Операции

[Activity](https://developer.android.com/reference/android/app/Activity.html?hl=ru) — это компонент приложения, который выдает экран, и с которым пользователи могут взаимодействовать для выполнения каких-либо действий, например набрать номер телефона, сделать фото, отправить письмо или просмотреть карту. Каждой операции присваивается окно для прорисовки соответствующего пользовательского интерфейса. Обычно окно отображается во весь экран, однако его размер может быть меньше, и оно может размещаться поверх других окон.

Как правило, приложение состоит из нескольких операций, которые слабо связаны друг с другом. Обычно одна из операций в приложении обозначается как «основная», предлагаемая пользователю при первом запуске приложения. В свою очередь, каждая операция может запустить другую операцию для выполнения различных действий. Каждый раз, когда запускается новая операция, предыдущая операция останавливается, однако система сохраняет ее в стеке («стек переходов назад»). При запуске новой операции она помещается в стек переходов назад и отображается для пользователя. Стек переходов назад работает по принципу «последним вошёл — первым вышел», поэтому после того как пользователь завершил текущую операцию и нажал кнопку *Назад*, текущая операция удаляется из стека (и уничтожается), и возобновляется предыдущая операция. (Подробные сведения о стеке переходов назад представлены в статье [Задачи и стек переходов назад](https://developer.android.com/guide/components/tasks-and-back-stack.html?hl=ru).)

Когда операция останавливается по причине запуска новой операции, для уведомления об изменении ее состояния используются методы обратного вызова жизненного цикла операции. Существует несколько таких методов, которые может принимать операция вследствие изменения своего состояния — создание операции, ее остановка, возобновление или уничтожение системой; также каждый обратный вызов представляет возможность выполнить определенное действие, подходящее для соответствующего изменения состояния. Например, в случае остановки операция должна освободить любые крупные объекты, например, подключение к сети или базе данных. При возобновлении операции вы можете повторно получить необходимые ресурсы и возобновить выполнение прерванных действий. Такие изменения состояния являются частью жизненного цикла операции.

Далее в этом документе рассматриваются основы создания и использования операций, включая полное описание жизненного цикла операции, чтобы вы могли лучше понять, как следует управлять переходами между различными состояниями операции.

Создание операции

Чтобы создать операцию, сначала необходимо создать подкласс класса [Activity](https://developer.android.com/reference/android/app/Activity.html?hl=ru) (или воспользоваться существующим его подклассом). В таком подклассе необходимо реализовать методы обратного вызова, которые вызывает система при переходе операции из одного состояния своего жизненного цикла в другое, например при создании, остановке, возобновлении или уничтожении операции. Вот два наиболее важных метода обратного вызова:

[onCreate()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onCreate(android.os.Bundle))

Этот метод необходимо обязательно реализовать, поскольку система вызывает его при создании вашей операции. В своей реализации вам необходимо инициализировать ключевые компоненты операции. Наиболее важно именно здесь вызвать [setContentView()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#setContentView(android.view.View)) для определения макета пользовательского интерфейса операции.

[onPause()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onPause())

Система вызывает этот метод в качестве первого признака выхода пользователя из операции (однако это не всегда означает, что операция будет уничтожена). Обычно именно здесь необходимо применять любые изменения, которые должны быть сохранены помимо текущего сеанса работы пользователя (поскольку пользователь может не вернуться назад).

Существуют также и некоторые другие методы обратного вызова жизненного цикла, которые необходимо использовать для того, чтобы обеспечить плавный переход между операциями, а также для обработки непредвиденных нарушений в работе операции, в результате которых она может быть остановлена или даже уничтожена. Более подробное описание всех методов обратного вызова жизненного цикла представлены в разделе, посвященном [управлению жизненным циклом операций](https://developer.android.com/guide/components/activities?hl=ru#Lifecycle).

Реализация пользовательского интерфейса

Для реализации пользовательского интерфейса операции используется иерархия представлений —объектов, полученных из класса [View](https://developer.android.com/reference/android/view/View.html?hl=ru). Каждое представление отвечает за определенную прямоугольную область окна операции и может реагировать на действия пользователей. Например, представлением может быть кнопка, нажатие на которую приводит к выполнению определенного действия.

В Android предусмотрен набор уже готовых представлений, которые можно использовать для создания дизайна макета и его организации. Виджеты — это представления с визуальными (и интерактивными) элементами, например, кнопками, текстовыми полями, чекбоксами или просто изображениями. Макеты — это представления, полученные из класса [ViewGroup](https://developer.android.com/reference/android/view/ViewGroup.html?hl=ru), обеспечивающие уникальную модель компоновки для своих дочерних представлений, таких как линейный макет, сетка или относительный макет. Также можно создать подкласс для классов [View](https://developer.android.com/reference/android/view/View.html?hl=ru) и [ViewGroup](https://developer.android.com/reference/android/view/ViewGroup.html?hl=ru) (или воспользоваться существующими подклассами), чтобы создать собственные виджеты и макеты, и затем применить их к макету своей операции.

Чаще всего для задания макета с помощью представлений используется XML-файл макета, сохраненный в ресурсах приложения. Таким образом вы можете хранить дизайн пользовательского интерфейса отдельно от исходного кода, который служит для задания поведения операции. Чтобы задать макет в качестве пользовательского интерфейса операции, можно использовать метод [setContentView()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#setContentView(int)), передав в него идентификатор ресурса для макета. Однако вы также можете создать новые [View](https://developer.android.com/reference/android/view/View.html?hl=ru) в коде вашей операции и создать иерархию представлений. Для этого вставьте [View](https://developer.android.com/reference/android/view/View.html?hl=ru) в [ViewGroup](https://developer.android.com/reference/android/view/ViewGroup.html?hl=ru), а затем используйте этот макет, передав корневой объект [ViewGroup](https://developer.android.com/reference/android/view/ViewGroup.html?hl=ru) в метод [setContentView()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#setContentView(android.view.View)).

Подробные сведения о создании пользовательского интерфейса см. в статье [Пользовательский интерфейс](https://developer.android.com/guide/topics/ui/index.html?hl=ru).

Объявление операции в манифесте

Чтобы операция стала доступна системе, ее необходимо объявить в файле манифеста. Для этого откройте файл манифеста и добавьте элемент [&lt;activity&gt;](https://developer.android.com/guide/topics/manifest/activity-element.html?hl=ru) в качестве дочернего для элемента[&lt;application&gt;](https://developer.android.com/guide/topics/manifest/application-element.html?hl=ru). Например:

<manifest ... >  
  <application ... >  
      <activity android:name=".ExampleActivity" />  
      ...  
  </application ... >  
  ...  
</manifest >

Существует несколько других атрибутов, которые можно включить в этот элемент, чтобы определить такие свойства, как метка операции, значок операции или тема оформления для пользовательского интерфейса операции. Единственным обязательным атрибутом является [android:name](https://developer.android.com/guide/topics/manifest/activity-element.html?hl=ru#nm) — он определяет имя класса операции. После публикации вашего приложения вам не следует переименовывать его, поскольку это может нарушить некоторые функциональные возможности приложения, например, ярлыки приложения (ознакомьтесь с публикацией [Вещи , которые нельзя менять](http://android-developers.blogspot.com/2011/06/things-that-cannot-change.html) в блоге разработчиков).

Дополнительные сведения об объявлении операции в манифесте см. в справке по элементу [&lt;activity&gt;](https://developer.android.com/guide/topics/manifest/activity-element.html?hl=ru).

Использование фильтров намерений

Элемент [&lt;activity&gt;](https://developer.android.com/guide/topics/manifest/activity-element.html?hl=ru) также может задавать различные фильтры намерений — с помощью элемента [&lt;intent-filter&gt;](https://developer.android.com/guide/topics/manifest/intent-filter-element.html?hl=ru) — для объявления того, как другие компоненты приложения могут активировать операцию.

При создании нового приложения с помощью инструментов Android SDK в заготовке операции, создаваемой автоматически, имеется фильтр намерений, который объявляет операцию. Эта операция реагирует на выполнение «основного» действия, и ее следует поместить в категорию переходсредства запуска. Фильтр намерений выглядит следующим образом.

<activity android:name=".ExampleActivity" android:icon="@drawable/app\_icon">  
    <intent-filter>  
        <action android:name="android.intent.action.MAIN" />  
        <category android:name="android.intent.category.LAUNCHER" />  
    </intent-filter>  
</activity>

Элемент [&lt;action&gt;](https://developer.android.com/guide/topics/manifest/action-element.html?hl=ru) указывает, что это «основная» точка входа в приложение. Элемент [&lt;category&gt;](https://developer.android.com/guide/topics/manifest/category-element.html?hl=ru)указывает, что эту операцию следует указать в средстве запуска приложений системы (чтобы пользователи могли запускать эту операцию).

Если приложение планируется создать самодостаточным и запретить другим приложениям активировать его операции, то других фильтров намерений создавать не нужно. В этом случае только в одной операции должно иметься «основное» действие, и ее следует поместить в категорию средства запуска, как в примере выше. В операциях, которые не должны быть доступны для других приложений, не следует включать фильтры намерений. Вы можете самостоятельно запустить такие операции с помощью явных намерений (как описывается в следующем разделе).

Однако, если вам необходимо, чтобы операция реагировала на неявные намерения, получаемые от других приложений (а также из вашего приложения), для операции необходимо определить дополнительные фильтры намерений. Для каждого типа намерения, на который необходимо реагировать, необходимо указать объект [&lt;intent-filter&gt;](https://developer.android.com/guide/topics/manifest/intent-filter-element.html?hl=ru), включающий элемент[&lt;action&gt;](https://developer.android.com/guide/topics/manifest/action-element.html?hl=ru) и необязательный элемент [&lt;category&gt;](https://developer.android.com/guide/topics/manifest/category-element.html?hl=ru) или [&lt;data&gt;](https://developer.android.com/guide/topics/manifest/data-element.html?hl=ru) (или оба этих элемента). Эти элементы определяют тип намерения, на который может реагировать ваша операция.

Дополнительные сведения о том, как операции могут реагировать на намерения, приведены в статье [Намерения и фильтры намерений](https://developer.android.com/guide/components/intents-filters.html?hl=ru).

Запуск операции

Для запуска другой операции достаточно вызвать метод [startActivity()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#startActivity(android.content.Intent)), передав в него объект [Intent](https://developer.android.com/reference/android/content/Intent.html?hl=ru), который описывает запускаемую операцию. В намерении указывается либо точная операция для запуска, либо описывается тип операции, которую вы хотите выполнить (после чего система выбирает для вас подходящую операцию, которая может даже находиться в другом приложении). Намерение также может содержать небольшой объем данных, которые будут использоваться запущенной операцией.

При работе с собственным приложением зачастую требуется лишь запустить нужную операцию. Для этого необходимо создать намерение, которое явно определяет требуемую операцию с помощью имени класса. Ниже представлен пример запуска одной операцией другой операции с именем SignInActivity.

Intent intent = new Intent(this, SignInActivity.class);  
startActivity(intent);

Однако в вашем приложении также может потребоваться выполнить некоторое действие, например, отправить письмо, текстовое сообщение или обновить статус, используя данные из вашей операции. В этом случае в вашем приложении могут отсутствовать такие действия, поэтому вы можете воспользоваться операциями из других приложений, имеющихся на устройстве, которые могут выполнять требуемые действия. Как раз в этом случае намерения особенно полезны — можно создать намерение, которое описывает необходимое действие, после чего система запускает его из другого приложения. При наличии нескольких операций, которые могут обработать намерение, пользователь может выбирать, какое из них следует использовать. Например, если пользователю требуется предоставить возможность отправить электронное письмо, можно создать следующее намерение:

Intent intent = new Intent(Intent.ACTION\_SEND);  
intent.putExtra(Intent.EXTRA\_EMAIL, recipientArray);  
startActivity(intent);

Дополнительный компонент [EXTRA\_EMAIL](https://developer.android.com/reference/android/content/Intent.html?hl=ru#EXTRA_EMAIL), добавленный в намерение, представляет собой строковый массив адресов электронной почты для отправки письма. Когда почтовая программа реагирует на это намерение, она считывает дополнительно добавленный строковый массив и помещает имеющиеся в нем адреса в поле получателя в окне создания письма. При этом запускается операция почтовой программы, а после того, как пользователь завершит требуемые действия, возобновляется ваша операция.

Запуск операции для получения результата

В некоторых случаях после запуска операции может потребоваться получить результат. Для этого вызовите метод [startActivityForResult()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#startActivityForResult(android.content.Intent,%20int)) (вместо [startActivity()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#startActivity(android.content.Intent))). Чтобы получить результат после выполнения последующей операции, реализуйте метод обратного вызова [onActivityResult()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onActivityResult(int,%20int,%20android.content.Intent)). По завершении последующей операции она возвращает результат в объекте [Intent](https://developer.android.com/reference/android/content/Intent.html?hl=ru) в вызванный метод [onActivityResult()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onActivityResult(int,%20int,%20android.content.Intent)).

К примеру, пользователю потребуется выбрать один из контактов, чтобы ваша операция могла выполнить некоторые действия с информацией об этом контакте. Ниже представлен пример создания такого намерения и обработки результата.

private void pickContact() {  
    // Create an intent to "pick" a contact, as defined by the content provider URI  
    Intent intent = new Intent(Intent.ACTION\_PICK, Contacts.CONTENT\_URI);  
    startActivityForResult(intent, PICK\_CONTACT\_REQUEST);  
}  
  
@Override  
protected void onActivityResult(int requestCode, int resultCode, Intent data) {  
    // If the request went well (OK) and the request was PICK\_CONTACT\_REQUEST  
    if (resultCode == Activity.RESULT\_OK && requestCode == PICK\_CONTACT\_REQUEST) {  
        // Perform a query to the contact's content provider for the contact's name  
        Cursor cursor = getContentResolver().query(data.getData(),  
        new String[] {Contacts.DISPLAY\_NAME}, null, null, null);  
        if (cursor.moveToFirst()) { // True if the cursor is not empty  
            int columnIndex = cursor.getColumnIndex(Contacts.DISPLAY\_NAME);  
            String name = cursor.getString(columnIndex);  
            // Do something with the selected contact's name...  
        }  
    }  
}

В этом примере демонстрируется базовая логика, которой следует руководствоваться при использовании метода [onActivityResult()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onActivityResult(int,%20int,%20android.content.Intent)) для обработки результата выполнения операции. Первое условие проверяет, успешен ли запрос, и если он успешен, то результат для resultCode будет [RESULT\_OK](https://developer.android.com/reference/android/app/Activity.html?hl=ru#RESULT_OK); также проверяется, известен ли запрос, для которого получен этот результат, и в этом случае requestCode соответствует второму параметру, отправленному в метод [startActivityForResult()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#startActivityForResult(android.content.Intent,%20int)). Здесь код обрабатывает результат выполнения операции путем запроса данных, возвращенных в [Intent](https://developer.android.com/reference/android/content/Intent.html?hl=ru) (параметр data).

При этом [ContentResolver](https://developer.android.com/reference/android/content/ContentResolver.html?hl=ru) выполняет запрос к поставщику контента, который возвращает объект[Cursor](https://developer.android.com/reference/android/database/Cursor.html?hl=ru), обеспечивающий считывание запрошенных данных. Дополнительные сведения представлены в статье [Поставщики контента](https://developer.android.com/guide/topics/providers/content-providers.html?hl=ru).

Дополнительные сведения об использовании намерений см. в статье [Намерения и фильтры намерений](https://developer.android.com/guide/components/intents-filters.html?hl=ru).

Завершение операции

Для завершения операции достаточно вызвать ее метод [finish()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#finish()). Также для завершения отдельной операции, запущенной ранее, можно вызвать метод [finishActivity()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#finishActivity(int)).

**Примечание.** В большинстве случаев вам не следует явно завершать операцию с помощью этих методов. Как указано в следующем разделе, посвященном управлению жизненным циклом операции, система Android выполняет такое управление за вас, поэтому вам не нужно завершать ваши собственные операции. Вызов этих методов может отрицательно сказаться на ожидаемом поведении приложения. Их следует использовать исключительно тогда, когда вы абсолютно не хотите, чтобы пользователь возвращался к этому экземпляру операции.

Управление жизненным циклом операций

Управление жизненным циклом операций путем реализации методов обратного вызова имеет важное значение при разработке надежных и гибких приложений. На жизненный цикл операции напрямую влияют его связи с другими операциями, его задачами и стеком переходов назад.

Существует всего три состояния операции:

***Возобновлена***

Операция выполняется на переднем плане экрана и отображается для пользователя. (Это состояние также иногда называется «Выполняется».)

***Приостановлена***

На переднем фоне выполняется другая операция, которая отображается для пользователя, однако эта операция по-прежнему не скрыта. То есть поверх текущей операции отображается другая операция, частично прозрачная или не занимающая полностью весь экран. Приостановленная операция полностью активна (объект [Activity](https://developer.android.com/reference/android/app/Activity.html?hl=ru) по-прежнему находится в памяти, в нем сохраняются все сведения о состоянии и информация об элементах, и он также остается связанным с диспетчером окон), однако в случае острой нехватки памяти система может завершить ее.

***Остановлена***

Операция полностью перекрывается другой операцией (теперь она выполняется в фоновом режиме). Остановленная операция по-прежнему активна (объект [Activity](https://developer.android.com/reference/android/app/Activity.html?hl=ru) по-прежнему находится в памяти, в нем сохраняются все сведения о состоянии и информация об элементах, но объект больше *не* связан с диспетчером окон). Однако операция больше не видна пользователю, и в случае нехватки памяти система может завершить ее.

Если операция приостановлена или полностью остановлена, система может очистить ее из памяти путем завершения самой операции (с помощью метода [finish()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#finish())) или просто завершить ее процесс. В случае повторного открытия операции (после ее завершения) ее потребуется создать полностью.

Реализация обратных вызовов жизненного цикла

При переходе операции из одного вышеописанного состояния в другое, уведомления об этом реализуются через различные методы обратного вызова. Все методы обратного вызова представляют собой привязки, которые можно переопределить для выполнения подходящего действия при изменении состояния операции. Указанная ниже базовая операция включает каждый из основных методов жизненного цикла.

public class ExampleActivity extends Activity {  
    @Override  
    public void [**onCreate**](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onCreate(android.os.Bundle))(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        // The activity is being created.  
    }  
    @Override  
    protected void [**onStart()**](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onStart()) {  
        super.onStart();  
        // The activity is about to become visible.  
    }  
    @Override  
    protected void [**onResume()**](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onResume()) {  
        super.onResume();  
        // The activity has become visible (it is now "resumed").  
    }  
    @Override  
    protected void [**onPause()**](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onPause()) {  
        super.onPause();  
        // Another activity is taking focus (this activity is about to be "paused").  
    }  
    @Override  
    protected void [**onStop()**](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onStop()) {  
        super.onStop();  
        // The activity is no longer visible (it is now "stopped")  
    }  
    @Override  
    protected void [**onDestroy()**](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onDestroy()) {  
        super.onDestroy();  
        // The activity is about to be destroyed.  
    }  
}

**Примечание.** При реализации этих методов жизненного цикла всегда вызывайте реализацию суперкласса, прежде чем выполнять какие-либо действия, как показано в примерах выше.

Вместе все эти методы определяют весь жизненный цикл операции. С помощью реализации этих методов можно отслеживать три вложенных цикла в жизненном цикле операции:

* **Весь жизненный цикл** операции происходит между вызовом метода [onCreate()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onCreate(android.os.Bundle)) и вызовом метода [onDestroy()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onDestroy()). Ваша операция должна выполнить настройку «глобального» состояния (например, определение макета) в методе [onCreate()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onCreate(android.os.Bundle)), а затем освободить все оставшиеся в [onDestroy()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onDestroy()) ресурсы. Например, если в вашей операции имеется поток, выполняющийся в фоновом режиме, для загрузки данных по сети, операция может создать такой поток в методе [onCreate()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onCreate(android.os.Bundle)), а затем остановить его в методе [onDestroy()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onDestroy()).
* **Видимый жизненный цикл** операции происходит между вызовами методов [onStart()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onStart()) и [onStop()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onStop()). В течение этого времени операция отображается на экране, где пользователь может взаимодействовать с ней. Например, метод [onStop()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onStop()) вызывается в случае, когда запускается новая операция, а текущая больше не отображается. В промежутке между вызовами этих двух методов можно сохранить ресурсы, необходимые для отображения операции для пользователя. Например, можно зарегистрировать объект [BroadcastReceiver](https://developer.android.com/reference/android/content/BroadcastReceiver.html?hl=ru) в методе [onStart()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onStart()) для отслеживания изменений, влияющих на пользовательский интерфейс, а затем отменить его регистрацию в методе [onStop()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onStop()), когда пользователь больше не видит отображаемого. В течение всего жизненного цикла операции система может несколько раз вызывать методы [onStart()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onStart()) и [onStop()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onStop()), поскольку операция то отображается для пользователя, то скрывается от него.
* **Жизненный цикл операции, выполняемый на переднем плане**, происходит между вызовами методов [onResume()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onResume()) и [onPause()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onPause()). В течение этого времени операция выполняется на фоне всех прочих операций и отображается для пользователя. Операция может часто уходить в фоновый режим и выходить из него — например, метод [onPause()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onPause()) вызывается при переходе устройства в спящий режим или при появлении диалогового окна. Поскольку переход в это состояние может выполняться довольно часто, код в этих двух методах должен быть легким, чтобы не допустить медленных переходов и не заставлять пользователя ждать.

На рисунке 1 иллюстрируются проходы и пути, которые операция может пройти между состояниями. Прямоугольниками обозначены методы обратного вызова, которые можно реализовать для выполнения действий между переходами операции из одного состояния в другое.



**Рисунок 1.** Жизненный цикл операции.

Эти же методы жизненного цикла перечислены в таблице 1, в которой подробно описан каждый метод обратного вызова и указано его место в жизненном цикле операции в целом, включая сведения о том, может ли система завершить операцию по завершении метода обратного вызова.

**Таблица 1.** Сводные сведения о методах обратного вызова жизненного цикла операции.

| Метод | | | Описание | Завершаемый? | Следующий |
| --- | --- | --- | --- | --- | --- |
| [onCreate()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onCreate(android.os.Bundle)) | | | Вызывается при первом создании операции. Здесь необходимо настроить все обычные статические элементы — создать представления, привязать данные и т. д. Этот метод передает объект Bundle, содержащий предыдущее состояние операции (если такое состояние было зафиксировано ранее; см. раздел [Сохранение состояния операции](https://developer.android.com/guide/components/activities?hl=ru#actstate)).  За ним всегда следует метод onStart(). | Нет | onStart() |
|  | [onRestart()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onRestart()) | | Вызывается после остановки операции непосредственно перед ее повторным запуском.  За ним всегда следует метод onStart(). | Нет | onStart() |
| [onStart()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onStart()) | | Вызывается непосредственно перед тем, как операция становится видимой для пользователя.  За ним следует метод onResume(), если операция переходит на передний план, или метод onStop(), если она становится скрытой. | Нет | onResume()  или onStop() |
|  | [onResume()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onResume()) | Вызывается непосредственно перед тем, как операция начинает взаимодействие с пользователем. На этом этапе операция находится в самом верху стека операций, и в нее поступают данные, вводимые пользователем.  За ним всегда следует метод onPause(). | Нет | onPause() |
| [onPause()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onPause()) | Вызывается, когда система собирается возобновить другую операцию. Этот метод обычно используется для записи несохраненных изменений в постоянное место хранения данных, остановки анимаций и других элементов, которые могут использовать ресурсы ЦП и т. д. Здесь крайне важна оперативность, поскольку следующая операция не будет возобновлена до тех пор, пока она не будет возвращена на передний план.  За ним следует либо метод onResume(), если операция возвращается на передний план, либо метод onStop(), если операция становится скрытой для пользователя. | Да | onResume()  или onStop() |
| [onStop()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onStop()) | | Вызывается в случае, когда операция больше не отображается для пользователя. Это может произойти по причине того, что операция уничтожена, или ввиду возобновления поверх нее другой операции (существующей или новой).  За ним следует либо метод onRestart(), если операция возобновляет взаимодействие с пользователем, либо метод onDestroy(), если операция переходит в фоновый режим. | Да | onRestart() или onDestroy() |
| [onDestroy()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onDestroy()) | | | Вызывается перед тем, как операция будет уничтожена. Это финальный вызов, который получает операция. Его можно вызвать либо по причине завершения операции (вызов метода [finish()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#finish())), либо ввиду временного уничтожения системой этого экземпляра операции с целью освободить место. Чтобы различить эти два сценария, используется метод [isFinishing()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#isFinishing()). | Да | *Ничего* |

В столбце «Завершаемый?» указывается, может ли система в любое время завершить процесс, содержащий операцию, *после возвращения метода* без выполнения другой строки кода операции. Для трех методов в этом столбце указано «Да»: ([onPause()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onPause()), [onStop()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onStop()) и [onDestroy()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onDestroy())). Поскольку метод [onPause()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onPause()) является первым из этих трех после создания операции, метод [onPause()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onPause()) является последним, который гарантированно будет вызван перед тем, как процесс *можно будет* завершить; если системе потребуется срочно восстановить память в случае аварийной ситуации, то методы [onStop()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onStop()) и [onDestroy()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onDestroy()) вызвать не удастся. Поэтому следует воспользоваться [onPause()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onPause()), чтобы записать критически важные данные (такие как правки пользователя) в хранилище постоянных данных. Однако следует внимательно подходить к выбору информации, которую необходимо сохранить во время выполнения метода [onPause()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onPause()), поскольку любая блокировка процедур в этом методе может вызвать блокирование перехода к следующей операции и тормозить работу пользователя.

Методы, для которых в столбце **Завершаемый?** указано «Нет», защищают процесс, содержащий операцию , от завершения сразу с момента их вызова. Поэтому завершить операцию можно в период между возвратом [onPause()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onPause()) и вызовом [onResume()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onResume()). Его снова не удастся завершить, пока снова не будет вызван и возвращен [onPause()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onPause()).

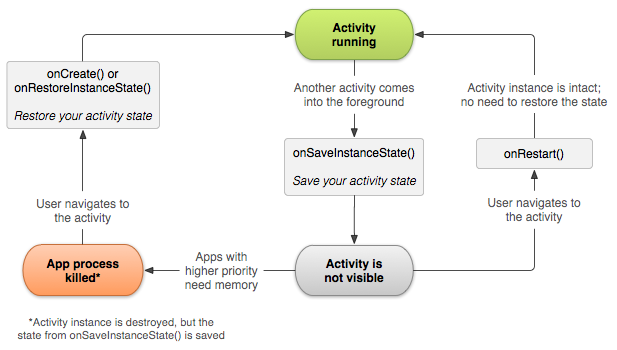
**Примечание.** Операцию, которую технически невозможно завершить в соответствии с определением в таблице 1, по-прежнему может завершить система, однако это может произойти только в чрезвычайных ситуациях, когда нет другой возможности. Случаи, когда возможно завершение операции, более подробно рассматриваются в статье[Процессы и потоки](https://developer.android.com/guide/components/processes-and-threads.html?hl=ru).

Сохранение состояния операции

В обзорных сведениях об [управлении жизненным циклом операции](https://developer.android.com/guide/components/activities?hl=ru#Lifecycle) кратко упоминается, что в случае приостановки или полной остановки операции ее состояние сохраняется. Это действительно так, поскольку объект [Activity](https://developer.android.com/reference/android/app/Activity.html?hl=ru) при этом по-прежнему находится в памяти , и вся информация о ее элементах и текущем состоянии по-прежнему активна. Поэтому любые вносимые пользователем в операции изменения сохраняются, и когда операция возвращается на передний план (когда она «возобновляется»), эти изменения остаются в этом объекте.

Однако когда система уничтожает операцию в целях восстановления памяти, объект [Activity](https://developer.android.com/reference/android/app/Activity.html?hl=ru)уничтожается, в результате чего системе не удается просто восстановить состояние операции для взаимодействия с ней. Вместо этого системе необходимо повторно создать объект [Activity](https://developer.android.com/reference/android/app/Activity.html?hl=ru), если пользователь возвращается к нему. Но пользователю неизвестно, что система уже уничтожила операцию и создала ее повторно, поэтому, возможно, он ожидает, что операция осталась прежней. В этой ситуации можно обеспечить сохранение важной информации о состоянии операции путем реализации дополнительного метода обратного вызова, который позволяет сохранить информацию о вашей операции: [onSaveInstanceState()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onSaveInstanceState(android.os.Bundle)).

Прежде чем сделать операцию доступной для уничтожения, система вызывает метод[onSaveInstanceState()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onSaveInstanceState(android.os.Bundle)). Система передает в этот метод объект [Bundle](https://developer.android.com/reference/android/os/Bundle.html?hl=ru), в котором можно сохранить информацию о состоянии операции в виде пар «имя-значение», используя для этого такие методы, как [putString()](https://developer.android.com/reference/android/os/BaseBundle.html?hl=ru#putString(java.lang.String,%20java.lang.String)) и [putInt()](https://developer.android.com/reference/android/os/BaseBundle.html?hl=ru#putInt(java.lang.String,%20int)). Затем, если система завершает процесс вашего приложения и пользователь возвращается к вашей операции, система повторно создает операцию и передает объект [Bundle](https://developer.android.com/reference/android/os/Bundle.html?hl=ru) в оба метода: [onCreate()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onCreate(android.os.Bundle)) и [onRestoreInstanceState()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onRestoreInstanceState(android.os.Bundle)). С помощью любого из этих методов можно извлечь из объекта [Bundle](https://developer.android.com/reference/android/os/Bundle.html?hl=ru) сохраненную информацию о состоянии операции и восстановить ее. Если такая информация отсутствует, то объект [Bundle](https://developer.android.com/reference/android/os/Bundle.html?hl=ru) передается с нулевым значением (это происходит в случае, когда операция создается в первый раз).



**Рисунок 2.** Два способа возврата операции к отображению для пользователя в неизмененном состоянии: уничтожение операции с последующим ее повторным созданием, когда операция должна восстановить свое ранее сохраненное состояние, или остановка операции и ее последующее восстановление в неизмененном состоянии.

**Примечание.** Нет никаких гарантий, что метод [**onSaveInstanceState()**](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onSaveInstanceState(android.os.Bundle)) будет вызван до того, как ваша операция будет уничтожена, поскольку существуют случаи, когда нет необходимости сохранять состояние (например, когда пользователь покидает вашу операцию нажатием кнопки *Назад*, явным образом закрывая ее). Если система вызывает метод [**onSaveInstanceState()**](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onSaveInstanceState(android.os.Bundle)), она делает это до вызова метода [**onStop()**](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onStop()) и, возможно, перед вызовом метода [**onPause()**](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onPause()).

Однако, даже если вы ничего не предпринимаете и не реализуете метод [onSaveInstanceState()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onSaveInstanceState(android.os.Bundle)), часть состояния операции восстанавливается реализацией по умолчанию метода [onSaveInstanceState()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onSaveInstanceState(android.os.Bundle))класса [Activity](https://developer.android.com/reference/android/app/Activity.html?hl=ru). В частности, реализация по умолчанию вызывает соответствующий метод [onSaveInstanceState()](https://developer.android.com/reference/android/view/View.html?hl=ru#onSaveInstanceState()) для каждого объекта [View](https://developer.android.com/reference/android/view/View.html?hl=ru) в макете, благодаря чему каждое представление может предоставлять ту информацию о себе, которую следует сохранить. Почти каждый виджет в платформе Android реализует этот метод необходимым для себя способом так, что любые видимые изменения в пользовательском интерфейсе автоматически сохраняются и восстанавливаются при повторном создании операции. Например, виджет [EditText](https://developer.android.com/reference/android/widget/EditText.html?hl=ru) сохраняет любой текст, введенный пользователем, а виджет [CheckBox](https://developer.android.com/reference/android/widget/CheckBox.html?hl=ru) сохраняет информацию о том, был ли установлен флажок. От вас требуется лишь указать уникальный идентификатор (с атрибутом [android:id](https://developer.android.com/guide/topics/resources/layout-resource.html?hl=ru#idvalue)) для каждого виджета, состояние которого необходимо сохранить. Если виджету не присвоен идентификатор, то системе не удастся сохранить его состояние.

Вы также можете явно отключить сохранение информации о состоянии представления в макете. Для этого задайте для атрибута[android:saveEnabled](https://developer.android.com/reference/android/R.attr.html?hl=ru#saveEnabled) значение "false" или вызовите метод[setSaveEnabled()](https://developer.android.com/reference/android/view/View.html?hl=ru#setSaveEnabled(boolean)). Обычно отключать сохранение такой информации не требуется , однако это может потребоваться в случаях, когда восстановить состояние пользовательского интерфейса операции необходимо другим образом.

Несмотря на то что реализация метода [onSaveInstanceState()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onSaveInstanceState(android.os.Bundle)) по умолчанию позволяет сохранить полезную информацию о пользовательском интерфейсе вашей операции, вам по-прежнему может потребоваться переопределить ее для сохранения дополнительной информации. Например, может потребоваться сохранить значения элементов, которые изменялись в течение жизненного цикла операции (которые могут коррелировать со значениями, восстановленными в пользовательском интерфейсе, однако элементы, содержащие эти значения пользовательского интерфейса, по умолчанию не были восстановлены).

Поскольку реализация метода [onSaveInstanceState()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onSaveInstanceState(android.os.Bundle)) по умолчанию позволяет сохранить состояние пользовательского интерфейса, в случае , если вы переопределите метод с целью сохранить дополнительную информацию о состоянии, перед выполнением каких-либо действий вы всегда можете вызвать реализацию суперкласса для метода [onSaveInstanceState()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onSaveInstanceState(android.os.Bundle)). Точно так же реализацию суперкласса [onRestoreInstanceState()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onRestoreInstanceState(android.os.Bundle)) следует вызывать в случае ее переопределения, чтобы реализация по умолчанию могла сохранить состояния представлений.

**Примечание.** Поскольку вызов метода [**onSaveInstanceState()**](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onSaveInstanceState(android.os.Bundle)) не гарантируется, вам следует использовать его только для записи переходного состояния операции (состояние пользовательского интерфейса) — никогда не используйте его для хранения постоянных данных. Вместо этого используйте метод [**onPause()**](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onPause())для сохранения постоянных данных (например, тех, которые следует сохранить в базу данных), когда пользователь покидает операцию.

Отличный способ проверить возможность вашего приложения восстанавливать свое состояние — это просто повернуть устройство для изменения ориентации экрана. При изменении ориентации экрана система уничтожает и повторно создает операцию, чтобы применить альтернативные ресурсы, которые могут быть доступны для новой конфигурации экрана. Только по одной этой причине крайне важно, чтобы ваша операция могла полностью восстанавливать свое состояние при ее повторном создании, поскольку пользователи постоянно работают с приложениями в разных ориентациях экрана.

Обработка изменений в конфигурации

Некоторые конфигурации устройств могут изменяться в режиме выполнения (например, ориентация экрана, доступность клавиатуры и язык). В таких случаях Android повторно создает выполняющуюся операцию (система сначала вызывает метод [onDestroy()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onDestroy()), а затем сразу же вызывает метод [onCreate()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onCreate(android.os.Bundle))). Такое поведение позволяет приложению учитывать новые конфигурации путем автоматической перезагрузки в приложение альтернативных ресурсов, которые вы предоставили (например, различные макеты для разных ориентаций и экранов разных размеров).

Если операция разработана должным образом и должным образом поддерживает перезапуск после изменения ориентации экрана и восстановление своего состояния, как описано выше, ваше приложение можно считать более устойчивым к другим непредвиденным событиям в жизненном цикле операции.

Лучший способ обработки такого перезапуска — сохранить и восстановить состояние операции с помощью методов [onSaveInstanceState()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onSaveInstanceState(android.os.Bundle)) и [onRestoreInstanceState()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onRestoreInstanceState(android.os.Bundle)) (или [onCreate()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onCreate(android.os.Bundle))), как описано в предыдущем разделе.

Дополнительные сведения об изменениях конфигурации, происходящих в режиме выполнения, и о способах их обработки представлены в руководстве[Обработка изменений в режиме выполнения](https://developer.android.com/guide/topics/resources/runtime-changes.html?hl=ru).

Согласование операций

Когда одна операция запускает другую, в жизненных циклах обеих из них происходит переход из одного состояния в другое. Первая операция приостанавливается и заврешается (однако она не будет остановлена, если она по-прежнему видима на фоне), а вторая операция создается. В случае, если эти операции обмениваются данным, сохраненными на диске или в другом месте, важно понимать, что первая операция не останавливается полностью до тех пор, пока не будет создана вторая операция. Наоборот, процесс запуска второй операции накладывается на процесс остановки первой операции.

Порядок обратных вызовов жизненного цикла четко определен, в частности, когда в одном и том же процессе находятся две операции , и одна из них запускает другую. Ниже представлен порядок выполнения действий в случае, когда операция А запускает операцию Б.

1. Выполняется метод [onPause()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onPause()) операции А.
2. Последовательно выполняются методы [onCreate()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onCreate(android.os.Bundle)), [onStart()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onStart()) и [onResume()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onResume()) операции Б. (Теперь для пользователя отображается операция Б.)
3. Затем, если операция A больше не отображается на экране, выполняется ее метод [onStop()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onStop()).

Такая предсказуемая последовательность выполнения обратных вызовов жизненного цикла позволяет управлять переходом информации из одной операции в другую. Например, если после остановки первой операции требуется выполнить запись в базу данных, чтобы следующая операция могла считать их, то запись в базу данных следует выполнить во время выполнения метода [onPause()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onPause()), а не во время выполнения метода [onStop()](https://developer.android.com/reference/android/app/Activity.html?hl=ru#onStop()).

# Material Design

С появлением Android 5.0 был представлен новый дизайн Material, который позже стал доступен и для старых устройств через библиотеку совместимости в немного усечённом виде.

Если говорить общими словами, то дизайн пытается быть похожим на реальный мир. Это значит, что элементы подчиняются физическим законам. Например, элементы можно разделить и соединить обратно, изменить их цвет и форму, они твёрдые. Элементы могут отбрасывать тень. Но нельзя делать некоторых вещей - не могут уместиться в одном месте, если не позволяет объём. Причём, все элементы имеют толщину 1dp. Представьте себе, что элементы в виде толстой бумаги внутри вашего телефона под стеклом. Вы не можете сделать оборот элемента вокруг своей оси, так как толщина вашего телефона слишком мала для этой операции. Может только чуть-чуть повернуть. Элементы не могут просачиваться через другие элементы (они же твёрдые). Вообще, многие свойства материального дизайна присущи котам, за исключением разве только толщины.

Мы сфокусируемся в основном на библиотеке совместимости, чтобы уметь переделывать свои старые проекты под новый дизайн. При создании новых проектов в студии добавляется поддержка старых устройств через зависимость:

implementation 'com.android.support:appcompat-v7:26.1.0'

Когда материальный дизайн только появился, то студия генерировала шаблон с двумя файлами стилей - стандартный **values/styles.xml** и новый **values-v21/styles.xml**. Сейчас от такого подхода отказались. Но при необходимости, вы можете вручную поменять тему по умолчанию на **Theme.Material.Light** или ему подобными.

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

<resources>

<style name="AppTheme" parent="android:Theme.Material.Light">

</style>

</resources>

При использовании компонентов на панели действий не забывайте вызывать метод **getSupportActionBar().getThemedContext()** для получения правильного контекста.

Если продолжить разговор о библиотеке совместимости, то обратите внимание, что в новых проектах используется **AppCompatActivity** вместо **ActionBarActivity**.

### Elevation (высота)

У компонентов появился новый атрибут **elevation**:

<ImageView …

android:elevation="8dp" />

В соответствии с новым дизайном компоненты могут "парить" над экраном, отбрасывая тень. Система сама рассчитает нужную тень вокруг элемента, в том числе и вокруг круглой кнопки. Программным аналогом являются методы **getElevation()/setElevation()**.

[Подробнее...](http://developer.alexanderklimov.ru/android/elevation.php)

### Диалоговые окна

Появился новый класс **AppCompatDialog** с темой **Theme.AppCompat.Dialog**, а также новый **AlertDialog**(android.support.v7.app.AlertDialog) с поддержкой Material Design.

### Компоненты

Когда вы используете стандартные компоненты и тему материального дизайна, то предпринимать дополнительные усилия не нужно - компоненты будут выглядеть правильно. Но если вы разрабатываете собственные элементы, то нужно использовать специальные классы. Сейчас студия сама подсказывает об этом. Список некоторых классов в стиле Material Design.

* AppCompatButton
* AppCompatCheckBox
* AppCompatEditText
* AppCompatSpinner
* AppCompatTextView

# Файл манифеста AndroidManifest.xml

Файл манифеста **AndroidManifest.xml** предоставляет основную информацию о программе системе. Каждое приложение должно иметь свой файл **AndroidManifest.xml**. Редактировать файл манифеста можно вручную, изменяя XML-код или через визуальный редактор Manifest Editor (Редактор файла манифеста), который позволяет осуществлять визуальное и текстовое редактирование файла манифеста приложения.

## Назначение файла

* объявляет имя Java-пакета приложения, который служит уникальным идентификатором;
* описывает компоненты приложения — деятельности, службы, приемники широковещательных намерений и контент-провайдеры, что позволяет вызывать классы, которые реализуют каждый из компонентов, и объявляет их намерения;
* содержит список необходимых разрешений для обращения к защищенным частям API и взаимодействия с другими приложениями;
* объявляет разрешения, которые сторонние приложения обязаны иметь для взаимодействия с компонентами данного приложения;
* объявляет минимальный уровень API Android, необходимый для работы приложения;
* перечисляет связанные библиотеки;

## Общая структура манифеста

Файл манифеста инкапсулирует всю архитектуру Android-приложения, его функциональные возможности и конфигурацию. В процессе разработки приложения вам придется постоянно редактировать данный файл, изменяя его структуру и дополняя новыми элементами и атрибутами.

Корневым элементом манифеста является **<manifest>**. Помимо данного элемента обязательными элементами является теги **<application>** и **<uses-sdk>**. Элемент **<application>** является основным элементом манифеста и содержит множество дочерних элементов, определяющих структуру и работу приложения. Порядок расположения элементов, находящихся на одном уровне, произвольный. Все значения устанавливаются через атрибуты элементов. Кроме обязательных элементов, упомянутых выше, в манифесте по мере необходимости используются другие элементы.

### Описание

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

[<manifest />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#manifest)

[<uses-permission />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#uses-permission)

[<permission />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#permission)

[<permission-tree />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#permission-tree)

[<permission-group />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#permission-group)

[<instrumentation />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#instrumentation)

[<uses-sdk />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#uses-sdk)

[<uses-configuration />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#uses-configuration)

[<uses-feature />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#uses-feature)

[<supports-screens />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#supports-screens)

[<application>](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#application)

[<activity>](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#activity)

[<intent-filter>](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#intent-filter)

[<action />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#action)

[<category />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#category)

[<data />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#data)

[</intent-filter>](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#intent-filter)

[<meta-data />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#meta-data)

[</activity>](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#activity)

[<activity-alias>](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#activity-alias)

[<intent-filter>](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#intent-filter)

[<action />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#action)

[<category />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#category)

[<data />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#data)

[</intent-filter>](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#intent-filter)

[<meta-data />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#meta-data)

[</activity-alias>](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#activity-alias)

[<service>](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#service)

[<intent-filter>](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#intent-filter)

[<action />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#action)

[<category />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#category)

[<data />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#data)

[</intent-filter>](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#intent-filter)

[<meta-data />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#meta-data)

[</service>](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#service)

[<receiver>](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#receiver)

[<intent-filter>](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#intent-filter)

[<action />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#action)

[<category />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#category)

[<data />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#data)

[</intent-filter>](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#intent-filter)

[<meta-data />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#meta-data)

[</receiver>](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#receiver)

[<provider>](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#provider)

[<grant-uri-permission />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#grant-uri-permission)

[<path-permission />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#path-permission)

[<meta-data />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#meta-data)

[</provider>](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#provider)

[<uses-library />](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#uses-library)

[</application>](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#application)

[</manifest>](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#manifest)

### <manifest>

Элемент **<manifest>** является корневым элементом манифеста. По умолчанию Eclipse создает элемент с четырьмя атрибутами:

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="ru.alexanderklimov.helloandroid"

android:versionCode="1"

android:versionName="l.0">

#### Атрибуты

**xmlns:android**

определяет пространство имен Android. Оно всегда одно и то же

**package**

определяет уникальное имя пакета приложения, которое вы задали при создании проекта. Android Marketplace проверяет уникальность при приеме приложения, поэтому рекомендуется использовать свое имя для избежания конфликтов с другими разработчиками. Например, я использую имя своего сайта в обратном порядке: ru.alexanderklimov.appname

**android:versionCode**

указывает на внутренний номер версии, используемый для сравнения версий программы. «versionCode» должен быть целым, и Android Market использует это для определения, предоставили ли вы новую версию, передавая триггеру обновления на устройствах, на которых установлено ваше приложение. Как правило. начинается с 1 и увеличивается на единицу, если вы выпускаете новую версию приложения.

**android:versionName**

указывает номер пользовательской версии. Можно использовать строку или строковый ресурс. Этот номер видит пользователь.

Читайте совет [Получить номер версии программы](http://developer.alexanderklimov.ru/android/apps.php#version)

### <permission>

Элемент **<permission>** объявляет разрешение, которое используется для ограничения доступа к определенным компонентам или функциональности данного приложения. В этой секции описываются права, которые должны запросить другие приложения для получения доступа к вашему приложению. Приложение может также защитить свои собственные компоненты (деятельности, службы, приемники широковещательных намерений и контент-провайдеры) разрешениями. Оно может использовать любое из системных разрешений, определенных Android или объявленных другими приложениями, а также может определить свои собственные разрешения.

**android:name**

название разрешения

**android:label**

имя разрешения, отображаемое пользователю

**android:description**

описание разрешения

**android:icon**

значок разрешения

**android:permissionGroup**

определяет принадлежность к группе разрешений

**android:protectionLevel**

уровень защиты

### <uses-permission>

Элемент **<uses-permission>** запрашивает разрешение, которые приложению должны быть предоставлены системой для его нормального функционирования. Разрешения предоставляются во время установки приложения, а не во время его работы.

**android:name**

<uses-permission> имеет единственный атрибут с именем разрешения android:name. Это может быть разрешение, определенное в элементе <permission> данного приложения, разрешение, определенное в другом приложении или одно из стандартных системных разрешений, например: **android:name="android.permission.CAMERA"** или **android:name=""android.permission.READ\_CONTACTS"**

Наиболее распространенные разрешения

* **INTERNET** - доступ к интернету
* **READ\_CONTACTS** - чтение (но не запись) данных из адресной книги пользователя
* **WRITE\_CONTACTS** - запись (но не чтение) данных из адресной книги пользователя
* **RECEIVE\_SMS** - обработка входящих SMS
* **ACCESS\_COARSE\_LOCATION** - использование приблизительного определения местонахождения при помощи вышек сотовой связи или точек доступа Wi-Fi
* **ACCESS\_FINE\_LOCATION** - точное определение местонахождения при помощи GPS

#### <permission-tree>

Элемент <permission-tree> объявляет базовое имя для дерева разрешений. Этот элемент объявляет не само разрешение, а только пространство имен, в которое могут быть помещены дальнейшие разрешения.

#### <permission-group>

Элемент **<permission-group>** определяет имя для набора логически связанных разрешений. Это могут быть как объявленные в этом же манифесте с элементом <permission> разрешения, так и объявленные в другом месте. Этот элемент не объявляет разрешение непосредственно, только категорию, в которую могут быть помещены разрешения. Разрешение можно поместить в группу, назначив имя группы в атрибуте **permissionGroup** элемента <permission>.

#### <instrumentation>

Элемент **<instrumentation>** объявляет объект *instrumentation*, который дает возможность контролировать взаимодействие приложения с системой. Обычно используется при отладке и тестировании приложения и удаляется из release-версии приложения.

#### <uses-sdk>

Элемент **<uses-sdk>** позволяет объявлять совместимость приложения с указанной версией (или более новыми версиями API) платформы Android. Уровень API, объявленный приложением, сравнивается с уровнем API системы мобильного устройства, на который инсталлируется данное приложение.

##### **Атрибуты**

**android:minSdkVersion**

определяет минимальный уровень API, требуемый для работы приложения. Система Android будет препятствовать тому, чтобы пользователь установил приложение, если уровень API системы будет ниже, чем значение, определенное в этом атрибуте. Вы должны всегда объявлять этот атрибут, например: *android:minSdkVersion="11"*. Вы можете ради интереса установить значение 7, а потом 11 и сравнить внешний вид приложения. Например, у младшей версии не будет отображаться ActionBar.

**android:maxSDKVersion**

позволяет определить самую позднюю версию, которую готова поддерживать ваша программа. Ваше приложение будет невидимым в Google Play для устройств с более свежей версией. Рекомендуется устанавливать в том случае, когда вы точно уверены, что приложение не будет корректно работать на новой платформе.

**targetSDKVersion**

позволяет указать платформу, для которой вы разрабатывали и тестировали приложение. Устанавливая значение для этого атрибута, вы сообщаете системе, что для поддержки этой конкретной версии не требуется никаких изменений.

#### <uses-configuration>

Элемент **<uses-configuration>** указывает требуемую для приложения аппаратную и программную конфигурацию мобильного устройства. Например, приложение могло бы определить требования обязательного наличия на устройстве физической клавиатуры или USB-nopTa. Спецификация используется, чтобы избежать установки приложения на устройствах, которые не поддерживают требуемую конфигурацию. Если приложение может работать с различными конфигурациями устройства, необходимо включить в манифест отдельные элементы <uses-configuration> для каждой конфигурации. Вы можете задать любую комбинацию, содержащие следующие устройства

* **reqFiveWayNav** - используйте значение *true*, если приложению требуется устройство ввода, поддерживающее навигацию вверх, вниз, влево, вправо, а также нажатие выделенного элемента. К таким устройствам относятся трекболы и D-pad. В принципе устарело
* **reqHardKeyboard** - используйте значение *true*, если приложению нужна аппаратная клавиатура.
* **reqKeyboardType** - позволяет задать тип клавиатуры: nokeys, qwerty, twelvekey, undefined
* **reqNavigation** - укажите одно из значений: nonav, dpad, trackball, wheel или undefined, если требуется устройство для навигации
* **reqTouchScreen** - если требуется сенсорный экран, то используйте нужное значение из возможных вариантов: notouch, stylus, finger, undefined. Сейчас практически все устройства содержат сенсорный экран, поэтому тоже устарело

Приложение не будет устанавливаться на устройстве, которое не соответствует заданной вами конфигурации. В идеале, вы должны разработать такое приложение, которое будет работать с любым сочетанием устройств ввода. В этом случае **<uses-configuration>** не нужен.

#### <uses-feature>

Элемент **<uses-feature>** объявляет определенную функциональность, требующуюся для работы приложения. Таким образом, приложение не будет установлено на устройствах, которые не имеют требуемую функциональность. Например, приложение могло бы определить, что оно требует камеру с автофокусом. Если устройство не имеет встроенную камеру с автофокусом, приложения не будет инсталлировано.

Пример

**android.hardware.camera**

требуется аппаратная камера

**android.hardware.camera.autofocus**

требуется камера с автоматической фокусировкой

Список можно увидеть [здесь](http://developer.android.com/guide/topics/manifest/uses-feature-element.html).

Можно переопределить требование по умолчанию, добавив атрибут **required** со значением **false**. Например, если вашей программе не требуется, чтобы камера поддерживала автофокус, то используйте вариант:

<uses-feature android:name="android.hardware.camera.autofocus" android:required="false" />

#### <supports-screens>

Элемент **<supports-screens>** определяет разрешение экрана, требуемое для функционирования устройства. Данный тег позволяет указать размеры экран, для которого был спроектировано приложение. Система будет масштабировать ваше приложение на основе ваших макетов на тех устройствах, которые поддерживают указанные вами разрешения экран. Для других случаев система будет растягивать макет по мере возможности.

Возможные значения

**smallScreen**

как правило экраны QVGA

**normalScreen**

стандартные экраны HVGA и WQVGA

**largeScreen**

большие экраны

**xlargeScreen**

очень большие экраны, которые превосходят размеры планшетов

**anyDensity**

установите значение *true*, если ваше приложение способно масштабироваться для отображения на экране с любым разрешением.

По умолчанию, для каждого атрибута установлено значение *true*. Вы можете указать, какие размеры экранов ваше приложение не поддерживает.

<supports-screens

android:smallScreen=["false"]

android:normalScreen=["true"]

android:largeScreen=["true"]

android:anyDensity=["false"] />

Начиная с API 13 (Android 3), у тега появились новые атрибуты:

* **requiresSmallestWidthDp** - указываем минимальную поддерживаемую ширину экрана (наименьшая сторона устройства) в аппаратно-независимых пикселях. С его помощью можно отфильтровать устройства при размещении приложения в Google Play
* **compatibleWidthLimitDp** - задаёт верхнюю границу масштабирования для вашего приложения. Если экран устройства выходит за указанную границу, система включит режим совместимости.
* **largestWidthLimitDp** - задаёт абсолютную верхнюю границу, за пределами которой ваше приложение точно не может быть смаштабировано. В этом случае приложение запускается в режиме совместимости, которую нельзя отключить. Следует избегать подобных ситуаций и разрабатывать макеты для любых экранов.

<supports-screens android:smallScreens="false"

android:normalScreens="true"

android:largeScreens"="true"

android:requiresSmallestWidthDp="480"

android:compatibleWidthLimitDp="600"

android:largestWidthLimitDp="720" />

### <application>

Элемент **<application>** один из основных элементов манифеста, содержащий описание компонентов приложения, доступных в пакете: стили, значок и др. Содержит дочерние элементы, которые объявляют каждый из компонентов, входящих в состав приложения. В манифесте может быть только один элемент **<application>**.

#### <activity>

Элемент **<activity>** объявляет активность. Если приложение содержит несколько активностей, не забывайте объявлять их в манифесте, создавая для каждой из них свой элемент <activity>. Если активность не объявлена в манифесте, она не будет видна системе и не будет запущена при выполнении приложения или будет выводиться сообщение об ошибке.

Для этого класса зарегистрирован фильтр вызовов, определяющий, что это действие запущено в приложении (действие android:name=«android.intent.action.MAIN»). Определение категории (категория android:name=«android.intent.category.LAUNCHER» ) определяет, что это приложение добавлено в директорию приложений на Android-устройстве. Значения @ направляют файлы ресурсов, которые содержат актуальные значения. Это упрощает работу с разными ресурсами, такими как строки, цвета, значки.

Пример:

<activity android:name="ru.alexanderklimov.HelloWorld.AboutActivity"

android:label="@string/app\_name">

##### **Атрибуты**

**android:name**

имя класса. Имя должно включать полное обозначение пакета, но т. к. имя пакета уже определено в корневом элементе <manifest>, имя класса, реализующего деятельность, можно записывать в сокращенном виде, опуская имя пакета

**android:label**

текстовая метка, отображаемая пользователю

**android:launcMode**

управление стеком. [Подробнее](http://developer.alexanderklimov.ru/android/stack.php)

**android:parentActivityName**

В приложениях с API 16 и выше используется эта строка. Она сообщает, какая активность является родительской. Для старых устройств используйте метаданные

<meta-data

android:name="android.support.PARENT\_ACTIVITY"

android:value=".MainActivity" />

**android:exported**

Позволяет или запрещает запускать активность другим приложениям. Если атрибута нет, то по умолчанию равно **false** и другие приложения не могут запускать вашу активность. Но если у активности есть интент-фильтры, то по умолчанию это значение будет уже равно **true**. Это может запутать новичков. Поэтому, если вы хотите запретить запуск через сторонние приложения, то явно указывайте нужное значение.

Элемент <activity> содержит множество других атрибутов, определяющих разрешения, ориентацию экрана и т. д.

##### **Изменение конфигурации во время выполнения программы**

При изменении языка, региона или аппаратной конфигурации Android прерывает работу всех приложений и затем запускает их повторно, перезагружая значения из ресурсов. Подобное поведение не всегда уместно и желательно. Например, некоторые изменения конфигурации (ориентация экрана в пространстве, доступность клавиатуры) могут произойти только лишь из-за того, что пользователь повернул устройство или выдвинул клавиатуру. Вы можете настраивать, каким образом ваше приложение будет реагировать на подобные изменения, обнаруживая их и выполняя собственные действия. Чтобы заставить Активность отслеживать изменения конфигурации при выполнении программы, добавьте в ее узел в манифесте атрибут **android:configChanges**, указав, какие именно события хотите обрабатывать.

Перечислим некоторые значения, с помощью которых можно описать изменения конфигурации:

* **оrientation** — положение экрана изменено с портретного на альбомное (или наоборот);
* **keyboardHidden** — клавиатура (или D-pad и другое устройство) выдвинута или спрятана;
* **fontScale** — пользователь изменил предпочтительный размер шрифта;
* **locale** — пользователь выбрал новые языковые настройки;
* **keyboard** — изменился тип клавиатуры; например, телефон может иметь 12-клавишную панель, при повороте которой появляется полноценная клавиатура. Или была подключена внешняя клавиатура.
* **touchscreen** или **navigation** — изменился тип клавиатуры или способ навигации. Как правило, такие события не встречаются.
* **mcc** или **mnc** - обнаружена новая SIM-карта, при этом изменились страна и код сотовой сети соответственно.
* **uiMode** - изменился режим пользовательского интерфейса, например, при переключении между автомобильным, дневным и ночным режимами.
* **screenLayout** - изменились характеристики экрана, например, при активации другого дисплея.
* **screenSize** - изменлись размеры экрана, например, при смене ориентации. Появилось в Android 3 (API 12)
* **smallestScreenSize** - изменился физический размер экрана, например, при подключении внешнего дисплея. Появилось в Android 3 (API 12)

В некоторых случаях одновременно будут срабатывать несколько событий. Например, когда пользователь выдвигает клавиатуру, большинство устройств генерируют события **keyboardHidden** и **orientation**. Вы можете выбирать несколько событий, которые хотите обрабатывать самостоятельно, разделяя их символом **|**.

Наличие атрибута **android:configChanges** отменяет перезапуск приложения при заданных изменениях конфигурации. Вместо этого внутри активности срабатывает метод **onConfigurationChanged()**. Переопределите его, чтобы появилась возможность обрабатывать изменения в конфигурации. Используйте переданный объект **Configuration**, чтобы получить новые значения. Не забудьте вызвать одноименный метод из родительского класса и перезагрузить измененные значения со всех ресурсов, которые используются внутри активности.

@Override

public void onConfigurationChanged(Configuration \_newConfig) {

super.onConfigurationChanged(\_newConfig);

[ ... Обновите пользовательский интерфейс, используя данные из ресурсов ... ]

if (\_newConfig.orientation == Configuration.ORIENTATION\_LANDSCAPE) {

[ ... Реакция на измененную ориентацию экрана ... ]

}

if (\_newConfig.keyboardHidden == Configuration.KEYBOARDHIDDEN\_NO) {

[ ... Реакция на выдвигание/задвигание клавиатуры ... ]

}

}

На момент вызова метода, все данные из ресурсов будут обновлены, поэтому применять метод можно без опаски.

Любые изменения конфигурации, которые не были явно помечены для обработки внутри вашего приложения, приведут к перезапуску активности, минуя вызов метода **onConfigurationChanged()**.

#### <intent-filter>

Каждый тег **<activity>** поддерживает вложенные узлы **<intent-filter>**. Элемент **<intent-filter>** определяет типы намерений, на которые могут ответить деятельность, сервис или приемник намерений. Фильтр намерений объявляет возможности его родительского компонента — что могут сделать деятельность или служба и какие типы рассылок получатель может обработать. Фильтр намерений предоставляет для компонентов-клиентов возможность получения намерений объявляемого типа, отфильтровывая те, которые не значимы для компонента, и содержит дочерние элементы <action>, <category>, <data>.

#### <action>

Элемент **<action>** добавляет действие к фильтру намерений. Элемент <intent-filter> должен содержать один или более элементов <action>. Если в элементе <intent-fiiter> не будет этих элементов, то объекты намерений не пройдут через фильтр. Пример объявления действия:

<action android:name="android.intent.action.MAIN">

#### <category>

Элемент **<category>** определяет категорию компонента, которую должно обработать намерение. Это строковые константы, определенные в классе intent, например:

<category android:name="android.intent.category.LAUNCHER">

#### <data>

Элемент **<data>** добавляет спецификацию данных к фильтру намерений. Спецификация может быть только типом данных (атрибут mimeType), URI или ТИПОМ данных вместе с URI. Значение URI определяется отдельными атрибутами для каждой из его частей, т. е. URI делитСЯ на части: android:scheme, android:host, android:port, android:path или android:pathPrefix, android:pathPattern.

#### <meta-data>

Элемент **<meta-data>** определяет пару "имя-значение" для элемента дополнительных произвольных данных, которыми можно снабдить родительский компонент. Составляющий элемент может содержать любое число элементов <meta-data>.

#### <activity-alias>

Элемент **<activity-alias>** — это псевдоним для Activity, определенной в атрибуте **targetActivity**. Целевая деятельность должна быть в том же самом приложении, что и псевдоним, и должна быть объявлена перед псевдонимом деятельности в манифесте. Псевдоним представляет целевую деятельность как независимый объект. У псевдонима может быть свой собственный набор фильтров намерений, определяющий, какие намерения могут активизировать целевую деятельность и как система будет обрабатывать эту деятельность. Например, фильтры намерений на псевдониме деятельности могут определить флаги **android:name="android.intent.action.MAIN"** и **android:name="android.intent.category.LAUNCHER"**, заставляя целевую деятельность загружаться при запуске приложения даже в том случае, когда в фильтрах намерений на целевой деятельности эти флаги не установлены.

#### <service>

Элемент **<service>** объявляет службу как один из компонентов приложения. Все службы должны быть представлены элементом <service> в файле манифеста. Службы, которые не были объявлены, не будут обнаружены системой и никогда не будут запущены. Этот элемент имеет много атрибутов, определяющих имя, доступность, разрешения, процесс и т. д. Поддерживает вложенные узлы <intent-fiiter>

#### <receiver>

Элемент **<receiver>** объявляет приемник широковещательных намерений как один из компонентов приложения. Приемники широковещательных намерений дают возможность приложениям получить намерения, которые переданы системой или другими приложениями, даже когда другие компоненты приложения не работают.

#### <provider>

Элемент **<provider>** объявляет контент-провайдера (источник данных) для управления доступом к базам данных. Все контент-провайдеры, которые являются частью приложения, должны быть представлены в элементах <provider> в файле манифеста. Если они не объявлены, они не будут работать, т. к. система их не сможет увидеть. Элемент <provider> содержит свой набор дочерних элементов для установления разрешений доступа к данным:

* <grant-uri-permission>;
* <path-permission>;
* <meta-data>

Этот элемент имеет много атрибутов, определяющих имя, доступность, разрешения, процесс и т. д.

#### <grant-uri-permission>

Элемент **<grant-uri-permission>** является дочерним элементом для <provider>. Он определяет, для кого можно предоставить разрешения на подмножества данных контент-провайдера. Предоставление разрешения является способом допустить к подмножеству данных, предоставляемым контент-провайдером, клиента, у которого нет разрешения для доступа к полным данным. Если атрибут granturiPermissions контент-провайдера имеет значение true, то разрешение предоставляется для любых данных, поставляемых контент-провайдером. Однако, если атрибут поставлен в false, разрешение можно предоставить только подмножествам данных, которые определены этим элементом. Контент-провайдер может содержать любое число элементов <grant-uri-permission>.

#### <path-permission>

Элемент **<path-permission>** — дочерний элемент для <provider>. Определяет путь и требуемые разрешения для определенного подмножества данных в пределах поставщика оперативной информации. Этот элемент может быть определен многократно, чтобы поставлять множественные пути.

#### <uses-library>

Элемент **<uses-library>** определяет общедоступную библиотеку, с которой должно быть скомпоновано приложение. Этот элемент указывает системе на необходимость включения кода библиотеки в загрузчик классов для пакета приложения. Каждый проект связан по умолчанию с библиотеками Android, в которые включены основные пакеты для сборки приложений (с классами общего назначения типа Activity, Service, Intent, View, Button, Application, ContentProvider и т. д.). Однако некоторые пакеты находятся в отдельных библиотеках, которые автоматически не компонуются с приложением. Если же приложение использует пакеты из этих библиотек или других, от сторонних разработчиков, необходимо сделать явное связывание с этими библиотеками и манифест обязательно должен содержать отдельный элемент <uses-library>.

Context

**Context** – это объект, который предоставляет доступ к базовым функциям приложения: доступ к ресурсам, к файловой системе, вызов активности и т.д. **Activity** является подклассом **Context**, поэтому в коде мы можем использовать её как **ИмяАктивности.this**(напр. MainActivity.this), или укороченную запись **this**. Классы **Service**, **Application** и др. также работают с контекстом.

Доступ к контексту можно получить разными способами. Существуют такие методы как **getApplicationContext()**, **getContext()**, **getBaseContext()** или **this**, который упоминался выше, если используется в активности.

На первых порах не обязательно понимать, зачем он нужен. Достаточно помнить о методах, которые позволяют получить контекст и использовать их в случае необходимости, когда какой-нибудь метод или конструктор будет требовать объект **Context** в своих параметрах.

В свою очередь **Context** имеет свои методы, позволяющие получать доступ к ресурсам и другим объектам.

* getAssets()
* getResources()
* getPackageManager()
* getString()
* getSharedPrefsFile()

Возьмём к примеру метод **getAssets()**. Ваше приложение может иметь ресурсы в папке **assets** вашего проекта. Чтобы получить доступ к данным ресурсам, приложение использует механизм контекста, который и отвечает доступность ресурсов для тех, кто запрашивает доступ - активность, служба и т.д. Аналогично происходит с методом **getResources**. Например, чтобы получить доступ к ресурсу цвета используется конструкция **getResources().getColor()**, которая может получить доступ к данным из файла **res/colors.xml**.

Таким образом, создавая, например, вторую активность, мы можем сразу обеспечить ей доступ к своим ресурсам, так как активность относится к контексту. При создании собственных компонентов **View** также используется контекст в конструкторах, так как компонент тоже может использовать ваши ресурсы. При создании собственных классов, если вам нужно будет обращаться к контексту, то необходимо создать конструктор:

// В классе сделайте конструктор, куда будет передаваться контекст:

public MyClass(Context context) {

this.context = context;

}

// в активности

MyClass cat = new MyClass(this);

Через контекст можно узнать практически всю информацию о вашем приложении - имя пакета, класса и т.п.

// имя вашего пакета

String info = this.getApplicationInfo().packageName;

Тем не менее, следует различать контекст в разных ситуациях. Допустим, у вас есть приложение с несколькими активностями. В манифесте можно прописать используемую тему как для всего приложения, так и для каждой активности в отдельности. Соответственно, выбор контекста повлияет на результат. Как правило, при использовании собственной темы предпочтительнее использовать контекст активности, а не приложения.

Очень часто начинающие программисты впадают в ступор, когда ключевое слово **this** не работает в анонимных классах, например, при щелчке кнопки. В этом случае, используйте полное имя класса перед ним.

Button button = (Button) findViewById(R.id.button);

button.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View view) {

Toast.makeText(**MainActivity.this**, "Мяу", Toast.LENGTH\_SHORT).show();

}

});

При создании адаптеров для списков также обращаются к контексту.

if (convertView == null) {

convertView = LayoutInflater.from(**getContext()**).inflate(R.layout.item\_user, parent, false);

}

Или ещё пример для адаптера в фрагменте **ListFragment**:

@Override

public View onCreateView(LayoutInflater inflater, ViewGroup container,

Bundle savedInstanceState) {

String[] names; // имена котов

// бла-бла-бла, т.е. мяу-мяу-мяу

ArrayAdapter<String> adapter = new ArrayAdapter<>(

**inflater.getContext()**, android.R.layout.simple\_list\_item\_1,

names);

setListAdapter(adapter);

return super.onCreateView(inflater, container, savedInstanceState);

}

Здесь тоже следует быть внимательным, если используется своя тема для списка.

Последнее замечание относится к опытным программистам. Неправильный контекст может послужить источником утечки памяти. Если вы создадите собственный класс, в котором содержится статическая переменная, обращающая к контексту активности, то система будет держать ссылку на переменную. Если активность будет закрыта, то сборщик мусора не сможет очистить память от переменной и самой неиспользуемой активности. В таких случаях лучше использовать контекст приложения через метод **getApplicationContext()**.

ContextCompat

В библиотеки совместимости появился свой класс для контекста **ContextCompat**. Он может вам пригодиться, когда студия вдруг подчеркнёт метод в старом проекте и объявит его устаревшим.

context.getResources().getColor(R.color.some\_color\_resource\_id);

Допустим, мы хотим поменять цвет текста на кнопки.

public void onClick(View view) {

int color = getResources().getColor(R.color.colorPrimary); // ругается

Button button = (Button) findViewById(R.id.button);

button.setTextColor(color);

}

Студия ругается, что нужно использовать новый вариант **getColor(int, Theme)**. Заменим строчку.

// Теперь не ругается

int color = ContextCompat.getColor(this, R.color.colorPrimary);

Если посмотреть на исходники этого варианта, то увидим, что там тоже идёт вызов нового метода. Поэтому можно сразу использовать правильный вариант, если вы пишете под Marshmallow и выше.

// Только для API 23 и выше

int color = getResources().getColor(R.color.colorPrimary, getTheme());

# Activity (Активность, Деятельность)

[Что такое Activity](http://developer.alexanderklimov.ru/android/theory/activity-theory.php#what)  
[Методы](http://developer.alexanderklimov.ru/android/theory/activity-theory.php#metod)  
[Активность в виде диалогового окна](http://developer.alexanderklimov.ru/android/theory/activity-theory.php#dialog)  
[Програмнное создание активности](http://developer.alexanderklimov.ru/android/theory/activity-theory.php#program)  
[Классы Activity в Android](http://developer.alexanderklimov.ru/android/theory/activity-theory.php#classactivity)

## Что такое Activity

Термин Activity еще не устоялся в русском языке у разработчиков. Некоторые используют слово Активность, другие - Деятельность. Чтобы не путаться в названиях, я буду использовать термин Активность (смотри примечание).

Разработчики со стажем могут воспринимать Активность как форму. Простые приложения состоят из одной активности. Более сложные приложения могут иметь несколько окон, т.е. они состоят из нескольких активностей, которыми надо уметь управлять и которые могут взаимодействовать между собой.

Активность, которая запускается первой, считается главной. Из нее можно запустить другую активность. Причем не только ту, которая относится к нашему приложению, но и другого приложения. Пользователю будет казаться, что все запускаемые им активности являются частями одного приложения, хотя на самом деле они могут быть определены в разных приложениях и работают в разных процессах. Попробуйте воспринимать активности как страницы разных сайтов, открываемых в браузерах по ссылке.

Примечание: Если у слова **activity** взять первые три символа и перемешать, то получим **cat**, впрочем из слова **активность**тоже можно получить кота.

Обычно активность занимает весь экран устройства, но это не является обязательным требованием. Вы можете создавать полупрозрачные и плавающие окна активностей. И с развитием Android такой подход набирает обороты.

Чтобы создать активность, нужно унаследоваться от класса **Activity** и вызвать метод **onCreate()**. В результате мы получим пустой экран. Толку от такого экрана никакого. Поэтому в активность добавляют компоненты, фрагменты с помощью разметки.

## Жизненный цикл активности

Активность имеет жизненный цикл — начало, когда Android создает экземпляр активности, промежуточное состояние, и конец, когда экземпляр уничтожается системой и освобождает ресурсы. Активность может находиться в трех состояниях:

* активная (active или running) — активность находится на переднем плане экрана. Пользователь может взаимодействовать с активным окном;
* приостановленная (paused) — активность потеряла фокус, но все еще видима пользователю. То есть активность находится сверху и частично перекрывает данную активность. Приостановленная активность может быть уничтожена системой в критических ситуациях при нехватке памяти;
* остановленная (stopped) — если данная активность полностью закрыта другой активностью. Она больше не видима пользователю и может быть уничтожена системой, если память необходима для более важного процесса.

Если активность, которая была уничтожена системой, нужно снова показать на экране, она должна быть полностью перезапущена и восстановлена в своем предыдущем состоянии.

Подробнее о [жизненном цикле читайте в отдельной статье](http://developer.alexanderklimov.ru/android/theory/lifecycle.php).

## Активность в виде диалогового окна

Помните, изучая [темы](http://developer.alexanderklimov.ru/android/theme.php), мы создали вторую активность в виде диалогового окна. С таким же успехом вы можете создать окно и для основной активности. Добавим в манифесте для активности строчку:

<activity

android:name="ru.alexanderklimov.test.MainActivity"

android:label="@string/App\_name"

android:theme="@android:style/Theme.Dialog" >

Напишем код для запуска:

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setTitle("Activity");

TextView tv = new TextView(this);

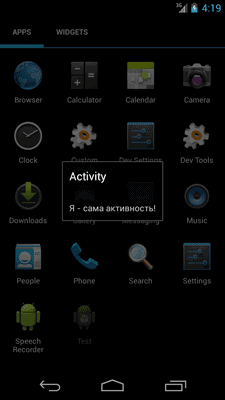
tv.setText("Я - сама активность!");

tv.setPadding(20, 20, 20, 20);

setContentView(tv);

}

Обратите внимание, что в данном примере мы не используем разметку, а программно создаём экземпляр класса TextView с необходимыми свойствами (текст, отступы) и передаём его в метод **setContentView()**.



## Програмнное создание активности

В основном мы пользуемся XML-разметкой для формирования внешнего вида активности. Это рекомендованный способ и в 99% вы будете использовать именно его. Но в некоторых случаях возможно вам понадобится создать активность программно. Сложного там ничего нет, для общего развития пригодится.

Если в стандартном случае мы подключаем XML-файл в методе **setContentView(R.layout.activity\_main)**, то при программном создании данный метод нам не понадобится. Удаляем его и пишем код:

package ru.alexanderklimov.test;

import android.app.Activity;

import android.content.Context;

import android.content.Intent;

import android.os.Bundle;

import android.view.View;

import android.view.View.OnClickListener;

import android.widget.ImageView;

import android.widget.LinearLayout;

import android.widget.LinearLayout.LayoutParams;

import android.widget.TextView;

import android.widget.Toast;

public class TestActivity extends Activity {

/\*\* Called when the activity is first created. \*/

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

// этот метод нам не нужен

// setContentView(R.layout.activity\_test);

LinearLayout layout = new LinearLayout(TestActivity.this);

layout.setId(R.id.layout\_id);

LayoutParams layoutParams = new LayoutParams(LayoutParams.MATCH\_PARENT,

LayoutParams.MATCH\_PARENT);

layout.setLayoutParams(layoutParams);

layout.setOrientation(LinearLayout.VERTICAL);

ImageView imageView = new ImageView(TestActivity.this);

imageView.setId(R.id.image\_id);

imageView.setImageResource(R.drawable.ic\_launcher);

LayoutParams imageViewLayoutParams = new LayoutParams(

LayoutParams.WRAP\_CONTENT, LayoutParams.WRAP\_CONTENT);

imageView.setLayoutParams(imageViewLayoutParams);

layout.addView(imageView);

setContentView(layout);

layout.setOnClickListener(onViewClickListener);

imageView.setOnClickListener(onViewClickListener);

}

OnClickListener onViewClickListener = new OnClickListener() {

@Override

public void onClick(View v) {

int myId = v.getId();

Toast.makeText(TestActivity.this,

"ID: " + String.valueOf(myId) + " clicked",

Toast.LENGTH\_LONG).show();

}

};

}

Принцип формирования следующий - создаем объект LinearLayout (или другой контейнер) и настраиваем необходимые свойства. Далее создаём необходимые дочерние элементы и также настраиваем их свойства. Не забываем подключить дочерний элемент к родительскому через метод **addView()**.

В данном примере мы также присваиваем компонентам идентификаторы через метод **setId()**, хотя необходимости в этом не было. В примере эта возможность показана для демонстрации. Если вы используете идентификаторы, то в файле **res/values/stings.xml**добавьте строчки:

<item type="id" name="layout\_id"/>

<item type="id" name="image\_id" />

Обычно опытные программисты создают отдельный файл для идентификаторов, например, **res/values/ids.xml**.

## Классы Activity в Android

Android SDK включает набор классов, наследованных от Activity. Они предназначены для упрощения работы с виджетами, которые часто встречаются в обычном пользовательском интерфейсе. Перечислим некоторые из них (наиболее полезные).

* **MapActivity**. Инкапсулирует обработку ресурсов, необходимых для поддержки элемента MapView внутри Активности.
* **ListActivity**. Обертка для класса Activity, главная особенность которой — виджет ListView, привязанный к источнику данных, и обработчики, срабатывающие при выборе элемента из списка.
* **ExpandableListActivity**. То же самое, что и ListActivity, но вместо ListView поддерживает ExpandableListView.
* **TabActivity**. Позволяет разместить несколько Активностей или Представлений в рамках одного экрана, используя вкладки для переключения между элементами.

## Методы

Чтобы сгенерировать метод в Android Studio, щёлкните правой кнопкой мыши в области исходного кода и в контекстном меню выберите команду **Generate... (Alt+Insert) | Override Methods…**. В появившемся диалоговом окне отображаются методы, которые могут быть переопределены или реализованы в классе. Либо можете набирать первые символы нужного метода, используя автодополнение.

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

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

Из перечисленных методов в вашем классе обязательно должен быть метод **onCreate()**, которая задает начальную установку параметров при инициализации активности. Вторым по популярности является метод **onPause()**, используемый для сохранения пользовательских настроек активности и подготовиться к прекращению взаимодействия с пользователем.

При реализации любого из этих методов необходимо всегда сначала вызывать версию этого метода из суперкласса. Например:

public void onCreate(Bundle savedInstanceState)

{

super.onCreate(savedInstanceState);

...

}

protected void onPause()

{

super.onPause();

...

}

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

* полное время жизни (entire 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 (foreground 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.

package ru.alexanderklimov.myapplication;

import android.app.Activity;

import android.os.Bundle;

public class MyActivity extends Activity {

// Вызывается при входе в "полноценное" состояние.

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

// Инициализируйте Активность.

}

// Вызывается, когда метод onCreate завершил свою работу,

// и используется для восстановления состояния пользовательского

// интерфейса

@Override

public void onRestoreInstanceState(Bundle savedInstanceState) {

super.onRestoreInstanceState(savedInstanceState);

// Восстановите состояние UI из переменной savedInstanceState.

// Этот объект типа Bundle также был передан в метод onCreate.

}

// Вызывается перед тем, как Активность становится "видимой".

@Override

public void onRestart(){

super.onRestart();

// Загрузите изменения, учитывая то, что Активность

// уже стала "видимой" в рамках данного процесса.

}

// Вызывается в начале "видимого" состояния.

@Override

public void onStart(){

super.onStart();

// Примените к UI все необходимые изменения, так как

// Активность теперь видна на экране.

}

// Вызывается в начале "активного" состояния.

@Override

public void onResume(){

super.onResume();

// Возобновите все приостановленные обновления UI,

// потоки или процессы, которые были "заморожены",

// когда данный объект был неактивным.

}

// Вызывается для того, чтобы сохранить пользовательский интерфейс

// перед выходом из "активного" состояния.

@Override

public void onSaveInstanceState(Bundle savedInstanceState) {

// Сохраните состояние UI в переменную savedInstanceState.

// Она будет передана в метод onCreate при закрытии и

// повторном запуске процесса.

super.onSaveInstanceState(savedInstanceState);

}

// Вызывается перед выходом из "активного" состояния

@Override

public void onPause(){

// "Замораживает" пользовательский интерфейс, потоки

// или трудоемкие процессы, которые могут не обновляться,

// пока Активность не находится на переднем плане.

super.onPause();

}

// Вызывается перед тем, как Активность перестает быть "видимой".

@Override

public void onStop(){

// "Замораживает" пользовательский интерфейс, потоки

// или операции, которые могут подождать, пока Активность

// не отображается на экране. Сохраняйте все введенные

// данные и изменения в UI так, как будто после вызова

// этого метода процесс должен быть закрыт.

super.onStop();

}

// Вызывается перед выходом из "полноценного" состояния.

@Override

public void onDestroy(){

// Очистите все ресурсы. Это касается завершения работы

// потоков, закрытия соединений с базой данных и т. д.

super.onDestroy();

}

}

Как видно из кода, переопределяя эти обработчики, вы всегда должны вызывать одноименные методы родительского класса.

Методы [жизненного цикла описаны в отдельной статье](http://developer.alexanderklimov.ru/android/theory/lifecycle.php). Здесь их опишем кратко и рассмотрим другие методы.

## Метод addContentView()

Метод **addContentView()** добавляет компонент к уже существующей разметке. Пример [смотрите здесь](http://developer.alexanderklimov.ru/android/opengles/opengles1buttons.php).

## Метод findViewById()

Метод **findViewById()** позволяет получить ссылку на **View**, которая размещена в разметке через его идентификатор.

TextView myTextView = (TextView)findViewById(R.id.tvCat);

Если вы используете фрагменты, то когда они загружаются в активность, то компоненты, входящие в состав фрагмента, становятся частью иерархии активности. И вы можете использовать метод **findViewById()** для получения ссылки к компоненту фрагмента.

Не путать с одноимённым методом для класса **View**.

## Метод finish()

C помощью метода **finish()** можно завершить работу активности. Если приложение состоит из одной активности, то этого делать не следует, так как система сама завершит работу приложения. Если же приложение содержит несколько активностей, между которыми нужно переключаться, то данный метод позволяет экономить ресурсы.

## Метод getFragmentManager()

Каждая активность включает в себя *Менеджер фрагментов* для управления фрагментами, если они используются. Метод **getFragmentManager()** позволяет получить доступ к данному менеджеру. На сайте есть отдельные статьи, посвящённые фрагментам.

FragmentManager fragmentManager = getFragmentManager();

## Метод getParentActivityIntent()

Возвращает намерение, которое может запускать активность, являющей родительской. Родительская активность прописывается в манифесте. Вы можете переопределить данное намерение для своих целей. Метод появился в API 16.

@TargetApi(Build.VERSION\_CODES.JELLY\_BEAN)

@Override

public Intent getParentActivityIntent() {

return super.getParentActivityIntent().addFlags(Intent.FLAG\_ACTIVITY\_CLEAR\_TOP);

}

## Метод onActivityResult()

Дочерняя активность может произвольно возвратить назад объект **Intent**, содержащий любые дополнительные данные. Вся эта информация в родительской активности появляется через метод обратного вызова **Activity.onActivityResult()**, наряду с идентификатором, который она первоначально предоставила.

Если дочерняя активность завершится неудачно или будет закрыта пользователем без подтверждения ввода через кнопку Back, то родительская активность получит результат с кодом **RESULT\_CANCELED**.

Метод принимает несколько параметров:

* Код запроса - тот код, который использовался для запуска дочерней активности, возвращающий результат.
* Результирующий код - код результата, поступающий от дочерней активности, как правило, **RESULT\_OK** или **RESULT\_CANCELED**
* Данные - намерение может включать в себя различные данные в виде параметра **extras** внутри намерения.

## Метод onBackPressed()

Метод, позволяющий отследить нажатине на кнопку Back. Появился в Android 2.0 (API 5). Пример использования можно посмотреть в статье [Кнопка Back: Вы уверены, что хотите выйти из программы?](http://developer.alexanderklimov.ru/android/keyboard.php#back).

## Метод onConfigurationChanged()

Метод, который вызывается при изменении конфигурации устройства. Если в манифесте были установлены [специальные параметры у атрибута **android:configChanges**](http://developer.alexanderklimov.ru/android/theory/AndroidManifestXML.php#activity), то данный метод не будет вызван.

## Метод onKeyShortcut()

Смотри [Горячие клавиши в меню](http://developer.alexanderklimov.ru/android/theory/menu.php#shortcut)

## Метод onPostCreate()

Новый метод, который появился в API 21. Он вызывается позже **onCreate()** и в нём можно получить значения размеров компонентов, которые недоступны при построении интерфейса в методе **onCreate()**.

## Метод overridePendingTransition()

Метод **overridePendingTransition()** позволяет задать анимацию при переходе от одной активности к другой. [Пример смотрите здесь](http://developer.alexanderklimov.ru/android/animation/activityoptions.php#overridependingtransition).

## Метод onRestoreInstanceState()

У метода **onRestoreInstanceState()** есть такой же параметр **Bundle**, как у **onCreate()**, и вы можете восстанавливать сохранённые значения из метода **onSaveInstanceState()**. Во многих случаях это пример личных предпочтений, какой из двух методов использовать для восстановления данных.

Метод вызывается после метода **onStart()**. Система вызывает метод **onRestoreInstanceState()** только в том случае, если имеются сохранённые данные для восстановления. Таким образом вам не нужно проверять **Bundle** на *null*, как в методе **onCreate()**:

public void onRestoreInstanceState(Bundle savedInstanceState) {

// Always call the superclass so it can restore the view hierarchy

super.onRestoreInstanceState(savedInstanceState);

// Restore state members from saved instance

mCurrentScore = savedInstanceState.getInt(STATE\_SCORE);

mCurrentLevel = savedInstanceState.getInt(STATE\_LEVEL);

}

## Метод onSaveInstanceState()

Когда система завершает активность в принудительном порядке, чтобы освободить ресурсы для других приложений, пользователь может снова вызвать эту активность с сохранённым предыдущим состоянием. Чтобы зафиксировать состояние активности перед её уничтожением, в классе активности необходимо реализовать метод **onSaveinstancestate()**.

Сам метод вызывается прямо перед методом **onPause()**. Он предоставляет возможность сохранять состояние пользовательского интерфейса активности в объект **Bundle**, который потом будет передаваться в методы **onCreate()** и **onRestoreInstanceState()**. В объект **Bundle** можно записать параметры, динамическое состояние активности как пары "ключ-значение". Когда активность будет снова вызвана, объект **Bundle** передаётся системой в качестве параметра в методы **onCreate()** и **onRestoreInstanceState()**, которые вызываются после **onStart()**, чтобы один из них или они оба могли установить активность в предыдущее состояние. Прежде чем передавать изменённый параметр **Bundle** в обработчик родительского класса, сохраните значения с помощью методов **putXXX()** и восстановите с помощью **getXXX()**.

Используйте обработчик **onSaveInstanceState()** для сохранения состояния интерфейса (например, состояния флажков, текущего выделенного элемента или введенных, но не сохранённых данных), чтобы объект **Activity** при следующем входе в активное состояние мог вывести на экран тот же UI. Рассчитывайте, что перед завершением работы процесса во время активного состояния будут вызваны обработчики **onSaveInstanceState** и **onPause**.

В отличие от базовых методов, методы **onSaveInstanceState()** и **onRestoreInstanceState()** не относятся к методам жизненного цикла активности. Система будет вызывать их не во всех случаях. Например, Android вызывает **onSaveinstancestate()** прежде, чем активность становится уязвимой к уничтожению системой, но не вызывает его, когда экземпляр активности разрушается пользовательским действием (при нажатии клавиши BACK). В этом случае нет никаких причин для сохранения состояния активности.

Метод **onSaveInstanceState()** вызывается системой в случае изменения конфигурации устройства в процессе выполнения приложения (например, при вращении устройства пользователем или выдвижении физической клавиатуры устройства.

Поскольку метод **onSaveinstanceState()** вызывается не во всех случаях, его необходимо использовать только для сохранения промежуточного состояния активности. Для сохранения данных лучше использовать метод **onPause()**.

private static final String TEXTVIEW\_STATE\_KEY = "TEXTVIEW\_STATE\_KEY";

static final String STATE\_SCORE = "playerScore";

static final String STATE\_LEVEL = "playerLevel";

@Override

public void onSaveInstanceState(Bundle saveInstanceState) {

// получаем ссылку на текстовую метку

TextView myTextView = (TextView)findViewById(R.id.textView);

// Сохраняем его состояние

saveInstanceState.putString(TEXTVIEW\_STATE\_KEY, myTextView.getText().toString());

// Сохраняем состояние игрока

savedInstanceState.putInt(STATE\_SCORE, mCurrentScore);

savedInstanceState.putInt(STATE\_LEVEL, mCurrentLevel);

// всегда вызывайте суперкласс для сохранения состояний видов

super.onSaveInstanceState(saveInstanceState);

}

Этот обработчик будет срабатывать всякий раз, когда жизненный цикл активности начнёт подходить к концу, но только в том случае, если её работа не будет завершена явно (при вызове метода **finish()**). Вследствие этого обработчик используется для проверки целостности состояния активности между активными жизненными циклами одиночной пользовательской сессии.

Сохранённый параметр **Bundle** передается методам **onRestoreInstanceState()** и **onCreate()**, если приложение принудительно перезапускается на протяжении сессии. В листинге показано, как извлечь значения из этого параметра и использовать их для обновления состояния экземпляра активности.

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

TextView myTextView = (TextView)findViewById(R.id.textView);

String text = "";

if (savedInstanceState != null && savedInstanceState.containsKey(TEXTVIEW\_STATE\_KEY))

text = savedInstanceState.getString(TEXTVIEW\_STATE\_KEY);

myTextView.setText(text);

}

Помните, что обработчик onSaveInstanceState() вызывается только тогда, когда активность переходит в пассивное состояние, а не когда она закрывается при вызове метода finish() или пользователь нажимает кнопку Назад.

Примеры использования можно увидеть в статьях [Секундомер](http://developer.alexanderklimov.ru/android/stopwatch.php), [Программное удаление пункта меню](http://developer.alexanderklimov.ru/android/theory/menu.php#removemenu) и [Прячем и показываем ActionBar](http://developer.alexanderklimov.ru/android/actionbar.php#hide)

## Метод onUserLeaveHint()

Позволяет отследить нажатие кнопки Home

@Override

protected void onUserLeaveHint() {

Toast toast = Toast.makeText(getApplicationContext(), "Нажата кнопка HOME", Toast.LENGTH\_SHORT);

toast.show();

super.onUserLeaveHint();

}

## Метод requestWindowFeature()

Метод позволяет задействовать дополнительные возможности для активности, например, выводить экран активности без заголовка. Примеры смотрите [здесь](http://developer.alexanderklimov.ru/android/customtitlebar.php#requestwindowfeature).

## Метод onWindowFocusChanged()

Метод позволяет определить момент получения фокуса вашим приложением.

@Override

public void onWindowFocusChanged(boolean hasFocus) {

super.onWindowFocusChanged(hasFocus);

if(hasFocus)

Toast.makeText(this, "Focus", Toast.LENGTH\_LONG).show();

}

Метод может быть полезен, так как он срабатывает позже метода **onCreate()**. Например, для вычисления размеров кнопки на экране этот метод предпочтительнее, так как уже известно, что все элементы загрузились и доступны, тогда как в **onCreate()** могут возвратиться пустые значения ширины и высоты кнопки. [Пример использования](http://developer.alexanderklimov.ru/android/popupwindow.php#custompopup).

Другой пример: [Получить координаты компонента](http://developer.alexanderklimov.ru/android/views/ongloballayoutlistener.php#getlocation)

## Метод setContentView()

Изначально экран активности пуст. Чтобы разместить пользовательский интерфейс, необходимо вызвать метод **setContentView()**. У метода есть две перегруженные версии. Вы можете передать в параметре либо экземпляр компонента (View), либо идентификатор ресурса (наиболее распространённый способ).

Пример с использованием экземпляра компонента:

@Override

public void onCreate(Bundle saveInstanceState) {

super.onCreate(saveInstanceState);

TextView myTextView = new TextView(this);

**setContentView(myTextView);**

myTextView.setText("Hello, Kitty");

}

В этом примере вы увидите на экране текстовое поле с текстом. Но при таком способе вы можете использовать только один компонент. А если экран состоит из множества кнопок и прочих элементов управления, то нужно использовать разметку.

## Метод setFeatureDrawableResource()

С помощью данного метода можно вывести значки в правой части заголовка. Смотри [пример](http://developer.alexanderklimov.ru/android/customtitlebar.php#icon).

## Метод setRequestedOrientation()

Метод позволяет программно изменить ориентацию экрана. [Пример использования](http://developer.alexanderklimov.ru/android/orientation.php#setorientation).

## Метод startActivity()

Чтобы запустить новую активность, используется метод **startActivity(Intent)**. Этот метод принимает единственный параметр — объект **Intent**, описывающий активность, которая будет запускаться. Смотри пример [Activity](http://developer.alexanderklimov.ru/android/activity.php).

## Метод startActivityForResult()

Иногда требуется вернуть результат активности, когда она закрывается. Например, можно запустить активность, которая позволяет пользователю выбирать человека в списке контактов. При закрытии активность возвращает данные человека, который был выбран: его полное имя и телефон. В этом случае необходимо вызвать метод **startActivityForResult()**

Метод **startActivityForResult(Intent, int)** со вторым параметром, идентифицирующим запрос позволяет возвращать результат. Когда дочерняя активность закрывается, то в родительской активности срабатывает метод **onActivityResult(int, int, Intent)**, который содержит возвращённый результат, определённый в родительской активности.

// идентификатор запроса

private static final int SHOW\_SUBACTIVITY = 1;

Intent intent = new Intent();

// определение класса запускаемой активности

intent.setClass(this, NewContactActivity.class);

// вызов активности

startActivityForResult(intent, SHOW\_SUBACTIVITY);

## Метод setResult()

Когда активность завершится, вы можете вызвать метод **setResult(int)**, чтобы возвратить данные назад в родительскую активность (до метода **finish()**). Этот метод возвращает код результата закрытия активности, который может быть стандартными результатами **Activity.RESULT\_CANCELED**, **Activity.RESULT\_OK** или определяемым пользователем результатом **RESULT\_FiRST\_USER** (можете придумать любую константу с целочисленным значением).

private EditText mName;

private EditText mPhone;

Intent intent new Intent();

// вставляем имя человека

intent.putExtra(ContactListActivity.NAME, mName.getText().toString());

// вставляем номер телефона

intent.putExtra(ContactListActivity.PHONE, mPhone.getText().toString());

// возвращаем результат в вызывающую активность

setResult(RESULT\_OK, intent);

finish();

Если в дочерней активности есть кнопка отмены, то код может быть следующим:

setResult(RESULT\_CANCELED);

finish();

Если метод **finish()** вызвать раньше метода **setResult()**, то результирующий код установится в **RESULT\_CANCELED** автоматически, а возвращённое намерение покажет значение *null*.

# Жизненный цикл приложения на Android



Жизненный цикл приложения в Android жёстко контролируется системой и зависит от нужд пользователя, доступных ресурсов и т. д. Например, пользователь хочет запустить браузер. Решение о запуске приложения принимает система. Хотя последнее слово и остаётся за системой, она подчиняется определённым заданным и логическим правилам, позволяющим определить, можно ли загрузить, приостановить приложение или прекратить его работу. Если в данный момент пользователь работает с определённым окном, система даёт приоритет соответствующему приложению. И наоборот, если окно невидимо и система решает, что работу приложения необходимо остановить, чтобы мобилизовать дополнительные ресурсы, будет прекращена работа приложения, имеющего более низкий приоритет. В Android ресурсы более ограниченны, поэтому Android более жёстко контролирует работу приложений.

## Основные методы жизненного цикла приложения

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

### onCreate()

Метод **onCreate()** вызывается при создании или перезапуска активности. Система может запускать и останавливать текущие окна в зависимости от происходящих событий. Внутри данного метода настраивают статический интерфейс активности. Инициализирует статические данные активности, связывают данные со списками и т.д. Связывает с необходимыми данными и ресурсами. Задаёт внешний вид через метод **setContentView()**.

В этом методе загружайте пользовательский интерфейс, размещайте ссылки на свойства класса, связывайте данные с элементами управления, создавайте сервисы и потоки. Метод **onCreate()** принимает объект **Bundle**, содержащий состояние пользовательского интерфейса, сохранённое в последнем вызове обработчика [onSaveInstanceState](http://developer.alexanderklimov.ru/android/theory/activity-theory.php#onsaveinstancestate). Для восстановления графического интерфейса в его предыдущем состоянии нужно задействовать эту переменную: внутри **onCreate()** или переопределив метод **onRestoreInstanceState()**.

Операции по инициализации, занимающие много времени, следует выполнять в фоновом процессе, а не с помощью метода **onCreate()**. В противном случае можно получить диалоговое окно ANR (Application Not Responding, приложение не отвечает).

В методе можно сделать проверку, запущено ли приложение впервые или восстановлено из памяти. Если значение переменной **savedInstanceState** будет **null**, приложение запускается первый раз:

// Приложение запущено впервые или восстановлено из памяти?

if ( savedInstanceState == null ) // приложение запущено впервые

{

currentBillTotal = 0.0; // инициализация суммы счета нулем

// другой код

}

else // приложение восстановлено из памяти

{

// инициализация суммы счета сохраненной в памяти суммой

currentBillTotal = savedInstanceState.getDouble(BILL\_TOTAL);

}

А значение переменной *currentBillTotal* можно сохранить в методе **onSaveInstanceState()**:

@Override

protected void onSaveInstanceState(Bundle outState) {

super.onSaveInstanceState(outState);

outState.putDouble(BILL\_TOTAL, currentBillTotal);

}

### onStart()

За **onCreate()** всегда следует вызов **onStart()**, но перед **onStart()** не обязательно должен идти **onCreate()**, так как **onStart()** может вызываться и для возобновления работы приостановленного приложения (приложение останавливается методом **onStop()**). При вызове **onStart()** окно еще не видно пользователю, но вскоре будет видно. Вызывается непосредственно перед тем, как активность становится видимой пользователю. Сопровождается вызовом метода **onResume()**, если активность получает передний план, или вызовом метода **onStop()**, если становится скрытой.

### onResume()

Метод **onResume()** вызывается после **onStart()**, даже когда окно работает в приоритетном режиме и пользователь может его наблюдать. В этот момент пользователь взаимодействует с созданным вами окном. Приложение получает монопольные ресурсы. Запускает воспроизведение анимации, аудио и видео. Также может вызываться после **onPause()**.

Имейте в виду, что система вызывает этот метод каждый раз, когда ваша активность идёт на переднем плане, в том числе, при первом создании. Таким образом, вы должны реализовать **onResume()** для инициализации компонентов, регистрации любых широковещательных приемников или других процессов, которые вы освободили/приостановили в **onPause()** и выполнять любые другие инициализации, которые должны происходить, когда активность вновь активна.

Пытайтесь размещать относительно быстрый и легковесный код, чтобы ваше приложение оставалось отзывчивым при скрытии с экрана или выходе на передний план.

Вам не нужно перезагружать состояние пользовательского интерфейса внутри него, так как эти функции возложены на обработчики **onCreate()** и **onRestoreInstanceState**.

Например, после метода **onPause()**, в котором мы приостановили работу камеры (см. ниже) снова запускаем камеру:

@Override

public void onResume() {

super.onResume();

// Get the Camera instance as the activity achieves full user focus

if (mCamera == null) {

initializeCamera(); // Local method to handle camera init

}

}

### onPause()

Когда пользователь решает перейти к работе с новым окном, система вызовет для прерываемого окна метод **onPause()**. По сути происходит свёртывание активности. Сохраняет незафиксированные данные. Деактивирует и выпускает монопольные ресурсы. Останавливает воспроизведение видео, аудио и анимацию. От **onPause()** можно перейти к вызову либо **onResume()**, либо **onStop()**.

В этом методе необходимо остановить анимацию и другие действия, которые загружают процессор. Зафиксировать несохранённые данные, например, черновик письма, потому как после его выполнения работа активности может прерваться без предупреждения. Освободить системные ресурсы, например, обработку данных от GPS.

Пытайтесь размещать относительно быстрый и легковесный код, чтобы ваше приложение оставалось отзывчивым при скрытии с экрана или выходе на передний план.

Исходя из архитектуры своего приложения, вы также можете приостановить выполнение потоков, процессов или широковещательных приемников, пока активность не появится на переднем плане.

Например, при работе с [камерой](http://developer.alexanderklimov.ru/android/camera.php) метод используется следующим образом:

@Override

public void onPause() {

super.onPause();

// Release the Camera because we don't need it when paused

// and other activities might need to use it.

if (mCamera != null) {

mCamera.release()

mCamera = null;

}

}

В тоже время вы не должны использовать **onPause()** для хранения пользовательских изменений (таких, как персональные данные, введённые в форму) для постоянного хранения. Исключение допускается, когда вы уверены, что пользователи ожидают изменения, которые будут автоматически сохранены (например, при составлении электронной почты). Тем не менее, вы должны избегать выполнения интенсивной работы в **onPause()**, таких как запись в базе данных, так как это может замедлить переход к следующей активности (вместо него вы должны выполнять тяжелую нагрузку во время операции отключения **onStop()**).

Когда активность приостановлена, то все компоненты сохраняются в памяти и при возобновления нет необходимости повторно инициализировать их.

### onStop()

Метод **onStop()** вызывается, когда окно становится невидимым для пользователя. Это может произойти при её уничтожении, или если была запущена другая активность (существующая или новая), перекрывшая окно текущей активности. Всегда сопровождает любой вызов метода **onRestart()**, если активность возвращается, чтобы взаимодействовать с пользователем, или метода **onDestroy()**, если эта активность уничтожается.

Когда ваша активность останавливается, объекты активности хранятся в памяти и восстанавливаются, когда активность возобновляет свою работу. Вам не нужно повторно инициализировать компоненты, которые были созданы ранее. Кроме того, система отслеживает текущее состояние для каждого представления, поэтому, если пользователь введёт текст в текстовое поле, то его содержание сохраняется и вам не нужно сохранять и восстанавливать его.

Примечание: Даже если система закрыла вашу активность, когда она была остановлена, она по-прежнему сохраняет состояние объектов, таких как текст в **EditText** в специальном объекте **Bundle** (в виде ключ-значение) и восстанавливает их, если пользователь переходит обратно к тому же экземпляру активности.

В этом методе можно сделать сложные операции по сохранению данных: для приостановки сложной анимации, потоков, отслеживания показаний датчиков, запросов к GPS, таймеров, сервисов или других процессов, которые нужны исключительно для обновления пользовательского интерфейса. Нет смысла потреблять ресурсы (такты центрального процессора или сетевой трафик) для обновления интерфейса, в то время как он не виден на экране. Примените методы **onStart()** или **onRestart()** для возобновления или повторного запуска этих процессов, когда активность опять станет видимой.

При нехватке памяти система может уничтожить скрытую активность, минуя метод **onStop()** с вызовом метода **onDestroy()**.

### onRestart()

Если окно возвращается в приоритетный режим после вызова **onStop()**, то в этом случае вызывается метод **onRestart()**. Т.е. вызывается после того, как активность была остановлена и снова была запущена пользователем. Всегда сопровождается вызовом метода **onStart()**.

**onRestart** предшествует вызовам метода **onStart()** (кроме самого первого). Используйте его для специальных действий, которые должны выполняться только при повторном запуске активности в рамках «полноценного» состояния.

### onDestroy()

Метод вызывается по окончании работы активности, при вызове метода **finish()** или в случае, когда система уничтожает этот экземпляр активности для освобождения ресурсов. Эти два сценария уничтожения можно определить вызовом метода **isFinishing()**. Вызывается перед уничтожением активности. Это последний запрос, который получает активность от системы. Если определённое окно находится в верхней позиции в стеке, но невидимо пользователю и система решает завершить это окно, вызывается метод **onDestroy()**. В этом случае метод удаляет все статические данные активности. Отдаёт все используемые ресурсы.

Так как все необходимые операции по освобождению ресурсов вы сделали в методе **onStop()**, то в этом методе вы можете подстраховаться и проверить ещё раз все неосвобождённые ресурсы.

На практике вам чаще всего придется сталкиваться с методами **onCreate()**, **onResume()** и **onPause()**. Метод **onCreate()** будет вызываться при создании пользовательского интерфейса для работы с окном. Данный метод позволит вам связывать данные с компонентами и подключать обработчики событий к компонентам пользовательского интерфейса. При помощи **onPause()** вы сможете сохранить важную информацию в базе данных вашего приложения. Это последний безопасный метод, который будет вызываться перед тем, как система завершит работу приложения. Метод **onDestroy()** не обязательно будет вызываться, поэтому не полагайтесь на этот метод при реализации критическом логики.

## Пример

Для наглядности создадим пример, где для каждого метода жизненного цикла подготовим всплывающее сообщение, а также для подстраховки будем вести запись в журнале. Мы сможем воочию увидеть весь жизненный цикл.

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:tools="http://schemas.android.com/tools"

android:id="@+id/LinearLayout1"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical"

tools:context=".MainActivity">

<TextView

android:id="@+id/textViewInfo"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:text="Приложение запущено впервые"

android:textSize="24dp" />

<Button

android:id="@+id/buttonTouchMe"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:onClick="onClick"

android:text="Коснись меня!" />

<Button

android:id="@+id/buttonExit"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:onClick="onClick"

android:text="Выход" />

</LinearLayout>

Код для методов:

package ru.alexanderklimov.lifecycle;

import android.support.v7.app.ActionBarActivity;

import android.os.Bundle;

import android.util.Log;

import android.view.Menu;

import android.view.MenuItem;

import android.view.View;

import android.widget.TextView;

import android.widget.Toast;

public class MainActivity extends ActionBarActivity {

private String TAG = "Жизненный цикл";

private TextView mInfoTextView;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

mInfoTextView = (TextView) findViewById(R.id.textViewInfo);

Toast.makeText(getApplicationContext(), "onCreate()", Toast.LENGTH\_SHORT).show();

Log.i(TAG, "onCreate()");

}

@Override

protected void onStart() {

super.onStart();

Toast.makeText(getApplicationContext(), "onStart()", Toast.LENGTH\_SHORT).show();

Log.i(TAG, "onStart()");

}

@Override

protected void onResume() {

super.onResume();

Toast.makeText(getApplicationContext(), "onResume()", Toast.LENGTH\_SHORT).show();

Log.i(TAG, "onResume()");

}

@Override

protected void onPause() {

super.onPause();

Toast.makeText(getApplicationContext(), "onPause()", Toast.LENGTH\_SHORT).show();

Log.i(TAG, "onPause()");

}

@Override

protected void onStop() {

super.onStop();

Toast.makeText(getApplicationContext(), "onStop()", Toast.LENGTH\_SHORT).show();

Log.i(TAG, "onStop()");

}

@Override

protected void onRestart() {

super.onRestart();

Toast.makeText(getApplicationContext(), "onRestart()", Toast.LENGTH\_SHORT).show();

Log.i(TAG, "onRestart()");

}

@Override

protected void onDestroy() {

super.onDestroy();

Toast.makeText(getApplicationContext(), "onDestroy()", Toast.LENGTH\_SHORT).show();

Log.i(TAG, "onDestroy()");

}

public void onClick(View v) {

switch (v.getId()) {

case R.id.buttonTouchMe:

mInfoTextView.setText("Приложение уже было запущено!");

break;

case R.id.buttonExit:

finish();

break;

default:

break;

}

}

}

Запускайте проект и следите за сообщениями. Они будут всплывать в нужной последовательности, давая вам представление о жизненном цикле приложения. Также сообщения будут транслироваться в окно **logcat** на вкладке **6.Android** в студии. Обратите внимание на следующий момент. Когда ваше приложение запущено, то нажмите на первую кнопку, чтобы изменить текст в **TextView**. Затем нажмите кнопку Home (не Back!), чтобы попасть на Домашний экран. После чего снова запустите ваше приложение. Вы увидите, что приложение не вызывает метод **onCreate()**, а текст в **TextView** будет свидетельствовать, что приложение не было закрыто, а только свёрнуто. Это очень важный момент, который нужно понять. Понимание этих вещей поможет вам правильно выстраивать логику приложения.

## Памятка

Запуск приложения

onCreate() → onStart() → onResume()

Нажимаем кнопку Назад для выхода из приложения

onPause() → onStop() → onDestroy()

Нажата кнопка Домой

onPause() → onStop()

После нажатия кнопки Домой, когда приложение запущено из списка недавно открытых приложений или через значок

onRestart() → onStart() → onResume()

Когда запускается другое приложение из области уведомлений или открывается приложение Настройки

onPause() → onStop()

Нажата кнопка Назад в другом приложении или в Настройках и ваше приложение стало снова видимым.

onRestart() → onStart() → onResume()

Открывается диалоговое окно

onPause()

Диалоговое окно закрывается

onResume()

Кто-то звонит на телефон

onPause() → onResume()

Пользователь отвечает на звонок

onPause()

Разговор окончен

onResume()

Экран телефона гаснет

onPause() → onStop()

Экран снова включён

onRestart() → onStart() → onResume()

На китайских планшетах иногда наблюдал, когда какие-то методы не срабатывали.

При повороте активность проходит через цепочку различных состояний. Порядок следующий.

onPause()  
onStop()  
onDestroy()  
onCreate()  
onStart()  
onResume()

## Порядок вызовов

После **onCreate()** - **onStart()**

После **onRestart()** - **onStart()**

После **onStart()** - **onResume()** или **onStop()**

После **onResume()** - **onPause()**

После **onPause()** - **onResume()** или **onStop()**

После **onStop()** - **onRestart()** или **onDestroy()**

После **onDestroy()** - ничего