

Давайте напишем приложение, которое при смене местоположения отправляет запрос сайту с погодой, а потом выводит её пользователю. Начнём с той сущности, которая должна быть в любом приложении.

1 Android Manifest

Манифест - файл, лежащий в директории main.

- В Manifest-файле описывается сам проект, в том числе настройки и конфигурации (например, версии проекта).
- Права, которые должны запросить другие приложения для получения доступа к вашему приложению.
- Составляющие проекта, такие как:
 1. Activity
 2. BroadcastReceiver
 3. Service

О них мы поговорим дальше.

Пока что наш манифест выглядит так:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3     package="edu.spbau.android.forecast" >
4
5     <application
6         android:allowBackup="true"
7         android:icon="@mipmap/ic_launcher"
8         android:label="@string/app_name"
9         android:theme="@style/AppTheme" >
10    </application>
11
12 </manifest>
```

Внутри тега Application как раз должны быть описаны части приложения, а также устанавливаются настройки приложения. icon, label, theme - описывают значок приложения в меню телефона.

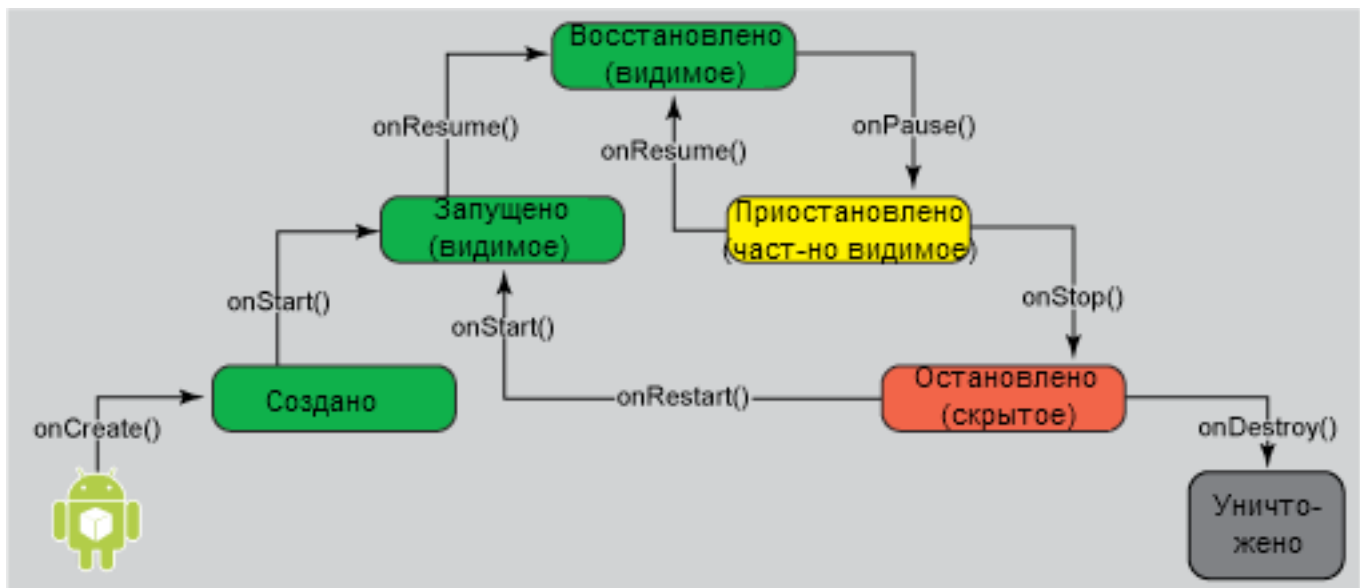
В принципе, части приложения можно регистрировать и программно. Но мы этого касаться не будем.

2 Activity

Поговорим о Activity. Activity - это окно приложения. Простые приложения состоят из одного Activity. Более сложные приложения могут иметь несколько окон, т.е. они состоят из нескольких Activity, которыми надо уметь управлять и которые могут взаимодействовать между собой.

Activity, которая запускается первой, считается главной. Из Activity можно запустить другие Activity.

Activity - это класс и у него есть некоторые методы, которые вызываются во время его жизни. Каждый Activity имеет определённый жизненный цикл:



Когда запускается Activity вызывается метод `.onCreate()`, потом `.onStart()` и потом `.onResume()`, затем приложение показывается на экране. Соответственно, ActivityRunning на картинке означает, что Activity на экране и с ним взаимодействует пользователь. Все статические действия(создание каких-нибудь классов, переменных) обычно делают в `.onCreate()`. Сбор окна приложения - создание элементов или загрузка Layout - обычно происходит в `.onStart()`.

`.onPause` - значит, что наше окно перекрылось каким-то другим, то есть оно есть, но не активно и пользователь с ним не взаимодействует. После того, как ему снова вернулась активность - снова вызывается `.onResume`, таким образом, `.onResume` парное к `onPause` - если что-то надо выключить, когда приложение становится неактивным, то включать его надо в `.onResume()`. `onStope()` - парный к `onStart()` - надо освободить ресурсы. И `onDestroy` - к `onCreate`.

Теперь, давайте посмотрим, как Activity прописывать в манифесте. У нас будут только одно окно - активности, а именно - главное. Таким образом, манифест сейчас выглядит так:

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3      package="edu.spbau.android.forecast" >
4
5      <application
6          android:allowBackup="true"
7          android:icon="@mipmap/ic_launcher"
8          android:label="@string/app_name"
9          android:theme="@style/AppTheme" >
10         <activity
11             android:name=".MainActivity"
12             android:label="@string/app_name" >
13             <intent-filter>
14                 <action android:name="android.intent.action.MAIN" />
15
16                 <category android:name="android.intent.category.LAUNCHER" />
17             </intent-filter>
18         </activity>
19     </application>
20 </manifest>
  
```

Прописывая Activity, мы также прописывает intent, на который он реагирует. Intent - класс, позволяющий частям приложения общаться между собой(или между приложениями). В нашем случае, когда пользователь нажимает на иконку приложения в меню, приложению посылается Intent MAIN. Поэтому, мы говорим, что наше Activity реагирует на intent Main, а категория intent-а говорит о том, что Activity надо запустить.

Разберём по частям наше Activity.

```

1  public class MainActivity extends ActionBarActivity {
2
3      private LocationManager locationManager;
4      private PendingIntent locationManagerPendingIntent;
5
6      @Override
7      protected void onCreate(Bundle savedInstanceState) {
8          ...
9      }
10
11     @Override
12     protected void onResume() {
13         ...
14     }
15
16     @Override
17     protected void onPause() {
18         ...
19     }
20
21
22     @Override
23     public boolean onCreateOptionsMenu(Menu menu) {
24         ...
25     }
26
27     @Override
28     public boolean onOptionsItemSelected(MenuItem item) {
29         ...
30     }
31
32 }

```

mLocationManager - поле, которое будет хранить в себе специальный класс, с помощью которого мы будем узнавать о нашем местоположении.

mLocationChangedIntent - это поле класса PendingIntent, который делает почти то же, что и Intent, с помощью него мы потом запустим другую часть нашего приложения. Единственное отличие PendingIntent от Intent - это то, что когда PendingIntent запускает какую-нибудь сущность, то она запускается с правами исходного приложения.

Мы наследуемся от ActionBarActivity - которая добавляет полосочку(ActionBar) наверху экрана. Там есть меню, которое раскрывается, а onOptionsItemSelected обрабатывает нажатие на какой-то пункт. onCreateOptionsMenu - вызывается при создании ActionBar. Причём, если он вернёт false, то меню не покажется, а если true - то отрисовывается. onOptionsItemSelected должно возвращать true, если у него получилось обработать событие.

Реализуем их:

```

1  @Override
2  public boolean onCreateOptionsMenu(Menu menu) {
3      getMenuInflater().inflate(R.menu.menu_main, menu);
4      return true;
5  }
6
7  @Override
8  public boolean onOptionsItemSelected(MenuItem item) {
9      int id = item.getItemId();
10
11     if (id == R.id.action_settings) {
12         return true;
13     }
14

```

```

15     return super.onOptionsItemSelected(item);
16 }

```

Здесь мы в функции onCreateOptionsMenu говорим, что Menu будет рисоваться так, как мы это описали в layout(это xml файлы, описывающие разметку), а в onOptionsItemSelected - в if обрабатывается действие нажатия на конкретный пункт меню. В данном случае, мы ничего не делаем, просто говорим true - обработали. Можно передать обработку нажатия на пункт меню нашему предку, что мы и делаем во всех остальных случаях.

Все layouts для Activity лежат в директории main/res/layout. А layout для menu лежит в main/res/menu. Наш файл xml называется menu_main.xml(собственно, по этому имени и происходит загрузка этого layout как разметки для меню). В файлике:

```

1  <menu
2      xmlns:android="http://schemas.android.com/apk/res/android"
3      xmlns:app="http://schemas.android.com/apk/res-auto"
4      xmlns:tools="http://schemas.android.com/tools"
5      tools:context=".MainActivity">
6
7      <item
8          android:id="@+id/action_settings"
9          android:title="@string/action_settings"
10         android:orderInCategory="100"
11         app:showAsAction="never" />
12
13 </menu>

```

У нас ровно 1 пункт в меню, описывающийся в теге item. Мы настроили для него title(строка, которая отображается в меню), его id - по нему мы находим этот item в onOptionsItemSelected, и другие.

В андроиде принято все константы хранить в директории res/values. И использовать переменные, им соответствующие. Там же(в res/values) есть специальное место для хранения строковых литералов, называется файл strings.xml. Там мы храним строковые литералы. Вот наш файл:

```

1  <resources>
2      <string name="app_name">Forecast</string>
3      <string name="action_settings">Settings</string>
4  </resources>

```

Как видно, мы храним строковый литерал "Settings" в переменной action_settings. И для описания item в нашем layout для меню мы используем переменную, хранящую этот литерал.

Идём дальше. Рассмотрим метод onCreate() нашей Activity:

```

1  @Override
2  protected void onCreate(Bundle savedInstanceState) {
3      super.onCreate(savedInstanceState);
4      setContentView(R.layout.activity_main);
5      if (savedInstanceState == null) {
6          getSupportFragmentManager().beginTransaction()
7              .add(R.id.container, new ForecastFragment())
8              .commit();
9      }
10
11     locationManager = (LocationManager) getSystemService(Context.LOCATION_SERVICE)
12         ;
13     Intent intent = new Intent(this, LocationChangedReceiver.class);
14     locationManager.registerReceiver(new LocationChangedReceiver(),
15         PendingIntent.getBroadcast(this, 0, intent, PendingIntent.FLAG_UPDATE_CURRENT));
16 }

```

Класс Bundle создаётся для сохранения важной информации(например, пользовательского UI), в onCreate его можно восстановить. А onSaveInstanceState(Bundle savedInstanceState) можно переопре-

делить и добавит в Bundle ещё что-то своё. Вызывается `onSaveInstanceState` перед `onPause()`. Но нам сейчас это не особенно важно - важно, что обязательно надо вызвать тот же метод у супер-класса.

`setContentView` - устанавливает разметку Activity, мы загружаем её из xml файла в директории `layout`, как говорилось ранее - в `layout` хранятся разметки для Activity.

В `mLocationManager` мы получаем `LocationManager`.

Далее создаём `Intent`, на который будет реагировать только класс `LocationChangedReceiver`, а затем посылаем его. `PendingIntent.getBroadcast` преобразует `Intent` в `PendingIntent`, а затем посылает его всем. Но мы задали, что реагировать на него будет только `LocationChangedReceiver`, поэтому он придёт только туда. Мы используем метод `.getBroadcast`, так как получатель - класс `BroadcastReceiver` (мы рассмотрим его дальше). Есть также методы `.getActivity`, `.getService`.

Впоследствии мы будем использовать этот `PendingIntent`. `FLAG_UPDATE_CURRENT` нужен для того, чтобы мы могли извещать `PendingIntent` об изменениях в нашем местоположении. Когда будем записывать наше местоположение в `PendingIntent`, он будет заново рассылаться. Проще говоря, флаг говорит о том, что когда кто-нибудь изменяет данные `PendingIntent`, он отправляется всем очередной раз.

Теперь рассмотрим:

```
1  @Override
2  protected void onResume() {
3      super.onResume();
4
5      final int FIVE_MINUTES = 5 * 60 * 1000;
6
7      locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER,
8          FIVE_MINUTES, 0, mLocationChangedIntent);
9  }
```

Заметим, что во всех методах жизненного цикла необходимо вызвать тот же у предка. Так было с `onCreate`, `onResume`, так будет и с остальными. Наш метод `onResume` довольно прост - он просит `LocationManager` дёргать наш `PendingIntent` раз в 5 минут и класть туда информацию о местоположении, первый параметр говорит какое устройство нам передаёт информацию, а третий - минимальная разница в метрах, при которых мы считаем произошёл update. В данном случае - мы будем дёргать `Intent` всегда.

`OnPause` метод у нас будет тоже довольно простой:

```
1  @Override
2  protected void onPause() {
3      super.onPause();
4      locationManager.removeUpdates(mLocationChangedIntent);
5  }
```

Мы просто вызываем тот же метод у предка (как и для любого метода из жизненного цикла), а потом останавливаем оповещение `PendingIntent` об изменении в текущем положении - при методе `onPause`, Activity не активно, часто не видно, поэтому нам не надо обновлять информацию.

Вот и всё. Мы не будем переопределять `onStop` и `onDestroy`, так как в данном случае это не требуется.

3 BroadcastReceiver

Ещё помните, куда мы там посылали наш `PendingIntent`? В класс `BroadcastReceiver`! Вот как раз о нём мы сейчас и поговорим.

`BroadcastReceiver` - класс-обработчик широковещательных сообщений (intent). Может быть подписан на несколько разных интенгов. Когда приходит интенг, `BroadcastReceiver`-ы, подписанные на него, выстраиваются в цепочку в соответствии с приоритетами. И если один из `BroadcastReceiver`-ов обрабатывает сообщение слишком долго, то другие вынуждены ждать. Поэтому Android прерывает исполнение `BroadcastReceiver`-ов, которые работают больше определённого времени. Из-за этого количество

действий, которое вы можете сделать с помощью BroadcastReceiver, довольно ограничено(обычное использование - послать другой интент, чтобы запустить Activity, Service или что-нибудь подобное).

Другое назначение BroadcastReceiver это получать системные оповещения, такие как оповещения о маленьком заряде батарейки или о том, что Android загрузился.

BroadcastReceiver регистрируется в Android Manifest'е так:

```
1 <receiver android:name=".LocationChangedReceiver" />
```

Таким образом, наш Manifest-файл теперь будет такой:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3     package="edu.spbau.android.forecast" >
4
5     <application
6         android:allowBackup="true"
7         android:icon="@mipmap/ic_launcher"
8         android:label="@string/app_name"
9         android:theme="@style/AppTheme" >
10         <activity
11             android:name=".MainActivity"
12             android:label="@string/app_name" >
13             <intent-filter>
14                 <action android:name="android.intent.action.MAIN" />
15
16                 <category android:name="android.intent.category.LAUNCHER" />
17             </intent-filter>
18         </activity>
19
20         <receiver android:name=".LocationChangedReceiver" />
21     </application>
22
23 </manifest>
```

Как понятно из Manifest-файла, имя нашего BroadcastReceiver будет LocationChangedReceiver. Указать, на какие intent'ы подписан BroadcastReceiver, можно двумя способами:

- В Android Manifest (обычно при подписке на системные события)
- Программно:

```
1 mLocationManager = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
2 Intent intent = new Intent(this, LocationChangedReceiver.class);
3 mLocationChangedIntent =
4 PendingIntent.getBroadcast(this, 0, intent, PendingIntent.FLAG_UPDATE_CURRENT);
```

- как мы и делали.

Посмотрим на наш BroadcastReceiver:

```
1 public class LocationChangedReceiver extends BroadcastReceiver {
2
3     private static final String TAG = LocationChangedReceiver.class.getSimpleName()
4     ;
5
6     public LocationChangedReceiver() { }
7
8     @Override
9     public void onReceive(Context context, Intent intent) {
10         final String locationKey = LocationManager.KEY_LOCATION_CHANGED;
11
12         if (intent.hasExtra(locationKey)) {
13             Location location = (Location) intent.getExtras().get(locationKey);
```

```

13
14         Intent startService = new Intent(context, ForecastUpdateService.class);
15         startService.putExtra(LocationManager.KEY_LOCATION_CHANGED, location);
16         context.startService(startService);
17     }
18 }
19 }

```

Теперь пришло время поговорить об Intent-ах чуть подробнее. Все Intent могут хранить в себе информацию. Добавлять её можно вручную - у Intent есть специальные методы `.putExtra(String key, ...)` - где второй параметр это то, что мы хотим добавить. Эти методы определены для основных классов Java. Полный список методов можно найти тут: [тык](#)

И, если кто-то реагирует на Intent с данными и начинает его обрабатывать, то мы можем из Intent вытащить дополнительную информацию.

Когда приходит интент, который предназначен для данного BroadcastReceiver, вызывается метод `onReceive()`.

Одним из его аргументов является экземпляр класса Context - базовый класс для частей приложения. Через него можно обращаться к ресурсам Android. К примеру, Activity - наследник класса Context. Поэтому, мы можем вызвать, например, метод `getSystemService()`, определённый в классе Context - для доступа к какому-то сервису. А вот BroadcastReceiver не является наследником класса Context, но доступ к различным менеджерам может понадобиться, поэтому контекст и передаётся методу `onReceive()`. Наследниками Context также являются разнообразные Service и Activity классы (например, IntentService или AliasActivity).

Вторым аргументом является Intent, на который он среагировал. Мы берём этот интент и вытаскиваем из него информацию, которую положил туда LocationManager. Известно, что LocationManager при вызванном нами методе кладёт информацию с ключом `LocationManager.KEY_LOCATION_CHANGED` (это можно найти в документации), поэтому мы и достаём оттуда значение по этому ключу. `.getExtras` возвращает Bundle - специальный класс для хранения данных вида ключ(строка)-значение(Object). Так что `.get()` возвращает Object, его надо прикастовать к Location - класс, в котором содержится информация о нашем местоположении.

Дальше, в силу скромных возможностей BroadcastReceiver мы перенаправляем запрос классу Service. Про этот класс мы поговорим дальше.

Чтобы передать управление классу Service, мы также используем Intent. Здесь мы используем обычный Intent, и всё работает. Но почему мы не могли создать обычный интент при обработке LocationManager-ом? Всё дело в том, что LocationManager принимает только PendingIntent, поэтому нам пришлось идти длинным путём.

Итак, мы создали интент ручками и указали класс, который на него подписан - ForecastUpdateService. Дальше положили полученную информацию уже в новый интент и отправили сервису. Это можно сделать с помощью команды `startService`. А если бы мы отправляли интент Activity, то метод назывался бы `startActivity` - осторожно!

4 Log

Наверняка вы заметили странное статическое поле в BroadcastReceiver:

```
1 private static final String TAG = LocationChangedReceiver.class.getSimpleName();
```

Оно используется для логирования.

- Записи в log выводятся на экран Device Monitor.Logcat.
- Логирование происходит так:


```

1    ...
2    private static final String TAG =
3        SMSReceiver.class.getSimpleName();
4    ...
5    Log.d(TAG, "SMS received");

```

В нашем случае мы ничего не логируем, но в целом логирование - хороший тон.

5 Service

Теперь, наконец, поговорим про Service.

- Service также надо прописывать в манифесте:

```

1    <service
2        android:name=".DatabaseService"
3        android:exported="false" >
4    </service>

```

Таким образом, в нашем манифест-файле теперь появилась новая сущность:

```

1    <?xml version="1.0" encoding="utf-8"?>
2    <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3        package="edu.spbau.android.forecast" >
4
5        <application
6            android:allowBackup="true"
7            android:icon="@mipmap/ic_launcher"
8            android:label="@string/app_name"
9            android:theme="@style/AppTheme" >
10           <activity
11               android:name=".MainActivity"
12               android:label="@string/app_name" >
13               <intent-filter>
14                   <action android:name="android.intent.action.MAIN" />
15
16                   <category android:name="android.intent.category.LAUNCHER" />
17               </intent-filter>
18           </activity>
19
20           <receiver android:name=".LocationChangedReceiver" />
21
22           <service
23               android:name=".ForecastUpdateService"
24               android:exported="false" />
25       </application>
26
27    </manifest>

```

Ключ exported задаёт, позволено ли другим приложениям запускать наш сервис, Activity и тд. В нашем случае - не позволено.

- Если intent занят какой-то другой сущностью, то Service также становится в очередь.
- При обработке intent запускается метод onHandleIntent(Intent intent);

Давайте разберём, что мы здесь делаем:


```

1 public class ForecastUpdateService extends IntentService {
2
3     private final static String TAG = ForecastUpdateService.class.getSimpleName();
4
5     public ForecastUpdateService() {
6         super(TAG);
7     }
8
9     private void parseJsonData(String json) {
10         final String JSON_LIST = "list";
11         final String JSON_SPEED = "speed";
12         final String JSON_DEGREE = "deg";
13         final String JSON_TEMPERATURE = "temp";
14         final String JSON_DAY_TEMPERATURE = "day";
15         final String JSON_NIGHT_TEMPERATURE = "night";
16         final String JSON_WEATHER = "weather";
17         final String JSON_WEATHER_ID = "id";
18
19         try {
20             JSONObject forecastJson = new JSONObject(json);
21             JSONArray weatherArray = forecastJson.getJSONArray(JSON_LIST);
22
23             ArrayList<ContentValues> forecast = new ArrayList<>(weatherArray.length());
24
25             Time dayTime = new Time();
26             dayTime.setToNow();
27             int julianStartDay = Time.getJulianDay(System.currentTimeMillis(),
28                 dayTime.gmtoff);
29             dayTime = new Time();
30
31             for(int i = 0; i < weatherArray.length(); i++) {
32                 JSONObject dayForecast = weatherArray.getJSONObject(i);
33
34                 long dateTime = dayTime.setJulianDay(julianStartDay + i);
35
36                 double speed = dayForecast.getDouble(JSON_SPEED);
37                 double degree = dayForecast.getDouble(JSON_DEGREE);
38
39                 JSONObject temperatureObject = dayForecast.getJSONObject(
40                     JSON_TEMPERATURE);
41                 double day = temperatureObject.getDouble(JSON_DAY_TEMPERATURE);
42                 double night = temperatureObject.getDouble(JSON_NIGHT_TEMPERATURE);
43
44                 JSONArray weatherObjects = dayForecast.getJSONArray(JSON_WEATHER);
45                 JSONObject weatherObject = weatherObjects.getJSONObject(0);
46                 int weather = weatherObject.getInt(JSON_WEATHER_ID);
47
48                 ContentValues weatherValues = new ContentValues();
49                 weatherValues.put(WeatherContract.WeatherEntry.COLUMN_DATE,
50                     dateTime);
51                 weatherValues.put(WeatherContract.WeatherEntry.COLUMN_DEGREES,
52                     degree);
53                 weatherValues.put(WeatherContract.WeatherEntry.COLUMN_WIND_SPEED,
54                     speed);
55                 weatherValues.put(WeatherContract.WeatherEntry.COLUMN_NIGHT_TEMP,
56                     night);
57                 weatherValues.put(WeatherContract.WeatherEntry.COLUMN_DAY_TEMP, day);
58                 weatherValues.put(WeatherContract.WeatherEntry.COLUMN_WEATHER_ID,
59                     weather);
60             }
61         } catch (JSONException e) {
62             // Handle exception
63         }
64     }
65 }

```

```

53         forecast.add(weatherValues);
54     }
55
56     if (forecast.size() > 0) {
57         ContentValues values[] = new ContentValues[forecast.size()];
58         forecast.toArray(values);
59         getResolver().bulkInsert(WeatherContract.WeatherEntry.
            CONTENT_URI, values);
60     }
61 } catch (JSONException e) {
62     Log.e(TAG, "Error while parsing JSON", e);
63 }
64 }
65
66 @Override
67 protected void onHandleIntent(Intent intent) {
68     final String locationKey = LocationManager.KEY_LOCATION_CHANGED;
69
70     final String FORECAST_BASE_URL =
71         "http://api.openweathermap.org/data/2.5/forecast/daily?";
72     final String LATITUDE_PARAM = "lat";
73     final String LONGITUDE_PARAM = "lon";
74     final String MODE_PARAM = "mode";
75     final String JSON = "json";
76     final String UNITS_PARAM = "units";
77     final String METRIC = "metric";
78
79     if (!intent.hasExtra(locationKey)) {
80         Log.w(TAG, "Received intent without location");
81         return;
82     }
83
84     Location location = (Location) intent.getExtras().get(locationKey);
85     HttpURLConnection connection = null;
86     BufferedReader reader = null;
87
88     try {
89         Uri uri = Uri.parse(FORECAST_BASE_URL).buildUpon()
90             .appendQueryParameter(LATITUDE_PARAM, Double.toString(location.
91                 getLatitude()))
92             .appendQueryParameter(LONGITUDE_PARAM, Double.toString(location
93                 .getLongitude()))
94             .appendQueryParameter(MODE_PARAM, JSON)
95             .appendQueryParameter(UNITS_PARAM, METRIC).build();
96         URL url = new URL(uri.toString());
97         connection = (HttpURLConnection) url.openConnection();
98         connection.setRequestMethod("GET");
99         connection.connect();
100
101         InputStream inputStream = connection.getInputStream();
102         StringBuilder buffer = new StringBuilder();
103         if (inputStream == null) {
104             return;
105         }
106         reader = new BufferedReader(new InputStreamReader(inputStream));
107         String line;
108         while ((line = reader.readLine()) != null) {
109             buffer.append(line);
110             buffer.append("\n");
111         }

```

```

111         if (buffer.length() == 0) {
112             return;
113         }
114
115         parseJsonData(buffer.toString());
116     } catch (IOException e) {
117         Log.e(TAG, "Error", e);
118     } finally {
119         if (reader != null) {
120             try {
121                 reader.close();
122             } catch (IOException e) {
123                 Log.e(TAG, "Error while closing stream", e);
124             }
125         }
126         if (connection != null) {
127             connection.disconnect();
128         }
129     }
130 }
131
132 }

```

Выглядит страшно, но скоро всё будет понятно. Начнём сначала, вот Intent начинает обрабатывать функция `onHandleIntent`. Первым делом мы задаём некоторые значения, которые впоследствии могут нам понадобиться - первое для извлечения локации из Intent. Остальные - специальные параметры для декодирования и кодирования определённых форматов.

Дальше мы проверяем, что пришедший интент действительно содержит нужную информацию (функция `hasExtra(String key)`), и если нет - используем логирование, чтобы сообщить об ошибке и выходим. Мы уже умные, мы знаем, что это такое.

Если информация в Intent действительно содержится, мы извлекаем информацию в `location`. Дальше нам нужно подключиться по интернету. Но для этого нужны некоторые права.

6 Permissions

- Чтобы работать с сетью надо подключить соответствующие permission.

```

1 <uses-permission android:name="android.permission.INTERNET" />
2 <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>

```

Таким образом, манифест-файл выглядит сейчас так:

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3     package="edu.spbau.android.forecast" >
4
5     <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
6     <uses-permission android:name="android.permission.INTERNET" />
7     <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
8
9     <application
10         android:allowBackup="true"
11         android:icon="@mipmap/ic_launcher"
12         android:label="@string/app_name"
13         android:theme="@style/AppTheme" >
14         <activity
15             android:name=".MainActivity"
16             android:label="@string/app_name" >
17             <intent-filter>

```

```

18         <action android:name="android.intent.action.MAIN" />
19
20         <category android:name="android.intent.category.LAUNCHER" />
21     </intent-filter>
22 </activity>
23
24 <receiver android:name=".LocationChangedReceiver" />
25
26 <service
27     android:name=".ForecastUpdateService"
28     android:exported="false" />
29 </application>
30 </manifest>

```

- Для получения данных относительно точной локации, также нужно разрешение - оно первое в списке.
- Второе позволяет открывать сетевые сокет.
- А третье позволяет приложениям получать доступ к информации о сетях.

7 Connection

Продолжим разбираться с нашим Service. Мы остановились на том, что мы хотим выйти в сеть.

- Самый простой способ работать с сетью - это `URLConnection`. Сначала мы долго-долго строим URL-адрес, в котором находится наш запрос - мы достаём из `Location` широту и долготу, а потом получаем `URLConnection` по этому URL. Возвращается методом `.openConnection()`:

```

1 URL url = new URL(uri.toString());
2 connection =
3     (URLConnection) url.openConnection();

```

Здесь нужен каст, так как `.openConnection` на самом деле возвращает `URLConnection`, но по факту это - `URLConnection`, так как наш сайт работает по протоколу `http`.

- Протокол `http` может обрабатывать различные запросы. Например, запрос на получение информации. Если мы хотим получить информацию - пишем следующее:

```

1 connection.setRequestMethod("GET");

```

А мы действительно хотим получить информацию по запросу, который мы отправили вместе с URL. Затем мы собственно, создаём соединение - функцией `connection.connect()`;

Несложно, да? Мы создали соединение на основе `connection`. Осталось оттуда прочесть данные. Считываем информацию мы так:

```

1 InputStream inputStream = connection.getInputStream();
2 StringBuilder buffer = new StringBuilder();
3 if (inputStream == null) {
4     return;
5 }
6 reader = new BufferedReader(new InputStreamReader(inputStream));
7 String line;
8 while ((line = reader.readLine()) != null) {
9     buffer.append(line);
10    buffer.append("\n");
11 }
12

```

```

13 if (buffer.length() == 0) {
14     return;
15 }

```

А теперь пояснение. Мы берём InputStream, связанный с нашим соединением. Прочитаем всё в StringBuilder - мы будем читать строчками, поэтому нам надо сконкатенировать. StringBuilder здесь лучший выбор. Чтобы удобнее было работать с потоком чтения, мы оборачиваем его в BufferedReader - и читаем построчно, пока не закончатся строчки. Если ничего не пришло, то длина buffer окажется равной нулю - и мы выходим.

Отлично, теперь у нас есть StringBuilder из строчек результата. Ответ надо распарсить.

Обычно ответ формируется либо в формате xml, либо в формате json - это два разных представления данных. В данном случае, мы используем json. Нужно этот формат распарсить и обработать - для этого мы написали функцию parseJsonData(String), но подробно на ней останавливаться не будем. Для этого пришлось бы углубиться ещё и в структуру формата json и API сайта. После этого, в onHandleIntent мы обрабатываем все ошибки, а в блоке finally закрывает открытые ресурсы.

Что важно не пропустить в parseJsonData - последний if - если данные о погоде есть, то мы их куда-то вставляем. В будущем мы поймём, что куда-то - это базы данных.

8 Data bases

Каждый раз при смене своих координат не очень разумно слать запрос в интернет. Это долго, это не надёжно - интернет или сервис могут не работать. Поэтому стоит задуматься о том, как приложение будет работать в offline-режиме. К примеру, разумный вариант - когда у нас есть доступ к сети, получать погоду и сохранять её в базу данных. А когда сети нет - просто достаём из базы данных самое актуальное значение и показываем. Оно, может быть, и устаревшее, но хоть какое-то. Соответственно, следующая наша цель - получить результат и сохранить его в базу данных.

В нашем проекте есть такой класс WeatherProvider - наследник класса ContentProvider.

- ContentProvider - это такой класс-обёртка над базой данных.
- Его также надо прописывать в манифесте:

```

1 <provider
2     android:authorities="edu.spbau.android.forecast"
3     android:name=".WeatherProvider" />

```

android:authorities - обязательно должен быть и должен указывать путь к провайдеру(проще - пакет провайдера).

Таким образом, манифест получается такой:

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3     package="edu.spbau.android.forecast" >
4
5     <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
6     <uses-permission android:name="android.permission.INTERNET" />
7     <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
8
9     <application
10         android:allowBackup="true"
11         android:icon="@mipmap/ic_launcher"
12         android:label="@string/app_name"
13         android:theme="@style/AppTheme" >
14         <activity
15             android:name=".MainActivity"
16             android:label="@string/app_name" >

```

```

17         <intent-filter>
18             <action android:name="android.intent.action.MAIN" />
19
20             <category android:name="android.intent.category.LAUNCHER" />
21         </intent-filter>
22     </activity>
23
24     <receiver android:name=".LocationChangedReceiver" />
25
26     <service
27         android:name=".ForecastUpdateService"
28         android:exported="false" />
29
30     <provider
31         android:authorities="edu.spbau.android.forecast"
32         android:name=".WeatherProvider" />
33 </application>
34
35 </manifest>

```

Это - окончательный вариант и больше он меняться не будет.

9 Contract

Перд тем, как подробнее говорить о ContentProvider, познакомимся с ещё одним классом Contract, реализация которого является правилом хорошего тона. Там обычно описывается структура базы данных. То есть, он необязателен, чтобы работать с базами данных, но лучше его реализовывать.

Соответственно:

- Его не надо прописывать в манифесте, так как это вспомогательный класс.
- Там как константы хранятся названия таблиц, названия колонок, строк и другая подобная информация.
- Для каждой таблички внутри контракта заводится класс-наследник интерфейса BaseColumns. И в нём уже описывается вся информация о конкретной табличке.
- В нашей базе данных будет только одна табличка, в которой будет храниться прогноз погоды по времени.

Наш Contract выглядит так:

```

1  public class WeatherContract {
2
3      public static final String CONTENT_AUTHORITY = "edu.spbau.android.forecast";
4      public static final Uri BASE_CONTENT_URI = Uri.parse("content://" +
5          CONTENT_AUTHORITY);
6      public static final String PATH_WEATHER = "weather";
7
8      public static long normalizeDate(long gmt) {
9          Time time = new Time();
10         time.set(gmt);
11         int day = Time.getJulianDay(gmt, time.gmtoff);
12         return time.setJulianDay(day);
13     }
14
15     public static final class WeatherEntry implements BaseColumns {
16
17         public static final Uri CONTENT_URI =

```

```

17         BASE_CONTENT_URI.buildUpon().appendPath(PATH_WEATHER).build();
18
19     public static final String CONTENT_TYPE =
20         ContentResolver.CURSOR_DIR_BASE_TYPE + "/" + CONTENT_AUTHORITY + "/"
21         + PATH_WEATHER;
22
23     public static final String CONTENT_ITEM_TYPE =
24         ContentResolver.CURSOR_ITEM_BASE_TYPE + "/" + CONTENT_AUTHORITY + "/"
25         + PATH_WEATHER;
26
27     public static final String TABLE_NAME = "weather";
28
29     public static final String COLUMN_DATE = "date";
30     public static final String COLUMN_DAY_TEMP = "day_temp";
31     public static final String COLUMN_NIGHT_TEMP = "nigh_temp";
32     public static final String COLUMN_WIND_SPEED = "wind";
33     public static final String COLUMN_DEGREES = "direction";
34     public static final String COLUMN_WEATHER_ID = "weather_id";
35
36     public static long getDateFromUri(Uri uri) {
37         return Long.parseLong(uri.getPathSegments().get(1));
38     }
39
40     public static Uri buildWeatherUri(long date) {
41         return ContentUris.withAppendedId(CONTENT_URI, date);
42     }
43
44 }

```

В классе WeatherContract:

- CONTENT_AUTHORITY - путь к ContentProvider, иначе - пакет ContentProvider(в нашем случае - WeatherProvider)
- BASE_CONTENT_URI - URI по пути к ContentProvider
- PATH_WEATHER - локальный путь к табличке weather. Чаще всего, просто название таблички. А так из локального пути к табличке и URI ContentProvider можно построить полный путь до таблички:

```

1     public static final Uri CONTENT_URI =
2         BASE_CONTENT_URI.buildUpon().appendPath(PATH_WEATHER).build();

```

- public static long normalizeDate(long gmt) - как понятно из названия, нормализует время.
- Подкласс, соответствующий нашей единственной табличке. В нём:
 - public static final Uri CONTENT_URI – Uri, по которому мы будем обращаться к конкретной табличке ContentProvider, так как доступ к ContentProvider осуществляется только через Uri. В данном случае - табличка, называемая weather.
 - public static final String CONTENT_TYPE – тип набора записей в ContentProvider.
 - public static final String CONTENT_ITEM_TYPE – тип одной записи в ContentProvider
 - А затем разные атрибуты, описывающие табличку - её название, и колонки, которые в ней есть.
 - Также, в классе прописываются разные служебные методы. У нас реализованы преобразование даты в Uri, и наоборот. Чтобы мы смогли потом обратиться к какой-то записи, соответствующей определённой дате.

Выражения для `CONTENT_TYPE` и `CONTENT_ITEM_TYPE` всегда одни и те же, но их всё равно приходится прописывать.

10 SQLiteOpenHelper

Следующий класс, про который мы поговорим перед `ContentProvider`, это абстрактный класс `SQLiteOpenHelper`.

- Это, собственно, тот класс, который работает с базой данных. Именно через этот класс, мы будем делать запросы к бд.
- Как нетрудно догадаться по названию, в Android используется SQLite. А этот класс отвечает за создание базы данных, если её нет, открытие, если есть и обновления, когда нужно. Соответственно, он имеет три метода: `onCreate`, `onOpen`, `onUpgrade`. Конструктор `SQLiteOpenHelper` создаёт объект, помогающий в управлении базой данных.
- Нашего класса наследника мы назовём `WeatherDBHelper`. При первом запуске у `SQLiteOpenHelper` вызывается метод `public void onCreate(SQLiteDatabase db)`. С помощью класса `SQLiteDatabase` можно создать таблички базы данных со своей структурой. Надо вызвать метод `.execute(String)` у `db`, где в качестве строки используется обычный запрос на языке SQLite.

Рассмотрим наш код. Итак, мы переопределим два метода - `onCreate` и `onUpgrade`. Третий мы использовать не будем:

```
1 public class WeatherDBHelper extends SQLiteOpenHelper {
2
3     private static final int DATABASE_VERSION = 1;
4     private static final String DATABASE_NAME = "weather.db";
5
6     public WeatherDBHelper(Context context) {
7         super(context, DATABASE_NAME, null, DATABASE_VERSION);
8     }
9
10    @Override
11    public void onCreate(SQLiteDatabase db) {
12        ...
13    }
14
15    @Override
16    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
17        ...
18    }
19
20 }
```

Наш класс будет хранить версию и название базы данных, первое представляет из себя число, а второе - строку. Мы назовём нашу базу данных "weather.db". В конструкторе класса обязательно надо вызвать конструктор родительского(ну или написать свой, который создаёт этот дополнительный объект).

Теперь опишем класс `onCreate`:

```
1 @Override
2 public void onCreate(SQLiteDatabase db) {
3     String SQL_CREATE_WEATHER_TABLE = "CREATE TABLE " + WeatherEntry.
4         TABLE_NAME + " (" +
5         WeatherEntry._ID + " INTEGER PRIMARY KEY AUTOINCREMENT," +
6         WeatherEntry.COLUMN_DATE + " INTEGER NOT NULL," +
7         WeatherEntry.COLUMN_DAY_TEMP + " REAL NOT NULL," +
```

```

7         WeatherEntry.COLUMN_NIGHT_TEMP + " REAL NOT NULL," +
8         WeatherEntry.COLUMN_WIND_SPEED + " REAL NOT NULL," +
9         WeatherEntry.COLUMN_DEGREES + " REAL NOT NULL," +
10        WeatherEntry.COLUMN_WEATHER_ID + " INTEGER NOT NULL, " +
11        "UNIQUE (" + WeatherEntry.COLUMN_DATE + ") ON CONFLICT REPLACE)
12        ";
13        db.execSQL(SQL_CREATE_WEATHER_TABLE);
14    }

```

Как мы уже говорили, в методе onCreate мы задаём структуру базы данных. Здесь ничего - просто создание SQL-запроса, если вы знакомы с базами данных, то для вас здесь не будет ничего сложного. Надо заметить, что название колонок берётся из класса WeatherEntry - это вложенный класс в WeatherContract, который отвечает за нашу единственную табличку. В остальном мы просто задаём данные какого вида будут храниться в колонках.

Внутри каждого запроса к базе данных есть также поле версии, в которой описано, какой версии должна соответствовать база данных. И, если текущая версия базы данных не соответствует той, которую ожидает приложение, будет вызван метод onUpgrade:

```

1  @Override
2  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
3      db.execSQL("DROP TABLE IF EXISTS " + WeatherEntry.TABLE_NAME);
4      onCreate(db);
5  }

```

oldVersion - текущая версия бд, newVersion - версия, требуемая приложением. Итак, если версии не совпадают, а значит, надо нашу базу данных обновить. Самый простой способ это сделать - удалить всё и создать заново, что мы, собственно, и делаем - удаляем табличку, если она существует, а затем вызываем onCreate.

Надо учитывать, что при таком подходе все данные теряются, поэтому если нам хочется сохранить какие-то данные, это нужно как-то обработать, но в нашем случае, нам это не важно.

11 И снова про ContentProvider

Теперь, наконец, разберём поподробнее класс ContentProvider: Мы создадим класс-наследник от ContentProvider - WeatherProvider. Он будет выглядеть так:

```

1  public class WeatherProvider extends ContentProvider {
2
3      private static final int WEATHER = 100;
4      private static final int WEATHER_WITH_DATE = 101;
5
6      private static final UriMatcher sUriMatcher = buildUriMatcher();
7
8      private static UriMatcher buildUriMatcher() {
9      }
10
11     private static void normalizeDate(ContentValues values) {
12     }
13
14     private WeatherDBHelper mDbHelper;
15
16     private Cursor getWeather(String[] projection, String selection, String[]
17         selectionArgs,
18         String sortOrder)
19     {

```

```

20
21     private Cursor getWetherByDate(Uri uri, String[] projection, String sortOrder)
22     {
23
24         @Override
25         public boolean onCreate() {
26         }
27
28         @Override
29         public Cursor query(Uri uri, String[] projection, String selection, String[]
30             selectionArgs, String sortOrder) {
31         }
32
33         @Override
34         public String getType(Uri uri) {
35         }
36
37         @Override
38         public Uri insert(Uri uri, ContentValues values) {
39         }
40
41         @Override
42         public int delete(Uri uri, String selection, String[] selectionArgs) {
43         }
44
45         @Override
46         public int update(Uri uri, ContentValues values, String selection, String[]
47             selectionArgs) {
48         }
49
50         @Override
51         public int bulkInsert(Uri uri, @NonNull ContentValues[] values) {
52         }
53     }

```

Поля WEATHER и WEATHER_WITH_DATE - это специальные коды, которые описывают какой запрос к нам пришёл. В методе buildUriMatcher - создаётся специальный UriMatcher, который запросу к какой-то конкретной строчке ставит в соответствие код WEATHER_WITH_DATE, а запросам к группе строчек - WEATHER. Этот UriMatcher сохраняется в поле sUriMatcher. Подробнее про этот метод мы поговорим дальше.

У ContentProvider можно переопределить только метод onCreate - так как именно там обычно происходит подключение к базе данных, но и этот метод переопределять необязательно. Однако нам этого недостаточно.

Рассмотрим методы нашего класса поближе. Во-первых, метод onCreate. Этот метод вызовется автоматически - до того, как мы обратимся к классу WeatherProvider в первый раз.

В нашем случае, подключение к бд - это просто создание WeatherDBHelper. Мы сохраняем WeatherDBHelper в отдельное поле, чтобы можно было впоследствии управлять версиями(mDbHelper - поле, которое определено через три метода).

```

1  @Override
2  public boolean onCreate() {
3      mDbHelper = new WeatherDBHelper(getContext());
4      return true;
5  }

```

Теперь пробежимся по остальным методам:

- buildUriMatcher(...) Мы про него кое-что уже знаем - к примеру зачем он и что он идеологически

делает. Код его такой:

```
1 private static UriMatcher buildUriMatcher() {
2     UriMatcher matcher = new UriMatcher(UriMatcher.NO_MATCH);
3     String authority = WeatherContract.CONTENT_AUTHORITY;
4     matcher.addURI(authority, WeatherContract.PATH_WEATHER, WEATHER);
5     matcher.addURI(authority, WeatherContract.PATH_WEATHER + "/" + "#",
6         WEATHER_WITH_DATE);
7     return matcher;
}
```

Как мы видим, здесь мы сначала создаём UriMatcher, а затем добавляем два соответствия по правилу, про которое мы уже говорили и возвращаем соответствующий matcher, который сохраняется в поле sUriMatcher.

- public Cursor query(...) - вызывается при запросе к ContentProvider(читай, к бд). Здесь обрабатываются запросы типа SELECT. Вглядит он так:

```
1 @Override
2 public Cursor query(Uri uri, String[] projection, String selection, String[]
3     selectionArgs, String sortOrder) {
4     Cursor cursor;
5     switch (sUriMatcher.match(uri)) {
6         case WEATHER:
7             cursor = getWeather(projection, selection, selectionArgs, sortOrder
8                 );
9             break;
10        case WEATHER_WITH_DATE:
11            cursor = getWetherByDate(uri, projection, sortOrder);
12            break;
13        default:
14            throw new UnsupportedOperationException("Unknown uri: " + uri);
15    }
16    cursor.setNotificationUri(getContext().getContentResolver(), uri);
17    return cursor;
18 }
```

Что мы тут делаем? Во-первых, мы проверяем тип запроса с помощью sUriMatcher. Uri нам передают как параметр. Если тип Uri - запрос к конкретной записи, то мы вызываем getWeatherByDate, если же запрос ко всей табличке, то вызывается метод getWeather. Оба метода возвращают Cursor - некий указатель на выбранные значения.

Метод getWeather принимает три параметра - колонки, которые надо выбрать(projection - массив названия колонок), условие выборки(selection, например, нам надо только записи, где значение даты в каком-то интервале), конкретные значения для условия(selectionArgs, задаются значения интервала) и если выбранные строчки надо будет как-то отсортировать, то мы на этот случай передаём ещё sortOrder - порядок сортировки. Все эти параметры задаются строчками или массивами строчек - которые потом можно будет составить в запрос SQL.

Метод getWeatherByDate принимает uri - путь к соответствующей строчке, колонки, которые надо будет выбрать оттуда(projection) и sortOrder.

Соответственно, параметры, которые принимает метод query - это все параметры, которые могут понадобиться методам getWeather или getWeatherByDate.

Мы возвращаем этот курсор, который получили.

cursor.setNotificationUri - регистрирует наш Uri, чтобы потом тем, кто следит за этим Uri при изменениях в Uri пришло оповещение.

- `public Uri insert(...)` - вызывается, если мы хотим вставить что-то в `ContentProvider` (читай, нашу бд). Здесь обрабатываются запросы типа `INSERT`. Для таких запросов код у нас получится следующий:

```

1  @Override
2  public Uri insert(Uri uri, ContentValues values) {
3      switch (sUriMatcher.match(uri)) {
4          case WEATHER:
5              normalizeDate(values);
6              long date = values.getAsLong(WeatherContract.WeatherEntry.
                    COLUMN_DATE);
7              long id = mDbHelper.getWritableDatabase().insert(
5              WeatherContract.WeatherEntry.TABLE_NAME, null, values);
9              if (id != -1) {
10                 getContext().getContentResolver().notifyChange(uri, null);
11                 return WeatherContract.WeatherEntry.buildWeatherUri(date);
12             } else {
13                 throw new android.database.SQLException("Failed to insert row
                    into " + uri);
14             }
15             default:
16                 throw new UnsupportedOperationException("Unknown uri: " + uri);
17         }
18     }

```

В `insert` нам передаётся `uri` таблички, куда мы хотим вставить значение и значение в классе `ContentValues` - это такая обёртка над переменными, чтобы с ней мог работать класс `ContentResolver`. С точки зрения идеологии - это множество пар ключ-значение. И есть много методов типа `getAsLong(String)`, `getAsInteger(String)`....

Первым делом, в методе мы снова проверяем какого типа `Uri` нам дали. Нам здесь не могут дать `uri` на конкретную запись, иначе это - ошибка (как можно вставить значения в запись?), поэтому в `case` блоке только один случай.

Сначала мы приводим значение к нормальному виду функцией `normalizeDate`, что конкретно она делает, рассмотрим чуть попозже. Затем, извлекаем дату из `ContentValues` - она нам может понадобиться позже. Мы используем метод `getAsLong` - так как значение у нас типа `long`. А извлекаем мы по ключу, по которому положили. Мы говорим, что будем дату всегда класть по ключу `WeatherContract.WeatherEntry.COLUMN_DATE`, то есть по названию колонки в нашей табличке - вот, снова появился класс, вложенный в класс `Contract`.

Из `SQLiteOpenHelper` можно доставать `SQLiteDatabase`, есть два разных метода - `getReadableDatabase` - возвращает базу данных, предназначенную для чтения, и `getWritableDatabase` - возвращает базу данных предназначенную для записи. Здесь нам надо вставить значения, поэтому мы вызываем метод `getWritableDatabase`.

На самом деле, в предыдущем рассмотренном нами методе в `getWeather` и `getWeatherByDate` будет вызываться `getReadableDatabase`.

А затем делаем запрос `insert` к этой базе данных. `values` здесь описывает что и куда вставлять, то есть ключ означает название столбца, а значение - непосредственно значение, которое вставляется в этот столбец. И вот именно здесь должен передаваться именно `ContentValues`. Туда также передаётся название таблички. А второй параметр `null` говорит о том, что если передался пустой `ContentValues`, то ничего вставлять не надо.

Команда `insert` у `SQLiteDatabase` возвращает индекс строки, куда вставилось значение. Если вставить его не получилось то вернёт `-1`. Поэтому при значении `-1` мы кидаем `Exception`. Иначе, преобразовываем известные нам данные в `Uri`, чтобы вернуть его из метода. Мы используем метод `buildWeatherUri`, который специально для этого определили в `Contract`.

Остался один непонятный момент со следующей строчкой:

```
1 getContext().getContentResolver().notifyChange(uri, null);
```

Мы изменили таблицку. Но при этом кто-то может слушать это изменение, то есть кто-то, возможно, отслеживает изменение этой таблички. Чтобы уведомить о том, что табличка изменилась, надо вызвать эту строчку(она извещает того, кто следит за табличкой и того, кто следит непосредственно по uri).

- `public int delete(...)` - вызывается, если мы хотим удалить что-то из `ContentProvider`(читай, из нашей бд). Здесь обрабатываются запросы типа `DELETE`.

Наш код выглядит так:

```
1 @Override
2 public int delete(Uri uri, String selection, String[] selectionArgs) {
3     switch (sUriMatcher.match(uri)) {
4         case WEATHER:
5             int count = mdbHelper.getWritableDatabase().delete(
6                 WeatherContract.WeatherEntry.TABLE_NAME, selection,
7                 selectionArgs);
8             if (count > 0) {
9                 getContext().getContentResolver().notifyChange(uri, null);
10            }
11            return count;
12        default:
13            throw new UnsupportedOperationException("Unknown uri: " + uri);
14    }
```

Удаление может происходить только по табличке, поэтому мы проверяем, что наш `Uri` верный(что указывает он на всю табличку, а не на запись). Затем, аналогично предыдущим методам мы вызываем метод у `mdbHelper` - возвращающий `SQLiteDatabase` для изменений, и у неё вызываем метод `delete`, как мы делали с `insert`. В качестве параметров надо указать название таблички и выборку, по которой мы будем удалять - `selection`(условие) и `selectionArgs`(значения этого условия).

Метод возвращает число - количество удалённых записей. И если это число больше нуля, то табличка изменилась и об этом надо всех уведомить.

- `public int update(...)` - вызывается, если мы хотим обновить какую-то запись в `ContentProvider`(читай, в нашей бд). Здесь обрабатываются запросы типа `UPDATE`. Наш код выглядит так:

```
1 @Override
2 public int update(Uri uri, ContentValues values, String selection, String[]
3     selectionArgs) {
4     switch (sUriMatcher.match(uri)) {
5         case WEATHER:
6             int count = mdbHelper.getWritableDatabase().update(
7                 WeatherContract.WeatherEntry.TABLE_NAME, values, selection,
8                 selectionArgs);
9             if (count > 0) {
10                getContext().getContentResolver().notifyChange(uri, null);
11            }
12            return count;
13        default:
14            throw new UnsupportedOperationException("Unknown uri: " + uri);
15    }
```

Здесь мы тоже обрабатываем только `uri` указывающий на всю табличку. Аналогично предыдущим методам вызывается метод у `SQLiteDatabase`, возвращается количество изменённых, а затем если оно больше 0, то табличка изменилась и надо оповестить об этом.

- `public int bulkInsert(...)` - метод, позволяющий вставлять сразу пачку значений(запросы типа INSERT). Он выглядит так:

```

1  @Override
2      public int bulkInsert(Uri uri, @NonNull ContentValues[] values) {
3          switch (sUriMatcher.match(uri)) {
4              case WEATHER:
5                  SQLiteDatabase db = mDbHelper.getWritableDatabase();
6                  int count = 0;
7                  db.beginTransaction();
8                  try {
9                      for (ContentValues value : values) {
10                         normalizeDate(value);
11                         long id = db.insert(WeatherContract.WeatherEntry.
12                             TABLE_NAME, null, value);
13                         if (id != -1) {
14                             count++;
15                         }
16                     }
17                     db.setTransactionSuccessful();
18                 } finally {
19                     db.endTransaction();
20                 }
21                 getContext().getContentResolver().notifyChange(uri, null);
22                 return count;
23             default:
24                 throw new UnsupportedOperationException("Unknown uri: " + uri);
25         }
26     }
27 }

```

Мы ожидаем получить на вход `uri` таблички и затем значения, которые хотим вставить. Аналогично `insert-y`, если `uri` переданный нам не соответствует табличке, а соответствует записи, мы не сможем его обработать и кидаем `Exception`. Аналогично `insert-y` же мы получаем `SQLiteDatabase`, доступную на запись. В `count` будем считать количество удачно вставленных записей.

`db.beginTransaction` - переключает нас в "монопольный режим". Чтобы пока мы изменяем базу данных никто больше её не трогал. В конце после этого надо будет вызвать метод `db.endTransaction`. Если перед этим методом не вызвать `db.setTransactionSuccessful`, то все изменения откатятся.

Потом начинаем вставлять. Для каждого значения нормализуем его, как это было с `insert`, а потом вставляем в базу данных и если прошло удачно, то увеличиваем количество удачных операций. Возвращаем его.

- `public String getType(Uri uri)`. Метод берёт `Uri`, на который может отвечать наш `ContentProvider`, и возвращает по нему тип записи для этого `Uri`. В нашем случае, для возвращаемого значения может быть только два варианта. Либо, это специфичный `Uri`, который указывает на одну запись, либо он указывает сразу на группу записей. Тип `Uri` определяется с помощью `sUriMatcher`, который мы построили в самом начале с помощью метода `buildUriMatcher`.

В нашем случае, метод работает следующим образом - если в качестве `Uri` передаётся запись, указывающая на всю табличку, то мы возвращаем тип множественный(то есть это запрос ко всем элементам таблички), а если путь к конкретной записи, то тип `Uri` - тип одной записи.

```

1  @Override
2      public String getType(Uri uri) {
3          switch (sUriMatcher.match(uri)) {
4              case WEATHER:
5                  return WeatherContract.WeatherEntry.CONTENT_TYPE;

```



```

6         case WEATHER_WITH_DATE:
7             return WeatherContract.WeatherEntry.CONTENT_ITEM_TYPE;
8         default:
9             throw new UnsupportedOperationException("Unknown uri: " + uri);
10    }
11 }

```

- getWeather и getWeatherByDate.

```

1 private Cursor getWeather(String[] projection, String selection, String[]
    selectionArgs, String sortOrder)
2 {
3     return mDbHelper.getReadableDatabase().query(WeatherContract.WeatherEntry.
        TABLE_NAME,
4         projection,
5         selection,
6         selectionArgs,
7         null,
8         null,
9         sortOrder);
10 }
11
12 private Cursor getWetherByDate(Uri uri, String[] projection, String sortOrder)
    {
13     long date = WeatherContract.WeatherEntry.getDateFromUri(uri);
14     String selection = WeatherContract.WeatherEntry.COLUMN_DATE + " = ? ";
15     String selectionArgs[] = new String[] { Long.toString(date) };
16
17     return mDbHelper.getReadableDatabase().query(WeatherContract.WeatherEntry.
        TABLE_NAME,
18         projection,
19         selection,
20         selectionArgs,
21         null,
22         null,
23         sortOrder);
24 }

```

Как мы видим, за getWeather и getWeatherByDate скрывается всего лишь то же получение базы данных из mDbHelper - .getReadableDatabase, так как мы считываем значения из базы данных. А затем вызываем метод query с запросом.

.getWeather совсем простой метод, он просто берёт и передаёт всё, что ему дали в качестве параметров в запрос к SQLiteDatabase.

.getWeatherByDate немного более сложный метод, так как нам надо преобразовать запрос к строчке в запрос ко всей бд(SQLiteDatabase умеет обрабатывать запросы только ко всей табличке). Таким образом, мы создаём selection на равенство значения в COLUMN_DATE, а selectionArgs достаём из Uri - это просто строковое представление нашей даты. Сначала используем определённый нами в Contract-е метод .getDateFromUri, а затем преобразуем дату в строчку.

Возвращаем мы и там, и там - Cursor.

- Остался единственный метод, который мы не обсудили:

```

1 private static void normalizeDate(ContentValues values) {
2     if (values.containsKey(WeatherContract.WeatherEntry.COLUMN_DATE)) {
3         long date = WeatherContract.normalizeDate(
4             values.getAsLong(WeatherContract.WeatherEntry.COLUMN_DATE));
5         values.put(WeatherContract.WeatherEntry.COLUMN_DATE, date);
6     }
7 }

```

Он проверяет, верно ли, что `values` содержит дату, то есть проверяет есть ли там пара, в которой ключ совпадает с названием колонки для даты, и если да, то нормализует её с помощью метода нормализации, который мы описали в `Contract`, а потом кладёт обратно уже преобразованное значение.

- Закрывать базу данных не нужно, несмотря на то, что, казалось бы, работа с базой данных - это практически то же самое, что работа с файлами. Есть метод, который позволяет это сделать, но он создан исключительно для дебага.

Ну вот и всё, с базами данных мы разобрались.

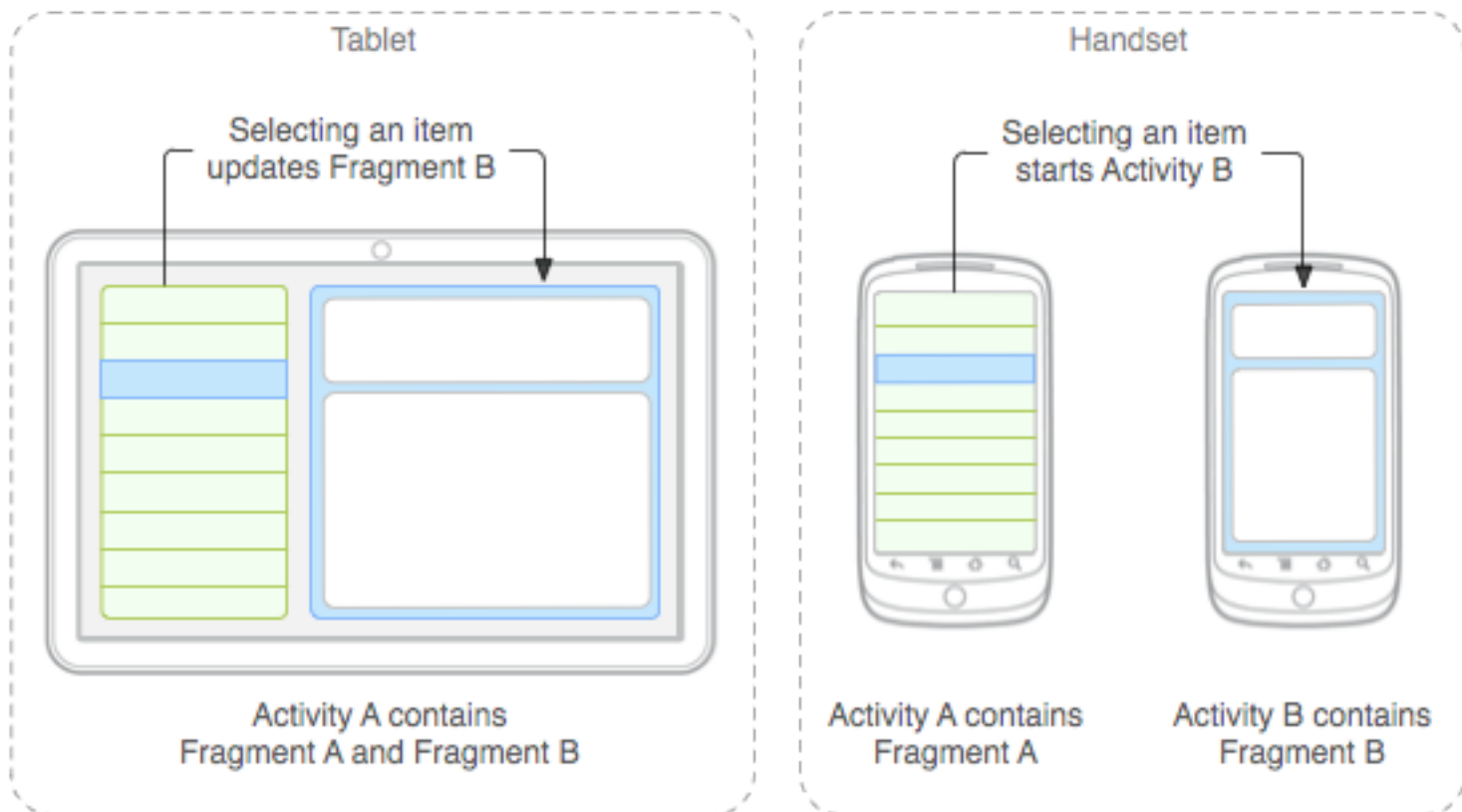
В общем, `ContentProvider` - это просто класс, который делает запросы к базе данных, скрывая это за интерфейсом `Uri`.

12 Fragments

Создадим класс `ForecastFragment` - наследник класса `Fragment`.

Фрагменты - в некотором роде мини-`Activity`, которые располагаются на основном `Activity` и имеют свой собственный `lifecycle`, немного отличающийся от `lifecycle Activity`. На одном `Activity` может быть несколько фрагментов, их жизненный цикл прекращается после прекращения жизненного цикла `Activity`.

Зачем нужны фрагменты? Например, чтобы приложение нормально выглядело и на телефоне, и на планшете: если на экране планшета останется много свободного места, это плохо, поэтому возможно на `Activity` в телефоне показывать один фрагмент, а в планшете - два:



Перейдем к рассмотрению нашего `ForecastFragment`:

```
1 public class ForecastFragment extends Fragment implements LoaderManager.  
   LoaderCallbacks<Cursor> {  
2  
3     private static final int FORECAST_LOADER_ID = 0;  
4
```

```

5     private static final String[] FORECAST_COLUMNS = {
6         WeatherContract.WeatherEntry._ID,
7         WeatherContract.WeatherEntry.COLUMN_DATE,
8         WeatherContract.WeatherEntry.COLUMN_DAY_TEMP,
9         WeatherContract.WeatherEntry.COLUMN_NIGHT_TEMP,
10        WeatherContract.WeatherEntry.COLUMN_WIND_SPEED,
11        WeatherContract.WeatherEntry.COLUMN_DEGREES,
12        WeatherContract.WeatherEntry.COLUMN_WEATHER_ID
13    };
14
15    public static final int COLUMN_WEATHER_DATE = 1;
16    public static final int COLUMN_WEATHER_DAY = 2;
17    public static final int COLUMN_WEATHER_NIGHT = 3;
18    public static final int COLUMN_WEATHER_WIND_SPEED = 4;
19    public static final int COLUMN_WEATHER_WIND_DEGREES = 5;
20    public static final int COLUMN_WEATHER_ID = 6;
21
22    private ForecastAdapter mForecastAdapter;
23
24    public ForecastFragment() { }
25
26    @Override
27    public void onCreate(Bundle savedInstanceState) {
28        super.onCreate(savedInstanceState);
29        getLoaderManager().initLoader(FORECAST_LOADER_ID, null, this);
30    }
31
32    @Override
33    public View onCreateView(LayoutInflater inflater, ViewGroup container,
34        Bundle savedInstanceState) {
35        super.onCreateView(inflater, container, savedInstanceState);
36
37        View root = inflater.inflate(R.layout.fragment_main, container, false);
38        ListView mListView = (ListView) root.findViewById(R.id.forecast_list_view);
39        mForecastAdapter = new ForecastAdapter(getActivity(), null, 0);
40        mListView.setAdapter(mForecastAdapter);
41        return root;
42    }
43
44    ...
45 }

```

Здесь мы переопределяем два метода класса Fragment - onCreate и onCreateView. Первый вызывается после того, как завершен вызов метода onCreate у Activity, к которой прикреплен наш Fragment. onCreateView вызывается, когда фрагменту нужно нарисовать его UI в первый раз и возвращает View для UI фрагмента.

Теперь добавим ForecastFragment на MainActivity:

```

1 public class MainActivity extends ActionBarActivity {
2
3     private LocationManager locationManager;
4     private PendingIntent mLocationChangedIntent;
5
6     @Override
7     protected void onCreate(Bundle savedInstanceState) {
8         super.onCreate(savedInstanceState);
9         setContentView(R.layout.activity_main);
10        if (savedInstanceState == null) {
11            getSupportFragmentManager().beginTransaction()
12                .add(R.id.container, new ForecastFragment())
13                .commit();

```

```

14         }
15         ...
16     }
17     ...
18 }

```

Здесь мы получаем `FragmentManager` - интерфейс для взаимодействия с `Fragment` внутри `Activity`, и у него вызываем метод `beginTransaction`, который начинает конвейер операций с фрагментами, дальше добавляем новый `ForecastFragment`, указав контейнер, куда он отправится, и, наконец, исполняем все действия с помощью метода `commit`. По идентификатору `R.id.container` мы найдем `layout` для `MainActivity`, так что мы просто располагаем фрагмент на нем.

13 ListView & Adapter

Единственная компонента нашего фрагмента - это список, который отрисовывается с помощью `ListView`. `ListView` и `Adapter` - это в некотором смысле андроидовская реализация того, что называют `Model-View Controller`.

Идея в следующем - есть какая-то `View`, рисующая UI. Есть набор данных, по которым этот UI рисуется. И они разделены по разным классам. Таким образом, мы, например, можем подменить один `View` другим, не меняя модельку, в которой эти данные хранятся.

- Для того, чтобы рисовать списки, используют `ListView`.
- `Adapter` - класс, который предоставляет нам доступ к данным. А так как у нас есть база данных, то в нашем случае мы хотим, чтобы `Adapter` предоставлял нам доступ к нашей базе данных. Для этого в `Android` есть специальный класс, называемый `CursorAdapter` (он, например, при обновлении БД оповещает `View` об этом).
- При создании `CursorAdapter` необходимо переопределить несколько методов - `public View newView(...)`, `public void bindView(...)`.
- `public View newView(Context context, Cursor cursor, ViewGroup parent)` - создаёт новую `View`. Вызывается, когда для каких-то данных в `ListView` надо добавить новое `View`. `context` - для доступа к системным ресурсам. `cursor` - указывает на текущую строчку в базе данных (на ту строчку, для которой мы хотим создать новую `View`). `parent` - элемент, внутрь которого мы должны новую `View` положить.

```

1  @Override
2  public View newView(Context context, Cursor cursor, ViewGroup parent) {
3      View view = LayoutInflater.from(context).inflate(R.layout.
4          forecast_list_item_view,
5          parent, false);
6      view.setTag(new ViewHolder(view));
7      return view;
8  }

```

- `LayoutInflater` - это сущность, которая по `xml`-описанию `View` создаёт эту `View`.

Пример `xml`-описания:

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout
3      xmlns:android="http://schemas.android.com/apk/res/android"
4      android:orientation="horizontal"
5      android:gravity="center_vertical"
6      android:layout_width="match_parent"
7      android:layout_height="wrap_content"

```

```

8      android:layout_margin="20dp">
9
10     <LinearLayout
11         android:orientation="vertical"
12         android:layout_width="0dp"
13         android:layout_height="wrap_content"
14         android:layout_weight="1">
15
16         <TextView
17             android:id="@+id/date"
18             android:layout_width="wrap_content"
19             android:layout_height="wrap_content"
20             android:layout_margin="10dp"/>
21
22         <TextView
23             android:id="@+id/wind"
24             android:layout_width="wrap_content"
25             android:layout_height="wrap_content"
26             android:layout_margin="10dp"/>
27
28     </LinearLayout>
29
30     <LinearLayout
31         android:orientation="vertical"
32         android:layout_width="0dp"
33         android:layout_height="wrap_content"
34         android:layout_weight="1"
35         android:gravity="right">
36
37         <ImageView
38             android:id="@+id/icon"
39             android:layout_width="wrap_content"
40             android:layout_height="wrap_content"
41             android:layout_margin="10dp"/>
42
43         <TextView
44             android:id="@+id/temp"
45             android:layout_width="wrap_content"
46             android:layout_height="wrap_content"
47             android:layout_margin="10dp"/>
48
49     </LinearLayout>
50
51 </LinearLayout>

```

Здесь, внешний LinearLayout - это такая Group, в которую добавленные элементы горизонтально(`android:orientation="horizontal"`) ложаться друг за другом.

Два внутренних LinearLayout - это аналогичные Group, только элементы ложаться по вертикали(`android:orientation="vertical"`).

TextView - это просто текст, ImageView - изображение.

Надо заметить, что View - это какая-то рисуемая сущность, а Group - сущность разметки, то есть она размещает по каким-то правилам View или другие Group. Так вот все *Layout - это Group.

У каждого элемента можно задать id, по которому впоследствии его можно будет найти(например, с помощью метода `findViewById(int id)` у Context. Без id элемент найти будет трудно, поэтому если какой-то элемент используется или изменяется в коде, ему обязательно нужно дать своё id.

- Можно размечать экран и с помощью кода, создавая Layout-ы и View на ходу, но xml в некотором смысле практичней, поскольку мы просто можем заменить один layout другим без изменения кода.

- Итак, `LayoutInflater` получает `xml`-файл, создаёт дерево объектов и возвращает корневой.
- Рядом с любой `View` мы можем хранить данные, добавляя их с помощью метода `.setTag(Data data)`;
- `ViewHolder`, который мы добавляем в качестве дополнительной информации к `View`, один раз находит нужные нам `View`, которые мы будем изменять, и кеширует их, чтобы не приходилось каждый раз бегать по дереву объектов. Это действительно помогает, потому что `newView` не всегда вызывается, когда на экране появляется очередная `View`. Андроид располагает небольшими ресурсами, поэтому созданные `View` он может переиспользовать, подав их же ещё раз на вход `bindView`. Например, так происходит при прокручивании списка.
- Следующий метод, про который нам надо поговорить - `public void bindView(View view, Context context, Cursor cursor)`. `bindView` связывает `View` с данными, которые мы хотим на ней отобразить. То есть `bindView` в нашем случае должен заполнить поля у текстовых `View`, которые отвечают за температуру, чтобы они отображали актуальные данные:

```

1      @Override
2      public void bindView(View view, Context context, Cursor cursor) {
3          ViewHolder holder = (ViewHolder) view.getTag();
4
5          long date = cursor.getLong(ForecastFragment.COLUMN_WEATHER_DATE);
6          holder.date.setText(Utility.getFriendlyDayString(date));
7
8          double day = cursor.getDouble(ForecastFragment.COLUMN_WEATHER_DAY);
9          double nigh = cursor.getDouble(ForecastFragment.COLUMN_WEATHER_NIGHT);
10         holder.temp.setText(Utility.getFormattedTemperature(day, nigh));
11
12         double speed = cursor.getDouble(ForecastFragment.
13             COLUMN_WEATHER_WIND_SPEED);
14         double degrees = cursor.getDouble(ForecastFragment.
15             COLUMN_WEATHER_WIND_DEGREES);
16         holder.wind.setText(Utility.getFormattedWind(speed, degrees));
17
18         int weatherId = cursor.getInt(ForecastFragment.COLUMN_WEATHER_ID);
19         holder.icon.setImageResource(Utility.getWeatherConditionIcon(weatherId));
20     }
21
22     public static class ViewHolder {
23         public TextView date;
24         public TextView temp;
25         public TextView wind;
26         public ImageView icon;
27
28         public ViewHolder(View view) {
29             date = (TextView) view.findViewById(R.id.date);
30             temp = (TextView) view.findViewById(R.id.temp);
31             wind = (TextView) view.findViewById(R.id.wind);
32             icon = (ImageView) view.findViewById(R.id.icon);
33         }
34     }

```

К `CursorAdapter` данные приходят от `Cursor`, он позволяет обращаться к различным методам `Cursor`: `getDouble()`, `getPosition()`.

Теперь посмотрим, что же происходит у нашего `ForecastFragment` в методе `onCreateView`:

```

1  @Override
2  public View onCreateView(LayoutInflater inflater, ViewGroup container,

```

```

3             Bundle savedInstanceState) {
4         super.onCreateView(inflater, container, savedInstanceState);
5
6         View root = inflater.inflate(R.layout.fragment_main, container, false);
7         ListView mListView = (ListView) root.findViewById(R.id.forecast_list_view);
8         mForecastAdapter = new ForecastAdapter(getActivity(), null, 0);
9         mListView.setAdapter(mForecastAdapter);
10        return root;
11    }

```

Здесь мы:

- Вызываем метод родителя.
- С помощью LayoutInflater получаем root нашей иерархии View.
- С помощью root находим наш ListView.
- Создаем ForecastAdapter и связываем его с View, для которой он отображает данные.
- Возвращаем root.

Для получения данных есть отдельный класс LoaderManager - это штука, которая управляет загрузками. Бывают встроенные Loader, если не устраивают, можно реализовать свой. Для этого нужно реализовать интерфейс LoaderCallbacks. Этот интерфейс вызывается, когда Loader загрузит какие-то данные.

LoaderManager можно получить с помощью метода getLoaderManager() у Context. Так как возможна потребность в сразу нескольких загрузках, методу getLoader() у LoaderManager нужно передавать id, чтобы у нас была возможность отличать загрузки друг от друга.

Для Cursor мы используем CursorLoader, чтобы загружать данные из нашего ContentProvider.

А чтобы до нас доходили оповещения Loader о том, что что-то загрузилось, нужно реализовать методы у интерфейса LoaderCallbacks<Cursor>.

Теперь посмотрим на полный код ForecastFragment:

```

1  public class ForecastFragment extends Fragment implements LoaderManager.
    LoaderCallbacks<Cursor> {
2
3      private static final int FORECAST_LOADER_ID = 0;
4
5      private static final String[] FORECAST_COLUMNS = {
6          WeatherContract.WeatherEntry._ID,
7          WeatherContract.WeatherEntry.COLUMN_DATE,
8          WeatherContract.WeatherEntry.COLUMN_DAY_TEMP,
9          WeatherContract.WeatherEntry.COLUMN_NIGHT_TEMP,
10         WeatherContract.WeatherEntry.COLUMN_WIND_SPEED,
11         WeatherContract.WeatherEntry.COLUMN_DEGREES,
12         WeatherContract.WeatherEntry.COLUMN_WEATHER_ID
13     };
14
15     public static final int COLUMN_WEATHER_DATE = 1;
16     public static final int COLUMN_WEATHER_DAY = 2;
17     public static final int COLUMN_WEATHER_NIGHT = 3;
18     public static final int COLUMN_WEATHER_WIND_SPEED = 4;
19     public static final int COLUMN_WEATHER_WIND_DEGREES = 5;
20     public static final int COLUMN_WEATHER_ID = 6;
21
22     private ForecastAdapter mForecastAdapter;
23
24     public ForecastFragment() { }
25

```



```

26     @Override
27     public void onActivityCreated(Bundle savedInstanceState) {
28         super.onActivityCreated(savedInstanceState);
29         getLoaderManager().initLoader(FORECAST_LOADER_ID, null, this);
30     }
31
32     @Override
33     public View onCreateView(LayoutInflater inflater, ViewGroup container,
34                             Bundle savedInstanceState) {
35         super.onCreateView(inflater, container, savedInstanceState);
36
37         View root = inflater.inflate(R.layout.fragment_main, container, false);
38         ListView mListView = (ListView) root.findViewById(R.id.forecast_list_view);
39         mForecastAdapter = new ForecastAdapter(getActivity(), null, 0);
40         mListView.setAdapter(mForecastAdapter);
41         return root;
42     }
43
44     @Override
45     public Loader<Cursor> onCreateLoader(int id, Bundle args) {
46         String sortOrder = WeatherContract.WeatherEntry.COLUMN_DATE + " ASC";
47         Uri weatherUri = WeatherContract.WeatherEntry.CONTENT_URI;
48
49         return new CursorLoader(getActivity(), weatherUri, FORECAST_COLUMNS, null,
50                                 null, sortOrder);
51     }
52
53     @Override
54     public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
55         mForecastAdapter.swapCursor(data);
56     }
57
58     @Override
59     public void onLoaderReset(Loader<Cursor> loader) {
60         mForecastAdapter.swapCursor(null);
61     }
62 }

```

Здесь мы:

- Инициализируем загрузчик и запускаем его в методе `onActivityCreated` с помощью метода `initLoader` у `LoaderManager`. Ему мы передаем три параметра - идентификатор загрузчика, дополнительные аргументы и `LoaderCallbacks`, которому `LoaderManager` сообщит о изменении состояния загрузчика.
- Описываем `Loader`, который мы хотим использовать для загрузки. Конструктору `CursorLoader` передается `Context`, ссылка на данные, указание на то, какие колонки вернуть, а также порядок их сортировки.
Внутри этого конструктора будет вызываться метод `query()` с теми же самыми параметрами у `ContentProvider`.
- Обрабатываем две ситуации: `onLoadFinished` и `onLoaderReset`. В первой мы успешно загрузили информацию, тогда воспользуемся методом `swapCursor` у нашего адаптера, который освободит старые данные и будет работать с новыми.
- Во втором варианте информация недоступна, и мы должны освободить старые данные с помощью `mForecastAdapter.swapCursor(null)`.

14 Директория res

Она же ресурсы. Если мы хотим сохранить некоторую картинку, которой будет пользоваться наше приложение(например, иконку), или некоторый gif-файл, который мы хотим отображать на Activity, то их можно положить заранее сюда и доставать в тех местах в коде, где нам это нужно.Здесь хранятся:

- Иконки - в res/mipmap-hdpi, res/mipmap-mdpi, res/mipmap-xhdpi, res/mipmap-xxhdpi. Какая из них будет использована, зависит от размера экрана.
- Изображения - в res/drawable.
- Layout - в res/layout.
- Строки - в res/values/strings.xml.
- Стили - в res/styles.
- Содержимое меню приложения - в res/menu
- Значения bool, color, dimen - в res/values.

Приведем пример того, как используются resources, а именно покажем, как их применять для локализации.

Рассмотрим файл res/values/strings.xml некоторого приложения:

```
1 <resources>
2     <string name="app_name">ActivityExample</string>
3
4     <string name="hello_world">Hello world!</string>
5     <string name="action_settings">Settings</string>
6     <string name="title_activity_sms">SMSActivity</string>
7 </resources>
```

Допустим, мы хотим получить строку, на которой написано "Привет, мир!" на английском, если в настройках телефона указан не французский язык, а если французский, то просто "Bonjour tout le monde!" и разместить на TextView:

```
1 TextView textView = findViewById(R.id.textView);
2 String helloWorld = getResources().getString(R.string.hello_world);
3 textView.setText(helloWorld);
```

Здесь мы:

- Находим с помощью метода findViewById(int id) у Context нашу TextView.
- Обращаемся к ресурсам с помощью метода getResources() у Context и находим там нашу строку с помощью метода getString(int id)
- Размещаем строку на TextView.

Заметим, что здесь мы два раза пользуемся id - идентификатором, который есть как у View, так и у ресурсов. Получить идентификатор для строки, хранящейся в ресурсах, можно, написав R.string.*name*, где name соответствует полю name в xml-файле.

Теперь хотим добавить вариант на французском. Для этого создадим директорию res/values-fr и в ней создадим файл strings.xml, внутри которого напомним:

```
1 <resources>
2     <string name="app_name">ActivityExample</string>
3
4     <string name="hello_world">Bonjour tout le monde!</string>
5     <string name="action_settings">Settings</string>
6     <string name="title_activity_sms">SMSActivity</string>
7 </resources>
```

Код оставим таким же, без изменений. Теперь, если на телефоне выставлен французский язык, на TextView будет размещена надпись по-французски, а если нет - то на английском. Аналогично можно создавать альтернативные ресурсы и для других языков.