Android (junior)

* - не обязательные ответы, но желаемые

1. **Основные компоненты в Android Framework**

   Ответ:
- Activity
- Service
- ContentProvider
- Broadcast Reciever

2. **Виды контекста в Android**

   Ответ:
- Application
- Activity

3. **Хранит ли Fragment контекст?**

   Ответ:
- нет, у фрагмента нет своего контекста, но он ссылается на контекст родительской активности

4. **Для чего нужен AndroidManifest.xml и какие параметры можем менять?**

- uses-permission, тэг
- application,
- provider (для доступа к файловой системе),
- activity (объявление активити и их параметров)

5. **Что такое Layout. Какие виды Layout вы знаете?**

- LinearLayout (отображает View-элементы в виде одной строки (если он Horizontal) или одного столбца (если он Vertical). Я использовал это на прошлом уроке, когда демонстрировал использование layout-файлов при смене ориентации)
- TableLayout (отображает элементы в виде таблицы, по строкам и столбцам)
- RelativeLayout (для каждого элемента настраивается его положение относительно других элементов)
- AbsoluteLayout (для каждого элемента указывается явная позиция на экране в системе координат (x,y))
- FrameLayout (позволяет накладывать элементы друг на друга),
- ConstraintLayout (спросить подробнее)

6. **Жизненный цикл Activity. В каких методах какие операции лучше производить?**

Запуск приложения

```
onCreate() → onStart() →  onResume()
```

Нажимаем кнопку Назад для выхода из приложения

```
onPause() → onStop() → onDestroy()
```

Нажата кнопка Домой

```
onPause() → onStop()
```

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

```
onRestart() → onStart() → onResume()
```

Когда запускается другое приложение из области уведомлений или открывается приложение Настройки

```
onPause() → onStop()
```

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

```
onRestart() → onStart() → onResume()
```

Открывается диалоговое окно

```
onPause()
```

Диалоговое окно закрывается

```
onResume()
```

7. Что происходит при смене ориентации экрана (какие методы вызываются ЖЦ вызываются и в каком порядке)?

- onPause()
- onStop()
- *onSaveInstanceState()
- onDestroy()
- onCreate()
- onStart()
- onRestoreInstanceState()
- onResume()

Дополнительно: Что будет с данными в TextView или EditText при повороте экрана?
Ответ: Данные в TextView останутся, так как они сохранятся в методе ЖЦ onSaveInstanceState, и восстановятся в onRestoreInstanceState

8. В чем разница между фрагментом и активностью? Объясните взаимосвязь между ними. Каким образов осуществляется передача данных между ними?

Передача данных с Активности во Фрагмент осуществляется во время инициализации Фрагмента посредством вызова функции newInstance и setArguments
Так как Фрагмент ссылается на контекст Активности, он может получить доступ к любым публичным параметрам и методам.
*Лучше всего передачу данных осуществлять через ViewModel

9. **В чем разница между Serializable и Parcelable? Что предпочтительнее использовать в Android? (+ примеры)**

Serializable медленнее, но лучше для маленького объема данных
Parcelable в разы быстрее, но лучше всего для большого объема данных

10. **RecyclerView vs ListView (что лучше и в чем отличия)**

RecyclerView лучше, хотя бы потому, что динамически инициализирует элементы списка

11. **Почему ресурсы нужно выносить в xml и использовать ссылки на них (например, выносить строки в strings.xml или размеры в dimens.xml)**

Ответ от себя

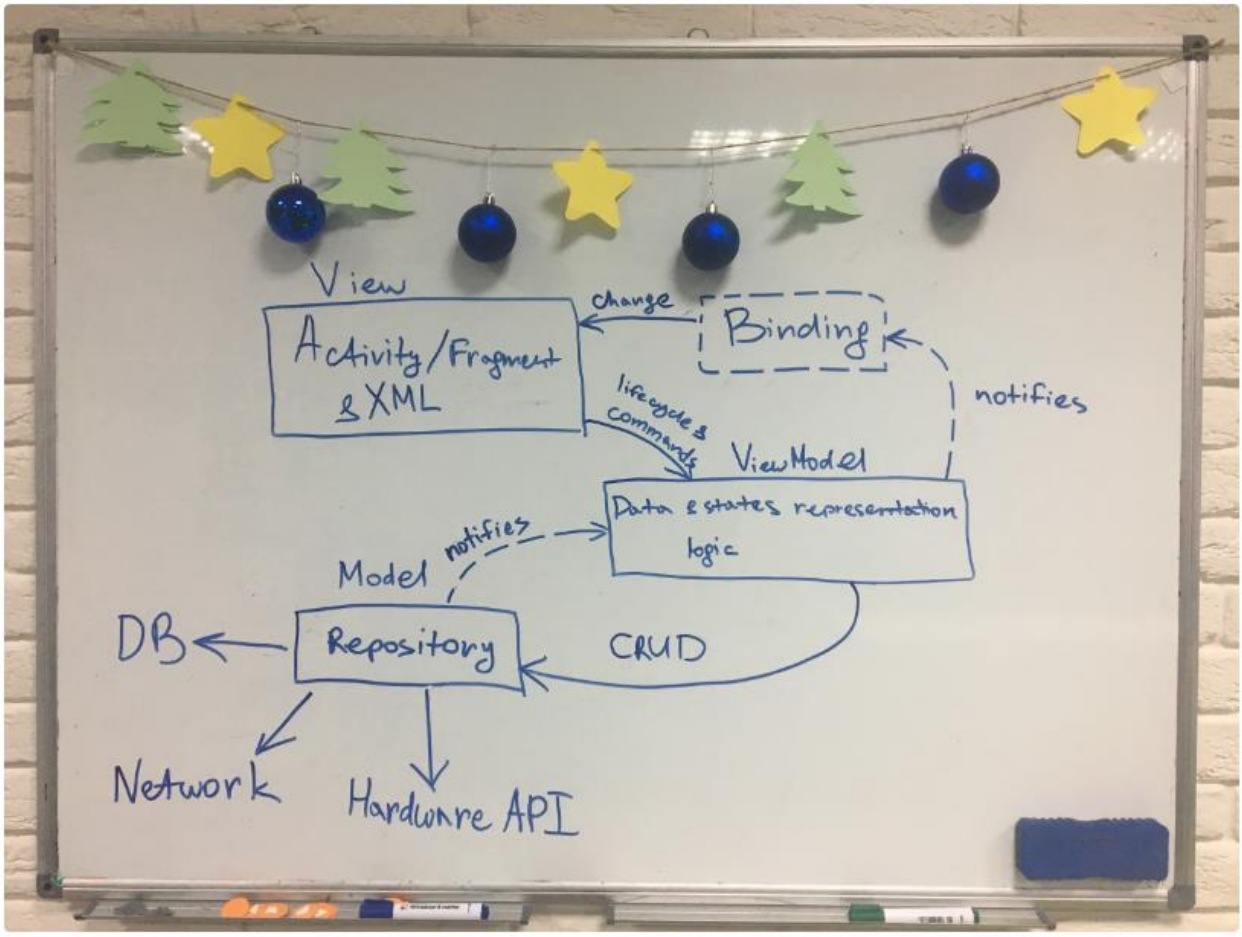
12. **CustomView. Есть опыт в написании CustomView?**

Пусть расскажет про написания собственных View-компонентов

13. **Android Jetpack Framework, в частности Navigation Component**

Использовал ли Navigation Component – если да, пусть расскажет (не критичный вопрос)

14. **MVVM. Model + View + View Model**



Java (junior)

* - не обязательные ответы, но желаемые

1. ООП. Принципы на которых оно держится

Полиморфизм — реализация задач одной и той же идеи разными способами;
Наследование — способность объекта или класса базироваться на другом объекте или классе. Это главный механизм для повторного использования кода. Наследственное отношение классов четко определяет их иерархию;
Инкапсуляция — размещение одного объекта или класса внутри другого для разграничения доступа к ним.

2. SOLID

Вот что входит в принципы SOLID:

- Single Responsibility Principle (Принцип единственной ответственности)

никогда не должно быть больше одной причины изменить класс

- Open Closed Principle (Принцип открытости/закрытости)

программные сущности (классы, модули, функции и т.п.) должны быть открыты для расширения, но закрыты для изменения

- Liskov's Substitution Principle (Принцип подстановки Барбары Лисков)

объекты в программе можно заменить их наследниками без изменения свойств программы

- Interface Segregation Principle (Принцип разделения интерфейса)

клиенты не должны быть вынуждены реализовывать методы, которые они не будут использовать

- Dependency Inversion Principle (Принцип инверсии зависимостей)

зависимости внутри системы строятся на основе абстракций

3. Модификаторы доступа Java (в порядке от private до public);

public: публичный, общедоступный класс или член класса. Поля и методы, объявленные с модификатором public, видны другим классам из текущего пакета и из внешних пакетов.
private: закрытый класс или член класса, противоположность модификатору public. Закрытый класс или член класса доступен только из кода в том же классе.
protected: такой класс или член класса доступен из любого места в текущем классе или пакете, или в производных классах, даже если они находятся в других пакетах

4. Коллекции

- Collection: базовый интерфейс для всех коллекций и других интерфейсов коллекций
- Queue: наследует интерфейс Collection и представляет функционал для структур данных в виде очереди
- *Deque: наследует интерфейс Queue и представляет функционал для двунаправленных очередей
- List: наследует интерфейс Collection и представляет функциональность простых списков
- Set: также расширяет интерфейс Collection и используется для хранения множеств уникальных объектов
- *SortedSet: расширяет интерфейс Set для создания сортированных коллекций
- *NavigableSet: расширяет интерфейс SortedSet для создания коллекций, в которых можно осуществлять поиск по соответствию

- Map: предназначен для созданий структур данных в виде словаря, где каждый элемент имеет определенный ключ и значение. В отличие от других интерфейсов коллекций не наследуется от интерфейса Collection
- ArrayList: простой список объектов
- LinkedList: представляет связанный список
- *ArrayDeque: класс двунаправленной очереди, в которой мы можем произвести вставку и удаление как в начале коллекции, так и в ее конце
- HashSet: набор объектов или хеш-множество, где каждый элемент имеет ключ - уникальный хеш-код
- *TreeSet: набор отсортированных объектов в виде дерева
- *LinkedHashSet: связанное хеш-множество
- *PriorityQueue: очередь приоритетов
- HashMap: структура данных в виде словаря, в котором каждый объект имеет уникальный ключ и некоторое значение

5. Методы equals() and hashcode()

Equals и hashCode являются фундаментальными методами объявленные в классе Object и содержатся в стандартных библиотеках Java. Метод equals() используется для сравнения объектов, а hashCode - для генерации целочисленного кода объекта.
Примеры использования equals
*Примеры использования hashcode

6. StringBuilder vs String;

StringBuilder – класс, позволяющий с удобством работать над преобразованием строк
String – просто строка

7. Потоки. Работали с многопоточность? Thread, Runnable, AsyncTask? В каком потоке можно выполнять операции над view?

Несмотря на то, что главный поток создаётся автоматически, им можно управлять через объект класса Thread. Для этого нужно вызвать метод currentThread(), после чего можно управлять потоком.
Класс Thread содержит несколько методов для управления потоками.

- getName() - получить имя потока
- *getPriority() - получить приоритет потока
- *isAlive() - определить, выполняется ли поток
- *join() - ожидать завершение потока
- run() - запуск потока. В нём пишите свой код
- sleep() - приостановить поток на заданное время
- start() - запустить поток

Есть более сложный вариант создания потока. Для создания нового потока нужно реализовать интерфейс Runnable. Вы можете создать поток из любого объекта, реализующего интерфейс Runnable и объявить метод run().
Внутри метода run() вы размещаете код для нового потока. Этот поток завершится, когда метод вернёт управление.
Когда вы объявите новый класс с интерфейсом Runnable, вам нужно использовать конструктор: *Thread(Runnable объект_потока, String имя_потока)*

Класс AsyncTask предлагает простой и удобный механизм для перемещения трудоёмких операций в фоновый поток. Благодаря ему вы получаете удобство синхронизации обработчиков событий с графическим потоком, что позволяет обновлять элементы пользовательского интерфейса для отчета о ходе выполнения задачи или для вывода результатов, когда задача завершена.

Следует помнить, что AsyncTask не является универсальным решением для всех случаев жизни. Его следует использовать для не слишком продолжительных операций - загрузка небольших изображений, файловые операции, операции с базой данных и т.д. Напрямую с классом AsyncTask работать нельзя (абстрактный класс), вам нужно наследоваться от него (extends). Ваша реализация должна предусматривать классы для объектов, которые будут переданы в качестве параметров методу execute(), для переменных, что станут использоваться для оповещения о ходе выполнения, а также для переменных, где будет храниться результат.

В каком потоке можно выполнять операции над view? - Только в main потоке

8. *Как избежать ConcurrentModificationException во время перебора коллекции?

ConcurrentModificationException к многопоточности никакого отношения не имеет. Возникает, когда мы пытаемся модифицировать коллекцию во время итерирования по ней
Основной и возможно лучший выход: Использовать Iterator

Kotlin (junior)

1. **MutableList vs List**;

List - Общий упорядоченный набор элементов. Методы в этом интерфейсе поддерживают только доступ только для чтения к списку; доступ для чтения / записи поддерживается через интерфейс MutableList.
MutableList - общий упорядоченный набор элементов, который поддерживает добавление и удаление элементов.
Вы можете изменить MutableList: изменить, удалить, добавить … его элементы. В списке вы можете их прочитать.

2. **Kotlin features**. **Что такого классного в Kotlin? Что нового по сравнению с Java?**

- Интерполяция строк

```
val x = 10
val y = 20

println("x=$x y=$y")
```

- Null Safety (безопасное использование null объектов),
- Классы и объекты

В языке Kotlin первичный конструктор не может содержать кода и код должен быть вынесен в блок инициализации. Если же первичный конструктор содержит параметры, то другие конструкторы должны делегировать (явно или неявно через другой, вторичный конструктор) создание объекта первичному конструктору

- Properties

```
val isEmpty: Boolean
    get() = this.size == 0
```

Здесь isEmpty — это вычислимое свойство (предполагается, что у класса, внутри которого находится функция, есть свойство size). Также для свойства можно задать и сеттер через set() = ..., и устанавливать значение можно будет через присваивание имя_свойства=значение.

- Значения и вывод типов

Вместо модификатора final в языке Kotlin явным образом различаются константы и переменные через различные ключевые слова. Для объявления констант (значений) используется ключевое слово val, а для переменных — ключевое слово var. И снова это должно быть хорошо знакомо тем, кто программирует на Scala. Вообще, определение в Kotlin синтаксически решено иначе, чем в Java:

```
val x: Int = 10
```

Использование подобной конструкции дает большую свободу, потому что в языке присутствует механизм вывода типов и во многих случаях указание типа можно опустить

3. **data class — в чём преимущество?**

Если у класса указать ключевое слово data, то автоматически будут созданы и переопределены методы toString(), equals(), hashCode(), copy(). Скорее всего вы будете использовать этот вариант для создания полноценного класса-модели с геттерами и сеттерами.
data class Client(val name: String, val postalCode: Int)

4. Модификаторы **open**, **internal**, **abstract**

- Как и в Java, класс можно объявить абстрактным, добавив ключевое слово abstract. Создать экземпляр такого класса нельзя. Абстрактные методы всегда открыты. Абстрактный класс может содержать абстрактные свойства и функции. Абстрактный класс также может содержать и неабстрактные свойства и функции. А ещё абстрактный класс может не содержать ни одного абстрактного свойства или функции. Если класс содержит какие-либо свойства и функции, помеченные как абстрактные, весь класс должен быть абстрактным.
- В Kotlin добавлен новый модификатор internal, обеспечивающий видимость в границах модуля. Модуль - это набор файлов, компилируемых вместе (модуль в IDEA, проект Eclipse, Maven, Gradle и т.д.). Члены класса и объявления верхнего уровня доступны в пределах модуля.
- По умолчанию все классы в Kotlin имеют статус final, который блокирует возможность наследования. Чтобы сделать класс наследуемым, его нужно пометить ключевым словом open.

5. Модификатор object в Kotlin — где используется и что даёт?

Ключевое слово object одновременно объявляет класс и создаёт его экземпляр. С его помощью можно реализовать шаблон "Одиночка".

6. Lateinit, val, var

Lateinit – отложенная инициализация объекта
Val – свойство для неизменяемого параметра/объекта
Var – свойство для изменяемого параметра/объекта

RxJava (middle)

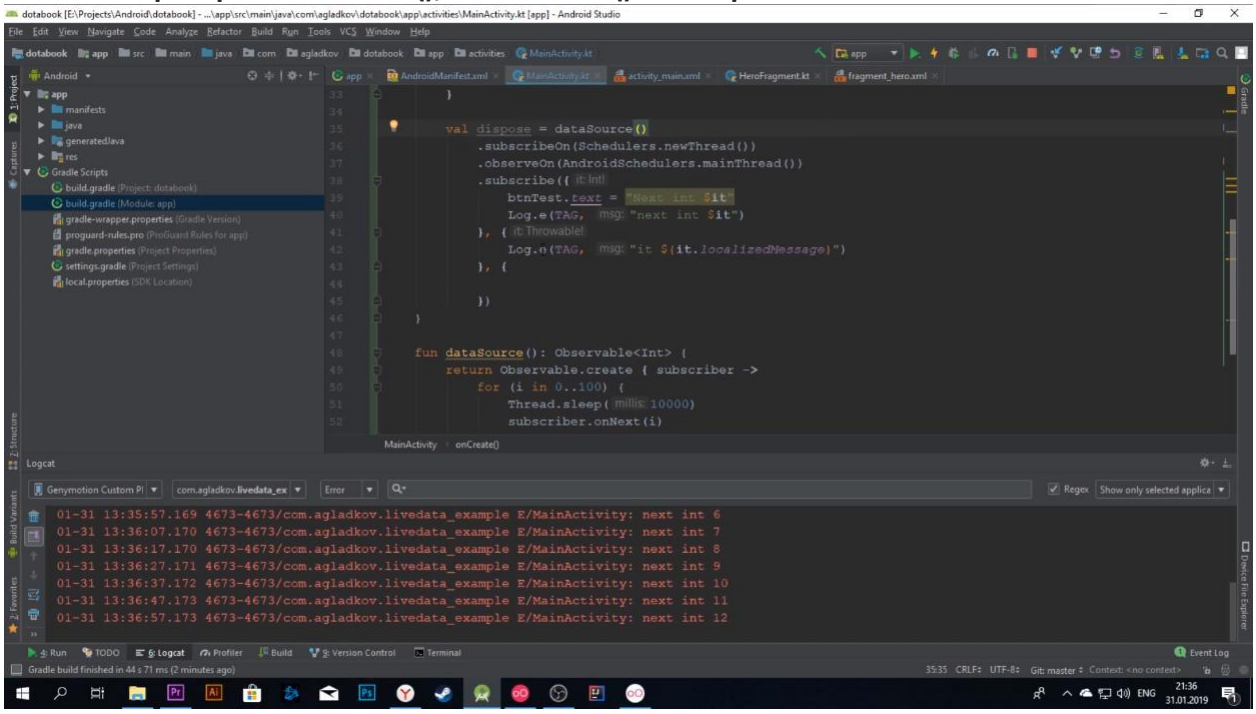
1. Базовые классы

Base classes

RxJava 3 features several base classes you can discover operators on:

- `io.reactivex.rxjava3.core.Flowable` : 0..N flows, supporting Reactive-Streams and backpressure
- `io.reactivex.rxjava3.core.Observable` : 0..N flows, no backpressure,
- `io.reactivex.rxjava3.core.Single` : a flow of exactly 1 item or an error,
- `io.reactivex.rxjava3.core.Completable` : a flow without items but only a completion or error signal,
- `io.reactivex.rxjava3.core.Maybe` : a flow with no items, exactly one item or an error.

2. Операторы: observeOn(), subscribeOn() — как работают?



3. Операторы: map, flatMap, concatMap, switchMap — для чего и в чем отличия?

- Оператор Map() преобразует каждый элемент, излучаемый Observable-ом, и эмиттит измененный элемент. Мы можем создать цепочку из такого количества Map(), какое посчитаем нужным для того, чтобы придать данным наиболее удобную и простую форму для Observer-а.
- FlatMap() принимает на вход данные, излучаемые одним Observable , и возвращает данные, излучаемые другим Observable, подменяя таким образом один Observable на другой.
- Технически данный оператор выдает похожий с FlatMap() результат, меняется только последовательность эмиттируемых данных. ConcatMap() поддерживает порядок эмиссии данных и ожидает исполнения текущего Observable. Поэтому лучше использовать его когда необходимо обеспечить порядок выполнения задач. При этом нужно помнить, что выполнение ConcatMap() занимает больше времени чем FlatMap().
- SwithMap() кардинально отличается от FlatMap() и ConcatMap(). Он лучше всего подходит, если вы хотите проигнорировать промежуточные результаты и рассмотреть последний. SwitchMap отписывается от предыдущего источника Observable всякий раз, когда новый элемент начинает излучать данные, тем самым всегда эмитит данные из текущего Observable.

4. **Flowable vs Observable**?

Всё сводится к такой штуке, как backpressure. Не углубляясь в подробности, скажу лишь, что backpressure позволяет замедлить работу источника данных. Существующие системы имеют ограниченные ресурсы. И с помощью backpressure мы можем сказать всем, кто шлёт нам данные, чтобы они притормозили, потому что мы не можем обрабатывать информацию с той скоростью, с которой она к нам поступает.

В RxJava 1 была поддержка backpressure, но она была добавлена довольно поздно в процессе развития API. В RxJava 1 каждый тип в системе имеет механизм backpressure. И хотя концепцию backpressure поддерживают все типы, далеко не все источники её реализуют, так что использование этого механизма может привести к падению приложения. Применение backpressure должно проектироваться и учитываться заранее. Именно поэтому в RxJava 2 два разных типа источников. Поэтому теперь вы можете указывать с помощью типа источника, должна ли осуществляться поддержка backpressure.

Единственная разница между двумя этими типами заключается в том, что один поддерживает backpressure, а другой — нет.

5. **Single**, **Completable**, **Maybe — случаи использования**

- Completable фактически является Rx-аналогом Runnable. Он представляет собой операцию, которая может быть выполнена или нет. Если проводить аналогию с Kotlin, то это *fun completable()* из мира Rx. Соответственно, для подписки на него необходимо реализовать onComplete и onError. Он не может быть создан из значения (Observable#just, ...), потому что не рассчитан на это.
- Single — реактивный Callable, потому что тут появляется возможность вернуть результат операции. Продолжая сравнение с Kotlin, можно сказать, что Single — это fun single(): T { }. Таким образом, чтобы подписаться на него, необходимо реализовать onSuccess(T) и onError.
- Maybe — нечто среднее между Single и Completable, потому что поддерживает одно значение, отсутствие значений и ошибку. Тут сложнее провести однозначную параллель с методами, но я думаю, что Maybe — это fun maybe(): T? { }, которая возвращает null, когда результата нет. Несложно догадаться, что для подписки потребуется определить onSuccess(T), onComplete и onError.