**Работа (Android разработчик)**

Темы для изучения:

1. Проф терминология
2. Android SDK
3. SOLID, KISS, DRY
4. Github Action
5. ООП, паттерны проектирования
6. Java/Kotlin
7. Работа с сетевыми протоколами (HTTP, REST, JSON и др.)
8. Moxy, Dagger2, RxJava2, LiveData, Flow
9. CI/CD системы
10. Тестирование
11. Gradle
12. UI guidelines
13. UI
14. Особенности работы мобильных приложений
15. Навыки работы с VSS
16. IoC
17. API Android
18. material design
19. coroutines, flow
20. опыт создание кастомных View-компонентов
21. работа с Google сервисами
22. Опыт работы с GraphQL, RxJava, Coroutines, Dependency Injection
23. Dagger DI
24. ЖЦ разработки ПО
25. ЖЦ android
26. Знание и понимание основных паттернов проектирования, GoF, MVP, MVVM + архитектурные шаблоны

Проф терминология

Альфа — период работы над программным обеспечением, когда разработчики добавляют новые функции, а тестировщики проверяют корректность их работы.

Аппрувить — согласовывать что-то (англ. approve).

Аттач — приложение к чему-либо, прикрепляемый файл (англ. attachment). Аттачить — прикреплять.

Бэкап — резервная копия, бэкапить — создавать ее. Бэкапить информацию надо как можно чаще, чтобы вернуть удаленные данные или сохранить работу, если произойдет сбой.

Бэклог — это перечень задач, расположенных в порядке важности, для команды разработчиков.

Грумить — «причесывать», то есть приводить в порядок и устранять недочеты.

Дамп — файл с полной или частичной копией содержимого памяти ПК или базы данных.

Дебажить код — проверять код на ошибки или «баги» (англ. to debug).

Деплой — публикация рабочей версии приложения в том месте, где оно должно функционировать.

Код-ревью — проверка кода.

Легаси-код — очень старый код, написанный предыдущей командой.

Мануал — инструкция.

Митап — встреча специалистов для обмена опытом.

Овертайм — переработка.

Ребутнуть — перезагрузить.

Редирект — перенаправление пользователя с одного сайта на другой.

Софт — программное обеспечение.

Факап — неудача.

Прод – сервер, на котором размещена стабильная версия веб-приложения, используемая для нужд конечных клиентов

Хаб — класс устройств для объединения компьютеров в сетях Ethernet.

**Декларативный код** отвечает на вопрос Что? – Отсортируйте этот список чисел по возрастанию

**Императивный код** отвечает на вопрос Как? – Условия если больше меньше и тд

**Котлин не поддерживает множественное наследование.**

**const val** – константа времени компиляции, значение должно быть присвоено во время компиляции.

**vararg** - с помощью этого ключевого слова мы можем объявить функцию, принимающую произвольное количество аргументов.

Android SDK

**Android SDK (Software Development Kit** или комплект разработки программного обеспечения) — это набор инструментов разработки для операционной системы Android.

**Инструменты разработчика SDK:**

1. **Android SDK Build tool** – основные функции сборка, отладка, запуск и тестирование;
2. **Android Emulator** – эмулятор андроид;
3. **Android SDK Platform-tools** – adb (инструмент командной строки, который помогает взаимодействовать с устройством. Он позволяет нам выполнять такие действия, как установка приложения, отладка приложения и т. д.), fastboot (позволяет прошить устройство новым образом системы), systrace (помогают собирать и проверять информацию о времени);
4. **Android SDK Tools** - полный набор инструментов для отладки и разработки на андроид.

**Основные компоненты, которыми можно пользоваться для разработки приложения, следующие:**

1**. Компоненты API**— набор программных интерфейсов приложений (API), которые позволяют приложениям взаимодействовать с различными функциями операционной системы Android.

2. **Эмулятор устройства** — инструмент, который позволяет разработчикам тестировать приложения на виртуальном устройстве Android без необходимости иметь физическое устройство.

3. **Драйвера устройств**— программное обеспечение, которое позволяет компьютеру подключаться к устройствам Android и передавать данные между ними.

4. Библиотеки

5. Инструменты отладки

6. Документация

7. Fastboot

8. ADB

Android SDK обеспечивает совместимость приложения с различными версиями операционной системы Android и различными устройствами, что делает его более доступным для широкой аудитории.

Gradle

**Gradle** — система для автоматизации сборки приложений.

JDK

**JDK (Java Development Kit)** - это комплексное программное обеспечение, предоставляемое компанией Oracle (или другими поставщиками, такими как OpenJDK), которое является основным инструментарием для разработки Java-приложений. JDK содержит в себе все необходимые компоненты для написания, компиляции, отладки, запуска и документирования Java-кода.

Android API

API - Это просто интерфейс, уровень абстракции, который обеспечивает связь между двумя разными «частями» программного обеспечения.

minSdkVersion, compileSdkVersion, targetSdkVersion

**minSdkVersion** - минимальная версия Android, на которой может работать приложение.

**compileSdkVersion** - версия SDK, с которой приложение компилируется.

**targetSdkVersion** - версия Android, для которой приложение оптимизировано и с которой оно было протестировано.

compileSdkVersion <= targetSdkVersion

SOLID, KISS, DRY

**SOLID** - набор принципов проектирования, созданных для разработки программного обеспечения при помощи объектно-ориентированных языков.

**Single responsibility (SRP)** - принцип единственной ответственности

**Open-closed (OCP)** - принцип открытости/закрытости

**Liskov substitution (LSP)** - принцип замены Барбары Лисков

**Interface segregation (ISP)** - принцип разделения интерфейса

**Dependency inversion (DIP)** - принцип инверсии зависимости

**S** - **Принцип единственной ответственности**

Должна быть единственная причина появления/изменения класса.

«Антипаттерн» (так, как лучше не делать).

Класс, который берет на себя слишком много (хранит много и делает много).

**O** **- Принцип открытости/закрытости**

Сущности должны быть открыты для расширения, но закрыты для модификации.

Разработанный (вошедший в определенную версию) класс неприкосновенен для изменений (можно только исправлять ошибки).

Новый функционал – в новые классы, которые могут переопределять существующие.

**L - принцип замены Барбары Лисков**

Замена в коде экземпляров классов на экземпляры их подклассов (наследников) не должна влиять на правильность работы программы.

Или

Функции, которые используют ссылки на базовые классы, должны иметь возможность использовать объекты подклассов, не зная об этом.

**I** **- Принцип разделения интерфейса**

Много интерфейсов, предназначенных для разных пользователей (других классов) лучше одного большого интерфейса, в который свален весь функционал.

**D** **- принцип инверсии зависимости**

Зависимости классов должны опираться на абстракции, а не на конкретную реализацию.

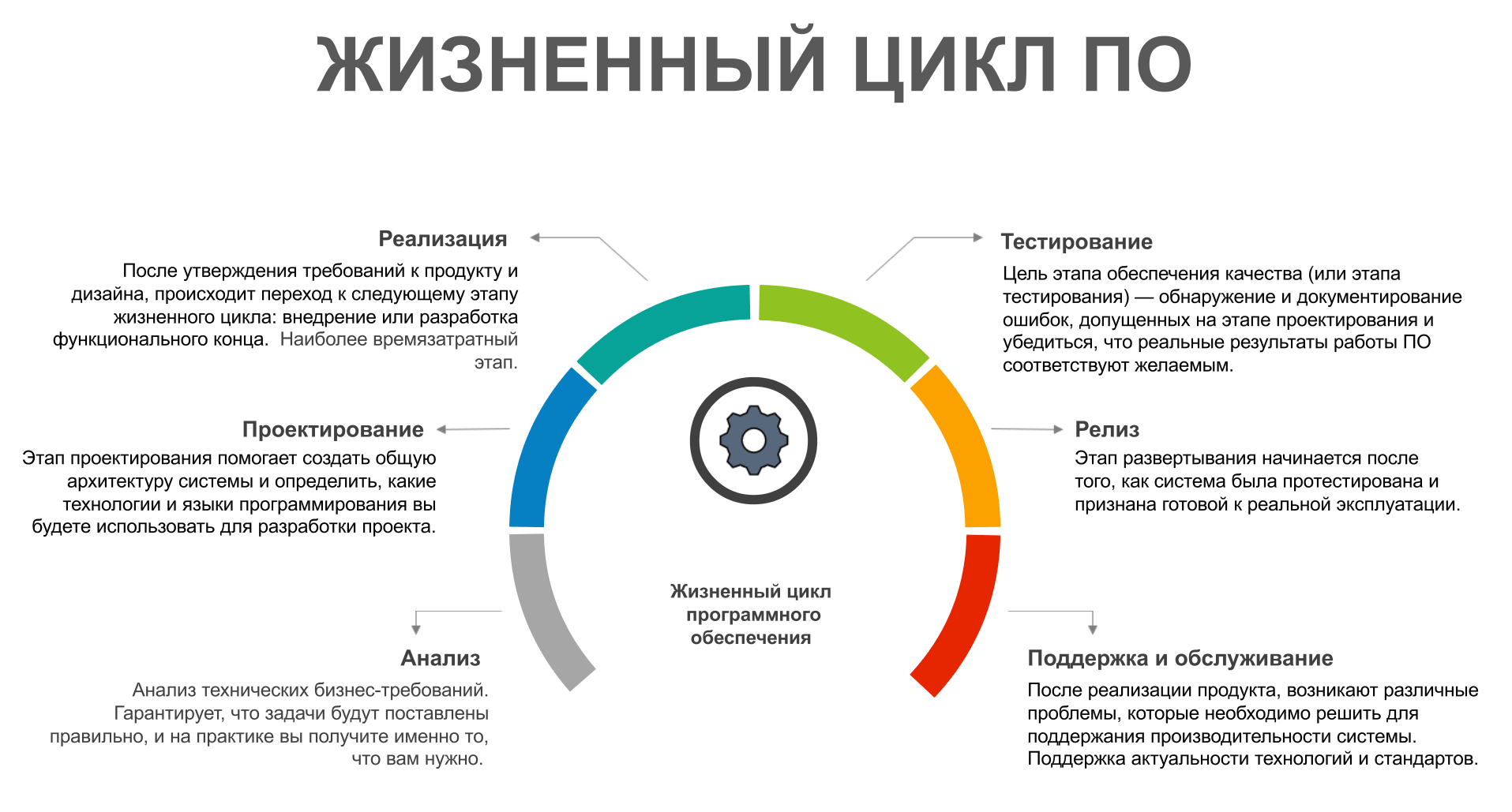
Принцип проектирования **KISS (keep it simple and straightforward)** провозглашает, что простота кода – превыше всего, потому что простой код – наиболее понятный.

! Подходы:

1. Декларативный – что делать?
2. Императивный – как делать?

**DRY (Don’t repeat yourself — «Не повторяйся»)** — это принцип разработки, который означает, что программист должен избегать повторов в реализации кода и в логике работы, а вместо этого использовать то, что есть.

ЖЦ разработки ПО



ЖЦ Activity



ЖЦ Fragment

fragment lifecycle states and their relation both the fragment's
            lifecycle callbacks and the fragment's view lifecycle

Kotlin

Мьютекс (mutex) — это концепция синхронизации, которая позволяет управлять доступом к разделяемым ресурсам в многопоточной среде. В языке программирования Kotlin есть встроенная поддержка мьютексов, что делает его выбором предпочтительным для разработчиков, которым нужно иметь гранулярный контроль над потоками выполнения.

ООП, паттерны проектирования

<https://habr.com/ru/articles/136766/>  
<https://in-kotlin.com/design-patterns/>

ООП принципы

1. **Наследование** – это механизм, позволяющий создавать новые классы на основе существующих. (способствует повторному использованию кода, иерархической организации классов и полиморфизму);
2. **Инкапсуляция** – это механизм, который позволяет скрывать внутреннюю реализацию объекта и предоставлять доступ к нему через хорошо определенный интерфейс. (помогает обеспечить целостность данных, способствует модульности и гибкости кода);
3. **Абстракция -** это процесс выделения существенных характеристик объекта и отбрасывания несущественных деталей. Реализуется с помощью абстрактных классов и интерфейсов, которые определяют общий набор методов и свойств для различных конкретных классов. (способствует модульности, повторному использованию кода и улучшению понимания системы);
4. **Полиморфизм -** это способность объектов различных классов выполнять одни и те же методы, но с разной реализацией. Достигается за счет перегрузки и переопределения методов. (позволяет создавать более гибкие и расширяемые программы, так как код может работать с объектами различных типов).

Архитектура Android

1. **Linux Kernel (Ядро Linux)**

Android использует ядро Linux в качестве базового уровня операционной системы. Ядро обеспечивает основные функции, такие как управление памятью, драйверы устройств, безопасность и управление процессами.

1. **Android Runtime (ART)**

ART - это среда выполнения приложений в Android. Она отвечает за компиляцию и выполнение кода приложения на устройстве. ART использует формат файлов исполняемого кода .dex (Dalvik Executable) и может предварительно компилировать приложение в байт-коде в момент установки (AOT - Ahead-Of-Time) или во время выполнения (JIT - Just-In-Time).

1. **Libraries (Библиотеки)**

Android предоставляет различные наборы библиотек для работы с графикой, мультимедиа, базами данных, сетевыми протоколами и другими аспектами разработки приложений.

1. **Android Runtime Libraries (Библиотеки Android Runtime)**

Этот слой включает в себя стандартные библиотеки Java (например, java.util, java.lang) и библиотеки, специфичные для Android (например, android.util, android.os).

1. **Application Framework (Фреймворк приложений)**

Фреймворк приложений предоставляет API для разработки приложений на Android. Он включает в себя набор компонентов, таких как активности (Activity), службы (Service), контент-провайдеры (Content Provider) и приемники широковещательных сообщений (Broadcast Receiver).

1. **Applications (Приложения)**

Верхний уровень архитектуры Android - это сами приложения, которые пользователь устанавливает и запускает на своем устройстве. Приложения могут быть предустановлены (например, приложения Google) или установлены из сторонних источников (например, из магазина приложений Google Play).

Процесс работы Android

1. **Загрузка инициализации**

Устройство загружает ядро Linux и инициализирует необходимые драйверы устройств и службы.

1. **Запуск Android Runtime**

После инициализации ядро запускает Android Runtime (ART), который загружает и запускает системные службы и процессы.

1. **Запуск пользовательского интерфейса**

Пользовательский интерфейс Android (например, Launcher) запускается в соответствии с настройками и предпочтениями пользователя.

1. **Запуск приложений**

Пользователь запускает приложения на устройстве, которые работают в среде выполнения Android Runtime (ART) и взаимодействуют с системными и пользовательскими службами через Application Framework.

1. **Взаимодействие с пользователем и обработка событий**

Приложения и системные службы взаимодействуют с пользователем через пользовательский интерфейс, обрабатывают ввод и события, отображают информацию на экране и выполняют различные задачи.

1. **Завершение работы и сброс устройства**

После завершения работы приложений и пользовательской активности устройство может перейти в спящий режим или быть выключено пользователем.

Паттерны Kotlin

1. **Шаблоны создания**: Как вы создаете объекты.

Builder, Factory, Singleton

1. **Структурные шаблоны**: Способ компоновки объектов.

Decorator, Facade

1. **Поведенческие паттерны**: То, как вы координируете взаимодействие объектов.

Observer, Strategy

Базовые компоненты андроид приложения  
1. **Content Provider** - это компонент операционной системы Android, который предоставляет данные другим приложениям. Он хранит и извлекает данные из базы данных и делает их доступными для других приложений на устройстве. Поставщики контента являются ключевой частью архитектуры обмена данными Android, поскольку они позволяют приложениям обмениваться данными и получать доступ к данным из внешних источников (приложение "Контакты" Android использует поставщика содержимого, чтобы сделать контактную информацию устройства доступной для других приложений, таких как приложение для набора номера).

2. **BroadcastReceiver** - это общесистемные события, которые могут произойти при запуске устройства, при поступлении сообщения на устройство или при входящих звонках, или при переходе устройства в режим полета и т. д. Broadcast Receivers используются для реагирования на эти общесистемные события. Широковещательные приемники позволяют нам регистрироваться для системных и прикладных событий, и когда это событие происходит, регистровые приемники получают уведомление. В основном существует два типа вещательных ресиверов:

* Статические вещательные ресиверы: Эти типы получателей объявлены в файле манифеста и работают даже при закрытии приложения.
* Динамические вещательные ресиверы: Эти типы приемников работают только в том случае, если приложение активно или свернуто.

3. **Activity** - это компонент приложения, который является одним из его фундаментальных строительных блоков. Его основное предназначение заключается в том, что оно служит точкой входа для взаимодействия приложения с пользователем, а также отвечает за то, как пользователь перемещается внутри приложения или между приложениями.

Проблемы activity:

\* Проблема адаптации на планшетах

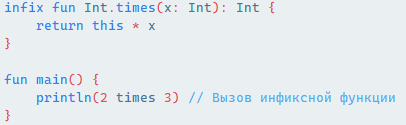
\* Проблема долгого запуска

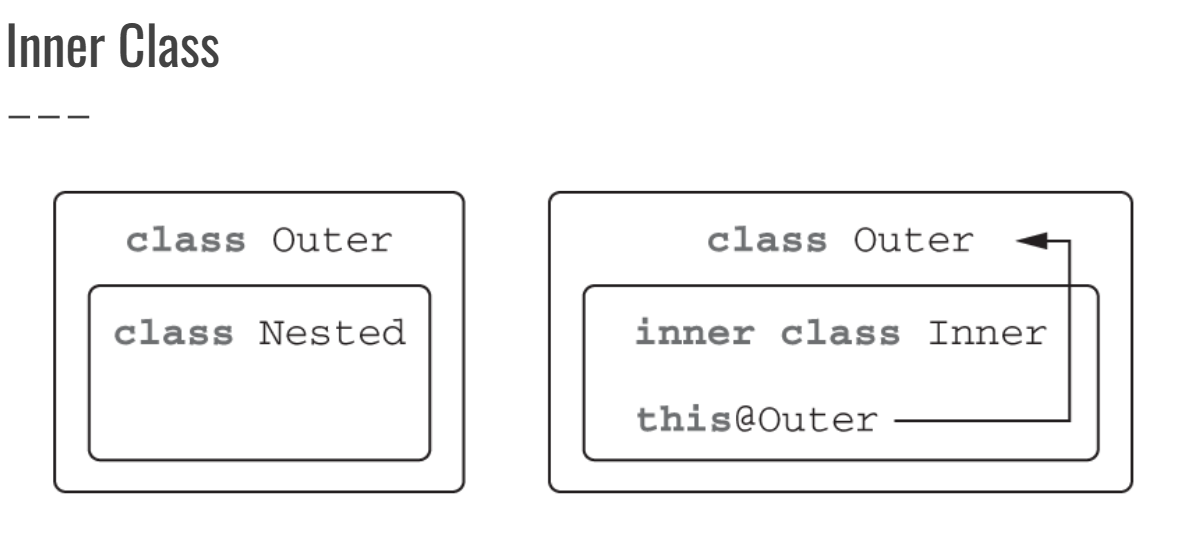
\* Нет нормальной поддержки navigation stack

4. **Service** - Службы в приложениях Android используются для выполнения некоторых фоновых задач в приложениях Android. Мы можем вызвать сервис для выполнения определенной задачи в фоновом режиме нашего приложения. Эта задача будет выполняться непрерывно, если наше приложение не видно пользователю. Сервисы обычно используются в приложениях Music Player, где песни воспроизводятся в фоновом режиме, и одновременно мы сможем использовать и другие приложения.

Infix

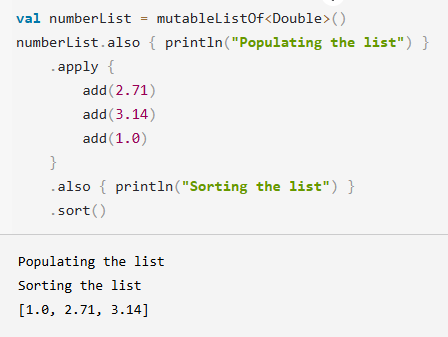
**Инфиксные функции (infix functions)** - это особый вид функций, которые можно вызывать с помощью инфиксной нотации, без использования точки или круглых скобок.



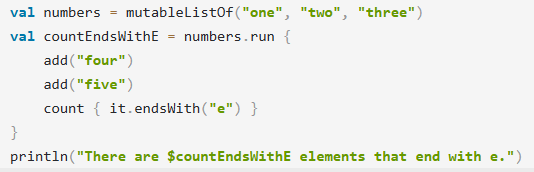
Inner class  


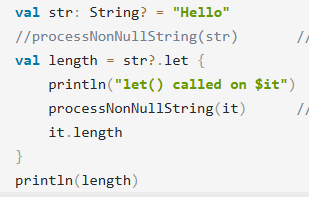
Scope функции (функции области видимости)

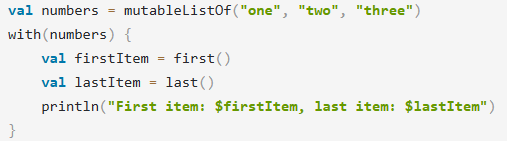
**apply, also** возвращают сам объект контекста



**let, run, with** возвращают результат лямбда-выражения







Flow

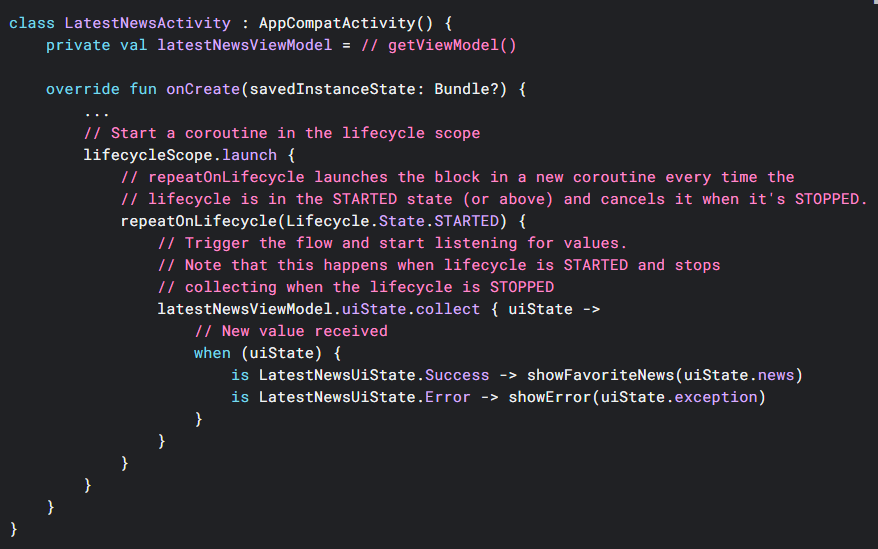
**Flow** – холодный поток данных (нет значения по умолчанию).

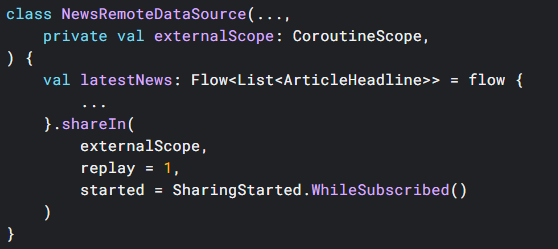
**StateFlow –** горячий поток данных. Когда новый потребитель начинает собирать данные из потока, он получает последний в потоке и любых последующих состояниях. Требует передачи начального состояния в конструктор. Не прекращает сбор автоматически.

В **StateFlow** вы можете получать текущее значение через свойство value, и для изменения значения вы можете использовать либо **value**, либо **emit()** (хотя второй способ не рекомендуется, так как StateFlow должен хранить состояние):



Никогда не собирайте поток из пользовательского интерфейса непосредственно из launch или launchIn.



С помощью shareIn можно холодный поток сделать горячим, то есть не привязанным к ЖЦ и в случае уничтожения скоупа, поток данных перестанет существовать:  


**SharedFlow -** не хранит текущее состояние; вместо этого он просто рассылает события всем подписчикам. Может хранить несколько значений, если задан параметр replay. Не сохраняет последнее состояние. Нет значения по умолчанию.

В **SharedFlow** вы всегда используете метод **emit()** для отправки значений:



**Используется в основном для:**

* **StateFlow** - управлять изменяемым состоянием UI или данных или когда вам важно хранить и иметь доступ к текущему значению состояния.
* **SharedFlow** - когда вам нужно передавать события, такие как уведомления, клики или любые другие временные события без необходимости сохранять состояние, или когда необходимо передавать одно или несколько значений новым подписчикам, но нет необходимости в хранении предыдущего значения.

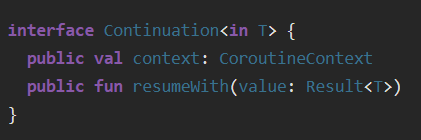
**collectAsState**: Не учитывает жизненный цикл, что может привести к избыточным подпискам и потенциальным утечкам памяти.

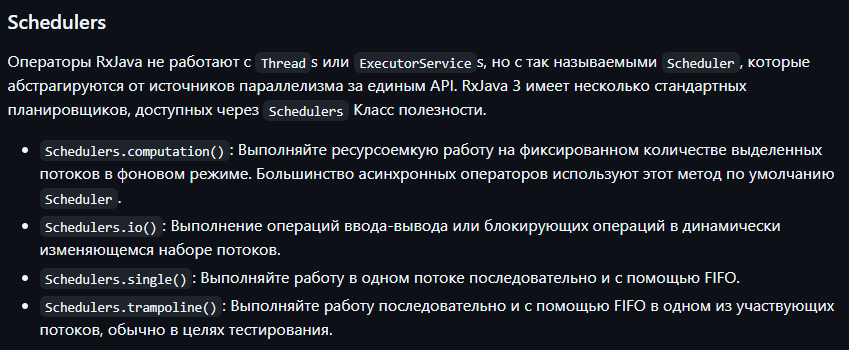
**collectAsStateWithLifecycle**: Учитывает жизненный цикл, автоматически управляя подписками на основе состояния компонента.

Coroutines

**Компилятор Kotlin создает специальную машину состояний для каждой suspend функции, эта машина берет управление корутиной на себя**

Suspend функции взаимодействуют друг с другом с помощью Continuation:





Intent, IntentFilter

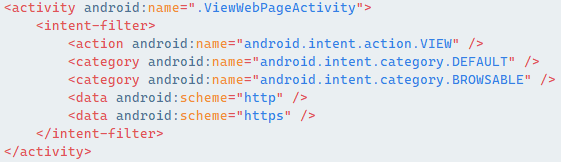
**Явные Intent** используются для указания точного компонента (например, Activity или Service), который должен обработать этот Intent. Это осуществляется путем указания полного имени пакета (например, com.example.MyActivity) или путем создания Intent для текущего приложения. Так же можно открыть какое-нибудь другое приложение через имя пакета.



**Неявные Intent** используются, когда вы хотите запустить компонент, который не является частью вашего приложения или когда вы не знаете, какой именно компонент должен обработать Intent. Вместо этого вы задаете действия (actions), категории (categories) и дополнительные данные (extras), которые описывают, что вы хотите сделать.



**Intent Filters** — позволяют вашему приложению обрабатывать различные Intent других приложений. (Открыть картинку из интернета в своем приложении).



Parcelable, Serializable

**Serializable** и **Parcelable** — это два механизма в Kotlin (и Java) для сериализации объектов. Они позволяют передавать объекты между компонентами Android, например, между Activity или Fragment через Intent.

Для простой сериализации нужно просто унаследовать класс, который хотим передать через Intent от **Serializable**.



**Parcelable** в свою очередь нужно реализовать. Parcelable лучше Serializable тк требует явного указания, как сериализовать поля, что может быть более сложным, но приводит к более быстрой обработке и меньшему использованию памяти.



Firebase

**Firebase** – платформа от Google, которая упрощает разработку мобильных приложений.

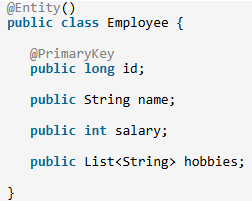
**Компоненты Firebase:**

1. Аутентификация пользователей;
2. Облачная БД NoSQL;
3. Хранилище для сохранения и извлечения файлов;
4. Облачные сообщения (отправка, получение);
5. Инструменты для отслеживания поведения пользователей внутри приложения.

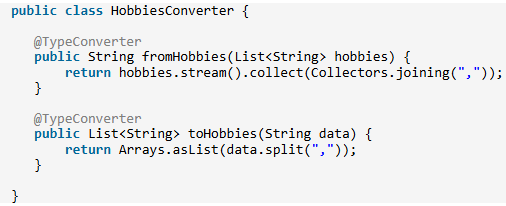
Room, TypeConverter

Иногда ваши Entity объекты могут содержать поля, которые не являются примитивами, и не могут быть сохранены в БД.

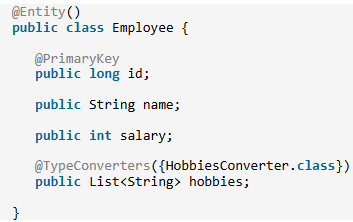
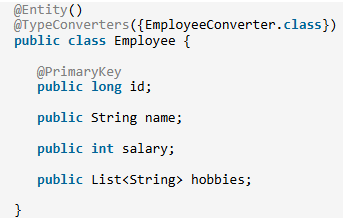
В качестве примера рассмотрим класс работника. У него вполне может быть поле, в котором мы хотим перечислить его хобби. Используем для этого поле hobbies с типом List<String>:



Создаем конвертер:



Осталось указать этот конвертер для поля hobbies.



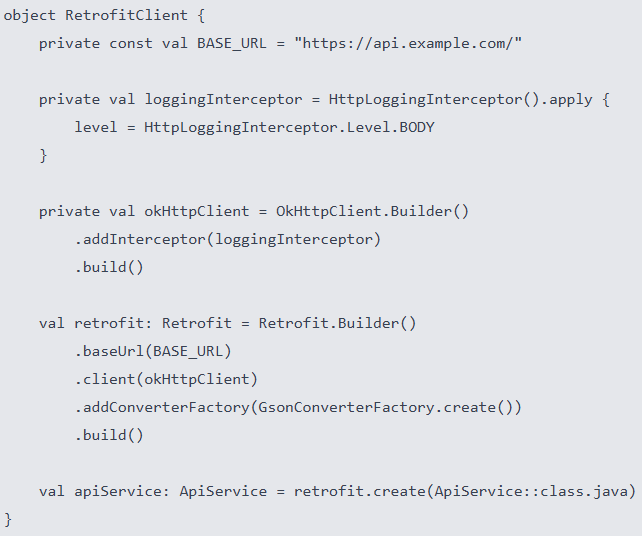
Retrofit, Interceptors

**Interceptor** - это мощная функция в Retrofit, которая позволяет перехватывать и изменять запросы до их отправки, а ответы — до их обработки. Это делает их идеальными для таких сценариев, как:

* Ведение журнала сетевых запросов и ответов для отладки.
* Добавление токенов аутентификации в запросы.
* Глобальная обработка конкретных ответов об ошибках.
* Изменение данных запроса/ответа для дополнительных заголовков или параметров.

Пример:

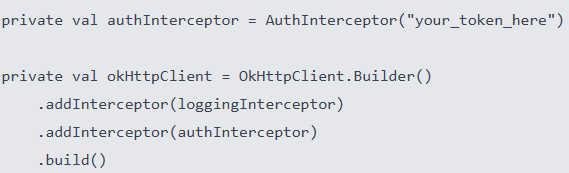
Создание экземпляр Retrofit, который настраивает базовый URL-адрес, преобразователь для сериализации JSON и перехватчик.



Простой пример перехватчика, который добавляет заголовок авторизации ко всем запросам:

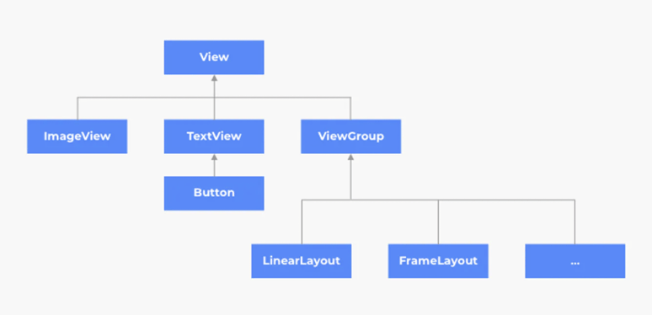


Добавьте свой пользовательский перехватчик в OkHttpClient при настройке Retrofit:



Создание кастомных view компонентов

**Android View** — это базовый класс для создания пользовательского интерфейса. View занимает прямоугольную область на экране, где он отвечает за измерение, компоновку и рисование себя вместе со своими дочерними элементами. Кроме того, представление обрабатывает все входные данные о событиях пользователя.



**ViewGroup** — это специальное представление, которое может содержать другие представления (дочерние) и определять свои собственные свойства макета.

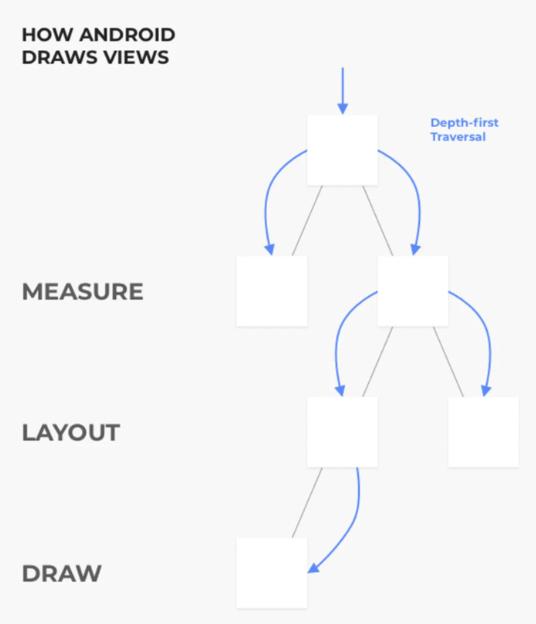
есть три этапа, которые должны произойти, прежде чем вид окажется на экране.

**Рисование представлений**

Эти три этапа — это измерение, разметка и рисование. Каждая из этих фаз представляет собой обход иерархии представления в глубину, идущей от родительского к дочернему. Для каждой фазы существует метод, который мы можем переопределить и изменить в зависимости от наших потребностей. Это методы onMeasure(), onLayout() и onDraw().

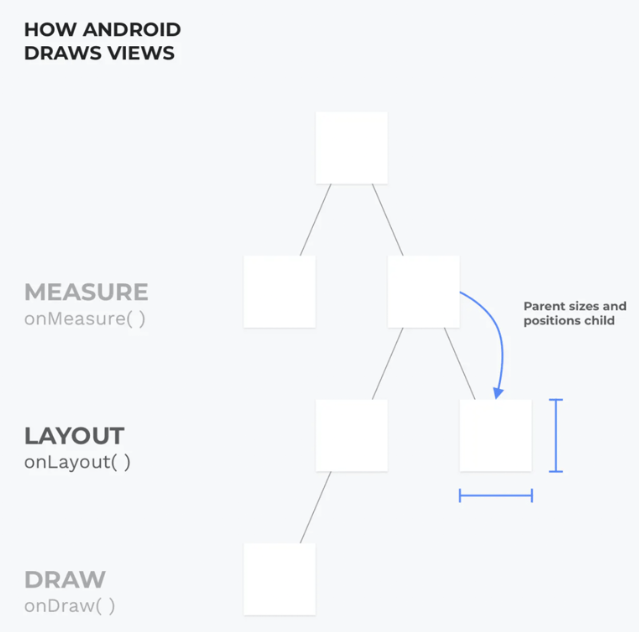
**Этот процесс можно разделить на два этапа:**

1. Этап измерения и компоновки
2. Стадия рисования



**Этап измерения и компоновки (onMeasure, onLayout)**

Процесс измерения похож на переговоры между родителем и ребенком. Дочерний элемент вычисляет желаемую ширину и высоту, но родитель является тем, кто делает окончательный вызов, устанавливая положение и размер своего потомка.



**Этап рисования**

Последним и самым важным шагом в рисовании пользовательского вида является переопределение метода **onDraw().** Canvas — это базовый класс, который определяет множество методов для рисования текста, растровых изображений, линий и других графических примитивов.

Каждый родитель нарисует себя сам, а затем попросит, чтобы каждый дочерний элемент сделал то же самое. Интересным побочным эффектом является то, что родитель рисует себя первым, а он оказывается внизу, в то время как его дочерние элементы рисуются сверху, покрывая его.

Jetpack Compose

**remember** — это функция, которая позволяет сохранить состояние между пересозданием компонентов.

val count **=** remember {…} – обращение к значению состояния через value

val count **by** remember {…} – обращение к значению состояния напрямую

**Виды remember**:

* **val count by remember { 10 }** – значение сохранено, но не будет меняться, но сохраняется при рекомпозиции;
* **val count by remember {mutableStateOf(10)}** – значение сохранено и будет меняться (mutableStateOf() - возвращает объект состояния, который отслеживается системой Compose и приводит к перерисовке интерфейса при изменении значения);
* **val count by rememberSaveable {mutableStateOf(10)}** – сохраняется при перевороте экрана;
* **val coroutineScope = rememberCoroutineScope()** – создание корутины;
* **val currentOnUpdate by rememberUpdatedState(onUpdate)** - используется для отслеживания состояния, которое может меняться в процессе работы. Composable, например, при использовании callback-функций. Еще всегда возвращает актуальное значение, которое обновляется автоматически при каждом новом рендере, что позволяет избежать устаревших ссылок на состояние.

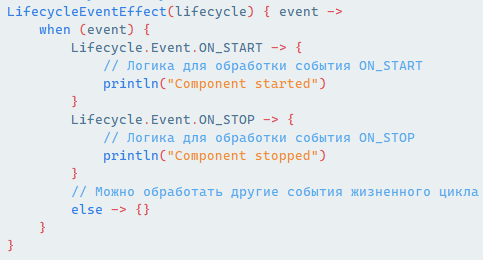
**LaunchedEffect** – инструмент для работы с асинхронными задачами, который позволяет выполнять побочные эффекты и управлять состоянием на основе изменения значений. Так же используется для запуска корутин.

**rememberUpdatedState** тоже самое что и LaunchedEffect, только следит за состоянием в переменной, а не в блоке кода.

**DisposableEffect** - в Jetpack Compose используется для выполнения операций, которые требуют очистки или отмены при выходе из композиции.

**SideEffect** - выполняется после завершения отрисовки текущего кадра. Повторно выполняется всякий раз, когда любой из состояний компонента изменяется и компонент производится заново.

**LifecycleEventEffect** — это вспомогательный компонент в Jetpack Compose, который позволяет управлять и реагировать на события жизненного цикла компонента.



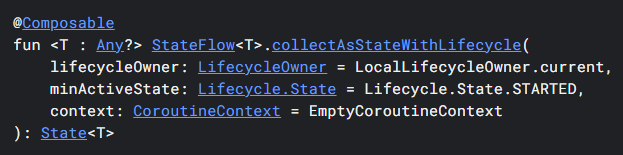
**produceState** - Используется для асинхронного получения данных.

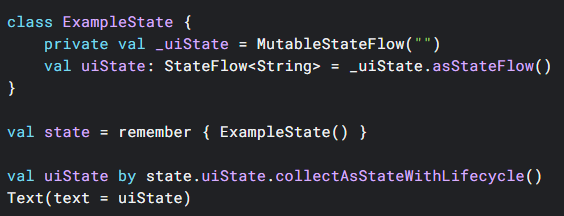
**derivedStateOf** - Если зависимости изменяются, то значение производного состояния будет повторно вычислено, и компонент будет перерисован только в случае необходимости.

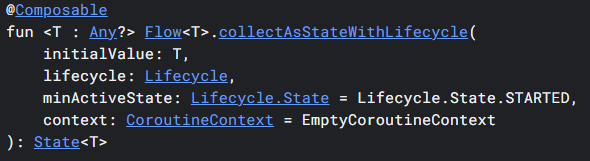
**snapshotFlow** — это функция, которая позволяет создавать поток (Flow) для отслеживания изменений состояний во время сессии Compose.

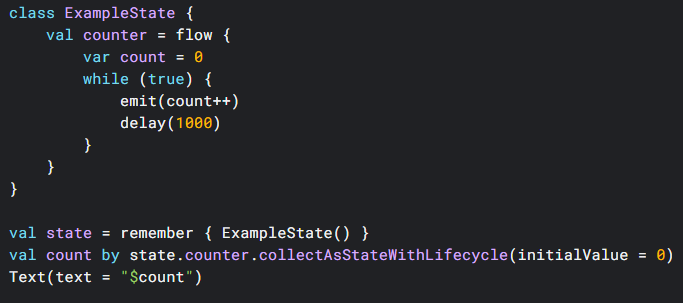
collectAsState, collectAsStateWithLifecycle, observeAsState

**collectAsStateWithLifecycle** - собирает значения из этого StateFlow или Flow и представляет свою последнюю стоимость через State с учетом жизненного цикла.

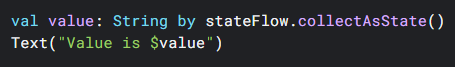








**collectAsState** - собирает значения из этого StateFlow или Flow и представляет свое последнеесостояние через State.





**observeAsState** – наблюдает за LiveData и отслеживает изменения через State.

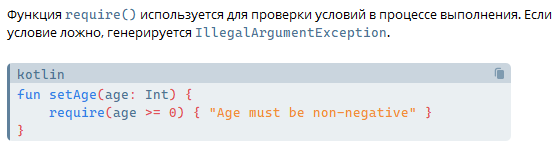


State

**State** – Интерфейс, держатель значения, считывает value во время выполнения Composable функции, текущая RecomposeScope будет подписана на изменения этого состояния.

Exceptions

1. **ConcurrentModificationException** - защитный механизм Kotlin для предотвращения несогласованности данных и состояния гонки. Мы можем столкнуться с этим исключением в одном потоке, если попытаемся изменить коллекцию во время ее перебора.
2. **ArithmeticException** – ошибка в арифметике (деление на 0)
3. **ArrayIndexOutOfBounds** – ошибка выхода за границы массива
4. **IllegalArgumentException** - возникает, когда метод получает аргумент, который не является допустимым или не соответствует ожидаемым требованиям.



1. **NumberFormatException** - возникает, когда вызывается метод для преобразования строки в числовой формат, но строка не имеет соответствующего формата.
2. **IOException** — возникает при использовании потока ввода-вывода файла.
3. **SQLException** - возникает при выполнении запросов к базе данных, связанных с синтаксисом SQL.
4. **ClassCastException** - выбрасывается при попытке привести объект к несовместимому типу.

Uri

**URI (Uniform Resource Identifier)** в Kotlin (и в программировании в целом) представляет собой стандартный способ идентификации ресурса. URI используется для указания местоположения ресурса в интернете и может принимать различные формы, в том числе URL (Uniform Resource Locator), который также включает в себя информацию о том, как получить доступ к этому ресурсу.

**URI может состоять из нескольких частей:**

- Схема: определяет тип ресурса (например, http, https, ftp, mailto, file и т. д.).

- Хост: доменное имя или IP-адрес.

- Путь: местоположение ресурса на сервере.

- Параметры: могут включаться дополнительные данные для доступа к ресурсу (например, параметры запроса).

- Фрагмент: часть ресурса, указывающая конкретную разделенную часть (например, якорь на HTML-странице).

**Пример URI**: https://www.example.com/path/to/resource?query=param#fragment

Уведомления

**Push-уведомления** – уведомления, которые отправляются с сервера

**Локальные уведомления** – уведомления, которые отправляются из вашего приложения.

Сервисы

**Foreground Srevice –** всегда запускается на переднем плане, это позволяет избежать повторного использования объектов службы системой Android, когда у ОС Android недостаточно ресурсов. Служба переднего плана Android может взаимодействовать с пользователями посредством уведомлений. (Медиа-плеер, работа с местоположением).

**Background Service –** компонент, который позволяет выполнять задачи в фоновом режиме, не требуя при этом непосредственного взаимодействия с пользователем.(загрузка больших файлов, обработка данных или получение содержимого с серверов).  
**Started Services** - запущенные сервисы создаются с помощью метода startService(). Они предназначены для выполнения долгоживущих операций, которые не требуют взаимодействия с пользователем.

**Bound Services** - привязанные сервисы создаются с помощью метода bindService(). Они предоставляют механизм для взаимодействия между компонентами (например, активностями) и самим сервисом.

Базовый класс Kotlin Any

**Any** – базовый класс, включает в себя 3 метода: equals, hashCode и toString.

**equals() (тоже что и == - по знач, === - по ссылкам)** – сравнивает по ссылкам объекты (class Person), примитивные типы данных сравнивает по значению (int, string…)

Исключения в data class – сравнивает по полям

Исключения в листах – mutableListOf(1,2,3), mutableListOf(2,3,4) сравнивает по значениям. mutableSetOf(1,2,3), mutableListOf(1,2,3), сравнивает по коллекциям или по ссылке.

**hashCode()** – сравнивает хэши

Сначала сравнивается хэш, если он одинаковый отрабатывает equals, если нет возвращает false и ничего больше не вызывает.

**toString()** - возвращающает строку, содержащую имя класса и хэш-код объекта.

Исключение data class, возвращает строку, которая содержит имя класса и значения всех его свойств в формате "ClassName(property1=value1, property2=value2, ...)"

Базовый класс Java Object

**getClass()** - используется для получения объекта класса, к которому принадлежит текущий экземпляр объекта, содержит информацию о самом классе, его метаданных и структуре.

**clone()** - создает и возвращает копию этого объекта.

**finalize() (deprecated)** - вызывается сборщиком мусора (garbage collector) перед тем, как объект будет уничтожен.

**toString()** - возвращает строковое представление объекта. возвращает строку, состоящую из имени класса, и хеш-кода объекта.



**equals(Object obj)** – проверяет равенство по ссылкам, если переопределить для классов - по значению, в списках сравнивается по значениям, а не по ссылкам, примитивы тоже сравниваются по значениям.

**hashCode()** - возвращает значение хеш-кода для объекта.

**wait(), wait(long timeoutMillis), wait(long timeoutMillis, int nanos) -** освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод notify().

**notify() -** продолжает работу потока, у которого ранее был вызван метод wait().

**notifyAll() -** возобновляет работу всех потоков, у которых ранее был вызван метод wait().

Data class

**equals()/hashCode()** - пары.

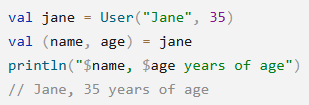
**toString()** – в виде “User(name=John, age=42)".

**componentN()** – соответсвует свойствам в порядке их объявления.

**copy()** - копирует объект, позволяет изменять некоторые его свойства, сохраняя при этом остальные без изменений.

**Data class должны удовлетворять следующим требованиям:**

1. Основной конструктор должен иметь хотя бы один параметр.
2. Все параметры первичного конструктора должны быть помечены как val or var.
3. Классы данных не могут быть abstract, open, sealed, or inner.

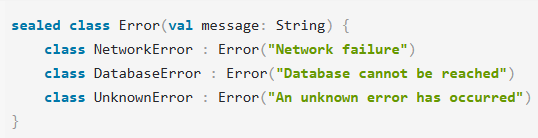


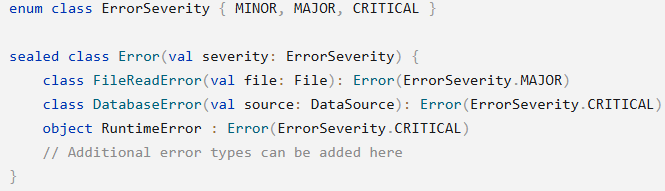
Sealed classes and interfaces

**Sealed классы и интерфейсы** обеспечивают контролируемое наследование иерархий классов. Все прямые подклассы запечатанного класса известны на момент компиляции. Никакие другие подклассы не должны появляться за пределами модуля и пакета, в которых определен запечатанный класс. Та же логика применима и к sealed интерфейсам и их реализациям: после компиляции модуля с sealed interface новые реализации не могут быть созданы.

Используются в when(), не добавляется else.

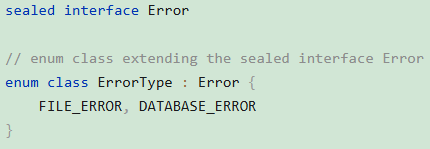
**Конструкторы предназначены не для создания экземпляров самого запечатанного класса, а для его подклассов:**





Конструктор может быть private и protected

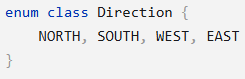
**Enum** классы не могут расширять класс sealed или любой другой класс. Тем не менее, они **могут реализовывать sealed интерфейсы**:

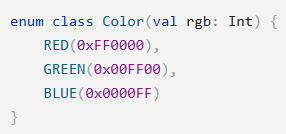


Нужны для: управления пользовательским интерфейсом, обработка способа оплаты, API запросов.

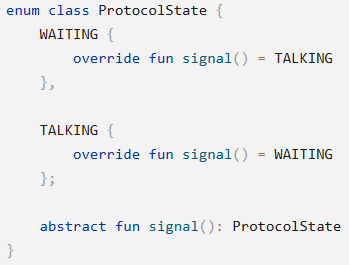
Enum class

**Enum class** – перечисления. Каждая константа перечисления является объектом. Константы перечисления разделяются запятыми.

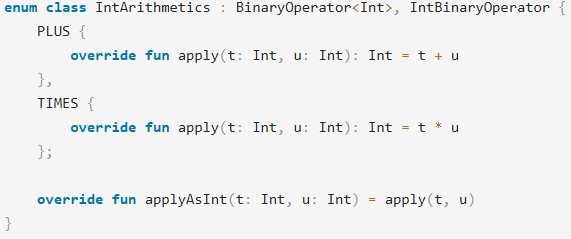




Константы перечисления могут объявлять свои собственные **анонимные классы** с соответствующими методами, а также с переопределяющими базовыми методами.



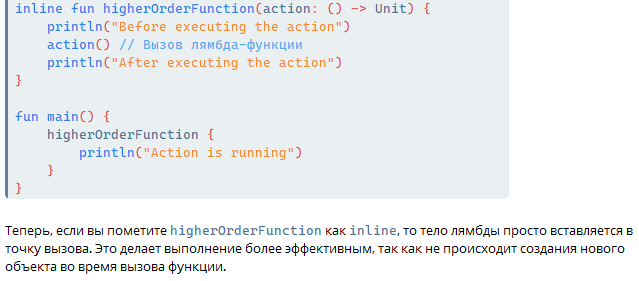
Класс перечисления может реализовывать интерфейс (но он не может быть производным от класса), предоставляя либо общую реализацию членов интерфейса для всех записей, либо отдельные реализации для каждой записи в своем анонимном классе.



Есть метод valueOf() для получения значения и entries, чтобы все перебрать (for). Также есть свойства name (имя), ordinal (позиция).

Inline, noinline function

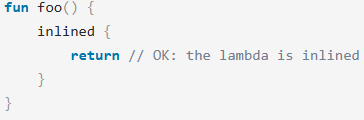
**Inline функции** — функции, тело которых будет вставлено ("встроено") в код вызова, а не вызывается в обычном (ритуальном) режиме. Улучшает производительность.



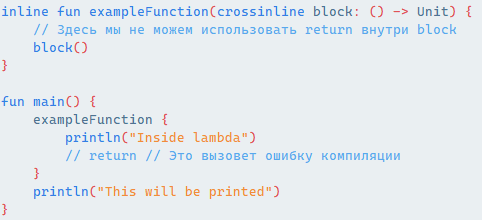
**noinline функции** – указываются для того, чтобы при передаче их как аргументов в inline функцию они не встраивались в нее.



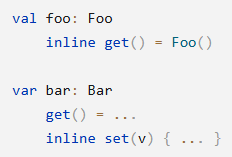
Можно писать return в inline функции – **нелокальные возвраты.**

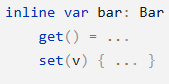


Чтобы указать, что лямбда-параметр встроенной функции не может использовать нелокальные возвраты, отметьте лямбда-параметр с помощью параметра **crossinline** модификатор.



Модификатор можно использовать для методов доступа к свойствам





Модификаторы доступа

**private** - означает видимость только внутри этого класса (включая его члены);

**protected** — то же самое, что и private + видимость в субклассах;

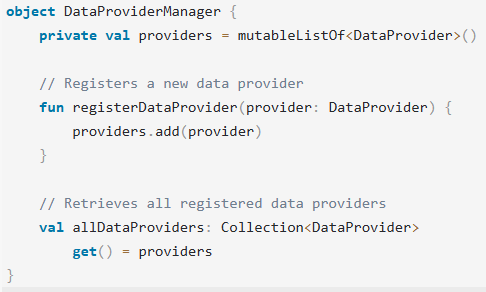
**internal** — любой клиент внутри модуля, который видит объявленный класс, видит и его internal члены;

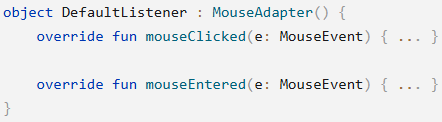
**public** — любой клиент, который видит объявленный класс, видит его public члены.

Object, companion object

Нужны для создания синглтона.

**object** - позволяет определить класс и создать его экземпляр за один шаг, что полезно для реализации синглтонов.



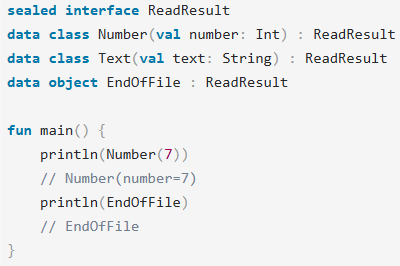


**Data object**, как и data class переопределяют методы:

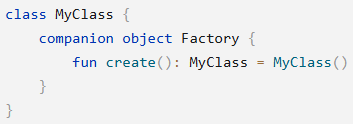
**toString()** – возвращает имя объекта данных (без data MyObject@hashcode, с data MyObject).

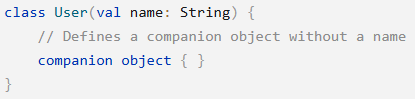
**equals()/hashCode()** - включает проверки на равенство и коллекции на основе хэша, в отличие от data class их нельзя переопределить.

Объявления объектов данных особенно полезны для sealed классы или sealed интерфейсы, тк он получит toString() без необходимости переопределять ее вручную:

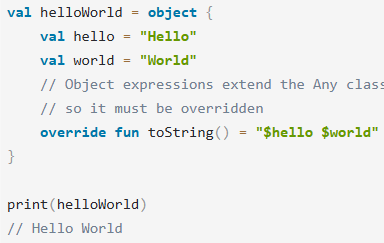


**companion object** - позволяют определять функции и свойства на уровне класса. Это упрощает создание заводских методов, удержание констант и доступ к общим утилитам. Его название может быть опущено.

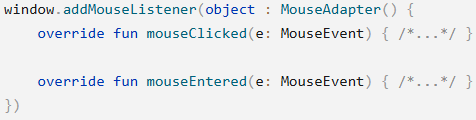




Анонимные объекты

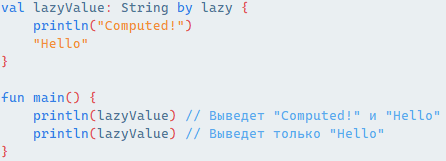


Чтобы создать анонимный объект, который наследует от какого-либо типа (или типов), укажите этот тип после object и двоеточие :. Затем реализуйте или переопределите члены этого класса, как если бы вы наследовали от него:



Делегаты lazy, observable, vetoable

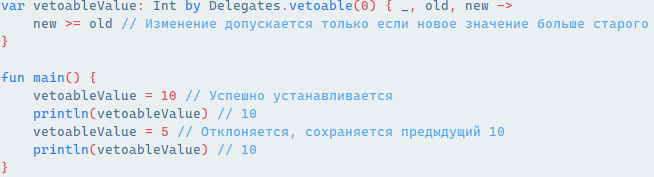
Делегат **lazy** предоставляет ленивую инициализацию свойства. Значение свойства не вычисляется до первого обращения.



Делегат **observable** позволяет отслеживать изменения в свойстве и выполнять код при каждом изменении.



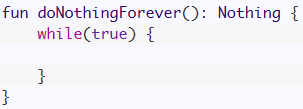
Делегат **vetoable** похож на **observable**, но позволяет контролировать, будет ли значение изменено, предоставляя возможность "отклонить" изменение.



Nothing

**Nothing** является типом, который полезен при объявлении функции, которая не только ничего не возвращает, но и не завершается.

Это может произойти, если функция либо заставляет программу полностью остановиться, генерируя исключение (ошибку) Exception, либо просто продолжается бесконечно без завершения (вечный цикл).

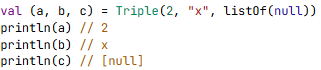


Pair, Triple

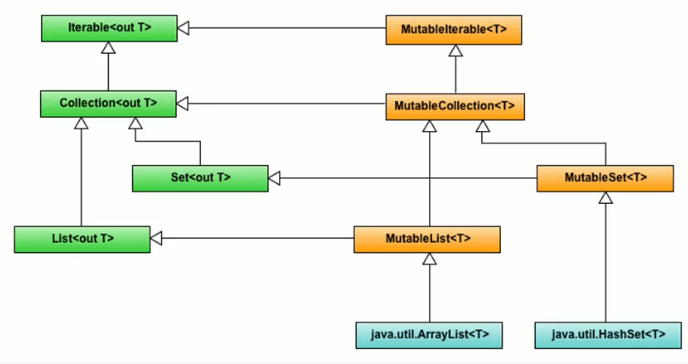
**Pair** - представляет собой общую пару из двух значений.



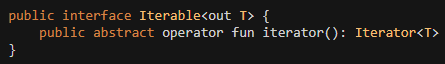
**Triple** - представляет триаду значений.



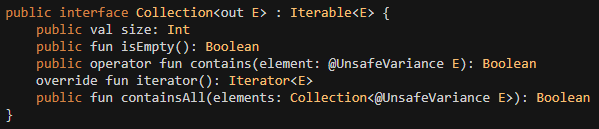
Iterable, Collection



**Iterable** — это интерфейс, который представляет коллекцию, элементы которой могут быть перебраны. Этот интерфейс является частью стандартной библиотеки Kotlin и предназначен для предоставления поддержки итерации по элементам коллекции.



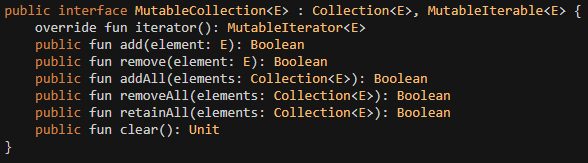
**Collection** – интерфейс является неизменяемым. Другими словами, у вас есть доступ к коллекциям только для чтения. Set and List интерфейсы в Kotlin расширяют этот интерфейс



MutableIterable - этот интерфейс в Kotlin дает нам специализированный изменяемый итератор от родителя Iterable интерфейс.



MutableCollection – это специализированный интерфейс, который позволяет изменять коллекции. Другими словами, операции добавления и удаления могут быть выполнены для данной коллекции. Этот интерфейс расширяет возможности как Collection и MutableIterable интерфейс



Sequence

**Sequence** — это интерфейс, представляющий последовательность элементов, которые могут быть вычислены по мере необходимости. Это позволяет работать с большими объемами данных, не загружая их полностью в память.

**Почему стоит использовать:**

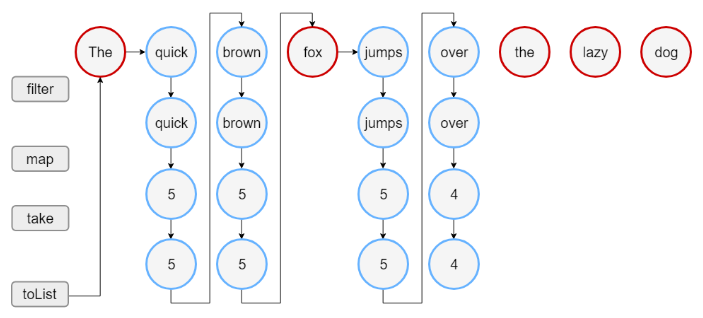
- Экономия памяти

- Ленивая оценка

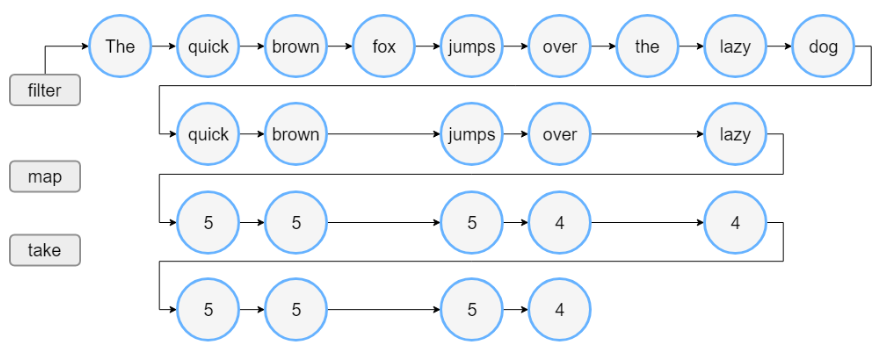
- Поддержка бесконечных последовательностей

В отличие от Iterable Sequence обрабатывает все действия filter,map для каждого объекта по очереди, а не весь список сначала фильтруется, потом мапится.

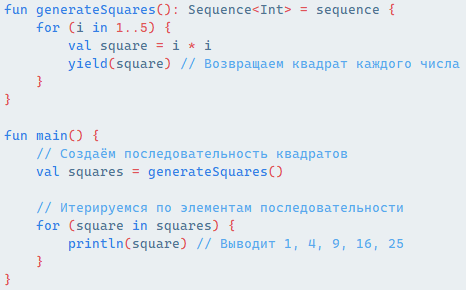
**Sequence:**



**Iterable:**

****

**yield(value: T) -** приостанавливает выполнение функции и возвращает текущее значение.

****

Set, HashSet, LinkedHashSet

**Set** – коллекция с уникальными значениями без определенного порядка внутри.

По умолчанию реализацией Set является **LinkedHashSet**, который сохраняет порядок вставки элементов. Следовательно, функции, которые зависят от порядка элементов, такие как first() или last(), возвращают предсказуемые результаты для таких множеств.

Альтернативная реализация - **HashSet** - не сохраняет порядок элементов, поэтому при вызове функций first() или last() вернётся непредсказуемый результат. Однако HashSet требует меньше памяти для хранения того же количества элементов.

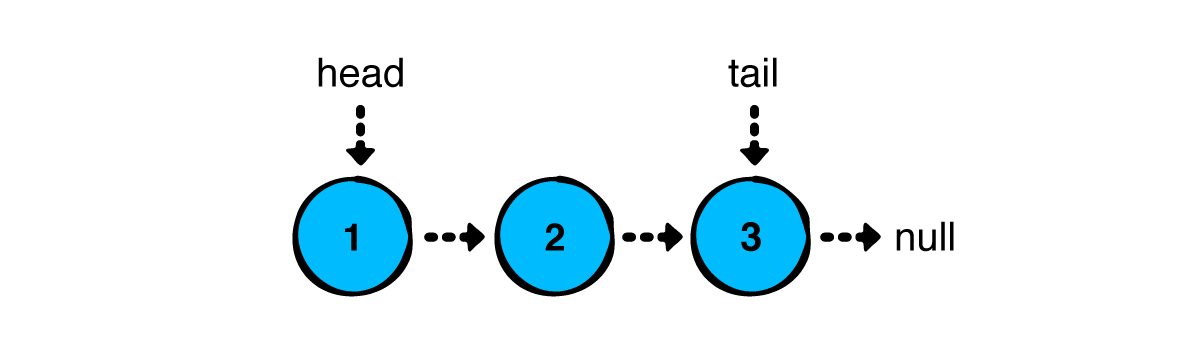
Map, HashMap, LinkedHash

**Map** хранит пары “ключ-значение” (или entries); ключи уникальны, но разные ключи могут иметь одинаковые значения.

По умолчанию реализацией Map является **LinkedHashMap** - сохраняет порядок элементов. Альтернативная реализация - **HashMap** - не сохраняет порядок элементов.

LinkedList

**LinkedList** - это структура данных, которая представляет собой коллекцию узлов, связанных между собой. Каждый узел в связном списке состоит из данных и указателей (или ссылок) на следующие (и в некоторых случаях предыдущие) узлы. Это делает LinkedList отличным инструментом для определённых типов задач, особенно когда требуется частое добавление и удаление элементов.



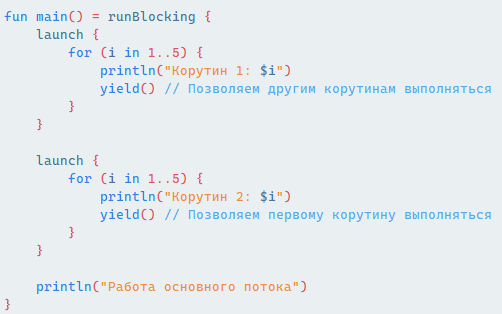
Может добавлять и удалять элементы в начало, конец, середину, а также хранит размер списка.

Yield

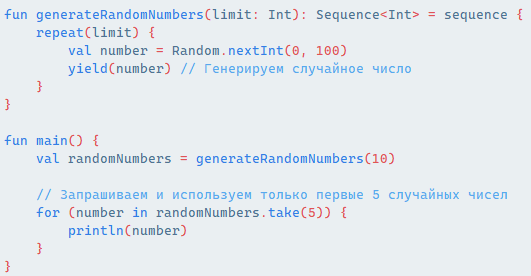
**yield()** — это то, что это функция приостановки, которая используется в контексте сопрограмм Kotlin. Эта функция приостанавливает выполнение одной задачи, передавая управление другой асинхронной задаче.

**yield(value: T)** – нужен для генерации в основном бесконечной последовательности.

**Пример с корутиной:**



**Пример с последовательностью:**



**yield** позволяет нам генерировать новые значения по мере их запроса. Если бы вы использовали обычный массив или список, все значения были бы сгенерированы и хранились бы в памяти. Вы можете запрашивать только часть значений сразу, без необходимости загружать всю коллекцию. Если вам нужны только 3 первых числа, вы можете использовать take(3).

Comparable

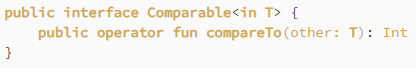
**Comparable** — это интерфейс в Kotlin, который позволяет объектам сравнивать себя с другими объектами того же типа. Он предлагает универсальный механизм для сравнения объектов, что позволяет сортировать их. Этот интерфейс содержит один метод:

- **compareTo(other: T): Int** **-** метод, который сравнивает текущий объект с другим объектом и возвращает:

- Отрицательное число, если текущий объект меньше.

- Ноль, если текущий объект равен.

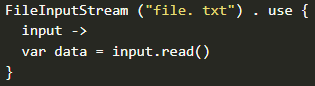
- Положительное число, если текущий объект больше.





Use

**use** принимает литерал функции и определяется как расширение экземпляра closeable. Он закроет ресурс, как и конструкция try-with-resources, после завершения функции, независимо от того, было ли вызвано исключение или нет.



Механизм стирания типов (type erasure)

1. **Проверка соответствия типов на этапе компиляции**:

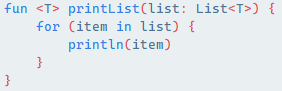
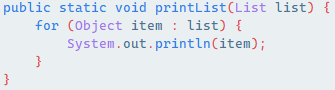
- Когда вы пишете код с использованием обобщений, IDE (интегрированная среда разработки) и компилятор проверяют соответствие типов во время компиляции. Если вы используете обобщенный тип, например, List<T>, компилятор убедится, что вы соблюдаете правила использования типов. Это означает, что тип T должен соответствовать ожидаемым типам везде, где вы его используете.

2. **Стирание типов при компиляции**:

- Во время компиляции типы в обобщениях заменяются на "сырой тип" (raw type) или на верхний предел (upper bound), что обычно является классом Object. Это происходит для всех обобщенных типов, и именно это называется стиранием типов. Например, List<String> будет компилироваться как просто List в байт-коде Kotlin.

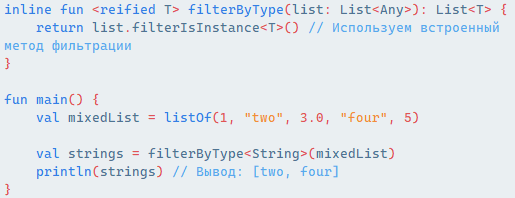
3. **Нет информации о типах во время выполнения**:

- Из-за стирания типов информация о параметрах типа не доступна во время выполнения. Например, в приведенном выше случае List<String> и List<Int> будут оба выглядеть как List в байт-коде.

 ->

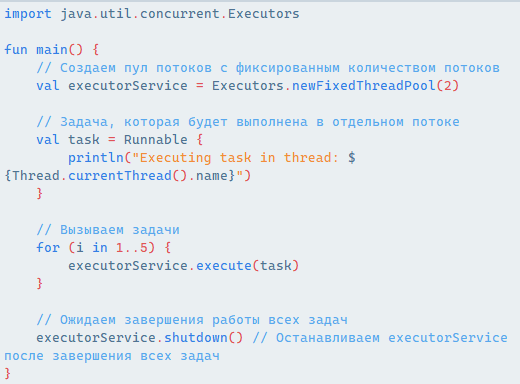
Reified

Использование inline функций с reified параметрами в Kotlin открывает возможности для выполнения различных задач, где необходима информация о типах во время выполнения.



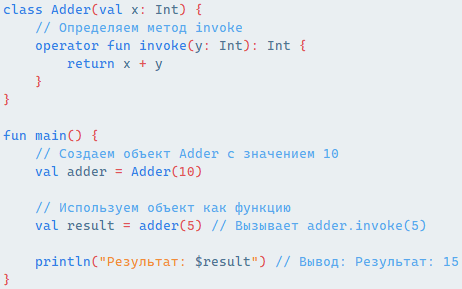
Thread, Executor

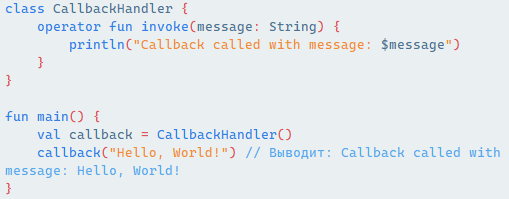
**Executor** - это интерфейс, который предоставляет метод для выполнения задач, таких как Runnable, в потоках. Он упрощает управление многопоточностью и позволяет более эффективно управлять ресурсами.



Invoke

**invoke()** — это специальная функция, которая позволяет объекту быть вызванным как функция.



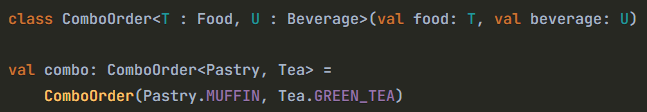


Дженерики Kotlin

Дженерики можно использовать для параметров функций и возвращаемых типов.

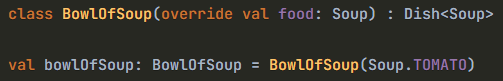


Комбинированный заказ может включать один параметр типа для еды и один параметр типа для напитка.

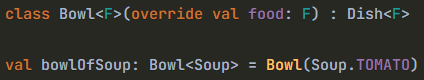


В дополнение к классам, интерфейсы также могут быть обобщенными.

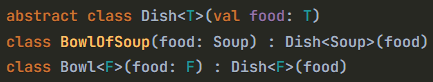




В качестве альтернативы, реализующий класс может сам объявить параметр типа и передать его интерфейсу.



Точно так же абстрактные и открытые классы также могут быть обобщенными, и их расширение работает так, как вы ожидаете.



Универсальные функции





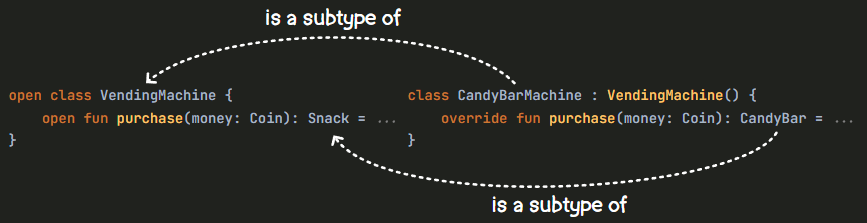
Мы даже можем создавать обобщенные функции расширения.



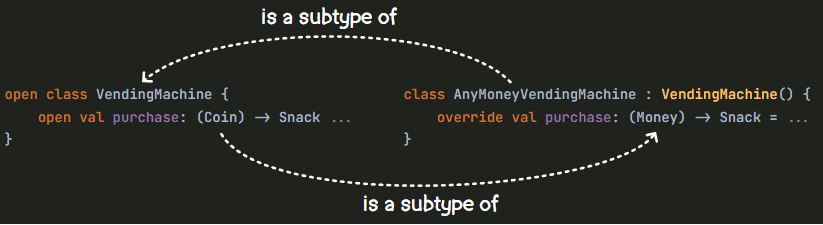
Вариативность дженериков

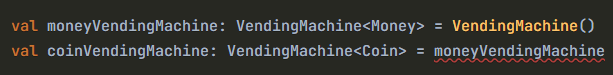
Что-то общее переходящее в конкретное называется **ковариацией.**

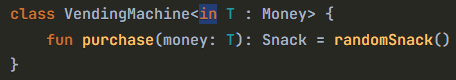
Подводя итог этому разделу, просто помните, что внутри подтипа (например,., CandyBarMachine) Функция может возвращать более определенный тип (например,, CandyBar) чем объявлено в его супертипе, но не в более общем типе. (При ковариации возвращаемое значение).



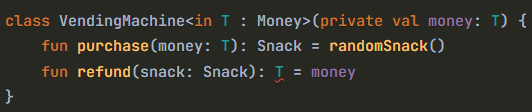
На этот раз мы можем убедиться в этом, когда сделаем более конкретное VendingMachine, Мы можем принять более общий тип параметра в purchase. Стрелки вверху указывают в противоположных направлениях, поэтому такой вид дисперсии называется **контравариантностью**. (При контравариации принимаемое значение).

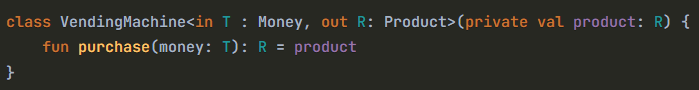


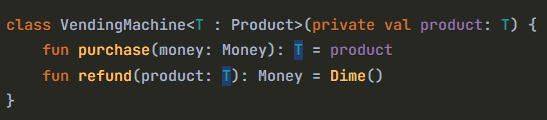
1)

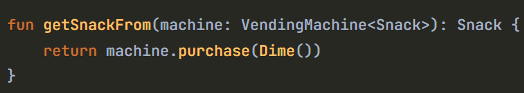
2)

3)

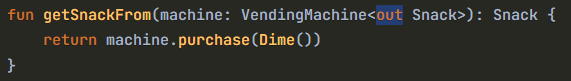




Если в классе не указан in, out  










Если метод не привязан к кнокретным типам, можно поставить \*, тогда он будет работать для каждого типа:  
