**3.1 Введение**

В ["Введение в разработку мобильных приложений"](http://www.intuit.ru/studies/courses/12643/1191/lecture/21980)были рассмотрены общие вопросы, связанные с операционной системой *Android*, а также инструменты, используемые для разработки мобильных приложений. Изучены основы разработки интерфейсов мобильных приложений. В данной теме обсуждаются вопросы, связанные, непосредственно, с разработкой мобильных приложений для устройств, работающих под управлением *Android*.

Для начала предполагается рассмотреть еще несколько общих вопросов: во-первых, какие виды мобильных приложений существуют и каковы особенности каждого вида; во-вторых, как организовано *исполнение* приложений в ОС *Android* и каким образом обеспечивается безопасная среда их функционирования. Понимание этих вопросов позволяет вести более осознанную разработку приложений.

Невозможно создать осмысленное *приложение*, не изучив внутреннюю организацию, свойственную приложениям, работающим на определенной платформе. В данном курсе, очевидно, необходимо изучить структуру и основные компоненты приложений, разрабатываемых для работы на смартфонах под управлением ОС *Android*. От типа мобильного устройства внутренняя организация приложений не зависит, т. е.*Android*-приложения, разработанные для смартфонов вполне смогут выполняться и на планшетах. В данной лекции рассматривается*архитектура* *Android* приложений, основанная на идее многократного использования компонентов, которые являются основными строительными блоками. Подробно описываются основные компоненты, а также такие важные понятия для мобильных приложений, работающих под управлением *Android*, как *манифест* приложения и ресурсы.

**3.2 Основные виды Android-приложений**

Приступая к разработке мобильных приложений хорошо бы иметь *представление* о том, какие виды приложений существуют. Дело в том, что если удастся определить к какому типу относится *приложение*, то становится понятнее на какие моменты в процессе его разработки необходимо обращать основное внимание. Можно выделить следующие виды приложений:

* **Приложения переднего плана** выполняют свои функции только, когда видимы на экране, в противном же случае их выполнение приостанавливается. Такими приложениями являются, например, игры, текстовые редакторы, видеопроигрыватели. При разработке таких приложений необходимо очень внимательно изучить жизненный цикл активности, чтобы переключения в фоновый режим и обратно проходили гладко (бесшовно), т. е. при возвращении приложения на передний план было незаметно, что оно вообще куда-то пропадало. Для достижения этой гладкости необходимо следить за тем, чтобы при входе в фоновый режим приложение сохраняло свое состояние, а при выходе на передний план восстанавливало его. Еще один важный момент, на который обязательно надо обратить внимание при разработке приложений переднего плана, удобный и интуитивно понятный интерфейс1.
* **Фоновые приложения** после настройки не предполагают взаимодействия с пользователем, большую часть времени находятся и работают в скрытом состоянии. Примерами таких приложений могут служить, службы экранирования звонков, SMS-автоответчики. В большинстве своем фоновые приложения нацелены на отслеживание событий, порождаемых аппаратным обеспечением, системой или другими приложениями, работают незаметно. Можно создавать совершенно невидимые сервисы, но тогда они будут неуправляемыми. Минимум действий, которые необходимо позволить пользователю: санкционирование запуска сервиса, настройка, приостановка и прерывание его работы при необходимости.
* **Смешанные приложения** большую часть времени работают в фоновом режиме, однако допускают взаимодействие с пользователем и после настройки. Обычно взаимодействие с пользователем сводится к уведомлению о каких-либо событиях. Примерами таких приложений могут служить мультимедиа-проигрыватели, программы для обмена текстовыми сообщениями (чаты), почтовые клиенты. Возможность реагировать на пользовательский ввод и при этом не терять работоспособности в фоновом режиме является характерной особенностью смешанных приложений. Такие приложения обычно содержат как видимые активности, так и скрытые (фоновые) сервисы, и при взаимодействии с пользователем должны учитывать свое текущее состояние. Возможно потребуется обновлять графический интерфейс, если приложение находится на переднем плане, или же посылать пользователю уведомления из фонового режима, чтобы держать его в курсе происходящего. И эти особенности необходимо учитывать при разработке подобных приложений.
* **Виджеты** - небольшие приложения, отображаемые в виде графического объекта на рабочем столе. Примерами могут служить, приложения для отображения динамической информации, такой как заряд батареи, прогноз погоды, дата и время. Разумеется, сложные приложения могут содержать элементы каждого из рассмотренных видов. Планируя разработку приложения, необходимо определить способ его использования, только после этого приступать к проектированию и непосредственно разработке.

**3.3 Безопасность**

Обратим внимание на организацию исполнения приложений в ОС *Android*. Как уже было отмечено приложения под *Android* разрабатываются на языке программирования *Java*, компилируется в *файл* с расширением .apk, после этот *файл* используется для установки приложения на устройства, работающие под управлением *Android*. После установки каждое *Android* *приложение* "живет" в своей собственной безопасной "песочнице", рассмотрим, как это выглядит:

* операционная система Android является многопользовательской ОС, в которой каждое приложение рассматривается как отдельный пользователь;
* по умолчанию, система назначает каждому приложению уникальный пользовательский ID, который используется только системой и неизвестен приложению;
* система устанавливает права доступа ко всем файлам приложения следующим образом: доступ к элементам приложения имеет только пользователь с соответствующим ID;
* каждому приложению соответствует отдельный Linux процесс, который запускается, как только это необходимо хотя бы одному компоненту приложения, процесс прекращает работу, когда ни один компонент приложения не использует его или же системе требуется освободить память для других (возможно, более важных) приложений;
* каждому процессу соответствует отдельный экземпляр виртуальной машины Dalvik, в связи с этим код приложения исполняется изолировано от других приложений.

Перечисленные идеи функционирования приложения в ОС *Android* реализуют принцип минимальных привилегий, т. е. каждому приложению, по умолчанию, разрешен *доступ* только к компонентам, необходимым для его работы и никаким больше. Таким образом обеспечивается очень безопасная среда функционирования приложений.

Однако, в случае необходимости приложения могут получить *доступ* к данным других приложений и системным сервисам (услугам). В случае, когда двум приложениям необходимо иметь *доступ* к файлам друг друга, им присваивается один и тот же пользовательский *ID*. Для экономии системных ресурсов такие приложения запускаются в одном Linux процессе и делят между собой один и тот же экземпляр виртуальной машины, в этом случае приложения также должны быть подписаны одним сертификатом. В случае же, когда приложению требуется *доступ* к системным данным, например, контактам, *SMS* сообщениям, картам памяти, камере, Bluetooth и т. д., пользователю необходимо дать приложению такие полномочия во время установки его на устройство.

**3.4 Архитектура приложения, основные компоненты**

Вот и пришла пора поговорить непосредственно о внутренней организации приложений под *Android*: обсудить их архитектуру и основные компоненты.

*Архитектура* *Android* приложений основана на идее многократного использования компонентов, которые являются основными строительными блоками. Каждый *компонент* является отдельной сущностью и помогает определить общее *поведение приложения*.

Система *Android* выстроена таким образом, что любое *приложение* может запускать необходимый *компонент* другого приложения. Например, если *приложение* предполагает использование камеры для создания фотографий, совершенно необязательно создавать в этом приложении*активность* для работы с камерой. Наверняка на устройстве уже есть *приложение* для получения фотографий с камеры, достаточно запустить соответствующую *активность*, сделать фотографию и вернуть ее в *приложение*, так что *пользователь* будет считать, что камера часть приложения, с которым он работает.

Когда система запускает *компонент*, она запускает процесс приложения, которому принадлежит *компонент*, если он еще не запущен, и создает экземпляры классов, необходимых компоненту. Поэтому в отличие от большинства других систем, в системе *Android* приложения не имеют единой точки входа (нет метода main(), например). В силу запуска каждого приложения в отдельном процессе и ограничений на *доступ* к файлам, *приложение* не может напрямую активировать *компонент* другого приложения. Таким образом для активации компонента другого приложения необходимо послать системе сообщение о намерении запустить определенный *компонент*, система активирует его.

Можно выделить четыре различных типа компонентов, каждый тип служит для достижения определенной цели и имеет свой особый*жизненный цикл*, который определяет способы создания и разрушения соответствующего компонента. Рассмотрим основные компоненты*Android*-приложений.

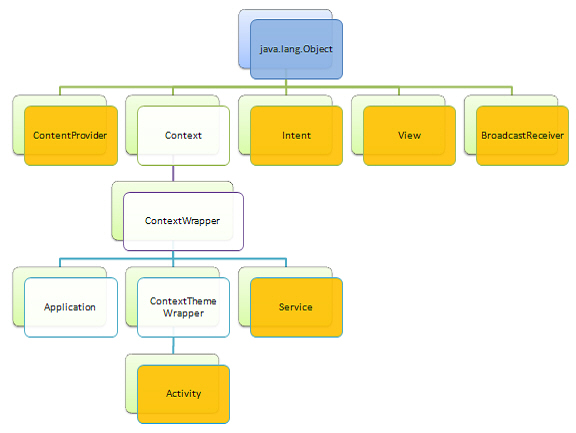
**Активности (Activities).** *Активность* - это видимая часть приложения (экран, окно, форма), отвечает за *отображение* графического интерфейса пользователя. При этом *приложение* может иметь несколько активностей, например, в приложении, предназначенном для работы с электронной почтой, одна *активность* может использоваться для отображения списка новых писем, другая *активность* - для написания, и еще одна - для чтения писем. Несмотря на то, что для пользователя *приложение* представляется единым целым, все активности приложения не зависят друг от друга. В связи с этим любая из этих активностей может быть запущена из другого приложения, имеющего *доступ* к активностям данного приложения. Например, *приложение* камеры может запустить *активность*, создающую новые письма, чтобы отправить только что сделанную фотографию адресату, указанному пользователем.

**Сервисы (Services).** Сервис - *компонент*, который работает в фоновом режиме, выполняет длительные по времени *операции* или работу для удаленных процессов. Сервис не предоставляет пользовательского интерфейса. Например, сервис может проигрывать музыку в фоновом режиме, пока *пользователь* использует другое *приложение*, может загружать данные из сети, не блокируя взаимодействие пользователя с активностью. Сервис может быть запущен другим компонентом и после этого работать самостоятельно, а может остаться связанным с этим компонентом и взаимодействовать с ним.

**Контент-провайдеры (Content providers).** Контент-провайдер управляет распределенным множеством данных приложения. Данные могут храниться в файловой системе, в базе данных SQLite, в сети, в любом другом доступном для приложения месте. Контент-провайдер позволяет другим приложениям при наличии у них соответствующих прав делать запросы или даже менять данные. Например, в системе *Android* есть контент-провайдер, который управляет информацией о контактах пользователя. В связи с этим, любое *приложение* с соответствующими правами может сделать *запрос* на чтение и *запись* информации какого-либо контакта. Контент-провайдер может быть также полезен для чтения и записи приватных данных приложения, не предназначенных для доступа извне.

**Приемники широковещательных сообщений (Broadcast Receivers).** Приемник - *компонент*, который реагирует на широковещательные извещения. Большинство таких извещений порождаются системой, например, извещение о том, что экран отключился или низкий заряд батареи. Приложения также могут инициировать *широковещание*, например, разослать другим приложениям сообщение о том, что некоторые данные загружены и доступны для использования. Хотя приемники не отображают пользовательского интерфейса, они могут создавать уведомление на панели состояний, чтобы предупредить пользователя о появлении сообщения. Такой приемник служит проводником к другим компонентам и предназначен для выполнения небольшого объема *работ*, например, он может запустить соответствующий событию сервис.

Все рассмотренные компоненты являются наследниками классов, определенных в *Android* *SDK*.

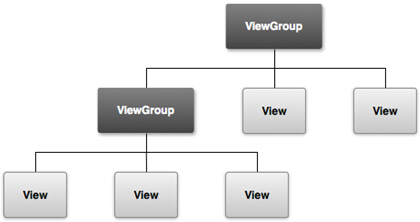


**Рис. 3.1.**Иерархия классов Android SDK

(источник: <http://habrahabr.ru/post/141201/>)

На [рис. 3.1](http://www.intuit.ru/studies/courses/12643/1191/lecture/21983?page=2#image.3.1) показана *иерархия* основных классов *Android* *SDK*, с которыми обычно имеет дело разработчик. На самом деле классов намного больше, желтым цветом выделены классы, с которыми разработчик работает непосредственно, наследует от них свои классы. Остальные классы не менее важны, но они реже используются напрямую. Для начала рассмотрим классы Intent и View.

*Класс* View является основным строительным блоком для компонентов пользовательского интерфейса (UI), он определяет прямоугольную область экрана и отвечает за прорисовку и обработку событий. Является базовым классом для виджетов (*GUI* widgets), которые используются для создания интерактивных компонентов пользовательского интерфейса: кнопок, текстовых полей и т. д. А также является базовым классом для класса ViewGroup, который является невидимым контейнером для других контейнеров и виджетов, определяет свойства расположения компонентов пользовательского интерфейса. *Интерфейс* *Android*-приложения представляет собой иерархию UI компонентов (см. [рис. 3.2](http://www.intuit.ru/studies/courses/12643/1191/lecture/21983?page=2#image.3.2)), можно описать эту иерархию программно, но более простым и эффективным способом задать расположение элементов интерфейса является*XML* *файл*, который предоставляет удобную для восприятия структуру компоновки (*layout* file). Во время исполнения *XML* *файл* автоматически превращается в *дерево* соответствующих объектов. Подробнее о классе View, свойствах и методах: <http://developer.android.com/reference/android/view/View.html>.



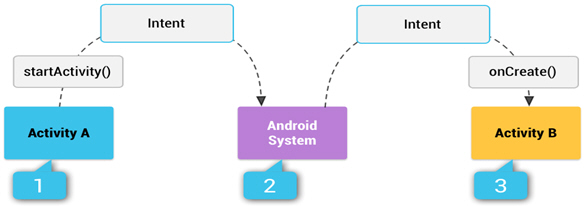
**Рис. 3.2.**Иерархия компонентов, определяющая компоновку интерфейса пользователя

Объекты-экземпляры класса Intent используются для передачи сообщений между основными компонентами приложений. Известно, что три из четырех основных компонентов: активности, сервисы и приемники широковещательных сообщений, могут быть активированы с помощью сообщений, которые называются намерениями. Такие сообщения являются инструментом позднего связывания компонентов одного или нескольких приложений. Экземпляр класса Intent представляет собой структуру данных, содержащую описание *операции*, которая должна быть выполнена, и обычно используется для запуска активности или сервиса. В случае с приемниками широковещательных сообщений *объект*Intent содержит описание события, которое произошло или было объявлено.

Для каждого типа компонентов существуют свои *механизмы* передачи намерений.

* Чтобы запустить активность или вызвать у работающей активности новое действие, необходимо передать объект-намерение в метод Context.startActivity() или Activity.startActivityForResult().
* Чтобы запустить сервис или доставить новые инструкции работающему сервису, необходимо передать объект-намерение в метод Context.startService(). Также объект-намерение может быть передан в метод Context.bindService(), чтобы связать между собой вызывающий компонент и сервис.
* Чтобы доставить объект-намерение всем заинтересованным приемникам широковещательных сообщений, необходимо передать его в любой из широковещательных методов: Context.sendOrderedBroadcast(), Context.sendStickyBroadcast(),Context.sendBroadcast().

В каждом случае система *Android* в ответ на намерение находит соответствующий *компонент*: *активность*, сервис или множество широковещательных приемников и запускает его если необходимо. В этой системе сообщений не случается накладок: сообщение-намерение, отправленное определенному компоненту, будет получено именно этим компонентом и никем другим.



**Рис. 3.3.**Передача намерений (Intent)

На [рис. 3.3](http://www.intuit.ru/studies/courses/12643/1191/lecture/21983?page=2#image.3.3) можно увидеть как происходит передача намерений (Intent), в данном случае одна *активность* запускает другую. [[6](http://www.intuit.ru/studies/courses/12643/1191/literature#literature.6)] *Активность* А создает намерение (Intent) с описанием действия и передает его в метод startActivity(). [[7](http://www.intuit.ru/studies/courses/12643/1191/literature#literature.7)] Система *Android* проверяет все приложения на совпадение с намерением, когда совпадение найдено, [[8](http://www.intuit.ru/studies/courses/12643/1191/literature#literature.8)] система запускает соответствующую *активность*, для чего вызывает методonCreate() и передает в него *объект*-намерение Intent.

Подробнее о классе Intent:

|  |
| --- |
| [http://developer.android.com/guide/components/intents-filters.html](http://developer.android.com/guide/components/intents-filters.html" \t "_blank), |
| <http://developer.android.com/reference/android/content/Intent.html> |

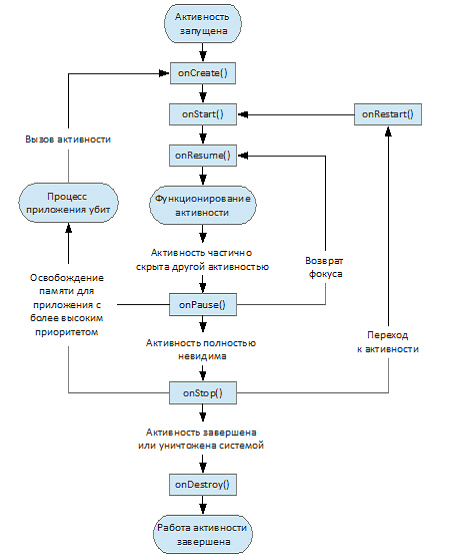
Пришло время более серьезно рассмотреть основные компоненты: активности, сервисы, контент-провайдеры, приемники широковещательных сообщений. В первую *очередь* нас будет интересовать *жизненный цикл* этих компонентов. Что же такое этот *жизненный цикл*? *Жизненный цикл*можно рассматривать, как процесс функционирования компонента: начиная с момента создания и запуска, включая *активный* и неактивный периоды работы, и, заканчивая уничтожением и освобождением ресурсов.

#### 3.4.1 Активности (Activities)

Активность - окно, несущее графический интерфейс пользователя. Окно активности обычно занимает весь экран устройства, однако вполне возможно создавать полупрозрачные или плавающие диалоговые окна. Мобильные приложения обычно являются многооконными, т. е. содержат несколько активностей, по одной на каждое окно. Одна из активностей определяется как "главная", и именно ее пользователь видит при первом запуске приложения.

Каждый экран приложения является наследником класса Activity. Для создания активности необходимо создать класс-наследник класса Activity напрямую или через любого его потомка. В этом классе необходимо реализовать все методы, вызываемые системой для управления жизненным циклом активности. Таких методов семь:

|  |  |
| --- | --- |
| onCreate() | - метод, вызываемый системой при создании активности. В реализации метода необходимо инициализировать основные компоненты активности и в большинстве случаев вызвать метод setContentView() для подключения соответствующего XML-файла компоновки (layout file). После метода onCreate() всегда вызывается метод onStart(). |
| onRestart() | - метод, вызываемый системой при необходимости запустить приостановленную активность. После этого метода всегда вызывается метод onStart(). |
| onStart() | - метод, вызываемый системой непосредственно перед тем, как активность станет видимой для пользователя. После этого метода вызывается onResume(). |
| onResume() | - метод, вызываемый системой непосредственно перед тем, как активность начнет взаимодействовать с пользователем. После этого метода всегда вызывается onPause(). |
| onPause() | - метод, вызываемый системой при потере активностью фокуса. В этом методе необходимо фиксировать все изменения, которые должны быть сохранены за пределами текущей сессии. После этого метода вызывается onResume(), если активность вернется на передний план, или onStop(), если активность будет скрыта от пользователя. |
| onStop() | - метод, вызываемый системой, когда активность становится невидимой для пользователя. После этого метода вызывается либо onRestart(), если активность возвращается к взаимодействию с пользователем, либо onDestroy(), если активность уничтожается. |
| onDestroy() | - метод, вызываемый системой перед уничтожением активности. Этот метод вызывается либо когда активность завершается, либо когда система уничтожает активность, чтобы освободить ресурсы. Можно различать эти два сценария с помощью метода isFinishing(). Это последний вызов, который может принять активность. |



**Рис. 3.4.**Жизненный цикл активности

(источник: <http://developer.android.com/guide/components/activities.html>).

При реализации вышеперечисленных методов первым делом всегда необходимо вызывать соответствующий метод предка.

Рассмотренные методы определяют жизненный цикл активности. На [рис. 3.4](http://www.intuit.ru/studies/courses/12643/1191/lecture/21983?page=3#image.3.4) можно увидеть пути, по которым активность может переходить из одного состояния в другое. В прямоугольниках указаны методы, которые вызываются при смене состояний активности.

Фактически активность может существовать в одном из трех состояний:

* **Выполняется (running).** Активность находится на переднем плане и удерживает фокус ввода. Если внимательно рассмотреть [рис. 3.4](http://www.intuit.ru/studies/courses/12643/1191/lecture/21983?page=3#image.3.4)можно заметить, что в это состояние активность попадает после вызова метода onResume(). Пока активность находится в этом состоянии ее процесс не может быть уничтожен системой.
* **Приостановлена.** Активность частично видима, однако фокус ввода потерян. В это состояние активность попадает после вызова методаonPause() ([рис. 3.4](http://www.intuit.ru/studies/courses/12643/1191/lecture/21983?page=3#image.3.4)). В этом состоянии активность поддерживается в "боевой готовности", т.е. в любой момент может получить фокус ввода и стать активной. Однако в этом состоянии процесс активности может быть уничтожен системой, в случае экстремальной нехватки памяти.
* **Остановлена.** Активность полностью невидима. В это состояние активность попадает после вызова метода onStop()([рис. 3.4](http://www.intuit.ru/studies/courses/12643/1191/lecture/21983?page=3#image.3.4)). В этом состоянии активность может быть "вызвана к жизни", она сохраняет все состояния и необходимую для восстановления информацию, однако процесс активности может быть уничтожен, если память понадобится для других целей.

#### 3.4.2 Сервисы (Services)

Сервис (Service) является компонентом приложения, предназначенным для выполнения длительных операций в фоновом режиме. Существует два способа существования сервисов:

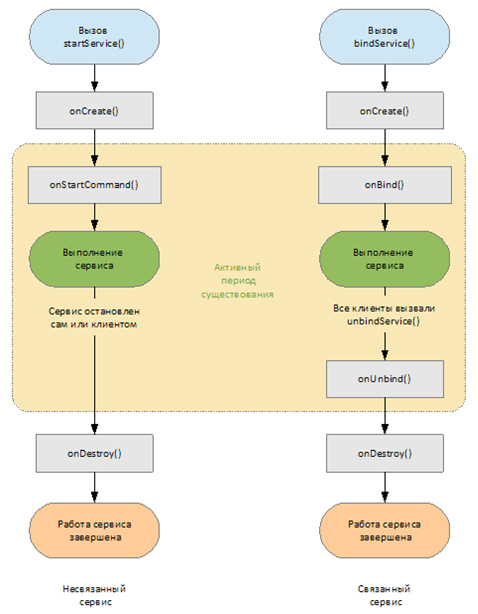
* первый заключается в том, что сервис запущен (started) и работает самостоятельно в фоновом режиме, так он может работать неопределенно долго, пока не выполнит свою задачу;
* второй заключается в том, что сервис привязан (bound) к некоторому компоненту или нескольким компонентам, в этом случае сервис предлагает интерфейс для взаимодействия с компонентом и работает пока привязан хотя бы к одному компоненту, как только связь со всеми компонентами разрывается сервис завершает свою работу.

Для создания сервиса необходимо создать класс-наследник класса Service напрямую или через любого его потомка. При этом в реализации класса необходимо переопределить (т. е. написать свою реализацию) некоторые методы, управляющие ключевыми аспектами жизненного цикла сервиса и обеспечивающие механизм связывания компонентов с сервисом, в соответствующем случае. Рассмотрим наиболее важные методы требующие реализации при создании сервиса.

|  |  |
| --- | --- |
| onStartCommand() | - метод, вызываемый системой, когда некоторый компонент, например активность, вызывает метод startService(). В этом случае сервис запускается и может работать в фоновом режиме неопределенно долго, поэтому необходимо позаботиться об остановке сервиса, когда он выполнит свою работу. Для остановки сервиса используется методstopSelf() в случае, когда сервис сам прекращает свою работу, или stopService() в случае, когда работу сервиса прекращает некоторый компонент. Нет необходимости писать реализацию метода onStartCommand(), если не предполагается самостоятельной работы сервиса (т. е. он будет работать только в связке с некоторыми компонентами). |
| onBind() | - метод, вызываемый системой, когда некоторый компонент желает привязать к себе сервис и вызывает метод bindService(). Этот метод должен возвращать реализацию интерфейса IBinder, которая может быть использована компонентом-клиентом для взаимодействия с сервисом. Метод onBind() необходимо реализовать в любом случае, но, если не предполагается связывания сервиса с какими-либо компонентами, возвращаемое значение должно быть равным null. |

Необходимо отметить, что сервис может быть запущен как самостоятельная единица, а в последствии может быть привязан к некоторым компонентам. В этом случае в сервисе должны быть обязательно реализованы оба метода onStartCommand() и onBind().

|  |  |
| --- | --- |
| onCreate() | - метод, вызываемый системой, при первом обращении к сервису для выполнения первоначальных настроек. Этот метод вызывается до вызова методов onStartCommand() и/или onBind(). |
| onDestroy() | - метод, вызываемый системой, когда сервис либо выполнил все действия, для которых создавался, либо больше не связан ни с одним компонентом, т. е. его услуги больше не требуются. В реализации этого метода необходимо предусмотреть освобождение всех ресурсов, таких как потоки, зарегистрированные слушатели, приемники и т. д. Вызов этого метода является последними вызовом, который может получить сервис. |



**Рис. 3.5.**Жизненный цикл сервиса

(источник: <http://developer.android.com/guide/components/services.html>).

На [рис. 3.5](http://www.intuit.ru/studies/courses/12643/1191/lecture/21983?page=3#image.3.5) показан жизненный цикл сервиса, левая диаграмма показывает жизненный цикл самостоятельного сервиса, правая - жизненный цикл сервиса, привязанного к некоторым компонентам. На рисунке хорошо видно, что жизненный цикл сервиса намного проще жизненного цикла активности. Однако для разработчика понимание того, как именно сервис создается, запускается и завершает свою работу, может оказаться даже более важным, т. к. сервис работает в фоновом режиме и пользователь может и не осознавать, что в некоторых случаях он имеет дело с работой сервисов.

Android принудительно останавливает работу сервисов только, когда ресурсов системы не хватает для активности, которая работает в данный момент на переднем плане. Приоритет работающих сервисов всегда выше, чем у приостановленных или полностью невидимых активностей, а если сервис привязан к выполняющейся активности, то его приоритет еще выше. С другой стороны, со временем приоритет самостоятельно работающего сервиса понижается и его шансы быть принудительно остановленным системой в случае нехватки ресурсов повышаются. В связи с этим имеет смысл проектировать сервис таким образом, чтобы через некоторое время он требовал у системы перезапуска. В случае если система все таки экстренно завершила работу сервиса, она перезапустит его как только освободятся ресурсы.

Подробнее о создании, использовании и удалении сервисов:

|  |
| --- |
| <http://developer.android.com/guide/components/services.html>; |
| <http://developer.android.com/guide/components/processes-and-threads.html>. |

#### 3.4.3 Контент-провайдеры (Content Providers)

Контент-провайдер управляет доступом к хранилищу данных. Для реализации провайдера в Android приложении должен быть создан набор классов в соответствии с манифестом приложения. Один из этих классов должен быть наследником класса ContentProvider, который обеспечивает интерфейс между контент-провайдером и другими приложениями. Основное назначение этого компонента приложения заключается в предоставлении другим приложениям доступа к данным, однако ничто не мешает в приложении иметь активность, которая позволит пользователю запрашивать и изменять данные, находящиеся под управлением контент-провайдера.

В мобильных приложениях контент-провайдеры необходимы в следующих случаях:

* приложение предоставляет сложные данные или файлы другим приложениям;
* приложение позволяет пользователям копировать сложные данные в другие приложения;
* приложение предоставляет специальные варианты поиска, используя поисковую платформу (framework).

Если приложение требует использования контент-провайдера, необходимо выполнить несколько этапов для создания этого компонента:

1. **Проектирование способа хранения данных.** Данные, с которыми работают контент-провайдеры, могут быть организованы двумя способами:

* Данные представлены файлом, например, фотографии, аудио или видео. В этом случае необходимо хранить данные в собственной области памяти приложения. В ответ на запрос от другого приложения, провайдер может возвращать ссылку на файл.
* Данные представлены некоторой структурой, например, таблица, массив. В этом случае необходимо хранить данные в табличной форме. Строка таблицы представляет собой некоторую сущность, например, сотрудник или товар. А столбец - некоторое свойство этой сущности, например, имя сотрудника или цена товара. В системе Android общий способ хранения подобных данных - база данных SQLite, но можно использовать любой способ постоянного хранения.

Больше о хранении данных в Android можно узнать по ссылке: <http://developer.android.com/guide/topics/providers/content-provider-creating.html#DataStorage>

2. **Создание класса-наследника от класса** ContentProvider напрямую или через любого его потомка. При этом в реализации класса необходимо переопределить (т. е. написать свою реализацию) обязательные методы.

|  |  |
| --- | --- |
| query() | - метод, извлекающий данные из провайдера, в качестве аргументов получает таблицу, строки и столбцы, а также порядок сортировки результата, возвращает объект типа Cursor. |
| insert() | - метод, добавляющий новую строку, в качестве аргументов получает таблицу, и значения элементов строки, возвращает URI добавленной строки. |
| update() | - метод, обновляющий существующие строки, в качестве аргументов получает таблицу, строки для обновления и новые значения элементов строк, возвращает количество обновленных строк. |
| delete() | - метод, удаляющий строки, в качестве аргументов принимает таблицу и строки для удаления, возвращает количество удаленных строк. |
| getType() | - метод, возвращающий String в формате MIME, который описывает тип данных, соответствующий URI. Подробнее:<http://developer.android.com/guide/topics/providers/content-provider-creating.html#MIMETypes> |
| onCreate() | - метод, вызываемый системой, сразу после создания провайдера, включает инициализацию провайдера. Стоит отметить, что провайдер не создается до тех пор, пока объект ContentResolver не попытается получить к нему доступ. |

Созданный контент-провайдер управляет доступом к структурированным данным, выполняя обработку запросов от других приложений. Все запросы, в конечном итоге, вызывают объект ContentResolver, который в свою очередь вызывает подходящий метод объектаContentProvider для получения доступа. Все вышеперечисленные методы, кроме onCreate(), вызываются приложением-клиентом. И все эти методы имеют такую же сигнатуру, как одноименные методы класса ContentResolver. Подробнее о классе ContentProvider:<http://developer.android.com/guide/topics/providers/content-provider-creating.html#ContentProvider>

3. **Определение строки авторизации провайдера, URI для его строк и имен столбцов.** Если от провайдера требуется управление намерениями, необходимо определить действия намерений, внешние данные и флаги. Также необходимо определить разрешения, которые необходимы приложениям для доступа к данным провайдера. Все эти значения необходимо определить как константы в отдельном классе, этот класс в последствии можно предоставить другим разработчикам.

Подробнее об URI:

<http://developer.android.com/guide/topics/providers/content-provider-creating.html#ContentURI>

Подробнее о намерениях:

<http://developer.android.com/guide/topics/providers/content-provider-creating.html#Intents>

#### 3.4.4 Приемники широковещательных сообщений (Broadcast Receivers)

Каждый широковещательный приемник является наследником класса BroadcastReceiver. Этот класс рассчитан на получение объектов-намерений отправленных методом sendBroadcast().

Можно выделить две разновидности широковещательных сообщений:

* **Нормальные широковещательные сообщения** передаются с помощью Context.sendBroadcast в асинхронном режиме. Все приемники срабатывают в неопределенном порядке, часто в одно и то же время.
* **Направленные широковещательные сообщения** передаются с помощью Context.sendOrderedBroadcast только одному приемнику в один момент времени. Как только приемник сработает, он может передать сообщение следующему приемнику, а может прервать вещание так, что больше ни один приемник это сообщение не получит.

Даже в случае нормального широковещания могут сложиться ситуации, в которых система будет передавать сообщения только одному приемнику в один момент времени. Особенно это актуально для приемников, которые требуют создания процессов, чтобы не перегружать систему новыми процессами. Однако в этом случае ни один приемник не может прервать широковещание.

Объект типа BroadcastReceiver действителен только во время вызова метода onRecieve(), как только метод выполнен, система завершает работу объекта и больше не активирует его.

Подробнее о приемниках широковещательных сообщений:

<http://developer.android.com/reference/android/content/BroadcastReceiver.html>

**3.5 Манифест приложения**

*Корневой каталог* каждого приложения под *Android* должен содержать *файл* AndroidManifest.*xml* (в точности с таким названием). *Манифест*приложения содержит всю необходимую информацию, используемую системой для запуска и выполнения приложения. Основная *информация*, содержащаяся в манифесте:

* Имя Java пакета приложения, которое используется как уникальный идентификатор приложения.
* Описание компонентов приложения: активностей, сервисов, приемников широковещательных сообщений и контент-провайдеров, которые составляют приложение. Для каждого компонента приложения определено имя соответствующего класса и объявлены их основные свойства (например, с какими сообщениями-намерениями они могут работать). Эта информация позволяет системе Android узнать какие компоненты и при каких условиях могут быть запущены.
* Определение процессов, в которых будут выполняться компоненты приложения.
* Объявление полномочий, которыми должно обладать приложение для доступа к защищенным частям API и взаимодействия с другими приложениями.
* Объявление полномочий, которыми должны обладать другие приложения для взаимодействия с компонентами данного.
* Список вспомогательных классов, которые предоставляют информацию о ходе выполнения приложения. Эти объявления содержатся в манифесте пока идет разработка и отладка приложения, перед публикацией приложения они удаляются.
* Определение минимального уровня Android API для приложения.
* Список библиотек связанных с приложением.

В файле манифеста только два элемента: <manifest> и <application> являются обязательными и при этом встречаются ровно по одному разу. Остальные элементы могут встречаться несколько раз или не появляться совсем, в этом случае *манифест* определяет пустое *приложение*.

Следующий листинг демонстрирует общую структуру файла манифеста.

<?xml version="1.0" encoding="utf-8"?>

<manifest>

<uses-permission />

<permission />

<permission-tree />

<permission-group />

<instrumentation />

<uses-sdk />

<uses-configuration />

<uses-feature />

<support-screens />

<compatible-screens />

<supports-gl-texture />

<application>

<activity>

<intent-filter>

<action />

<category />

<data />

</intent-filter>

<meta-data />

</activity>

<activity-alias>

<intent-filter> … </intent-filter>

<meta-data />

</activity-alias>

<service>

<intent-filter> … </intent-filter>

<meta-data />

</service>

<receiver>

<intent-filter> … </intent-filter>

<meta-data />

</receiver>

<provider>

<grant-uri-permission />

<meta-data />

<path-permission />

</provider>

<uses-library />

</application>

</manifest>

Листинг 3.1. Структура файла AndroidManifest.xml

В манифесте элементы одного уровня, такие как <activity>, <service>, <receiver>, <provider>, могут следовать друг за другом в любой последовательности. Элемент <activity-alias> является исключением из этого правила, он должен следовать за соответствующей активностью.

Более предметно разговор о файле манифеста и его основных элементах пойдет в лабораторных работах.

Подробности: <http://developer.android.com/guide/topics/manifest/manifest-intro.html>

**3.6 Ресурсы**

При разработке мобильных приложений необходимо выработать привычку отделять ресурсы приложения от кода. К ресурсам приложения могут относиться: изображения, строки, цвета, компоновки элементов пользовательского интерфейса (*layout*) и т. д. Отделение ресурсов от кода позволяет использовать *альтернативные* ресурсы для различных конфигураций устройств: язык, разрешение экрана и т. д. Для обеспечения совместимости с различными конфигурациями, ресурсы необходимо сгруппировать в директории по типу ресурсов и конфигурации устройства, полученные директории поместить в папку **res/**.

Для любого типа ресурсов можно определить две группы. Первая определяет ресурсы, которые будут использоваться независимо от конфигурации устройства или в том случае, когда под конфигурацию нет подходящих альтернативных ресурсов. Эта *группа* называется ресурсы по умолчанию (*default*). Вторая *группа* определяет ресурсы, подходящие для определенной конфигурации устройства, размещается в директории с названием, обозначающим данную конфигурацию. Такие ресурсы называются альтернативными.

|  |  |
| --- | --- |
| http://www.intuit.ru/EDI/14_06_16_2/1465856498-31733/tutorial/1258/objects/3/files/03_06a.jpg | http://www.intuit.ru/EDI/14_06_16_2/1465856498-31733/tutorial/1258/objects/3/files/03_06b.jpg |
| а) используется компоновка по умолчанию (приложение не содержит альтернативы) | б) каждое устройство использует соответствующую компоновку |
| **Рис. 3.6.** Использование ресурсов | |

Каждый тип ресурсов необходимо размещать в специальной поддиректории папки **res/**. Рассмотрим основные из этих поддиректорий:

|  |  |
| --- | --- |
| **animator/** | - содержит XML файлы, которые определяют свойства анимации; |
| **anim/** | - содержит XML файлы, которые определяют анимацию преобразований; |
| **color/** | - содержит XML файлы, которые определяют списки цветов; |
| **drawable/** | - содержит графические файлы или XML файлы, которые компилируются в графические ресурсы; |
| **layout/** | - содержит XML файлы, которые определяют компоновку элементов пользовательского интерфейса; |
| **menu/** | - содержит XML файлы, которые определяют все меню приложения; |
| **values/** | - содержит XML файлы, которые определяют простые значения, таких ресурсов как, строки, числа, цвета. |

Следует отметить, что *файлы ресурсов* нельзя размещать в папку **res/** напрямую, они обязательно должны размещаться в соответствующем каталоге, иначе будет выдана ошибка компиляции.

Все ресурсы, которые содержатся в рассмотренных поддиректориях являются ресурсами по умолчанию. Понятно, что различные типы устройств могут требовать различных типов ресурсов. Например, для устройств с разными размерами экрана компоновки элементов пользовательского интерфейса должны отличаться. Рис 3.6 показывает варианты внешнего вида приложения с использованием только компоновки по умолчанию (а) и с использованием альтернативных компоновок (б). Даже на схеме понятно, что при правильном подходе *приложение*, изменяющее свой внешний вид в зависимости от размера экрана привлекательнее, чем остающееся неизменным.

Чтобы определить зависимые от конфигурации альтернативы для *множества* ресурсов:

1. необходимо создать директорию в каталоге **res/**, присвоить этой директории имя в следующей форме: имя\_ресурса-спецификатор\_конфигурации, где
   * имя\_ресурса - имя директории, соответствующего ресурса по умолчанию (см. выше);
   * спецификатор\_конфигурации - имя, определяющее конфигурацию, для которой используются данные ресурсы. Полный список доступных спецификаторов: <http://developer.android.com/guide/topics/resources/providing-resources.html>
2. необходимо сохранить ресурсы в новой директории, файл ресурсов должен называться в точности так же, как соответствующий файл ресурсов по умолчанию.

Например, если *компоновка* элементов пользовательского интерфейса сохранена, как *ресурс* по умолчанию, в папке **res/layout/**, можно (скорее даже нужно) определить альтернативную компоновку элементов пользовательского интерфейса, соответствующую горизонтальной (альбомной) ориентации экрана смартфона и сохранить ее в папке **res/layout-land/**. *Android* автоматически определит подходящую компоновку, сверяя текущее состояние устройства с именами папок в каталоге **/res**.

Все ресурсы после определения могут быть доступны по ссылке на их *ID*, которые определены в автоматически генерируемом классе R. Для каждого типа ресурсов в R классе существует *подкласс*, например, R.drawable для всех графических ресурсов. *ID* ресурса всегда имеет две составляющие:

* тип ресурса - все ресурсы группируются по типам, например, string, drawable, layout;
* имя ресурса - либо имя файла без расширения, либо значение атрибута android:name в XML файле для простого значения.

Получить *доступ* к ресурсу можно двумя способами:

* в коде: можно использовать выражения вида R.тип\_ресурса.имя\_ресурса, например, R.string.hello;
* в XML: используется специальный XML синтаксис, который соответствует ID определенному в R классе, например, @string/hello.

Более предметно разговор об использовании ресурсов в лабораторных работах.

Подробности: <http://developer.android.com/guide/topics/resources/index.html>