1 Android Manifest

- Описание проекта, в том числе настройки и конфигурации, например версии.
- Прописаны permission и составляющие проекта.

2 Activity

- Первой в программе вызывается MainActivity(? не помню).
- Цикл жизни.
- Activity надо прописывать в манифесте.

3 BroadCastReceiver

??На один и тот же интент, на один и тот же тип интента может быть очень много. И они вызываются последовательно. Если там в какой-нибудь цепочке первый взял на час захватил процессор и работает упорно, все другие будут ждать оповещения об этом.??

BroadCastReceiver - класс-обработчик intent-ов, то есть класс-обработчик широковещательных сообщений. Он может быть подписан на несколько разных интентов. Если к нему приходит несколько интентов за раз, то они выстраиваются в цепочку и обрабатываются по-одному, поэтому есть определённые ограничения на работу BroadCastReceiver. Если вы попробуете обрабатывать интент больше 5 секунд, обработчик принудительно прикончат, чтобы другие интенты не ждали, пока их обработают. Из-за этой особенности количество действий, которое вы можете сделать с помощью BroadCastReceiver, довольно ограничено. Как правило, всё, что делают в BroadCastReceiver - это посылают другой интент, чтобы запустить Activity, Service(это такой класс, позже мы зачем он нужен и что это такое) или что-нибудь подобное.

Другое назначение BroadCastReceiver это получать системные оповещения. Есть некоторое количество системных оповещений, например о том, что андроид загрузился, или, например, у вас маленький заряд батарейки. Можно подписаться и на них и как-то среагировать.

Как и Activity, BroadCastReceiver должен быть прописан в андроид манифесте(на самом деле его можно зарегистрировать и программно, и иногда так и делают, но мы этого касаться не будем):

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

Теперь, надо как-то сказать приложению, на какие intent мы подписываем наш BroadCastReceiver. Вообще-то говоря, можно это сделать прямо в манифесте. Так обычно и делают, если вы подписываетесь на системные события. А можно сделать это програмно. Мы будем делать это 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);
```

Здесь я, когда создаётся Activity, получаю доступ к некоторому LocationManager, который вы уже видели раньше. Затем я создаю Intent ручками, где явно прописываю класс, куда интент должен быть отправлен. Это то, что называют explicit Intent. Вообще, интенты бывают implicit, а бывают explicit. Explicit интенты - это интенты, для которых мы явно прописываем, к какому классу его отправить, implicit - когда мы некоторым образом описываем, куда мы хотим передать сообщение, и потом уже сам андроид разбирается, кого надо запустить по этому описанию. В данном случае, мы при создании интента явно указываем класс приёмника. Соответсвенно, Intent ему и будет доставлен.

На самом деле, здесь отправляется не совсем интент, а то, что называется PendingIntent - наследник класса интент. Главное его отличие от обычного интента в том, что он учитывает права приложения.

Как мы уже говорили, вы можете запускать Activity(приложение) из другого приложения, но у этого Activity(приложения) может не быть прав на работу с какими-то там ресурсами. Если эти права нужны, вы можете послать PendingIntent и передать тем самым Activity(приложению), которое вы запускаете, свои права на исполнение. Вам не всегда это нужно, но некоторые API просто по умолчанию используют PendingIntent. Понятно, что в нашем случае BroadCastReceiver это часть нашего же приложения, и у него есть все те же самые права, поэтому PendingIntent как бы и не нужен. Но дело а том, что посылаем мы его с помощью LocationManager, а интерфейс LocationManager требует PendingIntent, поэтому отправляется здесь именно PendingIntent.

Когда приходит интент, который предназначен для данного BroadCastReceiver, вызывается метод onReceive(). Одним из его аргументов является экземпляр класса Context - базовый класс для частей приложения. Он предоставляет доступ ко всем андроидовским ресурсам. Через контест вы можете обращаться к ресурсам системы. К примеру, Activity - наследник класса Context. Поэтому, мы можем вызвать, к примеру, метод getSystemService(), определённый в классе Context - для доступа к какому-то сервису. Activity является наследником класса Context, поэтому этот метод вызывается без всякого префикса. А вот BroadCastReceiver не является наследником класса Context, но доступ ко всяким менеджерам вполне себе может понадобиться, поэтому-то контекст и передаётся методу onReceive().

В нашем Reciever мы не будем делать ничего такого, 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 андроид, завязан на несколько(может один) intent.
- Получает интент, к которому он привязан и быстро реагирует(с технической точки зрения при получении intent вызывается onReceive(), который должен быстро выполняться).
- Если один BroadCastReceiver соотетствует нескольким intent, то они построятся в цепочку и будут выполняться последовательно.
- Поэтому если по какому-то intent метод выполняется слишком долго, то его прикончат, 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 манифесте.

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

- Подписку на intent можно тоже записать в Manifest.
- А можно это сделать и програмно. Explicit intent - явно указываем класс, к которому intent отправить. Бывает ещё implicit intent. На самом деле здесь посылается не intent а наследник класса Intent - PendingIntent, в котором учитываются права приложения, из которого нас запустили.
- 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);
```

- Класс Context - это базовый класс для частей приложения. К примеру, Activity - наследник класса Context. Он предоставляет доступ ко всем андроидовским ресурсам. BroadCastReceiver не является наследником класса Context, поэтому он туда передаётся.

4 Log

Мы открываем наш DeviceMonitor. LogCat.

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

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

Усложним наш BroadCastReceiver - мы хотим при обновлении кординат качать прогноз погоды.

5 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"/>
```

6 Service

- Service также надо прописывать в манифесете:

```
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.

- Ключик exported задаёт, позволено ли другим приложениям запускать наш сервис, Activity и тд.
- Работает в фоновом режиме.
- В андроид есть класс Service, от которого можно отнаследоваться и написать свой сервис с блэкджеком. Но иногда нам нужно от сервиса только обрабатывать приходящие к нему интенты, тогда можно отнаследоваться от IntentService и просто определить метод onHandleIntent(Intent intent), который запускается, когда приходит очередной intent.
- Если к нему обращаются много intent, они просто становятся в очередь.
- Запускают intent к service через команду startService(...);
- Если нам достаточно обрабатывать один intent за раз, то IntentService; если нам надо обрабатывать много intent одновременно, то реализуем наследника обычного Service.

Что мы хотим сделать в нашем сервисе? Мы хотим достучаться по сети до сайта и скачать от туда прогноз погоды по текущим координатам. Сначала идёт куча разных параметров, описывающих разметку json-класса(этого мы касаться не будем). Потом, достаём из intent информацию о текущих координатах:

```
1 final String locationKey = LocationManager.KEY_LOCATION_CHANGED;
2 .....
3 Location location = (Location) intent.getExtras().get(locationKey);
```

.get(...) возвращает Object, поэтому нужно явно кастить к типу, который на нужен. Затем мы долго и страшно с помощью андроидовского Uri собираем Url. А затем работаем непосредственно с сетью. Для этого поймём, как работать с сетью, как подключиться к сайту.

7 Connection

- Самый простой способ работать с сетью - это HttpURLConnection.
- Возвращается методом .openConnection(). Осторожно, этот метод возвращает не HttpURLConnection, поэтому предварительно нужно скастить возвращаемое значение к HttpURLConnection:

```
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();
```

  Вместо "GET"может быть другая строчка, в зависимости от того, какой ваш запрос к серверу. В общем случае, описание метода надо смотреть в API того сервиса, которым вы пользуетесь.
- С данным типом соединения обычно происходит так - создание запроса, подключение, считывание информации из ответа(если нужно). Дальше соединение по большей части бесполезно. В конце нельзя забывать закрыть соединение:

```
1 connection.disconnect();
```

Пример считывания информации:

```
InputStream inputStream = connection.getInputStream();
StringBuilder buffer = new StringBuilder();
if (inputStream == null) {
    return;
}
reader = new BufferedReader(new InputStreamReader(inputStream));
String line;
while ((line = reader.readLine()) != null) {
    buffer.append(line);
    buffer.append("\n");
}

if (buffer.length() == 0) {
    return;
}
```

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. Соответсвенно, 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:

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

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.
- 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 вегда 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, соотетсвующей 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 + "/
                       " + 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 }
```

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 обёрнутым 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 новый таблички, к примеру.

  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(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             WeatherEntry._ID + " INTEGER PRIMARY KEY AUTOINCREMENT," +
14             WeatherEntry.COLUMN_DATE + " INTEGER NOT NULL," +
15             WeatherEntry.COLUMN_DAY_TEMP + " REAL NOT NULL," +
16             WeatherEntry.COLUMN_NIGHT_TEMP + " REAL NOT NULL," +
17             WeatherEntry.COLUMN_WIND_SPEED + " REAL NOT NULL," +
18             WeatherEntry.COLUMN_DEGREES + " REAL NOT NULL," +
19             WeatherEntry.COLUMMN_WEATHER_ID + " INTEGER NOT NULL, " +
20                 "UNIQUE (" + WeatherEntry.COLUMN_DATE + ") ON CONFLICT REPLACE);";
21
22         db.execSQL(SQL_CREATE_WEATHER_TABLE);
23     }
24
25     @Override
26     public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
27         db.execSQL("DROP TABLE IF EXISTS " + WeatherEntry.TABLE_NAME);
28         onCreate(db);
29     }
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 соответсвующий 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, соответсвующее 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 соответсвующие методы, в которых уже в зависимости от типа делаем запросы(query) к бд. В этой query projection задают колонки, которые мы котим получить; selection - это условие, по которому мы выбираем; selectionArgs - конкретные значения для условия и т.д. Для того, чтобы прочитать 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     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 }
25             ...
26 @Override
27 public Cursor query(Uri uri, String[] projection, String selection, String[]
       selectionArgs, String sortOrder) {
28     Cursor cursor;
29     switch (sUriMatcher.match(uri)) {
30         case WEATHER:
31             cursor = getWeather(projection, selection, selectionArgs, sortOrder
                   );
32             break;
33         case WEATHER_WITH_DATE:
34             cursor = getWetherByDate(uri, projection, sortOrder);
35             break;
36         default:
37             throw new UnsupportedOperationException("Unknown uri: " + uri);
38     }
39     cursor.setNotificationUri(getContext().getContentResolver(), uri);
40     return cursor;
41 }
```

- 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.
                    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  }
```

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: тык

9 Fragments

- Весь UI в большинстве приложений распихан по фрагментам.

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.

10 ListView

- Для того, чтобы рисовать списки, используют ListView.

11 Adapter

- 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.

```
@Override
public View newView(Context context, Cursor cursor, ViewGroup parent) {
    View view = LayoutInflater.from(context).inflate(R.layout.
       forecast_list_item_view,
            parent, false);
    view.setTag(new ViewHolder(view));
    return view;
}
```

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

  Пример xml-описания:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:gravity="center_vertical"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="20dp">

    <LinearLayout
        android:orientation="vertical"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1">

        <TextView
            android:id="@+id/date"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="10dp"/>

        <TextView
            android:id="@+id/wind"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="10dp"/>

    </LinearLayout>

    <LinearLayout
        android:orientation="vertical"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:gravity="right">

        <ImageView
            android:id="@+id/icon"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="10dp"/>

```

```
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 - это картиночка.

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. 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.