Содержание

Неделя 2

Компоненты Android и элементы интерфейса

2.1 Activity и View

2.1.1 Основные компонены Android. Context

Android с точки зрения разработчика это framework, то есть глобальная библиотека, которая берет на себя всю грязную работу, например, связанную с управлением памятью, процессами, приложениями, датчиками и тому подобное. Framework предоставляет нам несколько компонентов, наследуя и изменяя которые, мы строим свое приложение. Всего таких компонентов пять:

- **Application** — это компонент, который первым загружается при начале работы с приложением и умирает последним. Он представляет собой singleton (единственный экземпляр своего класса). Реализация Application необязательна, но если вам нужна глобальная точка доступа к переменными либо методам, то Application — вполне подходящий вариант. Для того, чтобы система знала о вашем Application, необходимо указать его класс в манифесте, в node Application через tagname, иначе будет использоваться стандартная реализация.

- **Activity** — это основной компонент, с помощью которого пользователь взаимодействует с приложением. Упрощённо, Activity представляет собой какой-либо экран приложения, логику в Java-коде и интерфейс в XML-верстке. Все Activity должны наследоваться от базового класса Activity, а в методах должны обязательно вызываться методы родителя. Activity, как и другие компоненты из списка, должны обязательно указываться в манифесте, чтобы система знала о нем и могла запустить.

- **Service** — это компонент, главное предназначение которого выполнять долгие операции, которые не требуют взаимодействия с интерфейсом. Например, музыка играющая при сворачивании приложения.

- **BroadcastReceiver** — это приемник широковещательных сообщений, посылаемых системой, другими приложениями либо вашим приложением, и

каким-либо образом реагирующий на них. Вы сами определяете, на что и как он будет реагировать. Например, приемник, подписанный на включение устройства, а реакция будет в виде запроса на сервер с обновлением данных. К слову, все мессенджеры работают подобным образом.

- **ContentProvider** — это мощный компонент, представляющий собой программный интерфейс, который позволяет нескольким приложениям пользоваться одним источником данных. Например, мессенджер имеющий доступ к телефонной книге, с которой он считывает номера или добавляет новые контакты. Или банковское приложение позволяющие переводить средства по номеру телефона, сравнивая номера из списка контактов и своей базе.

Все эти компоненты должны быть обязательно прописаны в манифесте (AndroidManifest.xml). Во всех этих компонентах так или иначе присутствует класс Context, за реализацию которого отвечает сам Android. Например, Activity — это одна из таких реализаций, так как она является потомком Context. Наиболее частый случай использования Context — это

- доступ к ресурсам приложения, будь то строки, цвета, меню, цифровые константы или RAW-файлы, не поддающиеся классификации.
- доступ к системным возможностям телефона, например, к сканеру отпечатков пальцев, будильнику, акселерометру или другим возможностям
- решение проблемы, когда интерфейс заранее неизвестен и зависит от данных с сервера, то для создания элементов интерфейса прямо в коде обязательно нужно передавать в конструктор этого элемента текущий контекст.
- это создание файлов, хранящихся на устройстве и используемых приложением. К примеру, база данных, скачанные файлы, сделанные фотоснимки или файлы preferences, в которых хранятся настройки приложения.

Класс очень важный, но напрямую с ним не работают - всё происходит через его реализации. К примеру, Context, который реализован в Application, как и сам Application, является singleton. При создании своих собственных singleton, которым необходим Context для работы, следует использовать именно Context Application. Напротив, Context, реализованные в Activity или Service, у каждого экземпляра свои. Логично, что такой Context живет столько, сколько живет Activity или Service. Поэтому хранить ссылки на них — не самая лучшая затея. В крайнем случае можно воспользоваться WeakReference, чтобы сборщик мусора мог утилизировать объект контекста и освободить ресурсы.
Дополнительно про контекст и про различие контекстов

2.1.2 Activity. Жизненный цикл

activity — это компонент, с помощью которого пользователь взаимодействует с вашим предложением через UI. Представляет собой один какой-либо экран приложения (логика работы) в общем случае. В любом приложении может быть несколько activity, каждая из которых представляет какой-либо экран. К примеру, экран авторизации с логикой обработки логина и пароля, главный экран приложения, чаще всего список каких либо элементов, экран настроек и другие варианты. При создании своего activity необходимо наследоваться от базового activity, одной из его реализации или от AppCompatActivity(из библиотеки поддержки, которая упрощает работу с более ранними версиями Android). Для запуска приложения через иконку на главном экране устройства в манифесте обязательно должна быть activity с категорией Launcher. Это и будет главная точка входа в приложение. Activity — это самостоятельные единицы в том плане, что они не зависят от других activity. Являясь наследником контекста, activity имеют доступ ко всем возможностям контекста и даже больше. Activity имеют множество методов оберток, упрощающих доступ к тем или иным ресурсам. Также во все методы, которые принимают контекст, можно передавать activity, и если дело происходит в самом activity, то просто передается ключевое слово this или activity.this.
Рассмотрим такой важный механизм работы activity, как жизненный цикл. Жизненный цикл — это совокупность состояний, через которые проходит каждая activity по ходу своей работы

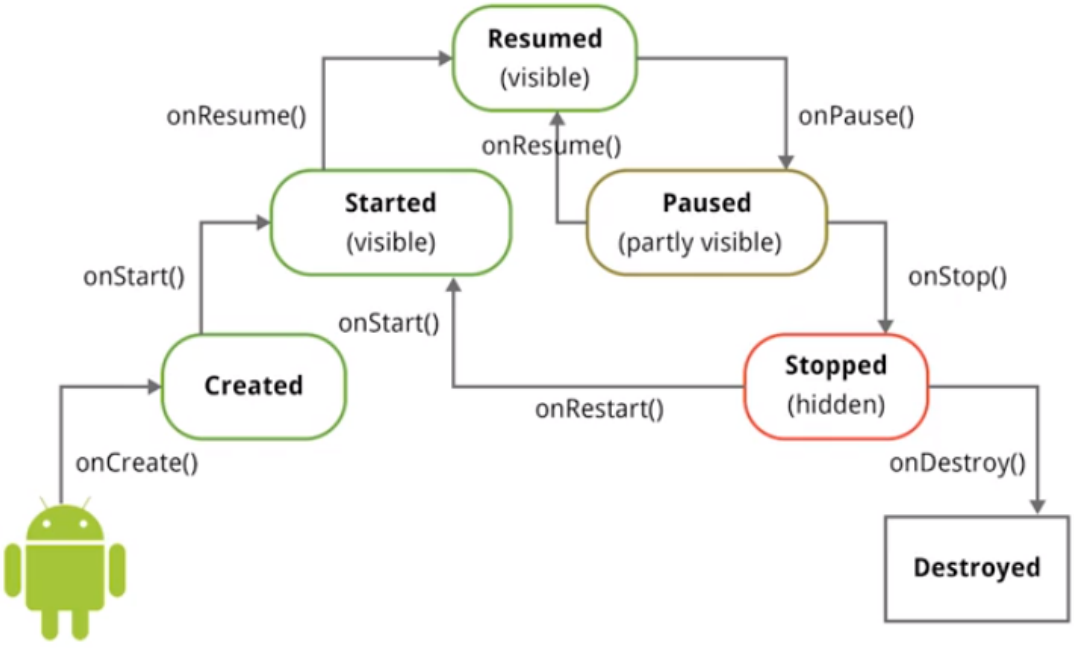


Рис. 2.1: Жизненный цикл Activity (упрощённая версия)

Переходы между состояниями происходят по требованию фреймворка. Различать их мы можем через callbacks (методы обратного вызова). Это упрощённая схема жизненного цикла Activity. Первая тройка callbacks вызывается, когда

activity готовится к взаимодействию с пользователем. Callbacks вызываются в следующем порядке:

1. **OnCreate()** - инициализируем наши поля и проводим первоначальную настройку activity, проверяя входные параметры или предыдущее состояние activity(в качестве аргумента в виде объекта bundle).
   Также задается xml-верстка интерфейса через метод SetContentView().
   После метода OnCreate() activity приходит состояние **Created** (еще не видно для пользователя).

2. **OnStart()** - мы проводим настройку, касающуюся видимой части activity. Например в этом методе мы можем зарегистрировать BroadcastReceiver, если он связан с изменением интерфейса.
   После onStart() activity приходит состояние **Started** (становится видимым для пользователя, но еще не готовым для взаимодействия).

3. **OnResume()** (вызывается сразу же после OnStart()) нам нужно настроить activity, чтобы оно было готово для взаимодействия с пользователем. В частности, мы навешиваем обработчики событий на кнопки, на прокручивающиеся списки, запускаем анимации.
   После вызова этого метода activity приходится в состояние **OnResume()** и остается в нем, пока пользователь работает с ним.

4. **OnPause()** - первый звоночек того, что пользователь собирается покинуть activity. В этом методе необходимо освободить ресурсы, связанные с взаимодействием, то есть все, то что мы добавляли в методе OnResume(). Этот метод скоротечен, и не годится для сохранения каких-либо значений.
   После него activity переходит в состояние **Paused**. Оно остается частично видимым для пользователя и должно делать все то, что ожидается от просто видимого экрана, в частности обновлять интерфейс. После этого метода может вызваться метод OnResume(), и тогда пользователь снова работает с нашим activity, либо OnStop(), и activity окончательно покинута.

5. **OnStop()** теперь activity больше не видимо для пользователя, скорее всего, потому что над ним находится другое activity, либо потому, что приложение закрыто или свернуто. Этот метод — самое лучшее место, чтобы выполнить запись в базу данных. Помимо этого необходимо освободить все ресурсы, которые не нужны, пока пользователь не пользуется activity. Опять же, это все то, что было проинициализировано в методе OnStart().
   После вызова этого метода, activity переходит в состояние **Stopped**.

6. **OnDestroy()** - activity окончательно уничтожено, вся память очищена

Рассмотрим пример запуска родительского activity, с которого мы переходим на дочернее activity и возвращаемся назад. Все activity-приложения собираются в

backstack activity: при запуске нового activity оно добавляется на вершину стека, при нажатии кнопки назад — удаляется.

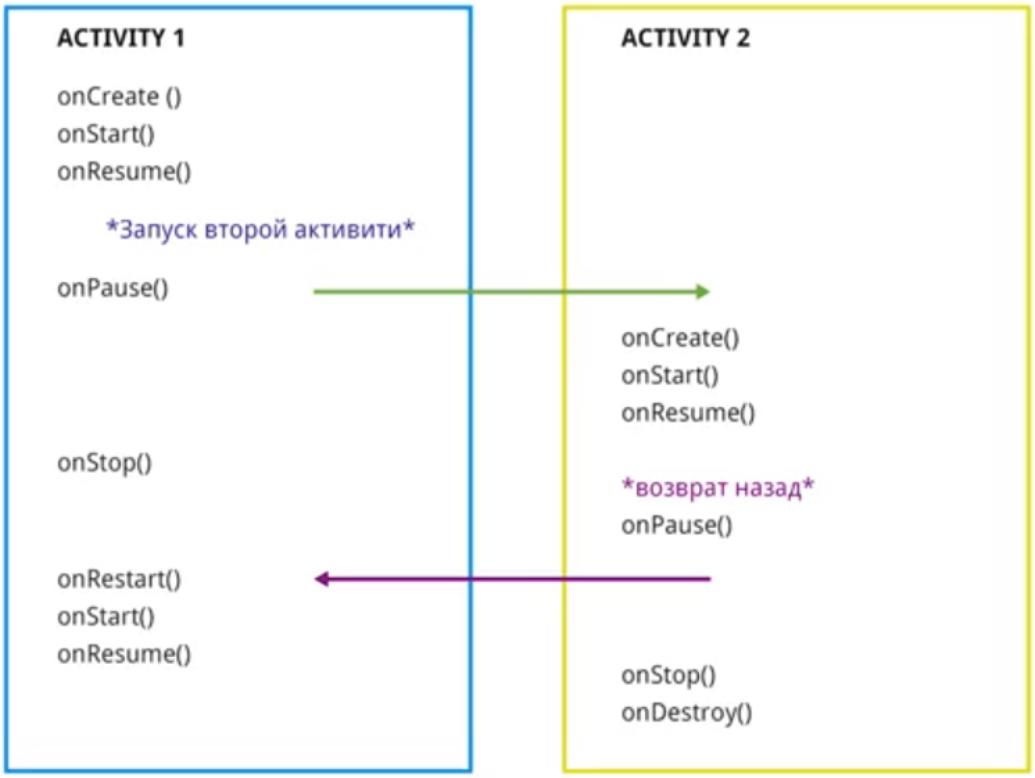


Рис. 2.2: Порядок вызова методов жизненного цикла при запуске дочерней Activity

Посмотрим, как это происходит:

1. Запускается первое activity, у него вызываются методы OnCreate(), OnStart(), OnResume(). Оно становится готовым для взаимодействия, находится на вершине своего стека.

2. Дальше мы запускаем дочернее activity. В родительском activity вызывается метод OnPause(), с ним мы больше не взаимодействуем.

3. У дочернего activity вызываются методы OnCreate(), OnStart() и OnResume(). И оно выходит на передний план. А у родителького activity вызывается метод OnStop(). Теперь родительское activity находится в состоянии Stopped, оно невидимо. Дочернее activity находится в состоянии Resumed, и оно готово к взаимодействию.

4. Если мы сейчас, будучи в дочернем activity, нажмем кнопку назад, то в дочернем activity вызовется метод OnPause(), оно отойдет, у родительского activity, соответственно, OnRestart(), OnStart() и OnResume(), оно станет готовым для взаимодействия. У дочернего Activity — OnStop() и OnDestroy(), оно будет окончательно уничтожено.

Возможны два основных случая уничтожения Activity:

- система считает, что уничтожение activity — это нормальное поведение приложения. Когда нажал кнопку «назад» либо при программном вызове метода finish
- ненормальным считается поведение, когда framework вынужден закрыть activity. Например, если activity, которое находится на вершине стека, испытывает какую-либо нехватку ресурсов, то activity, которые находятся под ним, могут быть уничтожены.

Для сохранения пользовательской работы при ненормальном уничтожении Activity, между OnPause() и OnStop() вызывается метод **OnSaveInstanceState()**, в котором будет сохранено текущее состояние activity. В частности, это касается view-элементов с указанными ID. Помимо этого мы можем переопределить метод OnSaveInstanceState() и дописать туда что-нибудь свое. Теперь, если пользователь вернется на это activity, уничтоженное activity, то вызов метода OnCreate() будет передан bundle с сохраненным стейтом. Если bundle не равен null, то извлекаем данные, которые мы записали.
Также существует метод OnRestoreInstanceState(), который вызывается, только если в bundle что-то записано. Относительно жизненного цикла этот метод вызывается после OnStart() и до OnResume(). Уничтожение activity системой — это не единственный случай сохранения стейта. Другим распространенным вариантом является смена конфигурации (в частности, поворот экрана). Пока что можете использовать вариант с OnSaveInstanceState().

2.1.3 Интерфейс. View и ViewGroup

Для пользователя интерфейс - самая важная часть любого приложения. Android-приложениях логика и интерфейс несколько разделены. Логика находится в Java-классах, а интерфейс в XML-разметке. Подобное разделение очень удобно, так как структура интерфейса сразу бросается в глаза, и в ней легко разобраться. Для того чтобы манипулировать элементами интерфейса в Java-коде, их различают с помощью Id — идентификатора, уникального для данного XML-файла. Элементы интерфейса — обычные Java-классы, поэтому их можно создавать программно, в рантайме, если интерфейс требует такой динамики. Элемент интерфейса, который находится в XML-файле, делится на две главные категории: это View и его наследник ViewGroup. Как достаточно понятно из названия, ViewGroup представляет собой контейнер, в котором по тому или иному принципу располагается View — конкретный элемент интерфейса.
ViewGroup — базовый класс для всех элементов интерфейса, в том числе и для ViewGroup. **ViewGroup** — это базовый класс для всех классов-контейнеров, В принципе, любой интерфейс можно построить в любом ViewGroup-контейнере, но тот или иной ViewGroup лучше и быстрее справляется с поставленной задачей, и, соответственно, более предпочтителен. Например:

- **LinearLayout** располагает свои элементы друг за другом в одном направлении - вертикально или горизонтально
- **RelativeLayout** представляет возможность располагать свои элементы относительно друг друга
- **FrameLayout** удобен для того, чтобы располагать элементы друг на друге, то есть верхний элемент будет частично загораживать нижний

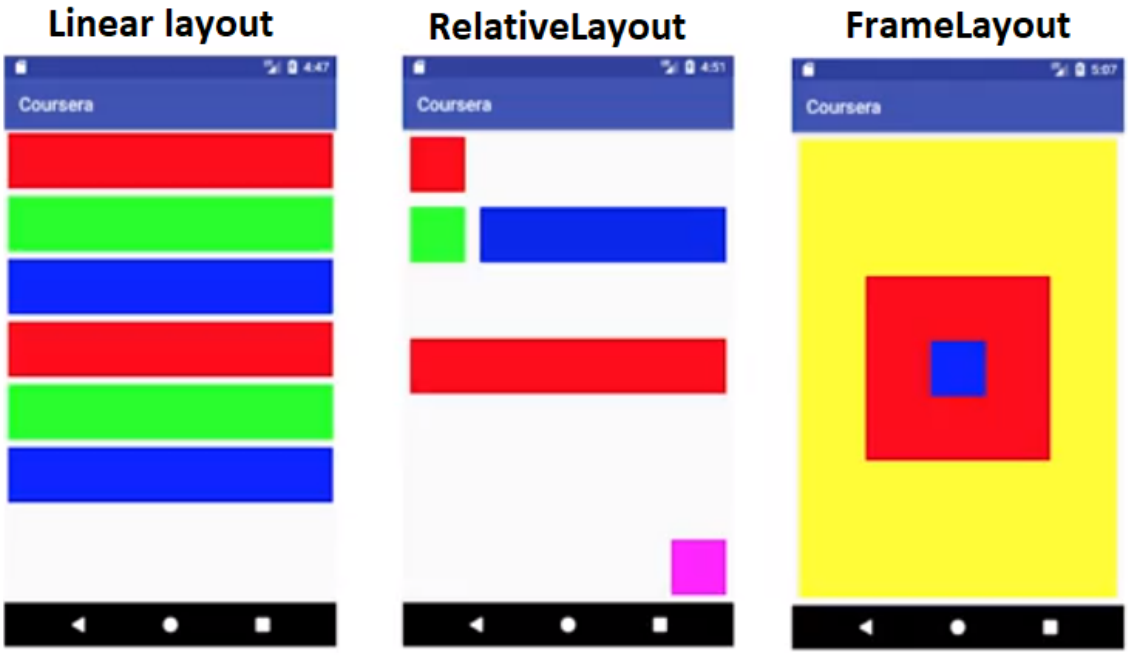


Рис. 2.3: Виды разных Layout-ов

Некоторые View встречаются повсеместно — текст в Android это TextView, поле ввода текста — EditText, кнопка — Button, картинка — ImageView, индикаторы прогресса — ProgressBar, CheckBox и RadioButton — соответственно выбор нескольких опций и одной. Все они входят в стандартный пакет Android SDK. Отдельно от вышеперечисленных элементов стоят наследники так называемого AdapterView, который используется для показа каких-либо данных в виде списка, по большей части однородного. Для заполнения View-элемента данными используется Adapter — конкретная реализация одноименного паттерна проектирования.

2.1.4 Реализация ViewGroup

ViewGroup используется как контейнеры для обычных View-элементов, и реализации ViewGroup отличаются друг от друга расположениями этих элементов.

- **LinearLayout** располагает свои элементы друг за другом в одном направлении - вертикально или горизонтально. Направление задаётся атрибутом orientation

  - распределяет свободное пространство по длине или ширине через веса: атрибут layout_weight - на дочерних View-элементах и weightSum - на LinearLayout контейнере. (Если $weightSum \neq \sum layout\_weight$ то на разметке будет неиспользованное пространство)
  - LinearLayout хорошо справляется с одной задачей — располагать элементы друг за другом. При более разветвленном дизайне интерфейса верстка через LinearLayout может стать проблемой из-за большого количества вложенных контейнеров.

- **RelativeLayout** позволяет более гибко располагать свои дочерние элементы с помощью большого списка атрибутов
  - layout_alignParentLeft(Right, Top, Bottom, Start, End) — выравнивает элемент по указанному краю родителя
  - layout_alignLeft(Right, Top, Bottom, Start, End) — выравнивает сторону элемента по соответствующей стороне указанного элемента
  - layout_below(above) — элемент под/над указанным элементом
  - layout_toLeftOf(Right, Start, End) — элемент с указанной стороны указанного элемента
  - layout_centerHorizontal(Vertical) — по центру контейнера и другие

- **FrameLayout** обычно используемый как контейнер для одной View, при добавление этой View происходит программно.
  - если есть необходимость расположить несколько View друг на друге, то FrameLayout — вполне подходящий вариант. Android рисует элементы в том порядке, в котором они описаны в Layout-файле.

- **ConstraintLayout** где для расположения дочерних элементов используют так называемый constraint'ы, или правила. Если верстку можно сделать с помощью другого layout'а, то лучше воспользоваться им, иначе есть вероятность погрязнуть в пучине constraint'ов, так и не добившись желаемого результата

- **GridLayout и TableLayout** выстраивают свои дочерние элементы в виде таблицы. Однако они не очень распространены из-за неудобства их использования. Тот же табличный вид можно получить с помощью других Layout

На скриншоте корневой LinearLayout с вертикальной ориентацией. В нем пара кнопок и вложенный горизонтальный Linear Layout, в котором, еще две кнопки. Веса этих кнопок указаны в числителях, в знаменателе значения weightSum горизонтального LinearLayout. Так как сумма числителя не равна знаменателю, LinearLayout оставляет пустое место, размером с остаток. Атрибут layout_width

в этом случае распространяется только на ширину кнопок, поэтому в целях производительности Studio подсказывает нам установит ширину кнопок 0dp.

Корневой LinearLayout (фиолетовый):
android:orientation="vertical"

Вложенный LinearLayout (малиновый):
android:orientation="horizontal"
android:weightSum="4"

Кнопки горизонтального LinearLayout:
android:layout_weight="1" / "2",
android:layout_width="0dp"

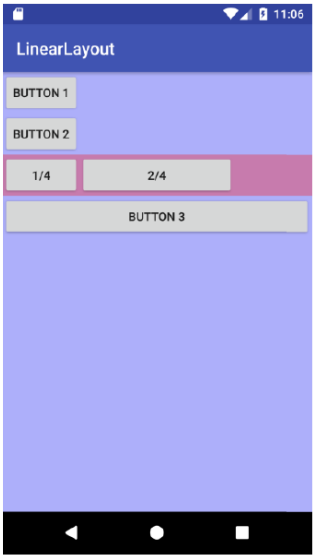


Рис. 2.4: Вложенные LinearLayout-ы

- Button 1 - выровнена по верхнему левому краю RelativeLayout
- Button 2 - находится под и справа от Button 1
- Button 3 - выровнена по центру RelativeLayout
- Button 4 - находится справа от Button 3 и нижняя граница соответствует нижней границе Button 3
- Button 5 - левый край соответствует левому краю Button 3, правый край соответствует правому краю Button 4, расположена под Button 3

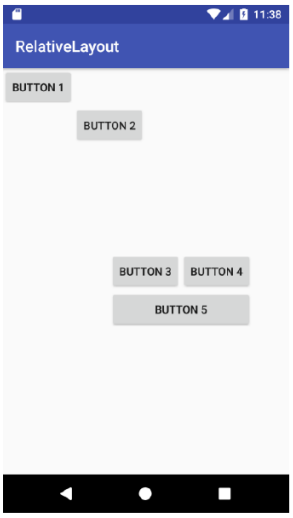


Рис. 2.5: пример RelativeLayout

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

  <View
    android:layout_width="200dp"
    android:layout_height="200dp"
    android:layout_gravity="center"
    android:background="@color/red" />

  <View
    android:layout_width="180dp"
    android:layout_height="50dp"
    android:layout_gravity="center"
    android:layout_marginTop="60dp"
    android:background="@color/yellow" />

</FrameLayout>
```

Рис. 2.6: пример FrameLayout

Чаще всего в разработке мы будем использовать Linear, Relative и FrameLayout. Остальные обязательно попробуйте сами. Кроме этого у View-элементов есть общие атрибуты. А здесь подробнее про TextView, Button, CheckBox, Radiobutton

2.1.5 ImageView, EditText

ImageView — это элемент интерфейса, который заточен показывать какие-либо изображения. Само изображение в xml можно задать через атрибут src и в качестве значения указать ссылку на файл в папке drawable. В xml:

```
android:scr="@drawable/my_image"
```

В Java:

```
image.setImageResource(R.drawable.my_image);
```

ImageView — это один из немногих элементов, размеры которого чаще всего задаются в конкретных значениях. Для маштабирования изображения используется атрибут ScaleType, имеющий возможные значения значения: fitXY, fitCenter, fitStart, fitEnd, center, centerCrop, centerInside, matrix:

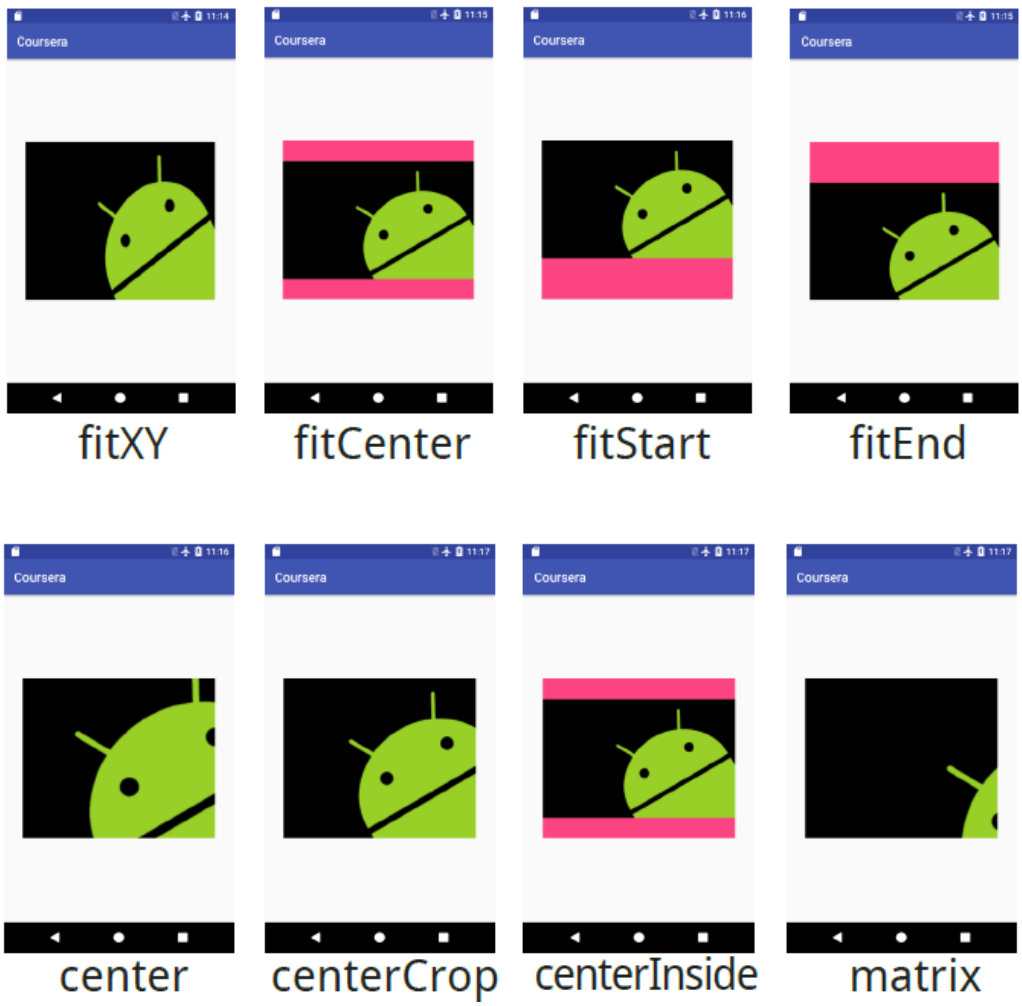


Рис. 2.7: Различные значения атрибута ScaleType

На картинке создано ImageView, с фиолетовым бэкграундом и изображением маскота Андроида, чтобы можно было легко понять, где заканчивается граница картинки и самой View.

1. fitXY — растягивается по высоте и ширине до размеров ImageView
2. FitCenter — (по умолчанию) изображение выравнивается по центру и масштабируется, пока не упрется в одну из сторон. Соотношение сторон не нарушается, изображение будет видно полностью.
3. FitStart — масштабирование работает как у FitCenter, но выравнивается у верхнего края View.
4. FitEnd — то же самое, только изображение выравнивается у нижнего края View.
5. Center — изображение выравнивается по центру, но не масштабируется, то есть если изображение не помещается, то оно обрежется, и наоборот, если View больше, чем изображение, то останется пустое пространство.

6. CenterCrop — изображение масштабируется до тех пор, пока не заполнит всю View, а не поместившаяся часть изображения обрежется.

7. enterInside — изображение масштабируется до тех пор, пока стороны не будут равны или меньше соответствующих сторон View, но маленькое изображение не будет увеличиваться, чтобы заполнить большую View.

8. matrix — изображение масштабируется согласно матрице, которая задается в коде через метод SetImageMatrix. Матрица используется для поворота, масштабирования, наклона и перемещения изображения, причем необязательно привязываться к ImageView.

Теперь про **EditText**. Это поле ввода текста, причем формат введенного текста и соответствующая ей клавиатура могут задаваться с помощью атрибута **InputType** (текст, номер, электронная почта, пароль, дата и другие).
Также есть возможность кастомизировать кнопку ввода на клавиатуре и показать вместо нее какую-нибудь другую через атрибут **imeOptions** (варианты actionDone, actionSearch, actionNext и другие).
Другой полезный атрибут **hint** используется для того, чтобы показать подсказку, что именно следует ввести в поле ввода.
Также для того чтобы задать максимальное количество символов в элементе, используется атрибут **maxLength**.

Всё о работе с **ресурсами** приложения
О **квалификаторах**

2.2 Инструменты сборки и отладки

2.2.1 Система сборки Gradle

Gradle — это система автоматической сборки, которая была разработана для расширяемых многопроектных сборок. Gradle поддерживает инкрементальные сборки, самостоятельно определяя, какие компоненты дерева сборки не изменились и какие задачи, зависимые от этих компонентов, не требуют перезапуска. Сам по себе Gradle ничего не знает о том, как собирать Android-проект. В этом ему помогает плагин, который разрабатывается и развивается вместе с Android SDK. В главном сборочном скрипте Build Gradle студия самостоятельно добавляет зависимость от этого плагина. Мы будем работать с недавно вышедшим Gradle 4.1.
Самая популярная возможность — это добавление сторонних зависимостей т.е. с помощью команды implementation во время сборки в наше приложение загрузятся все исходные коды зависимостей, которые мы указали в данном блоке.

```
dependencies {
  implementation fileTree(dir:'libs',include: ['*.jar'])
  androidTestImplementation('com.android.support.
      test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support',
        module: 'support-annotations'
  })
  implementation'com.android.support:appcompat-v7:26.1.0'
  implementation 'com.android.support.constraint:
      constraint-layout:1.0.2'
  testImplementation 'junit:junit:4.12'
  implementation 'org.jetbrains:annotations:15.0'
}
```

В версиях Gradle до 4-й вместо implementation использовалась команда compile. Gradle также позволяет собирать сборки, которые частично будут отличаться друг от друга. Делается это с помощью так называемых productFlavor и buildConfig. productFlavor — это Gradle-механизм, который полезен при создании нескольких версий одного приложения, разница в которых имеет значение для конечного пользователя, например платная или бесплатная версия приложения. Для этого достаточно добавить несколько строк кода в раздел Android файла build.gradle.

```
flavorDimensions 'buildType'
productFlavors {
  free {//бесплатная версия приложения
    buildConfigField 'String','YOUR_PARAM_NAME',
        '"YOUR STRING VALUE"'
    versionName "0.0.1-free"
    buildConfigField 'String', 'API_BASE_URL',
        '"https://yourapi.com/free/"'
  }
  pro {//платная версия приложения
    versionName "0.0.1-pro"
    buildConfigField 'String', 'API_BASE_URL',
        '"https://yourapi.com/pro/"'
  }
}
```

Альтернативно можно расписать различные buildTypes. Мы используем build-тайпы, чтобы собирать приложения с существенными отличиями. Эти отличия имеют большое значение для разработчиков. Например, сборка для отладки или сборка для релиза, сборка с обфускацией кода или без него, каким сертификатом будет подписано приложение и тому подобное. Соответственно, перед сборкой приложения мы выбираем конкретный build-тайп и productFlavor. В совокупности эти две сущности составляют Build Variant.

```
buildTypes {
  debug {
    buildConfigField 'String', 'API_BASE_URL',
        '"https://yourapi.com/debug/"'
  }
  beta {
    buildConfigField 'String', 'API_BASE_URL',
        '"https://yourapi.com/beta/"'
  }
  release {
    buildConfigField 'String', 'API_BASE_URL',
        '"https://yourapi.com/release/"'
  }
}
```

Gradle также позволяет добавлять статические поля в собираемый проект с помощью указания buildConfigField. Это поле может использоваться для любых целей и может меняться в зависимости от build-тайпа и productFlavor. В каждом из build-тайпов или productFlavor может содержаться одно и то же поле, но с разными значениями. Учтите, что в случае с одним и тем же полем и в build type, и в product failer значение будет браться из build type.

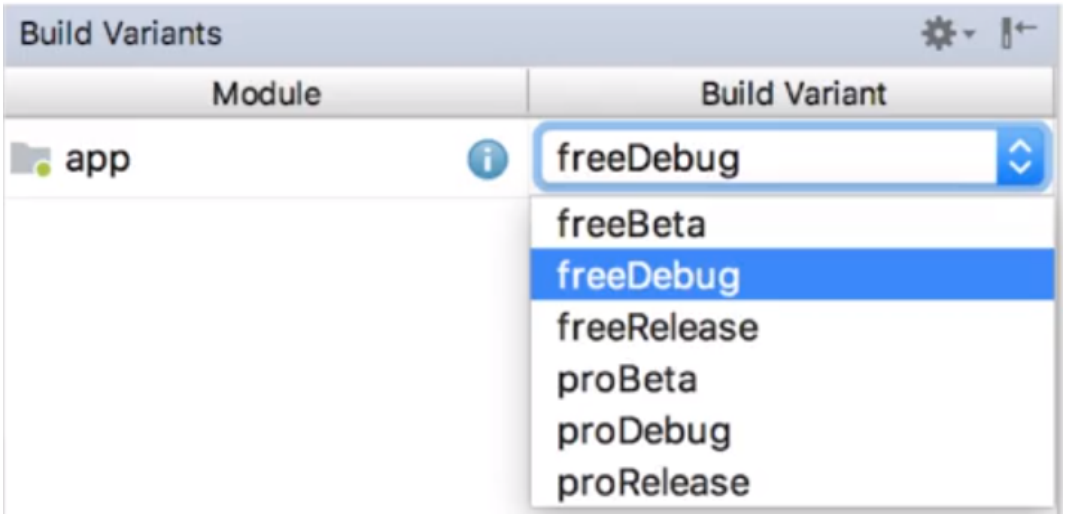


Рис. 2.8: Варианты сборки

Кстати, поле, записанное в defaultConfig будет присутствовать в каждом варианте сборки, но его можно переопределить в конкретном build-тайпе или product failer. Синтаксис buildConfigField достаточно прост: мы пишем buildConfigField и далее тип, имя поля и значение.

Листинг 2.1: BuildConfigField

```
defaultConfig {
  applicationId "com.e_legion.coursera"
  minSdkVersion 16
  targetSdkVersion 26
  versionCode 1
  versionName "1.0"
  testInstrumentationRunner
      "android.support.test.runner.AndroidJUnitRunner"
  buildConfigField 'String', 'YOUR_NAME',
          '"DEFAULT STRING"'
}
```

Во время сборки проекта Gradle сгенерирует класс buildConfig, в котором будут храниться наши buildConfig-поля. Чтобы использовать эти поля внутри приложения, нужно обращаться к ним как к обычным public static полям.

Листинг 2.2: Класс BuildConfig

```
public final class BuildConfig {
  public static final boolean DEBUG = false;
  public static final String APPLICATION_ID =
      "com.e_legion.coursera";
  public static final String BUILD_TYPE = "beta";
  public static final String FLAVOR = "free";
  public static final int VERSION_CODE = 1;
  public static final String VERSION_NAME = "0.0.1-free";
      // Fields from build type: beta
  public static final String API_BASE_URL =
  "https://yourapi.com/beta/";
      // Fields from product flavor: free
  public static final String YOUR_PARAM_NAME ="YOUR
    STRING VALUE";
      // Fields from default config.
  public static final String YOUR_NAME = "DEFAULT STRING
    VALUE";
}
```

Android Debug Bridge (ADB) - консольное приложение, которое позволяет с помощью терминала управлять подключенными устройствами (эмуляторами и реальными девайсами).
подробнее про ADB. Ещё немного про сам Debug

2.2.2 Toast

Toast (также тост) — это всплывающее сообщение. Легко реализуется. Может использоваться как в целях дебага, так и для уведомления пользователя.

```
//статический метод для создания тоста
Toast.makeText( //вызов статического метода для создания
    MainActivity.this, //1 параметр контекст
    "Connection lost", //2 параметр текст (или R.string.
      ресурс)
    Toast.LENGTH_LONG) //3 параметр - продолжительность
.show(); //вызов метода для показа тоста на экране
```

LENGTH_SHORT — 2 секунды, LENGTH_LONG — 3.5 секунды

Чтобы создать кастомный Toast, нужно:

1. Создать желаемую XML верстку
2. Объявить и инициализировать переменную класса Toast
3. Преобразовать верстку во View с помощью LayoutInflater

4. Задать желаемые значения для элементов View
5. Применить метод toast.setView(View v).

доп можно задать длительность и gravity.

Пример кастомного toast:

```
public void showCustomToast() {
  LayoutInflater inflater = getLayoutInflater();
  View layout = inflater.inflate(R.layout.custom_toast,
    null);
  TextView text = layout.findViewById(R.id.tvTitle);
  text.setText("This is a custom toast");
  ImageView image = layout.findViewById(R.id.image);
  image.setImageResource(R.mipmap.ic_launcher_round);
  Toast toast = new Toast(this); // контекст
  toast.setView(layout);
  toast.show();
}
```

Вот как это будет выглядеть с атрибутом gravity:

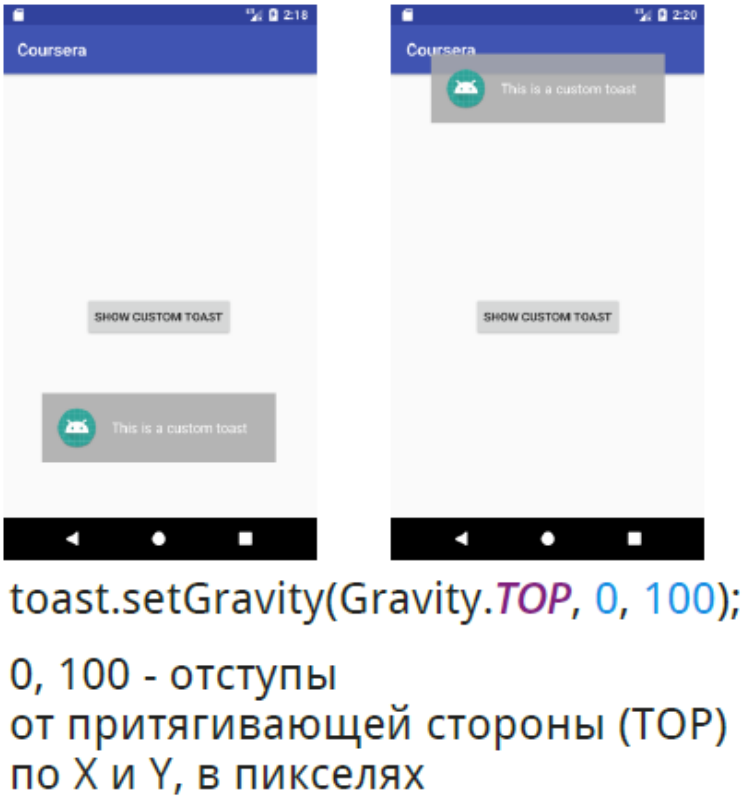


Рис. 2.9: Toast с изменённым атрибутом gravity

2.2.3 Menu, ContextMenu

Меню можно создать как в коде, так и через xml ресурс. Во втором способе правой кнопкой кликаем по res ⇒ New ⇒ Android resource file. В появившемся окне выбираем resource type - Menu и указать название. Создастся заготовка.

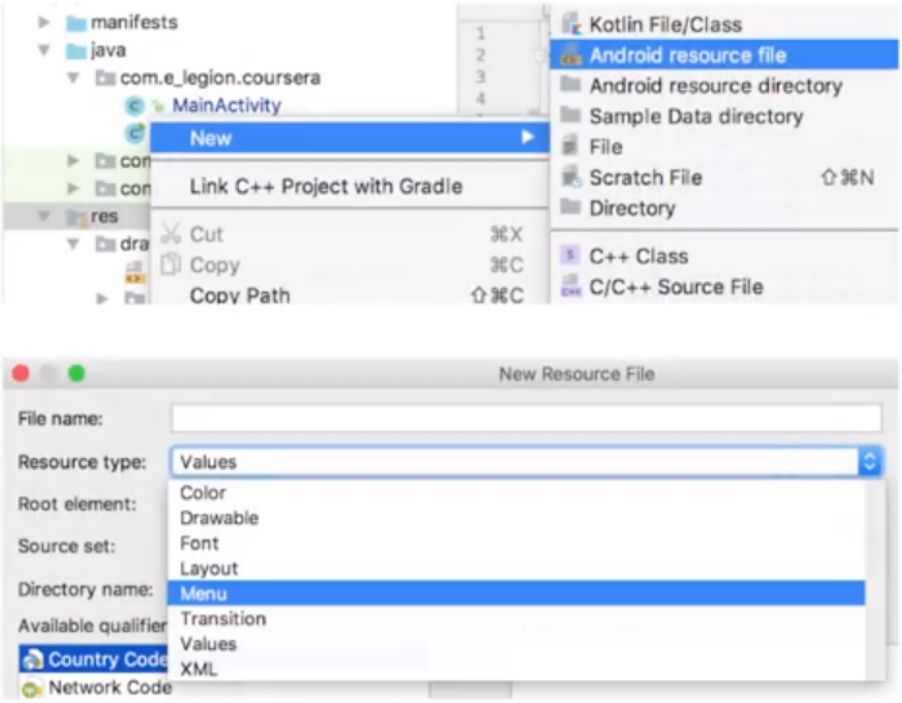


Рис. 2.10: Создание файла menu

Пример заполненного меню

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:android="http://schemas.android.com/apk/res/
    android">
  <item android:id="@+id/settings"
    android:title="Settings"
    android:icon="@drawable/ic_settings"
    app:showAsAction="never" />
  <item android:id="@+id/search"
    android:title="Search"
    android:icon="@drawable/ic_search"
    app:showAsAction="never" />
  <item android:id="@+id/logout"
    android:title="Logout"
app:showAsAction="never"/>
</menu>
```

id - идентификатор. title - название, видное пользователю, icon - иконка, видная пользователю. ShowAsAction - переместить в тулбар: never - никогда, always - всегда (не рекомендуется), ifRoom - если есть место, можно добавить |withText - вместе с title если есть место.
Добавим меню в activity:

```
@Override
public boolean onCreateOptionMenu(Menu menu) {
  getMenuInflater().inflate(R.menu.main_menu, menu);
  //наш объект меню мапится на созданный ресурс меню
  return true;
}
```

Возвращаемое значение булевского типа - будет ли меню показано, или нет. Т.е. можно добавить в метод условие и решать, будет меню показано или нет. Аргумент - объект типа Menu - наше меню, которое будет показываться. Теперь обработаем нажатия на пункты (логика работы с меню):

```
@Override
public boolean onOptionItemSelected (MenuItem item){
  switch (item.getItemId()) {
    case R.id.settings:
      Toast.makeText(this, "Settings clicked",
            Toast.LENGTH_SHORT).show();
      return true;//показываем, что мы сами вызвали метод
    case R.id.search:
      Toast.makeText(this, "Search clicked",
            Toast.LENGTH_SHORT).show();
      return true;
    default: return super.onOptionItemSelected(item);
  }
}
```

Если же предполагается менять меню в рантайме, то используем другой метод:

```
@Override
public boolean onPrepareOptionsMenu (Menu menu) {
  if (!isUserAuthorized()) {
    menu.removeItem(R.id.logout);
    return true;
  }
  return super.onPrepareOptionsMenu(menu);
}
```

Здесь у залогиненого пользователя будет возможность разлогиниться.
Теперь рассмотрим **контекстное меню** - меню, вызываемое при долгом нажатие на элементе экрана, и наполненное специфичными для данного элемента

пунктами. Элемент и пункты контекстного меню мы определяем сами. Допустим, что у нас есть TextView, и мы хотим повесить на него контекстное меню. Рассмотрим его создание:

```
mHelloTv = findViewById(R.id.tv_hello_world);
registerForContextMenu(mHelloTv);
//включение контекстного меню (в onResume())
unregisterForContextMenu(mHelloTv);
//отключение контекстного меню (в onPause())
public static final int GROUP_ID = Menu.NONE;
public static final int MENU_ITEM_ID = 42;
public static final int ORDER = Menu.NONE;
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
ContextMenu.ContextMenuInfo menuInfo) {
  if (v.getId() == R.id.tv_hello_world) {
   // динамическое заполнение методом add(), но можно как и в
       обычном
    menu.add(GROUP_ID, MENU_ITEM_ID, ORDER,"Context menu"
      );
  } else {
    super.onCreateContextMenu(menu, v, menuInfo);
  }
}
```

ContextMenu.ContextMenuInfo - предоставляет какую-нибудь информацию об объекте. Для обработки нажатия используется метод:

```
@Override
public boolean onContextItemSelected(MenuItem item) {
  switch (item.getItemId()) {
    case MENU_ITEM_ID:
      Toast.makeText(this, "Context menu clicked",
              Toast.LENGTH_SHORT).show();
    return true;
    default: return super.onContextItemSelected(item);
  }
}
```

Кроме разобранного ещё есть: **ActionView** — это View, которое появляется в тулбаре при нажатии на пункт меню. И распространенный пример такого View — это строка поиска. **ActionMode** — режим для работы с элементами.

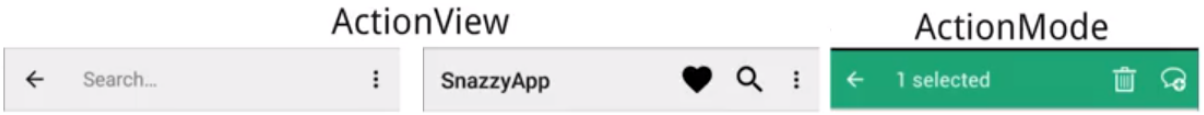


Рис. 2.11: Как выглядят ActionView и ActionMode

Например, долгое нажатие на фотографии в галерее приведет к активации ActionMode. В тулбаре интерфейс изменится. Вы сможете выбрать несколько фотографий и, например, удалить их. Либо нажать кнопку «Назад» и выйти из ActionMode. Кроме этого полезно будет разобраться с материалами:
Подробнее о Intents (неявные), IntentFilters
Backstack Activity, launchMode, intentFlags, taskAffinity

2.3 Фрагменты и файлы Preferences

2.3.1 Знакомство с Fragment

Fragment — это модульная часть activity, у которой свой жизненный цикл и свои обработчики различных событий.

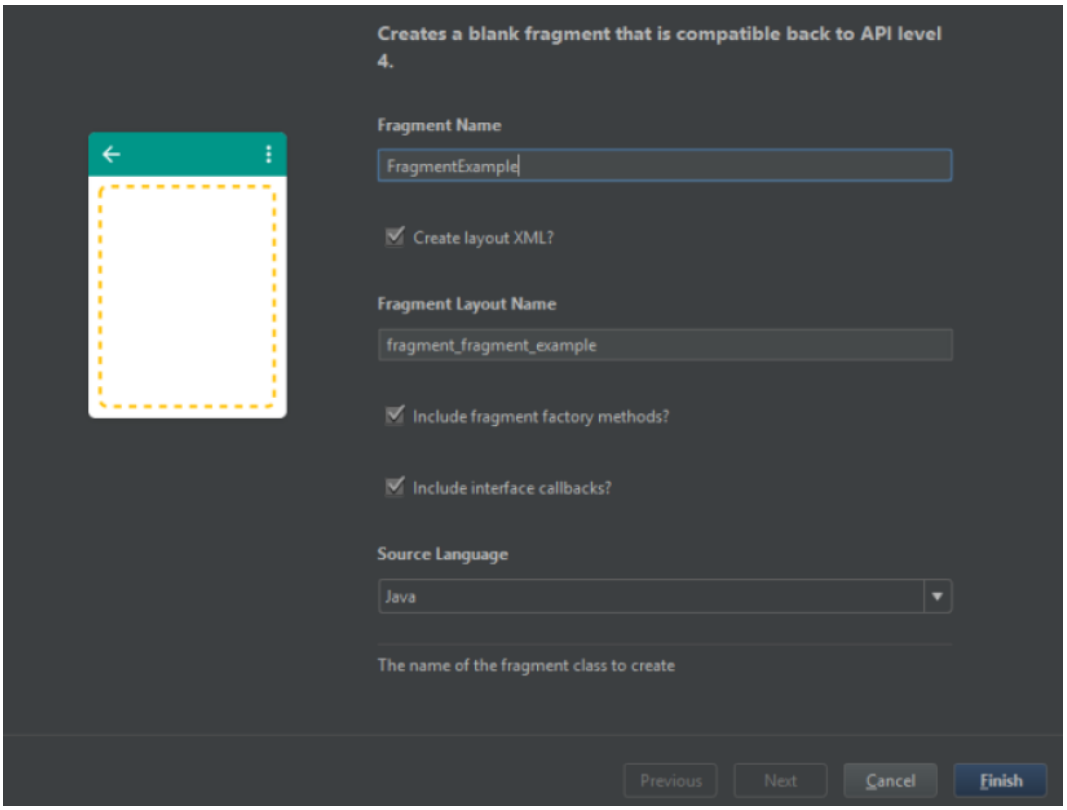


Рис. 2.12: Создание фрагмента

По сути, это подобие Activity — кусок интерфейса с логикой, но Fragment содержится внутри Activity. И, соответственно, одна Activity может отображать

одновременно несколько фрагментов. Фрагменты появились в Android с API11 (Android 3.0), для разработки более гибких пользовательских интерфейсов на больших экранах, таких как экраны планшетов. Но впоследствии они нашли применение не только для этого. Через некоторое время была написана библиотека Android Support Library, которая добавляла поддержку фрагментов и в более ранней версии Android. Существует два основных подхода в использовании фрагментов:

1. Первый основан на замещение родительского контейнера. Создается разметка, и в том месте, где нужно отобразить фрагменты, размещается контейнер, чаще всего это FrameLayout. В коде в контейнер помещается фрагмент.

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.
  com
              /apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/fr_container"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:fitsSystemWindows="true"/>
```

2. фрагмент добавляется в разметку через тег fragment, но учтите, что его потом нельзя будет поменять. То есть указанные фрагменты будут использоваться до конца работы Activity.

```
<fragment
  android:id="@+id/fragment_example"
  android:name="com.e_legion.coursera.ExampleFragment
    "
  android:layout_width="match_parent"
  android:layout_height="match_parent"/>
```

При работе с фрагментами мы используем три основных класса:

- **Fragment**, от которого наследуются все фрагменты
- **FragmentManager** — с помощью экземпляра этого класса происходит все взаимодействие между фрагментами и Activity
- **FragmentTransaction** — сама транзакция.

Также существуют различные подклассы фрагментов: DialogFragment, WebViewFragment, MapFragment и тому подобное.
Рассмотрим пример реализации фрагмента:

```
public class ExampleFragment extends Fragment {
  private TextView mExampleText;
  public static ExampleFragment newInstance() {
    return new ExampleFragment();
  }
  @Override
  public View onCreateView(LayoutInflater inflater,
  ViewGroup container, Bundle avedInstanceState) {
    View view = inflater.inflate
        (R.layout.fr_example,container, false);
    mExampleText = view.findViewById(R.id.tv_example);
    mExampleText.setText("Example text");
    return view;
  }
}
```

LayoutInflater — это класс, который позволяет построить нужный макет, получая информацию из указанного XML-файла Layout. Обратите внимание на то, что метод inflate() получает три параметра. Во-первых, это разметка нашего фрагмента, второе — контейнер, и булево значение **attachToRoot**. Если установлено значение true, то после вызова функции inflate() полученный View будет добавлено в Layout-контейнер. Но при работе с фрагментами здесь всегда будет false, так как фрагмент подключается к разметке через FragmentManager (добавить фрагмент, заменить фрагмент, удалить фрагмент). Его методы:

- FragmentTransaction transaction = getFragmentManager().beginTransaction();
- transaction.add(R.id.fr_container, fragment);
- transaction.replace(R.id.fr_container, fragment);
- transaction.remove(fragment);
- transaction.hide(fragment);
- transaction.show(fragment);
- transaction.detach(fragment);
- transaction.attach(fragment);
- transaction.commit();

Также обратите внимание на метод **newInstance()** — этот метод мы будем использовать для создания экземпляра нашего фрагмента. Конструкторы фрагментов не рекомендуется переопределять, так как фрагмент может быть уничтожен и восстановлен системой, и если не будет подходящего конструктора,

то приложение просто крашнется. В отличие от Activity, фрагмент не нужно добавлять в AndroidManifest.
Для того чтобы добавлять наши фрагменты в стек так, как это работает по умолчанию с Activity, можно вызвать метод **addToBackStack()** у FragmentManager. Чтобы перейти на предыдущий экран с фрагментом, нужно вызвать метод popBackStack(), но для правильной работы нужно, чтобы при вызове popBackStack() в стеке был хотя бы один фрагмент, иначе мы получим просто пустоту в контейнере фрагмента. По идее нужно проверять этот момент и закрывать Activity, если в стеке остался последний фрагмент.

```
transaction.addToBackStack(fragment.getClass().
        getSimpleName());
getFragmentManager().popBackStack();

@Override
public void onBackPressed() {
  if (getFragmentManager().getBackStack
          EntryCount() > 0) {
    getFragmentManager().popBackStack();
  } else {
    super.onBackPressed();
  }
}
```

С помощью метода getActivity() во фрагменте можно получить ссылку на HostActivity. Кстати, если наша HostActivity пересоздавалась, то FragmentManager сам должен восстановить все фрагменты. Проще всего уточнить этот момент можно спомощью проверки savedInstanceState на null.

```
getActivity();
if(savedInstanceState == null) {
  //то добавляем фрагмент
  ExampleFragment fragment = ExampleFragment.newInstance
    ();
  FragmentManager fragmentManager = getFragmentManager();
  fragmentManager.beginTransaction()
          .add(R.id.fr_container, fragment)
          .addToBackStack(ExampleFragment.
      class.getSimpleName()).commit();
}
```

Жизненный цикл фрагмента отличается отличается от жизненного цикла Activity, но состояние и Activity, и находящихся в ней фрагментов всегда коррелирует.
Жизненный цикл фрагмента(ф.):

1. **onAttach()** — ф. присоединяется к Activity. Пока Fragment и Activity еще не полностью инициализированы.
2. **onCreate()** — в этом методе можно инициализировать переменную (так же, как в Activity).
3. **onCreateView()** — создает интерфейс ф-а метод
4. **onActivityView()** вызывается когда отработает метод Activity onCreate(начиная с этого момента, во ф-е можно обращаться к компонентам Activity).
5. **onStart()** — вызывается, когда ф. видим
6. **onResume()** — ф. готов к взаимодействию
7. **onPause()** — ф. остается видимым, но с ним нельзя взаимодействовать.
8. **onStop()** — ф. невидим
9. **onDestroyView()**—уничтожает интерфейс ф.
10. **onDestroy()** — полное уничтожение ф-а
11. **onDetach()** — отсоединение ф-а от Activity.

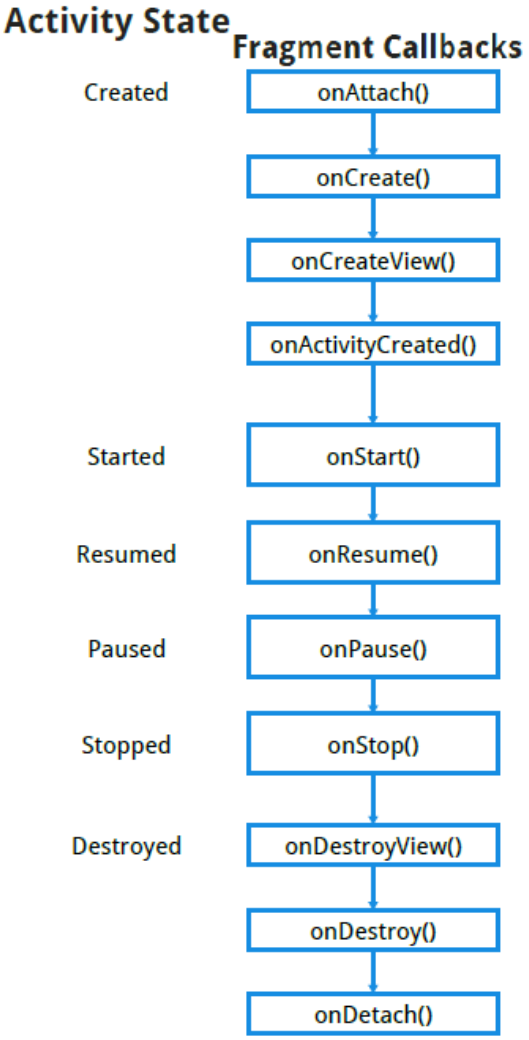


Рис. 2.13: Жизненный цикл фрагмента

Дополнительно о фрагментах

Для хранения пар ключ-значение в Android можно использовать SharedPreferences

2.3.2 Формат JSON. Библиотека GSON

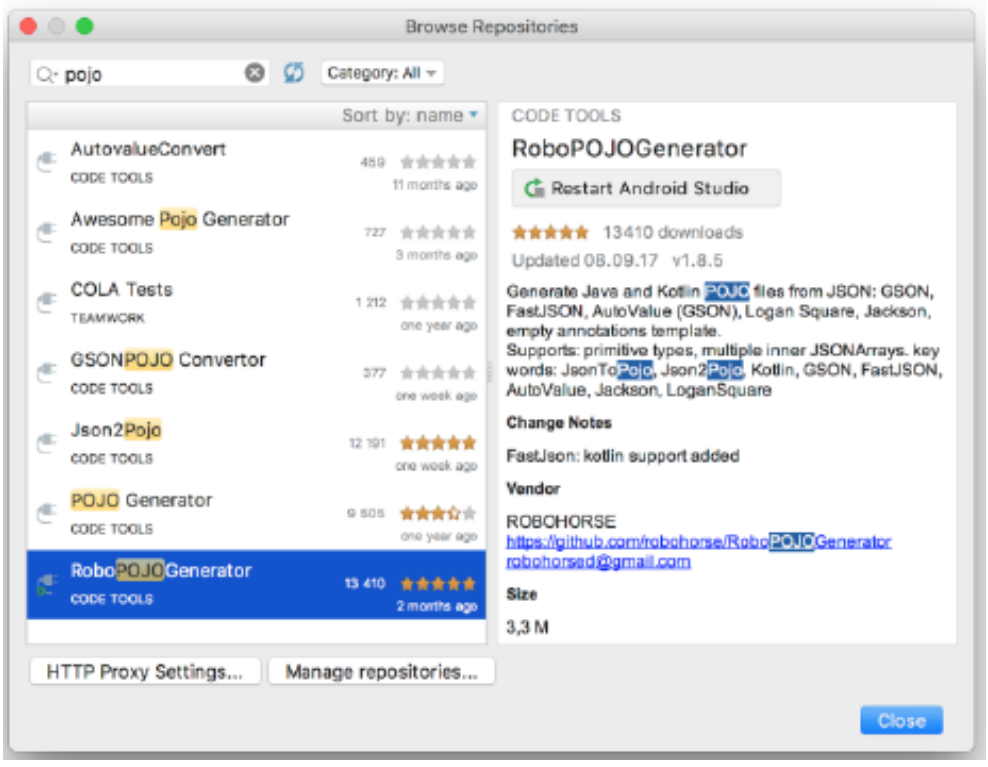
JSON — это текстовый формат данных, легко читаемый человеком и используемый для сериализации объектов и обмена данными.
Сериализация это сохранение состояния объекта для передачи и последующей десериализации — восстановления.

В любом правильно сформированном json-файле можно выделить два основных вида структур: это поля типа ключ-значение и упорядоченный набор значений. Ключ всегда является строковым типом, а значения могут принимать вид строки, числа, литералов true, false, null, массивов либо объектов, вложенных в json. Объекты обрамляются фигурными скобками, внутри которых хранятся неупорядоченное множество пар ключ-значение. Ключ от значения отделяется двоеточием. Между парами ставится запятая. Массивы в json заключаются в квадратные скобки, между значениями также ставится запятая.

```
{
  "name":"Ivanov Ivan",
  "age":42,
  "kids"\ :[
    {
      "name":"Ivanov Artiom",
      "age":8,
      "kids":null
    },
    {
      "name":"Ivanova Irina",
      "age":4,
      "kids":null
    }
  ]
}
```

Вот пример - отображение объекта, в котором видна модель человека. Поля: имя, возраст и детей. Дети — это массив объекта такого же типа «Человек», поэтому у них есть имя, возраст и поле "kids со значением null. Теперь попробуем собрать из этого сериализованного файла объект.

Прежде всего нам нужен класс «модель» или, как его еще называют, "**pojo**"— plain old java object, старый добрый java-объект. pojo-классы можно сгенерировать либо с помощью плагина для Studio, либо на специальных сайтах.



На основе json был создан pojo-класс. Теперь спокойно можно создать экземпляр этого класса и задать ему значение из json.

```
public class Human {
  @SerializedName("name")
  @Expose
  private String name;
  @SerializedName("age")
  @Expose
  private int age;
  @SerializedName("kids")
  @Expose
  private List<Human> kids = null;
  public String getName() { return name; }
  public void setName(String name) { this.name = name; }
  public int getAge() { return age; }
  public void setAge(int age) { this.age = age; }
  public List<Human> getKids() { return kids; }
  public void setKids(List<Human> kids){this.kids=kids;}
}
```

Это скучный процесс и были придуманы библиотеки, автоматизирующие конвертацию из json в объект и наоборот. Одна из таких библиотек — GSON. Подключим ее в файле gradle уровня app и рассмотрим простую сериализацию и десериализацию. **Сериализация** - это создание json из объекта. Сначала нам нужно создать такой объект. Создаём объект класс Human

```
Human father = new Human();
father.setAge(38);
father.setName("Petr Petrov");
Human kid = new Human();
kid.setName("Vasya Petrov");
kid.setAge(4);
kid.setKids(null);
List<Human> kids = new ArrayList<>();
kids.add(kid);
father.setKids(kids);
//Сериализация:
String json = new Gson().toJson(human);
System.out.println(json);
{"name":"Petr Petrov", "age":42,
"kids":[{"name": "Vasya Petrov", "age":8}]}
```

Теперь нам нужно у объекта класса Gson вызвать метод toJson(), который вернет нам json-строку.

```
//генерация метода toString()
@Override
public String toString()
    {
   return "Human{" +
     "name='" + name + '
       \''
     +", age=" + age +
     ", kids=" + kids +
     '}';
}
```

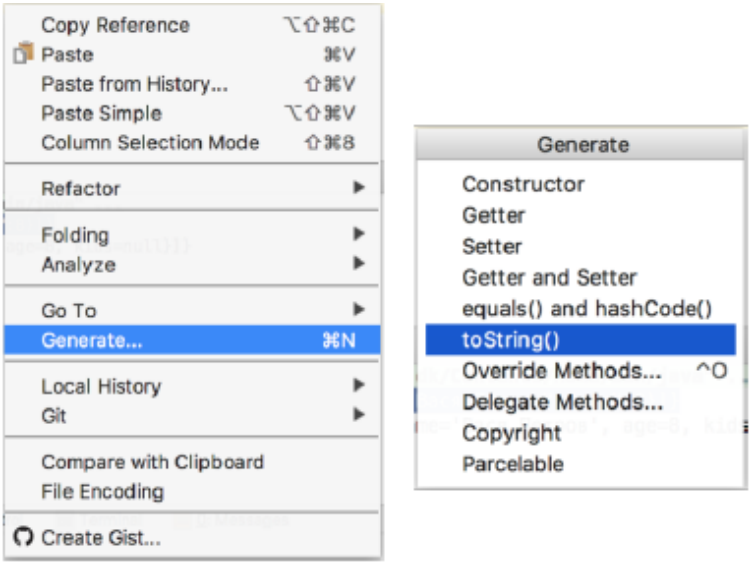


Рис. 2.14: toString()

Для десериализации у Gson нужно вызвать метод fromJson(), который на вход принимает json-строку и название класса, объект которого нужно создать. Попробуем воссоздать объект, который мы только что сериализовали. И выведем его в лог. Для этого мы переопределяем метод toString() у класса Human. Проще всего это можно сделать с помощью встроенной кодогенерации в Studio. Правой кнопкой кликаем по коду класса, Generate, toSting И теперь давайте сравним json и вывод toString.

```
Human fromJson = new Gson().fromJson(json,Human.class);
System.out.println(fromJson.toString());
toString() - Human{name='Petr Petrov',
     age=42, kids=[Human{name='Vasya Petrov',
           age=8, kids=null}]}
json - {"name":"Petr Petrov","age":42,"kids":
     [{"name":"Vasya Petrov","age":8}]}
```

Сериализовать и десериализовать объекты в json с помощью библиотеки Gson это основа для клиент-серверного взаимодействия. Помимо этого сериализованные в виде строки объекты можно хранить в файле preferences либо просто на устройстве.
Свой код можно публиковать на GitHub. про заливку кода на GitHub