Вопросы кастомные с моего опыта, best from бигтех, что запомнилось
Ответы AI (первые 3 общих блока)

Java

Методы класса Object
Коллекции Java
Стирание типов
hashcode&equals контракт
HashMap вкратце как работает
LinkedList vs ArrayList сложности вставки в середину, конец, что лучше когда
Многопоточка Sync volatile
race condition, lock
Статический класс vs обычный
Что такое Exceptions, в каком порядке отлавливать
Стек vs куча
Garbage collection

1. Методы класса Object
equals(Object obj) — сравнение объектов на равенство.
hashCode() — возвращает хэш-код объекта.
toString() — строковое представление объекта.
clone() — создание глубокой или поверхностной копии объекта.
finalize() — вызывается перед уничтожением объекта (устаревший метод).
getClass() — возвращает класс объекта.
notify(), notifyAll(), wait() — управление потоками, ожидание и уведомление.
2. Коллекции Java
Основные интерфейсы: Collection, List, Set, Map, Queue.
Основные реализации:
ArrayList — динамический массив, быстрый доступ по индексу.
LinkedList — связанный список, быстрее вставка/удаление в середине при обходе.
HashSet — хэш-таблица уникальных элементов.
TreeSet — отсортированное дерево.
HashMap — хэш-таблица ключ-значение.
TreeMap — отсортированное дерево по ключам.
3. Стирание типов (Generics, type erasure)
Время компиляции типы проверяются, а на этапе выполнения информация о типах исчезает.
Это обеспечивает совместимость с более старыми версиями Java, но требует осторожности (например, нельзя создавать массивы с параметризованными типами).
4. hashCode() & equals() контракт
Если два объекта равны (equals()), то их хэш-коды должны совпадать.
Обязательно переопределять оба метода вместе.
Использования: например, для корректной работы HashMap/HashSet.
5. HashMap: как работает кратко
Использует массив бакетов + связные списки или деревья для коллизий.
При добавлении вычисляется хэш, находится индекс массива.
Если коллизия — добавление в цепочку или дерево.
Чтобы получить элемент — снова вычисляется хэш и происходит поиск по цепочке.
6. LinkedList vs ArrayList
Операция ArrayListLinkedList Сложности Когда что лучше?
Вставка в середину (по индексу) O(n) O(1) при наличии итератора — LinkedList лучше при частых вставках/удалениях в середине, ArrayList — при преимущественно чтении и вставке в конец.
Вставка в конец O(1) амортизированно O(1) — обе подходят, лучше ArrayList для большинства случаев.
7. Многопоточность: synchronized, volatile
synchronized — блокировка/разблокировка, обеспечивает взаимное исключение.
volatile — переменная всегда читается из основной памяти, предотвращая кэширование и обеспечивая видимость изменений.

Важна для предотвращения условий гонки.
8. Race Condition, Lock
Race condition — ситуация, когда результат зависит от неупорядоченного выполнения потоков.
Lock — механизм для исключения одновременного доступа к данным (например, synchronized, Lock из java.util.concurrent).
9. Статический класс vs обычный
Статический вложенный класс — не имеет доступа к экземпляру внешнего класса.
Обычный внутренний класс — имеет доступ к членам внешнего класса, требует экземпляра.
10. Exceptions — что и как ловить
Исключения бывают проверяемые (checked) и непроверяемые (unchecked).
Ловить их можно в порядке: если есть необходимость, ловить конкретные (FileNotFoundException), затем более общие (IOException), или использовать универсальный Exception.
Хорошая практика — ловить только те исключения, которые можно корректно обработать.
11. Стек vs куча
Стек — область памяти для хранения вызовов методов, локальных переменных. Быстрый доступ, размер фиксирован.
Куча — для динамического выделения памяти объектов, медленнее, автоматическая уборка (GC).
12. Garbage Collection (GC)
Автоматический сбор мусора удаляет неиспользуемые объекты.
Различные алгоритмы: mark-and-sweep, generational, G1.
Важна для автоматического управления памятью, избегает утечек памяти.

Kotlin

Методы класса Any
Data class, момент с использованием его в коллекциях (объекты могут теряться при некоректном переопределении hashcode equals, а также var val параметров в первичном конструкторе)
Иерархия рутовых классов в котлин (Any, Nothing, Unit)
Companion object, как работает
модификатор inline, reified
Extension методы, как работают
scope методы
val vs const val
sealed class
интерфейс vs абстрактный класс

Методы класса Any (Kotlin)
equals(other: Any?): Boolean — сравнение объектов по равенству.
hashCode(): Int — хэш-код объекта.
toString(): String — строковое представление.
clone() — нет в Any.
also, apply, run — это scope-методы, не из Any, а расширения или стандартные функции.
2. Data class, и особенности при использовании в коллекциях
Data class автоматически генерирует equals(), hashCode(), toString(), copy() и componentN() для параметров.
При использовании в коллекциях:
hashCode() и equals() критичны для корректной работы Set, Map.
Обязательно правильно переопределять или позволяющить компилятору создавать эти методы.
Параметры в первичном конструкторе:
val — участвуют в equals() и hashCode() по умолчанию.
var — тоже участвуют, но их изменение может потенциально влиять на работу коллекций.
Обратите внимание: В data class оба — val и var, работают одинаково, важно лишь знать, что изменение var может повлиять на хэш при использовании в хэш-таблицах.
3. Иерархия рутовых классов в Kotlin
Any — базовый класс для всех классов, как Object в Java.
Nothing — тип, который обозначает "нет значения", часто используется для избегания ошибок, выбрасывает исключения или бесконечные циклы.
Unit — тип, аналогичный void в Java (ничего не возвращает), используется в функциях без возвращаемого значения.
4. Companion object, как работает
Это объект внутри класса, который делается статическим аналогом.
Он создается один раз и служит для реализации статических методов и переменных.
Можно обращаться как MyClass.companionFunction(), или через MyClass если объявлен @JvmStatic.
5. Модификатор inline, reified
inline — вставляет тело функции прямо в место вызова, уменьшает overhead, особенно с лямбдами.
reified — доступен только внутри inline функций, позволяет получить информацию о типе параметра через T::class, что обычно недоступно из-за стирания типов.
Пример:

kotlin

```
inline fun <reified T> isInstance(value: Any): Boolean {
    return value is T
}
```

6. Extension методы, как работают
Расширения позволяют добавлять функции к существующим классам без наследования или модификации исходных классов.
Внутренне реализуются как статические методы, вызываются через синтаксис object.extensionFunction().
Пример:

kotlin

fun String.reverse(): String = this.reversed()

val rev = "hello".reverse() // "olleh"
7. Scope методы (also, apply, run, let, with)
let — передача объекта как it, цепочка вызовов, возвращает последнюю строку.
apply — this внутри блока, возвращает объект.
also — it, возвращает объект, часто для побочных эффектов.
run — как apply, но возвращает результат блока.
with — принимает объект, выполняет блок, возвращает результат.
8. val vs const val
val — неизменяемая переменная, может быть определена в любой части кода.
const val — compile-time константа, должна быть объявлена на уровне объекта или файла (top-level) и допускает только примитивы и строки.
9. Sealed class
Ограничивает наследование: все подклассы должны находиться в одном файле.
Используются для представления семейства типов (например, Result или State).
Хорошо интегрируются с when, позволяя полностью покрыть все случаи.
10. Интерфейс vs абстрактный класс
Интерфейс — задает контракт, может содержать только абстрактные методы или с Kotlin 1.4+ — реализации по умолчанию.
Абстрактный класс — может содержать реализацию методов, поля, конструктор. Используется, если есть общий код и состояния.
В Kotlin рекомендуется предпочитать интерфейсы, если нет необходимости в хранении состояния или реализации.

Android

Компоненты Android основные 4 - перечислить, описать в кратце
жц Activity / Fragment
Виды архитектур в андроид разработке - mvp, mvvm, mvi (опционально, на проекте второе используется)
clean architecture - суть бизнес логика независима, все внешние слои зависят только от внутренних слоев
Работа в фоне парочку способов описать
Многопоточка в андроиде
Coroutines что такое, как работают, чем отличаются от потоков
Coroutine builders, scope, context
launch vs async coroutines
Parcelable vs serializable рефлексия
Горячие холодные потоки RxJava
FlatMap vs map
Stateflow, Sharedflow разница
Structured Concurrency вкратце
что такое recyclerview
Работал ли с custom views?
Методы View основные 3 вкратце что делают
Работал ли с compose? на проекте нет, просто для справки
Работа с бд. Работал с ли с room, preferences?
Как искать компоненты в ui средства студии
Как дебажить в студии
Как мокировал бекенды
Знаешь ли что такое adb, пару команд если пользовался
Что такое инжекция зависимостей
Dagger 2 что такое, @scope, @named
@provides vs @binds
Модуль vs компонент Dagger
Compose effects
@Immutable, @Stable

1. Основные компоненты Android (4 основных)
Activity — экран приложения, управляет UI и жизненным циклом. Основной компонент для взаимодействия с пользователем.
Fragment — часть UI внутри Activity, позволяет модульную архитектуру и повторное использование интерфейса.
Service — фоновая задача, работающая независимо от UI (например, загрузка данных).
Broadcast Receiver — слушает системные или другие широковещательные сообщения, обрабатывает их.
2. Виды архитектур в Android
MVP (Model-View-Presenter) — разделение ответственности: View (UI), Presenter (логика), Model (данные). Взаимодействие через интерфейсы.
MVVM (Model-View-ViewModel) — View связана с ViewModel через data binding. ViewModel содержит логику и живет дольше активности/фрагмента.
MVI (Model-View-Intent) — потоковое управление состоянием, каждый UI-вход это интент, состояние управляется через реактивные потоки.
На проекте чаще используют MVVM или MVI.

3. Clean Architecture
Внутренние слои — бизнес-логика, они полностью изолированы от внешних.
Внешние слои (например, слой UI или инфраструктура) зависят только от внутренних.
Основная идея — модульность, тестируемость и независимость бизнес-логики.
4. Работа в фоне — способы
AsyncTask (устаревший).
Handler / HandlerThread — низкоуровневая.
Executors — для управления пулом потоков.
Coroutine — современный и удобный способ.
WorkManager — долговременные задачи с учетом состояния сети и батареи.

5. Многопоточность в Android
Основные потоки: UI-поток (основной), фоновый (для тяжелых задач).
Управление потоками через Thread, Handler, Executor, или современные Kotlin Coroutines.
6. Kotlin Coroutines
Легкий способ писать асинхронный код.
Позволяют писать асинхронный код в привычной последовательной форме.
Отличие от потоков: меньшая нагрузка, контроль, структура кода.
Работают: через suspend-функции, builders, scope и контекст.

Builders:

launch {} — запускает корутину, ничего не возвращает.
async {} — возвращает Deferred/пул результата.
Scope и Context:

CoroutineScope — область действия корутин.
Контекст — содержит диспетчер (Dispatchers.Main, IO, Default).
7. launch vs async
launch — запускает корутину, возвращает Job, используется для побочных эффектов.
async — возвращает Deferred, нужен для получения результата.
8. Parcelable vs Serializable
Parcelable — платформа, более эффективный, требует больше кода.
Serializable — стандартный Java-интерфейс, проще в использовании, медленнее.
Рефлексия — Serializable использует её, Parcelable реализуется вручную, без рефлексии.

9. Горячие и холодные потоки в RxJava
Холодные — поток запускается при подписке, каждый получает свои данные.
Горячие — данные идут независимо от подписки, все подписчики видят одни и те же данные.
10. flatMap vs map
map — преобразует элементы потока.
flatMap — преобразует и разворачивает вложенные потоки в один.
11. StateFlow vs SharedFlow
StateFlow — хранит текущее состояние, обязательное значение, используется для сохранения состояния UI.
SharedFlow — мультикаст поток без состояния, используется для событий.
12. Structured Concurrency
Концепция, при которой жизненный цикл корутин связан с родительским scope.
Облегчает управление многими корутинами и их отменой.
13. RecyclerView
Разновидность ListView, обеспечивает гибкое и эффективное отображение больших списков.
Поддержка виртуализации, анимаций, адаптеров.
14. Custom Views
Было ли — да/нет.
Создаешь свой компонент, переопределяя методы onDraw, onMeasure.
15. Методы View (основные 3)
setOnClickListener() — обработка клика.
setVisibility() — управление видимостью.
invalidate() — перерисовка.
16. Compose (для справки)
Новая декларативная UI библиотека, создание UI через функции.
В проекте пока не используется — просто для сведения.
17. Работа с БД
Опыт с Room — ORM для SQLite.
Preferences — для хранения настроек, предпочтений (ключ-значение).
18. Поиск компонентов в UI (Android Studio)
Использовать UI Automator, Layout Inspector, Find View или Layout Tree.
19. Дебаг в Android Studio
Точки остановки, Logcat, профилирование, инспектор памяти.

20. Мокирование бекендов
Использовать Mockito, MockWebServer, Retrofit с мокающими ответами.
21. adb
Android Debug Bridge — инструмент командной строки.
Команды:
adb devices — список устройств.
adb logcat — просмотр логов.
adb shell — команда в shell устройства.
22. Инжекция зависимостей
Внедрение зависимостей для повышения тестируемости и модульности.
23. Dagger 2
Обеспечивает DI через аннотации.
@Scope — область видимости.
@Named — именованные зависимости.
@Provides — методы предоставления зависимостей.
@Binds — связывает интерфейс с реализацией.
24. Модуль vs компонент Dagger
Модуль — содержит методы @Provides.
Компонент — объединяет модули и предоставляет зависимости.
25. Compose Effects
LaunchedEffect, remember, SideEffect — для выполнения эффектов и логики при изменении состояния.
26. @Immutable, @Stable
Аннотации для указания, что объект не изменяется или стабилен, что помогает Compose оптимизировать перерисовки.

Development общее

ООП, SOLID
Типы паттернов проектирования
С какими version control работал
Флоу отправки кода в репозиторий
Что такое squash commits, amend
Что такое конфликты
Что такое cherry pick
Что такое ci cd на мобилке
Что такое unit tests

1. Объектно-ориентированное программирование (ООП) и принципы SOLID
ООП — парадигма программирования, основанная на объектах, которые содержат состояние (данные) и поведение (методы).
Основные принципы SOLID:

S — Single Responsibility Principle (Принцип единой ответственности): класс должен иметь только одну причину для изменения.
O — Open/Closed Principle: классы должны быть открыты для расширения, но закрыты для модификации.
L — Liskov Substitution Principle: объекты базового класса должны без проблем заменяться объектами подкласса.
I — Interface Segregation Principle: лучше иметь множество специфичных интерфейсов, чем один общий.
D — Dependency Inversion Principle: модули высокого уровня не должны зависеть от модулей низкого уровня; оба должны зависеть от абстракций.
2. Типы паттернов проектирования
Созидательные:
Singleton (один экземпляр)
Factory Method (фабричный метод)
Abstract Factory (абстрактная фабрика)
Builder (строитель)
Prototype (прототип)
Структурные:
Adapter (приспособитель)
Decorator (декоратор)
Facade (фасад)
Composite (компонент)
Proxy (прокси)
Поведенческие:
Observer (наблюдатель)
Strategy (стратегия)
Command (команда)
State (состояние)
Chain of Responsibility (цепочка обязанностей)
Visitor (посетитель)
3. С каким version control работал
В основном используют Git, иногда — SVN или другие системы.
4. Флоу отправки кода в репозиторий
Обычно:
Создать ветку (feature, bugfix).
Внести изменения.
Сделать коммит (git commit).
Обновить локальную ветку (git pull).
Запушить изменения (git push).
Создать Pull Request (или Merge Request) на сервере (GitHub, GitLab).
Проходит ревью и мерж в основную ветку.
5. Squash commits, amend

Squash commits — объединение нескольких коммитов в один при слиянии веток, чтобы сохранить чистую историю.
Amend — команда git commit --amend позволяет изменить последний коммит, добавив или изменив его содержание.
6. Конфликты
Возникают, когда изменения в разных ветках или файлах противоречат друг другу. Требуют ручного разрешения, выбор нужной версии кода.
7. Cherry pick
Команда git cherry-pick переносит отдельный коммит из одной ветки в другую, без полного слияния.
8. CI/CD на мобильных
CI (Continuous Integration) — автоматическая проверка и сборка кода при каждом коммите.
CD (Continuous Delivery/Deployment) — автоматическая подготовка и публикация приложения (например, в Google Play).
На мобильных: автоматические сборки, тестирование, проверка статического анализа, апк/аапки автоматически выкладываются.
9. Unit tests
Юнит-тесты — автоматические тесты, проверяющие работу отдельных методов или компонентов.
Позволяют убедиться, что небольшая часть кода работает правильно, помогают обнаруживать регрессии.

*Проекто-специфичные вопросы

1. **update editText silently**

Есть форма ввода для поиска диагнозов заболеваний. Классический TextInputLayout > EditText из material стандартной андроид библиотеки. На каждый ввод символа обычно сеттится слушатель ввода addTextChangedListener. Нам нужно программно установить текст найденного и выбранного диагноза в форму ввода (под формой некий список с тем, что нашлось при вводе символов). Вопрос - как это сделать и не триггернуть поиск заново?

Ответ: для начала необходимо сделать базовый editText.text = “Нужный текст”, но чтобы обновить текст тихо, без триггера запроса поиска на бек, надо провернуть хак с выключением слушателя, установкой текста, включением слушателя

https://stackoverflow.com/questions/9385081/how-can-i-change-the-edittext-text-without-triggering-the-text-watcher

```
class MyTextWatcher(private val editText: EditText) : TextWatcher {

  override fun afterTextChanged(e: Editable) {
    editText.removeTextChangedListener(this)
    e.replace(0, e.length, e.toString())
    editText.addTextChangedListener(this)
  }

  ...
}

et_text.addTextChangedListener(new MyTextWatcher(et_text));
```

2. **Баг с бесконечной загрузкой WebView**

Есть некий WebView, загружающий страницу авторизации. Ошибка малозначащая, не содержит явных рекомендаций.
Интересен ход размышлений, как человек будет подступиться к проблеме:
> просмотр логов в студии
> просмотр логов которые пишет папка в файл (если такое есть)
> поиск нужного экрана и как - через layout inspector
> переход к настройке WebView - **webView.settings.dom Storage Enabled = true**

Суть ошибки
> в логах студии/папки была ошибка связанная с domStoragePersmission, но явно об этом не сообщалось, падало на абстрактной какой-то другой ошибке
> сам элемент использующий сторадж присутствовал в .js коде страницы авторизации (если открыть ее и посмотреть там как раз было упоминание про некий storage - там сохранялись метаданные некие)
> здесь можно упомянуть использование ИИ, что частично помогло и мне
> вероятно на стороне ГосТех авторизации кто-то допилил функционал и он стал с веб вью пытаться сохранить данные на девайсе в папке. Соответственно под это существует свой перминш-настройка для WebView вьюшки domStorageEnabled, который не был задан в коде

3. **Баг с загрузкой картинок Picasso**

Есть классические ImageViews в которые загружаются картинки по сети посредством библиотеки Picasso. Также имеется информация, что раньше они успешно загружались, но в текущей сборке - нет. Как бы подступился к задаче, если сообщили, что девопс только что починил сетевую доступность аналогичного функционала на смежной платформе веб. Ну и сами картинки загружаются при открытии их url в строке

браузера. Стоит упомянуть, что проект исправно работает с продом без впн соединения, и с впн для тестовых сборок, те грузится все, кроме картинок.

Здесь также ход мыслей интересен, и знание одного факта нюанса про Пикассо - он имеет настраиваемый builder, который позволяет загружать данные с сервера, игнорируя отсутствие сертификатов. Соответственно дефолтная настройка которая была в коде предполагала наличие сертификата, и потребовался такой work-around, чтобы решить проблему на мобилке.

Кастомный Билдер для Picasso:

```
fun getUnsafeOkHttpClient(): OkHttpClient {
    val trustAllCerts = arrayOf<TrustManager>(
        object : X509TrustManager {
            override fun checkClientTrusted(chain: Array<out X509Certificate>?, authType: String?) {}
            override fun checkServerTrusted(chain: Array<out X509Certificate>?, authType: String?) {}
            override fun getAcceptedIssuers(): Array<X509Certificate> = arrayOf()
        }
    )

    val sslContext = SSLContext.getInstance("SSL")
    sslContext.init(null, trustAllCerts, java.security.SecureRandom())

    val sslSocketFactory = sslContext.socketFactory

    return OkHttpClient.Builder()
        .sslSocketFactory(sslSocketFactory, trustAllCerts[0] as X509TrustManager)
        .hostnameVerifier { _, _ -> true }
        .build()
}

val picasso = Picasso.Builder(context)
    .downloader(OkHttp3Downloader(getUnsafeOkHttpClient()))
    .build().apply {
        isLoggingEnabled = true
    }
```

Дополнительный вопрос. Также был моментик с разрастанием картинок до 150 мб и падением приложения по памяти после починки загрузки выше (на списке статей, где соответственно имеются картинки заболеваний статей). Решение было в отсутствии правильной настройки imageView после загрузки картинки - нужно обязательно вызывать **fit()** на imageView - imageView.fit().centerCrop()