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

1 Android Manifest

Манифест - файлик, 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.

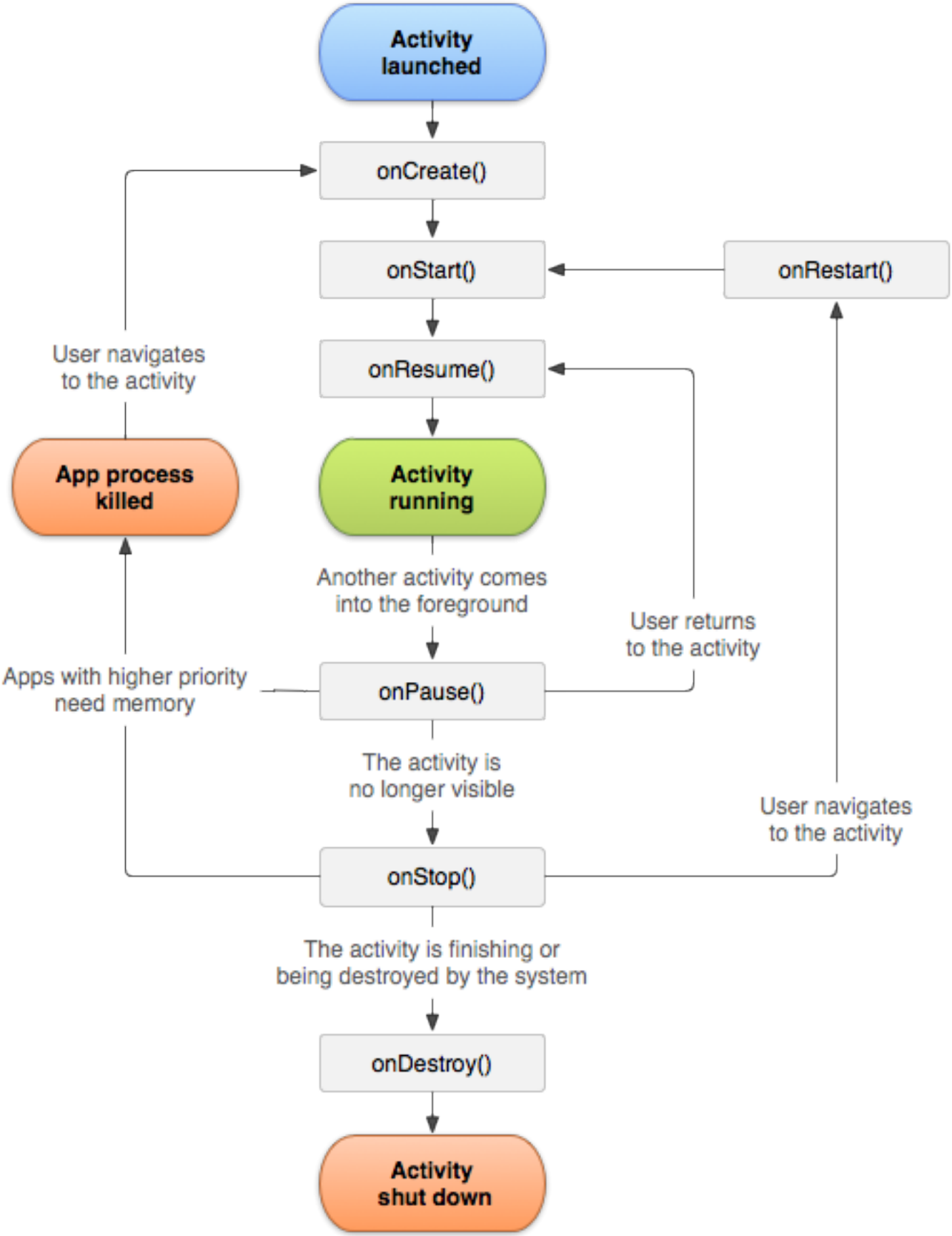
В принципе, части приложения можно регистрировать и программно. Но мы этого касаться не будем.

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. Соответсвенно, 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 взывается .onResume, таким образм, .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 - активити, а именно - главное. Таким образом, манифест сейчас выглядит так:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="edu.spbau.android.forecast" >

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</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.

```
public class MainActivity extends ActionBarActivity {

    private LocationManager mLocationManager;
    private PendingIntent mLocationChangedIntent;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
    }

    @Override
    protected void onResume() {
        ...
    }

    @Override
    protected void onPause() {
        ...
    }


    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        ...
```

```
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, котороый 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 отрисуется. 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 файлики, 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 файлик-xml назыается 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 отображаетсяв 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 соответсвующие. Там же(в res/values) есть специальное место для хранения строковых литералов, называется файл strings.xml. Там мы храним строковые литералы. Вот наш файлик:

```
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     mLocationManager = (LocationManager) getSystemService(Context.LOCATION_SERVICE)
           ;
12     Intent intent = new Intent(this, LocationChangedReceiver.class);
13     mLocationChangedIntent =
14             PendingIntent.getBroadcast(this, 0, intent, PendingIntent.
                   FLAG_UPDATE_CURRENT);
15 }
```

Класс Bundle создаётся для сохранения важной информации(например, пользовательского UI), в onCreate его можно восстановить. А onSaveInstanceState(Bundle savedInstanceState) можно переопределить 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. PandingIntent.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 будет 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 измененяет 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       mLocationManager.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       mLocationManager.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 будет переопределять onStope 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 интентов. Когда приходит интент, 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      public LocationChangedReceiver() { }
6
7      @Override
8      public void onReceive(Context context, Intent intent) {
9          final String locationKey = LocationManager.KEY\_LOCATION\_CHANGED;
10
11         if (intent.hasExtra(locationKey)) {
12             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 Inent 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 хранение данных вида ключ(строка)-значение(Object). Так что .get() возвращает Object, его надо прикастовать к Location - класс, в котором содержитьс 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 манифесете:

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

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

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="edu.spbau.android.forecast" >

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

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

        <service
            android:name=".ForecastUpdateService"
            android:exported="false" />
    </application>

</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:

```java
public class ForecastUpdateService extends IntentService {

    private final static String TAG = ForecastUpdateService.class.getSimpleName();

    public ForecastUpdateService() {
        super(TAG);
    }

    private void parseJsonData(String json) {
        final String JSON_LIST = "list";
        final String JSON_SPEED = "speed";
        final String JSON_DEGREE = "deg";
        final String JSON_TEMPERATURE = "temp";
        final String JSON_DAY_TEMPERATURE = "day";
        final String JSON_NIGHT_TEMPERATURE = "night";
        final String JSON_WEATHER = "weather";
        final String JSON_WEATHER_ID = "id";

        try {
            JSONObject forecastJson = new JSONObject(json);
            JSONArray weatherArray = forecastJson.getJSONArray(JSON_LIST);

            ArrayList<ContentValues> forecast = new ArrayList<>(weatherArray.length
                ());
```

```
24
25              Time dayTime = new Time();
26              dayTime.setToNow();
27              int julianStartDay = Time.getJulianDay(System.currentTimeMillis(),
                    dayTime.gmtoff);
28              dayTime = new Time();
29
30              for(int i = 0; i < weatherArray.length(); i++) {
31                  JSONObject dayForecast = weatherArray.getJSONObject(i);
32
33                  long dateTime = dayTime.setJulianDay(julianStartDay + i);
34
35                  double speed = dayForecast.getDouble(JSON_SPEED);
36                  double degree = dayForecast.getDouble(JSON_DEGREE);
37
38                  JSONObject temperatureObject = dayForecast.getJSONObject(
                        JSON_TEMPERATURE);
39                  double day = temperatureObject.getDouble(JSON_DAY_TEMPERATURE);
40                  double night = temperatureObject.getDouble(JSON_NIGHT_TEMPERATURE);
41
42                  JSONArray weatherObjects = dayForecast.getJSONArray(JSON_WEATHER);
43                  JSONObject weatherObject = weatherObjects.getJSONObject(0);
44                  int weather = weatherObject.getInt(JSON_WEATHER_ID);
45
46                  ContentValues weatherValues = new ContentValues();
47                  weatherValues.put(WeatherContract.WeatherEntry.COLUMN_DATE,
                        dateTime);
48                  weatherValues.put(WeatherContract.WeatherEntry.COLUMN_DEGREES,
                        degree);
49                  weatherValues.put(WeatherContract.WeatherEntry.COLUMN_WIND_SPEED,
                        speed);
50                  weatherValues.put(WeatherContract.WeatherEntry.COLUMN_NIGHT_TEMP,
                        night);
51                  weatherValues.put(WeatherContract.WeatherEntry.COLUMN_DAY_TEMP, day
                        );
52                  weatherValues.put(WeatherContract.WeatherEntry.COLUMMN_WEATHER_ID,
                        weather);
53                  forecast.add(weatherValues);
54              }
55
56              if (forecast.size() > 0) {
57                  ContentValues values[] = new ContentValues[forecast.size()];
58                  forecast.toArray(values);
59                  getContentResolver().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.
                           getLatitude()))
91                      .appendQueryParameter(LONGITUDE_PARAM, Double.toString(location
                           .getLongitude()))
92                      .appendQueryParameter(MODE_PARAM, JSON)
93                      .appendQueryParameter(UNITS_PARAM, METRIC).build();
94              URL url = new URL(uri.toString());
95              connection = (HttpURLConnection) url.openConnection();
96              connection.setRequestMethod("GET");
97              connection.connect();
98
99              InputStream inputStream = connection.getInputStream();
100             StringBuilder buffer = new StringBuilder();
101             if (inputStream == null) {
102                 return;
103             }
104             reader = new BufferedReader(new InputStreamReader(inputStream));
105             String line;
106             while ((line = reader.readLine()) != null) {
107                 buffer.append(line);
108                 buffer.append("\n");
109             }
110
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 извлечение локации из Intent. Остальные - специальные параметры для декодирования и кодирования определённых форматов.

Дальше мы проверяем, что пришёдший 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 содержиться, 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 соотетсвующие 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 получени данных относительно точной локации, также нужно разрешение - оно первое в списке.
- Второе позволяет открывать сетевые сокеты.
- 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 HttpURLConnection. Сначала мы долго-долго строим URL-адрес, в котором находится наш запрос - мы достаём из Location широту и долготу, а потом получаем HttpURLConnection по этому URL. Возвращается методом .openConnection():

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

  Здесь нужен каст, так как .openConnection на самом деле возвращает URLConnection, но по факту это - HttpURLConnection, 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 прочитать 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 закончаться 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. Соответсвенно, 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:

```
<provider
  android:authorities="edu.spbau.android.forecast"
    android:name=".WeatherProvider" />
```

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

Таким образом, манифест получается такой:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="edu.spbau.android.forecast" >

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

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

        <service
            android:name=".ForecastUpdateService"
            android:exported="false" />

        <provider
            android:authorities="edu.spbau.android.forecast"
            android:name=".WeatherProvider" />
    </application>

</manifest>
```

Это - окончательный вариант и больше он меняться не будет.

9 Contract

Перд тем, как подробней 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.

Соответсвенно:

- 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://" +
           CONTENT_AUTHORITY);
5      public static final String PATH_WEATHER = "weather";
6
7      public static long normalizeDate(long gmt) {
8          Time time = new Time();
9          time.set(gmt);
10         int day = Time.getJulianDay(gmt, time.gmtoff);
11         return time.setJulianDay(day);
12     }
13
14     public static final class WeatherEntry implements BaseColumns {
15
16         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 + "/
                       " + PATH_WEATHER;
21
22         public static final String CONTENT_ITEM_TYPE =
23                 ContentResolver.CURSOR_ITEM_BASE_TYPE + "/" + CONTENT_AUTHORITY + "
                       /" + PATH_WEATHER;
24
25         public static final String TABLE_NAME = "weather";
26
27         public static final String COLUMN_DATE = "date";
28         public static final String COLUMN_DAY_TEMP = "day_temp";
29         public static final String COLUMN_NIGHT_TEMP = "nigh_temp";
30         public static final String COLUMN_WIND_SPEED = "wind";
31         public static final String COLUMN_DEGREES = "direction";
32         public static final String COLUMMN_WEATHER_ID = "weather_id";
33
34         public static long getDateFromUri(Uri uri) {
35             return Long.parseLong(uri.getPathSegments().get(1));
36         }
37
```

```
38          public static Uri buildWeatherUri(long date) {
39              return ContentUris.withAppendedId(CONTENT_URI, date);
40          }
41
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 табличик. 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, соотетсвующий 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, называющаяся 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, соотетсвующей определённой дате.

  Выражения для CONTENT_TYPE и CONTENT_ITEM_TYPE вегда 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 обновлении, когда нужно. Соотетсвенно, 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 строчки 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 представленяет из себя число, а второе - строку. Мы назовём нашу базу данных "weather.db". В конструктуре 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.
               TABLE_NAME + " (" +
4              WeatherEntry._ID + " INTEGER PRIMARY KEY AUTOINCREMENT," +
5              WeatherEntry.COLUMN_DATE + " INTEGER NOT NULL," +
6              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.COLUMMN_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 струкутру 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 класа WeatherEntry - это вложенный класс в WeatherContract, который отвечает за нашу единственную табличку. В остальном мы просто задаём данные какого вида будут хранится в колонках.

  Внутри каждого запроса к базе данных есть также поле версии, в которой описано, какой версии должна соответвовать база данных. И, если текущая версия базы данных не соответсвует 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 ContentProvic - 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[]
           selectionArgs,
17                               String sortOrder)
18     {
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[]
           selectionArgs, String sortOrder) {
30     }
31
32     @Override
33     public String getType(Uri uri) {
34     }
35
36     @Override
```

```
37      public Uri insert(Uri uri, ContentValues values) {
38      }
39
40      @Override
41      public int delete(Uri uri, String selection, String[] selectionArgs) {
42      }
43
44      @Override
45      public int update(Uri uri, ContentValues values, String selection, String[]
            selectionArgs) {
46      }
47
48      @Override
49      public int bulkInsert(Uri uri, @NonNull ContentValues[] values) {
50      }
51
52  }
```

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 соответсвие 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 WeatherDBHelpe 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 идеалогически 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 + "/#",
             WEATHER_WITH_DATE);
  6      return matcher;
  7  }
  ```

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

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

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

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 массивыми строчек - которые потом можно будет составить в запрос SQL.

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

Соответсвенно, 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(
8                      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(Str 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 встваить 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 ConentValues - 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 говоит о том, что если передался пустой ContentValues, то ничего вставлять не надо.

Команда insert у SQLiteDatabase возвращает индекс строчки, 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 табличку. 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,
                            selectionArgs);
7               if (count > 0) {
8                   getContext().getContentResolver().notifyChange(uri, null);
9               }
10              return count;
11          default:
12              throw new UnsupportedOperationException("Unknown uri: " + uri);
13      }
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 звписей. 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[]
        selectionArgs) {
3       switch (sUriMatcher.match(uri)) {
4           case WEATHER:
5               int count = mDbHelper.getWritableDatabase().update(
6                       WeatherContract.WeatherEntry.TABLE_NAME, values, selection,
                            selectionArgs);
7               if (count > 0) {
8                   getContext().getContentResolver().notifyChange(uri, null);
9               }
10              return count;
11          default:
12              throw new UnsupportedOperationException("Unknown uri: " + uri);
13      }
14  }
```

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.
                                TABLE_NAME, null, value);
12                          if (id != -1) {
13                              count++;
```

```
                        }
                    }
                    db.setTransactionSuccessful();
                } finally {
                    db.endTransaction();
                }
                getContext().getContentResolver().notifyChange(uri, null);
                return count;
            default:
                throw new UnsupportedOperationException("Unknown uri: " + uri);
        }
    }

}
```

Мы ожидаем получить на вход uri таблички и затем значения, которые хотим вставить. Аналогично insert-у, если uri?переданный нам не соответсвует табличке, а соответсвует записи, мы не сможем его обработать и кидаем Exception. Аналогично insert-у 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 откатяться.

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.

```
@Override
public String getType(Uri uri) {
    switch (sUriMatcher.match(uri)) {
        case WEATHER:
            return WeatherContract.WeatherEntry.CONTENT_TYPE;
        case WEATHER_WITH_DATE:
            return WeatherContract.WeatherEntry.CONTENT_ITEM_TYPE;
        default:
            throw new UnsupportedOperationException("Unknown uri: " + uri);
    }
}
```

- getWeather и getWeatherByDate.

```
private Cursor getWeather(String[] projection, String selection, String[]
    selectionArgs, String sortOrder)
{
    return mDbHelper.getReadableDatabase().query(WeatherContract.WeatherEntry.
        TABLE_NAME,
            projection,
            selection,
            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 совсм 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 опредлённый 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 еолонки 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

- Весь UI в большинстве приложений распихан по фрагментам.
- Подробнее про Фрагменты можно почитать тут: тык

13 ListView & Adapter

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.
           forecast_list_item_view,
4              parent, false);
5      view.setTag(new ViewHolder(view));
6      return view;
7  }
```

- 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 какждый раз бегать по дереву объектов. Это действительно помогает, потому 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 актуальный данные.

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

Необходимо связать CursorAdapter с View, для которой он отображает данные(у View метод setAdapter(ada

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>.

Один из этих методов - onCreateLoader(int id, Bundle args), который возвращает собственно Loader:

```
1 @Override
2 public Loader<Cursor> onCreateLoader(int id, Bundle args) {
3   String sortOrder = WeatherContract.WeatherEntry.COLUMN_DATE + " ASC";
4   Uri weatherUri = WeatherContract.WeatherEntry.CONTENT_URI;
5
6   return new CursorLoader(getActivity(), weatherUri, FORECAST_COLUMNS, null, null,
        sortOrder);
7 }
```

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.

Еще два важных метода интерфейса LoaderCallbacks: onLoadFinished() и onLoaderReset().

При успешной загрузке:

```
1 @Override
2 public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
3   mForecastAdapter.swapCursor(data);
4 }
```

Если какие-то данные подгрузились, Loader об этом узнает, обновляет данные у себя, вызывает onLoadFinished Теперь, чтобы адаптер работал с новыми данными, вызывается swapCursor(), который освобождает старые данные.

При неуспешной загрузке(Loader "погиб"):

```
1 @Override
2 public void onLoaderReset(Loader<Cursor> loader) {
3   mForecastAdapter.swapCursor(null);
4 }
```

14 Директория resources

Отвечает за хранение стилей, иконок для различных ориентаций экрана, строковых значений для различных языков.