestAccessAsync();
    if (status == GeolocationAccessStatus.Allowed)
    {
        Geolocator locator = new Geolocator();
        Geoposition result = await locator.GetGeopositionAsync();
        MapScene scene =
MapScene.CreateFromLocationAndRadius(result.Coordinate.Point, 3000);
        await MyMap.TrySetSceneAsync(scene);
    }
}
```

As you can see, we create a **MapScene** represented by a set of coordinates (of type **Geopoint**, stored in the **Point** property of the results returned by the **Geolocator** object) and a radius in meters. Both parameters are passed to the static **CreateFromLocationAndRadius()** method exposed by the **MapScene** class. Once you have a **MapScene**, you can pass it to the **TrySetSceneAsync()** method offered by the **MapControl** object. The map will be centered on the specified area with a pleasant visual display.

Displaying elements on a map

The Universal Windows Platform supports multiple ways to display elements on a map.

Displaying pushpins

The **MapControl** class supports a way to display one or more pushpins on the map by using the **MapIcon** class, representing a base pushpin rendered with a name and an image. A **MapIcon** object is identified by two properties:

- **Location**, a **Geopoint** object that identifies the pushpin's coordinates. In the following sample, we use as location the current position detected by the **Geolocation** class.
- **Title**, the label displayed over the pushpin.

By default, the **MapIcon** class uses a default image to render the pushpin. However, you can customize it by using the **Image** property, which requires a reference to the image's stream (identified by the **RandomAccessStreamReference** class). The following sample shows how to create a pushpin and place it on the map at the current position:

```
private async void OnAddPushpin(object sender, RoutedEventArgs e)
{
    GeolocationAccessStatus status = await Geolocator.RequestAccessAsync();
    if (status == GeolocationAccessStatus.Allowed)
    {
        Geolocator locator = new Geolocator();
        Geoposition geoposition = await locator.GetGeopositionAsync();
        MapIcon icon = new MapIcon();
        icon.Location = geoposition.Coordinate.Point;
        icon.Title = "Your position";
        icon.Image = RandomAccessStreamReference.CreateFromUri(new Uri("ms-
appx:///Assets/logo.jpg"));
        MyMap.MapElements.Add(icon);

        MapScene scene =
MapScene.CreateFromLocationAndRadius(geoposition.Coordinate.Point, 3000);
        await MyMap.TrySetSceneAsync(scene);

    }
}
```

The pushpin created in the previous sample has a custom title and icon, the latter being an image included in the Assets folder of the Visual Studio project. The **MapControl** class includes a collection called **MapElements** with all the elements that should be displayed in overlay over the map. To display the pushpin, we just need to add it to the collection using the **Add()** method.

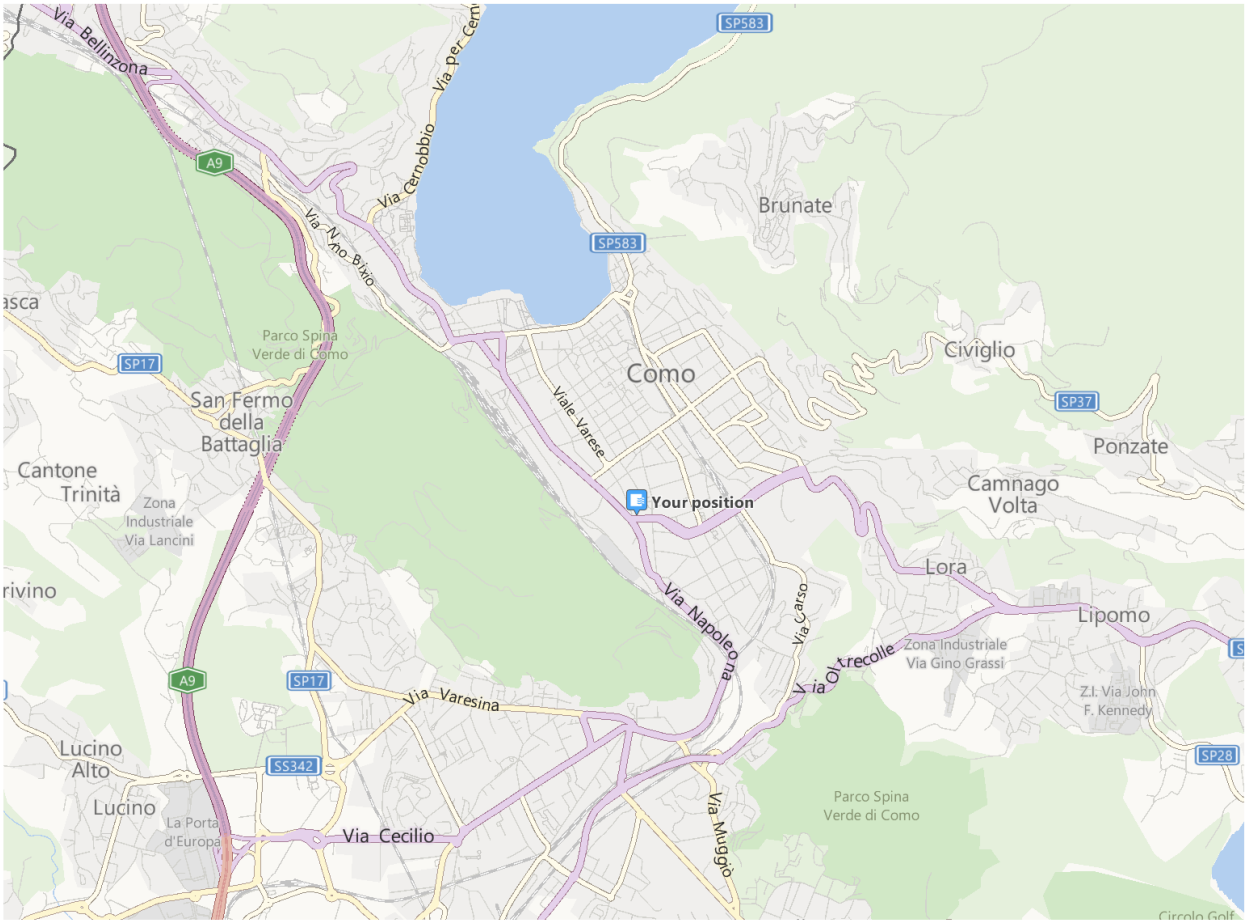

Figure 3: A pushpin placed on the map using the MapIcon control.

When you need to display a great collection of pushpins on the map, the **`MapIcon`** control is the best one to use since it offers the best performance. Windows is able to automatically hide or display different numbers of pushpins based on the zoom level and on the density of the area and, as such, it's able to display hundreds of pushpins without having a big impact on performance.

However, there's another way to create a set of pushpins that offers more flexibility to the developer. Instead of displaying just an image, we can define a custom layout using XAML and manage the user's interaction (like tapping on the pushpin). The approach is like the one we learned in the first *UWP Succinctly* about displaying collections with a **`ListView`** or a **`GridView`** control. We're going to define the template of a single pushpin and then we'll connect a collection of pushpins to the map. Each of them will be rendered using the specified template.

We can achieve this goal by using the **`MapItemsControl`** class, like in the following sample:

```
<maps:MapControl x:Name="MyMap">
    <maps:MapItemsControl x:Name="Pushpins">
        <maps:MapItemsControl.ItemTemplate>
            <DataTemplate>
                <Border Background="CornflowerBlue"
Tapped="OnPushpinClicked">
                    <TextBlock Text="{Binding Name}" Foreground="Black"
                               maps:MapControl.Location="{Binding
Location}"
                                Style="{StaticResource
TitleTextBlockStyle}" />
                </Border>
            </DataTemplate>
        </maps:MapItemsControl.ItemTemplate>
    </maps:MapItemsControl>
</maps:MapControl>
```

There's one important difference from a standard **DataTemplate**: The **MapControl** class offers an attached property, called **Location**, used to define the pushpin's position on the map. It's a **Geopoint** object that contains the coordinates of the position. Consequently, you'll also need a custom class to represent the pushpin, like in the following sample:

```
public class PushPin
{
    public string Name { get; set; }
    public Geopoint Location { get; set; }
}
```

The last step is to define a collection of **PushPin** objects and to assign it to the **ItemsSource** property of the **MapItemsControl** class, in the same way we would do with a **ListView** control:

```
private void OnAddPushpinsClicked(object sender, RoutedEventArgs e)
{
    BasicGeoposition position1 = new BasicGeoposition();
    position1.Latitude = 45.8080;
    position1.Longitude = 009.0800;
    Geopoint geopoint1 = new Geopoint(position1);
    PushPin pushpin1 = new PushPin();
    pushpin1.Name = "Pub";
    pushpin1.Location = geopoint1;

    BasicGeoposition position2 = new BasicGeoposition();
    position2.Latitude = 45.8052;
    position2.Longitude = 009.0825;
    Geopoint geopoint2 = new Geopoint(position2);
    PushPin pushpin2 = new PushPin();
    pushpin2.Name = "Bank";
    pushpin2.Location = geopoint2;

    List<PushPin> pushpins = new List<PushPin>();
    pushpins.Add(pushpin1);
    pushpins.Add(pushpin2);
    Pushpins.ItemsSource = pushpins;
}
```

Additionally, if you look at the **ItemTemplate** we've defined in XAML, you'll notice that we have subscribed to the **Tapped** event of the **Border** control. The purpose is to manage the user's interaction, so that when the user taps on a pushpin, we can display some additional information. The following sample shows how to manage this event to display a pop-up with the name of the selected location:

```
private async void OnPushpinClicked(object sender, TappedRoutedEventArgs e)
{
    Border border = sender as Border;
    PushPin selectedPushpin = border.DataContext as PushPin;
    MessageDialog dialog = new MessageDialog(selectedPushpin.Name);
    await dialog.ShowAsync();
}
```

The **Tapped** event (like any other event) provides a parameter called **sender**, the XAML control that triggered the event. Thanks to this parameter, we can get a reference to the **Border** control and access to its **DataContext**, which is the **Pushpin** object that has been selected. This way, it's easy to perform a cast and retrieve the value of the **Name** property so we can display it to the user.

However, it's important to remember that, despite being more flexible, this approach has a performance downside, since it is much more expensive for Windows to render a XAML control than just an icon. As such, this approach is suitable when the number of pushpins to display is limited.

Showing a line

Another kind of element that you can draw over a map is a line, identified by the class **`MapPolyline`**. The main property to set is called **`Path`**, which is a collection of **`BasicGeoposition`** objects. Since, in this case, we're talking about a line, the collection will be composed of just two elements: the starting and ending points.

Additionally, you have a set of properties like **`StrokeColor`**, **`StrokeThickness`**, or **`StrokeDashed`** to customize the way the line is designed over the screen. The **`MapPolyline`** object behaves like any other visual element we can draw over the map. Once we have created it, we can add it by calling the **`Add()`** method on the **`MapElements`** property of the **`MapControl`** control.

Look at the following sample:

```
private async void OnAddPushpin(object sender, RoutedEventArgs e)
{
    GeolocationAccessStatus status = await Geolocator.RequestAccessAsync();
    if (status == GeolocationAccessStatus.Allowed)
    {
        Geolocator locator = new Geolocator();
        Geoposition geoposition = await locator.GetGeopositionAsync();

        double centerLatitude =
geoposition.Coordinate.Point.Position.Latitude;
        double centerLongitude =
geoposition.Coordinate.Point.Position.Longitude;
        MapPolyline mapPolyline = new MapPolyline();
        mapPolyline.Path = new Geopath(new List<BasicGeoposition>()
        {
    new BasicGeoposition() {Latitude=centerLatitude - 0.0015,
Longitude=centerLongitude - 0.010 },
    new BasicGeoposition() {Latitude=centerLatitude + 0.0015,
Longitude=centerLongitude + 0.010 },
        });

        mapPolyline.StrokeColor = Colors.Red;
        mapPolyline.StrokeThickness = 4;
        MyMap.MapElements.Add(mapPolyline);

        MapScene scene =
MapScene.CreateFromLocationAndRadius(geoposition.Coordinate.Point, 3000);
        await MyMap.TrySetSceneAsync(scene);

    }
}
```

In this case, we're creating a line that has as starting and ending positions two points close to the current location detected by the **`Geolocator`** class. Additionally, we're customizing the look of the line by changing the color to red and making it thicker. The following image shows the result:

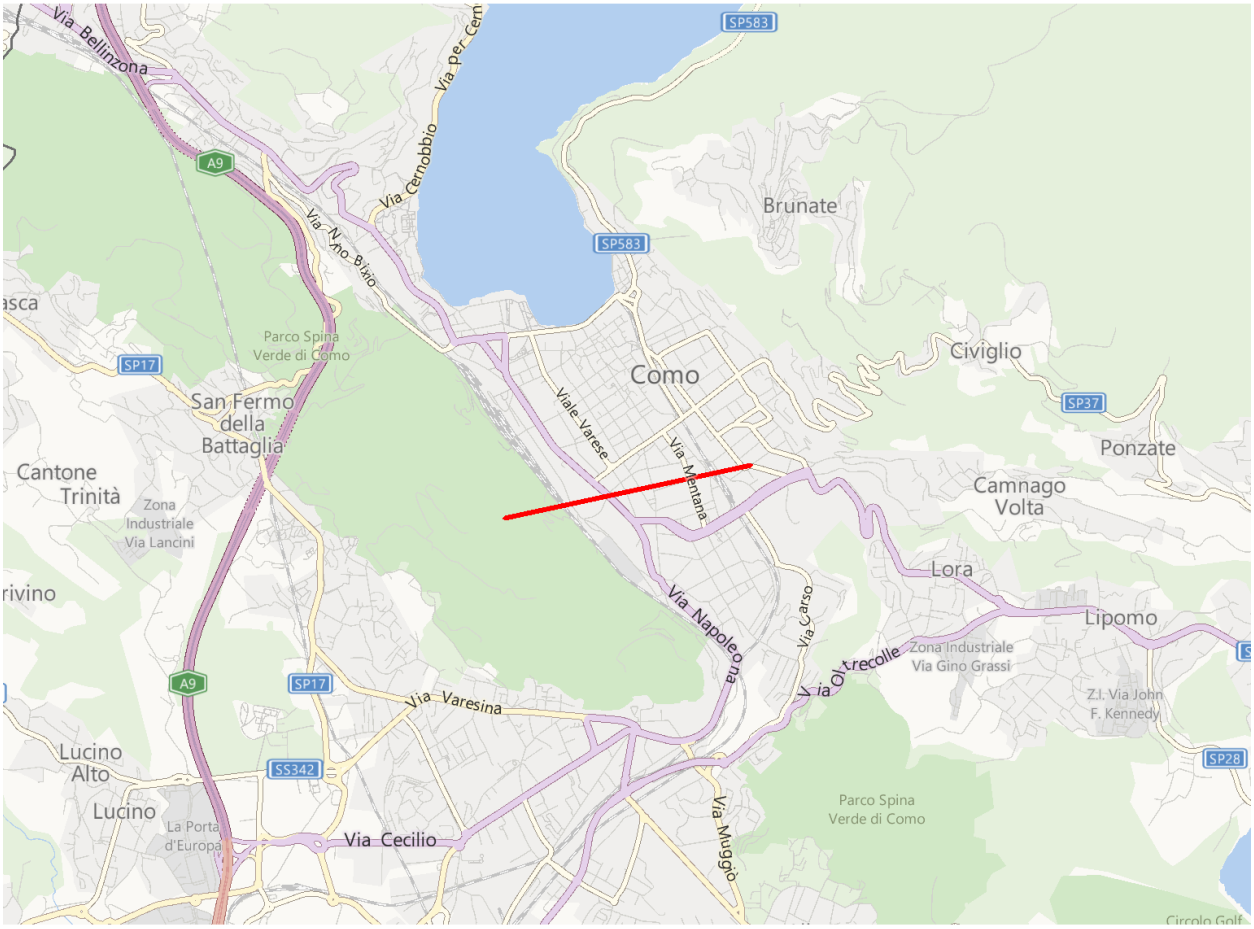

Figure 4: A line represented by a MapPolyline object.

Showing a polygon

Another kind of element you can draw over the map is a polygon, represented by the **`MapPolygon`** class. Its structure is like the **`MapPolyline`** one. The only difference is that, in this case, the **`Path`** collection can contain more than two **`BasicGeoposition`** elements, since we're drawing a more complex shape. Additionally, since it's a shape, we can color the border with the **`StrokeColor`** property, and optionally leverage the **`FillColor`** property if we want to fill the content of the polygon with a color. Look at the following code:

```
private async void OnAddPushpin(object sender, RoutedEventArgs e)
{
    GeolocationAccessStatus status = await Geolocator.RequestAccessAsync();
    if (status == GeolocationAccessStatus.Allowed)
    {
        Geolocator locator = new Geolocator();
        Geoposition geoposition = await locator.GetGeopositionAsync();

        double centerLatitude =
geoposition.Coordinate.Point.Position.Latitude;
        double centerLongitude =
geoposition.Coordinate.Point.Position.Longitude;
        MapPolygon mapPolygon = new MapPolygon();
        mapPolygon.Path = new Geopath(new List<BasicGeoposition>() {
    new BasicGeoposition() {Latitude=centerLatitude+0.0015,
Longitude=centerLongitude-0.010 },
    new BasicGeoposition() {Latitude=centerLatitude-0.0015,
Longitude=centerLongitude-0.010 },
    new BasicGeoposition() {Latitude=centerLatitude-0.0015,
Longitude=centerLongitude+0.010 },
    new BasicGeoposition() {Latitude=centerLatitude+0.0015,
Longitude=centerLongitude+0.010 },

});

        mapPolygon.StrokeColor = Colors.Red;
        mapPolygon.StrokeThickness = 4;
        mapPolygon.FillColor = Colors.Blue;
        MyMap.MapElements.Add(mapPolygon);

        MapScene scene =
MapScene.CreateFromLocationAndRadius(geoposition.Coordinate.Point, 3000);
        await MyMap.TrySetSceneAsync(scene);

    }
}
```

As you can see, the only differences from the line example are:

- We are using the **MapPolygon** class instead of the **MapPolyline** one.
- We are about to draw a square, so the **Path** collection contains four **BasicGeoposition** objects. Also in this case, the positions of the four points of the polygon are calculated based on the location returned by the **Geolocator** class.
- After using the red color to draw the border of the polygon, we are filling it using the blue color.

The rest of the code is the same as what we used to draw the line over the map. After creating the **MapPolygon** object, we add it to the **MapElements** collection exposed by the **MapControl** control.

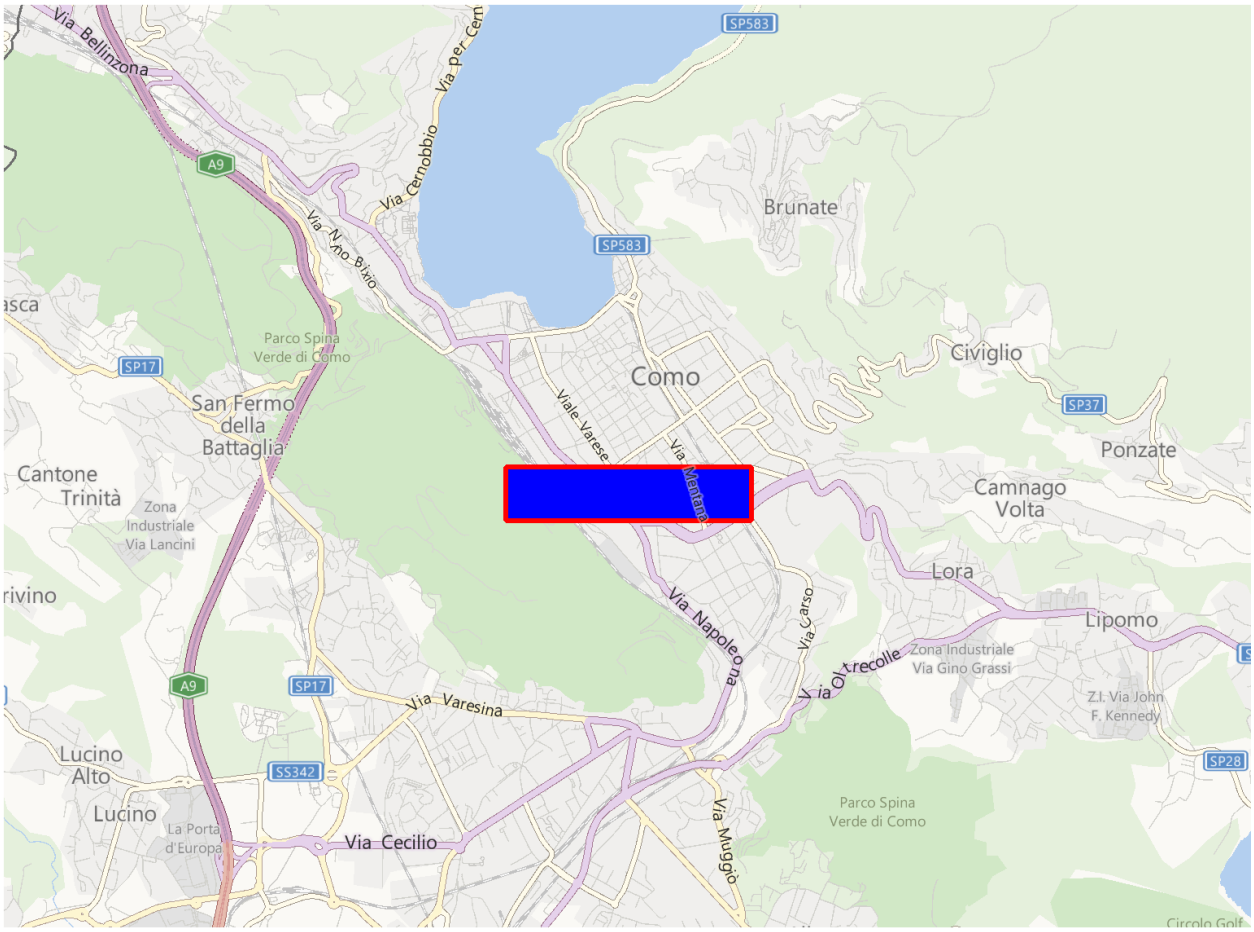

Figure 5: A polygon represented by a MapPolygon object.

Showing a route on the map

The map APIs offer a set of classes and methods that can calculate a route between two or more locations on the map, in the same way we can do with a GPS navigation app. We'll be able to display the route directly on the map and get detailed navigation steps. The operation is performed using the **`MapRouteFinder`** class, which offers a way to calculate a driving route (using the **`GetDrivingRouteAsync()`** method) or a walking route (using the **`GetWalkingRouteAsync()`** method). These two methods can calculate a route from point A to point B. However, you can also calculate a route with multiple waypoints by using the **`GetDrivingRouteFromWaypointsAsync()`** or **`GetWalkingRoutFromWaypointsAsync()`** methods. In both cases, we'll use the **`Geopoint`** class to identify the places' locations.

The following sample shows how to calculate the driving route between two points:

```
private async void OnShowRouteClicked(object sender, RoutedEventArgs e)
{
    MapService.ServiceToken = "K38JSFqmSBMVu0iTXR4aEg";

    BasicGeoposition position1 = new BasicGeoposition();
    position1.Latitude = 45.8080;
    position1.Longitude = 009.0800;
    Geopoint startPoint = new Geopoint(position1);

    BasicGeoposition position2 = new BasicGeoposition();
    position2.Latitude = 45.4608;
    position2.Longitude = 009.1763;
    Geopoint endPoint = new Geopoint(position2);

    MapRouteFinderResult result = await
MapRouteFinder.GetDrivingRouteAsync(startPoint, endPoint,
MapRouteOptimization.Time, MapRouteRestrictions.Highways);
    if (result.Status == MapRouteFinderStatus.Success)
    {
        Duration.Text = result.Route.EstimatedDuration.ToString();
        Length.Text = result.Route.LengthInMeters.ToString();
        List<string> directions = new List<string>();
        foreach (MapRouteManeuver direction in
result.Route.Legs[0].Maneuvers)
        {
            directions.Add(direction.InstructionText);
        }
    }
}
```

The usage of the **MapRouteFinder** class is one of the scenarios that requires a valid map token. Therefore, the first thing to do is properly set the **ServiceToken** property of the **MapService** class.

As you can see, the **GetDrivingRouteAsync()** method doesn't just support setting starting and ending points, but also some parameters to customize the route calculation. In the previous sample, we chose the kind of optimization we want to apply by using the **MapRouteOptimization** enumerator (in this case, it will calculate the shortest route based on time) and specifying that we want to avoid highways (by using the **MapRouteRestrictions** enumerator). The method will return a **MapRouteFinderResult** object, which offers many properties with the details of the calculated route. In the previous sample, we display to the user the duration (stored in the **EstimatedDuration** property) and the length of the route (stored in the **LenghInMeters** property). One important property is called **Legs**, a collection of all the routes that have been calculated. Every route offers a collection called **Manuevers**, a list of all the instructions to go from point A to point B. In the sample, we store in a collection all the values of the **InstructionText** property, which contain textual indications (like, "At the end of the street go left").

In addition, we can use the **MapRouteFinder** class not just to retrieve the information about the route, but to display it on the map using the **MapRouteView** class and the **Routes** collection offered by the **MapControl** class. Let's look at the following sample:

```
private async void OnShowRouteClicked(object sender, RoutedEventArgs e)
{
    MapService.ServiceToken = "K38JSFqmSBMVu0iTXR4aEg";

    BasicGeoposition position1 = new BasicGeoposition();
    position1.Latitude = 45.8080;
    position1.Longitude = 9.0800;
    Geopoint startPoint = new Geopoint(position1);

    BasicGeoposition position2 = new BasicGeoposition();
    position2.Latitude = 45.4608;
    position2.Longitude = 9.1763;
    Geopoint endPoint = new Geopoint(position2);

    MapRouteFinderResult result = await
MapRouteFinder.GetDrivingRouteAsync(startPoint, endPoint,
MapRouteOptimization.Time, MapRouteRestrictions.Highways);
    if (result.Status == MapRouteFinderStatus.Success)
    {
        MapRouteView routeView = new MapRouteView(result.Route);
        routeView.RouteColor = Colors.Red;
        routeView.OutlineColor = Colors.Black;
        MyMap.Routes.Add(routeView);
    }
}
```

The **MapRouteView** class takes care of encapsulating the **Route** property of the **MapRouteFinderResult** class. In addition, we can use it to customize the visual layout of the route by setting its color (**RouteColor**) and border (**OutlineColor**). In the end, you just need to add the just-created **MapRouteView** object to the **Routes** collection of the **MapControl** object you've placed in the page.

Working with coordinates and addresses

Up to now, we've always used the **MapControl** object with positions expressed by geographic coordinates, like latitude and longitude. However, in many scenarios, you need to work with information that is easier to understand for the user, like a civic address. The Universal Windows Platform offers a set of APIs to perform geocoding (which means converting an address into coordinates) and reverse geocoding (which means converting a set of coordinates into an address).

Geocoding is performed using the **MapLocationFinder** class. This class offers a method called **FindLocationsAsync()** that accepts as parameter the name of the location we want to find, like in the following sample:

```
private async void OnGeocodeClicked(object sender, RoutedEventArgs e)
{
    string address = "Piazza Duomo, Milan";
    MapLocationFinderResult result = await
MapLocationFinder.FindLocationsAsync(address,
    null);
    if (result.Status == MapLocationFinderStatus.Success)
    {
        Geopoint position = result.Locations.FirstOrDefault().Point;
        MapScene scene = MapScene.CreateFromLocationAndRadius(position,
3000);
        await MyMap.TrySetSceneAsync(scene);
    }
}
```

Optionally, you can also pass a second parameter to the **FindLocationsAsync()** method in the form of a **Geopoint** object with the coordinates of the approximate area where the location is placed. If you don't have this information, you can simply pass **null** as value, like we did in the previous sample. The method returns a **MapLocationFinderResult** object, which contains two pieces of important information. The first one, **Status**, informs the developer of whether the operation has been completed successfully or not. In the previous sample, we move on with the rest of the code only if we get **Success** as return value. The second piece, **Locations**, is a list of all the places that have been found for the specified search keyword. For each location, we have access to a property called **Point**, with the location's coordinate. In the previous sample, we take the coordinates of the first location and we display it on the map, by calling the **TrySetSceneAsync()** method.

The opposite operation (reverse geocoding) is also performed using the **MapLocationFinder** class. The difference is that, in this case, we're going to use the **FindLocationsAtAsync()** method, which requires as parameter a **Geopoint** object with the coordinates of the place. Also in this scenario, we'll get in return a **MapLocationFinderResult** object, which offers a **Locations** property with all the places that match the given coordinates. The difference is that, this time, instead of using the **Point** property (since we already know the coordinates), we access the **Address** one, like in the following sample:

```
private async void OnReverseGeocodeClicked(object sender, RoutedEventArgs
e)
{
    Geolocator locator = new Geolocator();
    Geoposition geoposition = await locator.GetGeopositionAsync();

    BasicGeoposition position = new BasicGeoposition();
    position.Latitude = geoposition.Coordinate.Latitude;
    position.Longitude = geoposition.Coordinate.Longitude;
    Geopoint point = new Geopoint(position);

    MapLocationFinderResult result = await
MapLocationFinder.FindLocationsAtAsync(point);
    if (result.Status == MapLocationFinderStatus.Success)
    {
        MapAddress address = result.Locations.FirstOrDefault().Address;
        string fullAddress = string.Format("{0}, {1}", address.Street,
address.Town);
        MessageDialog dialog = new MessageDialog(fullAddress);
        await dialog.ShowAsync();
    }
}
```

In this sample, we retrieve the user's position and we display with a pop-up message the corresponding address, retrieved by combining the values of the **`Street`** and **`Town`** properties of the **`Address`** class.

The motion sensors

Most portable devices on the market include a set of motion sensors, which can be used to detect if and how the device is moving in space. Many games use this approach to provide an alternative way to control the game, instead of relying on virtual joy pads that are displayed on the screen.

Windows devices offer many sensors. Each of them is represented by one of the classes that are included in the **`Windows.Devices.Sensors`** namespace. Here are the main ones:

- **Accelerometer,** identified by the **`Accelerometer`** class. It measures the phone's position on X, Y, and Z axes.
- **Inclinometer,** identified by the **`Inclinometer`** class. It measures roll, pitch, and yaw.
- **Compass,** identified by the **`Compass`** class. It measures the device's position in relation to magnetic north.
- **Magnetometer,** identified by the **`Magnetometer`** class. It measures the magnetic field's intensity.
- **Gyrometer**, identified by the **`Gyrometer`** class. It measures the angular speed of the device.

However, in most cases, what's important for the developer is to discover the orientation of the device with the best accuracy. For this purpose, the Universal Windows Platform offers a special class called **OrientationSensor**, which can combine all the information retrieved by the available sensors and apply a set of mathematical formulas to provide a set of optimized values. To work properly, this class requires the device to include at least an accelerometer and a compass. However, the best results are achieved when the device also includes a gyroscope.

Regardless of the sensor we're going to use, they all work in the same way. Every class offers a method called **GetDefault()**, which retrieves a reference to the sensors. It's important to always check that the returned value is not **null** before performing additional operations. The device the application is running on, in fact, could not offer that sensor, and therefore any other interaction would return an exception. After you have a valid reference to the sensor, you can use two different approaches, similar to what we've seen with the GPS tracking.

The first approach is to ask for a single reading by using the **GetCurrentReading()** method. The data type you will get in return can change from sensor to sensor. Typically, the name of the class matches the name of the sensors plus the **Reading** suffix (for example, the **Accelerometer** class returns an **AccelerometerReading** object). The following sample shows how to use the **Accelerometer** class to get a single reading of the current position:

```
private void OnStartAccelerometerClicked(object sender, RoutedEventArgs e)
{
    Accelerometer accelerometer = Accelerometer.GetDefault();
    if (accelerometer != null)
    {
        AccelerometerReading reading = accelerometer.GetCurrentReading();
        X.Text = reading.AccelerationX.ToString();
        Y.Text = reading.AccelerationY.ToString();
        Z.Text = reading.AccelerationZ.ToString();
    }
}
```

The position of the device on the three axes (stored in the **AccelerationX**, **AccelerationY**, and **AccelerationZ** properties of the reading object) are displayed on the page using three **TextBlock** controls.

The other approach is to continuously monitor the sensors so that your app will be notified every time the device is moved into a new position. For this scenario, we can subscribe to an event exposed by all the sensor's classes called **ReadingChanged**. The following sample shows how to subscribe to this event for the **Accelerometer** class, so that we can continuously update the page with current coordinates.

```
private void OnStartAccelerometerClicked(object sender, RoutedEventArgs e)
{
    Accelerometer accelerometer = Accelerometer.GetDefault();
    if (accelerometer != null)
    {
        accelerometer.ReadingChanged += accelerometer_ReadingChanged;
    }
}

void accelerometer_ReadingChanged(Accelerometer sender,
AccelerometerReadingChangedEventArgs args)
{
    Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
    {
        X.Text = args.Reading.AccelerationX.ToString();
        Y.Text = args.Reading.AccelerationY.ToString();
        Z.Text = args.Reading.AccelerationZ.ToString();
    });
}
```

Notice that the **ReadingChanged** event is executed in a background thread, so we need to use the **Dispatcher** to interact with the XAML controls placed in the page.

The SimpleOrientation sensor

The Universal Windows Platform offers a sensor called **SimpleOrientation**, which is always used to detect the position of a device in space, but in a simpler way. Unlike other sensors, like the **Accelerometer** or the **OrientationSensor**, it doesn't return the exact coordinates of the device (a scenario that could be too detailed for most of the application) but just the current orientation (landscape, portrait, etc.).

Its usage is very basic: after getting a reference to it using the **GetDefault()** method, you can leverage the **GetCurrentOrientation()** method to get a single reading or subscribe to the **OrientationChanged** event if you want to be notified every time the device orientation changes.

In both cases, you get in return a **SimpleOrientation** enumerator, which contains the different orientations that a device can assume, like **Facedown**, **Faceup**, or **Rotated90DegreesCounterclockwise**. You can find a list of all the supported values in the documentation.

Here is a sample code:

```
private void OnGetOrientation(object sender, RoutedEventArgs e)
{
    SimpleOrientationSensor simpleOrientation =
SimpleOrientationSensor.GetDefault();
    if (simpleOrientation != null)
    {
        SimpleOrientation orientation =
simpleOrientation.GetCurrentOrientation();
    }
}
```

The Pedometer sensor

The Universal Windows Platform contains an additional class that leverages a sensor called pedometer, which is typically available on mobile devices like phones or phablets. This sensor can detect the number of steps taken by the user and how they have been taken.

This sensor is represented by the **Pedometer** class, which works in a slightly different way than the other ones. The other sensors' classes we've seen so far were already available in Windows 8.1, while the **Pedometer** one has been added in the Universal Windows Platform.

The overall approach is very similar. You get access to the sensor, if it's available, and then you have the chance to get a single reading or to subscribe to the **ReadingChanged** event, which is triggered every time the number of steps taken by the user changes. The peculiarity of the **Pedometer** class is that it can be leveraged also by a background task, so that you can also get this notification when the foreground app isn't running.

Here is a sample usage of this class:

```
private async void OnGetPedometerReading(object sender, RoutedEventArgs e)
{
    Pedometer pedometer = await Pedometer.GetDefaultAsync();
    if (pedometer != null)
    {
        var results = pedometer.GetCurrentReadings();
        int runningSteps =
results[PedometerStepKind.Running].CumulativeSteps;
        int walkingSteps =
results[PedometerStepKind.Walking].CumulativeSteps;
    }
}
```

As you can see, in this case, the method to get access to the default pedometer (if available) is asynchronous. Consequently, the method of the **Pedometer** class to call is called **GetDefaultAsync()**.

Regardless of whether you want to get a single reading (by calling the **GetCurrentReadings()** method) or be notified of step changes (by subscribing to the **ReadingChanged** method), you'll get in return a special collection of key–value pair elements. The collection contains multiple values, based on how the steps have been taken. The keys are represented by the **PedometerStepKind** enumerator. By using it, we can differentiate the number of steps taken by walking (**PedometerStepKind.Walking**) from those taken by running (**PedometerStepKind.Running**). For each kind, we have access to a set of properties about the recording. The most notable one is **CumulativeSteps**, which contains the total number of steps taken.