# 1 SOLID

## Single Responsibility Principle — принцип единственной ответственности

принцип кажется очень лёгким, но, как правило, его неправильно понимают. Казалось бы, всё понятно — каждый модуль, будь то класс, поле или микросервис — должен отвечать за что-то одно. Тут и кроется крамола. Такой принцип тоже есть, но непосредственно Принцип единственной ответственности — совсем о другом.

Сформулировать его можно так:

*Модуль должен иметь только одну причину для изменения. Или: модуль должен отвечать только за одну заинтересованную группу.*

## *Open-Closed Principle — принцип открытости/закрытости*

*Принцип гласит:*

*Программные сущности должны быть открыты для расширения и закрыты для изменения.*

## Liskov Substitution Principle — принцип подстановки Барбары Лисков

*Принцип подстановки Барбары Лисков можно сформулировать так:*

*Для создания взаимозаменяемых частей эти части должны соответствовать контракту, который позволяет заменять эти части друг другом.*

## Interface Segregation Principle — принцип разделения интерфейсов

*Принцип можно сформулировать так:*

*Необходимо избегать зависимости от того, что не используется.*

## Dependency Inversion Principle — принцип инверсии зависимости

*Код, реализующий высокоуровневую политику, не должен зависеть от кода, реализующего низкоуровневые детали. Зависимости должны быть направлены на абстракции, а не на реализации.*

# 2. Паттерны

## Adapter

Адаптер — структурный шаблон проектирования, предназначенный для организации использования функций объекта, недоступного для модификации, через специально созданный интерфейс.

## Observer

Паттерн Наблюдатель (**Observer**) относится к поведенческим (behavioral) паттернам проектирования. Основная цель данного паттерна это определение зависимости один-ко-многим между объектами, чтобы при изменении состояния объекта, все зависимые от него объекты получали уведомление и обновлялись автоматически

## Singleton

Одиночка – это **шаблон проектирования, в котором данный класс имеет только один экземпляр внутри всего приложения**

## Builder

Паттерн Builder — это **паттерн проектирования, который позволяет поэтапно создавать сложные объекты с помощью четко определенной последовательности действий**. Строительство контролируется объектом-распорядителем (director), которому нужно знать только тип создаваемого объекта.

# 3) **Стек и куча в Java**

* ссылочная переменная **name** типа **String** будет создана в стеке, но сама строка "John" будет храниться в области, называемой [String Pool](https://topjava.ru/blog/rukovodstvo-po-string-pool-v-java) (является частью Кучи);

## 1.Куча

Эта область памяти используется для динамического выделения памяти для объектов и классов JRE во время выполнения. Новые объекты всегда создаются в куче, а ссылки на них хранятся в стеке.

## 2.Стек

Стек работает по схеме LIFO (последним вошел, первым вышел). Всякий раз, когда вызывается новый метод, содержащий примитивные значения или ссылки на объекты, то на вершине стека под них выделяется блок памяти. Из этого можно сделать вывод, что стек хранит значения примитивных переменных, создаваемых в методах, а также ссылки на объекты в куче на которые ссылается метод.

### Основы

#### REST, SOAP, GraphQL

Архитектурный подход для создания API, взаимодействия между серверными и клиентскими приложениями. *Формат обмена* данными: разные (JSON, XML). *Транспорт*: всегда HTTP. Методы HTTP-запроса: GET, POST, PUT(редактировать, обновить), DELETE.

Другие способы построения API-систем, например: gRPC, JSON-RPC, XML-RPC, GraphQL. Подробнее: <https://nuancesprog.ru/p/11310/>

Концепция REST выражается в том, что доступ к данным осуществляется с помощью URI. URI - (Uniform Resource Identifier) унифицированный (единообразный) идентификатор ресурса. По-сути, символьная строка, позволяющая идентифицировать какой-либо ресурс.

#### Http vs Https

- HTTPS не является отдельным протоколом передачи данных, а представляет собой расширение протокола HTTP с надстройкой шифрования;

- передаваемые по протоколу HTTP данные не защищены, HTTPS обеспечивает конфиденциальность информации путем ее шифрования;

- HTTP использует порт 80, HTTPS — порт 443.

#### Различие между методами GET и POST

GET — метод чтения информации. GET-запросы всегда только возвращают данные с сервера, и никогда их не меняют и не удаляют. POST — создание новых записей.

#### PUT, POST, PATH отличия

**POST**

Если Клиент отправляет данные без какого-либо идентификатора, используя метод POST, мы сохраним их и назначим новый идентификатор.

Если Клиент снова отправит те же данные без какого-либо идентификатора, используя метод POST, мы сохраним их и назначим новый идентификатор.

Примечание: Здесь допускается дублирование

**PUT**

Если Клиент отправляет данные с идентификатором, мы проверим, существует ли этот идентификатор. Если идентификатор существует, мы обновим данные, иначе мы создадим его и назначим новый идентификатор.

**PATCH**

Если Клиент отправляет данные с идентификатором, мы проверим, существует ли этот идентификатор. Если идентификатор существует, мы обновим данные, иначе мы создадим исключение.

*Примечание:* В методе Put мы не создаем исключение, если идентификатор не найден. Но в методе Patch мы создаем исключение, если идентификатор не найден.

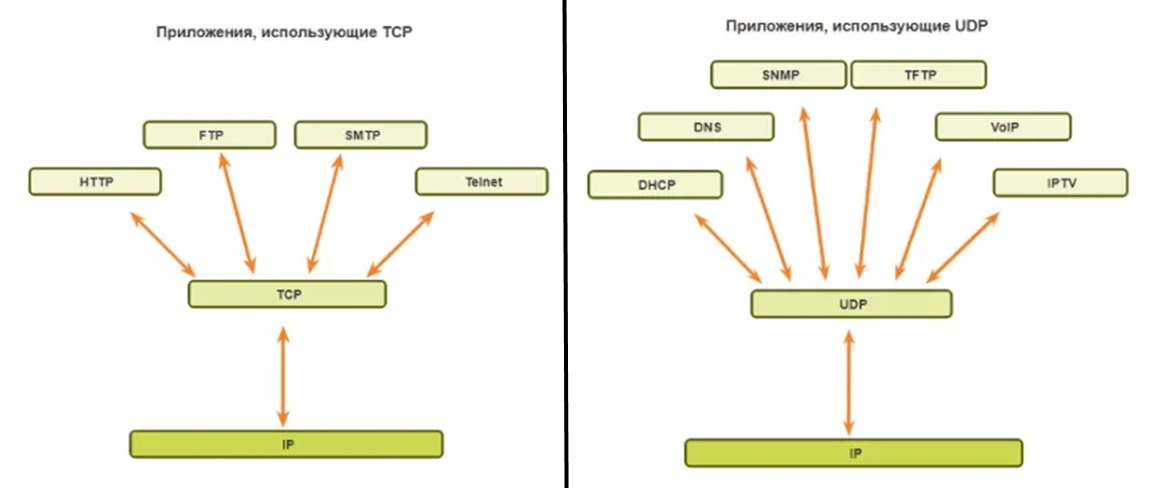
#### Инструменты для работы с запросами

Postman, Hoppscotch.io,

#### TCP / UDP

TCP: надежность, упорядочена, низкая скорость. UDP - наоборот.

TCP применяется там, где требуется точная и подтверждаемая передача данных – например, отправка фотографий, или переписка между пользователями. UDP, в свою очередь, нужен для общения в голосовом формате, или при передаче потокового видео, например, с веб-камер или IP-камер.



#### ArrayList vs LinkedList. В каком случае что лучше использовать?

ArrayList - это список на основе массива. LinkedList - связанный список на основе элементов и связи между ними.

*ArrayList* следует использовать, когда в приоритете доступ по индексу, так как эти операции выполняются за константное время. Добавление в конец списка в среднем тоже выполняется за константное время. Кроме того в ArrayList нет дополнительных расходов на хранение связки между элементами. Минусы в скорости вставки/удаления элементов находящихся не в конце списка, так как при этой операции все элементы правее добавляемого/удаляемого сдвигаются.

*LinkedList* удобен когда важнее быстродействие операций вставки/удаления, которые в LinkedList выполняются за константное время. Операции доступа по индексу производятся перебором с начала или конца (смотря что ближе) до нужного элемента. Дополнительные затраты на хранение связки между элементами.

Одним словом - если часто вставляете/удаляете - выбирайте в пользу LinkedList, в противном случае ArrayList

#### Как работает HashMap

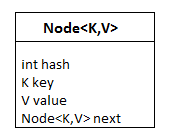
Структура данных — Map (“ассоциативный массив”). HashMap (хеш-таблица) реализует интерфейс ассоциативного массива. Абстрактная модель «ключ – значение» или entry обеспечивает очень быструю вставку и поиск. Ключ в HashMap всегда является уникальным. Пустой HashMap условно рассчитан только на 16 элементов. Если 2 объекта имеют одинаковый хэш-код, то они будут храниться в одной ячейке в виде связного списка (LinkedList).

hashCode() используется для получения хэш кода ключа. Реализация по умолчанию hashCode() в OpenJDK не имеет ничего общего с адресом памяти, как это порой принято считать.

В HashMap хеш-таблица реализована на основе массива (а если точнее — динамического, так как таблица может расширяться) односвязных списков. Элементы массива (ячейки) еще называются корзинами «buckets».

Вложенный класс внутри HashMap - Node (где-то видел Entry), который имеет следующие поля:

* final int hash — хеш текущего элемента, который мы получаем в результате хеширования ключа;
* final K key — ключ текущего элемента. Именно сюда записывается то, что вы указываете первым объектом в методе put();
* V value — значение текущего элемента. А сюда записывается то, что вы указываете вторым объектом в методе put();
* Node < K, V> next — ссылка на следующий узел в пределах одной корзины. Список же связный, поэтому ему нужна ссылка не следующий узел, если такой имеется.



Хеш-отображение не гарантирует порядок расположения своих элементов. Следовательно, порядок, в котором элементы вводятся в хеш-отображение, не обязательно соответствует тому порядку, в котором они извлекаются итератором.

#### Виды сортировок

**Bubble Sort** (Сортировка пузырьком, Простейшая сортировка). **Сложность** O(n ^ 2).

**Быстрая сортировка** (Quick Sort). **Сложность** O(n log n).

Этот алгоритм состоит из трёх шагов. Сначала из массива нужно выбрать один элемент - опорный. Затем другие элементы в массиве перераспределяют так, чтобы элементы меньше опорного оказались до него, а большие или равные — после. А дальше рекурсивно применяют первые два шага к подмассивам справа и слева от опорного значения. Алгоритм не занимает дополнительного пространства, вся сортировка происходит «на месте», отсутствуют затратные вызовы распределения

**Глупая сортировка**. **Сложность** O().

**Шейкерная сортировка** (сортировка перемешиванием, коктейльная сортировка). **Сложность** O(n ^ 2).

Чётно-нечётная сортировка. **Сложность** O().

Сортировка расчёской. **Сложность** O(n log n).

**Сортировка выбором** (Selection Sort). **Сложность** O(n ^ 2).

неустойчива — одинаковые элементы могут изменить свой порядок

**Сортировка вставками** (Insertion Sort). **Сложность** O(n ^ 2).

устойчива — одинаковые элементы не изменят свой порядок

Челночная сортировка (Shuttle Sort). **Сложность** O().

**Сортировка Шелла**. **Сложность** O().

**Cортировка слиянием** (merge sort). **Сложность** O(n log n).

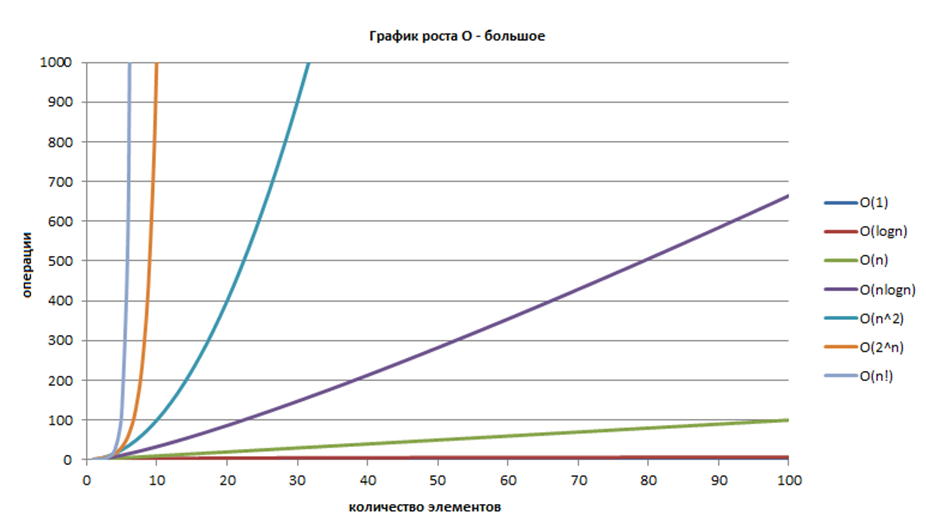
является основным для сортировки массивов ссылочных типов;

неустойчива — одинаковые элементы могут изменить свой порядок;

Массив разбивается на две примерно равные части и каждая из них сортируется по отдельности. Затем два отсортированных подмассива сливаются в один (рекурсивно).

##### График сложности по времени.

Легенда: сверху быстрее - от О(1) -> O(n!)



#### Сортировки в Java

В стандартном пакете java — Java Collections Framework (на примере JDK 7) — основные:

> Два алгоритма сортировки: быстрая сортировка, сортировка слиянием.

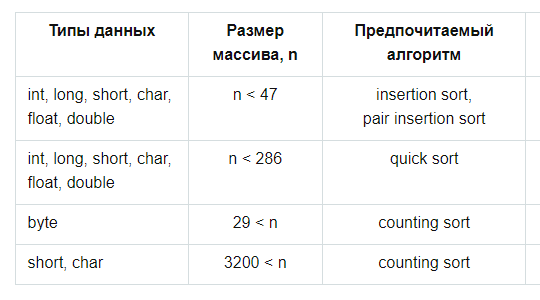
> Один алгоритм поиска: бинарный поиск

##### Arrays.sort

**Массивы примитивных типов**

Основной алгоритм сортировки для массивов примитивных типов в Arrays — *быстрая сортировка*. Дополнительные алгоритмы: *сортировка вставками*, “обычная” *быстрая сортировка* (с одной опорной точкой), *сортировка подсчетом*.

Инженеры Oracle эмпирическим путем вычислили оптимальные размерности массивов для задействования каждого дополнительного алгоритма сортировки.



источник: <https://habr.com/ru/post/344288/>

**Массивы ссылочных типов**

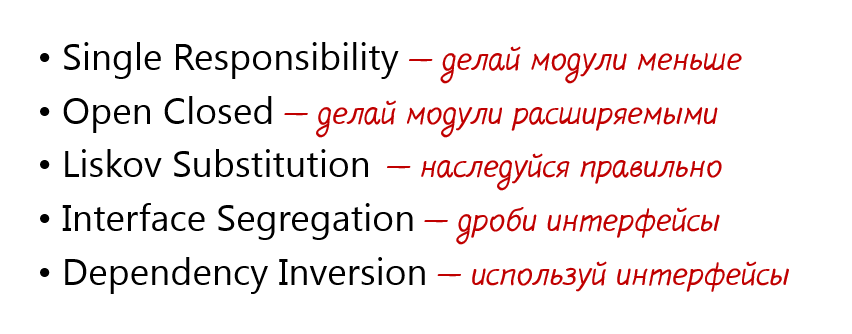
Для массивов ссылочных типов в качестве основного предусмотрен *алгоритм сортировки слиянием*. Дополнительный алгоритм: бинарная сортировка вставками, если длина массива меньше 32.

##### Collections.sort

В Collections сортировка предусмотрена только для списков. Интересно, что сначала производится преобразование списка в массив с последующим применением сортировки слиянием для массивов ссылочных типов. Сортировка осуществляется вызовом метода Arrays.sort.

#### SOLID, DRY

**SOLID** — 5 базовых принципов ООП, предложенные Робертом Мартином.



S — Single-responsibility principle / Принцип единственной ответственности

один класс отвечает за один функционал. Проектируя класс, попробуйте сгруппировать его методы по тем сущностям и процессам, с которыми они работают. Если легко можно выделить несколько групп, то стоит подумать о выделении отдельных сущностей и переносе этих методов в них.

O — Open–closed principle / Принцип открытости-закрытости

сущности должны быть открыты к расширению, но закрыты — к изменению. То есть, например, класс должен с самого начала проектироваться так, что любое дальнейшее развитие функционала не потребует изменения кода самого этого класса, но позволит расширить функционал с помощью подклассов и т.п.

L — Liskov substitution principle / Принцип подстановки Лисков

сущность, использующая объект, который реализует определенный интерфейс, должна иметь возможность использовать другой объект с тем же интерфейсом, даже «не зная» о факте подмены. То есть программируем к интерфейсу, а не к его конкретной реализации.

I — Interface segregation principle / Принцип разделения интерфейсов

клиентский код не должен зависеть от методов, которые не использует. То есть нужно избегать интерфейсов, имеющих много методов — не все из них будут востребованы клиентами. Лучше иметь несколько интерфейсов с небольшим числом методов, чем один — с большим.

D — Dependency inversion principle / Принцип инверсии зависимостей

модули высшего порядка не должны зависеть от модулей низшего порядка, и те, и другие должны зависеть от абстракций; детали должны зависеть от абстракций, но не наоборот. В общем, класс не должен иметь жестко прописанных в нем зависимостей от объектов других классов, например, через оператор new. Вместо этого зависимости должны быть вынесены из класса, чтобы уменьшить степень связанности кода. Этот принцип реализуется несколькими способами, мы уже писали про самый популярный — внедрение зависимости.

**DRY** — Don’t Repeat Yourself (не повторяйтесь) — не нужно «изобретать велосипед». Любой функционал в коде должен быть реализован ровно один раз, не говоря уже о том, что copy-paste-кода вообще не должно быть.

#### Принципы ООП

**Абстракция** — означает выделение главных, наиболее значимых характеристик предмета и наоборот — отбрасывание второстепенных, незначительных (отделение концепции от ее экземпляра);

**Полиморфизм** — это возможность работать с несколькими типами так, будто это один и тот же тип. При этом поведение объектов будет разным в зависимости от типа, к которому они принадлежат (реализация задач одной и той же идеи разными способами). Пример: перегрузка методов, переопределение методов;

**Наследование** — механизм, который позволяет описать новый класс на основе существующего (родительского). При этом свойства и функциональность родительского класса заимствуются новым классом.;

**Инкапсуляция** — ограничение доступа к данным и возможностям их изменения (размещение одного объекта или класса внутри другого для разграничения доступа к ним). В языке Java инкапсуляция реализована с помощью системы классов, которые позволяют собрать информацию об объекте в одном месте; пакетов, которые группируют классы по какому-либо критерию, и модификаторов доступа, которыми можно пометить весь класс или его поле или метод.

#### Регулярные выражения

Regular Expressions (regexp, regex) — это механизм для поиска и замены текста; строго говоря, это специальный язык для описания шаблонов строк.

*Общее (по умолчанию)*: регистрозависимые; первое вхождение;

##### Метасимволы

заменяют собой конкретный диапазон значений. Например:

| Символ | Эквивалент | Пояснение |
| --- | --- | --- |
| \d | [0-9] | Цифровой символ |
| \D | [^0-9] | Нецифровой символ |
| \s | [ \f\n\r\t\v] | Пробельный символ |
| \w | [[:word:]] | Буквенный или цифровой символ или знак подчёркивания |

##### Спецсимволы

нужно экранировать добавлением \ перед спецсимволом (список: [ ] \ / ^ $ . | ? \* + ( ) { } );

##### Квантификаторы

указывают количество повторений текста; применяется только к символу, который стоит перед ним.

Число повторений:

? Ноль или одно

\* Ноль или более

+ Один или более

Виды квантификаций:

> ленивая (добавить знак вопроса),

> жадная (максимально длинная строка из возможных),

> ревнивая (справа символа +) - запрет возвращаться к предыдущим шагам поиска, искать в оставшейся части регулярного выражения.

##### Пример

Как найти два слова разделенных любым количеством символов в строке?

#### Git

##### Разница между pull и fetch

**git pull** — это, по сути, команда git fetch, после которой сразу же следует git merge. Выполнение git pull, как правило, извлекает (fetch) данные с сервера, с которого вы изначально склонировали, и автоматически пытается слить (merge) их с кодом, над которым вы в данный момент работаете.

**git fetch** получает изменения с сервера и сохраняет их в каталог refs/remotes/. Это никак не влияет на локальные ветки и текущие изменения. А git merge уже вливает все эти изменения в локальную копию. Команда fetch забирает данные в ваш локальный репозиторий, но не сливает их с какими-либо вашими наработками и не модифицирует то, над чем вы работаете в данный момент.

Git pull – скачать коммиты и применить к рабочей директории.

Git fetch – скачать коммиты не применяя их к рабочей директории.

#### Разница между merge and rebase

#### Виды тестов

#### Зачем нужен CI/CD

### Java

#### Абстрактный класс vs. Интерфейс

**>** Интерфейс описывает только поведение (методы) объекта, а вот состояний (полей) у него нет (кроме public static final), в то время как у абстрактного класса они могут быть. **>** Абстрактный класс связывает между собой и объединяет классы, имеющие очень близкую связь. В то же время, один и тот же интерфейс могут реализовать классы, у которых вообще нет ничего общего.

**>** Абстрактный класс наследуется (etxends), а интерфейс — реализуется (implements). Мы можем наследовать только один класс, а реализовать интерфейсов — сколько угодно. Интерфейс может наследовать (extends) другой интерфейс/интерфейсы.

##### Абстрактный класс (АС)

- в АС можно объявить и определить конструкторы; компилятор по умолчанию создаёт конструктор без аргументов;

- АС могут реализовывать интерфейсы;

- не может быть final;

- нельзя создать экземпляр АС;

##### Абстрактный метод

Это метод без тела. Вы просто объявляете метод, не определяя его, с использованием ключевого слова abstract в объявлении метода. Все объявленные внутри интерфейса в языке Java методы – по умолчанию абстрактные.

##### Интерфейс

Интерфейс — это ссылочный тип в Java. Он схож с классом. Это совокупность абстрактных методов. Класс реализует интерфейс, таким образом наследуя абстрактные методы интерфейса.

Вместе с абстрактными методами интерфейс в Java может содержать константы, обычные методы, статические методы и вложенные типы. Тела методов существуют только для обычных методов и статических методов.

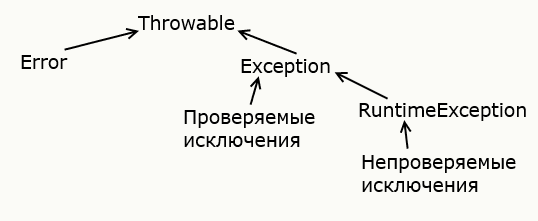
#### Виды исключений

У всех исключений есть общий класс-предок Throwable. От него происходят две большие группы — исключения (Exception) и ошибки (Error).

**Exceptions** — это, собственно, исключения: исключительная, незапланированная ситуация, которая произошла при работе программы.

##### checked / unchecked

Все исключения делятся на 2 вида — проверяемые (checked) и непроверяемые (unchecked).



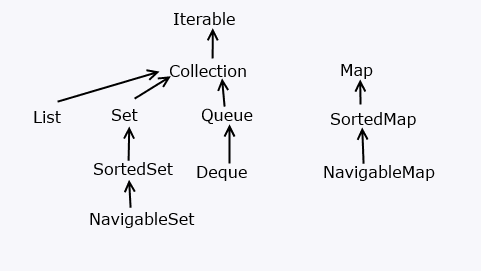
**Checked** - компилятор Java знает о самых распространенных исключениях, и знает, в каких ситуациях они могут возникнуть.

**Unchecked** - компилятор не проверяет факт обработки таких исключений и их можно не указывать вместе с оператором throws в объявлении метода. Чаще всего являются следствием ошибок программиста.

##### собственные

создать собственный класс и унаследоваться от класса Exception

#### Виды коллекций. Что от чего наследуется?



**Collection** — базовый интерфейс для всех коллекций и других интерфейсов коллекций. Коллекции могут хранить любые ссылочные типы данных.

#### Вопросы по коллекциям

##### ArrayList и LinkedList

**ArrayList** реализован внутри в виде обычного массива. Поэтому при вставке элемента в середину, приходится сначала сдвигать на один все элементы после него, а уже затем в освободившееся место вставлять новый элемент. Зато в нем быстро реализованы взятие и изменение элемента – операции get, set, так как в них мы просто обращаемся к соответствующему элементу массива.

**LinkedList** реализован в виде связного списка: набора отдельных элементов, каждый из которых хранит ссылку на следующий и предыдущий элементы. Чтобы вставить элемент в середину такого списка, достаточно поменять ссылки его будущих соседей. Операции set и get тут реализованы очень медленно.

LinkedList требует больше памяти для хранения такого же количества элементов, потому что кроме самого элемента хранятся еще указатели на следующий и предыдущий элементы списка, тогда как в ArrayList элементы просто идут по порядку

##### Массив vs. ArrayList

Если коротко, то Oracle пишет – используйте ArrayList вместо массивов. Если ответить на этот вопрос нужно по-другому, то можно сказать следующее: массивы могут быть быстрее и кушать меньше памяти. Списки теряют в производительности из-за возможности автоматического увеличения размера и сопутствующих проверок. Плюс к этому, что размер списка увеличивается не на 1, а на большее кол-во элементов (+15)\*. Также доступ к [10] в массиве может быть быстрее чем вызов get(10) у списка.

\*Читатель прислал комментарий “У ArrayList увеличение происходит в 1.5 раза”.

##### В Map положить два значения с одинаковым ключом?

Последнее значение перезапишет предыдущее.

##### Почему Map не наследуется от Collection?

Они не совместимы, т.к. созданы для различных структур данных. Map использует пару ключ-значение.

##### Iterator и Enumeration

Enumeration в два раза быстрее Iterator и использует меньше памяти. Iterator потокобезопасен, т.к. не позволяет другим потокам модифицировать коллекцию при переборе. Enumeration можно использовать только для read-only коллекций.

##### Iterator vs. for (цикл)

#### Примитивы vs. Классы

Настоящая разница в том, что примитивные типы не являются ссылочными типами.

В системе типов Java ссылочные типы имеют общий корень, а примитивные типы - нет: все ссылочные типы являются подтипами Object. Это проблема, если вы хотите создать общие полезные структуры данных. Подумайте, как бы вы реализовали структуру данных «динамический массив», такую как ArrayList или Vector. Вы можете использовать массив Object[] для хранения элементов, и это будет работать для всех ссылочных типов.

Для решения этой проблемы были созданы классы-оболочки. Теперь вместо того, чтобы требовать 8 отдельных реализаций динамического массива (1 для ссылочных типов и 7 для примитивных типов), вы можете заключить каждое примитивное значение в объект и просто использовать реализацию для ссылочных типов.

В идеале, в объектно-ориентированном языке - все сущности должны быть объектами. Отец основатель Java - Джеймс Гослинг это прекрасно понимал, но он также и будучи реалистом понимал, что если все сущности сделать объектами язык потеряет в скорости, посему было решено сохранить понятие примитивных типов: int, byte и проч. и дополнить библиотеку объектов их объектными аналогами: Integer, Byte и т.д. Естественно была предусмотрена конверсия между примитивными типами и их объектными аналогами.

#### Автоупаковка

(автоматическое преобразование из примитивных типов данных к ссылочным и наоборот)

Начиная с Java 5 конверсия производится автоматически (называется boxing/unboxing).

Во многих случаях при присвоении или передаче значения Java конвертирует примитивные типы в их объектные обёртки и обратно автоматически.

#### Циклы (while, do while, for i, foreach)

#### String, StringBuffer, StringBuilder

String, StringBuffer (синхронизирован), StringBuilder (не синхронизирован)

#### equals, ==, compareTo

Оператор == сравнивает ссылки.

Метод equals сравнивает значения.

Метод сравнения compareTo() применяется, если надо определить лексикографический порядок строк. Он выполняет сравнение значения char, действуя аналогично equals(). Отличается результатом - вернёт int (0 - если равны; больше или меньше нуля если первая строка больше/меньше.

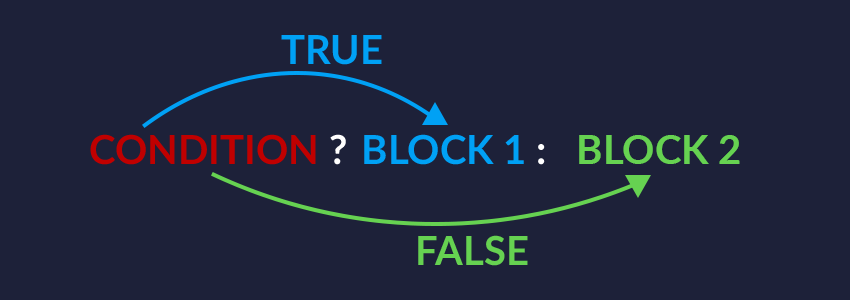
#### Enum

#### Модификаторы доступа

#### final

Ключевое слово final означает, что класс на вершине иерархии, и у него не может быть наследников. Компилятор Java выдаст ошибку, если сделать класс одновременно abstract и final.

Что такое тернарный оператор выбора?



Для чего нужен сборщик мусора?

* Если переменная ссылочного типа, которая ссылается на объект, установлена в положение "0", объект подлежит утилизации, в том случае, если на него нет других ссылок.
* Если переменная ссылочного типа, которая ссылается на объект, создана для ссылки на другой объект, объект подлежит утилизации, в том случае, если на него нет других ссылок.
* Объекты, созданные локально в методе, подлежат утилизации, когда метод завершает работу, если только они не экспортируются из этого метода (т.е, возвращаются или генерируются как исключение).
* Объекты, которые ссылаются друг на друга, могут подлежать утилизации, если ни один из них не доступен живому потоку.

Что такое «пул строк»?

Как подсказывает название, пул строк – это набор строк, который хранится в памяти Java heap. Мы знаем, что String это специальный класс в Java, и мы можем создавать объекты этого класса, используя оператор new точно так же, как и создавать объекты, предоставляя значение строки в двойных кавычках.

Диаграмма ниже объясняет, как пул строк размещается в памяти Java heap и что происходит, когда мы используем различные способы создания строк.

Пул строк возможен исключительно благодаря неизменяемости строк в Java и реализации идеи интернирования строк. Пул строк также является примером паттерна Приспособленец (Flyweight).

Пул строк помогает экономить большой объем памяти, но с другой стороны создание строки занимает больше времени.

Когда мы используем двойные кавычки для создания строки, сначала ищется строка в пуле с таким же значением, если находится, то просто возвращается ссылка, иначе создается новая строка в пуле, а затем возвращается ссылка. Тем не менее, когда мы используем оператор new, мы принуждаем класс String создать новый объект строки, а затем мы можем использовать метод intern() для того, чтобы поместить строку в пул, или получить из пула ссылку на другой объектString с таким же значением.

Ниже приведен пример, показывающий работу пула строк.

public class StringPool {

public static void main(String[] args) {

String s1 = "Cat";

String s2 = "Cat";

String s3 = new String("Cat");

System.out.println("s1 == s2 :"+(s1==s2));

System.out.println("s1 == s3 :"+(s1==s3));

}

}

Программа выведет следующее:

s1 == s2 :true

s1 == s3 :false

Serializable vs Parcelable

### Контракт между equals и hashcode

## **Почему equals и hashCode в Java переопределяются вместе**

Сперва производится сравнение по хешу, чтобы понять, совпадают ли объекты, а только после подключается equals, чтобы определить, совпадают ли значения полей объекта.

Рассмотрим два сценария.

### **1. equals есть, hashCode нет**

С точки зрения метода equals два объекта будут логически равны, но по hashCode они не будут иметь ничего общего. Таким образом, помещая некий объект в хэш-таблицу, мы рискуем не получить его обратно по ключу:

Map<Point, String> m = new HashMap<>();

m.put(new Point(1, 1), "Point A");

//pointName == null

String pointName = m.get(new Point(1, 1));

### **2. hashCode есть, equals нет**

Метод equals по умолчанию сравнивает указатели на объекты, определяя, ссылаются ли они на один и тот же объект. Следовательно, пример из предыдущего пункта по идее должен выполняться. Но мы по-прежнему не сможем найти наш объект в хэш-таблице.

Для успешного поиска объекта в хэш-таблице помимо сравнения хэш-значений ключа используется также определение логического равенства ключа с искомым объектом.

### Что такое Generic?

Дженерики (обобщения) — это особые средства языка Java для реализации обобщённого программирования: особого подхода к описанию данных и алгоритмов, позволяющего работать с различными типами данных без изменения их описания.

На сайте Oracle дженерикам посвящён отдельный tutorial: "[Lesson: Generics](https://docs.oracle.com/javase/tutorial/java/generics/index.html)".

### Устройство памяти в Java

**class** **Person** {

**int** id;

String name;

**public** **Person**(**int** id, String name) {

**this**.id = id;

**this**.name = name;

}

}

**public** **class** **PersonBuilder** {

**private** **static** Person **buildPerson**(**int** id, String name) {

**return** **new** Person(id, name);

}

**public** **static** **void** **main**(String[] args) {

**int** id = 23;

String name = "John";

Person person = **null**;

person = buildPerson(id, name);

}

}

**Рассмотрим выполнение кода по шагам:**

1. До начала выполнения метода *main()*, в стеке будет выделено пространство для хранения примитивов и ссылок этого метода:

* примитивное значение ***id*** типа ***int*** будет храниться непосредственно в стеке;
* ссылочная переменная **name** типа **String** будет создана в стеке, но сама строка "John" будет храниться в области, называемой [String Pool](https://topjava.ru/blog/rukovodstvo-po-string-pool-v-java) (является частью Кучи);
* ссылочная переменная ***person*** типа ***Person*** будет также создана в памяти стека, но будет указывать на объект, расположенный в куче;

2. Для вызова конструктора с параметрами *Person (int, String)* из метода *main()* в стеке, поверх предыдущего вызова метода *main(),* будет выделен блок памяти*,* который будет хранить:

* ***this*** — ссылка на текущий объект;
* примитивное значение ***id ;***
* ссылочную переменную ***name*** типа ***String***, которая указывает на объект строки из пула строк;

3. В методе ***main*** дополнительно вызывается метод ***buildPerson*** для которого будет выделен блок памяти в стеке поверх предыдущего вызова. Этот блок снова сохранит переменные способом, описанным выше.

4. Для вновь созданного объекта ***person*** типа Person все переменные будут сохранены в памяти кучи.

Как работает GarbageCollector

Какие особенности памяти для работы со строками

Если создать строку через new где она хранится, как поместить такую строку в пул строк

https://topjava.ru/blog/rukovodstvo-po-string-pool-v-java

#### Рассказать про Handler, Loop, Thread, Message

По своей сути Looper, Handler и MessageQueue реализуют шаблон *producer/consumer*. Тред-продюсер отправляет сообщения через Handler в коллекцию-буфер, реализованную классом MessageQueue. Тред-потребитель блокирован с помощью класса Looper, который ожидает и принимает сообщения из MessageQueue и передает их на обработку хэндлеру.

Looper трудится, обслуживая looperMessageQueue для текущего потока. MessageQueue это очередь которая содержит в себе задачи, называющиеся сообщениями, которые нужно обработать. Looper перемещается по этой очереди и отправляет сообщения в соответствующие обработчики для выполнения. Любой поток может иметь единственный уникальлный Looper, это ограничение достигается с помощью концепции ThreadLocal хранилища. Связка Looper+MessageQueue выглядит как конвейер с коробками. Задачи в очередь помещают Handler‘ы.

#### Looper

Looper запускает цикл обработки сообщений, связанный с потоком. Поток работает, пока связанный с ним лупер не будет остановлен.

Для создания лупера, вызывается статический метод Looper.prepare(). Созданный лупер будет связан с потоком, в котором вызван этот метод.

Для старта лупера используется статический метод Looper.loop(). Между вызовами методов prepare() и loop() обычно создается Handler, который будет обрабатывать сообщения, приходящие в MessageQueue лупера.

Для остановки лупера используется метод quit() или quitSafely(). Разница между этими методами в том, что quit() останавливает лупер незамедлительно, а quitSafely() завершает обработку сообщений, которые уже добавлены в очередь.

Лупер UI потока можно получить методом Looper.getMainLooper().



#### Handler

Handler – это класс, который используется для работы с очередью сообщений, связанной с потоком. Хэндлер позволяет отправлять сообщения в другие потоки с задержкой или без, а также обрабатывать полученные сообщения.

Хэндлер всегда связан с лупером, который в свою очередь связан с каким-либо потоком.

#### MessageQueue

Это очередь сообщение. MessageQueue не имеет публичных методов для работы с сообщениями. Работа с очередью выполняется через методы класса Handler.

MessageQueue хранит объекты класса Message. В объекте message можно передать Bundle, который добавляется и получается методами setData() и getData().

Помимо data-bundle в message есть два публичных поля типа int: arg1 и arg2, а также публичное поле obj типа Object. Эти поля используются в тех случаях, когда нужно передать целочисленное значение или произвольный объект, и нет необходимости создавать bundle.

Также класс Message имеет публичное целочисленное поле what, которое по смыслу подобно параметру requestCode в методе startActivityForResult() и используется для определения о чем это сообщение.

#### Различие Thread и HandlerThread

Единственное существенное отличие между HandlerThread и Thread заключается в том что первый содержит внутри себя Looper, Thread и MessageQueue.

#### Thread и Runnable

*Thread* - это абстракция над физическим потоком.

*Runnable* - это абстракция над выполняемой задачей.

Плюс использования Runnable состоит в том, что это позволяет логически отделить выполнение задачи от логики управления потоками.

Runnable - это интерфейс описывающий метод Run, с помощью которого Вы можете передать в другой класс свой код для выполнения.

Thread - это класс, который реализует поток и использует интерфейс Runnable для встраивания Вашего кода в поток.

#### AsyncTask, какие особенности

Типы ссылок в Java (Weak и т.д.)

Как устроен ArrayList, что происходит при добавлении нового элемента, когда внутренний массив заполнен. Какая функция используется для расширения массива

#### Разница между ArrayList и LinkedList

| **Описание** | **Операция** | **ArrayList** | **LinkedList** |
| --- | --- | --- | --- |
| Взятие элемента | get | Быстро | Медленно |
| Присваивание элемента | set | Быстро | Медленно |
| Добавление элемента | add | Быстро | Быстро |
| Вставка элемента | add(i, value) | Медленно | Быстро |
| Удаление элемента | remove | Медленно | Быстро |

Ещё у *LinkedList* дополнительные затраты на хранение связки между элементами.

#### Как устроен LinkedList

Он реализован в виде связного списка: набора отдельных элементов, каждый из которых хранит ссылку на следующий и предыдущий элементы. Чтобы вставить элемент в середину такого списка, достаточно поменять ссылки его будущих соседей.

LinkedList удобен когда важнее быстродействие операций вставки/удаления, которые в LinkedList выполняются за константное время. Операции доступа по индексу производятся перебором с начала или конца (смотря что ближе) до нужного элемента.

Есть ли в самом классе LinkedList информация и длине списка

Iterator

### Kotlin

#### val и var

#### lateinit var

Правила использования модификатора lateinit:

* Переменная обязательно должна быть изменяемой (var).
* Не должна относиться к примитивным типам (Int, Double, Float и т.д).
* Не должна иметь собственных геттеров/сеттеров.
* свойство может быть объявлено только внутри тела класса (не в основном конструкторе)

Если вы обратитесь к переменной до её инициализации, то получите исключение "lateinit property has not been initialized".

Для проверки факта инициализации переменной вызывайте метод *isInitialized()*:

if (::catName.isInitialized) {

Log.d("Kot", "Hi, $catName")

}

#### const / val

Ключевое слово const используется для инициализации во время компиляции.

Ключевое слово val используется для инициализации во время выполнения.

#### Диапазоны

Отличия в от джавы в обработке исключений (Kotlin не требует перечислять контролируемые исключения)

Какие классы коллекций используются?

Отличие интерфейсов от Java7 (могут включать свойства и реализации методов по умолчанию)

Какой модификатор видимости по умолчанию?

Лямбда-выражения. Что это и как применяется?

let, apply, with

#### Типы с поддержкой Null и без нее

В Kotlin null может формировать nullable-типы (произносится как «ну́ллабл»). Они обозначаются добавлением знака ? в конце типа. Например, String? — это тип String + null. То есть значение может быть строковым, а может быть null. В Java такие дополнения не нужны — там любой объект может быть null, и это приводит нас к одному из преимуществ языка Kotlin перед Java — *null safety*.

Если в конце типа не стоит ? — вы гарантируете, что объект никогда не может быть null. Это помогает выявить возможные ошибки, связанные с NPE, ещё на этапе написания кода.

#### Оператор безопасного вызова

Инструмент для работы с nullable-типами. Выглядит как вопросительный знак с последующей точкой:

val name: String? = "John"

val nameLength: Int? = name?.length

Производит проверку на null перед вызовом метода. Если значение переменной равно null, то вместо вызова исключения это выражение просто вернёт null.

#### оператор «Элвис» (?:)

Ещё один удобный оператор для работы с null safety. «Элвис» записывают как ?: (знак вопроса с последующим двоеточием).

val nameLength: Int = name?.length ?: 1

Принцип работы следующий: если выражение слева вернёт не null, то мы получим длину имени; если же null, то вернётся значение выражения справа от оператора.

#### Any, Unit, Nothing

##### Any

Класс Any находится на вершине иерархии — все классы в Kotlin являются наследниками Any. Any — это аналог Object в Java, но с меньшим количеством методов:

public open class Any {

public open operator fun equals(other: Any?): Boolean

public open fun hashCode(): Int

public open fun toString(): String

}

В java.lang.Object одиннадцать методов, и пять из них касаются многопоточности. Несмотря на меньшее количество методов, при компиляции в Java-код у любого класса появятся недостающие — тут можно быть спокойными.

##### Unit

Unit - аналог void. В стандартной библиотеке Kotlin Unit определён как объект, наследуемый от Any и содержащий единственный метод, переопределяющий toString():

public object Unit {

override fun toString() = "kotlin.Unit"

}

Обратите внимание на ключевое слово object. Это значит, что Unit является синглтоном. Unit ничего не возвращает, а метод toString всегда будет возвращать “kotlin.Unit”. При компиляции в java-код Unit всегда будет превращаться в void.

##### Nothing

Nothing используется в качестве типа возвращаемого значения для функций, которые в обычном режиме не завершаются.

Nothing — класс, который является наследником любого класса в Kotlin, даже класса с модификатором final. При этом Nothing нельзя создать — у него приватный конструктор. В коде он объявлен так:

public class Nothing private constructor()

Так как невозможно передать или вернуть тип Nothing, он описывает результат «функции, которая никогда ничего не вернёт».

*Примером(1)* может быть функция, которая выбрасывает exception или в которой запущен бесконечный цикл: в любом из этих случаев она никогда не вернёт значения. В приложениях — независимо от того, какой тип данных возвращает функция, — она может никогда не вернуть данные, потому что произошла ошибка или вычисления затянулись на неопределённый срок. В этом случае имеет смысл использовать Nothing.

*Пример(2)*: функция TODO(), которая часто служит заглушкой в автоматически генерируемых методах.

public inline fun TODO(): Nothing = throw NotImplementedError()

#### Object

Ключевое слово **object** позволяет одновременно объявить класс и создать его экземпляр. Как и класс, *объект* может обладать свойствами, функциями, блоками инициализации, может наследоваться от классов и реализовывать интерфейсы. Единственное, что не допускается - основные и вторичные конструкторы.

#### companion object

аналог **static** из Java

*Объекты* можно объявлять внутри класса. При этом нет каких-либо ограничений по их количеству. Но лишь **один** объект можно пометить ключевым словом companion. Такому объекту можно не указывать имя, а к его компонентам обращаться через имя класса, в котором он находится.

Как правило объекты-компаньоны используются для объявления переменных и функций, к которым требуется обращаться без создания экземпляра класса. Либо для объявления констант. По сути они своего рода замена статическим членам класса (в отличие от Java, в Kotlin нет статики).

##### анонимные объекты / Объект-выражение

Анонимные объекты не являются синглтонами. Каждый раз при выполнении объекта-выражения создаётся новый объект.

Объект-выражение - это выражение, которое “на ходу” создает анонимный объект, который в свою очередь является заменой анонимным внутренним классам в Java.

При разработке приложений анонимный объект чаще всего используется для реализации обработчика событий (клики по компонентам экрана).

button.setOnClickListener(object : View.OnClickListener {

override fun onClick(v: View?) {

...

}

})

Для объекта-выражения не указывается имя. Если же объекту всё таки требуется имя, то его можно сохранить в переменной. Анонимный объект может реализовывать несколько интерфейсов, тогда они перечисляются через запятую. А может и вовсе не реализовывать ни один.

val tree = object {

var name = "Сосна"

}

#### Data классы

Классы, предназначенные для хранения каких-либо данных.

*toString(), equals() и hashCode()* - обычно данные методы имеют одинаковую реализацию и чтобы каждый раз не писать один и тот же код можно просто отметить класс ключевым словом **data** - все необходимые методы будут сгенерированы автоматически.

Не каждый класс можно отметить ключевым словом **data**. Для этого он должен соответствовать определённым требованиям:

-> В основном конструкторе должен быть как минимум один параметр.

-> Все параметры основного конструктора должны быть отмечены ключевыми слова val или var (рекомендуется val).

-> Классы данных не могут быть отмечены ключевыми словами abstract, open, sealed, inner.

#### Generics

Обобщения позволяют определять шаблоны, в которые можно подставлять различные типы. Методы и классы могут использовать объекты, типы которых на момент определения классов и функций неизвестны.

##### Ограничение обобщений

Ограничения обобщений (generic constraints) ограничивают набор типов, которые могут передаваться вместо параметра в обобщениях. Ограничение указывается после названия параметра через двоеточие: <T: Comparable<T>> - то есть в данном случае тип T ограничен типом Comparable<T>, иначе говоря должен представлять тип Comparable<T>.

##### Несколько ограничений

Если параметра типа надо установить несколько ограничений, то все они указываются после возвращаемого типа функции после слова **where** через запятую.

fun <T> getBiggest(a: T, b: T): T where T: Comparable<T>,

T: Number {

return if(a > b) a

else b

}

##### Вариантность, ковариантность и контравариантность

Отношение типов «можно присвоить» – не совсем наследование, такие типы называются совместимыми (отношение «is a»).

**Инвариантность**

Инвариантность предполагает, что, если у нас есть классы Base и Derived, где Derived - производный класс от Base, то класс C<Base> не является ни базовым классом для С<Derived>, ни производным. Инвариантность – это отсутствие свойств ковариантности и контрвариантности.

**Ковариантность - out**

Ковариантностью называется сохранение иерархии типов в производных типах в том же порядке. Например, если у нас есть классы Base и Derived, где Base - базовый класс для Derived, то класс SomeClass<Base> является базовым классом для SomeClass<Derived>.

Для определения обобщенного типа как ковариантного параметр обобщения определяется с ключевым словом **out**.

**Контравариантность - in**

Контравариантность предполагает в какой-то степени обратную ситуацию. Контравариантность предполагает, что, если у нас есть классы Base и Derived, где Base - базовый класс для Derived, то объекту SomeClass<Derived> мы можем присвоить значение SomeClass<Base>. Определяется с ключевым словом **in**.

Проще говоря, когда у ковариантных сущностей типами-параметрами являются родитель и наследник, они сами становятся как бы родителем и наследником. Контравариантные наоборот, становятся наследником и родителем.

#### reified

Ключевое слово reified говорит о том, что нужно иметь доступ к информации о типе T. Таким образом, в некоторых случаях, даже явно не передавая тип, мы можем работать с ним.

#### Функция lazy

Ключевое слово by lazy служит для отложенной инициализации через механизм делегатов. Делегат *lazy* принимает лямбда-выражение с кодом, который вы бы хотели выполнить для инициализации свойства.

Иногда требуется объявить переменную, но отложить инициализацию и вы хотите избегать значения *null*. Kotlin предлагает для подобных случаев использовать by lazy:

val catName: String by lazy { getName() }

В этом случае функция getName() будет вызвана только один раз, при первом обращении к catName. Вместо лямбда-выражения можно также использовать ссылку на функцию:

val catName: String by lazy(::getName)

#### Изменяемость коллекций

Все изменяемые коллекции реализуют интерфейс *MutableIterable*. Он представляет функцию итератора для перебора коллекции.

Для изменения данных в Kotlin также определен интерфейс *kotlin.MutableCollection*, который расширяет интерфейс *kotlin.Collection* и предоставляет методы для удаления и добавления элементов.

Этот интерфейс расширяется интерфейсами *MutableList*, который представляет изменяемый список, и *MutableSet*, который представляет изменяемую неупорядоченную коллекцию уникальных элементов.

И еще одна изменяемая коллекция представлена интерфейсом *MutableMap* - изменяемая карта, где каждый элемент представляет пару ключ-значение.

#### Sealed классы

##### Семантика

На стороне пользователя, мы получим обработку состояний, где доступность атрибутов будет определяться на уровне языка, а неверное использование будет порождать ошибки на этапе компиляции. Так как в экземплярах присутствуют только значимые атрибуты можно говорить об экономии памяти и, что немаловажно, об улучшении семантики.

##### Производительность

В изолированных классах каждый элемент представляет из себя либо класс данных (data class), либо объект (object).

Декомпилированный код (из Kotlin в Java) выражения с when будет с изобилием if instanceof else. Sealed реализация работает медленнее, чем вариант с Enum.

#### init

В целом с init все просто — это обычная inline функция, которая отрабатывает **до** вызова кода самого конструктора.

#### вложенные vs. внутренние классы

Оба находятся внутри другого класса. Для внутреннего добавляется модификатор inner. Внутренние классы содержат ссылку на объект внешнего класса и будет иметь доступ к членам внешнего класса.

#### Extension function

экстеншны являются просто синтаксическим сахарком и компилируются в обычный статический метод

public final class TestKt {

@NotNull

public static final String getEmpty(@NotNull String $receiver) {

Intrinsics.checkParameterIsNotNull($receiver, "receiver$0");

return "";

}

}

#### inline, crossinline, noinline

Inline-функции появились в Kotlin по причине избыточности местной реализации лямбд. Каждый раз, когда вызывается функция, которая принимает лямбду в качестве аргумента, виртуальной машине приходится создавать анонимный объект для хранения этой функции. А так как это затратная в плане ресурсов операция, разработчики Koltin придумали решение: если объявить функцию с ключевым словом *inline*, то компилятор «развернет» эту функцию и ее лямбду в последовательный код.

Правило простое: если ваш код множество раз вызывает функцию, принимающую лямбду в качестве аргумента, лучше сделать ее inline-функцией. Также запомните следующие вещи:

* ключевое слово return, вызванное из лямбды, переданной inline-функции, будет возвращать не из лямбды или inline-функции, а из функции, вызвавшей inline-функцию;
* кроме ключевого слова inline, также существуют noinline и crossinline. Первое можно использовать в отношении отдельно взятых лямбд, когда inline-функция принимает несколько лямбд. Второе пригодится в случае, когда лямбда должна быть выполнена в другом контексте исполнения. Пример:

inline fun exampleFun(crossinline body: () &rarr; Unit) {

Runnable {

body()

}.run()

}

#### lambda vs. inline

Использование функций высшего порядка влечёт за собой снижение производительности: во-первых, функция является объектом, а во-вторых, происходит захват контекста замыканием, то есть функции становятся доступны переменные, объявленные вне её тела. А выделения памяти (как для объекта функции, так и для её класса) и виртуальные вызовы занимают системные ресурсы.

Но во многих случаях эти "накладные расходы" можно устранить с помощью инлайнинга (встраивания) лямбда-выражений.

Встраивание функций может увеличить количество сгенерированного кода, но если вы будете делать это в разумных пределах (не инлайнить большие функции), то получите прирост производительности, особенно при вызове функций с параметрами разного типа внутри циклов.

#### Функции высшего порядка

#### field (Теневые поля)

<https://kotlinlang.ru/docs/reference/properties.html>

<https://programming086.blogspot.com/2017/07/kotlin-4.html>

#### Coroutines

*Coroutines* — это способ выполнения неблокирующих асинхронных операций. Кроме того, создание coroutines — это более «лёгкая» операция, по сравнению с потоками.

Каждый созданный поток (настоящий поток) имеет выделенную память под его стек, как правило, 1 Mб. *Coroutines* же (в текущей реализации kotlinx.coroutines) представляют собой лишь граф последовательности исполнения программы, а выполняться могут как на множестве потоков параллельно, так и на одном потоке.

Выходит, что если запустить одновременно сотню тысяч потоков, то для этого потребуется минимум около 97 Гб памяти. Если же запустить такое же количество *Coroutines*, то нужен пул потоков количеством равным количеству ядер процессора. Потребление памяти при этом составит около пары десятков мегабайт. При этом уровень параллелизма будет одинаков, так как параллельно может выполняться только по одному потоку на ядро процессора.

По сути корутина представляет блок кода, который может выполняться параллельно с остальным кодом.

##### suspend

Модификатор suspend определяет функцию, которая может приостановить свое выполнение и возобновить его через некоторый период времени. suspend fun - “прерываемая” функция.

##### Continuation

Continuation - код, который должна выполнять корутина. Получается, что Continuation является колбэком для suspend функции. Если в корутине есть несколько suspend функций, то Continuation будет колбэком для всех них.

Под капотом на уровне Java будет метод invokeSuspend(), в котором switch-case.

##### Dispatchers

виды:

* Default.Аналог в RxJava - Schedulers Computation. Как правило равен кол-ву ядер CPU. Предназначен для вычислений алгоритмов, интенсивных операций (сортировка списка, парсинг json)
* IO. Предназначен для Input-Output операций (чтение из базы, чтение файла, поход в Сеть)
* Main. В Андроид это главный поток. На каждой платформе по-разному, даже может не быть.
* Unconfined. Делегирует выбор диспатчера вызывающей корутине. Использовать с осторожностью или избегать.
* можно создать свой

##### Scope

Это такая область, где запускаем корутину и где она будет “жить”. И в случае “смерти” скоупа корутины тоже “погибнут”.

виды:

GlobalScope, viewModelScope, lifeCycleScope

##### Job

Это управление корутиной и ее статус. Job хранит в себе состояние корутины: активна/ отменена/ завершена.

lifecycle: Active, Completing, Completed, Cancelling, Cancelled

#### SupervisorJob

##### Context

Это вся информация о корутине - её джоба, диспатчер, имя корутины, обработчик exception и т.д.

##### Builders

**launch** - создать корутину и выполнить какое-либо действие, при этом не важен результат. Т.е. работает по принципу “запустил и забыл”. Возвращает Job, который является ссылкой на корутину.

**async** - позволяет вернуть результат. Возвращает параметризированный тип Deffered, который является наследником от Job. Отличается способ обработки Exception - будет вызвано при вызове await().

##### Обработка ошибок/исключений

**try-catch**

Надо поместить в try-catch код, который выбрасывает ошибку. В этом случае все работает, как и должно.

Если обернуть весь вызов корутины в try-catch, то приложение свалится с крэшем. Т.к. он покрывал только создание и запуск корутины, но не выполнение корутины, т.к. выполнение ушло в отдельный поток.

**CoroutineExceptionHandler**

Это обработчик ошибок, который мы сами можем создать и поместить в контекст корутины. Job получает из Continuation ошибку. Он сообщает об этом родителю (scope), который по такому поводу отменяет себя и всех своих детей. А сам Job пытается передать ошибку в CoroutineExceptionHandler. Если такого обработчика ему не предоставили, то ошибка уходит в глобальный обработчик, что приводит к крэшу приложения.

val handler = CoroutineExceptionHandler { context, exception ->

log("$exception was handled in Coroutine\_${context[CoroutineName]?.name}")

}

val scope = CoroutineScope(Job() + Dispatchers.Default + handler)

scope.launch(CoroutineName("1")) {

launch(CoroutineName("1\_1")) {

TimeUnit.MILLISECONDS.sleep(500)

Integer.parseInt("a")

}

launch(CoroutineName("1\_2")) {

TimeUnit.MILLISECONDS.sleep(1000)

}

}

##### CoroutineStart.LAZY

CoroutineStart.LAZY - Отложенное выполнение.

По умолчанию корутин-билдер launch создает и сразу же запускает корутину. Однако Kotlin также позволяет применять технику отложенного запуска корутины (lazy-запуск), при котором корутина запускается при вызове метода start() объекта Job.

Для установки отложенного запуска в функцию launch() передается значение start = CoroutineStart.LAZY.

##### Flow

### RxJava

Какой паттерн реализует?

Что такое Scheduler? Почему RxJava использует Schedulers? (По умолчанию работает в основном потоке. шедулеры позволяют организовать многопоточную обработку)

Что такое цепочка Observable?

Можно ли в цепочку вставить несколько одинаковых операторов подряд?

В чем отличие между observeOn() и subscribeOn()?

Можно ли расположить несколько операторов subscribeOn() в цепочке? (Можно, но применится только первый)

Можно ли расположить несколько операторов observeOn() в цепочке? (Можно, после каждого будет менятся планировщик)

Разница между map() и flatMap()?

Какие операторы обработки ошибок вы знаете? (doOnError(...)), onErrorReturn(...), onErrorResumeNext(...))

Как можно организовать получение данных с сети и из кэша?

### Многопоточность

Атомарность

Какие коллекции синхронизированы? Как получить синхронизированную коллекцию из не синхронизированной? (Collections.synchronizedList(list);)

Зачем нужна синхронизация?

Что такое дедлок и гонка состояний?

#### volatile vs. synchronized

volatile -это модификатор поля , в то время как synchronized изменяет блоки кода и методы

#### synchronized

Использование synchronized предотвращает получение любым другим потоком монитора (или блокировки) для одного и того же объекта , тем самым предотвращая одновременное выполнение всех блоков кода, защищенных синхронизацией на одном и том же объекте .

#### volatile

Используйте volatile , чтобы гарантировать, что каждый доступ на чтение к переменной будет видеть последнее значение, записанное в эту переменную.

Volatile только гарантирует, что операция чтения всегда дает последнее состояние из памяти в потоках. Однако это не гарантирует никакой безопасности записи / упорядочения операций, т. Е. Два потока могут обновлять переменную volatile в любом случайном порядке. Кроме того, это не гарантирует, что несколько операций над переменной являются атомарными.

### Android

Зачем нужен манифест

#### Базовые компоненты

Activity, Service, BroadcastReceiver, ContentProvider - точки входа в приложение

Какие приоритеты у процессов?

Видимый активный процесс (Foreground process)

Видимый процесс

Background Service

Закэшированный процесс

#### Activity

Activity. Как запустить?

##### Как работает Activity

- <https://habr.com/ru/company/tinkoff/blog/703548/> : про Binder, ActivityStarter, Zygote, Task (стек Activity

##### Жизненный цикл

Ситуация: одно активити запускает другое

Порядок обратных вызовов жизненного цикла четко определен, особенно когда два активити находятся в одном процессе (приложении) и одно запускает другое. Вот порядок операций, которые происходят, когда активити A запускает активити B:

1. Выполняется метод onPause () активити A.
2. Методы onCreate (), onStart () и onResume () активити B выполняются последовательно. (активити B теперь сосредоточено на пользователе.)
3. Затем, если активити A больше не отображается на экране, выполняется его метод onStop ().

Эта предсказуемая последовательность обратных вызовов жизненного цикла позволяет вам управлять переходом информации от одного активити к другому.

Activity: флаги запуска (CLEAR\_TOP, SINGLE\_INSTACE и тд.)

Activity: Как пережить поворот экрана?

##### onSaveInstanceState

onSaveInstanceState() не вызывается, когда пользователь явно закрывает активити или в других случаях, когда вызывается finish().

Вызывается после onStop() и перед onDestroy(). (вроде в ранних версиях чуть раньше вызывался - между onPause и onStop).

##### восстановление (Restore)

Есть 2 способа:

* в *onCreate* с проверкой (saveInstanceState != null)
* с помощью метода onRestoreInstanceState(), который вызовется после *onStart*. Система вызывает onRestoreInstanceState() только в том случае, если есть сохраненное состояние для восстановления, поэтому вам не нужно проверять, является ли Bundle null.

Нужно ли думать о сохранении состояния, если приложение поддерживает только портретную ориентацию?

#### Fragment

##### add / replace

add - просто добавляет. replace - удаляет (remove) и добавляет другой.

#### commit / commitNow

Вызов commit() не выполняет транзакцию немедленно. Скорее, транзакция планируется для запуска в UI-потоке, как только он сможет это сделать. 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 вы 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 использования commit() - это все, что вам нужно.

Fragment: что это и как ими управлять? (FragmentManager)

Опишите жизненный цикл фрагмента

Расскажите о фрагментах, о статическом и динамическом добавлении, жизненный цикл

Как получить ссылку на фрагмент из активити? (findFragmentByTag() и findFragmentById())

Как показать диалог? (Dialog и DialogFragment)

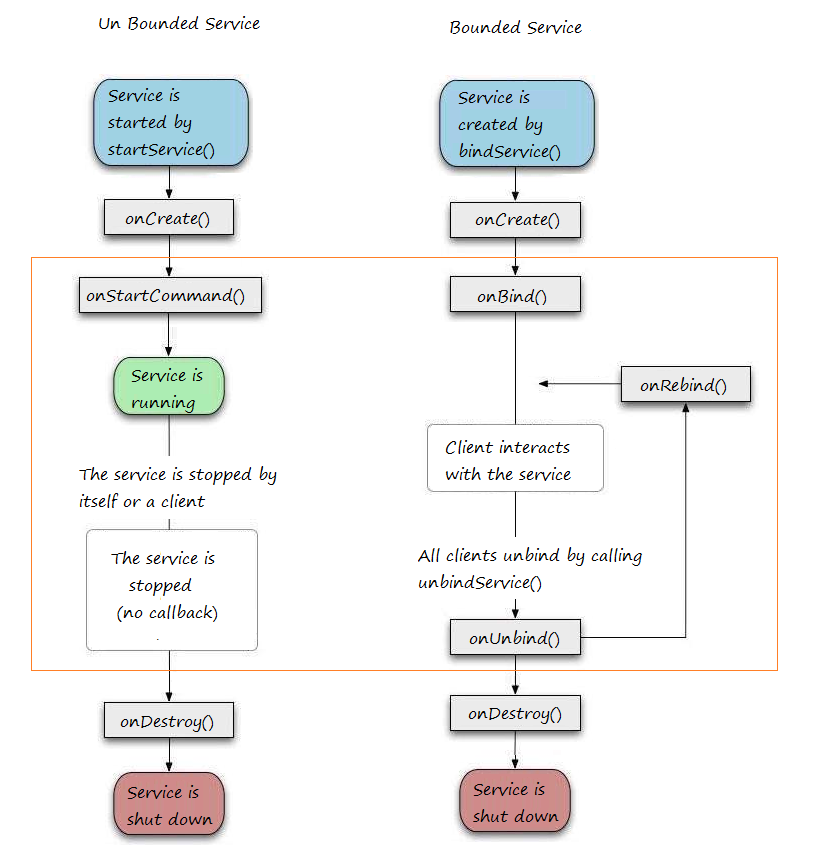
#### Service. Виды сервисов

Сервис (Service) это фоновые процессы в системе выполняющие длительные операции и не нуждаются в взаимодействии с пользователем и работает даже при уничтожении приложения. Сервис имеет два состояния.

##### Started и Bound

| Состояние | Описание |
| --- | --- |
| Started  (Запущенный) | Сервис называется started (запущенный) когда компонент приложения, например Activity запускает его вызовом startService(). При вызове, этот сервис может работать в фоновом режиме в течении неограниченного времени, даже при уничтожении запущенного компонента.  Этот сервис также называет непривязанным (Unbounded Service). |
| Bound  (Привязанный) | Сервис является привязанной (bound) когда компонент приложения привязывается к ней вызовом bindService().  Привязанный сервис предлагает интерфейс клиент-сервер client-server, который позволяет компонентам взаимодействовать со службой, отправлять запросы, получать результаты и даже делать это между разными процессами посредством межпроцессного взаимодействия Interprocess communication (IPC). |

Сервис содержит методы обратного вызова жизненного цикла (life cycle callback methods) которые можно реализовать (implement) для контроля изменений состояния службы и выполнения работы в соответствующие моменты времени. Диаграмма ниже показывает жизненный цикл когда сервис создан с startService(), диаграмма справа показывает жизненный цикл когда сервис создан с bindService().



#### IntentService

Intent Service используется при выполнении одного задания, т.е. при завершении задания сервис самоуничтожается.

#### Сравнение сервисов

| **Unbound Service**  **(Непривязанный)** | **Bound Service**  **(Привязанный)** | **Intent**  **Service** |
| --- | --- | --- |
| **Unbounded Service** используется для выполнения долгосрочных и повторяющихся заданий. | **Bounded Service** используется для выполнения заданий в фоновом режиме (background) и связанный с компонентом интерфейса | **Intent Service** используется для выполнения заданий 1 раз, т.е. при завершения задания сервис самоуничтожается. |
| **Unbound Service** запускается при вызове ***startService()***. | **Bounded Service** запускается при вызове ***bindService()***. | **Intent Service** запускается при вызове ***startService()***. |
| **Unbound Service** останавливется или уничтожается при точном вызове ***stopService()***. | **Bounded Service** снимает ограничения или уничтожается при вызове ***unbindService().*** | **IntentService** вызывается неточно ***stopself()*** для уничтожения |
| **Unbound Service** независим от запускающего компонета. | **Bound Service** зависим от запускающего компонета. | **Intent Service** независим от запускающего компонета. |

#### onStartCommand

##### константы (STICKY)

коротко:

**START\_NOT\_STICKY** – сервис не будет перезапущен после того, как был убит системой

**START\_STICKY** – сервис будет перезапущен после того, как был убит системой

**START\_REDELIVER\_INTENT** – сервис будет перезапущен после того, как был убит системой. Кроме этого, сервис снова получит все вызовы startService, которые не были завершены методом stopSelf(startId).

подробнее:

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.

##### flags

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

#### Background и Foreground

Любой сервис, не зависимо от того Started или Bound, по умолчанию Background. Это значит, что сервис работает как фоновая задача (не путать с фоновым потоком), не требующая взаимодействия с пользователем.

Foreground Service – это сервис, о котором пользователь осведомлен. Это достигается с помощью отображения нотификации в статус-баре.

Пример foreground сервиса – отображение нотификации при проигрывании музыки в приложении-плеере.

Процесс в котором работает foreground сервис имеет больший приоритет, чем процесс с background сервисом. В примере с плеером foreground сервис выполняет сразу две функции:

1. Говорит системе, что этот процесс убивать не надо, т.к. пользователь взаимодействует с ним;

2. Обрабатывает нажатия на кнопки в нотификации.

Запущенный сервис переводится в состояние foreground методом startForeground(int id, Notification notification), принимающим параметрами id нотификации и саму нотификацию, которая будет показана пользователю. Важно знать, что startForeground() вызывается у сервиса, который уже запущен как background сервис, например методом startService().

Для перевода foreground сервиса в состояние background используется метод stopForeground(boolean removeNotification). Этот метод не останавливает сервис (нужно вызвать stopService() или stopSelf()), но увеличивает шансы того, что система убьет процесс.

#### В каком потоке работает сервис? В главном или фоновом?

Service по умолчанию работает в главном потоке, в андроиде это UI thread.

#### BroadcastReceiver

Это механизм для отсылки и получения сообщений в Android. *BroadcastReceiver* — приемник сообщений, посылаемых системой или другим приложением при каком-либо событии, реагирующий на них и выполняющий какую-либо работу.

**Два варианта регистрации приемника**:

* в файле AndroidManifest.xml. Приемник будет принимать сообщения даже если само приложение не запущено.
* в коде приложения (Context-registered receivers). Регистрируется в контексте (Context).

Подробнее ниже.

**Манифест**:

<receiver android:name="com.example.MyBroadcastReceiver">

<intent-filter>

<action android:name="android.net.conn.CONNECTIVITY\_CHANGE"/>

</intent-filter>

</receiver>

com.example.MyBroadcastReceiver - это класс который мы создали для получения уведомления - куда необходимо присылать уведомления, а в нем уже будет вызван метод onReceive(). В теге прописывается идентификаторы того, какие уведомления мы хотим получать.

##### Явные / Неявные

**Неявный (implicit broadcast)** — сообщения рассылаются всем желающим, а не конкретно вашему приложению. Вам нужно только зарегистрироваться для получения этих сообщений через фильтр IntentFilter в манифесте. Система просматривает все объявленные фильтры намерений в вашем манифесте и проверяет, есть ли совпадение. Из-за этого поведения неявные широковещательные сообщения не имеют целевого атрибута

**Явный (explicit broadcast)** предназначена для конкретных приложений. В атрибуте target указывают имя пакета приложения или имя класса компонента, по которому можно найти получателя

##### Передача сообщений

Создаёте объект Intent, загружаете в него нужную информацию и вызываете метод sendBroadcast(), передав ему в качестве параметра созданный объект Intent. Дополнительные данные можно использовать в extras как необязательные параметры. При необходимости поля action, data и category (действие, данные и категорию).

fun sendMessage(view: View?) {

val intent = Intent()

intent.setAction(WHERE\_MY\_CAT\_ACTION)

intent.putExtra("ru.alexanderklimov.broadcast.Message", ALARM\_MESSAGE)

intent.addFlags(Intent.FLAG\_INCLUDE\_STOPPED\_PACKAGES)

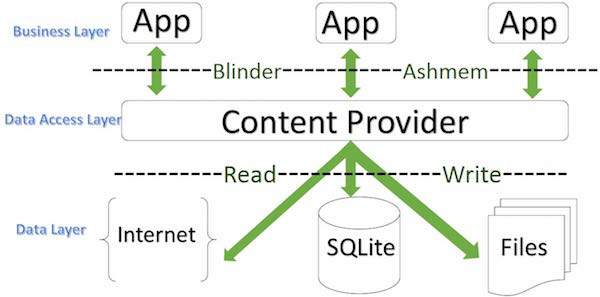
sendBroadcast(intent)

}

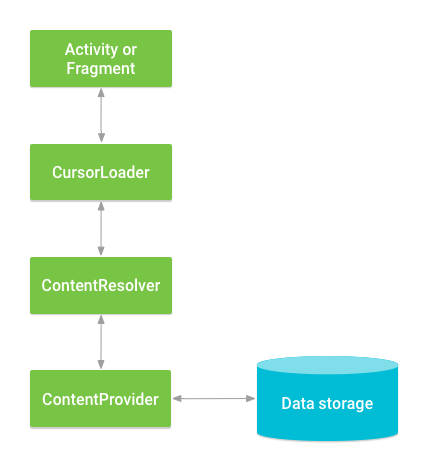
#### ContentProvider

ContentProvider используется для предоставления доступа к хранилищу данных приложения другим приложениям. ContentProvider прописывается в андроид манифесте, как элемент <provider>.

ContentProvider - это как следует из его названия провайдер некоего контента. Это необязательно SQL данные - это может быть все что угодно.Если данные не являются SQL данными, то надо писать собственный класс, который наследует от класса AbstractCursor.



Для доступа к данным контент провайдера используется класс **ContentResolver**, который предоставляет CRUD API для доступа к хранилищу данных и определяет к какому провайдеру направить запрос.



##### Встроенные

В Android имеются следующие встроенные провайдеры, определенные в пакете android.content:

* *AlarmClock*: управление будильником
* *Browser*: история браузера и закладки
* *CalendarContract*: каледарь и информаци о событиях
* *CallLog*: информация о звонках
* *ContactsContract*: контакты
* *MediaStore*: медиа-файлы
* *SearchRecentSuggestions*: подсказки по поиску
* *Settings*: системные настройки
* *UserDictionary*: словарь слов, которые используются для быстрого набора
* *VoicemailContract*: записи голосовой почты

#### Context

Context – это базовый абстрактный класс, реализация которого обеспечивается системой Android. Этот класс имеет методы для доступа к специфичным для конкретного приложения ресурсам и классам и служит для выполнения операций на уровне приложения, таких, как запуск активностей, отправка широковещательных сообщений, получение намерений и прочее.

Получить контекст внутри кода можно одним из следующих методов:

* getBaseContext (получить ссылку на базовый контекст)
* getApplicationContext (получить ссылку на объект приложения)
* getContext (внутри Activity или Service получить ссылку на этот объект)
* this (то же, что и getContext)
* MainActivity.this (внутри вложенного класса или метода получить ссылку на объект MainActivity)
* getActivity (внутри фрагмента получить ссылку на объект родительской активности)

##### Контекст приложения

Это singleton-экземпляр (единственный на всё приложение), доступ через функцию getApplicationContext(), привязан к жизненному циклу приложения.

##### Контекст Activity

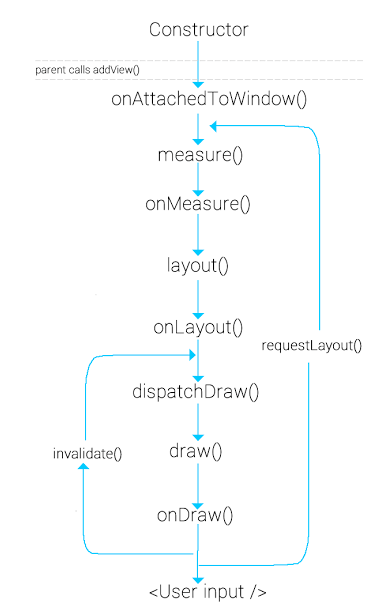
Этот контекст доступен в Activity и привязан к её жизненному циклу.

##### getContext() в ContentProvider

Этот контекст является контекстом приложения и может использоваться аналогично контексту приложения. Доступ через метод getContext().

#### View

##### жизненный цикл



**onAttachedToWindow**

После того как родитель View вызовет метод addView(View), наш View будет прикреплён к окну. На этой стадии наш View-компонент попадает в иерархию родителя.

**onMeasure**

Этот метод означает, что наш View находится на стадии определения собственного размера. Для того чтобы понять как распределить элементы на экране и сколько они занимают место нужно получить от каждого View его размер. В методе measure как раз и происходят расчеты.

Метод onMeasure() принимает 2 аргумента: widthMeasureSpec и heightMeasureSpec. Каждое из значений на самом деле содержит 2 параметра:

* *mode*. Указывает на то, какие правила применяются ко второму параметру size;
* *size*. Непосредственно размер View.

*mode* может принимать следующие значения:

* MeasureSpec.EXACTLY. Означает, что размер задан жёстко. Независимо от размера вашего View, вы должны установить определённую ширину и высоту;
* MeasureSpec.AT\_MOST. Означает что View может быть любого размера, которого пожелает, но, не больше чем размер родителя. Это значение match\_parent;
* MeasureSpec.UNSPECIFIED. Означение что View может само решить какой размер ему нужен не взирая ни на какие ограничения. Это значение wrap\_content.

**onLayout**

Этот метод позволяет присваивать позицию и размер дочерним элементам ViewGroup. В случае, если мы наследовались от View, нам не нужно переопределять этот метод.

**onDraw**

Это основной метод при разработки собственной View. В onDraw вы можете рисовать все что вам нужно.

##### invalidate & requestLayout

**invalidate()**. Запустит только метод onDraw(). Используется когда нужно только перерисовать ваш элемент. Когда изменился цвет или текст или нужно сделать какие-то еще визуальные изменения;

**requestLayout()**. Запустит все методы View. Используется когда нужно изменить размеры вашего View. Вызов requestLayout не только заставит View заново измериться, но и перерисует элемент.

#### ViewModel

Класс ViewModel разработан для хранения данных, связанных с пользовательским интерфейсом, и управления ими с учетом жизненного цикла. Класс ViewModel позволяет данным сохраняться при изменении конфигурации, например при повороте экрана.

##### onCleared

В этом методе вы сможете выполнить все необходимые операции по освобождению ресурсов, закрытию соединений/потоков и т.п. Когда Activity окончательно закрывается, провайдер (ViewModelProviders) удаляет ViewModel, предварительно вызвав его метод onCleared.

#### LiveData

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 отличие от обычного observable, LiveData учитывает жизненный цикл, то есть учитывает жизненный цикл других компонентов приложения, таких как Activity, Fragments и Services. Эта осведомленность гарантирует, что LiveData обновляет только тем компонентам, которые находятся в активном состоянии жизненного цикла.

##### setValue & postValue

Метод setValue должен быть вызван из UI потока. Для обновления данных из других потоков используйте метод postValue. Он перенаправит вызов в UI поток. Соответственно, подписчики всегда будут получать значения в основном потоке.

##### Activity неактивно

Если Activity было неактивно 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.

#### ConstraintLayout. В чем преимущества?

Какими DI фреймворками пользовались?

Зачем нужен proguard? R8?

RecyclerView какой паттерн реализиует

Свои дивайдеры для RecyclerView

#### Zygote, Binder

Чуть менее давно Dalvik превратился в Android Runtime (ART), но сути процесса это не изменило — после тапа на иконку Launcher получает сигнал, передает его в менеджер активностей, тот передается в Zygote, а она создает новое приложение.

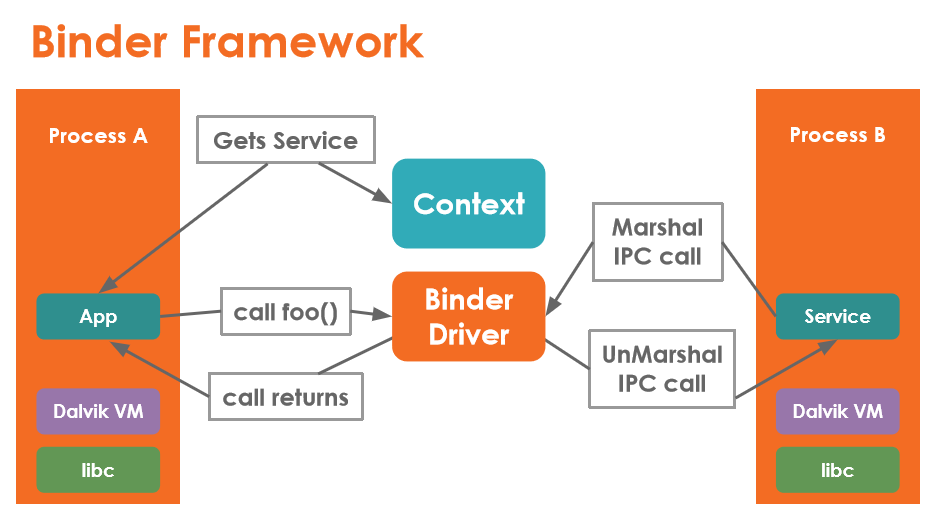
Zygote — демон, который запускается при старте системы и инициализирует первичную виртуальную машину. Zygote позволяет создавать процессы для любых приложений в Android, клонируя себя и корневые библиотеки, которые необходимы для запуска всех приложений. Так экономятся время и память, потому что в первичном экземпляре Zygote уже инициализированы все нужные библиотеки. Останется только использовать Copy-on-Write и изменить ProcessID.

* Де́мон — компьютерная программа в UNIX-подобных системах, запускаемая самой системой и работающая в фоновом режиме без прямого взаимодействия с пользователем.

|  |  |
| --- | --- |

##### Binder

Основной способ взаимодействия процессов — IPC через Binder. Это драйвер, через который работают все корневые структуры Android.



#### Handler, Looper

#### MultiDex

Файлы приложения Android (APK) содержат исполняемые файлы байт-кода в виде файлов Dalvik Executable (DEX), которые содержат скомпилированный код, используемый для запуска вашего приложения. Спецификация исполняемого файла Dalvik ограничивает общее количество методов, на которые можно ссылаться в одном файле DEX, до 65 536, включая методы платформы Android, методы библиотеки и методы в вашем собственном коде. Чтобы преодолеть этот предел, необходимо настроить процесс сборки приложения для создания более одного файла DEX, что называется конфигурацией multidex.

Итак, особенность такова: она позволяет компилировать ваше сложное приложение. Сценарии для его использования - это когда ваше приложение не может скомпилироваться из-за достижения предела ссылок на метод DEX в 64 КБ. Это выглядит как ошибка сборки.

#### ANR

Application Not Responding - Приложение не ответчает

Если блокируем главный поток более чем на 5 сек.

### Gradle

##### AAR vs JAR

Формат aar разработан на основе jar. Доработка потребовалась потому, что в библиотеках для Android должны присутствовать особые файлы, такие как AndroidManifest.xml, Resources, Assets и JNI, которых нет в обычных jar-файлах. Поэтому был изобретён формат aar, решающий все эти проблемы. По сути, это обычный zip-архив, как и jar, только с другой файловой структурой. Файл jar расположен внутри файла aar под именем classes.jar. Остальное перечислено ниже:

* /AndroidManifest.xml (обязательно)
* /classes.jar (обязательно)
* /res/ (обязательно)
* /R.txt (обязательно)
* /assets/ (необязательно)
* /libs/\*.jar (необязательно)
* /jni//\*.so (необязательно)
* /proguard.txt (необязательно)
* /lint.jar (необязательно)

Как видите, формат файла aar специально разработан для Android

##### JCenter и Maven Central

<https://www.roxiemobile.ru/blog/razmeshhenie-biblioteki-cherez-jcenter-i-maven-central-pri-pomoshhi-android-studio/>

### Паттерны

Какие паттерны известны?

Creational (фабрика, фабричный метод, строитель, одиночка)

Structural (адаптер, декоратор, фасад)

Behavioral (итератор, наблюдатель)

MVx, что это и чем отличаются? Чем пользовался?

### Архитектура

Что известно о чистой архитектуре?

С какими архитектурами приходилось работать?

Какой могла бы быть архитектура простого приложения, отображающего юзеров с гитхаба?

### Libraries

Различные сторонние библиотеки и зависимости

Retrofit

OkHttp

Room

Glade, Picasso

#### Dagger2

Для чего нужен модуль в Dagger2

https://ru.stackoverflow.com/questions/1115264/%D0%93%D0%B4%D0%B5-%D0%B8-%D0%B7%D0%B0%D1%87%D0%B5%D0%BC-%D0%BD%D1%83%D0%B6%D0%BD%D0%BE-%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D1%82%D1%8C-dagger2 (+здесь вложенные ссылки на статьи)

https://habr.com/ru/post/279125/

В чем разница Inject через конструктор или в самом классе, что предпочтительнее.

#### Subcomponent и многомодульность

#### Hilt vs Dagger2

### Разное

Agile, Scrum

Как развиваетесь? Следите ли за новинками? Участвуете ли в сообществе?

# 

# 