Разработка приложений под Android предполагает использование некоторых базовых компонентов, из которых и состоит приложение. Ключевыми компонентами в данном случае являются классы android.content.Context и android.app.Activity.

Класс контекста используется для управления специфичными для приложения ресурсами и конфигурацией. Так, с помощью класса Conetxt мы можем получить различные строковые ресурсы, графические ресурсы, получить доступ к системным сервисам, управлять базами данных SQLite, управлять файлами и каталогами приложения

Для получения текущего контекста приложения мы можем воспользоваться методом **getApplicationContext()**:

|  |  |
| --- | --- |
| 1 | Context context = getApplicationContext(); |

Для получения ресурсов в приложении в классе Context определен метод getResources(). Например, получим строковый ресурс hello\_world, который определен в файле *res/values/strings.xml*:

|  |  |
| --- | --- |
| 1 | String hello\_world= context.getResources().getString(R.string.hello\_world); |

Но в большинстве случае класс контекста не потребуется, поскольку в реальном приложении мы больше будет обращаться к функционалу класса Activity, который наследуется от класса Conetxt. Нередко activity ассоциируется с отдельным экраном или окном приложения, а переключение между окнами будет происходить как перемещение от одной activity к другой. Приложение может иметь одну или несколько activity.

**Жизненный цикл приложения**

Все приложения Android имеют строго определенный системой жизненный цикл. При запуске пользователем приложения система дает этому приложению высокий приоритет. Каждое приложение запускается в виде отдельного процесса, что позволяет системе давать одним процессам более высокой приоритет, в отличие от других. Благодаря этому, например, при работе с одними приложениями не блокировать входящие звонки. После прекращения работы с приложением, система освобождает все связанные ресурсы и переводит приложение в разряд низкоприоритетного и закрывает его.

Все объекты activity, которые есть в приложении, управляются системой в виде стека activity, который называется **back stack**. При запуске новой activity она помещается поверх стека и выводится на экран устройства, пока не появится новая activity. Когда текущая activity заканчивает свою работу (например, пользователь уходит из приложения), то она удаляется из стека, и возобновляет работу та activity, которая ранее была второй в стеке.

Каждая activity имеет три состояния:



* **Активно (Resumed)**: activity отображается на экране и взаимодействует с пользователем
* **Приостановлено (paused)**: activity теряет фокус, однако по-прежнему остается видимой и продолжает работать. А в ранг активного переводится другая activity, которая может быть либо невидимой, либо занимать весь экран. Приостановленная activity в то же время может быть завершена системой в случае, если другим activity и приложениям с более высоким приоритетом будет не хватать памяти.
* **Остановлено (stopped)**: activity сохраняет свое состояние, однако она уже невидима пользователю, и если другим процессам потребуется память, то activity будет завершена.

При переходе активности от одного состояния к другому, она получает уведомления через защищенные методы:

* protected void onCreate();
* protected void onStart();
* protected void onRestart();
* protected void onResume();
* protected void onPause();
* protected void onStop();
* protected void onDestroy()

Семь перечисленных методов определяют весь жизненный цикл активности. Есть три вложенных цикла, которые вы можете отслеживать в классе активности:

* полное время жизни (***Full Lifetime***) — время с момента первого вызова метода **onCreate()** до вызова **onDestroy()**. Активность делает всю начальную установку своего глобального состояния в методе **onCreate()** и освобождает все остающиеся ресурсы в **onDestroy()**. Например, если активность порождает дополнительный поток, выполняющийся в фоновом режиме, можно создать этот поток в методе **onCreate()** и затем остановить поток в методе **onDestroy()**;
* видимое время жизни (***Visible Lifetime***) — время между вызовом метода **onStart()** и вызовом **onStop()**. В это время пользователь может видеть окно активности на экране, хотя окно может не быть на переднем плане и может не взаимодействовать с пользователем. Между этими двумя методами вы можете поддерживать в коде ресурсы, которые необходимы, чтобы отображать активность пользователю;
* активное время жизни (***Active Lifetime***) — время между вызовами **onResume()** и **onPause()**. В это время окно активности находится на переднем плане и взаимодействует с пользователем. Активность в процессе работы приложения может часто переходить между состояниями *active* и *paused*, поэтому код в этих двух методах должен быть или небольшим по объему (чтобы не замедлять работу приложения во время выполнения), или порождать дополнительные потоки, если требуется выполнение задач, занимающих длительное время.Можно написать код с заглушками для методов внутри Активности, которые обрабатывают изменения состояний. Комментарии к каждой такой заглушке описывают действия, которые нужно учитывать при обработке этих событий.

После перевода activity в приостановленное или остановленное состояние система может удалить activity из памяти, либо уведомив о завершении процесса, либо просто убив соответствующий процесс. При последующем запуске и отображении activity полностью пересоздается и начинает свою работу заново.

После запуска activity проходит через ряд событий, которые обрабатываются системой и для обработки которых существует ряд обратных вызовов:

protected void OnCreate(Bundle saveInstanceState);

protected void OnStart();

protected void OnRestart();

protected void OnResume();

protected void OnPause();

protected void OnStop();

protected void OnDestroy();

Схематично взаимосвязь между всеми этими обратными вызовами можно представить следующим образом

При создании новой активности, например, при запуске приложения, Android вызывает метод OnCreate. В этом методе производится первоначальная настройка activity. В частности, создаются объекты визуального интерфейса. Этот метод получает объект Bundle, который содержит прежнее состояние activity, если оно было сохранено. Если activity заново создается, то данный объект имеет значение null. Если же activity уже ранее была создана, но находилась в приостановленном состоянии, то bundle содержит связанную с activity информацию.

Затем вызывается метод OnStart, а activity переходит в "видимое" состояние. А при вызове метода OnResume activity отображается на экране, и пользователь может с ней взаимодействовать.

Если пользователь решит перейти к другой активности, то система вызывает метод OnPause. После этого, если пользователь решит вернуться к прежней активности, то система вызовет снова метод OnResume, и activity снова появится на экране. Иначе, если activity больше невидима, то вызывается метод OnStop.

Если после вызова метода OnStop пользователь решит вернуться к прежней activity, тогда система вызовет метод OnRestart. Ну и завершается работа активности вызовом метода OnDestroy, который возникает либо, если система решит убить activity, либо при вызове метода finish().

Также следует отметить, что при изменении ориентации экрана система завершает activity и затем создает ее заново.

Мы можем управлять этими событиями жизненного цикла, переопределив соответствующие методы. Так, возьмем activity нашего проекта и изменим ее следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | package com.metanit.eugene.helloapplication;    import android.os.Bundle;  import android.support.v7.app.AppCompatActivity;  import android.util.Log;      public class MainActivity extends AppCompatActivity {        private final static String TAG = MainActivity.class.getSimpleName();        protected void onCreate(Bundle savedInstanceState) {          super.onCreate(savedInstanceState);          setContentView(R.layout.activity\_main);          Log.d(TAG, "Create");      }      @Override      protected void onDestroy(){          super.onDestroy();          Log.d(TAG, "Destroy");      }      @Override      protected void onStop(){          super.onStop();          Log.d(TAG, "Stop");      }      @Override      protected void onStart(){          super.onStart();          Log.d(TAG, "Start");      }      @Override      protected void onPause(){          super.onPause();          Log.d(TAG, "Pause");      }      @Override      protected void onResume(){          super.onResume();          Log.d(TAG, "Resume");      }        @Override      protected void onRestart(){          super.onRestart();          Log.d(TAG, "Restart");      }  } |

В данном случае обрабатываются все ключевые методы жизненного цикла. Вся обработка сведена к вызову метода Log.d(), в который передается TAG - случайное строковое значение и строка, которая выводится в консоли logcat, выполняя роль отладочной информации.

И при запуске приложения мы сможем увидеть в окне **logcat** отладочную информацию, которая определяется в методах жизненного цикла activity: