CHAPTER 7

We need to understand these thoughtfully!

Location Sensors

Android devices typically have two location sensors:

One sensor (network sensor):

1. **Network Sensor:**
   The first is based on the cell towers and/or the Wi-Fi access points your device is connected to. This sensor provides the approximate location of the device.
2. **GPS Sensor:**
   The other sensor is based on a built-in Global Positioning System (GPS) receiver. This sensor can provide position information **accurate to within a few meters, depending on conditions.**

However, the GPS sensor is much slower in acquiring its position information than the network sensor. Additionally, not all devices have a GPS sensor.

Maps

Maps are implemented using the GoogleMap object in the source code file and a `Map Fragment` in the layout file. These objects are not a part of the standard Android SDK but rather the Google Play Services SDK.

This SDK must be installed on your development machine to implement maps in your app.

Using Google Maps requires an API key. This key associates your app with an attempt to access the GoogleMap API.

This is how you, and Google, can track how often your users access the map portion of your app. The API key is free. Maps are implemented as a `MapFragment` widget in a layout.

The `Activity` that implements the code to provide the map's behavior must be a `FragmentActivity.`

Fragments

The `FragmentActivity` is a subclass of the `Activity` class.

An `Activity` that needs to implement a map must extend the `FragmentActivity` class rather than the `Activity` class.

This is required because maps are encapsulated in a `MapFragment` .

This allows a map to be a part of a layout rather than the only thing in a layout.

CHAPTER 8

From here we will have the code!

Sensors

the Android platform supports about 12 sensors, there is no requirement that a manufacturer of an Android device include all of them. For this reason, it is good practice to always check for the presence of a sensor before attempting to use it.

Sensors supported range from sensors that measure the devices' ambient environment, including temperature, relative humidity, atmospheric pressure, magnetic field, and light level, to sensors that detect how the device is moving or rotating.

Sensors are accessed through the `SensorManager` class.

The `SensorManager` is a system service and not instantiated by an app.

Two other items are needed to work with sensors, `SensorEvent` and `SensorEventListener` .

A `SensorEvent` is an object that is created by a Sensor when it has something to report.

Managers

Managers are objects to facilitate the monitoring of the status of this hardware.

For example, the Android OS has a `BatteryManager` that can be used to monitor the battery's status, a `StorageManager` that can be used to monitor the status of long-term storage, and a `PowerManager` that can be used to monitor power consumption.

Other Hardware

Android devices also have other hardware features, such as a phone and a camera. **These devices have an app associated with them to provide access to their functionality.** these hardware items are accessed by making calls to their **Application Program Interface (API).**

These apps can be opened **from within an app** to give the user access to their functionality. In this case, the user **leaves** the app to interact with the device and **returns to the app after completing the task**.

For more info return to the book at page **174.**

We use sensors to communicate with sensors and we use API to communicate with a hardware :)

Battery Code

Listing 8.2 **Monitoring the Battery**

```
BroadcastReceiver batteryReceiver = new BroadcastReceiver() {                    //1
@Override
public void onReceive(Context context, Intent intent) {
double batteryLevel= intent.getIntExtra(BatteryManager. EXTRA_LEVEL ,0);          //2
double levelScale= intent.getIntExtra(BatteryManager. EXTRA_SCALE ,0);            //3
int batteryPercent = ( int ) Math. floor (batteryLevel/levelScale*100);          //4
TextView textBatteryState=(TextView)findViewById(R.id. textBatteryLevel );
textBatteryState.setText(batteryPercent+ "%" );
}
};
IntentFilter filter = new IntentFilter(Intent. ACTION_BATTERY_CHANGED );          //5
registerReceiver(batteryReceiver, filter);                                        //6
```

//1 : A `BroadcastReceiver` receives `Intents` and has the code used to respond to the `Intent` . An `Intent` is broadcast from other apps or objects executing on the device.

//2 : This line gets the extra associated with the battery's current charge level from the broadcasted `intent`.

//3 : This line gets the extra associated with the scale used for measuring the charge from the broadcasted `intent`.

//5 : An `IntentFilter` listens for Intents that have been broadcast by the system and only lets through the ones the developer is looking for.

//6 : The `BroadcastReceiver` is registered, which means that the app is told to listen for battery status intents and handle them with the `BroadcastReceiver` defined in the activity.

Compass Code

Listing 8.5 **`SensorEventListener`** Code

```
private SensorEventListener mySensorEventListener = new SensorEventListener() {
        public void onAccuracyChanged(Sensor sensor, int accuracy) { }              //1
        float [] accelerometerValues ;                                               //2
        float [] magneticValues ;
        public void onSensorChanged(SensorEvent event) {                            //3
                if (event.sensor .getType() == Sensor.TYPE_ACCELEROMETER )
                accelerometerValues = event.values ;
                if (event.sensor .getType() == Sensor.TYPE_MAGNETIC_FIELD )
                magneticValues = event.values ;
                if ( accelerometerValues != null && magneticValues != null ) {      //4
                        float R[] = new float [9];
                        float I[] = new float [9];
                        boolean    success    =    SensorManager.getRotationMatrix    (R,    I,
                        accelerometerValues , magneticValues );
                        if (success) {                                               //5
                        float orientation[] = new float [3];
                        SensorManager.getOrientation (R, orientation);
                        float azimut = ( float ) Math.toDegrees (orientation[0]);    //6
                        if (azimut < 0.0f) { azimut+=360.0f;}                        //7
                        String direction;
                        if (azimut >= 315 || azimut < 45) { direction = "N" ; }          //8
                        else if (azimut >= 225 && azimut < 315) { direction = "W" ; }
                        else if (azimut >= 135 && azimut < 225) { direction = "S" ; }
                        else { direction = "E" ; }
                        textDirection.setText(direction);
                        }
                }
        }
};
```

1// : A SensorEventListener requires the implementation of two events, onAccuracyChanged and onSensorChanged.

2// : Two variables to hold the response from each sensor are declared.

3// : The onSensorEvent first determines which sensor triggered the event and then captures the values it provided.

4// : If there are values available for both sensors, the SensorManager is asked for two rotational matrices used for orientation calculation.

5// : If the matrices are successfully calculated, the SensorManager is asked to calculate the orientation of the device. Orientation is measured in three dimensions.

Calling Phone Code:

Listing 8.7 **callContact** Method

```
private void callContact(String phoneNumber) {
        Intent intent = new Intent(Intent.ACTION_CALL );                  //1
        intent.setData(Uri.parse ( "tel:" + phoneNumber));                        //2
        startActivity(intent);
}
```

1// :A new intent is instantiated with the parameter Intent.ACTION_CALL , which tells Android that you want to use the phone to make a call.

2// :The telephone number to be called is passed to the intent as a Uniform Resource Identifier (URI). A URI is similar to a Uniform Resource Locator (URL) except that a URL identifies a location on the World Wide Web, whereas a URI can be used to identify a local resource.

Camera Code

Listing 8.10 Starting the Camera and Capturing the Result

```
public void takePhoto(){
        Intent cameraIntent = new Intent(android.provider.MediaStore. ACTION_IMAGE_CAPTURE );
        //1
        startActivityForResult(cameraIntent, CAMERA_REQUEST ); //2
}
```

1// : A new `intent` is instantiated with a parameter that tells the system to open the camera in image capture mode.

2// : you want the activity to return a value to the app after it has completed, so you use the `startActivityForResult` method.

CHAPTER 9

What is XCode?

IOS Project Files (construction of IOS application inside XCode):

1. **AppDelegate.h and AppDelegate.m**—The App Delegate files manage issues related to the entire app and are primarily used to manage the life cycle of the app—how it is started, what happens when it goes to the background, and so on.
2. **Main.storyboard**—The storyboard is used to design the interaction between multiple screens in your app as well as designing the layout of the individual screens.
3. **ViewController.h and ViewController.m**—The view controller contains the code that controls the user interactions with the app.
4. **Images.xcassets**—This folder contains all the images, including icons, needed for your app.
5. **Supporting Files**—This directory contains a number of files that the app may or may not use. Here's a description of a few of them:
    a. **appname-Info.plist**—This file contains a few app-specific settings. Most of these are controlled in other parts of Xcode.
    b. **main.m**—The file that is responsible for launching the app.
6. **Frameworks**—These are various libraries that you can include in your project to add functionality to your app,
7. **Products**—This is your compiled app file.

App Behavior

We will need to create what's called **`outlets`** for those user interface elements we want to be able to access from the code.

If we need to interact with a UI element on a specific event we should create an **`action`**.

CHAPTER 10

Memorize the definitions of these important topics.

View Controller

The View Controller is managed in three files:

1. **a .storyboard file** that specifies the layout of user interface elements
2. **a .h file** that has **information** about any **outlets and actions** needed to control the user interface
3. **and a .m file** that contains the **implementation** of the user interface **actions**

as well as any setup needed for the user interface.

In some situations **there is no storyboard file for a view controller**. Instead, the user interface is described **entirely in code**.

Tab Bar Controller

The Tab Bar Controller shows up at the bottom of iPhone apps and allows the user to choose Between different screens in the app.

Navigation Controller

The Navigation Controller is used to allow the user to drill down through multiple screens while keeping track of the path so the user can later go back the same way.

There might be questions about how to and steps ☺ TF MSQ type, but mostly theoretical according to Dr. habib.

CHAPTER 11

Persistent data

There are several ways that you can save data on iOS:

File Data Storage

Like most other operating systems, iOS enables **saving data in files**, either in **regular text files** or **by archiving** (what's known as ***serialization*** in Java and C#).

On iOS, **apps are sandboxed**, which means that each app is **isolated** from the other apps and from the operating system.

One of the consequences is that each app has only a very simple file system that by default consists of a few standard directories: **Documents**, **Library**, and **tmp**.

As a developer, you can store files in the **Documents** and **tmp** directories, while you only read only in the **library** folder. The **Documents** folder **is backed up** when the device is backed up. **The tmp folder is not**.

Comparison between android and iOS File data storage:

IOS	Android
iOS can store files in sequential file format or file storage	Android saves the file in serialization (archiving format only)
iOS can store files in 3 folders, Library, Documents and tmp.	Files can be written to either internal or external storage.
Files are private to the app	Files are private to the app if they are stored in the internal storage, they are not in the external.

User Defaults

When you need to save **a little bit of data** in your app, the **NSUserDefaults** object is a very simple and easy way to do so.

NSUserDefaults is a front-end to a key-value file (often referred to **as Plist files** because of the .plist extension) that is stored in the app's Preferences directory.

There's **only one file**, but in this file you can store **as many values as you want**.

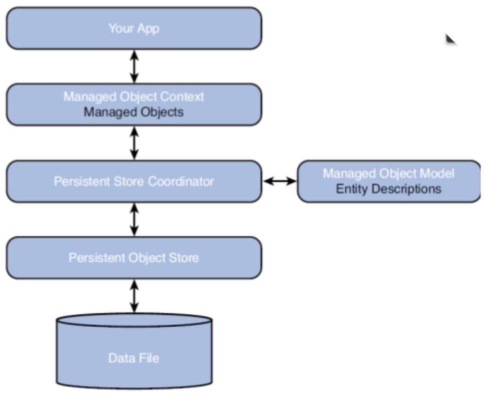
When storing values in the file, you have to **supply a key string** to identify the value when retrieving it later.

You can store many data types in **NSUserDefaults** , including all scalar types as well as

NSData , **NSString** , **NSNumber** , **NSDate** , **NSDictionary** , and **NSArray** . Other data types **can be archived and stored** as an **NSData** object.

In android, NSUserDefaults is equal to shared preferences.

Core Data



Core Data is a **powerful data persistence solution** developed by Apple to provide **object-oriented storage**. Core Data is an object-wrapper on top of a data store—**typically a SQLite database**. This allows the developer to work with **objects** that **map to entities** in the database **without worrying too much about the underlying database design and queries**. For apps that need business data stored, **Core Data is a strong solution to meet this need**.

DATA FILE:

The framework stores data in files. The default is SQLite database, but you can also choose to use XML or binary data.

PERSISTENT OBJECT STORE:

Wraps around the data file and presents a common interface to the rest of the stack.

THE PERSISTENT STORE COORDINATOR:

Allows for having multiple data stores in the same app, and will then coordinate access to those stores.

THE MANAGED OBJECT MODEL:

Is where the description of the layout of the data is defined.

THE MANAGED OBJECT CONTEXT:

Which allows you to access the objects that are stored in the data file.

The Managed Object Context can keep track of multiple changes to the objects, and will periodically, or when instructed, save the changes to the persistent store.

Working with **`NSUserDefaults`** Object

Firstly, Standard `NSUserDefaults` object with this line of code:

```
NSUserDefaults *settings = [NSUserDefaults standardUserDefaults];
```

Then, To store a value in the settings object, you use code like this:

```
[settings setObject:@"City" forKey:@"sortField"];
```

This **saves** the **value** “`City`” with the **key** “`sortField`”. **Every value must have a unique key**. In addition to the `setObject:forKey` : method, there are also methods that can be used to store scalar values such as **BOOL** and **int** . Here’s an example:

```
[settings setBool:YES forKey:@"sortDirectionAscending"];
```

After that, retrieving the data is equally simple. Using the reference called settings to the `NSUserDefaults` object, you would retrieve an object using this call:

```
NSString *sortField = [settings objectForKey:@"sortField" ];
```

Additionally, the data is periodically saved, **but to force saving**, you can call the `synchronize` method:

```
[settings synchronize];
```

This will **save any changes you’ve made to the Plist file**.

CHAPTER 15

Focus on the 3 strategies, other content may come as TF and MSQ

App Monetization Strategies

The app stores take a 30% cut of all the revenue that your app generates.

PAID APPS

The price is advertised in the app store and the user decides, based on your description of the app, whether to buy it. If the user buys the app, you get the money.

AD SUPPORTED APPS

The most common approach to making money from a free app is to embed ads within the app screens.

Ads take up screen real-estate, so you will have to plan and code the user interface with this in mind.

Google add platform is called **AddMob**, Apple has **iAds**, you can use Google **AddMob** in Apple devices.

Each click generates only a few cents, so you need a lot of clicks to make any real income.

The money it generates is measured in dollars per month.

Unlike Google's ad service, iAd pays both **for clicks** on the ad **and per impression** (**displays of ad in your app)**. However, the **rate per impression is very small**.

IN-APP PURCHASES

The basic theory for an in-app purchase monetization strategy is that you **generate downloads with the free app**, get the users hooked, and then allow them to **add** features by **advertising the feature** in the app. **The sale is made during use of the app.**

More than 75% of the revenue going to iPhone developers in February 2013 came from in-app purchases.

Another advantage over the basic paid app is that in-app purchasing opens up the possibility of a **regular revenue stream from the same user**, instead of relying on a **single purchase up front**.

Often an in-**app purchase strategy is combined with an ad-supported strategy**. The free version **includes ads** that are **eliminated** as a bonus **for an in-app purchase**.

The Economics of App Stores

Often developers are surprised that they have to pay Google and Apple **30% of any sales made in the app stores.**

Owning Your Own Business

The vehicle for setting up your own business is in most cases a Limited Liability Corporation, or LLC.

CREATE AN LLC

An LLC is a legal entity that is organized by registering the business with a state. The process should take no more than an hour or so and cost under $200.

CHAPTER 16

I suggest that you read the chapter from the book, the

App Distribution through the App/Play Stores

To publish an app in either of these stores, a developer must configure the app and conform to the requirements established by their sponsors.

Both Google and Apple have requirements and guidelines for apps that should be incorporated during development.

Apple is especially meticulous about these rules. **Every app is reviewed before it is published**, and if your app does **not** conform, it will be **rejected**.

Google **will publish an application that does not meet its store requirements**, but will **remove** the app from the store **later if** it finds that the app **violates its rules**.

The preparation is similar for both stores.

1. You will need to prepare **an icon** for your app.
   Android requires an app icon sized to 512 x 512, whereas iOS requires the icon to be 1024 x 1024 pixels.
2. Both platforms require **at least one screenshot** of your app **for each targeted screen size**. You can provide up to eight screenshots for each targeted Android device and up to four screenshots for each targeted iOS device.
   a. For **iOS** you will have to **prepare a Splash screen.**
3. Prepare a **description** of your app.
4. Provide a **description** of any **specific conditions** or requirements that **the tester will need** to know to adequately review your app.
5. Determining **what price** you want to charge for the app.
   In Android, **you enter this price.** In iOS you will be prompted to **select from a set** of pricing **tiers**.
6. You should also determine **what general category** best describes the app (for example, game, sports, tool, and so on)
7. You will need to determine **in which countries** your app should be made **available**.

Android Play Store Distribution

The Google Play Store **does not provide any copy protection for your app**. Instead, Google provides a **Licensing Library**, which allows developers to add copy protection to their apps.

Each time the user **opens** the app, **the Play Store is queried** to determine whether **the user bought** the app. If the user **did not buy the app**, the developer **can have the app close** or take **whatever action** is deemed appropriate. If you do **not** use licensing, your app **can be copied** to other devices quite easily.

Then the app needs to be **compiled into a signed Android application package** (APK) file. Prior to compiling the app, you need to go through the code and **remove any logging operations**, **all debug breakpoints,** and **address as many of the warnings** identified by Eclipse as you think necessary.

iOS App Store Distribution

The Apple App Store **provides strong controls over illegal copying of your app**. As a developer, unlike in Android, **you don't have to worry** about setting up any licensing.

The first step is to set up your **Distribution Certificate and Distribution Provisioning Profile**.

The next step is to **set up an entry for your app in iTunes Connect**, iTunes Connect is the website that is used to manage many aspects of your app, including seeing reports on how the app is performing.

In iTunes Connect you provide information about the app, **pricing, and screenshots** prior to being able to upload the app.

App Distribution for the Enterprise

App distribution within an organization differs between iOS and Android. Generally, distribution is easier because you are not required to conform to the specifications of the Play or App stores.

ANDROID ENTERPRISE DISTRIBUTION:

You **prepare an APK** just like you did for the Play Store, and then you **give it to your users**. The easiest way to do this is by sending users an email with the APK attached.

However, for this to occur, users must have set their device to **accept apps from unknown sources**.

Many organizations also implement **Mobile Device Management (MDM) solutions** to manage their mobile devices and app distribution.

IOS ENTERPRISE DISTRIBUTION

Distribution of iOS apps within the organization is **a bit more complicated in iOS** than it is in Android. The first step is to **get an iOS Enterprise Developer license**. The cost of the license is **$299** per year but allows unlimited distribution of apps within the organization.

Distributing within the organization requires **setting up both an enterprise distribution certificate and an enterprise distribution provisioning profile.** These are **then packaged with the app using**

Xcode. There is no need to use iTunes Connect with in-house apps. However, **the provisioning profile expires after a year.** Prior to that time, **a new profile must be created, packaged** with the app, **and redistributed**, or the app will **stop working**.

Testing and Fragmentation

A comprehensive test plan should include thorough black box unit testing, including equivalence partitioning, boundary value analysis, and cause-effect graphing.

Keeping Up with the Platform

Updates to the operating system can, and although infrequently, do disrupt apps that previously ran fine. To avoid problems with users of your app or to prevent getting bad reviews in the app stores, it pays to keep on top of platform changes.

This is a long topic and I suggest to read it from the book once.

لا تنسونا من صالح الدعاء!

وفق الله الجميع لما فيه الخير