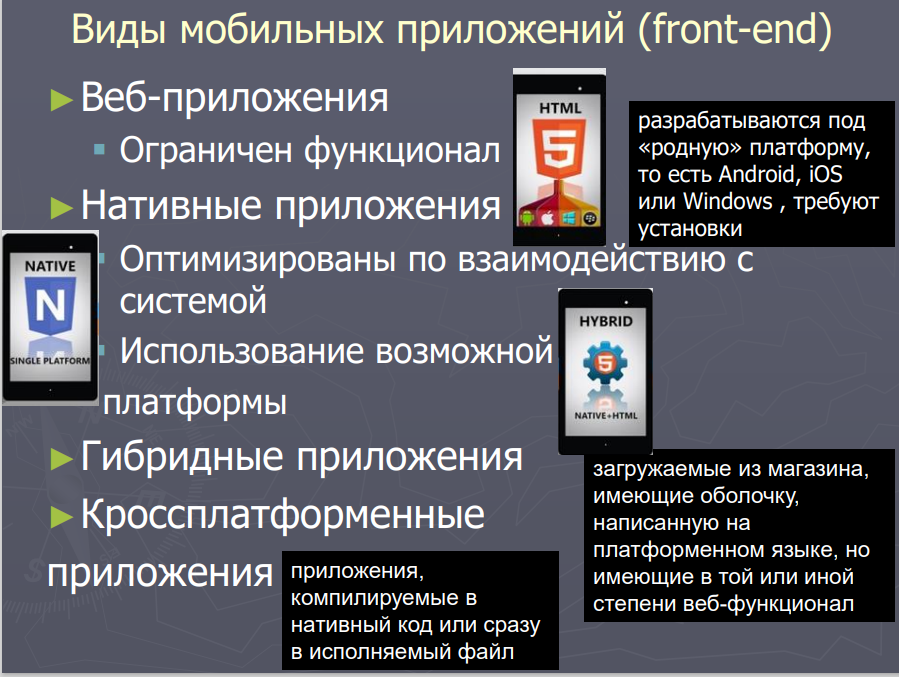
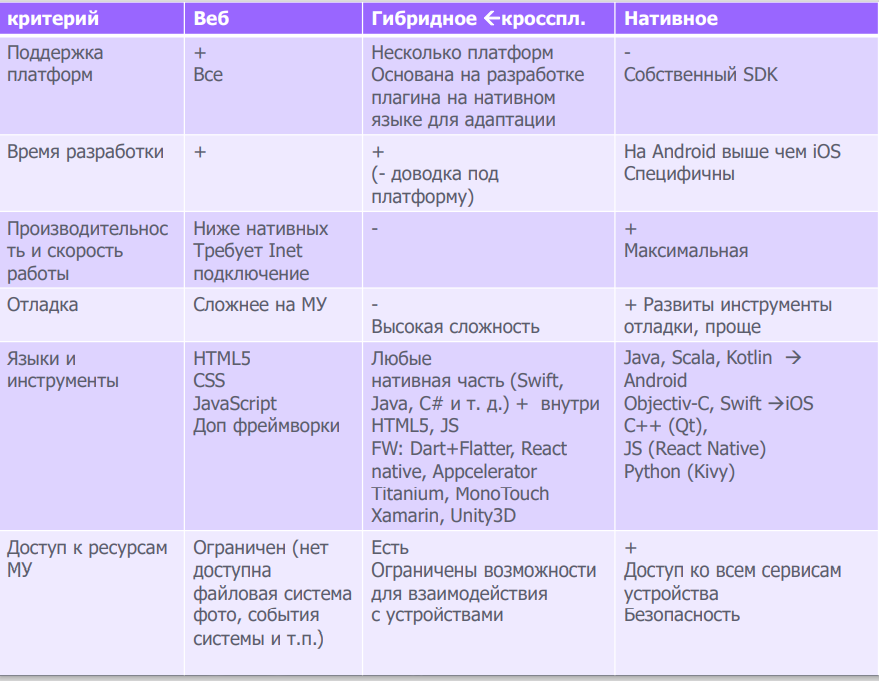
1. **Виды мобильных приложений (веб, нативное, гибридные и кроссплатформенные)**





1. **Архитектура ОС Android. Процесс сборки Android-приложения**

**Ядро Linux**

Основой платформы Android является ядро Linux. Например, Android Runtime (ART) полагается на ядро Linux для базовых функций, таких как потоковая обработка и управление уровнем низкого уровня памяти. Использование ядра Linux позволяет Android использовать ключевые функции безопасности и позволяет производителям устройств разрабатывать аппаратные драйверы для хорошо известного ядра.

**Уровень абстракции оборудования (HAL)**

Уровень абстракции аппаратного обеспечения (HAL) предоставляет стандартные интерфейсы, которые предоставляют аппаратные возможности устройства для платформы Java API более высокого уровня. HAL состоит из нескольких библиотечных модулей, каждый из которых реализует интерфейс для определенного типа аппаратного компонента, такого как камера или модуль bluetooth. Когда API-интерфейс инфраструктуры выполняет вызов для доступа к аппаратным средствам устройства, система Android загружает библиотечный модуль для этого аппаратного компонента.

**Android Runtime**

* **Библиотеки и исполняемая среда: каждое приложение запускается в собственном процессе с собственным экземпляром исполняемой среды Android. Android включает в себя набор основных библиотек, которые предоставляют большую часть функциональности языка программирования Java. Многие системные компоненты и службы Android созданы из нативного кода, для которого требуются библиотеки, написанные на C и C++. Эти библиотеки доступны для приложений через Java API.**

Для устройств под управлением Android версии 5.0 (API уровня 21) или выше каждое приложение запускается в собственном процессе и с собственным экземпляром Android Runtime (ART). ART записывается для запуска нескольких виртуальных машин на устройствах с низкой памятью путем выполнения DEX-файлов, формат байт-кода, разработанный специально для Android, который оптимизирован для минимального объема памяти. До версии Android 5.0 (уровень API 21) Dalvik был временем выполнения Android. Если ваше приложение хорошо работает с АRТ, то оно должно работать и на Дальвике, но обратное может быть неверным. Android также включает в себя набор основных библиотек времени выполнения, которые обеспечивают большую часть функциональных возможностей языка программирования Java, включая некоторые языковые функции Java 8, которые использует инфраструктура Java API.

**Нативные библиотеки C / C ++**

Многие базовые компоненты и сервисы Android, такие как ART и HAL, построены из собственного кода, для которого требуются собственные библиотеки, написанные на C и C ++. Платформа Android предоставляет API-интерфейсы Java framework, чтобы выявить функциональность некоторых из этих родных библиотек для приложений. Например, вы можете получить доступ к OpenGL ES через Java OpenGL API платформы Android, чтобы добавить поддержку для рисования и управления 2D и 3D-графикой в вашем приложении. Если вы разрабатываете приложение, для которого требуется код C или C ++, вы можете использовать Android NDK для доступа к некоторым из этих родных библиотек плат непосредственно из вашего собственного кода.

**Java API Framework**

Java API: всё необходимое для Android-разработки: компоненты пользовательского интерфейса, управление ресурсами и управление жизненным циклом — всё это доступно через программные интерфейсы (API).

Весь набор функций Android OS доступен вам через API, написанные на языке Java. Эти API-интерфейсы образуют строительные блоки, необходимые для создания приложений для Android, упрощая повторное использование основных, модульных системных компонентов и сервисов, которые включают в себя следующее:

* расширяемая система просмотра, которую вы можете использовать для создания пользовательского интерфейса приложения, включая списки, сетки, текстовые поля, кнопки и даже встраиваемый веб-браузер
* Диспетчер ресурсов, обеспечивающий доступ к ресурсам без кода, таким как локализованные строки, графики и файлы макета
* Диспетчер уведомлений, который позволяет всем приложениям отображать пользовательские предупреждения в строке состояния
* Диспетчер активности, который управляет жизненным циклом приложений и предоставляет общий стек возврата навигации
* Поставщики контента, которые позволяют приложениям получать доступ к данным из других приложений, таких как приложение «Контакты», или предоставлять свои собственные данные .

**Процесс сборки Android-приложения**

Первым делом читается AndroidManifest.xml, в нём есть важные параметры (package и targetSdkVersion).

Затем вызывается программа aapt (Android Asset Packaging Tool), которой передаётся AndroidManifest.xml, папка с ресурсами res/, assets/, путь к android.jar нужной target-версии. aapt проверяет ресурсы и компилирует их, создавая при этом класс R.java в котором содержатся идентификаторы ресурсов и файл resources.arsc в котором содержится информация об xml-ресурсах и их атрибутах.

Далее подхватываются все библиотеки, которые используются в проекте и запускается Java-компилятор javac. Полученные class-файлы передаются в программу dx, которая переводит их в dex-формат. Причём для оптимизации, готовые библиотеки дексируются отдельно, а классы проекта отдельно (оптимизация в том, что дексированные библиотеки можно закешировать). Если собралось несколько dex-файлов, то они все объединяются при помощи Dex Merger Tool. В конечном итоге мы получаем единственный файл classes.dex (или несколько, если используется multidex).

Теперь у нас есть все компоненты и можно собирать apk. По сути это просто упаковка всех файлов в zip-архив, но используется специальная программа apkbuilder. После её выполнения получаем неподписанный apk-файл, то есть без папки META-INF внутри.

Последний этап - подпись apk. Берётся заранее сгенерированный ключ и передаётся в jarsigner вместе с неподписанным apk-файлом. На выходе имеем готовое приложение, которое можно устанавливать.

1. **Структура и компоненты приложения**

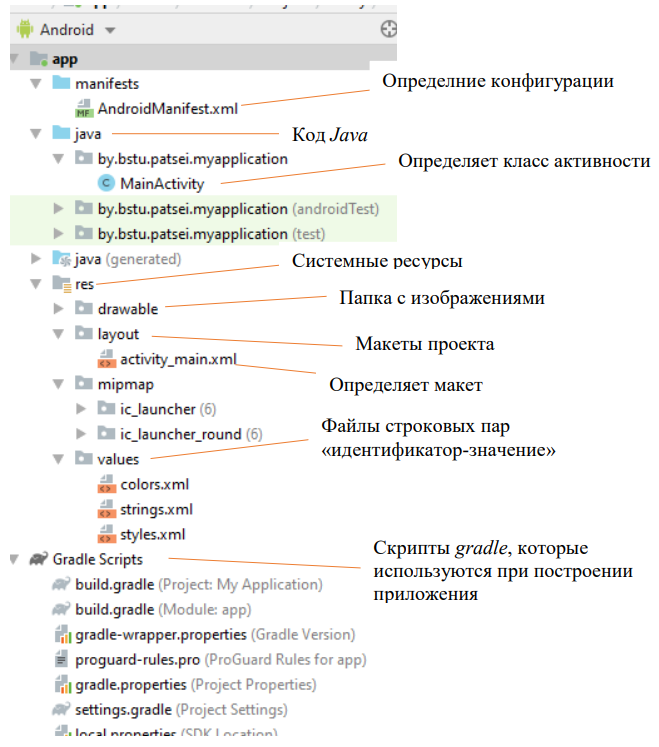
**Операция (Activity, активность)** представляет собой один экран с пользовательским интерфейсом. Например, в приложении для работы с электронной почтой одна активность может служить для отображения списка новых сообщений, другая – для составления сообщения и третья операция – для чтения сообщений. Несмотря на то, что активности совместно формируют взаимодействие пользователя с приложением, каждая из них не зависит от других. Любые из активностей могут быть запущены другим приложением. Например, приложение для камеры может запустить операцию в приложении по работе с электронной почтой, которая составляет новое сообщение, чтобы пользователь мог отослать фотографию. Операция относится к подклассу Activity.

**Служба (Service)** представляет собой компонент, который работает в фоновом режиме и выполняет длительные операции, связанные с работой удаленных процессов. Служба не имеет пользовательского интерфейса. Например, она может воспроизводить музыку в фоновом режиме, пока пользователь работает в другом приложении, или может получать данные по сети, не блокируя взаимодействие пользователя с активностью. Служба может быть запущена другим компонентом, который затем будет взаимодействовать с ней. Служба относится к подклассу Service.

**Поставщик контента (Content provider)** управляет общим набором данных приложения. Данные можно хранить в файловой системе, базе данных SQLite, в Интернете или любом другом месте хранения, к которому у приложения есть доступ. Посредством поставщика контента другие приложения могут запрашивать или изменять данные (если поставщик контента позволяет делать это). Например, в системе Android есть поставщик контента, который управляет информацией контактов пользователя. Любое приложение, получившее соответствующие разрешения, может запросить часть этого поставщика контента для чтения и записи. Поставщики контента также используются для чтения и записи данных, доступ к которым внешним компонентам приложение не предоставляет. Он относится к подклассу ContentProvider и должен реализовывать стандартный набор API-интерфейсов [2].

**Приемник широковещательных сообщений (Broadcast receiver)** представляет собой компонент, который реагирует на объявления, распространяемые по всей системе. Многие из этих объявлений рассылает система – например уведомление о том, что экран выключился, аккумулятор разряжен или был сделан фотоснимок. Объявления также могут рассылаться приложениями. Например, можно сообщить другим приложениям о том, что какие-то данные были загружены на устройство и теперь готовы для использования. Несмотря на то что приемники широковещательных сообщений не имеют пользовательского интерфейса, они могут создавать уведомления в строке состояния, чтобы предупредить пользователя о событии «рассылка объявления» [2]. Чаще всего они являются просто шлюзом для других компонентов и предназначены для выполнения минимального объема работы. Приемник широковещательных сообщений относится к подклассу BroadcastReceiver.

Уникальной особенностью системы Android является то, что любое приложение может запустить компонент другого приложения. Для пользователя это будет выглядеть как одно приложение. Когда система запускает компонент, она запускает процесс для этого приложения (если он еще не был запущен) и создает экземпляры классов, которые требуются этому компоненту. Поэтому, в отличие от приложений для большинства других систем, в приложениях для Android отсутствует единая точка входа (в них нет функции main()). Поскольку система выполняет каждое приложение в отдельном процессе с такими правами доступа к файлам, которые ограничивают доступ в другие приложения, приложение не может напрямую вызвать компонент из другого приложения. Но это может сделать система Android. Поэтому, чтобы вызвать компонент в другом приложении, необходимо сообщить системе о своем намерении (Intent) запустить определенный компонент. После этого система активирует для вас этот компонент



1. **Организация пользовательского интерфейса. Виды макетов пользовательского интерфейса**

Весь ui задается в xml-файлах макетов. Ui можно редактировать в 2 режимах: design – перетягивать нужные элементы и настраивать их параметры через ui android studio. Весь ui виден на экране; code – создание нужных элементов и их настройка через xml-конфигурацию.

Основным строительным блоком для пользовательского интерфейса является объект View, который создается из класса View и занимает прямоугольную область на экране и отвечает за рисование и обработку событий. Представление — это базовый класс для виджетов, которые используются для создания интерактивных компонентов пользовательского интерфейса, таких как кнопки, текстовые поля и т. Д.

ViewGroup является подклассом View и предоставляет невидимый контейнер, который содержит другие виды или другие ViewGroups и определяет их свойства макета.

На третьем уровне у нас есть различные макеты, которые являются подклассами класса ViewGroup, и типичный макет определяет визуальную структуру пользовательского интерфейса Android и может быть создан либо во время выполнения с использованием объектов View / ViewGroup, либо вы можете объявить свой макет с помощью простого XML-файла main\_layout .xml, который находится в папке res / layout вашего проекта.

Существует ряд макетов, предоставляемых Android, которые вы будете использовать практически во всех приложениях Android для обеспечения различного вида, внешнего вида и ощущений.

ConstrainLayout - Для позиционирования элемента внутри ConstraintLayout необходимо указать ограничения (constraints).

LinearLayout — это группа представлений, которая выравнивает все дочерние элементы в одном направлении, по вертикали или по горизонтали.

RelativeLayout — это группа представлений, которая отображает дочерние представления в относительных позициях.

TableLayout — это представление, которое группирует представления в строки и столбцы.

AbsoluteLayout позволяет указать точное местоположение его дочерних элементов.

FrameLayout — это заполнитель на экране, который можно использовать для отображения одного представления.

ListView — это группа представлений, которая отображает список прокручиваемых элементов.

GridView — это ViewGroup, которая отображает элементы в двумерной сетке с возможностью прокрутки.

1. **AndroidMainfest. Gradle files.**

AndroidManifest.xml представляет файл манифеста, который описывает фундаментальные характеристики приложения, его конфигурацию и определяет каждый из компонентов данного приложения. Для запуска компонента приложения системе Android необходимо знать, что компонент существует. Для этого она читает файл AndroidManifest.xml приложения (файл манифеста). В этом файле, который должен находиться в корневой папке приложения, должны быть объявлены все компоненты приложения.

Объявляет имя Java-пакета данного приложения (строка 3). Имя пакета служит уникальным идентификатором приложения. Атрибут android:icon в элементе указывает на ресурсы для значка, который обозначает приложение. Атрибут android:name в элементе указывает полное имя класса подкласса Activity, а атрибут android:label указывает строку, которую необходимо использовать в качестве метки операции, отображаемой для пользователя. Описывает компоненты приложения – Activity,Service, Broadcast Receiver и Content Provider, из которых состоит данное приложение. Эти объявления позволяют Андроид знать чем компоненты являются и при каких условиях они могут быть запущены.

Помимо объявления компонентов приложения, манифест служит и для других целей, среди которых: ⎯ указание всех полномочий пользователя, которые требуются приложению, например, разрешения на доступ в Интернет или на чтение контактов пользователя; ⎯ объявление минимального уровня API, требуемого приложению, с учетом того, какие API-интерфейсы оно использует; ⎯ объявление аппаратных и программных функций, которые нужны приложению или используются им, например камеры, службы Bluetooth или сенсорного экрана; ⎯ указание библиотек API, с которыми необходимо связать приложение (отличные от API-интерфейсов платформы Android), например библиотеки Google Maps ; ⎯ и многое другое И все же основная задача манифеста – это информировать систему о компонентах приложения.

Все модули в проекте описываются файлом setting.gradle.

Файл build.gradle содержит информацию, которая используется при построении проекта.

Каждый модуль имеет свой файл build.gradle, который определяет конфигурацию построения проекта, специфичную для данного модуля. Так, если мы посмотрим на содержимое папки app, то как раз найдем в ней такой файл. На начальном этапе данные файлы не столь важны, достаточно лишь понимать, для чего они нужны.



1. **Ресурсы и обращение к ним (R). strings.xml. Получения ресурсов в классе Activity. Предоставление альтернативных ресурсов. Квалификаторы конфигурации.**

Если возьмем стандартный проект Android Studio, который создается по умолчанию, то там уже есть несколько папок для различных ресурсов в res:

* + - animator/: анимация свойств;
    - anim/: xml-файлы, определяющие tween-анимацию;
    - color/: xml-файлы, определяющие список цветов; ⎯ drawable/: графические файлы (.png, .jpg, .gif);
    - mipmap/: графические файлы, используемые для иконок приложения под различные разрешения экранов;
    - layout/: макеты;
    - menu/: меню приложения;
    - raw/: различные файлы, которые сохраняются в исходном виде;
    - values/: xml-файлы, которые содержат различные значения, например, ресурсы строк;
    - xml/: Произвольные xml-файлы.

Альтернативные ресурсы предназначены для работы с определенными конфигурациями. Чтобы указать, что группа ресурсов предназначена для определенной конфигурации, добавьте соответствующий квалификатор к имени каталога.

Квалификатор hdpi (на рис.28) указывает, что ресурсы в этом каталоге предназначены для устройств, оснащенных экраном высокой плотности.

Land, port, h<>dp, w<>dp и т.д

По умолчанию для ресурсов строк (хранится в виде пар «имя/значение строки») применяется файл strings.xml, но разработчики могут добавлять дополнительные файлы ресурсов в каталог проекта res/values. Необходимо соблюдать структуру файла, иметь корневой узел <resources> и иметь один или несколько элементов <string>

Для получения ресурсов в классе активности используется метод getResources(), который возвращает объект android.content.res.Resources. Чтобы получить ресурс, надо у полученного объекта Resources вызвать один из методов : ⎯ getString(): получает строку из файла strings.xml по числовому идентификатору; ⎯ getDimension(): получает числовое значение из ресурса dimen; ⎯ getDrawable(): получает графический файл; ⎯ getBoolean(): получает значение boolean и т.д.

1. **Понятие Активности. Создание, состояние и жизненный цикл активности**

Каждая активность (Activity) – это экран (по аналогии с формой), который приложение может показывать пользователям. Чем сложнее создаваемое приложение, тем больше экранов (Activity) потребуется. При создании приложения потребуется, как минимум, начальный (главный) экран. При необходимости этот интерфейс дополняется второстепенными активностями, предназначенными для ввода информации, вывода и предоставления дополнительных возможностей. Запуск новой активности или возврат из нее приводит к «перемещению» между экранами. Для создания новой активности необхоимо наследование от класса Activity или AppCompatActivity. Внутри реализации класса надо определить пользовательский интерфейс и реализовать требуемый функционал.

Состояние каждой активности определяется ее позицией в стеке активностей, запущенных в данный момент. При запуске новой Activity представляемый ею экран помещается на вершину стека. Если пользователь нажимает кнопку Назад или эта активность закрывается каким-то другим образом, на вершину стека перемещается (и становится активной) нижележащая активность

Activity могут находиться в одном из четырех возможных состояний.

Active (активное).

Activity находится на переднем плане (на вершине стека) и имеет возможность взаимодействовать с пользователем. Android будет пытаться сохранить ее работоспособность любой ценой, при необходимости прерывая работу других Activity, находящихся на более низких позициях в стеке. При выходе на передний план другой Activity работа данной Активности будет приостановлена или остановлена.

Paused (приостановленное).

В этом состоянии Activity может быть видна на экране, но не может взаимодействовать с пользователем: в этот момент она приостановлена. Это случается, когда на переднем плане находятся полупрозрачные или плавающие (например, диалоговые) окна. Работа приостановленной активности может быть прекращена, если ОС необходимо выделить ресурсы активности переднего плана. Если активность полностью исчезает с экрана, она останавливается.

Stopped (остановленное).

Activity невидима, она находится в памяти, сохраняя информацию о своем состоянии. Такая активность становится кандидатом на преждевременное закрытие, если системе потребуется память. При остановке активности разработчику важно сохранить данные и текущее состояние пользовательского интерфейса (состояние полей ввода, позицию курсора и т. д.). Если активность завершает свою работу или закрывается, она становится неактивной.

Inactive (неактивное).

Когда работа Activity завершена, она находится в неактивном состоянии. Такие активности удаляются из стека и должны быть перезапущены, чтобы их можно было использовать.

Жизненный цикл

onCreate(Bundle saveInstanceState).

Это первый метод, с которого начинается выполнение активности (рис. 6.3). В нем производится первоначальная настройка активности. В частности, создаются объекты визуального интерфейса.

onStart ().

В методе onStart() осуществляется подготовка к выводу активности на экран устройства. Как правило, этот метод не требует переопределения и всю работу производит встроенный код. После завершения работы метода активность отображается на экране, вызывается метод onResume, а активность переходит в состояние Resumed.

onRestoreInstanceState().

После завершения метода onStart() вызывается метод onRestoreInstanceState(), который призван восстанавливать сохраненное состояние из объекта Bundle. Этот метод вызывается только тогда, когда Bundle не равен null и содержит ранее сохраненное состояние. Так, при первом запуске приложения метод onRestoreInstanceState() не будет вызываться.

onResume ().

Вызывается непосредственно перед тем, как активность начинает взаимодействие с пользователем. На этом этапе активность находится в самом верху стека активностей и в нее поступают данные пользователя.

onPause ().

Если пользователь решит перейти к другой активности, то система вызывает метод onPause. В этом методе можно освобождать используемые ресурсы, приостанавливать процессы, останавливать работу камеры (если она используется) и т. д., чтобы они меньше сказывались на производительности системы. Но на работу данного метода отводится очень мало времени, поэтому не стоит здесь сохранять какие-то данные, особенно если при этом требуется обращение к сети, например, отправка данных или обращение к базе данных. После выполнения этого метода активность становится невидимой, не отображается на экране, но она все еще активна. Если пользователь решит вернуться к ней, то система снова вызовет метод onResume() и активность опять появится на экране (см. рис. 6.3). Другой вариант работы может возникнуть, если система видит, что для работы активных приложений необходимо больше памяти. В этом случае ОС может сама завершить работу активности, которая невидима и находится в фоне.

onSaveInstanceState(Bundle saveInstanceState).

Метод onSaveInstanceState() вызывается после метода onPause(), но до вызова onStop(). В onSaveInstanceState производится сохранение состояния приложения в передаваемом в качестве параметра объекте Bundle. Если ОС уничтожает активность в целях восстановления памяти, объект уничтожается, в результате чего системе не удается просто восстановить состояние 6 активности. Вместо этого ОС необходимо повторно создать активность. Но пользователю неизвестно, что система уже ее уничтожила и создала повторно, поэтому он, скорее всего, ожидает, что Activity осталась прежней. В этой ситуации можно обеспечить сохранение важной информации о состоянии активности путем реализации дополнительного метода обратного вызова, который позволяет сохранить информацию (см. рис. 6.3).

onStop ().

Вызывается в случае, когда активность больше не отображается для пользователя. Пользователь может нажать на кнопку Back (Назад). В этом случае у Activity вызывается метод onStop(). Система вызывает этот метод в качестве первого признака выхода пользователя из Activity (это не всегда означает, что она будет уничтожена).

onDestroy().

Завершается работа активности вызовом метода onDestroy(), который возникает, если система решит убить активность либо при вызове метода finish().

При изменении ориентации экрана ОС завершает активность и затем создает ее заново, вызывая метод onCreate(). В большинстве случаев не следует ее явно завершать. Android выполнит такое управление за вас. Вызов методов завершения может отрицательно сказаться на ожидаемом поведении приложения.

1. **Сохранение и восстановление состояния Activity. onSaveInstanceState(Bundle).**

Нет никаких гарантий, что метод onSaveInstanceState() будет вызван до того, как активность будет уничтожена. Если система вызывает метод onSaveInstanceState(), она делает это до вызова метода onStop(). Однако даже если не реализовать метод onSaveInstanceState(), часть состояния Activity восстанавливается реализацией по умолчанию. В частности, реализация по умолчанию вызывает соответствующий метод onSaveInstanceState() для каждого объекта View в макете, благодаря чему каждое представление может предоставлять ту информацию о себе, которую следует сохранить. Поскольку вызов метода onSaveInstanceState() не гарантируется, следует использовать его только для записи переходного состояния активности (никогда не надо использовать его для хранения постоянных данных). Лучше пользоваться для этого методом onPause().

Данные, которые необходимо сохранить, хранятся в специальном объекте Bundle.

* **put()**: универсальный метод, который добавляет значение типа Object. Соответственно поле получения данное значение необходимо преобразовать к нужному типу
* **putString()**: добавляет объект типа String
* **putInt()**: добавляет значение типа int

и т.д

Получение:

* **get()**: универсальный метод, который возвращает значение типа Object. Соответственно поле получения данное значение необходимо преобразовать к нужному типу
* **getString()**: возвращает объект типа String
* **getInt()**: возвращает значение типа int

и т.д.

1. **Взаимодействие активностей и передача результата.**

Как правило, приложение состоит из нескольких activity, которые слабо связаны друг с другом. Обычно одна из activity в приложении обозначается как «основная», предлагаемая пользователю при первом запуске приложения. В свою очередь, каждая activity может запустить другую activity для выполнения различных действий. Каждый раз, когда запускается новая activity, предыдущая останавливается, однако система сохраняет ее в стеке («стек переходов назад»).

Когда одна activity запускает другую, в жизненных циклах обеих из них происходит переход из одного состояния в другое. Первая activity приостанавливается и завершается (однако она не будет остановлена, если она по-прежнему видима на фоне), а вторая activity создается. В случае, если эти операции обмениваются данным, сохраненными на диске или в другом месте, важно понимать, что первая activity не останавливается полностью до тех пор, пока не будет создана вторая activity. Наоборот, процесс запуска второй накладывается на процесс остановки первой activity. Порядок обратных вызовов жизненного цикла четко определен, в частности, когда в одном и том же процессе находятся две activity , и одна из них запускает другую

1->onPause()

2->onCreate(),onStart(),onResume()

1->onStop()

Обмен происходит через Intent, получение результата через startActivityForResult и onActivityResult

1. **Понятие и назначение Intent. Передача и получение значений. Запуск активности для получения результата.**

Для взаимодействия между различными объектами activity ключевым классом является android.content.Intent. Он представляет собой задачу, которую надо выполнить приложению. Для запуска другой activity достаточно вызвать метод startActivity(), передав в него объект Intent, который описывает запускаемую операцию. В намерении указывается либо точная activity для запуска, либо описывается тип activity, которую вы хотите выполнить (после чего система выбирает для вас подходящую операцию, которая может даже находиться в другом приложении). Intent также может содержать небольшой объем данных, которые будут использоваться запущенной activity. При работе с собственным приложением зачастую требуется лишь запустить нужную activity. Для этого необходимо создать intent, который явно определяет требуемую activity с помощью имени класса.

Первый параметр – это Context. Activity является подклассом Context, поэтому мы можем использовать ее – this. Вкратце, Context – это объект, который предоставляет доступ к базовым функциям приложения таким как: доступ к ресурсам, к файловой системе, вызов Activiy и т.д. Второй параметр – имя класса. При создании записи Activity в манифест-файле мы указываем имя класса. И теперь если мы укажем тот же класс в Intent – то система, просмотрев манифест-файл обнаружит соответствие и покажет соответствующий Activity.

Также при переходе от одной activity к другой мы можем передать различную информацию с помощью метода putExtra().

intent.putExtra("имя", значение);

Фактически мы передаем словарь, который состоит из пар ключ-значение. Вызов intent.putExtra("Product", "Планшет") добавляет в этот словарь пару с ключом "Product" и значением "Планшет". Причем можно передавать не только строковые значения, но и другие, например, числовые, логические. В коде java, применяя метод getIntent(), мы можем получить объект Intent, а с помощью его метода getExtras() - те данные, которые ранее были переданы с объектом Intent. Для получения данных мы вызываем соответствующий метод: например, для получения строковых данных - метод getStringExtra(), для получения данных типа float (если бы такие были) - метод getFloatExtra(), а в качестве параметра используется ключ.

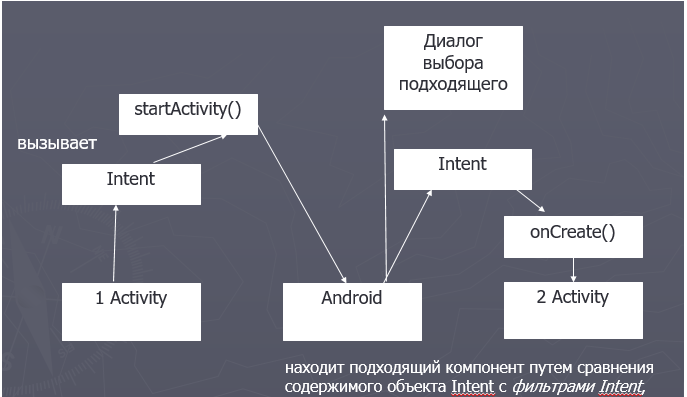
В некоторых случаях после запуска activity может потребоваться получить результат. Для этого вызовите метод startActivityForResult() (вместо startActivity()). Чтобы получить результат после выполнения последующей activity, реализуйте метод обратного вызова onActivityResult(). По завершении последующей activity она возвращает результат в объекте Intent в вызванный метод onActivityResult().

1. **Явные и неявные объекты Intent. Общие Intents. Информация в объекте Intent**

Есть два типа объектов Intent:  
**Явные объекты Intent** указывают компонент, который требуется запустить,  
по имени (полное имя класса). Явные объекты Intent обычно используются для  
запуска компонента из вашего собственного приложения, поскольку вам известно имя класса activity или службы, которую необходимо запустить. Например , можно запустить новую activity в ответ на действие пользователя или запустить службу, чтобы загрузить файл в фоновом режиме.

Intent intent = **new** Intent(**this**, Target.**class**);

startActivity(intent);  
**Неявные объекты Intent** не содержат имени конкретного компонента.  
Вместо этого они в целом объявляют действие, которое требуется выполнить, что дает возможность компоненту из другого приложения обработать этот запрос. Например, если требуется показать пользователю место на карте, то с помощью неявного объекта Intent можно запросить, чтобы это сделало другое приложение, в котором такая возможность предусмотрена.  
Когда создан неявный объект Intent, система Android находит подходящий  
компонент путем сравнения содержимого объекта Intent с *фильтрами Intent*,  
объявленными в **файлах манифеста** других приложений, имеющихся на  
устройстве. Если объект Intent совпадает с фильтром Intent, система запускает этот компонент и передает ему объект Intent. Если подходящими оказываются несколько фильтров Intent, система выводит диалоговое окно, где пользователь может выбрать приложение для выполнения данного действия.

 Фильтр Intent представляет собой выражение в файле манифеста приложения, указывающее типы объектов Intent, которые мог бы принимать компонент. Например, объявив фильтр Intent для activity, вы даете другим приложениям возможность напрямую запускать вашу активность с помощью некоторого объекта Intent. Точно так же, если вы *не* объявите какие-либо фильтры Intent для операции, то ее можно будет запустить только с помощью явного объекта Intent.

**Общие Intents  
*Alarm Clock***Фильтр:  
***Календарь – создать событие***Фильтр:

***Посылка Email с вложениями***ACTION\_SENDTO (for no attachment)  
ACTION\_SEND (for one attachment)  
ACTION\_SEND\_MULTIPLE (for multiple attachments)  
**Излечение специфического файла из файловой системы**

***Указание геолокации***  
***Набор номера и звонок***ACTION\_DIAL - Opens the dialer or phone app.  
• ACTION\_CALL - Places a phone call (requires the CALL\_PHONE permission)  
***Загрузка web URL***

Информация об объекте Intent

Объект Intent содержит информацию, на основании которой система Android  
определяет, какой компонент требуется запустить (например, точное имя  
компонента или категорию компонентов, которые должны получить этот объект Intent), а также сведения, которые необходимы компоненту получателю, чтобынадлежащим образом выполнить действие (а именно — выполняемое действие и данные, с которыми его требуется выполнить).  
*Основные сведения, содержащиеся в объекте Intent:*

**1)Имя компонента**Имя компонента, который требуется запустить. Эта информация является  
необязательной, но именно она и делает объект Intent **явным**. Ее наличие означает, что объект Intent следует доставить только компоненту приложения, определенному по имени. При отсутствии имени компонента объект Intent является **неявным**, а система определяет, какой компонент получит этот объект Intent по другим сведениям, которые в нем содержатся (например, по действию, данным и категории). Поэтому, если вам требуется запустить определенный компонент из своего приложения, следует указать

**2) Действие**Строка, определяющая стандартное действие, которое требуется выполнить  
(например, *view* (просмотр) или *pick*(выбор)). Действие в значительной степени определяет, каким образом структурирована остальная часть объекта Intent,—в частности, что именно содержится в разделе данных и дополнительных данных.  
Для использования объектами Intent в пределах своего приложения (либо для  
использования другими приложениями, чтобы вызывать компоненты из вашегоприложения) можно указать собственные действия. Обычно же следует использовать константы действий, определенные классом Intent или другимиклассами платформы. Вот несколько стандартных действий для запуска операции:  
ACTION\_VIEW  
Используйте это действие когда имеется определенная информация, которую  
активность может показать пользователю, например, фотография в приложении  
галереи или адрес для просмотра в картографическом приложении.  
ACTION\_SEND  
Его еще называют объектом Intent "share" (намерение предоставить общий  
доступ). Это действие следует использовать при наличии определенных данных, доступ к которым пользователь может предоставить через другое приложение, например приложение для работы с электронной почтой или социальными сетям  
**3)Данные**Данные – это URI (объект Uri), ссылающийся на данные, с которыми будет  
выполняться действие и/или тип MIME этих данных. Тип передаваемых данных обычно определяется действием объекта Intent. Например, если действием является **ACTION\_EDIT**, в данных должен содержаться URI документа, который требуется отредактировать. При создании объекта Intent, помимо URI, зачастую бывает важно указать тип данных (их тип MIME).Если требуется задать и URI, и тип MIME, **не** вызывайте **setData()** и **setType()**,  
поскольку каждый из этих методов аннулирует результат выполнения другого.  
Чтобы задать URI и тип MIME всегда используйте метод **setDataAndType()**.  
**4)Категория**Строка, содержащая прочие сведения о том, каким компонентом должна  
выполняться обработка этого объекта Intent. В объект Intent можно поместить любое количество описаний категорий, однако большинству объектов Intent категория не требуется. Вот некоторые стандартные категории:  
**CATEGORY\_BROWSABLE** Целевая activity позволяет запускать себя веббраузером для отображения данных, указанных по ссылке — например,  
изображения или сообщения электронной почты.  
**CATEGORY\_LAUNCHER** Эта activity является начальной для задачи, она  
указана в средстве запуска приложений системы.

**5) Дополнительные данные**Это пары "ключ-значение", содержащие прочую информацию, которая  
необходима для выполнения запрошенного действия. Точно так же, как некоторые действия используют определенные виды URI данных, некоторые действия используют определенные дополнительные данные.

**6) Флаги**Флаги, определенные в классе **Intent**, которые действуют как метаданные для  
объекта Intent. Флаги должны указывать системе Android, каким образом следует запускать активность (например, к какой задаче должна принадлежать activity ) и как с ней обращаться после запуска (например, будет ли она указана в списке последних).Подробные сведения см. в документе, посвященном методу setFlags().

**12. Настройка фильтров для Intent. Разрешение объектов Intent**

**Настройка фильтров для intent**Чтобы указать, какие неявные объекты Intent может принимать ваше  
приложение, объявите один или несколько фильтров Intent для каждого компонента  
приложения с помощью элемента **intent-filter**; в файле манифеста.  
Каждый фильтр Intent указывает тип объектов Intent, которые принимает  
компонент на основании действия, данных и категории, заданных в объекте Intent. Система передаст неявный объект Intent вашему приложению, только если он может пройти через один из ваших фильтров Intent. **Явный** объект Intent всегда доставляется его целевому компоненту, без учета любых фильтров Intent, объявленных компонентом.Компонент приложения должен объявлять отдельные фильтры для каждой уникальной работы, которую он может выполнить   
**data**Объявляет тип принимаемых данных, для чего используется один или  
несколько атрибутов, указывающих различные составные части URI данных  
(scheme, host, port, path и т. д.) и тип MIME  
**category;**Объявляет принимаемую категорию, заданную в объекте Intent, в  
атрибуте name. Значение должно быть текстовой строкой действия, а не константой класса. Дополнения и флаги не играют никакой роли в принятии решения, какой компонент получает намерение. Можно создавать фильтры, в которых будет несколько экземпляров **action,  
data, category.** В этом случае просто нужно убедиться в том, что компонент может справиться с любыми сочетаниями этих элементов фильтра.  
Неявный объект Intent проверяется фильтром путем сравнения объекта Intent с каждым из этих трех элементов. Чтобы объект Intent был доставлен компоненту, он должен пройти все три теста. Если он не будет соответствовать хотя бы одному из них, система Android не доставит этот объект Intent компоненту.

<**intent-filter**>  
<**action**>  
</**action**>  
<**data**>  
</**data**>  
<**category**>  
</**category**>  
</**intent-filter**>

*Константы действия*

|  |  |
| --- | --- |
| • | ACTION\_ANSWER — Открывает активность, которая связана с входящими звонками. Это действие обрабатывается стандартным экраном для приема звонков; ACTION\_CALL — инициализирует обращение по телефону; ACTION\_DELETE — Запускает активность, с помощью которой можно удалить данные, указанные в пути URI внутри намерения; ACTION\_EDIT — Отображает данные для редактирования пользователем; ACTION\_INSERT — Открывает активность для вставки в Курсор (Cursor) нового элемента, указанного с помощью пути URI. Дочерняя активность, вызванная с этим действием, должна вернуть URI, ссылающийся на вставленный элемент; ACTION\_HEADSET\_PLUG - Подключение наушников; ACTION\_MAIN — Запускается как начальная активность задания; ACTION\_PICK - Загружает дочернюю Активность, позволяющую выбрать элемент из источника данных, указанный с помощью пути URI. При закрытии должен возвращаться URI, ссылающийся на выбранный элемент. |

Активность, которая будет запущена, зависит от типа выбранных данных,  
например при передаче пути content://contacts/people вызовется системный  
список контактов;  
•ACTION\_SEARCH — Запускает активность для выполнения поиска.  
Поисковый запрос хранится в виде строки в дополнительном параметре  
намерения по ключу SearchManager.QUERY;

|  |  |
| --- | --- |
| • | ACTION\_SEND — Загружает экран для отправки данных, указанных в намерении. Контакт-получатель должен быть выбран с помощью полученной |

<intent-filter>  
<action android:name="android.intent.action.EDIT" />  
<action android:name="android.intent.action.VIEW" />  
...  
</intent-filter>

|  |  |
| --- | --- |
| • | ACTION\_SENDTO — Открывает активность для отправки сообщений контакту, указанному в пути URI, который передаётся через намерение; ACTION\_SYNC — Синхронизирует данные сервера с данными мобильного устройства; ACTION\_TIMEZONE\_CHANGED - Смена часового пояса; |

•ACTION\_VIEW — Наиболее распространенное общее действие. Для данных, передаваемых с помощью пути URI в намерении, ищется наиболее  
подходящий способ вывода. Выбор приложения зависит от схемы  
(протокола) данных. Стандартные адреса http: будут открываться в браузере,  
адреса tel: — в приложении для дозвона, geo: — в программе Google Maps, а  
данные о контакте — отображатся в приложении для управления контактной  
информацией;  
•ACTION\_WEB\_SEARCH — Открывает активность, которая ведет поиск в  
интернете, основываясь на тексте, переданном с помощью пути URI (как  
правило, при этом запускается браузер);  
**Принцип работы фильтров намерений**При использовании метода **startActivity()** передаваемое неявное намерение, как правило, доходит лишь до одной активности. Если для выполнения заданного действия с указанными данными годятся сразу несколько активностей, пользователю предоставляется список выбора. Процесс, когда решается, какуюактивность лучше запустить, называется Утверждением намерений. Его цель — найти наиболее подходящий фильтр намерений. В целом весь алгоритм работает следующим образом.  
1. Android собирает список всех доступных Фильтров намерений из  
установленных пакетов.  
2. Фильтры, которые не соответствуют действию или категории Намерения,  
удаляются из списка. Совпадение происходит только в том случае, если  
Фильтр намерений содержит указанное действие (или если действие для него  
вовсе не задано). Совпадения не произойдет, только если ни одно из действий Фильтра намерений не будет эквивалентно тому, которое задано в Намерении. Для категорий процесс соответствия более строгий. Фильтр намерений должен включать в себя все категории, заданные в полученном Намерении. Фильтр, для которого категории не указаны, может соответствовать только таким же Намерениям (нет категорий).  
3. Наконец, каждая часть пути URI из Намерения сравнивается с тегом data  
Фильтра намерений. Если в Фильтре указаны схема (протокол),  
сервер/принадлежность, путь или тип MIME, все эти значения проверяются насоответствие пути URI из Намерения. При любом несовпадении Фильтр будет удален из списка. Если в Фильтре намерений не указано ни одного параметраdata, его действие будет распространяться на любые данные.  
o MIME — тип данных, который должен совпасть. При сравнении типов  
данных вы можете использовать маски, чтобы охватывать все подтипы  
(например, cats/\*). Если в Фильтре намерения указан тип данных, он  
должен совпасть с тем, который значится в намерении, при отсутствии тега data подойдет любой тип.

o Схема — это протокольная часть пути URI, например http:, mailto: или  
tel:.  
o Имя сервера (или принадлежность данных) — часть URI между схемой  
и самим путем (например, www.google.com). Чтобы совпало имя  
сервера, схема Фильтра намерений также должна подойти.  
o После имени сервера идет путь к данным (например, /ig). Путь пройдет  
проверку только после схемы и имени сервера, содержащихся в теге.  
4. Когда вы неявным образом запускаете Активность и вышеописанный процесс  
возвращает более одного совпадения, пользователю выводится список со  
всеми вариантами

**13. Файловая система Android. Internal storage. External storage.**  
Приложения, их APK, файлы odex (скомпилированный ahead-of-time  
Java-код) и ELF-библиотеки устанавливаются в */system/app* (для  
приложений, поставляемых с системой) или в /*data/app* (для  
установленных пользователем приложений). Каждому  
предустановленному приложению при создании сборки Android  
выделяется папка с именем вида /*system/app/Terminal*, а для  
устанавливаемых пользователем приложений при установке создаются  
папки, имена которых начинаются с их имени пакета.  
Содержимое /*system* описывает систему и содержит большинство  
составляющих её файлов. /*system* располагается на отдельном разделе  
flash-памяти, который по умолчанию монтируется в режиме read-only;  
обычно данные на нём изменяются только при обновлении системы. /*data*также располагается на отдельном разделе и описывает изменяемое  
состояние конкретного устройства, в том числе пользовательские  
настройки, установленные приложения и их данные, кэши и т. п. Очистка  
всех пользовательских данных заключается просто в очистке содержимого  
раздела *data*; нетронутая система остаётся установлена в разделе *system*.  
Для хранения изменяемых данных каждому приложению выделяется  
папка в /*data*/*data*. Доступ к этой папке есть только у самого приложения –  
то есть только у UID, под которым запускается это приложение (если  
приложение использует несколько UID, или несколько приложений  
используют общий UID). В этой папке приложения сохраняют настройки,  
кэш (в подпапках *shared*\_*prefs* и *cache* соответственно) и любые другие  
нужные им данные:  
Итак  
*/data* - хранит изменяемые данные,  
/*system* - хранит систему,  
/*vendor* - предназначен для файлов, специфичных для этой сборки  
Android, а не входящих в «стандартный» Android,  
/*dev* - хранит файлы устройств и другие специальные файлы.  
Все устройства Android имеют две области хранения файлов:  
*внутреннюю память* и *внешние хранилища*. Эти области появились в  
первые годы существования Android, когда на большинстве устройств  
имелись встроенная память (внутреннее хранилище) и карты памяти,  
например micro SD, внешнее хранилище. Некоторые устройства делят  
встроенную память на внутренний и внешний разделы, так что даже без  
съемных носителей в системе имеется две области хранения файлов, и APIинтерфейс работает одинаково вне зависимости от типа внешнего  
хранилища.  
Внешнее хранилище играет роль домашней папки пользователя –  
именно там располагаются такие папки, как *Documents*, *Download*, *Music* и  
*Pictures*.  
В отличие от внутреннего хранилища, разделённого на папки  
отдельных приложений, внешнее хранилище представляет собой «общую  
зону». К нему есть полный доступ у любого приложения, получившего  
соответствующее разрешение от пользователя.  
Исходно предполагалось, что внешнее хранилище действительно  
будет располагаться на внешней SD-карте, поскольку в то время объем SDкарт значительно превышал объем встраиваемой в телефоны памяти. С тех  
пор условия изменились; в современных телефонах часто нет слота для SDкарты, зато устанавливается огромное количество встроенной памяти.  
Поэтому в современном Android практически всегда и внутреннее, и  
внешнее хранилища располагаются во **встроенной памяти**. Настоящий  
путь, по которому располагается внешнее хранилище в файловой системе,  
имеет форму /*data*/*media*/0 (для каждого пользователя устройства  
создаётся отдельное внешнее хранилище). В целях совместимости до  
внешнего хранилища также можно добраться по путям  
/*sdcard*, /*mnt*/*sdcard*, /*storage*/*self*/*primary*, /*storage*/*emulated*/0, нескольким  
путям, начинающимся с /*mnt*/*runtime*/ и другим.  
С другой стороны, у многих устройств все-таки есть слот для SDкарты. Вставленную в Android-устройство SD-карту можно использовать  
как обычный внешний диск. Кроме того, Android позволяет  
«заимствовать» SD-карту и разместить внутреннее и внешнее хранилище  
на ней (это называется заимствованным хранилищем – *adopted storage*).  
При этом система переформатирует SD-карту и шифрует содержимое –  
хранящиеся на ней данные невозможно прочесть, подключив ее к другому  
устройству.

Internal Storage (внутренняя память)

Внутренная память имеет следующие характеристики:  
⎯энергонезависимая;  
⎯х р а н я  
⎯cохраненные данные в памяти позволяют читать и записывать  
файлы;  
⎯файлы могут быть доступны только данному приложению, а  
не другим приложениям. Приложение *всегда имеет разрешение на чтение  
и запись файлов в свой внутренний каталог* на internal storage. Не требует  
разрешений для обращения;  
⎯файлы хранятся до тех пор, пока приложение остается на  
устройстве. Когда оно удаляется, все связанные файлы автоматически  
удалятся (рис. 6.1);

файлы хранятся в разделе data/data/ за которым следует имя

пакета приложения;

пользователь может явно разрешить другим приложениям

доступ к файлам;

чтобы сделать данные конфиденциальными, можно

использовать режим MODE\_PRIVATE;

данные хранятся в файле в битовом формате, поэтому

требуется преобразовать их перед добавлением в файл или п извлечением

из него;

для непосредственного чтения и записи файлов применяются стандартные классы Java из пакета *java*.*io*.  
Есть следующие режимы:  
⎯*MODE\_PRIVATE*: в этом режиме данные, сохраненные ранее,  
всегда переопределяются текущими данными, т. е. каждый раз, когда  
добавляется новая запись в файл, предыдущий контент удаляется или  
переопределяется (режим по умолчанию);  
⎯*MODE\_APPEND*: в этом режиме данные добавляются к  
существующему контенту.

**External Storage (внешняя память)**

Внешняя память имеет следующие характеристики:  
⎯энергонезавимая;

может быть в собственной памяти устройства (эмуляция) или на

внешнем носителе (SD-карте);

доступна не всегда, пользователь может подключать и отключать

хранилища, например USB-накопители;

для доступа к External-памяти требуется разрешение,

устанавливаемое в файле манифеста приложения;

хранилища доступны для чтения везде, поэтому невозможно

контролировать чтение сохраненных в них данных;

при удалении пользователем приложения система Android удаляет

из внешних хранилищ файлы этого приложения, только если они

сохраняются в директории из getExternalFilesDirectory();

доступность памяти должна проверяться;

могут содержать public- и private-файлы.

Внешнее хранилище лучше всего подходит для файлов без  
ограничений доступа и файлов, которые нужно сделать доступными  
другим приложениям или пользователю.

Хотя внешнее хранилище может быть изменено пользователем и  
другими приложениями, существует две категории файлов, которые в нем  
можно сохранять, это общедоступные (*public*) и личные (*private*) файлы:

**14. Файловая система Android. SharedPreferences. DataStore**

**SharedPreferences***SharedPreferences* (общие настройки) – тип файла, который позволяет  
сохранять информацию в формате ключ - значение для примитивных типов  
данных: *boolean*, *float*, *int*, *long* и *string*. Часто используется для сохранения  
настроек приложения.  
Пары ключ - значение записываются в файлы XML, которые  
сохраняются в течение сеансов, даже если приложение закрыто. Можно  
вручную указать имя файла для сохранения.  
***6.3.1. Получение доступа***Для получения доступа к настройкам в коде приложения используются  
три метода:  
⎯*getPreferences*() – вызывается внутри активности, чтобы обратиться  
к определенному для нее предпочтению; можно вызвать метод без  
указания названия настроек. Доступ к возвращенному ассоциативному  
массиву настроек ограничен активностью, из которой он был вызван.  
Каждая активность поддерживает только один безымянный объект  
*SharedPreferences*;  
⎯*getSharedPreferences*() – вызывается внутри активности, чтобы  
обратиться к предпочтению на уровне приложения;

getDefaultSharedPreferences() – вызывается из объекта

настройку,

PreferencesManager, чтобы получить общедоступную

предоставляемую Android.  
Все эти методы возвращают экземпляр класса *SharedPreferences*, из  
которого можно получить соответствующую настройку с помощью  
следующих методов:  
⎯*getBoolean(String key, boolean defValue)*;  
⎯*getFloat(String key, float defValue)*;  
⎯*getInt(String key, int defValue)*;  
⎯*getLong(String key, long defValue)*;  
⎯*getString(String key, String defValue)*.  
  
**6.4 DataStore**  
В последних версиях произошла миграция от SharedPreferences к  
DataStore.  
Недостатки SharedPreferences:  
•вручную необходимо переключиться на фоновый поток  
не может сигнализировать об ошибках, IOException  
•не защищены от исключения времени выполнения, может  
вызвать исключение синтаксического анализа.  
•нет транзакционного API  
Jetpack DataStore – решение, которое позволяет хранить пары ключ значение или типизированные объекты с protocol buffers. DataStore  
позволяет хранить асинхронно, согласованно и транзакционно.  
Protocol buffers - это не зависящий от языка и платформы механизм  
Google для сериализации структурированных данных – похож на XML, но  
меньше, быстрее и проще. Структурированные данные определяются один  
раз и используя различные потоки считываются и записываются.  
**DataStore** предоставляет две разные реализации: **Preferences  
DataStore** и **Proto DataStore**.  
***Preferences DataStore*** хранит данные и получает доступ с помощью  
ключей. Эта реализация не требует предопределенной схемы и не  
обеспечивает безопасность типов.  
  
***Proto DataStore*** хранит данные как экземпляры настраиваемого типа.  
Эта реализация требует, чтобы вы определяли схему с использованием  
protocol buffers, и обеспечивает безопасность типов.  
https://howtodoandroid.com/datastore-android-jetpack/  
https://developer.android.com/topic/libraries/architecture/datastore  
https://developer.android.com/codelabs/android-preferences-datastore#5

**15. Меню. Типы меню. Создание и отображение. Обработка нажатий. Изменение меню**

МенюНачиная с версии Android 3.0 (уровень API 11) меню переместилось в ActionBar.  
Существуют следующие виды меню:  
1)меню параметров и строка действий (Option Menu ActionBar): пункты меню параметров размещаются в строке действий в виде сочетания отображаемых на экраневариантов действий и раскрывающегося списка дополнительных вариантов выбор2) контекстное меню и режим контекстных действий (Contex Menu): это  
плавающее меню, которое открывается, когда пользователь длительно нажимает на элемент. В режиме контекстных действий в строке, расположенной вверху экрана, отображаются пункты действий, затрагивающие выбранный контент, причем пользователь может выбрать сразу несколько элементов;  
3)всплывающее меню (Popup menu): отображается вертикальный список пунктов, который привязан к представлению, вызвавшему меню.  
7.1.1 Определение меню в файле XMLДля определения пунктов меню используется стандартный формат XML в ресурсе меню. После этого ресурс меню можно будет загружать как объект Menu в активности или фрагменты.  
Использовать меню в ресурсах рекомендуется по нескольким причинам:  
1) в XML проще визуализировать структуру;  
2) позволяет отделить контент меню от кода, определяющего работу приложения;  
3) позволяет создавать альтернативные варианты меню для разных версий  
платформы, размеров экрана и других конфигураций.  
Чтобы определить меню, создается файл в папке res/menu/имя.xml проекта и  
определяется меню со следующими элементами (результат выполнения представлен на  
рис):  
<?xml version="1.0" encoding="utf-8"?><menu xmlns:android="http://schemas.android.com/apk/res/android">  
<item android:id="@+id/newA"android:icon="@drawable/metro\_android2\_black"android:title="@string/newel"/>  
<item android:id="@+id/nextB"android:icon="@drawable/beret"android:title="@string/nextel" />  
</menu>  
  
  
<menu xmlns:android="http://schemas.android.com/apk/res/android">  
  
<item android:id="@+id/file"android:title="@string/file" >  
<!-- вложенное меню --><menu>  
<item android:id="@+id/create\_new"android:title="@string/create\_new" />  
<item android:id="@+id/open"android:title="@string/open" />  
</menu>  
</item>  
</menu>  
Если приложение  
предназначено для версии Android 3.0 (уровень API 11) и более поздних, пункты меню параметров будут отображаться в строке действий. Чтобы обеспечить быстрый доступ к важным действиям, можно принудительно разместить несколько пунктов меню в строке действий, добавив android:showAsAction="ifRoom" к соответствующим элементам  
В современных устройствах меню является частью ActionBar.  
7.1.3 Вывод меню на экранМы определили меню, но само определение элементов в файле еще не создает меню.  
Это лишь декларативное описание. Чтобы вывести его на экран, надо вызвать его в классе  
Activity. Для этого надо переопределить метод onCreateOptionsMenu:  
@Override  
public boolean onCreateOptionsMenu(Menu menu) {  
MenuInflater inflater = getMenuInflater();  
inflater.inflate(R.menu.menu\_ops, menu);  
return true;  
}  
Программа должна сконвертировать созданный ресурс меню в программный объект.  
Для этой цели существует специальный метод MenuInflater.inflate(). Он предназначен для вывода меню при нажатии кнопки на устройстве. В качестве первого параметра метод принимает ресурс, представляющий декларативное описание меню в XML, и наполняет им объект menu, переданный в качестве второго параметра. Метод должен возвращать значение true, чтобы меню было видимым на экране. Метод getMenuInflater() возвращает экземпляр класса MenuInflater, который  
используем для чтения данных меню из XML..  
7.1.4 Обработка нажатийКогда пользователь выбирает пункт меню (в том числе пункты из строки действий), система вызывает метод onOptionsItemSelected() активности. Этот метод передает в параметрах класс MenuItem. Например, (результат работы фрагмента листинга представлен на рис. 7.4):  
@Override  
public boolean onOptionsItemSelected(MenuItem item) {  
// TODOToast.makeText(this, item.getTitle(), Toast.LENGTH\_LONG).show();  
return super.onOptionsItemSelected(item);  
}  
Когда пункт меню успешно обработан, возвращается true. Если пункт меню не обрабатывается, следует вызвать реализацию суперкласса onOptionsItemSelected()

***Программное создание меню и изменение пунктов меню во время выполнения  
приложения***Если требуется изменять меню в зависимости от событий, которые возникают в течение жизненного цикла активности, сделать это можно в методе *onPrepareOptionsMenu*(). Этот метод вызывается каждый раз, когда меню отображается или перерисовывается. Он передает объект *Menu* в том виде, в котором он в данный момент существуе. Его можно изменить путем, например, добавления, удаления или отключения пунктов меню:  
@Override  
**public boolean** onPrepareOptionsMenu (Menu menu){  
menu.add(**"Delete"**);  
menu.add(**"Move"**);  
*// menu.removeItem(R.id.nextB);***return super**.onPrepareOptionsMenu(menu);  
}  
У метода *add()* могут быть четыре параметра:  
*идентификатор группы –* позволяет связывать пункт меню с группой других пунктов  
этого меню;  
*идентификатор пункта –* для обработчика события выбора пункта меню;  
*порядок расположения пункта в меню –* позволяет определять позицию в меню, по  
умолчанию (*Menu*.*NONE* или 0) пункты идут в том порядке, как задано в коде;  
*заголовок –* текст, который выводится в пункте меню (результат выполнения  
представлен на рис.7.5):

**16. Контекстное плавающее меню. Режим контекстных действий. Меню параметров и строка действий Action Bar. Toolbar**

***7.1.6 Создание контекстного меню***Кроме стандартного меню в Android используется также *контекстное меню*,  
вызываемое при нажатии на объект в течение двух-трёх секунд (событие long-press). В отличие от обычного меню, в контекстном меню не поддерживаются значки и быстрые клавиши. Второе важно отличие *–* контекстное меню применимо к *View*, а меню к *Activity*.  
Поэтому в приложении может быть одно меню и несколько контекстных меню, например, у каждого элемента *TextView*.  
Существует два способа предоставления возможности контекстных действий:  
⎯*в плавающем контекстном меню:* меню отображается в виде плавающего списка пунктов меню (наподобие диалогового окна). Пользователи могут каждый раз выполнять контекстное действие только с одним элементом   
⎯*в режиме контекстных действий* (представлен на рис.7.7): этот режим является системной реализацией *ActionMode*, которая отображает *строку контекстных действий* вверху экрана с пунктами действий, которые затрагивают выбранные элементы. Когда этот режим активен, пользователи могут одновременно выполнять действие с несколькими элементами (если это допускается приложением). Если этот режим предусмотрен, именно *его рекомендуется использовать* для отображения контекстных действий.  
  
Рассмотрим алгоритм *создания плавающего контекстного меню***.** Вначале  
зарегистрируем класс *View*, с которым следует связать контекстное меню, вызвав метод  
*registerForContextMenu*() и передав ему *View*:  
setContentView(R.layout.***activity\_menu***);  
ImageView img=(ImageView) findViewById(R.id.***image\_book***);  
registerForContextMenu(img);  
Если активность использует *ListView* или *GridView* и требуется, чтобы каждый элемент предоставлял одинаковое контекстное меню. Зарегистрируем все элементы для контекстного меню. Затем реализуем метод *onCreateContextMenu*() в активности или фрагменте:  
@Override  
**public void** onCreateContextMenu( ContextMenu menu,View v,  
ContextMenu.ContextMenuInfo menuInfo) {  
**super**.onCreateContextMenu(menu, v, menuInfo);  
getMenuInflater().inflate(  
R.menu.***menu\_ops***,menu);  
}  
Когда зарегистрированное представление примет событие длительного нажатия, система вызовет метод *onCreateContextMenu*(). Именно здесь определяются пункты меню. Делается это обычно путем загрузки ресурса меню. Реализуем метод *onContextItemSelected*(). Когда пользователь выбирает пункт меню}  
Рассмотрим вариант реализации *режима контекстных действий*. Он представляет собой системную реализацию класса *ActionMode*. Когда пользователь использует этотрежим, выбирая элемент, вверху экрана открывается *строка контекстных действий*,содержащая действия, которые пользователь может выполнить с выбранными в данный  
момент элементами. Режим контекстных действий отключается, а строка контекстных действий исчезает, когда пользователь снимет выделение со всех элементов, нажмет кнопку «*Назад»* или выберет действие «*Готово»*, расположенное с левой стороны строки.  
*Строка контекстных действий не обязательно бывает связана со строкой  
действий. Они работают независимо друг от друга, даже несмотря на то, что визуально строка контекстных действий занимает положение строки действий.*Для включения режима контекстных действий для отдельных представлений следует:  
1) захватить элемент и добавить обработчик на длительное нажатие:  
EditText etx=(EditText)findViewById(R.id.***edit\_text***);  
etx.setOnLongClickListener(  
**new** View.OnLongClickListener() {  
@Override  
**public boolean** onLongClick(View v) {  
**return false**;  
}  
}  
);  
2) реализовать интерфейс *ActionMode*.*Callback*. В его методах обратного вызова можно указать действия для строки контекстных действий, реагировать на нажатия пунктов и обрабатывать другие события жизненного цикла для режима действий:

**меню параметров и строка действий (Option Menu** ActionBar**)*:* п**ункты меню параметров размещаются в строке действий в виде сочетания отображаемых на экране вариантов действий и раскрывающегося списка дополнительных вариантов выбора

Для определения пунктов меню используется стандартный формат XML в ресурсе . После этого ресурс меню можно будет загружать как объект *Menu* в активности или фрагменты.  
Использовать меню в ресурсах рекомендуется по нескольким причинам:  
1) в XML проще визуализировать структуру;  
2) позволяет отделить контент меню от кода, определяющего работу приложения;  
3) позволяет создавать альтернативные варианты меню для разных версий  
платформы, размеров экрана и других конфигураций.  
Чтобы определить меню, создается файл в папке *res/menu/имя.xml* проекта и  
определяется меню со следующими элементами (результат выполнения представлен на  
рис):  
*<?***xml version="1.0" encoding="utf-8"***?>*<**menu xmlns:android="http://schemas.android.com/apk/res/android"**>  
<**item android:id="@+id/newA"  
android:icon="@drawable/metro\_android2\_black"  
android:title="@string/newel"**/>  
<**item android:id="@+id/nextB"  
android:icon="@drawable/beret"  
android:title="@string/nextel"** />  
</**menu**>

**17. Класс Dialog. AlertDialog. FragmentDialog**

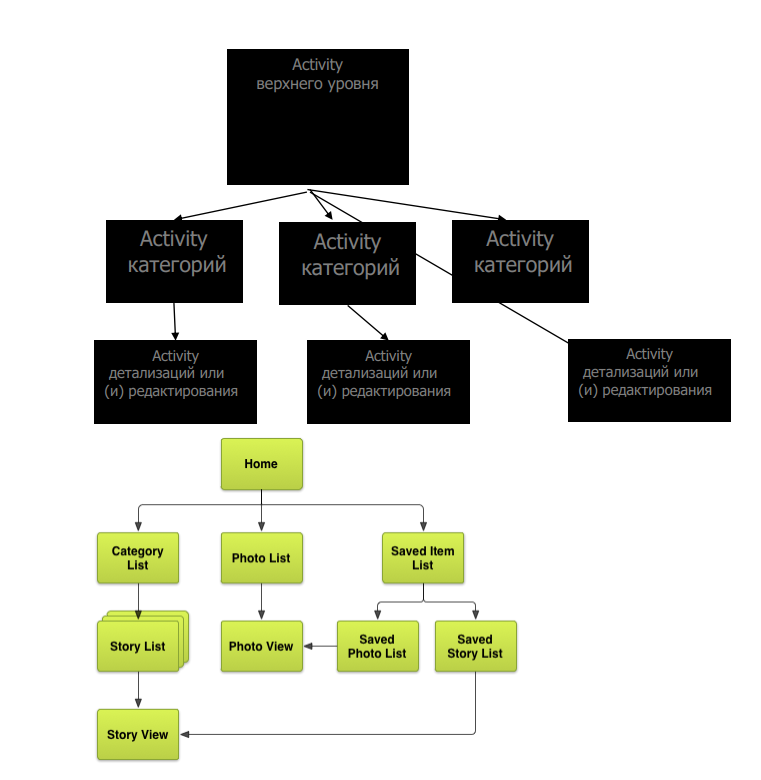
Класс *Dialog* является базовым для всех классов диалоговых окон. Каждое диалоговое окно должно быть определено внутри активности, и активности, в которой будет использоваться. Для его отображения необходимо вызвать метод *showDialog*() и передать в качестве параметра идентификатор диалога (константа, которую надо объявить в коде программы).  
Метод *dismissDialog*() прячет диалоговое окно (но не удаляет), не отображая его на экране. Окно остается в пуле диалоговых окон данной активности. При повторном отображении при помощи метода *showDialog*() будет использована кэшированная версия окна.  
Метод *removeDialog*() удаляет окно из пула окон данной активности. При повторном вызове метода *showDialog*() диалоговое окно придется создавать снова.

***7.2.2 Класс AlertDialog***Диалоговое окно *AlertDialog* является расширением класса *Dialog*, и это наиболее используемый класс диалоговых окон. В создаваемых окнах можно задавать элементы: заголовок; текстовое сообщение; кнопки (от одной до трех); список; флажки; переключатели.  
Рассмотрим пример создания *AlertDialog*. Сначала создадим ресурс:  
В *AlertDialog* можно добавить только по одной кнопке каждого типа: *Positive*, *Neutral*и *Negative*, т. е. максимально возможное количество кнопок в диалоге – три. Диалоговое окно *AlertDialog* очень гибкое в настройках.  
Чтобы диалоговые окна сохраняли своё состояние, рекомендуется использовать методы активности *onCreateDialog*() и *onPrepareDialog*().  
***7.2.3 Класс DialogFragment****DialogFragment* – отображается как диалог и имеет соответствующие методы.  
Построить диалог можно двумя способами: используя свой *layout*-файл и через  
*AlertDialog***.***Builder*.  
Сначала создаем класс, например *DialogF*.*java* (рис.7.14)  
Рис.7.14 Добавление класса диалога *DialogFragment*Для создания диалога с помощью билдера используется метод обратного вызова  
*onCreateDialog*. Диалог будет иметь заголовок, сообщение и три кнопки. Обработчиком  
для кнопок назначаем текущий фрагмент. В *onclick* выводится соответствующий текст в  
лог. Диалог сам закроется по нажатию на кнопку, а метод *dismiss* здесь не нужен (методы  
*onDismiss* и *onCancel* – это закрытие и отмена диалога):

Есть еще один вариант вызова диалога. Это метод *show*(), но на вход он уже принимает не *FragmentManager*, а *FragmentTransaction*. В этом случае система сама вызовет *commit* внутри *show*, можно предварительно поместить в созданную транзакцию какие-либо еще операции или отправить ее в *BackStack*. Можно использовать диалог-фрагмент, как обычный фрагмент и отображать его на *Activity*, а не в виде диалога.

**18. Навигация по активностям. Активности категорий. ListView и Adapters. ListActivity**

**Навигация по активностям**После того как вы разделите свою информацию на активности верхнего  
уровня, категорий и детализации/редактирования, эта классификация может  
использоваться для планирования навигации по приложению. Как правило,  
переход от активностей верхнего уровня к активностям  
детализации/редактирования должен осуществляться через активности  
категорий.  
Пользователь начинает работу с приложением с активности верхнего  
уровня, поэтому они размещаются в начале иерархии.  
Активности категорий занимают место между активностями верхнего  
уровня и активностями детализации/ редактирования. Пользователи будут  
переходить от активностей верхнего уровня к активностям категорий. В  
сложных приложениях иерархия может включать несколько уровней  
категорий и подкатегорий.  
Активности детализации/ редактирования образуют нижний уровень  
иерархии активностей. Пользователи будут переходить к ним от активностей категорий.



Для добавления спискового представления в макет используется  
элемент <ListView>. Дерево наследования представлено ниже:  
Чтобы заполнить списковое представление данными, используйте  
атрибут android:entries и присвойте ему массив строк. Строки из массива будут  
отображаться в списковом представлении в виде набора надписей TextView.  
Пример добавления в макет спискового представления, которое получает  
значения из массива строк:  
Массив определяется точно так же, как это уже делалось ранее, — данные  
включаются в массив strings.xml:  
Будет выведено:  
Такой способ подходит только только для данных, представленных  
статическим массивом в strings.xml.  
<**ListView  
android:id="@+id/list\_ops"  
android:layout\_width="match\_parent"  
android:layout\_height="match\_parent"  
android:entries="@array/options"**/>  
<**string-array name="options"**>  
<**item**>Students</**item**>  
<**item**>Listener</**item**>  
</**string-array**>  
</**resources**>  
  
Чтобы пункты списка реагировали на щелчки, следует реализовать  
слушателя событий. Реализация слушателя событий позволит вам  
обнаруживать конкретные действия пользователя — скажем, щелчки на  
вариантах списка — и реагировать на них.  
Если вы хотите, чтобы варианты списка реагировали на щелчки,  
создайте объект **OnItemClickListener** и реализуйте его метод **onItemClick()**.  
Слушатель **OnItemClickListener** отслеживает щелчки на вариантах списка, а  
метод **onItemClick()** определяет реакцию активности на щелчок. По  
параметрам, передаваемым методу **onItemClick(),** можно получить  
дополнительную информацию о событии — например, получить ссылку на  
вариант из списка, узнать его позицию в списковом представлении (начиная

Итак **Adapter** выступает в качестве посредника между источником  
данных и макетом **AdapterView** — **Adapter** извлекает данные (из источника,  
например, массива или запроса к базе данных) и преобразует каждую запись в представление, которое можно добавить в макет **AdapterView**.

Обзор адаптеровМожно предстваить схему java-иерархии интерфейсов и классов  
адаптеров. В скобках тип: I – это интерфейс, AC – абстрактный класс, C –  
класс. Линии –наследование. Читать следует сверху вниз.  
Интерфейс Adapter. Описывает базовые методы, которые должны  
содержать адаптеры: getCount, getItem, getView и пр.  
Интерфейс ListAdapter. Этот интерфейс должен быть реализован  
адаптером, который будет использован в ListView (метод setAdapter).  
Содержит описание методов для работы с разделителями(separator) списка.  
27  
Интерфейс SpinnerAdapter. Адаптеры, реализующие этот интерфейс,  
используются для построения Spinner(выпадающий список или drop-down).  
Содержит метод getDropDownView, который возвращает элемент  
выпадающего списка.  
Интерфейс WrapperListAdapter. Адаптеры, наследующие этот  
интерфейс используются для работы с вложенными адаптерами. Содержит  
метод getWrappedAdapter, который позволяет вытащить из основного адаптера  
вложенный.  
Класс HeaderViewListAdapter. Готовый адаптер для работы с Header и  
Footer. Внутри себя содержит еще один адаптер (ListAdapter), который можно  
достать с помощью выше рассмотренного метода getWrappedAdapter из  
интерфейса WrapperListAdapter.  
Абстрактный класс BaseAdapter. Содержит немного своих методов и  
реализует методы интерфейсов, которые наследует, но не все. Своим  
наследникам оставляет на обязательную реализацию методы: getView,  
getItemId, getItem, getCount из ListAdapter. Т.е. если хотите создать свой  
адаптер – это класс вам подходит.  
Класс ArrayAdapter<T>. Готовый адаптер, который мы уже  
использовали. Принимает на вход список или массив объектов, перебирает его  
и вставляет строковое значение в указанный TextView. Кроме наследуемых  
методов содержит методы по работе с коллекцией данных – add, insert, remove,  
sort, clear и метод setDropDownViewResource для задания layout-ресурса для  
отображения пунктов выпадающего списка.  
Класс SimpleAdapter. Также готовый к использованию адаптер.  
Принимает на вход список Map-объектов, где каждый Map-объект – это  
список атрибутов. Кроме этого на вход принимает два массива – from[] и to[].  
В to указываем id экранных элементов, а в from имена(key) из объектов Map,  
значения которых будут вставлены в соответствующие (из from) экранные  
элементы.  
Абстрактный класс ResourceCursorAdapter. Содержит методы по  
настройке используемых адаптером layout-ресурсов. Реализует метод  
newView из CursorAdapter.  
Класс SimpleCursorAdapter. Готовый адаптер, похож, на SimpleAdapter.  
Только использует не набор объектов Map, а Cursor, т.е. набор строк с полями.

**ListActivity**Списковая активность специализируется на работе со списком. Она  
автоматически связывается со списковым представлением, поэтому вам не  
придется создавать такое представление самостоятельно.  
Списковые активности определяют свой макет на программном уровне,  
поэтому вам не придется создавать или заниматься сопровождением разметки  
ListView listView = (ListView) findViewById(R.id.***list\_ops***);  
listView.setOnItemClickListener(itemClickListener);  
XML. Макет, генерируемый списковой активностью, содержит одно  
списковое представление. Для обращения к списковому представлению из  
кода активности используется метод **getListView()** списковой активности.  
Такое обращение необходимо для того, чтобы вы могли задать данные,  
которые должны выводиться в списковом представлении.  
Вам не нужно реализовать собственного слушателя. Класс **ListActivity**уже реализует слушателя событий, который обнаруживает щелчки на  
вариантах спискового представления. Вместо того, чтобы создавать  
собственного слушателя и привязывать его к списковому представлению,  
разработчику достаточно реализовать метод **onListItemClick()** списковой  
активности. При таком подходе вам будет проще организовать реакцию  
активности на выбор вариантов спискового представления.  
**ListActivity** вместо класса Activity.  
Другое отличие заключается в том, что вам не нужно назначать макет,  
используемый списковой активностью, вызовом **setContentView().** Дело в том, что списковые активности определяют свои макеты самостоятельно.  
Списковые активности, как и обычные, должны быть зарегистрированы в  
файле AndroidManifest.xml.

**19. Навигация по активностям вперед-назад. Понятие родительская активность**

Навигация вперед- назадВнутренне непротиворечивая навигация является важнейшей  
составляющей пользовательского интерфейса. Мало что так раздражает  
пользователей, как несогласованное или непредсказуемое поведение  
элементов навигации. Кнопка "Вверх" используется для навигации внутри приложения по иерархической структуре его экранов. Например, если на экране A отображается некоторый список и при выборе какого-либо элемента открывается экран B (с подробной информацией об этом элементе), то на экране B должна присутствовать кнопка "Вверх" для возврата к экрану A. Если экран является самым верхним в приложении (то есть главным), он не должен содержать кнопку"Вверх". Системная кнопка "Назад" используется для навигации в обратном хронологическом порядке среди экранов, недавно открытых пользователем.  
Такая навигация основана на порядке появления экранов, а не на иерархии  
приложения.

**Назначение родителя активности**Для назначения родительской активнности надо в файле манифеста  
AndroidManifest.xml  
<**activity  
android:name=".SendInfoActivity"  
android:label="@string/title\_activity\_send\_info"  
android:parentActivityName=".MenuActivity"**/>  
Ваше приложение должно облегчить пользователям поиск пути к главному  
экрану приложения. Простой способ сделать это - предоставить кнопку  
«Вверх» на панели для всех активностей, кроме основной. Когда пользователь  
выбирает кнопку «Вверх», приложение переходит к родительской активности.  
Чтобы включить кнопку «Вверх» для активности, которая имеет родительскую  
активность, вызовите метод **setDisplayHomeAsUpEnabled ()**. Как правило, вы  
делаете это, когда создается activity. Cледующий метод onCreate ()  
устанавливает панель инструментов и включает кнопку вверх на панели  
приложений:  
Если если нет родителя - null.

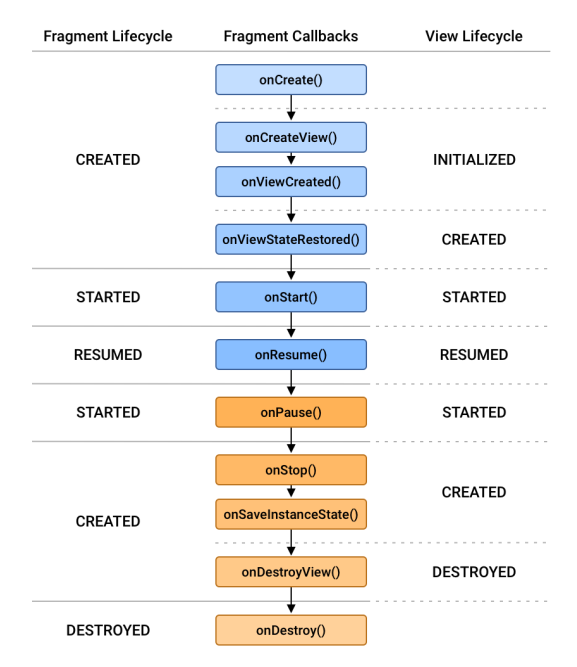
**20. Fragments. Назначение, создание и архитектура приложения на фрагментах. Разновидности фрагментов.**

**Что такое фрагменты**Фрагмент представляет собой многократно используемую часть  
пользовательского интерфейса приложения. Фрагмент определяет и управляет своим собственным макетом, имеет собственный жизненный цикл и может обрабатывать свои собственные входные события. Фрагменты не могут существовать сами по себе - они должны размещаться в активности или другом фрагменте. Иерархия представления фрагмента становится частью иерархии представления хоста или присоединяется к ней.  
Фрагменты впервые появились в Android версии 3.0 (API уровня 11), главным образом, для обеспечения большей динамичности и гибкости пользовательских интерфейсов на больших экранах, например, у планшетов. Поскольку экраны планшетов гораздо больше, чем у смартфонов, они предоставляют больше  
возможностей для объединения и перестановки компонентов пользовательского  
интерфейса.

Фрагмент (класс Fragment) представляет поведение или часть

пользовательского интерфейса в активности (класс Activity).

Фрагмент всегда должен быть встроен в activity, и на его жизненный цикл напрямую влияет жизненный цикл activity. Например, когда activity приостановлена, в том же состоянии находятся и все фрагменты внутри нее, а когда  
activity уничтожается, уничтожаются и все фрагменты. Однако пока activity  
выполняется можно манипулировать каждым фрагментом независимо, например  
добавлять или удалять их.



Для создания фрагмента необходимо создать подкласс класса Fragment (или  
его существующего подкласса). Класс **Fragment** как **onCreate(), onStart(), onPause() и onStop()** и реализует интерфейс  
*LifecycleOwner*. На практике, если требуется преобразовать существующее  
приложение Android так, чтобы в нем использовались фрагменты, достаточно  
просто переместить код из методов обратного вызова операции в соответствующие  
методы обратного вызова фрагмента.  
***Фрагменты содержатся в активностях и находятся под их  
управлением*.  
*Жизненный цикл фрагмента связан с жизненным циклом активности  
Состояния фрагмента*** (определены в Lifecycle.State enum.):  
*INITIALIZED  
CREATED  
STARTED  
RESUMED  
DESTROYED*Фрагмент и его визуальный интерфейс или View имеет отдельный жизненный цикл..  
Состояние можно сохранить во время выполнения метода обратного  
вызова **onSaveInstanceState()** во фрагменте и восстановить его во время  
выполнения onCreate(), onCreateView() или onActivityCreated().  
По умолчанию активность помещается в управляемый системой *стек  
переходов назад*, когда она останавливается (чтобы пользователь мог вернуться к  
ней с помощью кнопки *Назад)*. В то же время, фрагмент помещается в стек  
переходов назад, управляемый активностью, только когда разработчик явно  
запросит сохранение конкретного экземпляра, вызвав метод **addToBackStack()** вовремя транзакции, удаляющей фрагментом.  
В остальном управление жизненным циклом фрагмента очень похоже на  
управление жизненным циклом активности. Поэтому практические рекомендации  
по управлению жизненным циклом активностей применимы и к фрагментам.

***Разновидности и классы фрагментов  
DialogFragment*** – представляет собой перемещаемое диалоговое окно. Дает  
возможность вставить диалоговое окно фрагмента в управляемый стек переходов  
назад, что позволяет пользователю вернуться к закрытому фрагменту.  
***ListFragment*** – предоставляет отображение списка элементов, управляемых  
адаптером *SimpleCursorAdapter*. Предоставляет несколько методов для управления  
списком.  
***PreferenceFragmentCompat***– позволяет выполнять отображение иерархии  
объектов *Preference* в виде списка, аналогично классу *PreferenceActivity*.  
Для транзакций (добавление, удаление, замена) используется класс-помощник *android.app.FragmentTransaction*.

**21. FragmentManager. Back stack. Транзакции фрагментов. Передача значений между фрагментами**

***Fragment Manager****FragmentManager* - класс, отвечающий действия с фрагментами добавление,  
удаление или замена , добавление с стек).  
*Получение доступа в activity*Каждая *FragmentActivity* (это суперкласс активности) и ее подклассы, такие как *AppCompatActivity*, имеют доступ к *FragmentManager* через метод ***getSupportFragmentManager ()***.  
*Получение доступа во фрагменте*Фрагменты также могут содержать один или несколько дочерних фрагментов.  
Внутри фрагмента вы можете получить ссылку на *FragmentManager*, который  
управляет дочерними элементами фрагмента через ***getChildFragmentManager*** ().  
Если вам нужен доступ к его хосту *FragmentManager*, вы можете использовать  
***getParentFragmentManager*** (). *setReorderingAllowed*(true) оптимизирует изменения состояния фрагментов,  
участвующих в транзакции, чтобы анимация и переходы работали правильно.  
*addToBackStack*() фиксирует транзакцию в стеке.. Если добавили или удалили  
несколько фрагментов в рамках одной транзакции, все они будут отменены при извлечении из backstack. Необязательное имя, указанное в вызове  
*addToBackStack*(), дает возможность вернуться к этой конкретной транзакции с помощью *popBackStack* ().

***Fragment Transactions****FragmentManager* может добавлять, удалять, заменять и выполнять другие  
действия с фрагментами. Каждый набор изменений фрагментов называется  
транзакцией. Можно указать, что делать внутри транзакции, используя API,  
предоставляемые классом *FragmentTransaction*. Можно сгруппировать несколько  
действий в одну транзакцию, что полезно когда у вас есть несколько  
одноуровневых фрагментов, отображаемых на одном экране, например, с  
разделенными представлениями.  
Каждую транзакцию можно сохранить в стеке, что позволит пользователю  
перемещаться по изменениям фрагмента - аналогично перемещению по  
активностям.  
Получить экземпляр *FragmentTransaction* из *FragmentManager* можно вызвав  
*beginTransaction* (), как показано в следующем примере:  
FragmentManager fragmentManager = getSupportFragmentManager();  
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();  
Последний вызов каждой *FragmentTransaction* должен зафиксировать  
транзакцию. Вызов *commit* () сообщает *FragmentManager*, что все операции были  
добавлены в транзакцию.  
  
fragmentTransaction.commit();  
Для совместимости поведения флаг переупорядочения по умолчанию не  
включен. Однако требуется, чтобы *FragmentManager* правильно выполнял  
*FragmentTransaction*, особенно когда он работает с backstack и запускает анимацию и переходы. Включение флага гарантирует, что, если несколько транзакций  
выполняются вместе, любые промежуточные фрагменты не претерпевают  
изменений жизненного цикла или не выполняют их анимацию или переходы.  
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction()  
*//...*.setReorderingAllowed(true)  
.commit();  
Для создания динамического UI используют разные методы управления  
транзакций:  
⎯*add*(): добавляет фрагмент к активности;  
⎯*remove*(): удаляет фрагмент из активности;  
⎯*replace*(): заменяет один фрагмент на другой;  
⎯*hide*(): прячет фрагмент (делает невидимым на экране);  
⎯*show*(): выводит скрытый фрагмент на экран;  
⎯*detach*() (API 13): отсоединяет фрагмент от графического интерфейса, но  
экземпляр класса сохраняется;  
⎯*attach*() (API 13): присоединяет фрагмент, который был отсоединён  
методом *detach*().  
Чтобы сохранить эти изменения, вы можете вызвать *addToBackStack*()  
для *FragmentTransaction*.  
Вызов *commit*() не выполняет транзакцию немедленно. Транзакция  
планируется для запуска в основном потоке пользовательского интерфейса, кактолько это будет возможно. Однако при необходимости вы можете вызвать  
*commitNow*(), чтобы немедленно запустить транзакцию фрагмента в потоке  
пользовательского интерфейса.  
Обратите внимание, что *commitNow* несовместим с *addToBackStack*. В качестве альтернативы можете выполнить все ожидающие транзакции *FragmentTransactions*, отправленные вызовами *commit*(), которые еще не были выполнены, путем вызова *executePendingTransactions*(). Этот подход совместим с *addToBackStack*. Для подавляющего большинства случаев лучше подходит *commit*().

***Коммуникации фрагментов***Существует 4 способа коммуникаций между фрагментами

|  |  |
| --- | --- |
| • | **Interface** |
| • | **ViewModel** |
| • | **RxJava** |
| • | **Event Bus** |

**22. Реализация методов жизненного цикла фрагмента**

Как правило, необходимо реализовать следующие методы жизненного  
цикла:  
*onCreate()*Система вызывает этот метод, когда создает фрагмент. В своей реализации  
разработчик должен инициализировать ключевые компоненты фрагмента,  
которые требуется сохранить, когда фрагмент находится в состоянии паузы или возобновлен после остановки.  
*onCreateView()*Система вызывает этот метод при первом отображении пользовательского  
интерфейса фрагмента на дисплее. Для прорисовки пользовательского  
интерфейса фрагмента следует возвратить из этого метода объект View, который является корневым в макете фрагмента. Если фрагмент не имеет  
пользовательского интерфейса, можно возвратить n  
Вызывается для создания иерархии представлений, связанной с фрагментом  
*onPause()*Система вызывает этот метод как первое указание того, что пользователь  
покидает фрагмент (это не всегда означает уничтожение фрагмента). Обычно  
именно в этот момент необходимо фиксировать все изменения, которые должныбыть сохранены за рамками текущего сеанса работы пользователя (поскольку пользователь может не вернуться назад).  
В большинстве приложений для каждого фрагмента должны быть  
реализованы, как минимум, эти три метода. Однако существуют и другие методы обратного вызова, которые следует использовать для управления различными этапами жизненного цикла фрагмента:  
*onAttach()*Вызывается, когда фрагмент связывается с активностью (ему передается  
объект Activity).  
*onViewCreated*()  
Вызывается после создания представления фрагмента.  
*onActivityCreated()*Вызывается, когда метод onCreate(), принадлежащий актиности, возвращает  
управление.  
*onDestroyView()*Вызывается при удалении иерархии представлений, связанной с фрагментом.  
*onDetach()*Вызывается при разрыве связи фрагмента с активностью.

@Override  
protected void onCreate(Bundle savedInstanceState) {  
super.onCreate(savedInstanceState);  
if (savedInstanceState == null) {  
Bundle bundle = new Bundle();  
bundle.putInt("Counter", 0);  
getSupportFragmentManager().beginTransaction()  
.setReorderingAllowed(true)  
.add(R.id.*fragment\_container\_view*, ExampleFragment.class,  
bundle)  
.commit();  
}  
}

* 1. **Выдвижные панели (drawers). Создание TabHost и TabWidget. ViewPager**

За выдвигающую шторку отвечает элемент NavigationView, который входит последним в контейнере DrawerLayout и представляет собой навигационное меню. А перед меню находится вставка include, указывающая на разметку app\_bar\_main.xml.

В 2014 году Google показал новый дизайн и различные новые примеры по навигации. Но вначале они использовали подручные средства, которые были под рукой - фрагменты.

Спустя год компания разработала на основе предка **FrameLayout** новый компонент **NavigationView**, который стал частью библиотеки [Android Design Support Library](http://developer.alexanderklimov.ru/android/android-support-design.php).

Новый подход оказался неожиданным, но логичным. Раз выдвижная шторка содержит навигационное меню, то и класс был спроектирован как меню. Вы можете создать элементы меню в ресурсах **res/menu** стандартным способом и получить готовую навигацию.

Вкладки, как у браузеров, можно реализовать с помощью элементов **TabHost** и **TabWidget**. Компонент **TabHost** позволяет группировать связанные элементы управления в серию страниц-вкладок.

Находится в разделе **Containers**.

Хотя данный компонент не помечен, как устаревший, сама компания Google практически его не использует в своих примерах (если ошибаюсь, то ткните мордочкой в пример). В качестве альтернативы попробуйте [TabLayout (Android Design Support Library)](http://developer.alexanderklimov.ru/android/layout/tablayout.php).

**TabHost** является контейнером, который может содержать элементы **TabWidget**.

**TabHost** показывает ярлычки каждой страницы в своей коллекции. Когда пользователь выбирает вкладку, этот объект посылает сообщение в родительский контейнер **TabHost** для переключения на выбранную вкладку.

Когда вы перетаскиваете **TabHost** на форму по умолчанию, то создаётся готовый шаблон, который нужно немного модифицировать под ваши нужды - можно убрать корневой элемент **LinearLayout** и сделать таким сам **TabHost**.

[TabHost](http://developer.android.com/reference/android/widget/TabHost.html) – корневой элемент вкладок. В нем вертикальный LinearLayout, в котором расположены [TabWidget](http://developer.android.com/reference/android/widget/TabWidget.html)и FrameLayout. TabWidget будет отображать заголовки вкладок, а FrameLayout – содержимое вкладок. В этом FrameLayout мы размещаем все View-компоненты, которые хотим отображать на вкладках. Позже мы (в коде) сообщим вкладке, какой именно компонент она должна показать (явно укажем id), вкладка выберет из этой общей кучи нужный ей компонент и отобразит его, как свое содержимое.

Методы класса TabHost

* **setup()** — инициализирует контейнер вкладок. Необходимо вызывать перед добавлением вкладок, если **TabHost** загружается методом **findViewById()**
* **addTab()** — добавляет новую вкладку
* **setcurrentTab()** — выводит заданную вкладку на передний план

Методы класса TabWidget

Большинство методов для работы с вкладками реализованы в классе **TabWidget**. У вкладки есть индикатор позиции табуляции, информационное наполнение и идентификатор для работы в программном коде. Их необходимо определить созданием экземпляра вложенного класса **TabSpec** (см. код выше).

* **setContent()** - задает ресурс для вкладки
* **setIndicator()** - выводит текст на ярлычке вкладки

В Android есть компонент [ViewFlipper](http://developer.alexanderklimov.ru/android/views/viewflipper.php), который позволяет прокручивать экран влево-вправо. Но он имеет ряд недостатков. **ViewFlipper** не позволяет с лёгкостью добиться эффекта привязки, когда экраны двигаются вместе с пальцем. Также нужен механизм автоматической доводки экранов в ту или иную сторону.

Google предложила другой вариант - использовать компонент **ViewPager**, который входит в состав **Support Package**. В новых проектах необходимая библиотека добавляется автоматически, поэтому мы сразу можем попробовать написать демонстрационный пример.

**ViewPager** относится к категории **ViewGroup** и схож по работе с компонентами на основе **AdapterView** (**ListView** и **Gallery**). На панели инструментов Android Studio компонент можно найти в разделе **Containers**. При использовании **ViewPager** в разметке используйте полное имя класса. Старый вариант.

Данные для отображения компонент берёт из адаптеров:

* **PagerAdapter** (фрагменты не обязательны)
* **FragmentPagerAdapter** (если используется мало фрагментов)
* **FragmentStatePagerAdapter** (если много фрагментов)

Вам нужно унаследовать класс от нужного адаптера и реализовать свои методы. Добавление и удаление экранов реализуется с помощью методов **instantiateItem()**  и **destroyItem()** соответственно. Элементы **View** для отображения можно создавать прямо в адаптере. Такой подход хорош тем, что **ViewPager** можно настраивать так, чтобы в адаптере не хранились все экраны сразу. По умолчанию адаптер хранит текущий экран, и по одному слева и справа от него. Это может сэкономить память, если содержание экранов слишком сложное.

* 1. **RecyclerView CardView**

Компонент **RecyclerView** появился в Android 5.0 Lollipop и находится в разделе **Containers**. Для простоты буду называть его списком, хотя на самом деле это универсальный элемент управления с большими возможностями.

Раньше для отображения прокручиваемого списка использовался **ListView**. Со временем у него обнаружилось множество недостатков, которые было трудно исправить. Тогда решили создать новый элемент с нуля.

Вначале это был сырой продукт, потом его доработали. На данном этапе можно считать, что он стал полноценной заменой устаревшего **ListView**.

Схематично работу **RecyclerView** можно представить следующим образом. На экране отображаются видимые элементы списка. Когда при прокрутке списка верхний элемент уходит за пределы экрана и становится невидимым, его содержимое очищается. При этом сам "чистый" элемент помещается вниз экрана и заполняется новыми данными, иными словами переиспользуется, отсюда название **Recycle**.

Адаптер, который используется в RecyclerView, должен наследоваться от абстрактного класса **RecyclerView.Adapter**. Этот класс определяет три метода:

* **onCreateViewHolder**: возвращает объект ViewHolder, который будет хранить данные по одному объекту State.
* **onBindViewHolder**: выполняет привязку объекта ViewHolder к объекту State по определенной позиции.
* **getItemCount**: возвращает количество объектов в списке

Для хранения данных в классе адаптера определен статический класс ViewHolder, который использует определенные в **list\_item.xml** элементы управления.

Новый компонент **CardView** появился в Android Lollipop (API 21), но благодаря библиотеке совместимости доступен и для старых устройств. По сути является дальнейшим развитием **FrameLayout** и позволяет создавать красивую карточку с тенью и закруглёнными углами, который служит контейнером для других компонентов.

Доступен на панели инструментов в разделе **Containers**. Сейчас компонент входит в состав AndroidX. Если добавлять в студии из панели **Palette** в обычных проектах из шаблонов, то используются два стандартных пакета. Если вы используете собственный проект со своими зависимостями, то добавьте отдельную библиотеку.

* 1. **База данных SQLite. SQL Helper. Создание экземпляра, Вставка, удаление и чтение данных**

Для работы большинства систем управления базами данных необходим специальный процесс сервера базы данных. SQLite обходится без сервера; база данных SQLite представляет собой обычный файл. Когда база данных не используется, она не расходует процессорное время. Это особенно важно на мобильных устройствах, чтобы избежать разрядки аккумулятора

Android хранит базы данных в каталоге /data/data//databases на эмуляторе, на устройстве путь может отличаться.

Помощник SQLite

Помощник SQLite создается расширением класса SQLiteOpenHelper. Он предоставляет средства для создания и управления базами данных. SQLiteOpenHelper содержит два обязательных абстрактных метода:

• onCreate() — вызывается при первом создании базы данных

• onUpgrade() — вызывается при модификации базы данных

Также используются другие методы класса:

• onDowngrade(SQLiteDatabase, int, int)

• onOpen(SQLiteDatabase)

• getReadableDatabase()

• getWritableDatabase()

Класс SQLiteOpenHelper упрощает задачи создания и сопровождения баз данных. Считайте, что это своего рода личный ассистент, который берет на себя служебные операции по управлению базами данных. Рассмотрим некоторые типичные задачи, в решении которых вам поспособствует помощник SQLite

Создание базы данных

При первой установке приложения файл базы данных не существует. Помощник SQLite проследит за тем, чтобы файл базы данных был создан с правильным именем и с правильной структурой таблиц.

Обеспечение доступа к базе данных

Нашему приложению не обязательно знать все подробности о том, где хранится файл базы данных. Помощник SQLite предоставляет удобный объект, представляющий базу данных, и приложение работает с базой через этот объект — тогда, когда сочтет нужным

Сопровождение баз данных

Может случиться так, что структура базы данных изменится со временем. Положитесь на помощника SQLite — он преобразует старую версию в новую, с учетом самых последних изменений в структуре базы данных

Db.execSql

2) Вставляем данные в бд через ContentValues

При добавлении данных в таблицы СУБД применяются объекты класса ContentValues. Каждый такой объект содержит данные одной строки в таблице и, по сути, является ассоциативным массивом с именами столбцов и соответствующими значениями. Поэтому для вставки сначала подготавливаются данные с помощью класса ContentValues. Вы указываете имя колонки таблицы и значение для неё, т.е. работает по принципу "ключ-значение". Когда подготовите все данные во все столбцы, то вызывайте метод insert(), который сразу раскидает данные по столбцам. Способ добен, требует мало кода и легко читаем. Вы создаёте экземпляр класса, а затем с помощью метода put() записываете в нужную колонку нужные данные. После чего вызывается метод insert(), который помещает подготовленные данные в таблицу. У метода insert() три аргумента. В первом указывается имя таблицы, в которую будут добавляться записи. В третьем указывается объект ContentValues, созданный ранее. Второй аргумент используется для указания колонки. SQL не позволяет вставлять пустую запись, и если будет использоваться пустой ContentValue, то укажите во втором аргументе null во избежание ошибки.

Если запись уже существует, но вам нужно изменить какое-то значение, то вместо insert()используйте метод update(). В остальном принцип тот же.

Метод delete() класса SQLiteDatabase работает по тому же принципу, как и метод update()

Считывать данные также можно двумя способами. В любом случае результат возвращается в виде объекта Cursor.

Для чтения из базы данных используйте метод query() с передачей критериев выделения и желаемых столбцов. Метод сочетает элементы insert() и update(), за исключением того, что список столбцов определяет данные, которые вы хотите получить, а не данные для вставки.

Второй способ использует сырой (raw) SQL-запрос. Сначала формируется строка запроса и отдаётся методу rawQuery()

* 1. **SQLite Представление прочитанных данных.**

Одно из возможных решений выглядит так: прочитать список (студентов, заголовков) из базы данных и сохранить информацию в массиве, передаваемом адаптеру массива.

Такое решение работает, но... При такой маленькой базе данных, как у нас, чтение всей информации и хранение ее в массиве (то есть в памяти) не создает проблем. Но если приложение работает с очень большим объемом информации, ее чтение из базы данных займет некоторое время. Кроме того, для хранения массива потребуется много памяти. Вместо этого мы перейдем с класса адаптера массива ArrayAdapter на адаптер курсора CursorAdapter

Адаптер курсора читает столько данных, сколько нужно. Допустим, наша база данных намного больше — например, это списки для университета. Может оказаться, что вместо трех студентов в базе данных придется хранить 2000. Однако в списке при этом будет отображаться лишь малая часть записей. В представлении ListView одновременно может отображаться лишь небольшая часть записей. На устройствах с маленьким экраном изначально могут выводиться, скажем, первые 11. Если бы данные хранились в массиве, нам пришлось бы загрузить все 2000 записей из базы данных в массив и только потом вывести часть данных. С CursorAdapter все работает немного иначе. Когда списковое представление отображается впервые, оно масштабируется по размерам экрана. Предположим, в нем достаточно места для отображения пяти элементов. Компонент ListView не знает, где именно хранятся данные — в массиве или базе данных, — но знает, что он получит данные от адаптера. Соответственно, он обращается к адаптеру и запрашивает первые пять записей. Объект CursorAdapter получает курсор при создании, но обращается к курсору за данными только тогда, когда потребуется. Несмотря на то что таблица базы данных содержит 2000 записей, курсор должен прочитать только первые пять. Такой подход намного эффективнее, и он означает, что данные появятся на экране намного быстрее. Когда пользователь прокручивает список, CursorAdapter приказывает курсору прочитать другие записи из базы данных. Если пользователь 16 ограничивается минимальной прокруткой и открывает только одну новую строку, курсор читает из базы данных одну запись.

* 1. **Выполнение операций в отдельном потоке. AsyncTask. Методы асинхронной задачи. Возврат значений. Отмена асинхронной задачи в процессе выполнения**

Основная проблема с обращениями к медленной базе данных заключается в том, что она может замедлить реакцию приложения на действия пользователя. Чтобы понять, почему это происходит, необходимо задуматься над тем, как работают программные потоки в Android. Начиная с версии Lollipop, существуют три вида потоков, которые необходимо учитывать.

1) Основной поток событий (UI) Этот поток прослушивает интенты, получает сообщения о касаниях от экрана и вызывает все методы внутри ваших активностей. Т.е отвечает за все что происходит с экраном. 2) Поток визуализации. Этот поток, с которым вы обычно не взаимодействуете, читает список запросов на обновление экрана, а затем выдает команды низкоуровневому графическому оборудованию на перерисовку экрана. 3) Все остальные потоки, созданные вами

Класс AsyncTask – помощник над Thread и Handler. Подходит для небольших операций (считатется устаревшим с API 30).

Напрямую с классом **AsyncTask** работать нельзя (абстрактный класс), вам нужно наследоваться от него (extends). Ваша реализация должна предусматривать классы для объектов, которые будут переданы в качестве параметров методу **execute()**, для переменных, что станут использоваться для оповещения о ходе выполнения, а также для переменных, где будет храниться результат. Формат такой записи следующий:

AsyncTask<[Input\_Parameter Type], [Progress\_Report Type], [Result Type]>

Если не нужно принимать параметры, обновлять информацию о ходе выполнения или выводить конечный результат, просто укажите тип **Void** во всех трёх случаях. В параметрах можно использовать только обобщённые типы (Generic), т.е. вместо **int** используйте **Integer** и т.п.

У **AsyncTask** есть несколько основных методов, которые нужно освоить в первую очередь. Обязательным является метод **doInBackground()**, остальные используются исходя из логики вашего приложения.

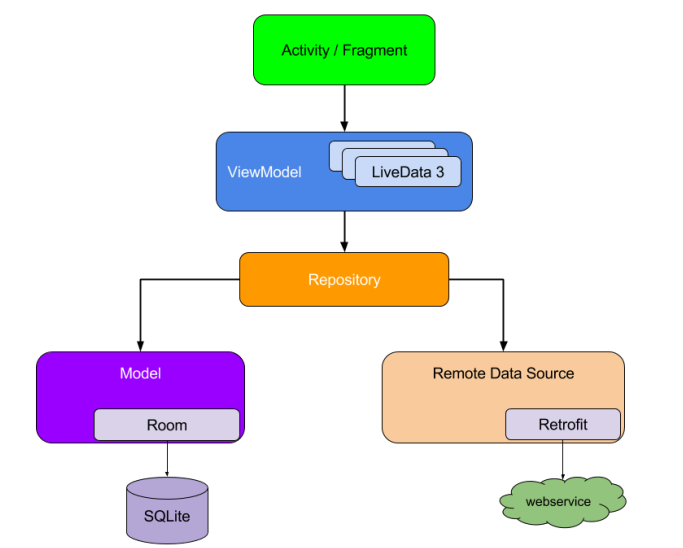
* **doInBackground()** – основной метод, который выполняется в новом потоке. **Не имеет доступа к UI**. Именно в этом методе должен находиться код для тяжёлых задач. Принимает набор параметров тех типов, которые определены в реализации вашего класса. Этот метод выполняется в фоновом потоке, поэтому в нем не должно быть никакого взаимодействия с элементами пользовательского интерфейса. Размещайте здесь трудоёмкий код, используя метод **publishProgress()**, который позволит обработчику **onProgressUpdate()** передавать изменения в пользовательский интерфейс. Когда фоновая задача завершена, данный метод возвращает конечный результат для обработчика **onPostExecute()**, который сообщит о нём в поток пользовательского интерфейса.
* **onPreExecute()** – выполняется перед **doInBackground()**. **Имеет доступ к UI**
* **onPostExecute()** – выполняется после **doInBackground()** (может не вызываться, если **AsyncTask** был отменен). **Имеет доступ к UI**. Используйте его для обновления пользовательского интерфейса, как только ваша фоновая задача завершена. Данный обработчик при вызове синхронизируется с потоком GUI, поэтому внутри него вы можете безопасно изменять элементы пользовательского интерфейса.
* **onProgressUpdate()**. **Имеет доступ к UI**. Переопределите этот обработчик для публикации промежуточных обновлений в пользовательский интерфейс. При вызове он синхронизируется с потоком GUI, поэтому в нём вы можете безопасно изменять элементы пользовательского интерфейса.
* **publishProgress()** - можно вызвать в **doInBackground()** для показа промежуточных результатов в **onProgressUpdate()**
* **cancel()** - отмена задачи
* **onCancelled()** - **Имеет доступ к UI**. Задача была отменена. Имеются две перегруженные версии.

Метод **cancel()** позволяет указать на отмену уже выполняющейся задачи. У метода один boolean-параметр, который указывает, может ли система прервать выполнение потока.

Кроме того, в методе **doInBackground()** можно проверять метод **isCancelled()**. Как только мы выполним метод **cancel()**, метод **isCancelled()** будет возвращать *true* и мы должны завершить метод **doInBackground()**. Таким образом, метод **cancel()** служит своеобразной меткой, что задачу нужно отменить. А метод **isCancelled()** будет считывать результат предыдущего метода и позволет выполнит код для завершения задачи.

* 1. **Архитектура приложения на основе Room. Lifecycle**

Room: держатель базы данных, который служит точкой доступа к базе данных SQLite, использует DAO для запросов.



Итак, Room предоставляет слой абстракции над SQLite, чтобы обеспечить свободный доступ к базе данных SQLite. Наиболее распространенным вариантом использования Room является кэширование фрагментов данных. Когда устройство не может получить доступ к сети, пользователь может просматривать контент, находясь в автономном режиме. Любые пользовательские изменения контента затем синхронизируются с сервером после того, как устройство снова подключится к сети.

Room имеет три основных компонента: Entity, Dao и Database. Рассмотрим их на небольшом примере, в котором будем создавать базу данных для хранения данных по сотрудникам (англ. - employee).

При работе с Room нам необходимо будет писать SQL запросы. Если вы не знакомы с ними, то имеет смысл прочесть хотя бы основы.

Аннотацией [Entity](https://developer.android.com/reference/android/arch/persistence/room/Entity.html) нам необходимо пометить объект, который мы хотим хранить в базе данных. Для этого создаем класс Employee, который будет представлять собой данные сотрудника: id, имя, зарплата:

В объекте [Dao](https://developer.android.com/reference/android/arch/persistence/room/Dao.html) мы будем описывать методы для работы с базой данных. Нам нужны будут методы для получения списка сотрудников и для добавления/изменения/удаления сотрудников.

Аннотацией [Database](https://developer.android.com/reference/android/arch/persistence/room/Database.html) помечаем основной класс по работе с базой данных. Этот класс должен быть абстрактным и наследовать RoomDatabase.

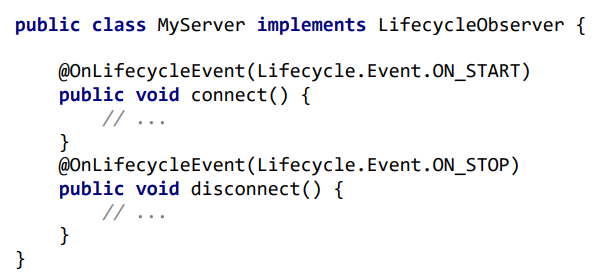
В параметрах аннотации Database указываем, какие Entity будут использоваться, и версию базы. Для каждого Entity класса из списка entities будет создана таблица.

В Database классе необходимо описать абстрактные методы для получения Dao объектов, которые вам понадобятся.

Lifecycle

Довольно часто часть логики приложения завязана на жизненный цикл Activity. Мы включаем что-либо в методах onStart или onResume и выключаем в onPause или onStop. Например. Есть какой-то класс для работы с сервером. Он должен взаимодействовать с сервером, пока Activity открыта. Соответственно, мы будем подключать его к серверу при показе Activity и отключать при скрытии Activity

Это вполне классическая, часто используемая схема. И в простом примере все выглядит понятно. Но в сложных приложениях содержание методов onStart, onStop и пр. может состоять из нескольких десятков строк и быть достаточно запутанным. Для улучшения архитектуры Google рекомендует выносить эту логику из Activity. Это можно сделать так. У Activity есть метод getLifecycle, который возвращает объект Lifecycle. На этот объект можно подписать слушателей, которые будут получать уведомления при смене lifecycle-состояния Activity. Activity и Fragment в Support Library, начиная с версии 26.1.0 реализуют интерфейс LifecycleOwner. Именно этот интерфейс и добавляет им метод getLifecycle.



В Activity методом getLifecycle получаем Lifecycle, и методом addObserver подписываем myServer.

Отписаться от Lifecycle можно методом removeObserver.

* 1. **Архитектура приложения Room и LiveData**

Итак, Room предоставляет слой абстракции над SQLite, чтобы обеспечить свободный доступ к базе данных SQLite. Наиболее распространенным вариантом использования Room является кэширование фрагментов данных. Когда устройство не может получить доступ к сети, пользователь может просматривать контент, находясь в автономном режиме. Любые пользовательские изменения контента затем синхронизируются с сервером после того, как устройство снова подключится к сети.

Room имеет три основных компонента: Entity, Dao и Database. Рассмотрим их на небольшом примере, в котором будем создавать базу данных для хранения данных по сотрудникам (англ. - employee).

При работе с Room нам необходимо будет писать SQL запросы. Если вы не знакомы с ними, то имеет смысл прочесть хотя бы основы.

Аннотацией [Entity](https://developer.android.com/reference/android/arch/persistence/room/Entity.html) нам необходимо пометить объект, который мы хотим хранить в базе данных. Для этого создаем класс Employee, который будет представлять собой данные сотрудника: id, имя, зарплата:

В объекте [Dao](https://developer.android.com/reference/android/arch/persistence/room/Dao.html) мы будем описывать методы для работы с базой данных. Нам нужны будут методы для получения списка сотрудников и для добавления/изменения/удаления сотрудников.

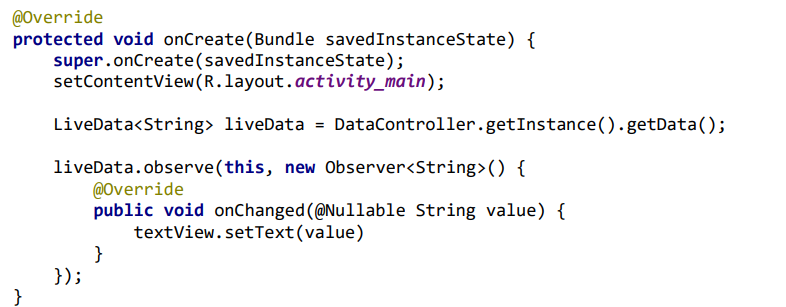
Аннотацией [Database](https://developer.android.com/reference/android/arch/persistence/room/Database.html) помечаем основной класс по работе с базой данных. Этот класс должен быть абстрактным и наследовать RoomDatabase.

В параметрах аннотации Database указываем, какие Entity будут использоваться, и версию базы. Для каждого Entity класса из списка entities будет создана таблица.

В Database классе необходимо описать абстрактные методы для получения Dao объектов, которые вам понадобятся.

LiveData - хранилище данных, работающее по принципу паттерна Observer (наблюдатель). Это хранилище умеет делать две вещи: 1) в него можно поместить какой-либо объект 2) на него можно подписаться и получать объекты, которые в него помещают. Т.е. с одной стороны кто-то помещает объект в хранилище, а с другой стороны кто-то подписывается и получает этот объект. В качестве аналогии можно привести, например, каналы в Telegram. Автор пишет пост и отправляет его в канал, а все подписчики получают этот пост.

Казалось бы, ничего особо в таком хранилище нет, но есть один очень важный нюанс. LiveData умеет определять активен подписчик или нет, и отправлять данные будет только активным подписчикам. Предполагается, что подписчиками LiveData будут Activity и фрагменты. А их состояние активности будет определяться с помощью их Lifecycle объекта.

****

Получаем LiveData из DataController, и методом observe подписываемся. В метод observe нам необходимо передать два параметра:

1) это LifecycleOwner. Activity и фрагменты в Support Library, начиная с версии 26.1.0 реализуют этот интерфейс, поэтому мы передаем this. LiveData получит из Activity его Lifecycle и по нему будет определять состояние Activity. Активным считается состояние STARTED или RESUMED. Т.е. если Activity видно на экране, то LiveData считает его активным и будет отправлять данные в его колбэк

2) это непосредственно подписчик, т.е. колбэк, в который LiveData будет отправлять данные. В нем только один метод onChanged. В нашем примере туда будет приходить String. Теперь, когда DataController поместит какой-либо String объект в LiveData, мы сразу получим этот объект в Activity, если Activity находится в состоянии STARTED или RESUMED.

Особенности:

Если Activity было не активно во время обновления данных в LiveData, то при возврате в активное состояние, его observer получит последнее актуальное значение данных.

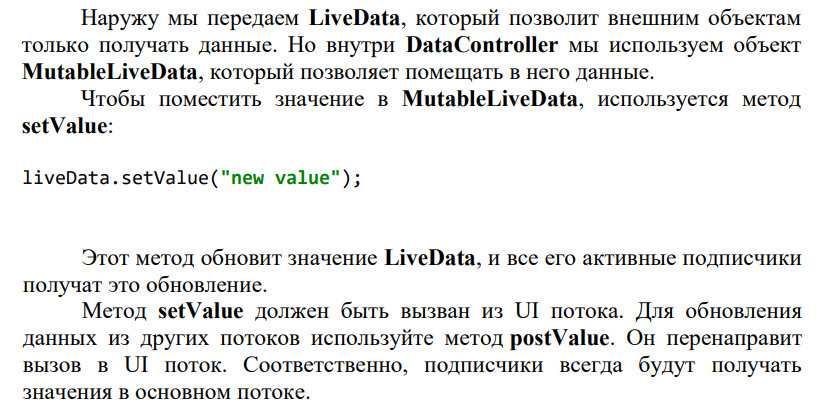
Если Activity будет закрыто, т.е. перейдет в статус DESTROYED, то LiveData автоматически отпишет от себя его observer.

Если Activity в состоянии DESTROYED попробует подписаться, то подписка не будет выполнена.

Если Activity уже подписывало свой observer, и попробует сделать это еще раз, то просто ничего не произойдет.

Вы всегда можете получить последнее значение LiveData с помощью его метода getValue.

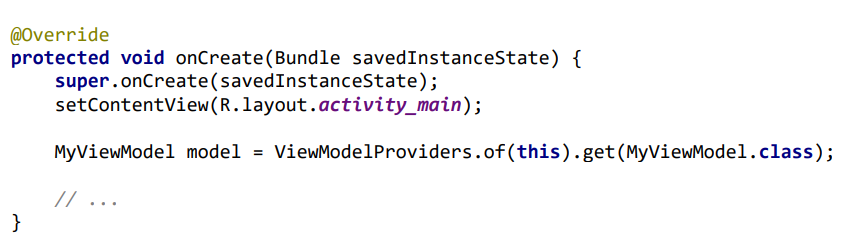
Как видите, подписывать Activity на LiveData - это удобно. Поворот экрана и полное закрытие Activity - все это корректно и удобно обрабатывается автоматически без каких-либо усилий с нашей стороны.



MediatorLiveData дает возможность собирать данные из нескольких LiveData в один. Это удобно если есть несколько источников из которых вы хотите получать данные. Вы объединяете их в один и подписываетесь только на него

* 1. **Архитектура приложения ViewModel. DAO, Entitry**

ViewModel - класс, позволяющий Activity и фрагментам сохранять необходимые им объекты живыми при повороте экрана.

****

В метод ViewModelProviders.of передаем Activity. Тем самым мы получим доступ к провайдеру, который хранит все ViewModel для этого Activity. Методом get запрашиваем у этого провайдера конкретную модель по имени класса - MyViewModel. Если провайдер еще не создавал такой объект ранее, то он его создает и возвращает нам. И пока Activity окончательно не будет закрыто, при всех последующих вызовах метода get мы будем получать этот же самый объект MyViewModel. Соответственно, при поворотах экрана, Activity будет пересоздаваться, а объект MyViewModel будет спокойно себе жить в провайдере. И Activity после пересоздания сможет получить этот объект обратно и продолжить работу, как будто ничего не произошло.

LiveData очень удобно использовать с ViewModel.

Основной метод здесь - это getData. Когда Activity захочет получить данные, оно вызовет именно этот метод. Мы проверяем, создан ли уже MutableLiveData. Если нет, значит этот метод вызывается первый раз. В этом случае создаем MutableLiveData и стартуем асинхронный процесс получения данных методом loadData. Далее возвращаем LiveData.

В этом примере ViewModel нужен, чтобы сохранить процесс получения данных при повороте экрана. А LiveData - для удобного асинхронного получения данных. Т.е. это будет выглядеть так: - Activity вызывает метод модели getData - модель создает MutableLiveData и стартует асинхронный процесс получения данных от репозитория - Activity подписывается на полученный от модели LiveData и ждет данные - происходит поворот экрана - на модели этот поворот никак не сказывается, она спокойно сидит в провайдере и ждет ответ от репозитория - Activity пересоздается, получает ту же самую модель от провайдера, получает тот же самый LiveData от модели и подписывается на него и ждет данные - репозиторий возвращает данные, модель передает их в MutableLiveData - Activity получает данные данные от LiveData

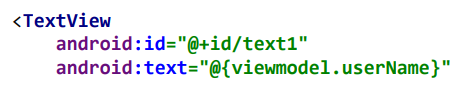
Аннотацией [Entity](https://developer.android.com/reference/android/arch/persistence/room/Entity.html) нам необходимо пометить объект, который мы хотим хранить в базе данных. Для этого создаем класс Employee, который будет представлять собой данные сотрудника: id, имя, зарплата:

В объекте [Dao](https://developer.android.com/reference/android/arch/persistence/room/Dao.html) мы будем описывать методы для работы с базой данных. Нам нужны будут методы для получения списка сотрудников и для добавления/изменения/удаления сотрудников.

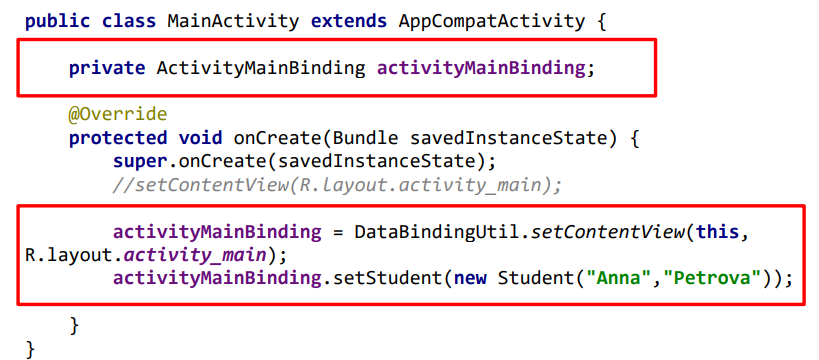
* 1. **Data Binding. Data Binding для RecycleView**

Data binding - это support library, которая позволяет связывать компоненты UI в макетах с источниками данных в приложении, используя декларативный формат, а не программно.

При использовании библиотеки привязки можно назначить текст виджету в файле макета. Это устраняет необходимость в коде Java. Используется синтаксис @{} в выражении присваивания:

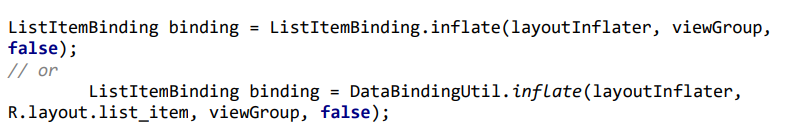
****

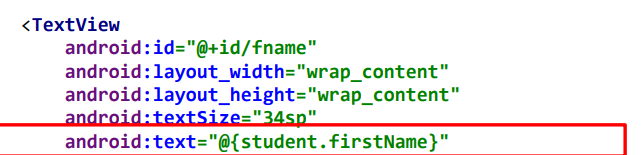
Корневым элементом теперь является layout , а ConstraintLayout сместился внутрь него. Data Binding по созданным xml-файлам генерирует специальные классы, которые и управляют всеми процессами передачи данных. Для каждого файла макета создается класс привязки. По умолчанию имя класса - имя файла макета с добавлением суффикса Binding. Имя файла макета - activity\_main.xml, поэтому соответствующий сгенерированный класс - ActivityMainBinding. Этот класс содержит все привязки от свойств макета (например, пользовательские переменные) к представлениям макета и знает, как назначать значения для выражений привязки.

****

Метод setStudent был сгенерирован в классе биндинга, т.к. была описана переменная student в layout файле. Этим методом мы передаем биндингу объект Student. Биндинг возьмет значения student.firstName и student.secondName и поместит их (методом setText) в соответствующие TextView.

Если нужно использовать элементы привязки данных внутри адаптера Fragment, ListView или RecyclerView, можно использовать метод inflate() классов привязок или класса DataBindingUtil

****

Язык выражений позволяет писать выражения, обрабатывающие события, отправляемые представлениями. Внутри тега можно объявлять переменные с помощью тегов , в качестве атрибутов указываем имя переменной и ее тип. Далее эти переменные можно использовать в разметке и получать из них данные:

Однако надо учитывать что, если в коде будет изменяться объект Student, то данные на экране меняться не будут. Они считались один раз и далее не отслеживаются (при такой реализации). Чтобы экран получил новые данные, надо снова передать биндингу измененный объект или вызывать метод invalidateAll().

Двунаправленная привязка

Пока она не двухсторонняя. Для того чтобы сдаелать two-way 1) Поставить знак = после @

Чтобы указать, что значение может меняться и что эти изменения нужно отслеживать, используется аннотация @Bindable, которая применяется к методу для получения значения какого-то поля. Если после этого выполнить сборку проекта, то мы увидим, что в сгенерированном классе BR (этот класс похож на класс R – класс всех ресурсов, только служит для идентификаторов всех свойств) появится поле. Это поле используется для уведомления о том, что связанное поле класса было изменено. Дальше нужно унаследовать класс Message от BaseObservable. И теперь мы можем использовать метод notifyPropertyChanged и передать ему идентификатор изменившегося свойства.

* 1. **Архитектурный паттерн MVVM для Android приложений**

Ключевой особенностью паттерна MVVM является то, что ни один компонент (Model, View, ViewModel) не знает о другом явно. Эти компоненты взаимодействуют между собой за счет механизма связывания данных (Bindings)

Изменение данных во ViewModel автоматически меняет данные, отображаемые во View. Аналогично, любое событие или изменение данных во View (нажатие на кнопку, ввода текста и другое) изменяет данные во ViewModel. Это позволяет не хранить явные ссылки на View во ViewModel и наоборот, а также держать эти компоненты очень слабо связными, что удобно при тестировании и не только.

Представление

Содержит структурное определение того, что пользователи получат на экранах. Вы можете поместить сюда статическое и динамическое содержимое (анимацию и смену состояний). Для нашего случая в представлении может быть активность или фрагмент. View == Activity (т.е. XML + Java-class).

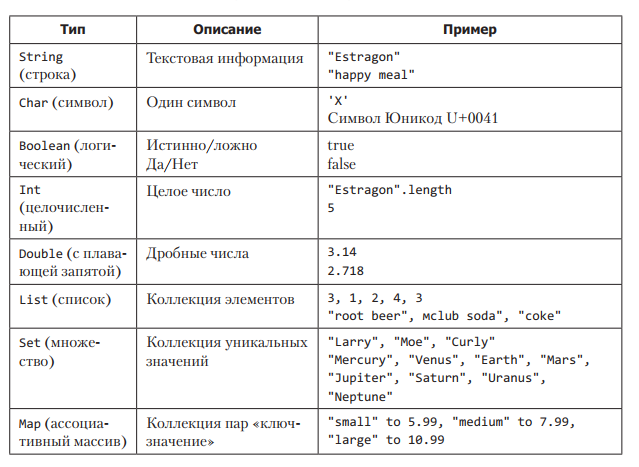
Модель представления

Этот компонент связывает модель и представление. Отвечает за управление ссылками данных и возможных конверсий. Здесь появляется биндинг. В Android мы не беспокоимся об этом, потому что можно напрямую использовать класс AndroidViewModel или ViewModel.

Модель

Это уровень бизнес-данных. В Android, согласно “чистой” архитектуре, модель может содержать базу данных, репозиторий и класс бизнес-логики.

**34. Kotlin. Система типов. Nullability. Преобразования**



Переменная, которую можно изменить, объявляется с помощью ключевого  
слова var. Чтобы объявить переменную, доступную только для чтения, используется ключевое слово val. В разговорной речи переменные, которые могут изменяться, мы называем vars,  
а переменные, доступные только для чтения, — vals. Мы будем соблюдать это правило с этого момента, так как словосочетания «переменная» и «переменная, доступная только для чтения» занимают слишком много места. vars и vals — это все «переменные», и мы будем использовать эти сокращения, чтобы обозначить их в тексте.  
Kotlin поддерживает *автоматическое определение типов*, что позволяет опустить типы для переменных, которым присваиваются значения при объявлении. Так как при объявлении переменной playerName присваивается значение типа String и переменной experiencePoints присваивается значение типа Int, компилятор Kotlin автоматически определяет тип каждой переменной.

**Nullability**Некоторым элементам в Kotlin может быть присвоено значение null, а другим  
нет. Первые — это *nullable* (null возможен), а вторые — это *non-nullable*. Например, переменная в NyetHack с признаком наличия скакуна у игрока должна поддерживать значение null, так как скакун есть не у каждого игрока. Однако переменная с количеством очков здоровья точно не должна принимать значение null. Каждый игрок должен иметь определенное количество очков здоровья, иное просто нелогично. Да, значение переменной может быть равно 0, но 0 — это не null. Null — это отсутствие значения.  
Откройте NyetHack и создайте новый файл с именем Tavern.kt. Добавьте функцию **main**, с которой будет начинаться выполнение кода. NullabilityПрежде чем открыть таверну для клиентов, проведем эксперимент. Добавьте  
в **main** переменную var и присвойте ей значение, а затем присвойте ей значение null.  
Kotlin не позволил присвоить значение null переменной signatureDrink, потому  
что эта переменная имеет тип, не поддерживающий null (строка, String). Тип  
String не поддерживает значение null. Текущее объявление гарантирует, что  
signatureDrink будет хранить строку, а не null.  
  
Java и многие другие языки поддерживают такую инструкцию на псевдокоде:  
«Эй, несуществующая строка, выполни конкатенацию строк!» В этих языках  
переменная может иметь значение null (кроме примитивов, которые Kotlin  
не поддерживает). В таких языках, которые допускают null для любого типа,  
ошибка **NullPointerException** является частой причиной сбоя приложения.  
Kotlin принимает противоположную позицию в обращении с null. Если не указано иное, то переменной нельзя присвоить значение null. Это предотвращает проблемы «Эй, несуществующий элемент, сделай что нибудь» еще во время компиляции, а не в момент выполнения программы

Однако все не так плохо, потому что в Kotlin есть функции для преобразования строк в другие типы, даже в числа. Вот самые широко используемые функции преобразования:  
 toFloat  
 toDouble  
 toDoubleOrNull  
 toIntOrNull  
 toLong  
 toBigDecimal

Для начала вычисляем общую сумму totalPurse и выводим ее. Обратите внимание, что делитель, используемый при переводе playerSilver для otalPurse,  
содержит дробную часть —100.0, а не просто 100.  
Если разделить playerSilver, значение типа Int, на число 100, которое тоже  
имеет тип Int, Kotlin вернет не 0.10 типа Double. Вместо этого вы получите  
другое значение Int — 0, потеряв такую важную дробную часть

**35. Kotlin. Итерации, циклы и интервалы. Исключения.**

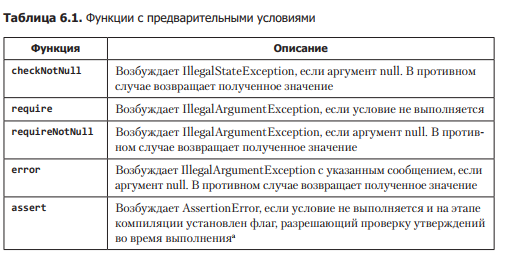
ИнтервалыВсе условия в выражении if/else для healthStatus, по сути, проверяют целочисленное значение healthPoints. В некоторых используется оператор сравнения для проверки равенства healthPoints какому-то значению, в других  
используется несколько операторов сравнения, чтобы проверить, попадает ли  
значение healthPoints в интервал между двумя числами. Для второго случая  
есть альтернатива получше: Kotlin предусматривает интервалы для представления линейного набора значений.  
Интервал определяется оператором .., например 1..5. Интервал включает все  
значения, начиная с находящегося слева от оператора .. и заканчивая находящимся справа. Например, 1..5 включает числа 1, 2, 3, 4 и 5. Интервалы могут  
представлять последовательности символов.  
Для проверки попадания заданного числа в интервал можно использовать  
ключевое слово in (внутри).

Один из способов выполнить итерацию со списком — использовать цикл for.  
Его логика такова: «Для каждого элемента в списке сделай то-то». Вы должны  
определить имя элемента, а компилятор Kotlin автоматически определит его тип.  
В языке Kotlin все циклы for опираются на итерации. В языках Java и С# им  
эквивалентны циклы foreach.  
Тех, кто знаком с Java, может удивить, что обычное для Java выражение (int  
i =0; i < 10; i++) { ... } невозможно в Kotlin. Вместо этого цикл for записывается так: for(i in 1..10){ ... }. Однако на уровне байт-кода компилятор  
оптимизирует цикл for языка Kotlin до Java-версии, если возможно, чтобы  
улучшить производительность.  
Обратите внимание на ключевое слово in:  
for (patron in patronList) { ... }  
in определяет объект для обхода в цикле for.  
Цикл for прост и легкочитаем, но если вы предпочитаете более функциональный стиль, используйте функцию **forEach**.  
Функция **forEach** обходит каждый элемент в списке — последовательно от начала до конца — и передает каждый элемент анонимной функции в аргументе.  
Цикл for и функция **forEach** обрабатывают индексы неявно. Если вам потребуется получить индекс каждого элемента в списке, используйте **forEachIndexed**.  
  
Функции **forEach** и **forEachIndexed** доступны и для некоторых других типов  
в Kotlin. Эта категория типов называется Iterable (итерируемые) и включает List, Set, Map, IntRange (диапазоны типа 0…9, которые вы видели в главе 3)  
и другие типы коллекций. Итерируемые типы поддерживают итерацию — другими словами, они позволяют выполнить обход хранимых элементов и выполнить некоторые действия с каждым из них

Kotlin, подобно другим языкам программирования, позволяет вручную послать  
сигнал о возникновении исключения. Это достигается применением оператора  
throw и называется *возбуждением* исключения. Есть много разных исключений,  
кроме Null Pointer Exception.

Вы узнали, как использовать оператор throw , чтобы сообщить об исключительной ситуации. Возбуждение исключения IllegalStateException позволяет  
сообщить, что возникло недопустимое состояние, и добавить строку с дополнительной информацией.  
Чтобы добавить больше деталей в исключение, можно создать пользовательское исключение для конкретной проблемы. Для этого нужно объявить  
новый *класс,* наследующий некоторое другое исключение. Классы позволяют  
определять «предметы» в программе — монстров, оружие, еду, инструменты  
и т. д. Больше о классах мы узнаем в главе 12, поэтому пока в синтаксис можете не вникать.  
Исключения деструктивны, какими и должны быть, — они представляют  
ошибочное состояние, которое надо исправить. Kotlin позволяет обрабатывать  
исключения, поместив код, в котором они могут возникнуть, в оператор try/  
catch. Синтаксис try/catch похож на синтаксис if/else.

В стандартной библиотеке Kotlin есть пять функций проверки условий; это  
разнообразие отличает их от других вариантов проверки на null. Пять функций  
проверки условий перечислены в табл. 6.1.  
**require** (требуется) весьма полезная проверка. Функции могут использовать  
**require**, чтобы определить граничные значения для передаваемых им аргументов



**36. Kotlin. Функции. Cвойства и функции верхнего уровня. Cвойства и функции расширения. Локальные функции. Функции высшего порядка**

Функция — это фрагмент кода, который выполняет определенную задачу  
и может использоваться повторно. Функции — это очень важная часть программирования. Более того, программа, по сути, является последовательностью функций, взаимосвязанных для выполнения более сложной задачи

fun имя\_функции (параметры) : возвращаемый\_тип{

    выполняемые инструкции

}

Функции можно определять в файле вне других функций или классов, сами по себе, как например, определяется функция main. Такие функции еще называют функциями верхнего уровня (top-level functions).

Здесь кроме главной функции main также определена функция hello, которая не принимает никаких параметров и ничего не возвращает. Она просто выводит строку на консоль.

Функция hello (и любая другая определенная функция, кроме main) сама по себе не выполняется. Чтобы ее выполнить, ее надо вызвать. Для вызова функции указывается ее имя (в данном случае "hello"), после которого идут пустые скобки.

Таким образом, если необходимо в разных частях программы выполнить одни и те же действия, то можно эти действия вынести в функцию, и затем вызывать эту функцию.

fun displayUser(name: String, age: Int = 18, position: String="unemployed"){

    println("Name: $name   Age: $age  Position: $position")

}

По умолчанию все параметры функции равносильны val-переменным, поэтому их значение нельзя изменить. Например, в случае следующей функции при компиляции мы Однако если параметр предствляет какой-то сложный объект, то можно изменять отдельные значения в этом объекте. Например, возьмем функцию, которая в качестве параметра принимает массив: получим ошибку:

Кроме функций-расширений Kotlin позволяет создавать свойства-расширения. Они позволяют добавлять в классы функции, к котором можно обращаться как к свойствам. Иногда подобные свойства удобны из-за краткого синтаксиса.

Для сравнения создадим функцию-расширение и свойство-расширение.

// функция-расширение

fun String.lastChar(): Char = get(length - 1)

// свойство-расширение

val String.lastChar: Char

get() = get(length - 1)

У класса **StringBuilder** для создания аналогичного свойства используем **var**, потому что содержимое экземпляра **StringBuilder** может меняться.

var StringBuilder.lastChar: Char

get() = get(length - 1)

set(value: Char) {

this.setCharAt(length - 1, value)

}

Вызываем в коде.

// получить последний символ строки

println("Cat".lastChar)

// заменить последний символ строки

val sb = StringBuilder("Cat?")

sb.lastChar = '!'

println(sb)

Функции расширения (extension function) позволяют добавить функционал к уже определенным типам. При этом типы могут быть определены где-то в другом месте, например, в стандартной библиотеке.

Функция расширения определяется следующим образом:

|  |  |
| --- | --- |
| 1  2  3 | fun тип.имя\_функции(параметры) : возвращаемый\_тип{      тело функции  } |

По большому счету определение аналогично определению обычной функции за тем исключением, что после слова **fun** идет название типа, для которого определяется функция, и через точку название функции.

**Локальные функции**

Одни функции могут быть определены внутри других функций. Внутренние или вложенные функции еще называют локальными.

Локальные функции могут определять действия, которые используются только в рамках какой-то конкретной функции и нигде больше не применяются.

fun compareAge(age1: Int, age2: Int){

    fun ageIsValid(age: Int): Boolean{

        return age > 0 && age < 111

    }

Функции высокого порядка (high order function) - это функции, которые либо принимают функцию в качестве параметра, либо возвращают функцию, либо и то, и другое.

**Функция как параметр функции**

Чтобы функция могла принимать другую функцию через параметр, этот параметр должен представлять тип функции:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | fun main() {        displayMessage(::morning)      displayMessage(::evening)  }  fun displayMessage(mes: () -> Unit){      mes()  }  fun morning(){      println("Good Morning")  }  fun evening(){      println("Good Evening")  } |

**Возвращение функции из функции**

В более редких случаях может потребоваться возвратить функцию из другой функции. В этом случае для функции в качестве возвращаемого типа устанавливается тип другой функции. А в теле функции возвращается лямбда выражение. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | fun main() {      val action1 = selectAction(1)      println(action1(8,5))    // 13        val action2 = selectAction(2)      println(action2(8,5))    // 3  }  fun selectAction(key: Int): (Int, Int) -> Int{      // определение возвращаемого результата      when(key){          1 -> return ::sum          2 -> return ::subtract          3 -> return ::multiply          else -> return ::empty      }  }  fun empty (a: Int, b: Int): Int{      return 0  }  fun sum(a: Int, b: Int): Int{      return a + b  }  fun subtract(a: Int, b: Int): Int{      return a - b  }  fun multiply(a: Int, b: Int): Int{      return a \* b  } |

**37. Kotlin. Стандартные функции. Лямбда-выражения**

**apply**Первая на нашем пути — функция **apply**. **apply** можно считать функцией настройки: она позволяет вызвать несколько функций для объекта-получателя  
и настроить его для дальнейшего использования. После выполнения указанного  
лямбда-выражения **apply** возвращает настроенный объект-получатель.  
**apply** можно использовать, чтобы уменьшить количество повторений при подготовке объекта к использованию.

**let**Еще одна часто используемая стандартная функция — это **let**, с которой вы  
сталкивались в главе 6. **let** определяет переменную в область видимости заданной лямбды и позволяет использовать ключевое слово it, с которым вы  
познакомились в главе 5, для ссылки на нее. Вот пример **let**, который возводит  
в квадрат первое число в списке:  
val firstItemSquared = listOf(1,2,3).first().let {  
it \* it  
}

**run**Следующая стандартная функция в списке — это **run**. Функция **run** похожа на  
**apply**, точно так же ограничивая относительную область видимости, но не возвращает объект-приемник.

**with  
with** — это разновидность **run**. Она ведет себя похожим образом, но использует  
другие соглашения вызова. В отличие от стандартных функций, рассмотренных ранее, **with** требует, чтобы объект-приемник передавался ей в первом  
аргументе, а не как субъект вызова, как это принято в других стандартных  
функциях:  
val nameTooLong = with("Polarcubis, Supreme Master of NyetHack") {  
length >= 20  
}

**also**Функция **also** похожа на функцию **let**. Как и **let**, **also** передает объект-приемник как аргумент в лямбду. Но есть одно большое различие между **let** и **also**:  
вторая возвращает объект-приемник, а не результат лямбды.  
Это делает **also** особенно полезной для добавления различных побочных  
эффектов. Пример ниже дважды вызывает **also** для выполнения двух разных  
операций: первая выводит имя файла, а вторая записывает содержимое файла  
в переменную fileContents.  
var fileContents: List<String>  
File("file.txt")  
.also {  
print(it.name)  
}.also {  
fileContents = it.readLines()  
}  
}

**takeIf**Последняя стандартная функция — это **takeIf**. **takeIf** работает немного иначе,  
чем другие стандартные функции: она вычисляет условие, или *предикат*, заданное в лямбде, которое возвращает истинное или ложное значение. Если  
условие истинно, **takeIf** вернет объект-приемник. Если условие ложно, она  
вернет null.

**takeUnless**Выше мы сказали, что закончили обзор, но есть еще одна функция, дополняющая функцию **takeIf**, о которой стоит упомянуть, чтобы предостеречь от ее  
использования: **takeUnless**. Функция **takeUnless** действует так же, как **takeIf**,  
но возвращает объект-приемник, если условие *ложно*. Следующий код читает  
файл, если он не скрытый (и возвращает null, если скрытый):  
val fileContents = File("myfile.txt").takeUnless { it.isHidden }?.readText()  
Мы рекомендуем ограничить использование **takeUnless**, особенно для проверки  
сложных условий, потому что понимание программы у читающих ее людей  
займет много времени. Сравните «понятность» этих двух фраз:  
 **takeIf** — возвращает значение, если условие истинно;  
 **takeUnless** — возвращает значение, если условие не истинно.

**Анонимные функции**Анонимные функции — важная часть языка Kotlin. Одно из их применений —  
адаптация функций из стандартной библиотеки Kotlin для решения конкретных  
задач. Анонимная функция позволит вам добавить правила для функций из  
стандартной библиотеки и настроить их поведение. Рассмотрим пример.  
Одна из многих функций стандартной библиотеки — это **count**. Вызванная со  
строкой функция **count** возвращает количество символов в строке. Код ниже  
считает количество букв в строке — "Mississippi":  
val numLetters = "Mississippi".count()  
print(numLetters)  
// Выведет 11  
Анонимные функции **103**(Здесь был использован *точечный синтаксис* вызова функции **count**. Этот  
синтаксис используется всегда, когда вызываемая функция является частью  
определения типа.)  
А что если нужно посчитать только определенные символы в слове "Mississipi",  
например только 's'?  
Для решения такой задачи стандартная библиотека Kotlin позволяет добавить  
правила к функции **count**, которые определят, какие символы должны быть  
подсчитаны. Вы задаете эти правила, передавая функции **count** аргумент с анонимной функцией. Это выглядит так:  
val numLetters = "Mississippi".count({ letter ->  
letter == 's'  
})  
print(numLetters)

**38. Kotlin. Коллекции и их классификация. Создание и преобразование коллекций. Деструктуризация. Функциональный API для работы с коллекциями.**

Коллекции в Kotlin используют множество различных интерфейсов: **Iterable**, **Collection**, [Set](http://developer.alexanderklimov.ru/android/kotlin/set.php), **List**, **MutableIterable**, **MutableCollection** и др.

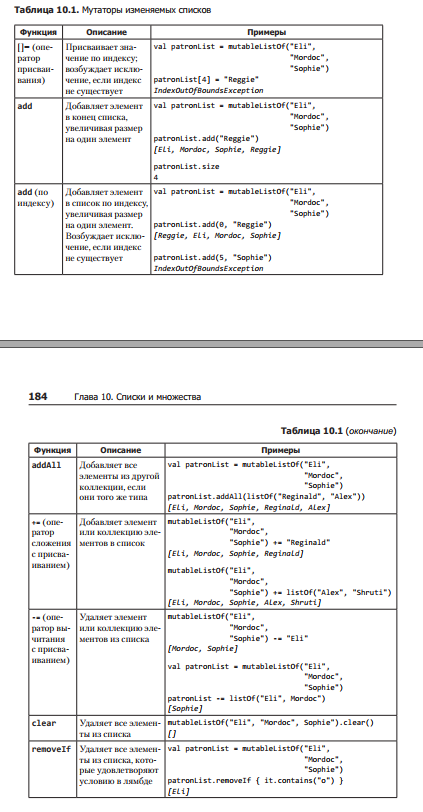
В Kotlin нет собственных коллекций, только Java-коллекции. Но при этом в Kotlin они обладают более широкими возможностями, используя расширения. Например, вы можете узнать последний элемент списка или найти максимальное значение в коллекции чисел.

Важная особенность - в Kotlin интерфейсы явно разделены на две группы - изменяемые и неизменяемые. Старайтесь всегда использовать неизменяемые коллекции с доступом для чтения. Если вам нужно изменять коллекцию, то тогда выбирайте другой вариант. Здесь не надо путать с **var** и **val**. Если вы создадите список для чтения и привяжете его к переменной **var**, список всё равно нельзя будет изменить после создания.

В NyetHack вы создали изменяемое множество уникальных имен посетителей,  
передавая элементы из списка в множество один за другим. Также преобразовать  
список в множество и обратно можно с помощью функций **toSet** и **toList** (или  
их изменяемых вариантов: **toMutableSet** и **toMutableList**). Распространенный  
трюк — вызвать **toSet**, чтобы отбросить неуникальные элементы в списке.   
  
Чтобы получить возможность доступа по индексу после удаления дубликатов,  
снова преобразуйте множество в список.  
  
Необходимость удалять дубликаты и использовать доступ по индексу очень  
распространена, поэтому Kotlin предоставляет функцию с именем **distinct**,  
которая внутренне вызывает **toSet** и **toList**.  
  
Множества полезны для представления наборов данных с уникальными элементами. В следующей главе мы закончим обзор типов коллекций в языке Kotlin,  
познакомившись с ассоциативными массивами.

*Коллекции* позволяют легко работать с группами  
значений и передавать их как аргументы в функции.  
В следующих двух главах мы рассмотрим самые часто применяемые типы  
коллекций: List (список), Set (множество) и Map (ассоциативный массив).  
Как многие другие типы переменных, изученные в главе 2, списки, множества  
и ассоциативные массивы бывают двух видов: изменяемые и доступные только  
для чтения. В этой главе мы рассмотрим списки и множества

До этого момента вы создавали переменные разных типов, просто объявляя их.  
Но, чтобы получить коллекцию, нужно выполнить два шага: создать коллекцию (в примере это список посетителей) и добавить в нее содержимое (имена  
посетителей). Kotlin предоставляет функции, такие как **listOf**, которые могут  
делать это одновременно.  
Теперь, когда у вас есть список, рассмотрим тип List подробнее.  
Хотя механизм автоматического определения типов распознает списки, мы  
включили информацию о типе — val patronList:List<String>, чтобы сделать его  
видимым для обсуждения. Обратите внимание на угловые скобки в List<String>.  
<String>— это *параметр типа*, и он сообщает компилятору тип значений, которые  
будут содержаться в списке, — в нашем случае это String. Изменение параметра  
типа изменяет тип значений, которые компилятор разрешит хранить в списке.  
Если вы попытаетесь записать целое число в patronList, компилятор этого не  
допустит. Попробуйте добавить число в объявленный список.  
  
...  
var patronList: List<String> = listOf("Eli", "Mordoc", "Sophie", **1)**...  
IntelliJ предупредит, что целочисленное значение не соответствует ожидаемому  
типу String. Параметр типа необходимо использовать с List, потому что сам  
тип List является *обобщенным типом*. Это означает, что список может хранить  
данные любого типа, включая текстовые данные, такие как строки (как в случае  
с patronList) или символы, числовые значения, такие как целые и дробные числа, и даже данные новых типов, определяемых пользователем. (Об обобщенных  
типах подробнее в главе 17.)



**Деструктуризация**Список также предлагает возможность деструктуризации до пяти первых  
элементов. Деструктуризация, как вы видели в главе 7, позволяет объявить  
несколько переменных и присвоить им значения в одном выражении. Вы используете прием деструктуризации для разделения заказа на составляющие:  
val (type, name, price) = menuData.split(',')  
Это объявление присваивает первые три элемента списка, возвращаемого  
функцией **split**, строковым значениям type, name и price.  
Кстати говоря, можно выборочно деструктурировать элементы из списка, используя символ \_ для пропуска нежелательных элементов. Например, трактирщик может захотеть выдать медали лучшим жонглерам мечами в королевстве,  
но ему не хватает серебряной медали. Если вы хотите деструктурировать только  
первое и третье значения из списка посетителей, это можно сделать так:  
val (goldMedal, \_, bronzeMedal) = patronList

**Множества**Списки, как вы видели, могут хранить повторяющиеся элементы (и упорядочиваться, поэтому повторяющиеся элементы легко выявить по их позициям).  
Но иногда нужна коллекция, гарантирующая уникальность своих элементов.  
Для этого можно использовать множества.  
Множества во многом похожи на списки. Они используют те же итерационные  
функции и могут быть изменяемыми или доступными только для чтения.  
Но есть два важных отличия между списками и множествами: множества гарантируют уникальность элементов и не поддерживают возможность их изменения  
по индексам, потому что не предусматривают какого-то строго определенного  
порядка размещения. (Тем не менее вы все равно можете читать элементы по  
конкретному индексу, к чему мы вскоре еще вернемся).  
**Создание множества**Список создается с помощью функции **listOf**. Создать множество можно с помощью функции **setOf**. Попробуйте создать множество в REPL.  
**val planets = setOf("Mercury", "Venus", "Earth")  
planets**["Mercury", "Venus", "Earth"]  
Если попробовать добавить в множество одну и ту же планету дважды, в нем  
останется только одна.  
**Листинг 10.21.** Пробуем создать множество с дубликатом (REPL)  
**val planets = setOf("Mercury", "Venus", "Earth", "Earth")  
planets**["Mercury", "Venus", "Earth"]  
Второе значение "Earth" было выброшено из множества.  
Так же как список, множество позволяет проверить присутствие конкретного  
элемента с помощью contains или containsAll. Попробуйте функцию contains  
в REPL.  
**Листинг 10.22.** Проверка планет (REPL)  
**planets.contains("Earth")**true  
**planets.contains("Pluto")**false  
Множества **193**Множество не индексирует свое содержимое — это означает, что оно не поддерживает встроенный оператор [] для доступа к элементам по индексу. Тем не  
менее можно запросить элемент по определенному индексу с помощью функции, которая использует итерации для решения своей задачи.

Преобразование коллекций

В NyetHack вы создали изменяемое множество уникальных имен посетителей,  
передавая элементы из списка в множество один за другим. Также преобразовать  
список в множество и обратно можно с помощью функций **toSet** и **toList** (или  
их изменяемых вариантов: **toMutableSet** и **toMutableList**). Распространенный  
трюк — вызвать **toSet**, чтобы отбросить неуникальные элементы в списке.   
**listOf("Eli Baggins", "Eli Baggins", "Eli Ironfoot").toSet()**[Eli Baggins, Eli Ironfoot]  
Чтобы получить возможность доступа по индексу после удаления дубликатов,  
снова преобразуйте множество в список.  
**Листинг 10.28.** Преобразование множества в список (REPL)  
**val patrons = listOf("Eli Baggins", "Eli Baggins", "Eli Ironfoot")  
.toSet()  
.toList()**[Eli Baggins, Eli Ironfoot]  
**patrons[0]**Eli Baggins  
Необходимость удалять дубликаты и использовать доступ по индексу очень  
распространена, поэтому Kotlin предоставляет функцию с именем **distinct**,  
которая внутренне вызывает **toSet** и **toList**.  
  
**val patrons = listOf("Eli Baggins", "Eli Baggins", "Eli Ironfoot").distinct()**[Eli Baggins, Eli Ironfoot]  
**patrons[0]**Eli Baggins  
Множества полезны для представления наборов данных с уникальными элементами. В следующей главе мы закончим обзор типов коллекций в языке Kotlin,  
познакомившись с ассоциативными массивами.

**39. Kotlin. Интерфейсы (реализация по умолчанию). Классы. Наследование. Модификаторы видимости.**

**Реализация по умолчанию**Мы уже неоднократно говорили: интерфейсы определяют, *что* надо реализовать, а не *как*. Тем не менее есть возможность определить реализацию по  
умолчанию для методов чтения свойств и функций в интерфейсе. Классы,  
реализующие интерфейс, могут использовать реализацию по умолчанию или  
определить свою.  
Добавьте метод чтения по умолчанию для damageRoll в **Fightable**. Он должен  
возвращать сумму очков, выпавших на всех костях, чтобы определить величину  
урона, нанесенного противнику в одном раунде битвы.  
**Листинг 16.4.** Объявление реализации геттера по умолчанию (Creature.kt)  
interface Fightable {  
var healthPoints: Int  
val diceCount: Int  
val diceSides: Int  
val damageRoll: Int  
**get() = (0 until diceCount).map {  
Random().nextInt(diceSides + 1)  
}.sum()**fun attack(opponent: Fightable): Int  
}  
Теперь, когда damageRoll имеет метод чтения по умолчанию, любой класс,  
реализующий интерфейс **Fightable**, может отказаться от определения своей  
реализации свойства damageRoll и использовать значение, возвращаемое реализацией по умолчанию.  
Не каждому свойству или функции нужна уникальная реализация в каждом  
классе, поэтому определение реализации по умолчанию — это хороший способ  
сократить повторы в вашем коде.

**Доступность и инкапсуляция**Добавление поведения в класс в виде функций класса (и данных в виде свойств  
класса, как мы увидим далее) создает описание, чем класс может быть и что  
может делать, и это описание доступно всем, у кого есть экземпляр класса.  
По умолчанию любая функция или свойство без модификатора видимости  
будет доступна всем — то есть из любого файла или функции в программе. Так  
как вы не указали модификатор видимости для **castFireball**, она может быть  
вызвана откуда угодно.  
В некоторых случаях, как с **castFireball**, вы захотите, чтобы другие части кода  
имели доступ к свойствам класса или вызывали функции класса. Но у вас также могут быть другие функции класса или свойства, которые не должны быть  
доступны повсюду.  
С ростом количества классов в программе растет и сложность кода. Сокрытие  
деталей реализации, которые не должны быть доступны другим частям кода,  
Доступность и инкапсуляция **219**помогает сохранить логику кода ясной и лаконичной. И в этом вам поможет  
ограничение области видимости.  
В то время как функцию класса public можно вызвать из любого места программы,  
функция класса private доступна только в классе, в котором объявлена. Эта идея  
ограничения видимости некоторых свойств и/или функций класса в объектноориентированном программировании называется *инкапсуляцией*. Согласно этой  
идее класс должен выборочно предоставлять функции и свойства другим объектам  
для взаимодействия с ним. Все, что не задействуется, включая детали реализации  
функций и свойств, должно быть недоступно.  
Например, код в Game.kt, вызывающий **castFireball**, не интересует, как реализована эта функция. Для него важно только получить бокал напитка. Поэтому  
пока функция доступна, детали реализации не должны быть важны для вызывающего.  
Более того, это даже может быть опасно, если код Game.kt сможет менять значения, на которые опирается **castFireball**, то есть на количество создаваемых  
бокалов с напитком или на свойства напитка.  
Проще говоря, создавая классы, оставляйте доступным только то, что вам надо.  
Таблица 12.1 содержит список модификаторов доступа.  
**Таблица 12.1.** Модификаторы доступа

|  |  |
| --- | --- |
| **Модификаторы** | **Описание** |
| public (по умол чанию) | Функция или свойство будут доступны вне класса. По умолчанию функции и свойства без модификатора видимости получают моди фикатор public |
| private | Функция или свойство будут доступны только внутри класса |
| protected | Функция или свойство будут доступны только внутри класса или подкласса |
| internal | Функция или свойство будут доступны внутри модуля |

Мы обсудим ключевое слово protected в главе 14.  
Если вы знакомы с Java, обратите внимание, что в Kotlin отсутствует уровень  
видимости, ограниченный рамками пакета. Причину мы объясним в разделе  
«Для любопытных: ограничение видимости рамками пакета» в конце главы.  
*Наследование* — это принцип объектно-ориентированного программирования,  
помогающий определить иерархические отношения между типами. В этой главе  
мы используем наследование, чтобы организовать совместное использование  
данных и поведения родственными классами.

**ПРО НАСЛЕДОВАНИЕ ПРОЩЕ ОДИН РАЗ В КНИГЕ ПОЧИТАТЬ А ТО ТАМ СЛИШКОМ ДОХУЯ И НИЧЕГО НОВОГО**

**40. Kotlin. Конструкторы классов. Классы данных. Блоки инициализации. Поздняя и отложенная инициализация.**

Для создания объекта необходимо вызвать конструктор класса. По умолчанию компилятор создает конструктор, который не принимает параметров и который мы можем использовать. Но также мы можем определять свои собственные конструкторы. Для определения конструкторов применяется ключевое слово **constructor**.

Классы в Kotlin могут иметь один **первичный конструктор** (primary constructor) и один или несколько **вторичных конструкторов** (secondary constructor).

**Первичный конструктор**

Первичный конструктор является частью заголовка класса и определяется сразу после имени класса:

|  |  |
| --- | --- |
| 1  2  3 | class Person constructor(\_name: String){    } |

Конструкторы, как и обычные функции, могут иметь параметры. Так, в данном случае конструктор имеет параметр \_name, который представляет тип String. Через параметры конструктора мы можем передать извне данные и использовать их для инициализации объекта. При этом первичный конструктор в отличие от функций не определяет никаких действий, он только может принимать данные извне через параметры.

Если первичный конструктор не имеет никаких аннотаций или модификаторов доступа, как в данном случае, то ключевое слово constructor можно опустить:

|  |  |
| --- | --- |
| 1  2  3 | class Person(\_name: String){    } |

**Инициализатор**

Что делать с полученными через конструктор данными? Мы их можем использовать для инициализации свойств класса. Для этого применяются блоки инициализаторов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | class Person(\_name: String){      val name: String      init{          name = \_name      }  } |

В классе Person определено свойство name, которое хранит имя человека. Чтобы передать эту свойству значение параметра \_name из первичного конструктора, применяется блок инициализатора. Блок инициализатора определяется после ключевого слова **init**.

Цель инициализатора состоит в инициализации объекта при его создании. Стоит отметить, что здесь свойству name не задается начальное значение, потому это свойство в любом случае будет инициализировано в блоке инициализатора, и при создании объекта оно в любом случае получит значение.

**Вторичные конструкторы**

Класс также может определять вторичные конструкторы. Они применяются в основном, чтобы определить дополнительные параметры, через которые можно передавать данные для инициализации объекта.

Вторичные конструкторы определяются в теле класса. Если для класса определен первичный конструктор, то вторичный конструктор должен вызывать первичный с помощью ключевого слова **this**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class Person(\_name: String){      val name: String = \_name      var age: Int = 0        constructor(\_name: String, \_age: Int) : this(\_name){          age = \_age      }  } |

Здесь в классе Person определен первичный конструктор, который принимает значение для установки свойства name:

|  |  |
| --- | --- |
| 1 | class Person(\_name: String) |

И также добавлен вторичный конструктор. Он принимает два параметра: \_name и \_age. С помощью ключевого слова this вызывается первичный конструктор, поэтому через этот вызов необходимо передать значения для параметров первичного конструктора. В частности, в первичный конструктор передается значение параметра \_name. В самом вторичном конструкторе устанавливается значение свойства age.

|  |  |
| --- | --- |
| 1  2  3 | constructor(\_name: String, \_age: Int) : this(\_name){      age = \_age  } |

Таким образом, при вызове вторичного конструктора вначале вызывается первичный конструктор, срабатывает блок инициализатора, который устанавливает свойство name. Затем выполняются собственно действия вторичного конструктора, который устанавливает свойство age.

Классы данных

Нередко мы создаём классы, единственным назначением которых является хранение данных. Функционал и некоторые служебные функции таких классов зависят от самих данных, которые в них хранятся. В Kotlin они называются *классами данных* и помечены data.

**data** **class** User(**val** name: String, **val** age: Int)

Компилятор автоматически формирует следующие члены данного класса из свойств, объявленных в основном конструкторе:

* пару функций equals()/hashCode(),
* функцию toString() в форме "User(name=John, age=42)",
* компонентные функции [componentN()](https://kotlinlang.ru/docs/reference/destructuring-declarations.html), которые соответствуют свойствам, в соответствии с порядком их объявления,
* функцию copy() (см. ниже).

Для обеспечения согласованности и осмысленного поведения сгенерированного кода классы данных должны удовлетворять следующим требованиям:

* Основной конструктор должен иметь как минимум один параметр;
* Все параметры основного конструктора должны быть отмечены, как val или var;
* Классы данных не могут быть абстрактными, open, sealed или inner;

Дополнительно, генерация членов классов данных при наследовании подчиняется следующим правилам:

* Если существуют явные реализации equals(), hashCode() или toString() в теле класса данных или final реализации в суперклассе, то эти функции не генерируются, а используются существующие реализации;
* Если суперкласс включает функции componentN(), которые являются открытыми и возвращают совместимые типы, соответствующие компонентные функции создаются для класса данных и переопределяют функции суперкласса. Если функции суперкласса не могут быть переопределены из-за несовместимости сигнатур или потому что они final, выдаётся сообщение об ошибке;
* Предоставление явных реализаций для функций componentN() и copy() не допускается.

Классы данных могут расширять другие классы (см. примеры в статье [Изолированные классы](https://kotlinlang.ru/docs/reference/sealed-classes.html)).

Для того, чтобы у сгенерированного в JVM класса был конструктор без параметров, значения всех свойств должны быть заданы по умолчанию (см. [Конструкторы](https://kotlinlang.ru/docs/reference/classes.html#constructors)).

**data** **class** User(**val** name: String = **""**, **val** age: Int = 0)

**lateinit**

Разработчики Kotlin крайне серьёзно относятся к проверкам на null. Поэтому, как правило, свойства, которые по логике программы должны хранить ненулевые значения инициализируются в конструкторе.

Тем не менее бывают ситуации, когда такой подход не особо удобен. Например, если вы хотите инициализировать свойства через внедрение зависимостей. Kotlin предусматривает такую возможность и предлагает использовать отложенную (позднюю) инициализацию. Осуществляется это с помощью модификатора lateinit.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **class** **Work**  **class** **Person** {  **lateinit** **var** work: **Work**  **fun** **init**() {  work = **Work**()  }  } |

Модификатор lateinit говорит о том, что данная переменная будет инициализирована позже. При этом инициализировать свойство можно из любого места, откуда она видна.

Правила использования модификатора lateinit:

используется только совместно с ключевым словом var;

свойство может быть объявлено только внутри тела класса (не в основном конструкторе);

тип свойства не может быть нулевым и примитивным;

у свойства не должно быть пользовательских геттеров и сеттеров;

с версии Kotlin 1.2 можно применять к свойствам верхнего уровня и локальным переменным.

**Lazy**

Помимо отложенной инициализации в Kotlin существует ленивая инициализация свойств. Такая инициализация осуществляется с помощью функции lazy(), которая принимает лямбду, а возвращает экземпляр класса Lazy<T>. Данный объект реализует ленивое вычисление значения свойства: при первом обращении к свойству метод get() запускает лямбда-выражение (переданное lazy() в качестве аргумента) и запоминает полученное значение, а последующие вызовы просто возвращают запомненное значение.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | **class** **Person** {  **val** work: **String** **by** **lazy** {  **println**("Start")  "End"  }  **fun** **main**(args: **Array**<**String**>) {  **println**(work)  **println**(work)  }  *// Код выведет:*  *// Start*  *// End*  *// End*  } |

Ленивая инициализация может быть использована только совместно с ключевым словом val.

Свойство, инициализированное подобным образом, называется делегированным свойством. Потому что мы делегировали вычисление значения классу-делегату Lazy<T>. Данный класс является частью стандартной библиотеки Kotlin и именно в нем реализован get-метод вычисляющий и возвращающий значение.

По умолчанию вычисление ленивых свойств синхронизировано: значение вычисляется только в одном потоке, а все остальные потоки могут видеть одно и то же значение. Однако способом вычисления можно управлять. Для этого функции lazy() нужно передать один из параметров:

LazyThreadSafetyMode.SYNCHRONIZED - режим по умолчанию, потокобезопасный.

LazyThreadSafetyMode.PUBLICATION - вычисление будет происходить в нескольких потоках, но вернётся то значение, которое будет вычислено первым.

LazyThreadSafetyMode.NONE - использовать с осторожностью, не является потокобезопасным. Нужно быть уверенным, что вычисление будет происходить в одном потоке.

**41. Kotlin. Понятие object. Назначение и определение. Объекты-компаньоны.**

**Ключевое слово object**В главе 13 вы научились конструировать классы. Конструктор класса возвращает экземпляр класса, и можно вызывать конструктор любое количество раз,  
чтобы создать любое количество экземпляров.  
Например, в NyetHack может быть любое количество игроков, поскольку конструктор **Player** можно вызвать столько раз, сколько захотите. Для **Player** это  
вполне целесообразно, так как мир NyetHack достаточно велик, чтобы вместить  
нескольких игроков.  
Но представьте, что вы решили создать класс **Game**, который будет следить за  
состоянием игры. Наличие нескольких экземпляров класса **Game** в игре нежелательно, потому что каждый может хранить свое состояние, что, скорее всего,  
приведет к отсутствию синхронизации между ними.  
Если вам необходим один экземпляр с непротиворечивым состоянием, существующий на протяжении всего времени работы программы, объявите *синглтон*.  
Экземпляр такого класса будет создан автоматически при первом обращении  
к нему. Этот экземпляр будет существовать на всем протяжении работы программы, и при каждом следующем обращении будет возвращаться первоначальный экземпляр.  
Есть три способа применения ключевого слова object: для создания *объявлений  
объектов (синглтонов), анонимных объектов и вспомогательных объектов*. Мы  
обозначим границы применения каждого из них в следующих разделах.

**Объявления объектов**Объявления объектов (синглтоны) полезны для организации и управления состоянием, особенно когда надо поддерживать какое-то состояние на протяжении  
работы программы. Для этого мы объявим объект **Game**.  
Объявление класса **Game** как синглтона также послужит нам удобным местом для определения цикла игры и позволит убрать его из функции **main**в Game.kt. Разделение кода на классы и объявление объектов еще больше  
способствует вашей цели, направленной на поддержание кода в организованном состоянии.  
Объявите объект **Game** в Game.kt, используя объявление объектов.  
**Листинг 15.1.** Объявление объекта Game (Game.kt)  
fun main(args: Array<String>) {  
...  
}  
private fun printPlayerStatus(player: Player) {  
println("(Aura: ${player.auraColor()}) " +  
"(Blessed: ${if (player.isBlessed) "YES" else "NO"})")  
println("${player.name} ${player.formatHealthStatus()}")  
}  
**object Game {  
}**

**Вспомогательные объекты ((static))**Если вы хотите связать инициализацию объекта с экземпляром класса, это  
можно организовать, определив вспомогательный объект. Вспомогательный  
объект объявляется внутри класса с помощью модификатора companion. У класса  
не может быть больше одного вспомогательного объекта.Инициализация вспомогательного объекта выполняется в двух случаях: при  
инициализации вмещающего класса, что делает его хорошим местом для хранения данных в единственном экземпляре, имеющих контекстную связь с объявлением класса, и при прямом обращении к одному из его свойств или функций.  
Так как фактически вспомогательный объект — это объявление объекта, вам не  
придется создавать экземпляр класса, чтобы использовать любые объявленные  
в нем функции или свойства. Давайте посмотрим на следующий пример вспомогательного объекта внутри класса с именем **PremadeWorldMap**:  
class PremadeWorldMap {  
...  
companion object {  
private const val MAPS\_FILEPATH = "nyethack.maps"  
fun load() = File(MAPS\_FILEPATH).readBytes()  
}  
}  
У **PremadeWorldMap** есть вспомогательный объект с единственной функцией **load**.  
Если потребуется вызвать **load** еще где-нибудь в коде, вы сможете сделать это  
без создания экземпляра **PremadeWorldMap**, как показано ниже:  
PremadeWorldMap.load()  
Содержимое файла будет загружено вспомогательным объектом только один  
раз, во время инициализации экземпляра **PremadeWorldMap** или при первом вызове функции **load**. И неважно, сколько экземпляров **PremadeWorldMap** будет  
создано, потому что вспомогательный объект всегда существует в единственном  
экземпляре.  
Понимание целей и задач синглтонов, анонимных и вспомогательных объектов — это ключ к их эффективному использованию. А их эффективное  
использование позволит вам написать хорошо организованный и масштабируемый код.

**42. Kotlin. Перегрузка операторов.**

**Перегрузка операторов**Вы уже знаете, что встроенные типы языка Kotlin поддерживают множество  
операций и некоторые даже адаптируют эти операции в зависимости от представляемых ими данных. Например, функция **equals** и связанный с ней оператор ==. С их помощью можно проверить, равны ли два экземпляра числового  
типа, содержат ли строки одинаковый набор символов и являются ли значения  
свойств двух экземпляров класса данных, объявленных в главном конструкторе, эквивалентными. Аналогично функция **plus** и оператор + складывают два  
числа, добавляют одну строку в конец другой и добавляют элементы из одного  
списка в другой.  
При работе с пользовательскими типами компилятор Kotlin не всегда знает, как  
к ним применить встроенные операторы. Как, например, сравнить два экземпляра **Player**? Если вы хотите использовать встроенные операторы со своими  
типами, переопределите функции-операторы, чтобы подсказать компилятору,  
что он должен делать с вашими типами. Этот процесс называется *перегрузкой  
операторов*.  
Вы видели широкое применение перегрузки операторов в главе 10 и в главе 11. Вместо вызова функции **get** для извлечения элемента из списка вы использовали оператор доступа по индексу []. Лаконичность синтаксиса Kotlin  
основывается на таких маленьких удобствах, как использование spellList[3]  
вместо spellList.get[3].  
**Coordinate** — явный кандидат на улучшение через перегрузку операторов. Вы  
перемещаете героя по миру, складывая свойства двух экземпляров **Coordinate**вместе. Вместо того чтобы описывать эту работу в **Direction**, можно перегрузить  
оператор **plus** для **Coordinate**.  
Сделаем так в Navigator.kt: добавьте функцию с модификатором operator.  
**Листинг 15.16.** Перегрузка оператора plus (Navigator.kt)  
enum class Direction(private val coordinate: Coordinate) {  
NORTH(Coordinate(0, -1)),  
EAST(Coordinate(1, 0)),  
SOUTH(Coordinate(0, 1)),  
WEST(Coordinate(-1, 0));  
fun updateCoordinate(playerCoordinate: Coordinate) =  
Coordinate(playerCoordinate.x + coordinate.x,  
playerCoordinate.y + coordinate.y)  
}  
data class Coordinate(val x: Int, val y: Int) {  
val isInBounds = x >= 0 && y >= 0  
**operator fun plus(other: Coordinate) = Coordinate(x + other.x, y + other.y)**}  
Теперь просто используйте оператор сложения (+), чтобы прибавить один экземпляр **Coordinate** к другому. Давайте сделаем это в **Direction**.  
**Листинг 15.17.** Использование перегруженного оператора (Navigator.kt)  
enum class Direction(private val coordinate: Coordinate) {  
NORTH(Coordinate(0, -1)),  
EAST(Coordinate(1, 0)),  
SOUTH(Coordinate(0, 1)),  
WEST(Coordinate(-1, 0));  
fun updateCoordinate(playerCoordinate: Coordinate) =  
**Coordinate(playerCoordinate.x + coordinate.x,  
playerCoordinate.y + coordinate.y)  
coordinate + playerCoordinate**}  
data class Coordinate(val x: Int, val y: Int) {  
val isInBounds = x >= 0 && y >= 0  
operator fun plus(other: Coordinate) = Coordinate(x + other.x, y + other.y)  
}В табл. 15.1 перечислены часто применяемые операторы, которые могут быть  
перегружены.  
**Таблица 15.1.** Распространенные операторы

|  |  |  |
| --- | --- | --- |
| **Оператор** | **Имя функции** | **Назначение** |
| + | **plus** | Складывает два объекта |
| += | **plusAssign** | Складывает два объекта и присваивает сумму первому |
| == | **Equals** | Возвращает true, если два объекта эквивалентны, и false, если нет |
| > | **compareTo** | Возвращает true, если объект слева от оператора имеет большее значение, чем объект справа, иначе возвращает false |
| [] | **get** | Возвращает элемент из коллекции по индексу |
| .. | **rangeTo** | Создает интервал |
| in | **contains** | Возвращает true, если объект присутствует в кол лекции |

Эти операторы могут быть перегружены в любом классе, но делать это стоит  
только тогда, когда в этом есть смысл. Собираясь изменить логику оператора  
сложения в классе **Player**, задайте себе сначала вопрос: «Что подразумевается  
под понятием “игрок плюс игрок”?». Попробуйте ответить на него, прежде чем  
перегружать оператор.  
Кстати, если вы решите переопределить **equals**, также стоит переопределить  
функцию **hashCode**. Пример переопределения этих функций через специальную  
команду IntelliJ показан в разделе «Для любопытных: алгебраические типы  
данных» в конце главы. Подробное объяснение, почему надо переопределять  
**hashCode**, выходит за рамки этой книги.

**43. Kotlin. Обобщенные типы во время выполнения: стирание и овеществление параметров типов. Вариантность: обобщенные типы и подтипы**

**Объявление обобщенных типов***Обобщенный тип* — это класс, конструктор которого принимает аргументы  
любого типа. Начнем с объявления своего обобщенного типа.  
Откройте проект Sandbox и добавьте новый файл с именем Generics.kt. Внутри  
Generics.kt объявите класс **LootBox** с *параметром обобщенного типа* для его содержимого, а также с private-свойством loot, которому присваивается аргумент  
item — награда. **Листинг 17.1.** Создание обобщенного типа (Generics.kt.)  
**class LootBox<T>(item: T) {  
private var loot: T = item  
}**Вы объявляете класс **LootBox** и делаете его обобщенным, указывая параметр  
обобщенного типа Т в угловых скобках (< >) для использования с классом.  
Параметр обобщенного типа Т служит заполнителем для определения типа  
награды.  
Главный конструктор класса **LootBox** принимает награду любого типа (item:  
T) и присваивает его значение private-свойству loot, также имеющему тип T.  
Обратите внимание, что параметр обобщенного типа часто обозначается  
единственной буквой Т, сокращенно от "type" (тип) слова «тип», хотя точно  
так же можно использовать любую букву или слово. Мы предлагаем использовать Т, так как это распространенная форма записи и в других языках,  
в которых есть обобщения, следовательно, использование этой записи будет  
удобно для чтения.  
Время опробовать новый класс **LootBox**. Добавьте функцию **main**, объявите  
пару видов наград и создайте пару экземпляров каждой награды в сундуке  
(loot box).  
**Листинг 17.2.** Объявление сундуков с наградой (Generics.kt.)  
class LootBox<T>(item: T) {  
private var loot: T = item  
}  
**class Fedora(val name: String, val value: Int)  
class Coin(val value: Int)  
fun main(args: Array<String>) {  
val lootBoxOne: LootBox<Fedora> = LootBox(Fedora("a generic-looking fedora",  
15))  
val lootBoxTwo: LootBox<Coin> = LootBox(Coin(15))**}  
Вы создали два вида наград (шляпы (fedora) и монеты (coins); и то и другое —  
это очень желанные награды) и два сундука, чтобы хранить их.  
  
Так как вы сделали класс **LootBox** обобщенным, можно использовать только  
одно объявление класса для поддержки разных сундуков с наградами: одних —  
только со шляпами, других — только с монетами и т. д.  
Обратите внимание на объявления переменных **LootBox**:  
val lootBoxOne: LootBox<Fedora> = LootBox(Fedora("a generic-looking  
fedora", 15))  
val lootBoxTwo: LootBox<Coin> = LootBox(Coin(15))  
Угловые скобки в объявлении типа переменной указывают, какой тип награды  
может хранить конкретный экземпляр **LootBox**.  
Обобщенные типы, как и другие типы в Kotlin, поддерживаются механизмом  
автоматического определения типов. Для иллюстрации мы явно указали типы,  
но их можно опустить, так как каждая переменная инициализируется значением конкретного типа. В своем коде вы обычно отбрасываете информацию  
о типе, если она не нужна. Так что не стесняйтесь и смело удаляйте ее, если  
вам захочется.  
**Обобщенные функции**Обобщенные параметры типа также работают и с функциями. Это хорошая  
новость, так как в данный момент игрок не может забрать награду из сундука.  
Настало время это исправить. Добавьте функцию, которая позволяет игроку  
достать награду, но только если сундук открыт. Следите за тем, открыт ли сундук, добавив свойство open.  
**Листинг 17.3.** Добавление функции fetch (Generics.kt.)  
class LootBox<T>(item: T) {  
**var open = false**private var loot: T = item  
**fun fetch(): T? {  
return loot.takeIf { open }  
}**}  
В примере вы объявили обобщенную функцию **fetch**, которая возвращает значение обобщенного типа Т, указанного в определении класса **LootBox** и служащего  
заполнителем для вида награды.  
Обратите внимание: если бы **fetch** была объявлена вне **LootBox**, тип Т был бы  
недоступен, так как Т привязан к объявлению класса **LootBox**. Однако обобщенные функции не требуют, чтобы класс использовал параметр обобщенного типа.  
В следующем разделе вы сможете в этом убедиться.  
Попробуйте забрать содержимое lootBoxOne в функции **main**, используя вашу  
новую функцию **fetch**, сперва из закрытого сундука.  
**Листинг 17.4.** Тестирование обобщенной функции fetch (Generics.kt.)  
...  
fun main(args: Array<String>) {  
val lootBoxOne: LootBox<Fedora> = LootBox(Fedora("a generic-looking fedora",  
15))  
val lootBoxTwo: LootBox<Coin> = LootBox(Coin(15))  
**lootBoxOne.fetch()?.run {  
println("You retrieve $name from the box!")  
}**}  
Используйте стандартную функцию **run** (с которой познакомились в главе 9)  
для вывода названия содержимого lootBoxOne, если оно имеется.  
Вспомните, что **run** ограничивает область видимости лямбды, которую принимает, экземпляром объекта-приемника. Это значит, что $name ссылается на  
свойство name класса **Fedora**.  
Запустите Generics.kt. Вывода не последует. Вы не смогли забрать награду, так  
как сундук закрыт. Теперь откройте сундук и снова запустите Generics.kt.  
**Листинг 17.5.** Открытие сундука (Generics.kt.)  
...  
fun main(args: Array<String>) {  
val lootBoxOne: LootBox<Fedora> = LootBox(Fedora("a generic-looking fedora",  
15))  
val lootBoxTwo: LootBox<Coin> = LootBox(Coin(15))  
**lootBoxOne.open = true**lootBoxOne.fetch()?.run {  
println("You retrieve a $name from the box!")  
}  
}  
В этот раз при выполнении Generics.kt вы увидите название полученной награды:  
You retrieve a generic-looking fedora from the box!  
**Несколько параметров  
обобщенного типа**Обобщенная функция или тип также могут иметь несколько параметров  
обобщенного типа. Допустим, вам хочется ввести еще одну функцию — **fetch**,  
которая принимает функцию, преобразующую награду одного вида в другой,  
например в монеты. Количество возвращаемых монет зависит от ценности  
исходной награды. Такое преобразование реализует функция высшего порядка lootModFunction, которая передается в **fetch**.  
Добавьте новую функцию **fetch** в LootBox, которая будет принимать функцию,  
преобразующую награду.  
**Листинг 17.6.** Использование нескольких параметров обобщенного типа  
(Generics.kt.)  
class LootBox<T>(item: T) {  
var open = false  
private var loot: T = item  
fun fetch(): T? {  
return loot.takeIf { open }  
}  
**fun <R> fetch(lootModFunction: (T) -> R): R? {  
return lootModFunction(loot).takeIf { open }  
}**}  
...  
В примере мы добавили в определение функции новый параметр обобщенного типа R , сокращенно от "return" (возврат), потому что он определяет  
тип возвращаемого значения. Мы поместили параметр обобщенного типа  
в угловых скобках непосредственно перед именем функции: fun <R> fetch.  
Функция **fetch** возвращает значение типа R?, то есть версию R с поддержкой  
значения null.  
Несколько параметров обобщенного типа **331**Вы также указали, что lootModFunction (посредством объявления типа функции  
(T)→R) принимает аргумент типа Т и возвращает тип R. Испытайте объявленную  
новую функцию **fetch**. В этот раз передайте функцию преобразования награды  
в качестве аргумента.  
  
...  
fun main(args: Array<String>) {  
val lootBoxOne: LootBox<Fedora> =  
LootBox(Fedora("a generic-looking fedora", 15))  
val lootBoxTwo: LootBox<Coin> = LootBox(Coin(15))  
lootBoxOne.open = true  
lootBoxOne.fetch()?.run {  
println("You retrieve $name from the box!")  
}  
**val coin = lootBoxOne.fetch() {  
Coin(it.value \* 3)  
}  
coin?.let { println(it.value) }**}  
Новая версия функции **fetch**, объявленная вами, возвращает тип переданной  
лямбды R. Вы вернули **Coin?** из лямбды, поэтому тип R в этом случае **Coin?**. Но  
новая версия **fetch** более гибкая и может возвращать не только монеты, следовательно, функция **fetch** вернет значение того же типа, что и лямбда, так как R  
зависит от типа значения, возвращаемого анонимной функцией.  
lootBoxOne содержит награду типа **Fedora**. Но ваша новая функция **fetch** вернет  
**Coin?** вместо **Fedora?**. Это возможно благодаря дополнительному параметру  
обобщенного типа R.  
Функция lootModFunction, передаваемая в **fetch**, вычисляет количество монет,  
умножая ценность награды в сундуке на три.  
Кстати, может быть, вы слышали о терминах *ковариантность* (*covariance*)  
и *контравариантность* (*contravariance*), которые описывают то, что делают  
out и in. По нашему мнению, эти термины очень туманно описывают работу in  
и out, поэтому мы постараемся избегать их. Мы упомянули их потому, что вы  
можете столкнуться с ними где-нибудь еще, но теперь будете знать: если слышите «ковариантность», то представляйте out, а если «контравариантность»,  
то представляйте in.  
В этой главе вы узнали, как использовать обобщения для расширения возможностей классов в языке Kotlin. Увидели, как можно ограничивать обобщенные  
типы и использовать ключевые слова in и out для определения роли производителя и потребителя обобщенного параметра.  
В следующей главе вы изучите расширения, позволяющие совместно использовать функции и свойства без наследования. Мы используем их для улучшения  
кода NyetHack.

* 1. **Kotlin. Корутины**

В языке Kotlin поддержка асинхронности и параллельных вычислений воплощена в виде **корутин** (**coroutine**). По сути корутина представляет блок кода, который может выполняться параллельно с остальным кодом. А базовая функциональность, связанная с корутинами, сосредоточена в библиотеке **kotlinx.coroutines**.

десь в функции main перебираем последовательность от 0 до 5 и выводит текущий элемент последовательности на консоль. Для имитации продолжительной работы внутри цикла вызываем специальную функцию **delay()** из пакета kotlinx.coroutines. В эту функцию передается количество миллисекунд, на которое выполняется задержка. Передаваемое значение должно иметь тип Long. То есть здесь функция будет выполнять задержку в 400 миллисекунд перед выводом на консоль текущего элемента последовательности.

После выполнения работы цикла выводим на консоль строку "Hello Coroutines".

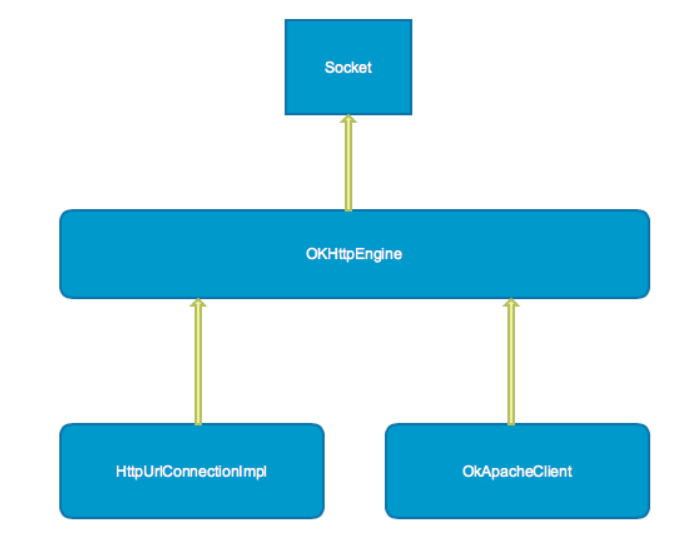
И чтобы использовать внутри функции main функцию delay(), функция main предваряется модификатром **suspend**. Модификатор **suspend** определяет функцию, которая может приостановить свое выполнение и возобновить его через некоторый период времени.

Сама функция **delay()** то же является подобной функцией, которая определена с модификатором **suspend**. А любая функция с модификатором suspend может вызывать только либо из другой функции, которая тоже имеет модификатор **suspend**, либо из корутины.

В этом и есть ключевая особенность suspend функции. Она не блочит поток. А значит мы можем эту корутину запускать в main потоке. Функция delay приостановит выполнение кода на 1 секунду, но не заблокирует main поток.

* 1. **Cетевые соединения и разрешения. HTTP Client. Чтение web-контента.**

В Android было два HTTP клиента: HttpURLConnection и Apache HTTP Client (HttpURLConnection). Команда Android в принципе не хотела активно работать над Apache HTTP Client. В 2013 году Square выпустила OkHttp. OkHttp был создан для прямой работы с верхним уровнем сокетов Java, при этом не используя какие-либо дополнительные зависимости. Она поставляется в виде JAR-файла. Для упрощения перехода, Square реализовала OkHttp используя интерфейсы HttpUrlConnection и Apache client

****

OkHttp получила большое распространение и поддержку сообществом, и, Google решили использовать версию 1.5 в Android 4.4 (KitKat). В 2015 Google официально признала AndroidHttpClient, основанный на Apache, устаревшим, вместе с выходом Android 5.1 (Lolipop). Сегодня OkHttp поставляется со следующим набором функций:

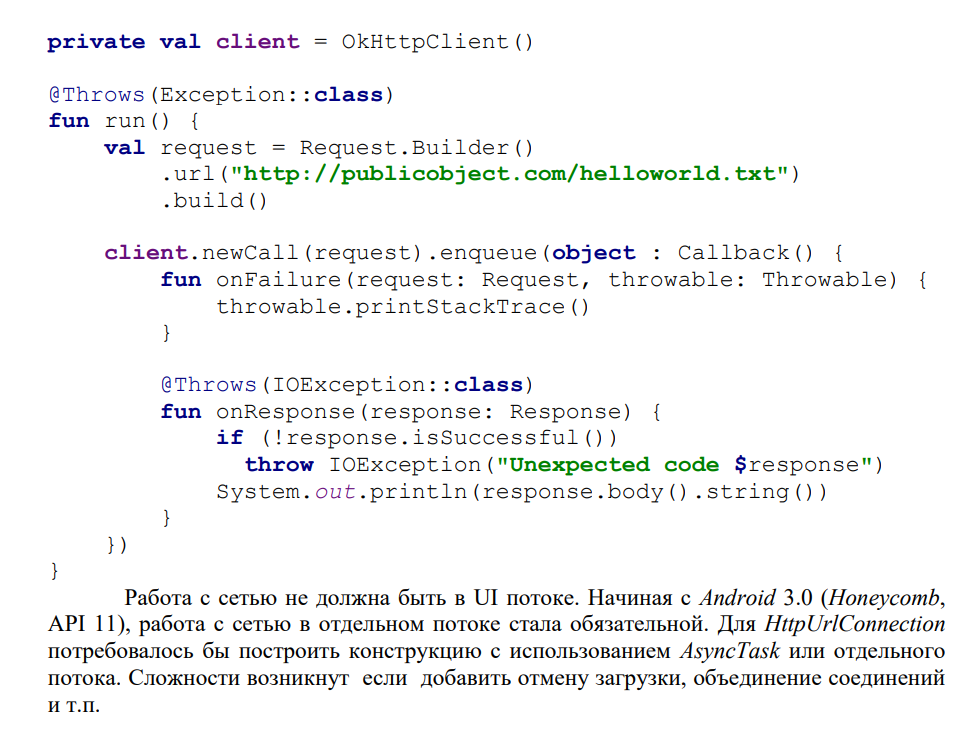
1. Поддержка HTTP/2 и SPDY позволяет всем запросам, идущим к одному хосту, делиться сокетом

2. Объединение запросов уменьшает время ожидания (если SPDY не доступен)

3. Прозрачный GZIP позволяет уменьшить вес загружаемой информации

4. Кэширование ответов позволяет избежать работу с сетью при повторных запросах.

5. Поддержка как и синхронизированных блокирующих вызовов, так и асинхронных вызовов с обратным вызовом (callback)



* 1. **Retrofit Взаимодействие потоков при сетевых загрузках**

Retrofit - это класс, с помощью которого ваши интерфейсы API превращаются в вызываемые объекты. По умолчанию Retrofit даст вам приемлимые значения по умолчанию для вашей платформы, но можно и настраивать. По умолчанию Retrofit может десериализовать только HTTP-тела в тип ResponseBody OkHttp, и он может принимать только тип RequestBody для @Body. Конвертеры могут быть добавлены для поддержки других типов. Шесть модулей адаптируют популярные библиотеки сериализации.

• Gson: com.squareup.retrofit2:converter-gson

• Jackson: com.squareup.retrofit2:converter-jackson

• Moshi: com.squareup.retrofit2:converter-moshi

• Protobuf: com.squareup.retrofit2:converter-protobuf

• Wire: com.squareup.retrofit2:converter-wire

• Simple XML: com.squareup.retrofit2:converter-simplexml

Android networking работает следующим образом:

Request — Выполнение HTTP-запроса к URL-адресу (конечной точке — endpoint) с соответствующими заголовками. При необходимости используется ключ авторизации.

Response — Запрос возвращает ответ, содержащий error или success. В случае успешного завершения запроса, ответ содержит содержимое endpoint (обычно в формате JSON)

Parse & Store — Парсинг JSON, получение необходимых значений и размещение их в классе данных.

Для работы с Retrofit понадобятся три класса.

1. POJO (Plain Old Java Object) или Model Class - json-ответ от сервера нужно реализовать как модель

2. Retrofit - класс для обработки результатов. Ему нужно указать базовый адрес в методе baseUrl()

3. Interface - интерфейс для управления адресом, используя команды GET, POST и т.д

**Как уже упоминалось выше:**определение интерфейса в Retrofit 2 одинаково для синхронных и асинхронных запросов. Другими словами, в Retrofit 2 каждый запрос упаковывается в объект Call. Фактический синхронный или асинхронный запрос выполняется по-разному с использованием **желаемого метода**для созданного позже объекта вызова.

* Используйте call.execute() для синхронного запуска.
* Используйте call.enqeue() для асинхронного запуска.
  1. **Анимации View animation**

Меняют только место отрисовки объекта, фактическое расположение остается неизменным, нужно перемещать самостоятельно.

До появления Animator в Android были только Animations. Основной недостаток которых был в том что они анимировали только представление вида и никак на самом деле не изменяли его свойства. Поэтому если хочется анимировать перемещение какого-либо элемента, то дополнительно по окончанию анимации нужно изменить ещё его свойства. Такой подход так или иначе не очень удобен, если вам нужна чуть более сложная анимация или нужно отлавливать нажатия в момент анимации.

• Применение:  
  
Там, где API не позволяет использовать Animator.  
  
• Достоинства:  
  
Отсутсвуют  
  
• Недостатки:  
  
Устаревший API, меняет только представление вида.

* 1. **Анимации. Property animation: ValueAnimator, ObjectAnimator**

Если нам нужно всего-лишь передвинуть что-нибудь на несколько пикселей в сторону или изменить прозрачность, чтобы не плодить миллион очень похожих друг на друга кадров на помощь приходит Animator. Фактически с помощью него можно анимировать любое свойство любых объектов.  
  
Базовый абстрактный класс называется Animator, у него есть несколько наследников, нам важны:  
  
ValueAnimator — позволяет анимировать любое свойство  
ObjectAnimator — наследуется от ValueAnimator и имеет упрощённый интерфейс для анимации свойств View. С ним вам не нужно вручную изменять какое-либо значение по событию изменения — вы просто даёте Animator'у объект и указываете поле, которое вы хотите изменить, например scaleX. С помощью Java Refliction ищется setter для этого поля (в данном случае — setScaleX. Далее Animator самостоятельно будет менять значение этого поля.  
ViewPropertyAnimator — Предоставляет ещё один удобный интерфейс для анимации View. Не унаследован от Animator и используется в методе View::animate()

В основе анимации Android лежат интерполяторы времени. ValueAnimator включает в себя эти интерполяторы — он имеет объект, который реализует интерфейс TimeInterpolator. Интерполяторы времени определяют, как анимация изменяется с течением времени.

* 1. **Использование crossfade анимации между Activity. Transitions framework (инфраструктура переходов)**

Crossfade анимация (также известная как затухание), это плавное исчезновение одного компонента с одновременным плавным появлением другого. Эта анимация удобна при переключении содержимого в приложении. Crossfade анимация довольно короткая, но предлагает плавное переключение с одного экрана на другой.

Для создания анимации:

1. Создаем переменные для объектов, которые хотим переключать. Необходимо создать ссылки после изменения объектов во время анимации.
2. Для объекта, который будет исчезать, ставим свойство visibility равное [GONE](https://easyandroid.ru/?go=http://developer.android.com/reference/android/view/View.html#GONE). Эта опция не позволит объекту занимать пространство разметки и не включит его в расчет разметки, что ускорит процесс.
3. Сохраним системное свойство [config\_shortAnimTime](https://easyandroid.ru/?go=http://developer.android.com/reference/android/R.integer.html#config_shortAnimTime) в переменную. Это свойство описывает стандартную “короткую” продолжительность анимации. Такая продолжительность идеальна для едва заметной анимации или для анимации, которая происходит очень часто. Параметры [config\_longAnimTime](https://easyandroid.ru/?go=http://developer.android.com/reference/android/R.integer.html#config_longAnimTime) и [config\_mediumAnimTime](https://easyandroid.ru/?go=http://developer.android.com/reference/android/R.integer.html#config_mediumAnimTime) также доступны и вы можете их использовать.

Когда объекты подготовлены, необходимо сделать следующее для переключения между ними:

1. Для объекта, который будет скрываться, устанавливаем значение alpha равное 0 и свойство visibility в значение [VISIBLE](https://easyandroid.ru/?go=http://developer.android.com/reference/android/view/View.html#VISIBLE). (Помните, что оно было инициализировано значением GONE). Это сделает объект видимым, но полностью прозрачным.
2. Для объекта, который будет появляться, изменяем значение alpha от 0 до 1. В то же самое время, для объекта, который будет исчезать, изменяем значение alpha от 1 до 0.
3. Используя метод [onAnimationEnd()](https://easyandroid.ru/?go=http://developer.android.com/reference/android/animation/Animator.AnimatorListener.html#onAnimationEnd(android.animation.Animator)) объекта [Animator.AnimatorListener](https://easyandroid.ru/?go=http://developer.android.com/reference/android/animation/Animator.AnimatorListener.html), устанавливаем свойство visibility объекта, который нужно показать, в значение GONE. Несмотря на то, что значение alpha равно 0, значение параметра visibility равное GONE не позволит объекту занимать пространство разметки и не включает его в расчет макета, что ускоряет обработку.

## Метод overridePendingTransition()

Если вы хотите задать собственную анимацию при переключении между активностями, то можете сделать это несколькими способами.

У класса **Activity** есть метод **overridePendingTransition()** и он должен идти сразу после вызова метода **startActivity()** или **finish()**. Метод принимает два параметра: ресурс анимации для запускающей активности и ресурс анимации для уходящей активности.

**В интенте вызвать этот метод, передать туда анимацию затухания и анимацию возникновения, в которых меняется прозрачность.**

Начиная с Android 4.4 в арсенале разработчиков появился дополнительный инструмент для создания анимаций — Transitions Framework. Изначально он предназначался для создания анимаций изменения состояния приложения путём манипулирования несколькими View. С выходом Android 5.0 набор доступных для использования анимаций был расширен, чтобы соответствовать представленной тогда же концепции Material Design.  
  
Transitions Framework позволяет быстро и безболезненно создавать различные анимации. Поэтому в процессе работы над iFunny было невозможно пройти мимо этого инструментария. Вниманию читателей предлагается частный случай использования Transitions API — создание анимации перехода между Activity с эффектом «бесшовности».

С визуальной точки зрения представленные в Transitions Framework анимации переходов между Activity можно условно разделить на два типа: обычные анимации и анимации с общим элементом.

Именно в KitKat Transition API появляется такое понятие как сцена — Scene, и Transition — переход между сценами. Для определения корневого layout вводится понятие Scene root, внутри которого и происходит изменение сцен. При этом сама сцена по сути является враппером над ViewGroup, описывающим своё состояние и состояние объектов View в нем.

Сам Transition — механизм, который позволяет считывать параметры View, изменяющиеся от сцены к сцене, и генерировать анимации для создания плавных переходов между ними.

**50. Physics-based Animations. Fling Animation. Spring Animation**

[**https://code.tutsplus.com/ru/tutorials/adding-physics-based-animations-to-android-apps--cms-29053**](https://code.tutsplus.com/ru/tutorials/adding-physics-based-animations-to-android-apps--cms-29053)

**51. Сервисы запущенные (started). Background и Foreground. Жизненный цикл**

Службы (Сервисы) в Android работают как фоновые процессы и представлены классом **android.app.Service**. Они не имеют пользовательского интерфейса и нужны в тех случаях, когда не требуется вмешательства пользователя. Сервисы работают в фоновом режиме, выполняя сетевые запросы к веб-серверу, обрабатывая информацию, запуская уведомления и т.д. Служба может быть запущена и будет продолжать работать до тех пор, пока кто-нибудь не остановит её или пока она не остановит себя сама. Сервисы предназначены для длительного существования, в отличие от активностей. Они могут работать, постоянно перезапускаясь, выполняя постоянные задачи или выполняя задачи, требующие много времени.

Клиентские приложения устанавливают подключение к службам и используют это подключение для взаимодействия со службой. С одной и той же службой могут связываться множество клиентских приложений.

Android даёт службам более высокий приоритет, чем бездействующим активностям, поэтому вероятность того, что они будут завершены из-за нехватки ресурсов, заметно уменьшается. По сути, если система должна преждевременно завершить работу запущенного сервиса, он может быть настроен таким образом, чтобы запускаться повторно, как только станет доступно достаточное количество ресурсов. В крайних случаях прекращение работы сервиса (например, задержка при проигрывании музыки) будет заметно влиять на впечатления пользователя от приложения, и в подобных ситуациях приоритет сервиса может быть повышен до уровня активности, работающей на переднем плане.

Используя сервис, можете быть уверены, что ваши приложения продолжат работать и реагировать на события, даже если они в неактивном состоянии. Для работы службам не нужен отдельный графический интерфейс, как в случае с активностями, но они по-прежнему выполняются в главном потоке хода приложения. Чтобы повысить отзывчивость вашего приложения, нужно уметь переносить трудоёмкие процессы (например, сетевые запросы) в фоновые потоки, используя классы **Thread** и **AsyncTask**.

Службы идеально подходят для проведения постоянных или регулярных операций, а также для обработки событий даже тогда, когда активности вашего приложения невидимы, работают в пассивном режиме или закрыты.

Сервисы запускаются, останавливаются и контролируются из различных компонентов приложения, включая другие сервисы, активности и приёмники широковещательных намерений. Если ваше приложение выполняет задачи, которые не зависят от прямого взаимодействия с пользователем, сервисы могут стать хорошим выбором.

Запущенные сервисы всегда имеют больший приоритет, чем бездействующие или невидимые активности, поэтому менее вероятно, что их работа завершится преждевременно при распределении ресурсов. Единственная причина, почему Android может досрочно остановить Сервис, — выделение дополнительных ресурсов для компонентов, работающих на переднем плане (как правило, для активностей). Если такое случится, ваш сервис автоматически перезапустится, когда будет достаточно доступных ресурсов.

Когда сервис напрямую взаимодействует с пользователем (например, проигрывая музыку), может понадобиться повысить его приоритет до уровня активностей, работающих на переднем плане. Это гарантия того, что сервис завершится только в крайнем случае, но при этом снижается его доступность во время выполнения, мешая управлять ресурсами, что может испортить общее впечатление от приложения.

Жизненный цикл служб

Подобно активностям служба имеет свои методы жизненного цикла:

* **onCreate()**
* **onStartCommand()**
* **onDestroy()**



Для быстрого создания заготовок нужных методов используйте команду меню **Code | Override Methods...** или набирайте сразу имя метода, используя автодополнение.

Реализуя эти методы обратного вызова в своей службе, вы можете контролировать жизненные циклы службы. В полном жизненном цикле службы существует два вложенных цикла:

* полная целая жизнь службы — промежуток между временем вызова метода **onCreate()** и временем возвращения **onDestroy()**. Подобно активности, для служб производят начальную инициализацию в **onCreate()** и освобождают все остающиеся ресурсы в **onDestroy()**
* активная целая жизнь службы — начинается с вызова метода **onStartCommand()**. Этому методу передаётся объект **Intent**, который передавали в метод **startService()**.

Из своего приложения службу можно запустить вызовом метода **Context.startService()**, остановить через **Context.stopService()**. Служба может остановить сама себя, вызывая методы **Service.stopSelf()** или **Service.stopSelfResult()**.

Можно установить подключение к работающей службе и использовать это подключение для взаимодействия со службой. Подключение устанавливают вызовом метода **Context.bindService()** и закрывают вызовом **Context.unbindService()**. Если служба уже была остановлена, вызов метода **bindService()** может её запустить.

Методы **onCreate()** и **onDestroy()** вызываются для всех служб независимо от того, запускаются ли они через **Context.startService()** или **Context.bindService()**.

Если служба разрешает другим приложениям связываться с собой, то привязка осуществляется с помощью дополнительных методов обратного вызова:

* **IBinder onBind(Intent intent)**
* **onUnbind(Intent intent)**
* **onRebind(Intent intent)**

В метод обратного вызова **onBind()** передают объект **Intent**, который был параметром в методе **bindService()**, а в метод обратного вызова **onUnbind()** — объект **Intent**, который передавали в метод **unbindService()**. Если служба разрешает связывание, метод **onBind()** возвращает канал связи, который используют клиенты, чтобы взаимодействовать со службой. Метод обратного вызова **onRebind()** может быть вызван после **onUnbind()**, если новый клиент соединяется со службой.

Термины *фон* и *передний план* перегружены, и могут применяться к:

1. жизненному циклу компонентов Android
2. потокам

В этой статье, по умолчанию будем считать, что термины *фон* и *передний план* относятся к жизненному циклу. Но, когда будет идти речь о потоках, мы будем явно говорить *фоновый поток* или *поток переднего плана*.

Существует подкласс android-служб, называемый [IntentService](https://developer.android.com/training/run-background-service/create-service.html), который запускает задачи в фоновом потоке "из коробки". Но мы не будем говорить о таких службах в этой статье.

Запущенные службы

Запущенные службы начинают свою работу после вызова метода startService(Intent) в вашей Activity или службе. При этом Intent должен быть явным. Это означает, что вы должны явно указать в Intent имя класса запускаемой вами службы. Или, если вам важно допустить некоторую неопределенность в отношении того, какая служба запускается, вы можете предоставить фильтры намерений для ваших служб и исключить имя компонента из Intent, но затем вы должны установить пакет для намерения с помощью setPackage(), который обеспечивает достаточное устранение неоднозначности для целевой службы.

**52. Сервисы привязанные (bound). Жизненный цикл**

Привязанные службы

В отличие от запущенных служб, привязанные службы позволяют установить соединение между компонентом Android, привязанным к службе, и службой. Это соединение предоставляется реализацией интерфейса IBinder, который определяет методы для взаимодействия со службой. Простым примером этого может быть реализация привязанной службы в одном процессе с клиентом (т.е. в рамках вашего собственного приложения). В этом случае Java-объект, подкласс Binder, передается клиенту, который может использовать его для вызова методов службы.

В более сложных сценариях, когда необходимо, чтобы интерфейс службы был доступен для разных процессов, для предоставления клиенту интерфейса службы следует воспользоваться объектом Messenger (является ссылкой на объект Handler, который получает обратный вызов для каждого вызова от клиента), благодаря чему со службой можно взаимодействовать с помощью объектов Message. Объект Messenger фактически основан на [AIDL](https://developer.android.com/guide/components/aidl.html) (Android Interface Definition Language). Messenger создает очередь из всех запросов клиентов в рамках одного потока, поэтому служба одновременно получает только один запрос. Если необходимо, чтобы служба обрабатывала одновременно сразу несколько запросов, можно использовать AIDL напрямую.

Отличия между привязанной и запущенной службами:

1. У клиентского компонента нет соединения с запущенной службой. Он просто использует объекты Intent посредством startService() или stopService(), которые обрабатываются службой в методе onStartCommand().
2. Когда клиентский компонент (Activity, Fragment или другая служба) соединяются с привязанной службой, они получают реализацию IBinder, посредством которой они могут вызывать методы у привязанной службы.

В любом случае, когда службе (привязанной или запущенной) необходимо отправлять сообщения привязанному клиенту, ей следует использовать что-то вроде [LocalBroadcastManager](https://developer.android.com/reference/android/support/v4/content/LocalBroadcastManager.html) (в том случае, если клиент и служба работают в одном процессе). Привязанные службы обычно не подключаются к привязанному клиентскому компоненту напрямую.

bindService() и onCreate()

Для того, чтобы клиентский компонент стал привязанным к службе, необходимо вызвать bindService() с явным Intent, как и в случае с запущенной службой.

Пример:

**public** **class** **MyActivity** **extends** **Activity**{

**void** **bind**(){

bindService(

**new** MyIntentBuilder(**this**).build(),

mServiceConnection,

BIND\_AUTO\_CREATE);

}

}

BIND\_AUTO\_CREATE это наиболее часто встречающийся флаг в случае вызова bindService(). Существуют и другие флаги (например, BIND\_DEBUG\_UNBIND или BIND\_NOT\_FOREGROUND). В случае BIND\_AUTO\_CREATE у привязанной службы вызывается onCreate(), если служба до этого еще не была создана. Фактически это означает, что служба создается в момент первой привязки к ней.

Как только вызывается bindService(), службе необходимо реагировать на запрос клиента и предоставить ему экземпляр IBinder, посредством которого клиент сможет вызывать методы привязанной службы. В примере выше, это реализуется с помощью ссылки mServiceConnection. Это обратный вызов (callback) ServiceConnection, который привязанная служба будет использовать для уведомления клиента о завершении привязки. Он также позволит клиенту узнать о разрыве соединения со службой.

Другими словами, привязка выполняется асинхронно. bindService() возвращается сразу же и *не* возвращает клиенту объект IBinder. Для получения объекта IBinder клиенту необходимо создать экземпляр ServiceConnection и передать его в метод bindService(). Интерфейс ServiceConnection включает метод обратного вызова, который система использует для того, чтобы выдать объект IBinder.

Ниже приведен пример реализации ServiceConnection:

**public** **class** **MyActivity** **extends** **Activity**{

**private** ServiceConnection mServiceConnection =

**new** ServiceConnection(){

**public** **void** **onServiceConnected**(

ComponentName cName, IBinder service){

MyBinder binder = (MyService.MyBinder) service;

mService = binder.getService();

// Get a reference to the Bound Service object.

mServiceBound = **true**;

}

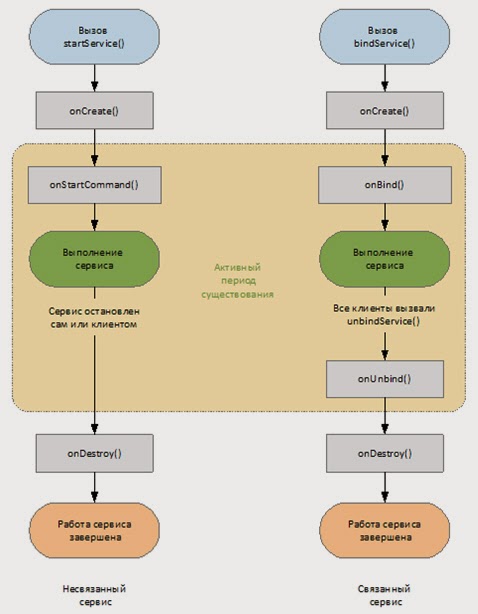
**public** **void** **onServiceDisconnected**(ComponentName cName){

mServiceBound= **false**;

}

};

}



**53. Создание службы. Класс Service. Организация взаимодействия Activity (fragment) – Service**

Создание службы

Чтобы определить службу, необходимо создать новый класс, расширяющий базовый класс **Service**. Можно воспользоваться готовым мастером создания класса для сервиса в Android Studio. Щёлкаем правой кнопкой мыши на папке **java** (или на имени пакета) и выбираем **New | Service | Service**:

В следующем окне выбираем имя сервиса (флажки оставляем) и нажимаем кнопку **Finish**.

Получим заготовку.

// Kotlin

package ru.alexanderklimov.service

import android.app.Service

import android.content.Intent

import android.os.IBinder

class MyService : Service() {

override fun onBind(intent: Intent): IBinder {

TODO("Return the communication channel to the service.")

}

}

// Java

package ru.alexanderklimov.service;

import android.app.Service;

import android.content.Intent;

import android.os.IBinder;

public class MyService extends Service {

public MyService() {

}

@Override

public IBinder onBind(Intent intent) {

// TODO: Return the communication channel to the service.

throw new UnsupportedOperationException("Not yet implemented");

}

}

При этом сервис автоматически зарегистрируется в манифесте в секции **<application>**.

<service

android:name=".MyService"

android:enabled="true"

android:exported="true" >

</service>

Если бы мы убрали флажки на экране мастера, то оба атрибута имели бы значение **false**. Например, атрибут **exported** даёт возможность другим приложениям получить доступ к вашему сервису.

Имеются и другие атрибуты, например, **permission**, чтобы сервис запускался только вашим приложением.

Также вы можете обойтись без мастера и создать вручную класс сервиса и запись в манифесте, теперь вы знаете, из чего он состоит.

Запуск сервиса и управление его перезагрузкой

В большинстве случаев также необходимо переопределить метод **onStartCommand()**. Он вызывается каждый раз, когда сервис стартует с помощью метода **startService()**, поэтому может быть выполнен несколько раз на протяжении работы. Вы должны убедиться, что ваш сервис это предусматривает.

Метод **onStartCommand()** заменяет устаревший метод **onStart()**, который использовался в Android 2.0. В отличие от **onStart()** новый метод позволяет указать системе, каким образом обрабатывать перезапуски, если сервис остановлен системой без явного вызова методов **stopService()** или **stopSelf()**.

@Override

public int onStartCommand(Intent intent, int flags, int startId) {

return Service.START\_STICKY;

}

Службы запускаются в главном потоке приложения; это значит, что любые операции, выполняющиеся в обработчике **onStartCommand()**, будут работать в контексте главного потока GUI. На практике при реализации сервиса в методе **onStartCommand()** создают и запускают новый поток, чтобы выполнять операции в фоновом режиме и останавливать сервис, когда работа завершена.

Такой подход позволяет методу **onStartCommand()** быстро завершить работу и даёт возможность контролировать поведение сервиса при его повторном запуске, используя одну из констант.

* **START\_STICKY** - Описывает стандартное поведение. Похоже на то, как был реализован метод **onStart()** в Android 2.0. Если вы вернёте это значение, обработчик **onStartCommand()** будет вызываться при повторном запуске сервиса после преждевременного завершения работы. Обратите внимание, что аргумент **Intent**, передаваемый в **onStartCommand()**, получит значение *null*. Данный режим обычно используется для служб, которые сами обрабатывают свои состояния, явно стартуя и завершая свою работу при необходимости (с помощью методов **startService()** и **stopService()**). Это относится к службам, которые проигрывают музыку или выполняют другие задачи в фоновом режиме
* **START\_NOT\_STICKY** - Этот режим используется в сервисах, которые запускаются для выполнения конкретных действий или команд. Как правило, такие службы используют **stopSelf()** для прекращения работы, как только команда выполнена. После преждевременного прекращения работы службы, работающие в данном режиме, повторно запускаются только в том случае, если получат вызовы. Если с момента завершения работы Сервиса не был запущен метод **startService()**, он остановится без вызова обработчика **onStartCommand()**. Данный режим идеально подходит для сервисов, которые обрабатывают конкретные запросы, особенно это касается регулярного выполнения заданных действий (например, обновления или сетевые запросы). Вместо того, чтобы перезапускать сервис при нехватке ресурсов, часто более целесообразно позволить ему остановиться и повторить попытку запуска по прошествии запланированного интервала
* **START\_REDELIVER\_INTENT** - В некоторых случаях нужно убедиться, что команды, которые вы посылаете сервису, выполнены. Этот режим — комбинация предыдущих двух. Если система преждевременно завершила работу сервиса, он запустится повторно, но только когда будет сделан явный запрос на запуск или если процесс завершился до вызова метода **stopSelf()**. В последнем случае вызовется обработчик **onStartCommand()**, он получит первоначальное намерение, обработка которого не завершилась должным образом.

Обратите внимание, что при окончании всех операций каждый из этих режимов требует явной остановки сервиса с помощью методов **stopService()** или **stopSelf()**.

Режим перезапуска, который вы указываете в качестве значения, возвращаемого методом **onStartCommand()**, будет влиять на параметры, передаваемые при последующих вызовах.

Изначально намерение выступает в качестве параметра, который передастся в метод **startService()** при запуске сервиса. После перезапуска системой он может иметь значение *null* (если установлен режим **START\_STICKY**) или оригинальное (если установлен флаг **START\_REDELIVER\_INTENT**).

Параметр **flag** может помочь узнать, как именно был запущен сервис:

* START\_FLAG\_REDELIVERY — указывает на то, что параметр Intent повторно передан при принудительном завершении работы сервиса перед явным вызовом метода stopSelf()
* START\_FLAG\_RETRY — указывает на то, что сервис повторно запущен после непредвиденного завершения работы; передается в том случае, если ранее сервис работал в режиме START\_STICKY

@Override

public int onStartCommand(Intent intent, int flags, int startId) {

if ((flags & START\_FLAG\_RETRY) == 0) {

// TODO Если это повторный запуск, выполнить какие-то действия.

}

else {

// TODO Альтернативные действия в фоновом режиме.

}

return Service.START\_STICKY;

}

**54. Кроссплатформенная разработка на Flutter и Dart. Особенности, компиляция.**