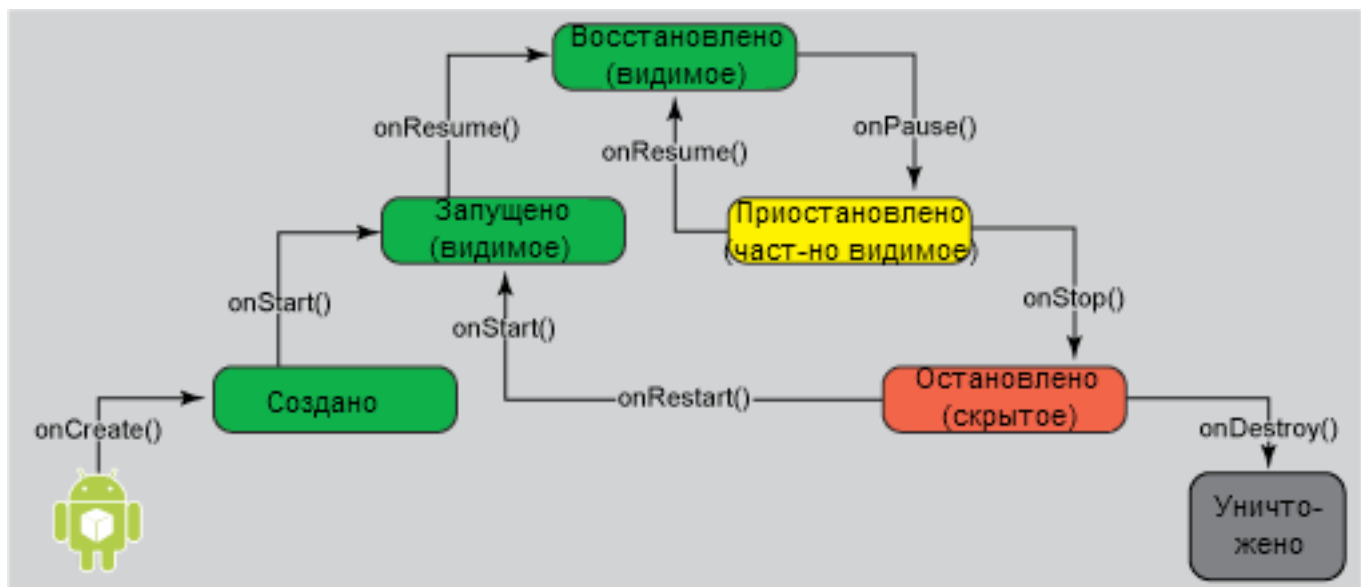


1 Android Manifest

- Описание проекта, в том числе настройки и конфигурации(например, версии).
- Прописаны permission и составляющие проекта, такие как:
 1. Activity
 2. BroadcastReceiver
 3. Service

2 Activity

- Экран приложения
- Первой в программе вызывается Activity, к которому приходит интент MAIN. Это также прописывается в Android Manifest.
- Цикл жизни:



3 BroadcastReceiver

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 батареи или о том, что Android загрузился.

BroadcastReceiver регистрируется в Android Manifest'e так:

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

В принципе, можно регистрировать и программно.

Указать, на какие intent'ы подписан BroadcastReceiver, можно двумя способами:

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

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

Здесь:

- Получаем доступ к LocationManager
- Создаем Intent и прописываем класс, куда интент должен быть направлен.
- Рассылаем этот Intent, как широковещательное сообщение.

Класс Intent - это такой способ общаться между частями приложения. В него можно записать дополнительную информацию.

```
1  startService.putExtra(LocationManager.KEY_LOCATION_CHANGED, location);
```

Доставать информацию из Intent можно так:

```
1  Location location = (Location) intent.getExtras().get(locationKey);
```

PendingIntent - наследник класса Intent. Главное его отличие от обычного интента в том, что он учитывает права приложения. Можно запускать Activity(приложение) из другого приложения, но у этого Activity(приложения) может не быть прав на работу с некоторыми ресурсами. Если эти права нужны, вы можете послать PendingIntent и передать тем самым Activity(приложению), которое вы запускаете, свои права на исполнение. Также некоторые API просто по умолчанию используют PendingIntent(например, так делает LocationManager).

Интенты бывают:

- explicit - для них явно прописывается, к какому классу его отправить
- implicit - неявно описывается, куда нужно передать сообщение. Дальше в этом разбирается Android.

В коде выше интент - explicit.

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).

В нашем Receiver мы просто напечатаем в Log некое сообщение(что такое Log мы поподробнее ещё обсудим).

```
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 }

```

Если обобщить:

- Один из стандартных классов Android, завязан на несколько(может один) intent.
- Получает intent, к которому он привязан и **быстро** реагирует(с технической точки зрения при получении intent вызывается onReceive(), который должен быстро выполняться).
- Если один BroadcastReceiver соответствует нескольким intent, то они построятся в цепочку и будут выполняться последовательно.
- Поэтому если по какому-то intent метод выполняется слишком долго, то его выполнение оборвется, чтобы другие intent не ждали.
- Так что обычно, его используют только чтобы послать какой-то другой intent или получить системное оповещение.

- Receiver должен быть прописан в Android Manifest.

```

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

```

- Подписку на intent можно тоже записать в Manifest.
- А можно это сделать и программно. Explicit intent - явно указываем класс, к которому intent отправить. Бывает ещё implicit intent.
- Класс Context - это базовый класс для частей приложения. К примеру, Activity - наследник класса Context. Он предоставляет доступ ко всем андроидовским ресурсам и поэтому является параметром метода onReceive у BroadcastReceiver.

4 Log

- 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");

```

5 Service

- Service также надо прописывать в манифесте:

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

- Для того, чтобы делать продолжительные действия BroadcastReceiver не подходит. Для этого нужно использовать Service.
- Ключ exported задаёт, позволено ли другим приложениям запускать наш сервис, Activity и тд.
- Если к нему обращаются много intent, они также становятся в очередь.
- Запускают intent к service через команду startService(...);
- При обработке intent запускается метод onHandleIntent(Intent intent);

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" />
```

7 Connection

- Самый простой способ работать с сетью - это HttpURLConnection.
 - Возвращается методом .openConnection():
- ```
1 URL url = new URL(uri.toString());
2 connection =
3 (HttpURLConnection) url.openConnection();
```
- Протокол http может обрабатывать различные запросы. Например, запрос на получение информации. Если мы хотим получить информацию - пишем следующее:

```
1 connection.setRequestMethod("GET");
2 connection.connect();
```

Пример считывания информации:

```
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 }
```

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.

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.

## 8 Data bases

- `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" />
```

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.
- В этом классе необходимо определить:
  - `public static final Uri CONTENT_URI` – `Uri`, по которому мы будем обращаться к `ContentProvider`, так как доступ к `ContentProvider` осуществляется только через `Uri`.
  - `public static final String CONTENT_TYPE` – тип набора записей в `ContentProvider`.
  - `public static final String CONTENT_ITEM_TYPE` – тип одной записи в `ContentProvider`

Последние два всегда выглядят одинаково, но их всё равно приходится прописывать.

- 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.

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 + "/"
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 }

```

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, который нам подаётся обернутым в интерфейс.
- Также, именно через него мы создаём базу данных в первый раз. При первом запуске у SQLiteOpenHelper вызывается метод public void onCreate(SQLiteDatabase db), в котором мы создаём базу данных с помощью запросов к SQLiteDatabase.
- В классе нужно переопределить также метод public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion), который позволяет обновить базу данных в процессе работы приложения. Как-то изменять структуру или добавлять новые таблицы, к примеру.

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(oldVersion - текущая версия бд, newVersion - версия, требуемая приложением)`.

Самый простой способ изменить бд - это удалить всё и создать заново, что мы, собственно, и делаем. Надо учитывать при этом, что мы при таком подходе теряем все данные. Но в нашем случае, нам это не важно.

Мы наследуемся от `SQLiteOpenHelper` следующим образом:

```
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 String SQL_CREATE_WEATHER_TABLE = "CREATE TABLE " + WeatherEntry.TABLE_NAME
13 + " (" +
14 WeatherEntry._ID + " INTEGER PRIMARY KEY AUTOINCREMENT," +
15 WeatherEntry.COLUMN_DATE + " INTEGER NOT NULL," +
16 WeatherEntry.COLUMN_DAY_TEMP + " REAL NOT NULL," +
17 WeatherEntry.COLUMN_NIGHT_TEMP + " REAL NOT NULL," +
18 WeatherEntry.COLUMN_WIND_SPEED + " REAL NOT NULL," +
19 WeatherEntry.COLUMN_DEGREES + " REAL NOT NULL," +
20 WeatherEntry.COLUMN_WEATHER_ID + " INTEGER NOT NULL, " +
21 "UNIQUE (" + WeatherEntry.COLUMN_DATE + ") ON CONFLICT REPLACE);";
22
23 db.execSQL(SQL_CREATE_WEATHER_TABLE);
24 }
25
26 @Override
27 public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
28 db.execSQL("DROP TABLE IF EXISTS " + WeatherEntry.TABLE_NAME);
29 onCreate(db);
30 }
31 }
```

Теперь, наконец, разберём поподробнее класс `ContentProvider`:

- Мы создадим класс-наследник от `ContentProvider` - `WeatherProvider`. У `ContentProvider` можно переопределить только метод `onCreate` - так как именно там обычно происходит подключение к базе данных, но и этот метод переопределять необязательно.
- В нашем случае, подключение к бд - это просто создание `WeatherDBHelper`.

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

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

- `public Uri insert(...)` - вызывается, если мы хотим вставить что-то в `ContentProvider`(читай, нашу бд). Здесь обрабатываются запросы типа `INSERT`.
- `public int delete(...)` - вызывается, если мы хотим удалить что-то из `ContentProvider`(читай, из нашей бд). Здесь обрабатываются запросы типа `DELETE`.
- `public int update(...)` - вызывается, если мы хотим обновить какую-то запись в `ContentProvider`(читай, в нашей бд). Здесь обрабатываются запросы типа `UPDATE`.
- `public int bulkInsert(...)` - метод, позволяющий вставлять сразу пачку значений(запросы типа `INSERT`).
- `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 утилитарного класса `UriMatcher`, который проверяет `Uri` относительно некоторых шаблонов для `Uri` и возвращает соответствующий код.

В нашем случае, метод работает следующим образом - если в качестве `Uri` передаётся запись, в конце которой имя нашей таблички, то мы возвращаем тип множественный(то есть это запрос ко всем элементам таблички), а если через слеш записано какое-то число, то тип `Uri` - тип одной записи(число мы потом преобразуем в дату и вернём значение, соответствующее этой конкретной дате).

```

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 }
```

- 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.

Поподробнее о методах:

- `query`. Допустим пользователь вызвал `query`. Сначала мы определяем тип этого запроса по `Uri` - конкретная запись, конкретная запись с какими-то параметрами или группа записей. Затем вызываем соответствующие методы, в которых уже в зависимости от типа делаем запросы(`query`) к бд. В этой `query` `projection` задают колонки, которые мы хотим получить; `selection` - это условие, по которому мы выбираем; `selectionArgs` - конкретные значения для условия и т.д. Для того, чтобы прочесть что-нибудь из базы данных, мы используем метод `.getReadableDatabase()`.

```

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 {
14 long date = WeatherContract.WeatherEntry.getDateFromUri(uri);
15 String selection = WeatherContract.WeatherEntry.COLUMN_DATE + " = ? ";
16 String selectionArgs[] = new String[] { Long.toString(date) };
17
18 return mDbHelper.getReadableDatabase().query(WeatherContract.WeatherEntry.
19 TABLE_NAME,
20 projection,
21 selection,
22 selectionArgs,
23 null,
24 null,
25 sortOrder);
26 }
27
28 ...
29
30 @Override
31 public Cursor query(Uri uri, String[] projection, String selection, String[]
32 selectionArgs, String sortOrder) {
33 Cursor cursor;
34 switch (sUriMatcher.match(uri)) {
35 case WEATHER:
36 cursor = getWeather(projection, selection, selectionArgs, sortOrder
37);
38 break;
39 case WEATHER_WITH_DATE:
40 cursor = getWetherByDate(uri, projection, sortOrder);
41 break;
42 default:
43 throw new UnsupportedOperationException("Unknown uri: " + uri);
44 }
45 cursor.setNotificationUri(getContext().getContentResolver(), uri);
46 return cursor;
47 }

```

- insert. Чтобы записать что-либо в базу данных, мы используем метод `.getWritableDatabase()`. Принцип точно такой же - мы не пишем явно SQL-запрос, мы передаём только параметры.

```

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.
7 COLUMN_DATE);
8 long id = mDbHelper.getWritableDatabase().insert(
9 WeatherContract.WeatherEntry.TABLE_NAME, null, values);
10 if (id != -1) {
11 getContext().getContentResolver().notifyChange(uri, null);
12 return WeatherContract.WeatherEntry.buildWeatherUri(date);
13 } else {
14 throw new android.database.SQLException("Failed to insert row
15 into " + uri);
16 }
17 default:
18 throw new UnsupportedOperationException("Unknown uri: " + uri);
19 }
20 }

```

Хорошо бы после изменения чего-либо в базе данных оповестить тех, кто использует данные, об этом. Для этого есть метод `.notifyChange(uri, null)`, вызывающийся так:

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

Этому методу передаётся `Uri`, по которому мы что-то изменили.

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

Общая реализация `ContentProvider`: здесь

## 9 Fragments

- Весь UI в большинстве приложений расписан по фрагментам.

## 10 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, внутри 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`. Android 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.

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(adapter)`).

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 он отличал 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,
7 sortOrder);
8 }
```

Конструктору `CursorLoader` передается `Context`, ссылка на данные, указание на то, какие колонки вернуть, а также порядок их сортировки.

На самом деле здесь 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 }

```

## 11 Директория resources

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