Lab 6: WalkAbout

Intro

For this lab you will be developing a new GPS recording application called WalkAbout. The purpose of the application is to allow users to record their GPS location information as they travel. While the application records the user's GPS data, it displays it back to the user in the form of a path drawn on a Google Map. While recording data, the user can launch a Camera activity that will capture and store pictures on a storage device. When finished recording, the application gives the user the option of storing the current GPS data as a private application file to be loaded and displayed at a later time. The application will populate the map with shapes and markers to indicate actions taken, and be able to save and load this data to repopulate the map at a later time.

Objectives

At the end of this lab you will be expected to know:

- *How to display an interactive Google Map inside an application.*
    - *How to register for an Android Google Maps API key online.*
    - *How to enable Android Google Maps in an application project.*
- *How to register for and receive GPS location information.*
- *How to launch and receive outside Activities.*
- *How to change the way the Google Maps viewing camera looks at the map.*
- *How to draw basic shapes and Markers on a Google Map.*

- *How to use the Camera.*
    - *How to call the Camera Activity using an Intent.*
    - *How to store pictures taken to internal storage (SD card, internal storage, etc.).*

- *How to modify private application files for saving and loading data.*

Activities

For this lab you will be working with a brand new application, completely independent from the previous labs. Over the course of the lab, you will be iteratively refining and adding functionality to the WalkAbout app. With each iteration you will be improving upon the previous iteration's functionality.

You'll start by setting up and familiarizing yourself with the Eclipse project. You will then register for a Google Maps API key and begin incrementally developing the main map-viewing Activity. These first few exercises will have you display a map and the user's current position. Next, you will add functionality to record the user's GPS location by registering for and receiving data from what is known as the GPS Location Provider. After that, you will implement a Camera activity for taking pictures and saving them to the SD card. In the final section, you will save a user's GPS path to a file that is private to the application, and allow the application to restore itself from the file as well.

IMPORTANT:
You will be given a Skeleton Project to work with. This project contains all of the Java and resource files you will need to complete the lab. Some method stubs, member variables, and resource values and ids have been added as well. It is important that you not change the names of these methods, variables, and resource values and ids. These are given to you because there are unit tests included in this project as well that depend on these items being declared exactly as they are. These unit tests will be used to evaluate the correctness of your lab. You have complete access to these test cases during development, which gives you the ability to run these tests yourself. In fact, you are encouraged to run these tests every so often to ensure that your application is functioning properly.

IMPORTANT:
It is highly recommended that you run this application on a physical Android device that has access to GPS and a camera. There are several methods to simulate GPS on an emulator (some of which are covered in part 3.2.2 below) but this lab assumes you will walk around with a physical device plugged into GPS instead of simulating GPS.

IMPORTANT:

Go to the Google Play Store and install Google Maps if your device doesn't have it installed yet. This lab will assume that you have Google Maps installed; it will not cover how to check to see if Google Maps is installed on the test device. For more information on how to make the application check for a Google Maps installation, see here.

Contents

A table of contents will be added soon.

1. Setting Up

To begin, download and extract the skeleton project for the WalkAbout application. Click here to download the skeleton project.

- Extract the project, making sure to preserve the directory structure.
  - *Take note of the path to the root folder of the skeleton project.*

Next you will need to set up an Android project for this app. Since the skeleton project was created in Eclipse, the easiest thing is to import this project into Eclipse.

- Select **File -> Import**.
- In the Import Wizard, expand **General** and select **Existing Projects into Workspace**. Click **Next**.
- In the **Import Project** wizard, click **select root directory** and click **Browse**. Select the root directory of the skeleton project that you extracted. Click **Open** and then **Finish**.
- Click on the project name in the Package Explorer. Select **File** -> **Rename** and change the name of your project to **lab6<userid>** where <userid> is your user id (e.g. jsmith).
- Check to make sure the package name is **edu.calpoly.android.walkabout** (**src** folder).
- Make sure that your project is targeting the latest version of Android but supporting API 10 and higher.
  - In the Manifest file, find the **uses-sdk** XML component and

change **android:targetSdkVersion** to the latest API version (17 at the time of writing this lab).

        - If **android:minSdkVersion** is not set to 10, make it so.

Note: This project contains errors from the get-go. You will fix some of them right now. Feel free to clean and rebuild the project throughout these steps if you think your project should be compiling properly.

If you haven't set up your project to use ActionBarSherlock (ABS), do so now. If you already have the library project for ABS, you should be able to skip this step.

- Use the setup step from Lab 3 as a guide for downloading and creating the ABS library project and adding ABS as a library project to your imported stub project under **Properties** -> **Android** -> **Library**.
    - You will need to extend **SherlockFragmentActivity** in this lab instead of just SherlockActivity so you can use Fragments.

Next you will need to set the correct build target. In the Android SDK Manager (available from Windows menu in Eclipse), ensure that you have Google APIs for all of the versions of Android that run on the devices you wish to test this application on (minimum API of 10). You will at least want the latest Google APIs package (17 at the time of writing this lab, this is the default build target in the stub project).

- If you don't have all Google APIs packages you need, then install them through the SDK Manager. (see here for further information).

Once you have the Google APIs add-on installed, you need to use it as the target for your project. The project is currently set to API 17, but this is incorrect. You want to target the latest Google APIs version, or at least the version your test device is compatible with.

- Right-click on the lab6<userid> project, then select **Properties**. Select **Android** and select **Google APIs** for the API level of the version of Android your test device has (the most recent Google APIs version, 17 at this point in time, will work).
- *Newer versions of the Google APIs package are apparently backwards-compatible. This is because newer versions are capable of behaving the same way as older versions via making the same method calls, etc. For example, if you want to run this lab on a device*

that runs Gingerbread (API 10), setting the target to API 17 will still work. Disappointingly, Google's reference page for installing Google APIs does not address this subject.

After all this, there will still be errors. You will fix them in section 2 and hopefully see them fade away slowly.

Note: You cannot attempt to run this application successfully until step 2.1.4.

1.2 Familiarize Yourself with the Project

The project is extremely small. It contains a single Java class file and a single XML layout file which you will have to implement. **WalkAbout.java** will contain the definition for the main WalkAbout Activity class. This is the class that will display the map and the user's recorded path. The WalkAbout class makes use of a very simple XML layout file called **map_layout.xml** which you will have to fill in.

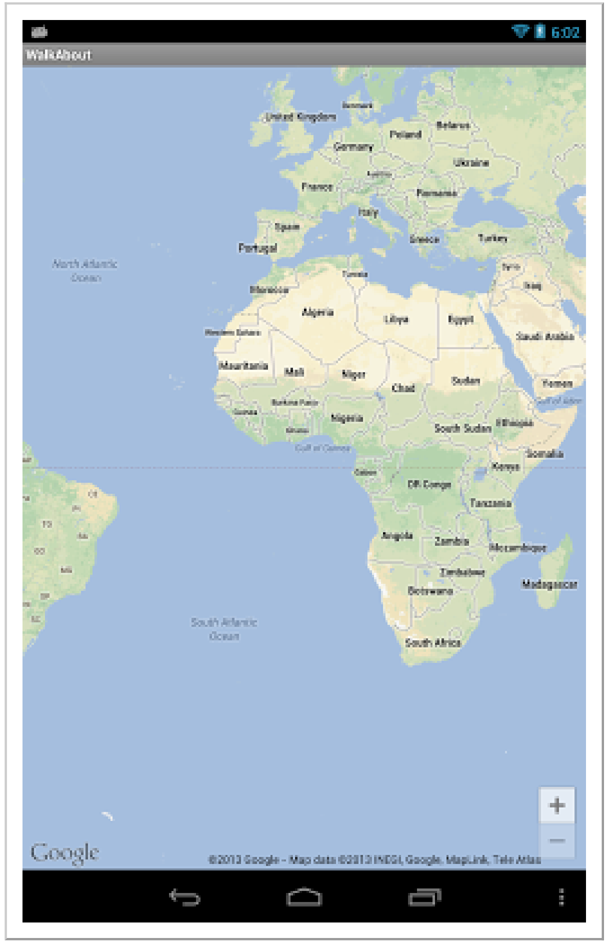
2. Using Google Maps

In this section of the lab you will be working extensively with the Google Maps package. You will follow numerous setup steps to be able to incorporate a Google Maps map in your application, and then provide basic map utilities such as the My Location button and Zoom controls. We will not be calling the Google Maps app directly; rather, we will implement our own MapFragment in our application and use Google's services to maintain it.

Before we dive into the code, it is important to distinguish between the **Google Maps API** and the **Google Maps *Android* API**. The latter is a child of the former, and is maintained specifically for using Google Maps services in Android devices. The term "Google Maps API" represents a collection of APIs related to using Google Maps services. The primary Google Maps API is the JavaScript web-based version that allows for embedding and manipulating Google Maps in web pages. This is the version that many sites, including maps.google.com, use to display Google Maps in a web browser (for example, if you go to maps.google.com and view the page source, you will see a ton of JavaScript). The JavaScript Google Maps API (for web) is currently up to version 3 (v3), and the Google Maps Android API is currently up to version 2 (v2). Keep this distinction in mind when searching for Google Maps API information.

2.1 Displaying a Map

You will begin by implementing the functionality necessary to display a full-screen map in the WalkAbout Activity class. The layout for the WalkAbout Activity should be specified in the res/layout/map_layout.xml file, which you will have to fill in. This will require you to use the SupportMapFragment class. In order to do this, you will have to register the debug keystore (that Eclipse uses to run your applications) with Google in order to receive a Maps API Key. When finished, your application should appear as depicted in the figure below:



6:02
WalkAbout
Google
©2013 Google - Map data ©2013 INEGI, Google, MapLink, Tele Atlas

Map of the world, centered. Device: Nexus 7

For more information and examples on working with the Google Maps Package, see the documentation site.

2.1.1 Get a Google Maps Android API Key

In order to use the Google Maps Android API and classes you will have to sign into your Google account, then register the keystore you use with the appropriate service in the Google APIs Console. Once your keystore is registered, you will be provided with a Google Maps Android API key that can be used with any application signed by your keystore. We will be using Google Maps Android v2, the current Android Maps version at the time of rewriting this lab.

Important: *Google Maps v2 does not reuse the setup process from v1*, the latter of which has been deprecated at the time of writing this lab.

Every distribution of the Android Development Toolkit comes with a debug keystore that is used to sign your application when it is launched and run from Eclipse. This is how you are able to run your applications on a device or emulator without signing it yourself. **For the purposes of this lab, you need only register with a debug keystore. If you end up releasing an application to the Google Play Store, however, you will need to register an actual keystore to officially sign your application correctly before publishing it in the store.**

Note: *If you get stuck at any point in this or the next substep, try to pinpoint problems using* the startup guide or the Map Fragment Quick Start document.

Open these instructions and read the first few paragraphs of information. Now follow the steps listed below while going through the instructions at the linked page:

- Retrieve the SHA-1 fingerprint of your debug keystore file:
    - Follow the instructions beneath the "Displaying certificate information" section on the instruction page. Follow the instructions to show the debug certificate, since this lab is not going to be released to the Google Play Store.
        - *If you find you cannot run the **keytool** command, edit your system environment variables. **keytool** comes with the Java JDK; make sure to add the location of the folder that **keytool** is in to the PATH variable, then try again.*
- Using the Google APIs Console in another browser tab or window, create a project for your Android application and register for the Maps API.
    - Follow the instructions beneath the "Creating an API Project" section on the instruction page. Read the Terms of Service for using the Google Maps Android API v2 service ***carefully***, then accept them to continue.

- Still in the Google APIs Console, retrieve an API Key as a resource string in your Application:
    - Follow the instructions beneath the "Obtaining an API Key" section on the instruction page. Make sure to use **edu.calpoly.android.walkabout** as the package name when applying for an API key.
    - Once the application is complete, you'll see a new "Key for Android apps (with certificates)" API key on the APIs console webpage beneath the **Simple API Access** section. This contains information about who owns the key, when it was activated and which Android applications it may be used for, as well as the key itself.
- Add the API Key to your Application:
    - Follow the instructions beneath the "Adding the API Key to your application" section on the instruction page.
        - *We no longer need to store the API Key as a (String) resource, since we now no longer require the API key when adding Maps in the code base.*
- Add the Google Play services version to your Application:
    - Follow the instructions beneath the "Add the Google Play services version to your app's manifest"

      ```
      <meta-data
          android:name="com.google.android.gms.version"
          android:value="@integer/google_play_services_version" />
      ```
    - If this gives an error, check to be sure that you imported the correct version of the Google Play Services project in your IDE.
- In your project's Manifest file, add the following line just above the closing </application> tag:

```
<uses-library android:required="true" android:name="com.google.android.maps" />
```

This (along with the libraries in the next section) will let our application recognize special Google Maps development classes such as LatLng, as will be explained later in this lab.

- While we're at it, we will add permissions and a prompt for OpenGL ES 2.0 to the Manifest file as well. The latter is the method of rendering Google Maps objects in applications.
    - Add the recommended permissions mentioned under the "Specifying permissions" section on the instruction page.
    - Follow the steps mentioned under the "Requiring OpenGL ES version 2" section on the instruction page. Add the required snippet at the same level as the permissions.
        - *This is **vital** for older devices that do not have OpenGL ES 2.0 support, as the Google Play Store will not show the app on those devices. It would be bad to give an older device the chance to run an application that it is unable to run!*

2.1.2 Add Google Play and Android Support

Having the key, permissions etc. set up is only half the battle. We now need to set up Eclipse and our project so that it uses the Google Play Services development kit and the Android Support Library (explained below).

First we need the Google Play Services and Android Support Library packages.

- Open the Android SDK Manager and make sure you install or have installed and updated the **Google Play Services** and **Android Support Library** packages.

Our application requires a dependency on the Google Play Services project, so add it as a library project as follows:

- Click **File** -> **Import...** to open the window for import selection in Eclipse, then choose **Android** -> **Existing Android Code into Workspace**.

    - Navigate to **<android-sdk-folder>**/extras/google/google_play_services/libproject/google-play-services_lib and then finalize the import.

- Right-click on your project in the Package Explorer and choose **Properties**, then navigate to **Android** -> Library (bottom), then add the **google-play-services_lib** project and click Okay.

The Google Maps Android API v1 used MapActivity to display maps. However, the Google Maps Android API v2 uses a different (and quicker and more flexible) way of rendering maps: they are now contained inside MapFragments. However, MapFragment only works for API 12 or higher. Using the Android Support library, we can utilize the MapFragment functionality through a class called SupportMapFragment that works for API versions below 12.

Note: There is already a JAR file for this in the ***libs*** *folder. However, it is not guaranteed to be up-to-date.*

- In Eclipse, right-click on your Android project in the Package Explorer and choose **Android Tools** -> **Add Support Library...**
- Choose the most up-to-date Android Support Library, click **Accept** and then click **Install**.
- Make sure that the **android-support-v4.jar** file is present under your project in the Package Explorer, under *the libs directory*.
- You should get the newest version of this JAR file and have it in your libs folder, but keep in mind that you will have to give your ActionBarSherlock library project the exact same file in the exact same location or else there will be a mismatch.

Now we have finally set up our environment to work with Google's map services and obtained our API key. Let's delve into development! Keep the instructions page open, as it will still be our reference.

2.1.3 Fill in the MapView XML Layout File

The main WalkAbout MapActivity class uses a very simple layout consisting of a single GoogleMap variable. This variable will utilize the (Support)MapFragment element that you set here in order to render a (you guessed it) Google Map in your application.

Implement this layout by filling in the **map_layout.xml** file:

- Follow the steps under the "Add a Map" section on the instructions page and add a new fragment to the empty XML file.
    - Instead of using the class **com.google.android.gms.maps.MapFragment**, use **com.google.android.gms.maps.SupportMapFragment**.

2.1.4 Display the Map

The WalkAbout MapActivity will display the map. All initialization relating to the layout should be done in **WalkAbout.initLayout()**, which gets called by **onCreate()**:

- Inflate your **map_layout.xml** file.
- Initialize the GoogleMap **m_vwMap** member variable by retrieving a reference to the SupportMapFragment element in the XML layout file.
    - This is done by making a call to the SupportFragmentManager via **getSupportFragmentManager()**, then calling **findFragmentById()** with the appropriate resource ID. You must then cast the result to a SupportMapFragment, and then call **getMap()** on that object.
    - Make sure to check the GoogleMap variable for null before modifying it in any way.
- By default, the Zoom Controls (two buttons, + and -) are displayed in the lower-right corner of the map. You can enable or disable them by calling **getUiSettings()** on your GoogleMap variable then setting the Zoom Controls via **setZoomControlsEnabled()**.

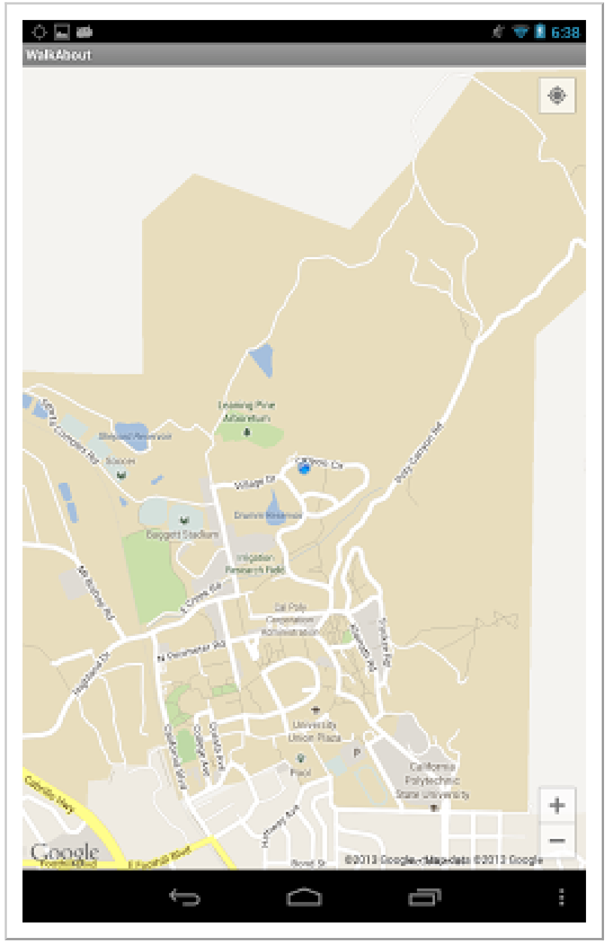
You should be now be able to run your application and see a Google Map. The Zoom Controls should be displayed in the bottom-right corner of the screen. Scrolling the map via touch should move the map around and load new map tiles.

The code to display a map was simple compared to the setup, wasn't it? Now let's move on to the bulk of the lab.

2.2 Adding My Location and Compass

In v1 of the Google Maps Android API, we had to use Overlays to draw objects such as an indicator for the user's location and a compass over a MapView, which represented the map. However, in v2 we are treated to simpler ways of adding these objects over the map via accessing the UISettings of a GoogleMap object.

You will add indicators for the user's location and the compass on your map. When finished, your application should appear as depicted in the figure below:



Map with My Location layer enabled.

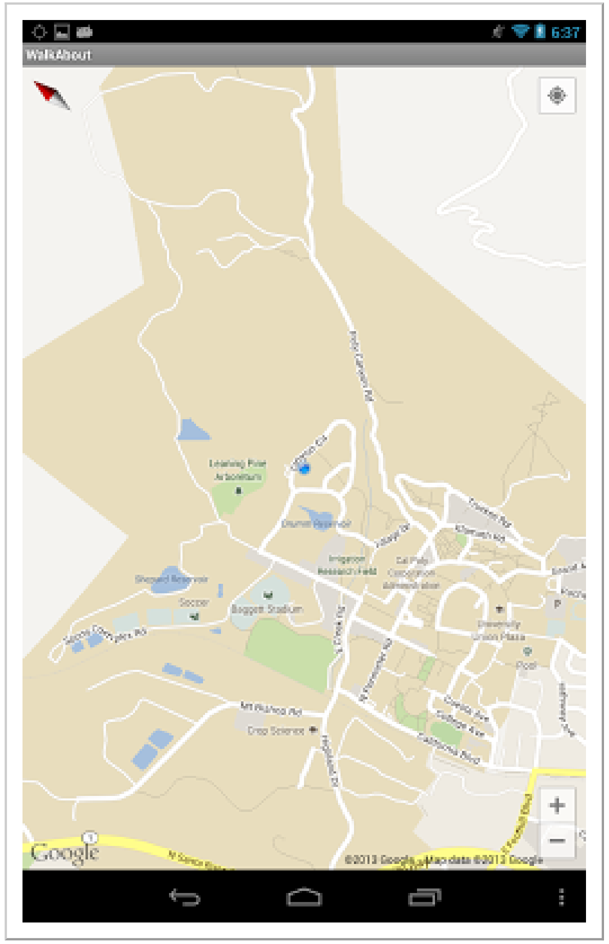
Map with My Location layer enabled.

2.2.1 Display Your Location and Compass

Enable the map to display your current location and the compass. Note that the compass only appears when you turn the map using gestures on the screen, as the orientation is defaulted to North pointing directly upwards, East pointing directly right, etc. The compass points North with red and South with white.

- Inside of the null GoogleMap check, change MyLocation to be enabled.
    - *This will enable the MyLocation layer, which will show the My Location button on top of the map by default.*
    - *Note: You can change whether or not this button appears by calling* ***setMyLocationEnabled(...)*** *on the Google Map object.*
- Inside of the null GoogleMap check, retrieve the UISettings for the GoogleMap object and change Compass to be enabled.
    - *Hint: These can be done similarly to how the ZoomControls were changed.*

You should be able to run your application and see a Google Map that will point and zoom to your current location if you press the My Location button in the corner, as well as a compass displaying your current orientation if the map is turned via gesturing in the other corner.



The compass only shows when the map's orientation is changed from the default (use a two-finger twist gesture).

There are more ways to allow the user to interact with your map. For more information, visit the Interacting with the map documentation.

2.3 Initialize the WalkAbout Options Menu

The WalkAbout Activity has an Options Menu that will display five different menu items with no submenu items. Create the Options Menu as follows. Do not worry about adding actual menu functionality yet, since you will do that later.

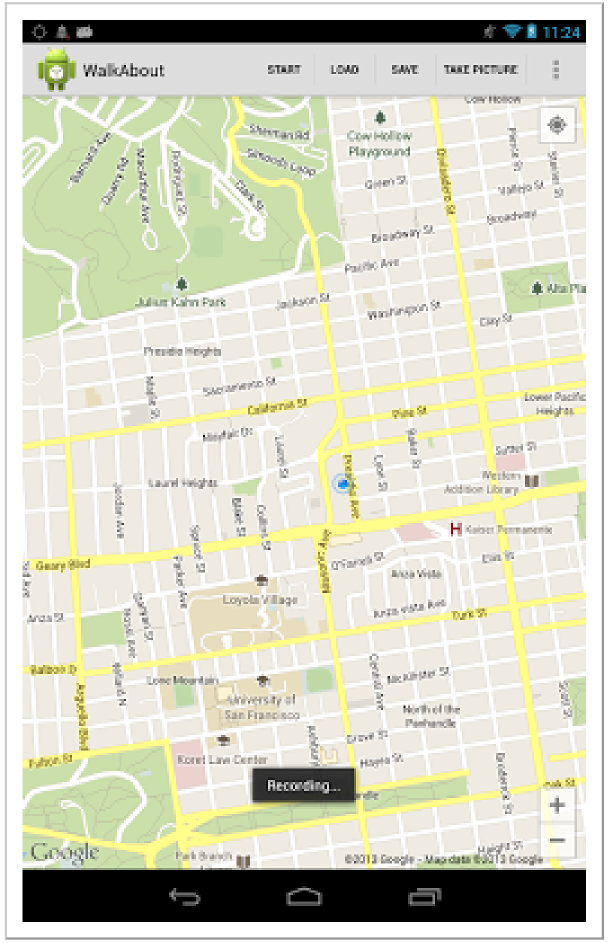
- "Start/Stop": This MenuItem acts as a toggle for either starting or stopping GPS Location recording. This recording is a record of the path that you have traveled. While recording, the MenuItem should display "Stop." While stopped, the MenuItem should display "Start."

  - Initialize the text for the MenuItem with **R.string.menuTitle_startRecording** resource string, since you won't have a recording in progress when first starting the application.
    - Later on, you will dynamically update the text to reflect the current recording state.
  - Make this menu item ***always*** show in the Action Bar.
  - Give this item an ID of **menu_recording**.
- "Save": This MenuItem will save the current recorded path.
  - Initialize the text for the MenuItem with **R.string.menuTitle_save**.
  - Give this item an ID of **menu_save**.
- "Load": This MenuItem loads the last saved path.
  - Initialize the text for the MenuItem with **R.string.menuTitle_load**.
  - Give this item an ID of **menu_load**.
- "Take Picture": This MenuItem launches a Camera Activity that allows you to take pictures.
  - Initialize the text for the MenuItem with **R.string.menuTitle_takePicture**.
  - Give this item an ID of **menu_takePicture**.
- "Enable GPS": This MenuItem launches the Device Settings Activity that allows you to enable the GPS Provider.
  - Initialize the text for the MenuItem with **R.string.menuTitle_enableGPS**.
  - Give this item an ID of **menu_enableGPS**.

Add a bare-bones **onOptionsItemSelected(...)** method:

- Override **SherlockFragmentActivity.onOptionsItemSelected(...)**.
- You can add Toasts and run your application to make sure that each menu item responds when pressed, even though they will do nothing yet. When in doubt, make Toast™.

Now *there's* a menu! (Click image for full size)



Testing the menu items' responses. The Start menu item was just pressed.

3. Using Location Services

The Android System provides services for determining the current location of the device. The framework for working with these location based services lives under the android.location package. A number of useful classes live inside this package:

- **LocationManager**: Provides an interface for the location-based services. You will be interacting with this class most of the time when trying to obtain location information.
- **LocationProvider**: LocationProviders are classes that provide updates on the current location of the device. There exists a LocationProvider for each different technology that determines location. There exists a GPS LocationProvider and a Network LocationProvider. Each LocationProvider specifies criteria that must be satisfied in order for it to be used. For example, the GPS LocationProvider requires that the device have GPS hardware and that it be enabled.
- **Criteria**: This class allows you to programmatically outline criteria you would like from a LocationProvider. Such criteria may include accuracy, power consumption, altitude reporting, speed reporting, monetary cost, etc. You can then setup and give an instance of this class to the LocationManager and allow it to choose the LocationProvider that best matches your criteria.
- **LocationListener**: This is an interface that defines call backs for different events that are generated by LocationProviders. These can be registered with a particular LocationProvider to receive updates when the location changes or the state of the LocationProvider changes.
- **Location**: A data object containing location based information. Generally contains latitude and longitude, as well as a date and timestamp containing the time at which the location was determined. Might also contain elevation, speed, and heading information.
- **Geocoder**: A utility class for Geo-Coding (what a surprise, right?) which can transform addresses to GPS coordinates and back.
    - *Other geo-based utility classes provide functionality for for addresses and some classes for monitoring the status of the GPS satellites the device is locked onto, among other things.*

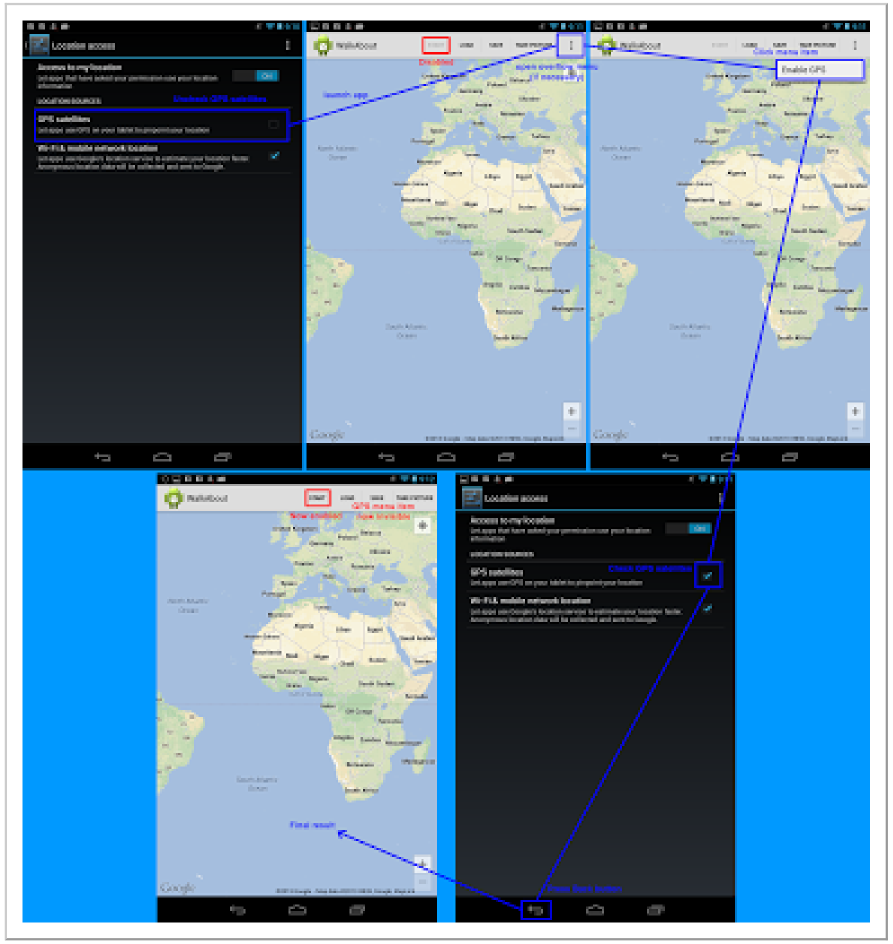
In general, you usually query the LocationManager for the information you are seeking (this should sound familiar from lab 4, where you queried the LoaderManager several times). Just to name a few of the LocationManager's powers, you can use it to check the current status of a LocationProvider, you can check the last known location reported by a LocationProvider, and you can register to receive updates from a LocationProvider. To receive updates on location information directly from a LocationProvider, you need to implement the LocationListener interface and register yourself with the LocationProvider.

In the subsections that follow, you will add functionality to monitor and enable the GPS LocationProvider in the WalkAbout class. You will record changes in location as the user's path. Finally, you will display the path the user took on the map using a series of line segments and circles.

3.1 Enabling the GPS LocationProvider

It is entirely possible that the user has disabled the GPS hardware. Before you can monitor and record data from the GPS LocationProvider, the GPS hardware must be enabled. In this particular subsection, you will begin by querying the LocationManager for whether the GPS hardware is enabled on the device. If GPS is not enabled, you will allow the user to launch the Location Settings Activity to enable it from the "Enable GPS" Options MenuItem. After the user is done editing the Location Settings, they can return to the WalkAbout Activity by hitting the back button. When the GPS hardware is enabled the Recording menu item will become enabled and the "Enable GPS" menu item will no longer be visible in the Action Bar.

This is how your application should behave after finishing this substep (click image for full size):


Single use case for step 3.1.

3.1.1 Initialize LocationManager

In various parts of the WalkAbout class, you will request location information from the LocationManager attached to this application context. The WalkAbout class will store a reference to this LocationManager object in the **m_locManager** member variable for convenience.

Initialize this member variable in the **WalkAbout.initLocationData()** method:

- Retrieve the LocationManager by calling the **getSystemService(...)** method, passing in the **Context.LOCATION_SERVICE** constant.

- Set m_locManager to the object returned to you. You will have to cast the return value, because **getSystemService(...)** returns an object of type Object.

3.1.2 Dynamically Update Options Menu

The Options menu should display the "Start/Stop" MenuItem as disabled if the GPS Location Provider is disabled. Additionally, the Options menu should **NOT** display the "Enable GPS" MenuItem at all if the GPS Location Provider is enabled.

Each time the Options menu is displayed, you should check to see if the GPS Location Provider is enabled and update the "Start/Stop" and "Enable GPS" MenuItems accordingly.

- You must do this from **onPrepareOptionsMenu(...)**. If this is done from **onCreateOptionsMenu(...)** it will only be done the first time the menu is shown.
- You can determine whether a Location Provider is enabled by calling the **LocationManager.isProviderEnabled(...)** method and passing in the name of the Provider. See the Android Documentation on this method for more details.
- The names of the different Providers are stored as string constants in the LocationManager class. Check the LocationManager Constants Documentation to determine which string to pass in.

You should be able to run your application and test that the "Start/Stop" and "Enable GPS" MenuItems are enabled and invisible respectively when the GPS Provider is enabled. They should also be disabled and visible respectively when the GPS Provider is disabled.

3.1.3 Implement "Enable GPS" MenuItem

When the GPS Location Provider is disabled, the user will be able to launch the settings activity to enable the GPS Location Provider from the "Enable GPS" Options MenuItem.

Fill in the logic for the "Enable GPS" MenuItem in **onOptionsItemSelected(...)**:

- Instantiate a new Intent object, passing in the **Settings.ACTION_LOCATION_SOURCE_SETTINGS** String constant.
- Call the **startActivityForResult(...)** method passing in the intent you just created and the **WalkAbout.ENABLE_GPS_REQUEST_CODE** static constant.

    - We pass in the Request Code so that when we handle the result, we know which *request* generated the result. The result of any Activity that we launch gets processed by the same method and the Request Code mechanism lets us know which result is coming from which Activity. More on this in the next step.

You should be able to run your application and test that you can launch the settings activity to enable the GPS provider from the "Enable GPS" MenuItem. When you hit the Back button you should return to the WalkAbout Activity. However, you'll notice that the menu isn't refreshing properly to reflect GPS enabling. We'll fix that in the next part.

3.1.4 Handle the Location Settings Activity Result

After the user is done with the Location Settings Activity and returns to the WalkAbout Activity, you need to make the menu reflect that change. On older devices that relied on a physical menu button, this refresh happened automatically since the menu was always (re)constructed when the menu button was pressed (since the menu was hidden otherwise), and therefore changes to the menu in **onPrepareOptionsMenu(...)** would be invoked. However, the Action Bar is always onscreen and it is not reconstructed upon returning to the WalkAbout application. Since you called the **Activity.startActivityForResult(...)** method to launch the Location Settings Activity, the **WalkAbout.onActivityResult(...)** method will be called.

The ***requestCode*** parameter in this method will contain the value that you passed into the **startActivityForResult(...)** method. You should test this parameter to see if it matches the **WalkAbout.ENABLE_GPS_REQUEST_CODE** constant. You test the requestCode because this same method will be called when any Activity that you launch returns a result *(You must test this value because you will be launching another activity later and you want to be able to identify which Activity is returning a result)*.

Override and fill in the **onActivityResult(...)** method in WalkAbout:

- Make sure to call **super.onActivityResult(...)**.
- Test the requestCode argument to ensure that it matches the request code that you used when you launched the Intent from the above step.
    - If the request codes don't match, then don't do anything.
    - If they do match, then make a call to **supportInvalidateOptionsMenu()**. This

is the support version of the **invalidateOptionsMenu()** method call, and will schedule a reconstruction of the menu. This will factor the required menu changes into the Action Bar.

You should now run your application with all Location Providers disabled and make sure that the Action Bar menu behaves appropriately once you enable the GPS LocationProvider via changing the GPS settings (see image above for guidance).

3.2 Retrieving GPS Location Information

In this subsection, you will implement the functionality necessary to monitor and record GPS location information. You will monitor GPS location information by making the WalkAbout Activity register itself with the GPS LocationProvider. By doing this, the WalkAbout Activity will be notified by the GPS LocationProvider when the location changes. However, before the WalkAbout Activity class can register itself with the GPS LocationProvider, it must implement the **android.location.LocationListener** interface. You will start by making the WalkAbout Activity implement this interface.

You will then implement a "Start/Stop" MenuItem that will toggle the WalkAbout Activity between actively-recording and recording-stopped states. While in the actively-recording state, the WalkAbout Activity will be registered to receive updates from the GPS LocationProvider. As the WalkAbout Activity receives notices about location changes, it will re-center its MapView about the new location. It will also store changes in location as latitude-longitude coordinates in an ArrayList. Additionally, while in the actively-recording state the "Start/Stop" MenuItem will display the word "Stop" to indicate that clicking on the MenuItem will cause the WalkAbout Activity to switch to the recording-stopped state.

While in the recording-stopped state, the WalkAbout Activity will no longer be registered to receive updates from the GPS LocationProvider. Additionally, while in the actively-recording state the "Start/Stop" MenuItem will display the word "Start" to indicate that clicking on the MenuItem will cause the WalkAbout Activity to switch to the recording-stopped state. Note that when the Application starts up, it is by default in the recording-stopped state. When the user switches from the recording-stopped state to the actively-recording state, the ArrayList used to record changes in location should be cleared.

3.2.1 Implement LocationListener Interface

The WalkAbout Activity class will receive updates from the GPS Location Provider by registering itself with the Provider. In order to register with the Provider, the WalkAbout Activity must implement the LocationListener interface. This interface provides a set of callback methods that the Provider will call when the location changes, the Provider is enabled, the Provider is disabled, or when the status of the Provider changes.

In particular, you will monitor and record location changes, and stop recording in the event that the Provider is disabled for some reason. The WalkAbout class records location changes in an ArrayList<LatLng> member variable named **m_arrPathPoints**. It also maintains the current recording state (whether it is currently recording or not) in a boolean member variable named **m_bRecording**.

Perform the following initializations in the **initLocationData()** method:

- Initialize m_arrPathPoints to a new ArrayList<LatLng>.
- Initialize m_bRecording to false.

Make the WalkAbout class implement the **LocationListener** interface:

- There are four methods that you must implement. See the Android Documentation on LocationListener for a complete listing and description of each.
    - Of the four methods, you only need to fill in two of them, **onLocationChanged(...)** and **onProviderDisabled(...)** (You still need to declare the other two methods, but you can leave them as empty stubs).
- Record location changes in the **onLocationChanged(...)** method.
    - *The Location argument passed in has both degrees latitude and longitude values, represented as type double. The old way of storing points used the GeoPoint class, but v1 of Google Maps has been deprecated. Now we use the LatLng class instead, which doesn't require tedious conversions like the previous GeoPoint system required.*
    - Get the degrees latitude value from the Location argument by calling its **getLatitude()** method.
    - Get the degrees longitude value from the Location argument by calling its **getLongitude()** method.
    - Instantiate a new LatLng object, passing into the constructor the latitude

and longitude values you just obtained.

- Add the LatLng point to **m_arrPathPoints**.
- Make a call to your Google Map object's **animateCamera(...)** method, using **CameraUpdateFactory.newLatLng(...)** to create a CameraUpdate for you.
    - This is such a cool, slick and simple process in the new Maps v2 API that we are now going to take some time to explain what this does. Because it's just that awesome.

What's the Map Camera?
If you have experience in the field of computer graphics, you're very likely familiar with the subject of cameras and views (including terms such as *viewing frustum*). If you don't, that's okay--you don't need to know those things, but you do need to understand what the Android Google Maps camera is capable of and how to control it.

When you run Google Maps, you see a map on your device's screen. You are actually looking at a Google Map object from a certain relative position, or a **View**. No, this is not a View as in a ViewGroup; this is a representation of the world given a specific **projection**. In this context, think of the projection as a telescope or a pair of binoculars positioned somewhere above the world map that shows you a certain part of the world (in our case, Maps uses the Mercator projection, which is a standard for all map systems).

The View itself is represented as a **camera** that looks down on a map of the world. This camera by default is looking straight down on the map, and is thus showing you a flat 2D view of the map. The camera is represented by the following properties: location (latitude and longitude), zoom, bearing and tilt. Together, all four of these properties can collectively be referred to as the **transformation** of the camera (in computer graphics, this corresponds to a transform matrix). However, the camera's properties can be modified in a plethora of ways. Read about these properties more in detail here.

Simply put, each property represents the following:

> **location** - this is the latitude and longitude of the point the camera is looking at. It is ***not*** the location of the camera itself. In computer graphics, you could say this is the **translation** of the camera, or the point in space that the camera sits at/is moved to.

zoom - effectively the **scale** of the view of the map. The higher the zoom value, the closer the camera is moved to the location it is looking at (and therefore the closer-up the map seems), and vice-versa.

bearing - effectively the horizontal orientation of the camera, and part of the **rotation** of the camera. If you grabbed the camera where it was and twisted it left or right, that would change the orientation of the camera horizontally. The compass is a metric standard for maintaining bearing in Google Maps (just like every other map system out there).

tilt - effectively the other half of the **rotation** of the camera relative to its location. Pretend the camera sits on a half-sphere centered over the location. You can move the camera anywhere along that half-sphere and it will still look at that same location (the half-sphere will still be centered over that location), but you will see that location and its surroundings from a different viewing angle. In Maps, this gives the illusion of a 3D view.

Finally, it is interesting to note that in some rendering systems such as OpenGL, the camera actually stays still, and the entire world is moved around instead (using a reverse of the desired camera transforms). In Google Maps, it is the opposite: the camera is moved and the world stays still.

Great, now how do I code this complicated mess?

Confused? That's okay, because Google Maps API v2 makes it easy for you to manipulate the camera through the use of two classes: CameraUpdate and CameraUpdateFactory. In fact, you won't even touch the CameraUpdate class because CameraUpdateFactory (which follows the Factory design pattern) makes CameraUpdate objects that we can use to change the map's camera. We can use this Factory class to modify any of the camera's properties. It is as simple and elegant as it sounds.

There are two ways to update our Camera: move and animate. Move makes the Camera update and "jump" immediately, and animate makes the camera update smoothly. The corresponding methods are **CameraUpdateFactory.moveCamera(...)** and **CameraUpdateFactory.animateCamera(...)**. If you've used Google Maps before, then you know how pleasant it is to have smooth transitions between locations on the map instead of randomly jumping between them. We are using **animateCamera(...)** for this reason. You can even set CameraUpdate callbacks to do certain things if the transform animation completes or gets cancelled (we won't be doing this, though).

Take a look at the documentation for the **animateCamera(...)** method we are using. This method obviously transforms the camera (i.e. alters its properties), but it only alters the **location**. All it does is move the center of the screen to the designated location on the Google Map, and all other camera properties remain the same. Can you believe that's all you need to include to get the camera to constantly center itself on your current location on the map while you're moving around?

Okay, back to coding!

- In the **onProviderDisabled(...)** method, instruct the WalkAbout Activity to stop recording location changes.
    - Do this by making a call to **WalkAbout.setRecordingState(false)**. You will implement this method in the future.

3.2.2 Implement "Start/Stop" MenuItem

When the "Start/Stop" MenuItem is clicked, the recording state should be toggled. If the activity is currently recording, then the activity should stop recording. Conversely, if the activity is not recording, then the activity should start recording.

The functionality for setting the recording state will be encapsulated in the **WalkAbout.setRecordingState(...)** method. Passing in a value of true indicates that the activity should start recording and passing in a value of false indicates that the activity should stop recording.

Fill in the "Start/Stop" portion of the **onOptionsItemSelected(...)** method so that it toggles the recording state by calling **WalkAbout.setRecordingState(...)** with the proper value.

- Make use of the **m_bRecordingState** member variable to determine the current recording state.

- Make a call to **supportInvalidateOptionsMenu()**.

    - Similar to how we needed to change the Filter menu item text upon actual filtering in addition to when AdvancedJokeList was destroyed in Lab 4, we need to make changes to the menu when we aren't destroying the Activity here too.

Fill in the **setRecordingState(...)** method:

- You should update the m_bRecording member variable with the new state.
- If the new state is **not recording** you should tell the GPS Provider to stop sending you Location Updates.
    - You can do this by calling the **LocationManager.removeUpdates(...)** method and pass in the LocationListener that you want to un-register.
- Otherwise, if the new state is **recording** you should re-initialize your recording data.
    - All previously recorded LatLng points should be erased. Do not instantiate a new list of LatLng points, just clear your current one.
        - *You should try to reuse objects when possible. It is expensive to create new ones and it may be some time before the old ones get garbage collected.*
    - Initialize your list of LagLng points with the last known location from the GPS Provider.
        - You can do this by calling the **LocationManager.getLastKnownLocation(...)** method.
        - You can then manually call your **WalkAbout.onLocationChanged(...)** method with the Location you just retrieved. This will update your list of LagLng points for you.
    - Tell the GPS Provider to start sending you Location Updates.
        - You can do this by calling the LocationManager.requestLocationUpdates(...) method.
        - Pass in the String name of the Location Provider you want to receive

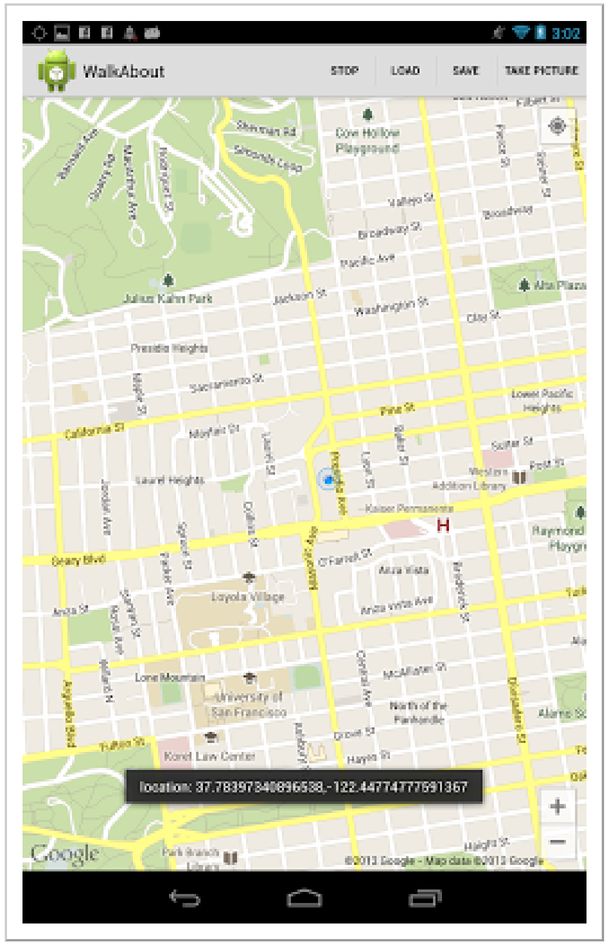
Location Updates from

- Pass in **WalkAbout.MIN_TIME_CHANGE**, to specify the minimum amount of time between updates.
- Pass in **WalkAbout.MIN_DISTANCE_CHANGE**, to specify the minimum amount of distance between updates.
- Pass in the LocationListener which should receive the updates.

If you run your application, it should now record GPS Location changes. You can test this by adding a Toast notification to the **WalkAbout.onLocationChanged(...)** method that prints out the new latitude and longitude.

- Get up and walk around! Press **Start** and take your device for a stroll. When you first activate the recording or when you reach a different latitude and/or longitude, the app should show a toast for the new location and the map will center its location appropriately.
- The application will not zoom in for you unless you press the My Location button. You may optionally animate the camera using a different CameraUpdateFactory static method to change the zoom as you see fit.
- Don't forget to remove the Toast after you've finished testing.

Below is a picture of the app running with a Toast appearing onscreen after pressing Start and moving the testing device to a new location (click image for full size):



location: 37.78397340896538,-122.44774777591367

Make sure Stop displays on the menu item after you press Start, and vice-versa.

You can check to make sure you're receiving proper latitude and longitude values by using an online latitude/longitude to location converter, such as the one found here, and comparing them to the printed Toast values.

Note: Although we are assuming you will run this application on a physical device (the recommended method since walking around is optimal for this lab), if you are running this on the emulator you can simulate location changes on the emulator two different ways. You can do this from the DDMS perspective in Eclipse, or from the Console. For instructions on how to open up a console, see the Android reference on Using the Console.

- **DDMS:** see the DDMS documentation on Emulator Controls. Feel free to use the testGPS.gpx file included in the root folder of the skeleton project as a test path.

- **Console:** see the documentation on Location Services.

3.3 Shapes

Break out the crayons, because it's time to doodle all over the map! If you're familiar with Google Maps, you know that it is capable of showing paths, markers or icons at certain locations on the map (such as when you Get Directions from point A to point B). You will add functionality to your application to visually trace the path you take as WalkAbout records your traveling path. It's nice to constantly see where we are on the map, but it would be even better if we could see where we've gone from the beginning of recording until the end. This is astoundingly simple compared to v1 of the Google Maps API thanks to the advent of Shapes.

The two shapes we will be using are Polylines and Circles.

3.3.1 A Polyline Path

A **Polyline** is a series of lines that are drawn between certain LatLng points on a Google Map. Why are they special? Because in v1 of the Android Maps API, we had to pull all sorts of strings to draw lines in the correct order on the map, with a specific color, location and projection. For those with computer graphics experience, it was akin to drawing a simple line between points (such as GL_POINTS in OpenGL): you had to get the appropriate projection and perform some conversions while looping over each point in the line to draw it on the map. In fact, have a taste of what the draw method for drawing just one line used to look like for this lab:

```
public void draw(Canvas canvas, MapView mapView, boolean shadow) {
 Projection projection = mapView.getProjection();
 int ndx = 0;

 if (m_arrPathPoints != null && m_arrPathPoints.size() > 0) {
 projection.toPixels(m_arrPathPoints.get(0), m_point);
 m_paint.setARGB(255, 0, 255, 0);
 m_paint.setAntiAlias(true);
 m_rect.set(m_point.x-START_RADIUS, m_point.y-START_RADIUS, m_point.x+START_RADIUS,
m_point.y+START_RADIUS);
 canvas.drawOval(m_rect, m_paint);
```

```
  for (ndx = 1; ndx < m_arrPathPoints.size(); ndx++) {
  projection.toPixels(m_arrPathPoints.get(ndx), m_point2);
  projection.toPixels(m_arrPathPoints.get(ndx-1), m_point);
  m_paint.setARGB(255, 255, 0, 0);
  m_paint.setStrokeWidth(PATH_WIDTH);
  canvas.drawLine(m_point.x, m_point.y, m_point2.x, m_point2.y, m_paint);
  }
  }
 }
```

This is a method that was in an entirely different class that had to be integrated into WalkAbout.java as an Overlay. Each line had to be looped over and drawn directly on the canvas that existed above the Google Map. And this is just the drawing process, the above code does not highlight the process of incorporating the lines into (or rather, over) the map. Simply put, this v1 method of drawing lines over a Google Map is horrible, but we had little choice.

In v2, the drawing ***and*** incorporation process has been simplified down to just a few lines of code in the WalkAbout class. But it is not entirely straightforward: those few lines need to be implemented in specific methods inside WalkAbout.java.

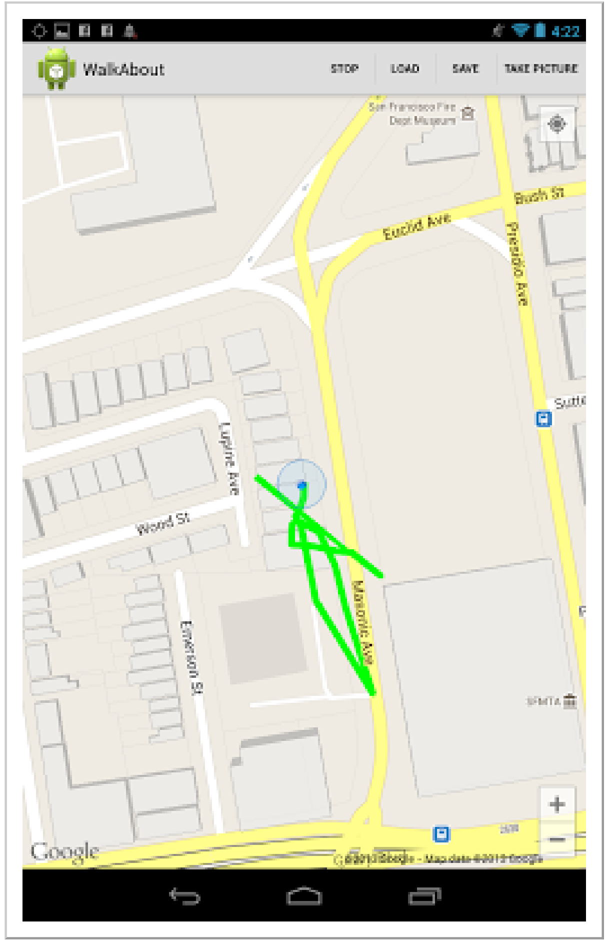
In **initLayout()**:

- After changing all of the map settings, add a new Polyline to your Google Map object, passing in a new PolylineOptions method as the parameter.
- Initialize the **m_pathLine** variable by setting it equal to the return value of the above method call.
    - *Congratulations, you now have the power to change the set of lines belonging to the Google Map in any way you wish, anywhere in your code.*
- Set the newly-retrieved Polyline's stroke color to **Color.GREEN**.
    - For Polylines, this is done using **setColor(...)**.

In **onLocationChanged(...)**:

- After adding the new LatLng point to m_arrPathPoints, make a call to your Polyline object's **setPoints(...)** method, passing in the array of LatLng points you have been keeping track of thus far.
    - Read more about what this method does in the documentation section for Polylines (see link above).

Run your application and gaze in wonder at the trail of green that follows you everywhere you go (click image for full view):



It's that simple.

3.3.2 Circle Points

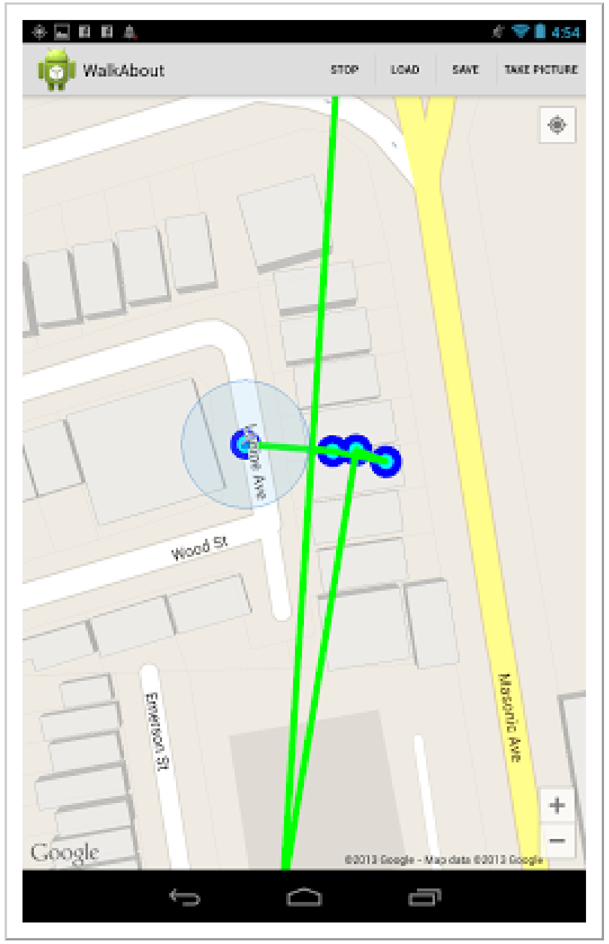
That was almost too easy, so let's also add a second Shape as well: **Circles**. You can read more about Circles here. You are going to draw a circle at each point you move to on the map. You can also guess how elegant and easy implementing these bad boys will be...Circles are implemented similarly to Polylines, only they are even simpler. Terrifying, isn't it?

In **onLocationChanged(...)**:

- Below the line where you called your Polyline variable's **setPoints(...)** method, make a call to your Google Map's **addCircle(...)** method.

- Pass in a new CircleOptions object, and initialize it as follows:
  - **center**: The point on the map where the Circle will be drawn. Pass in the LatLng point you just added to your list of points.
  - **radius**: this is the radius in *meters* of the circle you are going to draw. Pass in **WalkAbout.CIRCLE_RADIUS**. You probably want this value to be very *very* small, much smaller than the default value of 30, so experiment with different values yourself.
  - **fillColor**: The color of the Circle's inside. Pass in **Color.CYAN**.
  - **strokeColor**: The color of the Circle's outside, or its outline color. Pass in **Color.BLUE**.
  - *You can pick whichever colors you want as long as it makes your circle obvious on the map.*

Run your application from the beginning, start recording your path and walk around. You'll now see circles at each point that you visited (click image for full view):



Zoomed in extra-close to show each circle's fill color.

One more thing: we have to properly wipe the map of Polylines and Circles if we stop and start recording again. Because Polylines are tied directly to the list of LatLng points we maintain, and we clear that list in **setRecordingState(...)**, we don't have to worry about having to wipe the map of Polylines. Circles, however...

In **setRecordingState(...)**:

- If we are starting to record, make a call to your Google Map's **clear()** method.
    - This gets rid of all Shapes and Markers that we may have added to the map up

until that point.

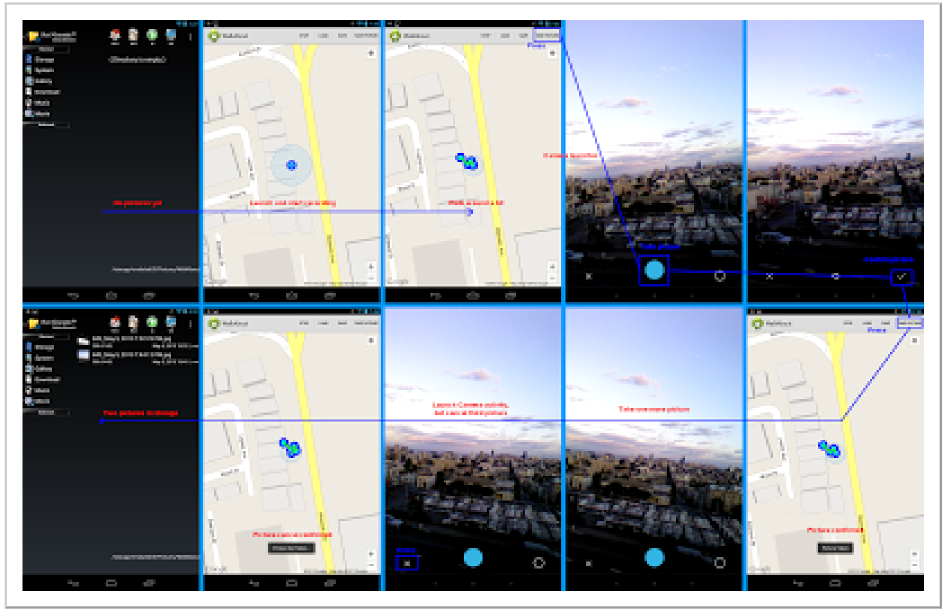
- While we don't have to worry about the Polyline on the map, we do have to add a new Polyline to the map since calling **clear()** *removes* the current one, not *clears* it.
    - Move the two lines initializing **m_pathLine** in **initLayout()** into this method. Place them after you clear the list of LatLng points and the map.

Now we have a clear visual aid that shows our path as we walk it. Now let's move onto something tougher: the camera. No, not the camera looking down through a projection at the Google Map, the camera in your Android device that lets you take pictures.

4. Camera & Picture Storage

It's picture time! Android allows applications direct access to the Camera Hardware. This means we can immediately invoke Camera features in any applications we create. If you're a user of Facebook or other applications that allow for picture-taking, you are familiar with how smooth Camera integration is (that is to say, very smooth).

Below is how your application will look and function after completing this step (click image for full view):



Several in-between steps omitted for brevity's sake.

Note: the Nexus 7 has no built-in file manager, so the first and last screens show the device's files in a file manager app.

4.1 Utilizing the Camera

By reading the Camera developer guide, you will see that there are two primary ways to take pictures: By accessing the Camera API, or by starting and capturing the result of a camera Intent. We will implement the latter. You know what that means, don't you? That's right, it's the return of URIs! You know from Lab 4 that URIs are used to access content (specifically using content URIs). Just like how the Content Provider, well...provided content, the Camera can provide us with picture-taking services if we request it.

This time around, we still need to store our application's data (the pictures) somewhere. However, a database is a terrible choice for storing images (and do we really need to integrate a database for this problem?). When you take a picture, where do you normally store images? Some form of physical internal memory, right? Let's cut to the chase--the SD card is the best place to store such images since there are known ways of loading data from and unloading data to an SD card. However, **not all devices have the capacity for an SD card**. The earliest model of the Nexus 7, for instance, has no such capability. Have no fear--we will be using a method for retrieving a device-specific storage path. That way, if there is an SD card that can be accessed and written to then it will be used, but if not then some other device-specific path will be written to instead. However, this means you may have a different path or method structure.

First, however, we want to set up for our Camera intent.

In the **Manifest** file:

- Add the following feature:

  ```
  <uses-feature android:name="android.hardware.camera" />
  ```

  - *Note: You may optionally add the android property **android:required="false"** to this XML feature declaration. What this does is tell the Google Play Store that your application uses a camera but does not **require** it for usage. If you were going to publish this application, you would have to carefully enable or disable the Take Picture menu item if you were to build around this concept.*

- While you're at it, add the following permission to allow access to write to external storage:

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

  - *This will let our application write to our device's SD card or other external storage location.*

In **WalkAbout**:

- When the WalkAbout Options Menu is displayed, the "Take Picture" menu item should only be enabled if the Activity is recording GPS location changes. If the Activity is not recording, the MenuItem should be disabled.
  - *You are left to implement this on your own. It's very likely just a single line of code.*

4.1.1 Media Files

When you take a picture, the picture gets saved *somewhere* as *something*. We'll get to the somewhere later, but for now let's focus on the image file itself, the *something*.

Android is capable of handling a variety of media file formats. Naturally Android supports various image formats, including common ones such as JPEG and PNG. We won't go into image format details. Instead, we're going to just pick the JPEG picture file format and use it in our application. The process of taking pictures in our application will go as follows:

-Run application
-Start Recording
-Click "Take Picture" menu item
-Launch Camera via intent
    -Take a picture
    -Exit Camera
-Handle Camera return
    -Save picture to storage

We will start by creating helper methods for our application to properly save an image in JPEG (.JPG) format and create a URI to pass to the Camera Intent. See the documentation as a *general* guide (not word for word) for the following steps.

In **WalkAbout**:

- Create a private static method called **getOutputMediaFile** that returns a File and takes a single integer parameter that represents a file type (although it's more of an intention than a type).
    - *Note: the Saving Media guide suggests checking to make sure that the memory device we are going to write to, is ready to be written to (mounted and opened read/write) using **Environment.getExternalStorageState()**. However, newer devices such as the Nexus 7 that employ multi-user functionality (something new to Jelly Bean) do not work properly with this call due to the way they mount storage devices. For example, the first Nexus 7 model has no physical SD card functionality.*
      *Calling Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_PICTURES) to get the storage path we need returns us the location `/storage/emulated/0/Pictures` which is one of many paths that point to the device's virtual SD card. For the purposes of this lab, we will skip over this check. This means that **this method may vary for you depending on how your***

device handles storage.

    - *See more on the topic of virtual SD cards here.*

- Create a local File variable **mediaFile** and assign it the value of **null**. This is the File that you will return from this method, and it will contain a String that represents an absolute image destination path for the picture we take using the Camera. This absolute path contains both the storage directory to store the image and the image name itself.

    - *An absolute path that contains a directory and a file name? Doesn't this sound similar to our Lab 4 URIs and how they have an authority and a path?*

- Fetch the Pictures storage directory on the device:

    - Instantiate a new File using **java.io.File**'s constructor.

        - The first parameter needs to be the base directory where the file gets stored. Fetch it by making a call to **Environment.getExternalStoragePublicDirectory(...)**, passing in **Environment.DIRECTORY_PICTURES** as the parameter.

        - The second parameter is the name of the storage file. In our case, this file will be the folder inside of the Pictures directory for our application, so that would be **"WalkAbout"**.

        - *Using the above note as an example, if we are trying to get the Pictures storage directory of a Nexus 7 then the first parameter translates to* `/storage/emulated/0/Pictures` *and the second parameter appends* `WalkAbout` *to the end of that, giving us* `/storage/emulated/0/Pictures/WalkAbout` *as the full storage* *directory path.*

        - Create a local File variable named **mediaStorageDir**, and assign it the return value of the above File instantiation. This will hold the full storage directory path.

    - Now that we have the storage directory path, we have to make sure that the file at that path actually exists before continuing:

        - Check to see if there is **NOT** a file at the storage directory path contained in mediaStorageDir by calling its **exists()** method.

        - If there isn't one, we have to create it. Call mediaStorageDir's **mkdirs()** method and check to see if it did **NOT** create the directory appropriately.

        - If both of these checks fail, send a debug Log message to Logcat from WalkAbout stating that the directory creation process was a fail, then return **null**.

    - Now we know there is a place to store the file we are about to create, so we can finally create the picture file's absolute path:

        - Create a new String local variable and assign it the value of calling **DateFormat.getDateTimeInstance().format(...)**.

            - For the **format(...)** method, pass in a new Date that takes the current system time in milliseconds.

                - See the documentation for the System Java class for more information.

            - *This effectively gives us a timestamp in the format* ***`<Month> <Day>`****,*

`<Year> <HH:MM:SS> <AM/PM>`. An example of this would be `May 5, 2013 4:37:18 PM`.

- Check to see if the type passed in matches **PICTURE_REQUEST_CODE**, as this is the indicator that we want to create an image File.

    - *This type of checking is similar to URIMatcher from Lab 4, and it is important if you want to work with multiple types and/or categories of media file types. We only want to create JPEG Files, but we are following this convention anyway.*

    - If the type matches, assign **mediaFile** the result of instantiating a new File with the following path (concatenate the following all together in order):
        - **mediaStorageDir.getPath()** (the full Pictures storage directory path)
        - A file separator (use **File.separator** for consistency and flexibility)
        - The string **"IMG_"**
        - The timestamp
        - The string **".jpg"**

    - **mediaFile** should now contain a path similar to the following (again, using the ongoing Nexus 7 example): `/storage/emulated/0/Pictures/WalkAbout/IMG_May 5, 2013 4:37:18 PM.jpg`

        - *Your path will vary, particularly the first part of the directory (the full Pictures storage directory path).*

- Return **mediaFile**.

- Create a private static method called **getOutputMediaFileUri(...)** that returns a Uri and takes a single integer parameter that represents a file type (although it's more of an intention than a type).
    - Create a local File variable and assign it the value of calling your **getOutputMediaFile(...)** method.
    - Return the result of calling **Uri.fromFile(...)** on the File variable.
    - *This will parse the path of the file created in **getOutputMediaFile(...)** and make it part of a recognizable URI.*

4.1.2 Launching the Camera

Now invoking the Camera using an Intent is a piece of cake thanks to the above helper methods.

- In **onOptionsItemSelected(...)**, if the **Take Picture** menu item is pressed:
    - Create and instantiate a new Intent variable, passing in the action **MediaStore.ACTION_IMAGE_CAPTURE**.
    - Create a local URI variable from a call to **getOutPutMediaFileUri(...)**. Pass in the appropriate type.
        - You have **PICTURE_REQUEST_CODE** and **ENABLE_GPS_REQUEST_CODE** to choose between.
    - Put the URI into the Intent variable using **putExtra(...)**, passing in **MediaStore.EXTRA_OUTPUT** for the first parameter.

    - Call **startActivityForResult(...)** similarly to how you did for enabling GPS.

We still have to take care of business after the Camera activity exits before we can test picture-taking functionality.

4.2 Handling the Camera's Exit

Assuming the Camera works properly once we invoke it (a fairly safe assumption to make), we now have to pick up where it leaves off when it exits.

In **onActivityResult(...)**:

- Check the case where the requestCode is equal to **PICTURE_REQUEST_CODE**.
    - If it matches, check **resultCode**. This is the status of the Activity we're picking up after. It left us a status code upon finishing or exiting.
        - If **resultCode** matches **RESULT_OK**, this means that the picture was correctly taken and saved as an image to the path we provided it.
            - Display a Toast using **R.string.pictureSuccess** as the message.
        - If resultCode matches **RESULT_CANCELED**, this means that the Activity's operation was canceled and did not complete successfully.
            - Display a Toast using **R.string.pictureFail** as the message.

Run your application and make sure that it functions as outlined in the storyboard image at the top of section 4. If not, read on for possible fixes to common problems.

4.2.1 Troubleshooting

If you are having trouble getting the Camera functionality working, there are some possible fixes. Consult the issues and solutions below for more details if you are having trouble.

1. Devices that run older versions of Android may have issues with the media storage directory path if it is formatted with the timestamp using DateFormat. Try removing spaces or other unwanted characters from the path obtained from **mediaStorageDir** in **getOutputMediaFile(...)** before assigning a value to **mediaFile**.

2. Depending on your test device, launching the Camera may result in WalkAbout being destroyed due to orientation change or other reasons. If this is the case, you can either prevent configuration or orientation changes, or you will need to save and load your application's data before and after your activity is destroyed and recreated, respectively.

You are tasked to do this on your own. If you choose to do the latter you can use your knowledge of Instance State from Lab 4, or by using other means. Several hints follow:

- You will likely make good use of **putParcelable(...)** to place objects in a Bundle to be saved for later. You've been working with several Parcelable objects already.
- Save and load the recording state, and remember to update menu items respectively.
- You will likely need to make changes to your Manifest file to allow for orientation change.

5. Markers

So we have a working camera feature, great! However, right now it has nothing to do with the map whatsoever. Let's change that by marking each location you take a picture at on the map with a timestamp. We'll do this using Markers. Markers are added similarly to Circles, but they can contain much more information inside them. To keep things simple, we'll just display each marker with a title that contains the timestamp.

In **WalkAbout**:

- Create a new class-level static variable of type String called **timestamp**.
- In **initLocationData()**, initialize the **m_arrPicturePoints** array to a new ArrayList of type Marker.

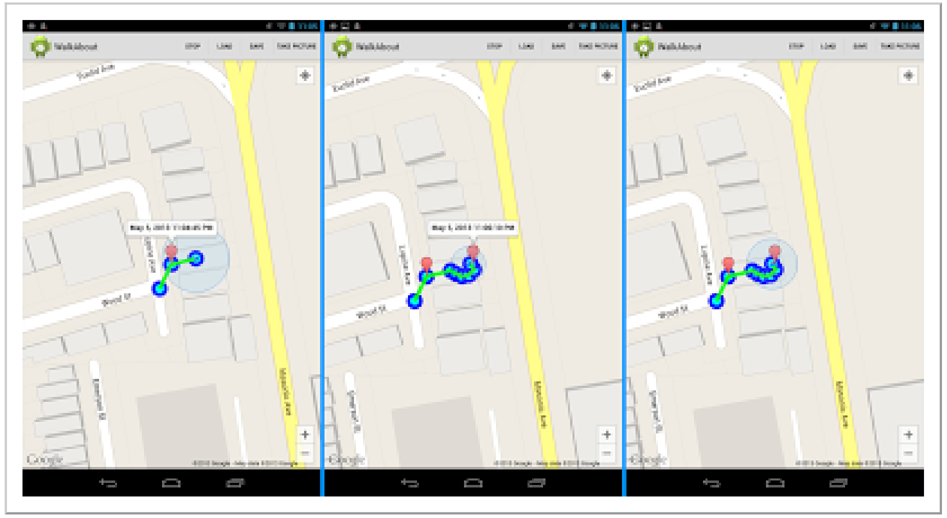
    - *We will use this variable to keep track of all marked points for later.*

- In **getOutputMediaFile(...)**, when you create the timestamp for the media path name, set your new timestamp variable equal to that timestamp.
    - *This way, you always have the latest timestamp for any picture you may take.*

- In **onActivityResult(...)**, after receiving **RESULT_OK** from the Camera:
    - Retrieve the last known Location from the LocationManager.
    - Add a new Marker to **m_arrPicturePoints** with the appropriate location and title.
        - *A Marker is returned whenever you add one to the map.*

- In **setRecordingState(...)**, if you are starting a new recording, clear the list of Markers as well.

Run your application, and now whenever you take a picture at a given location WalkAbout should mark it on the map. Touching a marker gives it focus and causes it to display its title if it has one: (click image for full size):

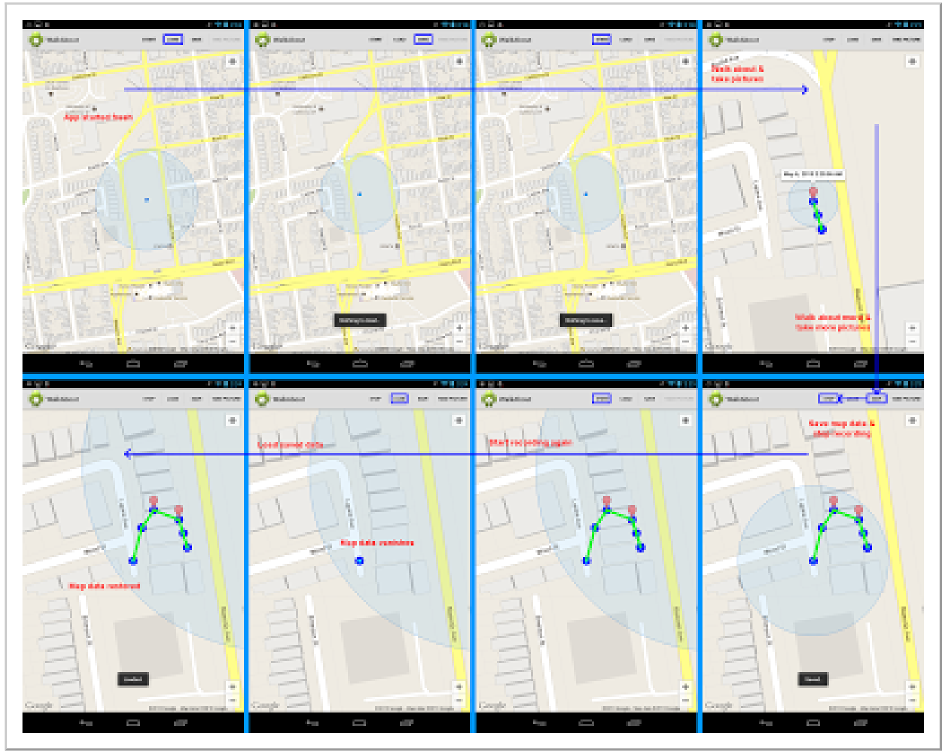

You can touch the map to hide a marker's title.

6. Private Application Files

The home stretch! Our application is nearing completion, but if the previous lab taught us anything, we want to preserve as much data as possible. In the case of WalkAbout, this corresponds to the last recorded path and all of its markers.

Android allows you to Create, Write, Read and Delete local files. These files become private to the application that created them by default. You have the option of overriding this privacy mechanism, allowing them to be shared with other applications. You will save certain map data to a private WalkAbout file and then load it later, refreshing your app data and the Google Map in the process.

The following shows how your application should behave after implementing saving and loading (click image for full size):


App started fresh
Walk about & take pictures
Walk about more & take more pictures
Save map data & start recording
Start recording again
Map data vanishes
Load saved data
Map data restored

6.1 Creating, Writing, & Deleting a File

You will now implement WalkAbout Activity's "Save" menu item logic. When selected, this should only make a single call to the **WalkAbout.saveRecording()** method. The functionality necessary to save the current recorded path should be composed in the **WalkAbout.saveRecording()** method. You should fill in this method so that it properly writes out only the contents of **m_arrPathPoints** and **m_arrPicturePoints** to a private application file. You should only ever create one file and it should be truncated/cleared *each* time you write to it.

The format of the file is simple. It will contain two lines. The contents of **m_arrPathPoints** will be written into a single continuous line, and then on the next line will be the contents of **m_arrPicturePoints**.

On the first line, the latitude and longitude of each point in **m_arrPathPoints** should be written out in that order, separated by a comma, and no spaces between them. Each point should be separated by a semicolon, and no spaces between them. So for example:

Given:

```
m_arrPathPoints = [(lat1,lng1),(lat2,lng2),(lat3,lng3)]
```

The line in the file should look like:

```
lat1,lng1;lat2,lng2;lat3,lng3;
```

On the second line, the latitude, longitude and title (timestamp) of each point in **m_arrPicturePoints** should be written out in that order, separated by a comma, and no spaces between them. Each point should be separated by a semicolon, and no spaces between them. So for example:

Given:

```
m_arrPicturePoints = [(lat1,lng1,title1),(lat2,lng2,title2)]
```

The line in the file should look like:

```
lat1,lng1,title1;lat2,lng2,title2;
```

If the Save was performed successfully, then a Toast notification should be displayed containing the **R.string.saveSuccess** resource string. If an exception is thrown you should display a Toast notification containing the **R.string.saveFailed** resource string. If there is no data to save, as is the case on the initial loading of the application, then a Toast notification should be displayed containing the **R.string.saveNoData** resource string.

You are tasked to do this on your own. However, you should make use of the following hints:

- Use the **R.string.LatLngPathFileName** string resource as the filename.
- You will have to make use of the Context.openFileOutput(String name, int mode) and Context.deleteFile(String name) methods.
- You will have to use the **Context.MODE_PRIVATE** constant.
- The comma in a Marker's timestamp may interfere with your delimitation when loading. You may handle this as you see fit.
- Documentation on the java.io.PrintWriter class *(This is merely a suggestion, you are free to use whatever Java I/O classes you would like to use to write to the file)*.

6.2 Reading a File

Your last task is to implement the WalkAbout Activity's "Load" menu item logic. When selected, this should only set the recording state to false and make a call to the **WalkAbout.loadRecording()** method. You should fill in the **loadRecording()** method so that it properly initializes **m_arrPathPoints** and **m_arrPicturePoints** to contain only the data in the file that **saveRecording()** writes out, and then repopulate your Google Map with all of the circles, the polyline, and markers from before. Once finished, the path loaded from the file should be displayed exactly as it was when it was first recorded.

If there is nothing to be loaded (before Save has been pressed), then a Toast notification should be displayed containing the **R.string.loadNoFile** resource string. If the Load was performed successfully, then a Toast notification should be displayed containing the **R.string.loadSuccess** resource string. If an exception is thrown, you should display a Toast notification containing the **R.string.loadFailed** resource string.

You are to do this task on your own. However, you should make use of the following hints:

- Use the **R.string.LatLngPathFileName** string resource as the filename.
- You will have to make use of the Context.openFileInput(String name) method in order to open the file.
- The comma in a Marker's timestamp may interfere with your delimitation. You may handle this as you see fit.
- Documentation on the java.util.Scanner class *(This is merely a suggestion, you are free to use whatever Java I/O classes you would like to use to read from the file).*

Run your application and make sure that the saving and loading functionality is working properly. When you demo to Dr. Janzen, be sure you have a way to show the picture that was taken. For instance, you might install a file manager (e.g. Astro File Manager from Play Store), and know the path for where your app is storing images (e.g. /storage/emulated/0/Pictures/WalkAbout).

In conclusion...
In this lab you created an application that tracks a user in an interactive Google Map using various Shapes. You summoned the device's camera to take pictures and store them internally in phone storage, and the user's picture-taking location was recorded on the map in the form of a Marker with the picture timestamp. You also made it so the user can save their recorded path and then load it into the map later.

As you have seen and can likely guess, there is much more you can do with Google Maps for Android. This lab offered only a small sample of the power that map fragments have. The Android Google Maps API v2 is a simple, elegant and powerful API for map-related activities. You are highly encouraged to further explore Google Maps for Android beyond the scope of this lab. If you choose to do so, start at the Maps documentation homepage here or refer to previously visited documentation pages that you visited while working through this lab. Happy Mapping!

7. Deliverables

To complete this lab you will be required to:

1. Put your entire project directory into a .zip or .tar file, similar to the stub you were given. Submit the archive to PolyLearn. This effectively provides time-stamped evidence that you submitted the lab on time should there be any discrepancy later on in the quarter. The name of your archive should be **lab6<userid>** with a **.zip** or **.tar** extension. So if your username is **jsmith** and you created a zip file, then your file would be named **lab6jsmith.zip**.
2. Load your app on a mobile device and bring it to class on the due date to demo for full credit.
3. Complete the survey for Lab 6**:** https://www.surveymonkey.com/s/435F13Lab6

Primary Authors: James Reed and Kennedy Owen
Adviser: Dr. David Janzen

Comments

You do not have permission to add comments.