// JavaJub

Собес в T1

Иннотех

Java Junior (Middle)

Вопросы, задачи и подготовка
к live-coding и техническому интервью

Стек

Java · Spring Boot · Hibernate · PostgreSQL · Kafka · Docker · Microservices

// содержание

Содержание

01 Про T1 и формат собеса

ИТ-Холдинг T1 — крупнейший частный IT-интегратор России. 30 000+ сотрудников, проекты для Сбера, ВТБ, Росатома, министерств. Иннотех — дочерняя компания T1, разрабатывающая продукты для банков (кредиты физлиц, страхование).

Вакансия заявлена как «Junior», но требования чётко Middle: 3+ лет коммерческого опыта Java, 2–3 года на высоконагруженных проектах, микросервисы, Kafka. Готовиться нужно на Middle.

Структура собеседования (~1.5–2 часа)

Блок	Время	Что проверяют
Опыт + проект	~10 мин	Стек, нагрузка, зона ответственности
Java Core + многопоточность	~25 мин	equals/hashCode, HashMap, volatile, CAS
Spring + Hibernate + транзакции	~20 мин	Жизненный цикл, прокси, N+1, Propagation
Kafka + БД + микросервисы	~20 мин	Гарантии, MVCC, Saga, Circuit Breaker
Code Review	~15 мин	Найти 5–10 проблем в Spring-сервисе
SQL практика	~15 мин	JOIN 3 таблиц + GROUP BY + HAVING
Архитектурный кейс	~10 мин	Rate limiting, health checks, UUID vs Long

ФИШКА · Специфика T1/Иннотех

На собесах в T1 любят: Code Review реального Spring-сервиса, архитектурные кейсы (rate limiting между сервисами с разной TPS, circuit breaker при «моргании сети»), SQL-серию из 4 задач. Сдвиг 2026: от синтаксиса к инженерному мышлению — обоснование выбора через производительность и стоимость.

ВНИМАНИЕ · Заявлен Junior, но...

Требования: 3+ лет опыта Java, 2–3 года на высоконагруженных, микросервисы, Kafka/RabbitMQ, Docker, тесты. Это полноценный Middle. Готовьтесь соответственно.

02 Стек по вакансии

Основной стек

- Java (3+ лет) — Core, многопоточность, Stream API
- Spring Boot — основной фреймворк, микросервисы
- Hibernate / JPA — ORM, маппинг, кэширование
- PostgreSQL — основная СУБД
- Kafka / RabbitMQ — брокеры сообщений
- Docker — контейнеризация
- JUnit, Mockito — тестирование
- Микросервисная архитектура — опыт 2–3 года
- Git, Maven/Gradle — сборка и CI/CD

Будет плюсом

- Kubernetes / OpenShift — оркестрация
- Redis — кэширование
- Elasticsearch — полнотекстовый поиск
- gRPC — межсервисное взаимодействие
- Resilience4j — Circuit Breaker, Retry
- Опыт в банковских/страховых проектах

03 Java Core

- Расскажите про контракт equals/hashCode. Что случится с HashMap при константном hashCode?

  → *Если a.equals(b), то hashCode одинаков. Обратное необязательно. При константном hashCode — все в одном bucket: Java 7 — список O(n), Java 8+ при 8 коллизиях — red-black tree O(log n). Деградация. При рандомном — объект «потеряется» при get().*

- Почему ключ HashMap должен быть иммутабельным?

  → *Если изменить поле, участвующее в hashCode — объект окажется в «неправильном» bucket. contains() вернёт false. Объект потерян — ни найти, ни удалить. Утечка памяти. Решение: immutable-ключи (String, Integer, record).*

- Как написать иммутабельный класс?

  → *final class (нельзя наследовать). private final поля. Нет сеттеров. Конструктор с defensive copy для mutable-аргументов. Геттеры возвращают defensive copy для mutable-полей (List → Collections.unmodifiableList()). Пример: record в Java 16+ — immutable по умолчанию.*

- String vs StringBuilder vs StringBuffer.

  → *String: immutable, при конкатенации создаёт новый объект. StringBuilder: mutable, НЕ потокобезопасный (быстрый). StringBuffer: mutable, потокобезопасный (synchronized — медленнее). В цикле конкатенация String → O(n²), StringBuilder → O(n). Компилятор оптимизирует простые случаи, но не циклы.*

- Чем опасны типы-обёртки (Integer, Boolean)?

  → *NPE при unboxing null: Integer i = null; int x = i; → NPE. Лишняя память: Integer = 16 байт vs int = 4 байта. Медленнее: автобоксинг создаёт объекты. IntegerCache: valueOf(127) == valueOf(127) → true, valueOf(128) — false. Мораль: для вычислений — примитивы, для коллекций — обёртки.*

- Лямбда-выражения: почему переменные должны быть effectively final?

  → *Лямбда захватывает копию значения переменной, не ссылку на неё. Если переменная изменится после захвата — копия устареет → неочевидное поведение. Компилятор требует final или effectively final для защиты от ошибок. Workaround: AtomicInteger или массив из одного элемента (антипаттерн).*

- Stream API: что произойдёт при двух терминальных операциях?

  → *IllegalStateException: stream has already been operated upon or closed. Стрим одноразовый — после терминальной операции закрывается. Если нужно два результата — Collectors.teeing() (Java 12+) или два разных стрима из одного источника.*

- Parallel streams: в чём опасность?

  → *Все parallel streams делят общий ForkJoinPool.commonPool(). Если один параллельный стрим заблокировался на I/O — остальные ждут. Решение: кастомный ForkJoinPool: new ForkJoinPool(4).submit(() -> stream.parallel()...). Не использовать для I/O-bound задач.*

- Java 17/21: что нового для повседневной разработки?

  → *Records: immutable data-класс с equals/hashCode/toString. Sealed Classes: ограничение иерархии (exhaustive switch). Pattern Matching: instanceof без cast, switch по типам. Virtual Threads: миллионы лёгких потоков (I/O-bound). Sequenced Collections: addFirst(), getLast(), reversed(). Structured Concurrency: StructuredTaskScope для управления задачами.*

04 Многопоточность

Многопоточность — явный фокус вакансии (высоконагруженные проекты). Спрашивают глубоко: volatile, CAS, happens-before, пулы, Virtual Threads.

- volatile: что гарантирует и почему недостаточно для i++?

  → *Видимость (запрет кэширования), happens-before (volatile write → read), запрет reordering. i++ = read+increment+write — три операции. Между ними другой поток может прочитать старое значение. Решения: AtomicInteger (CAS), synchronized, LongAdder.*

- synchronized на двух методах одного объекта — будут ли работать параллельно?

  → *Нет. Оба используют один монитор — this. Пока один поток в methodA(), другой не может войти в methodB(). Решение: synchronized(privateLockA) и synchronized(privateLockB) — разные мониторы.*

- Чем synchronized(this) отличается от synchronized(privateLock)?

  → *this доступен извне — любой может случайно или намеренно synchronized(obj) на том же объекте. privateLock (private final Object lock = new Object()) — полный контроль, никто извне не может захватить.*

- happens-before: какие пары существуют?

  → *volatile write → read. synchronized release → acquire. Thread.start() → первая инструкция потока. Последняя инструкция → Thread.join(). final поля после конструктора (если this не утёк). Запись в ConcurrentHashMap → чтение. Без happens-before видимость НЕ гарантирована.*

- Race condition: как воспроизвести и как бороться?

  → *Check-then-act без атомарности: if (map.containsKey(k)) map.get(k). Между проверкой и получением другой поток может удалить. Решение: computeIfAbsent (атомарно). Воспроизвести: Thread.sleep(1) между check и act + два потока. Бороться: атомарные операции, synchronized, CAS.*

- Virtual Threads (Java 21): pinning problem.

  → *Virtual Thread на synchronized блоке → pinning (привязка к platform thread, теряется лёгкость). Причина: synchronized работает через монитор объекта, JVM не может «припарковать» виртуальный поток. Решение: ReentrantLock вместо synchronized. Virtual Threads идеальны для I/O-bound, НЕ для CPU-bound (нет preemption).*

- Structured Concurrency — что это?

  → *StructuredTaskScope (preview в Java 21): управление временем жизни задач. Все подзадачи привязаны к scope — при закрытии scope все незавершённые отменяются. ShutdownOnFailure: при первой ошибке все остальные отменяются. ShutdownOnSuccess: при первом успехе остальные отменяются. Замена разрозненным CompletableFuture.*

- Иммутабельность как способ потокобезопасности.

  → *Immutable-объекты потокобезопасны по определению — нет shared mutable state. Нет необходимости в synchronized/volatile. Примеры: String, Integer, LocalDate, record. В многопоточном коде: вместо изменения объекта — создание нового. CopyOnWriteArrayList — тоже вариант (иммутабельность массива при записи).*

05 Коллекции

- HashMap: расчёт бакета на примере.

  → *Формула: (n-1) & hash(key). Если capacity=16 (n=16): (15) & hash. 15 в бинарном = 1111. hash = 2_000_000_000: 2000000000 & 15 = 0 (последние 4 бита = 0000). Результат: bucket 0. Поэтому capacity — степень двойки, и hash дополнительно перемешивается (XOR верхних бит с нижними).*

- HashMap vs Hashtable vs LinkedHashMap vs WeakHashMap.

  → *HashMap: не потокобезопасен, null-ключ. Hashtable: потокобезопасен (synchronized на всё), устаревший, нет null. LinkedHashMap: порядок вставки (или access order для LRU). WeakHashMap: ключи через WeakReference — GC собирает неиспользуемые записи (для кэшей). ConcurrentHashMap — замена Hashtable.*

- Два одинарных индекса vs один составной — что лучше?

  → *Зависит от запроса. WHERE a=1 AND b=2: составной (a,b) — один Index Scan. Два одинарных: Bitmap Index Scan + Bitmap AND — медленнее. WHERE a=1 OR b=2: два одинарных — Bitmap OR. Составной (a,b) не поможет для WHERE b=2 (leftmost prefix). Общее правило: составной для AND-запросов, одинарные для OR или независимых WHERE.*

- ConcurrentModificationException: причина и решение.

  → *Модификация коллекции во время итерации не через итератор. modCount-счётчик: при следующем next() проверяется. Решения: Iterator.remove(), CopyOnWriteArrayList, ConcurrentHashMap, removeIf() (Java 8+), stream + filter + collect в новый список.*

- CopyOnWriteArrayList: плюсы и минусы.

  → *Плюсы: lock-free чтение (читатели не блокируются), fail-safe итератор (работает со snapshot). Минусы: при каждой записи — копирование всего массива → дорого. Подходит: много чтений, редкие записи (listeners, подписчики, конфигурации). Не подходит: частые записи.*

06 JVM, память и GC

- Структура памяти JVM: где что хранится?

  → *Heap: объекты (Young=Eden+Survivor, Old). Stack: фреймы методов (на каждый поток свой, LIFO), примитивы, ссылки. Metaspace: метаданные классов, статические поля. Code Cache: JIT-скомпилированный код. Direct Memory: NIO-буферы. StackOverflowError: переполнение стека (рекурсия). OOM: переполнение heap/metaspace/direct.*

- Generational ZGC — что нового?

  → *ZGC с Java 21 стал generational (раздельная сборка Young/Old). Паузы <1ms даже на терабайтных heap. Предыдущий ZGC был non-generational (собирал весь heap). Generational ZGC — стандарт для high-load в 2026. G1 — для случаев, когда ZGC overkill (heap <4 GB).*

- Как воспроизвести OOM?

  → *static Map без eviction: статический Map, добавляешь объекты в цикле → heap exhaustion. Бесконечное создание Thread: каждый поток = ~1MB стека → native OOM. Metaspace: динамическая генерация классов (CGLIB, ASM). Direct Memory: ByteBuffer.allocateDirect() без освобождения.*

- Виды ссылок: Strong, Soft, Weak, Phantom.

  → *Strong: обычная, GC не трогает. Soft: GC собирает при нехватке памяти (кэши). Weak: GC собирает при первой сборке (WeakHashMap). Phantom: get() всегда null, уведомление через ReferenceQueue (постфинализационная очистка). Циклические ссылки: GC соберёт — определяет достижимость от GC Roots, а не reference counting.*

07 Spring и транзакции

- Singleton-бин: потокобезопасен ли?

  → *НЕТ. Singleton означает один экземпляр на контекст, но это НЕ делает его потокобезопасным. Если есть mutable state (поля, не final) — race condition. Решения: stateless бины (без полей-состояний), ThreadLocal, synchronized, ConcurrentHashMap. В Spring MVC: контроллеры и сервисы — singleton, но stateless.*

- @Transactional на private методе — что произойдёт?

  → *Ничего — транзакция не создастся. CGLIB-прокси наследует класс, но private методы не переопределяемы → прокси их не видит. JDK Dynamic Proxy работает через интерфейс — private вообще не в интерфейсе. Решение: сделать метод package-private или public.*

- Два @Transactional метода в одном классе: A() вызывает B(). Сколько транзакций?

  → *Одна. Вызов this.B() минует прокси → @Transactional на B() не работает. B() выполняется в транзакции A() (Propagation.REQUIRED по умолчанию). Если нужна отдельная транзакция для B() — вынести в другой бин или self-injection через @Lazy.*

- Когда нужен REQUIRES_NEW?

  → *Когда нужна НЕЗАВИСИМАЯ транзакция, которая сохранится даже при rollback основной. Примеры: запись аудит-лога (должна остаться при ошибке), отправка уведомления, счётчик попыток. Минус: две открытые транзакции одновременно → два соединения из пула.*

- @Transactional + долгий REST-вызов внутри — чем опасно?

  → *Транзакция удерживает соединение к БД всё время REST-вызова (секунды). Пул соединений исчерпается → все потоки ждут → сервис висит (connection starvation). Решение: read → close transaction → REST → open transaction → write. Или вынести REST-вызов за @Transactional.*

- В какой момент происходит коммит @Transactional?

  → *После успешного завершения метода (без исключений). Прокси: try { beginTransaction; target.method(); commit; } catch { rollback; }. Rollback по умолчанию: RuntimeException и Error. Checked — НЕ откатывают (нужен rollbackFor). flush != commit: flush отправляет SQL, commit фиксирует.*

- AOP: кейс «замерить время всех методов сервиса».

  → *@Aspect + @Around("execution(* com.example.service.*.*(..))"). ProceedingJoinPoint: long start = System.nanoTime(); Object result = pjp.proceed(); long elapsed = System.nanoTime() - start; log.info("{} took {} ms", methodName, elapsed/1_000_000); return result. В проде: Micrometer @Timed вместо ручного AOP.*

- Конфликт зависимостей: как искать?

  → *mvn dependency:tree — показывает дерево зависимостей. Ищем разные версии одной библиотеки. Gradle: ./gradlew dependencies. Решение: exclusion в Maven/Gradle, dependencyManagement/BOM, force в Gradle. Классика: Jackson 2.14 из Spring vs Jackson 2.12 из другой библиотеки.*

08 Hibernate и JPA

- JPA vs Hibernate.

  → *JPA — спецификация (набор интерфейсов и аннотаций). Hibernate — реализация (конкретная библиотека). EntityManager — JPA-интерфейс, Session — Hibernate-интерфейс. Можно сменить Hibernate на EclipseLink без изменения кода (если использовать только JPA API).*

- Каскадные операции: что делает CascadeType.ALL?

  → *PERSIST + MERGE + REMOVE + REFRESH + DETACH. Опасность: CascadeType.REMOVE при OneToMany — удаление родителя удалит ВСЕХ детей. Если детей 10 000 — 10 000 DELETE запросов. orphanRemoval=true — удаляет «осиротевших» детей при удалении из коллекции.*

- Entity как ключ HashMap — что может пойти не так?

  → *Если equals/hashCode используют lazy-связи (ManyToOne, OneToMany) — при первом вызове hashCode загрузится N связанных сущностей (N+1). Хуже: если связи двусторонние — StackOverflowError (бесконечная рекурсия). Решение: equals/hashCode только по @Id (или бизнес-ключу без связей). Lombok @EqualsAndHashCode(onlyExplicitlyIncluded=true).*

- JdbcTemplate — когда уместен вместо Hibernate?

  → *Сложные аналитические запросы (оконные функции, CTE). Batch-операции (тысячи INSERT). Запросы, где ORM генерирует неоптимальный SQL. Миграция legacy-кода. Простые CRUD без сложных связей. В T1: JdbcTemplate используют для ETL и репортинга, Hibernate — для бизнес-логики.*

// JavaJub

Это только начало гайда.

T1 Иннотех · Junior–Middle

Полная версия — бесплатно в Telegram-канале:

- все вопросы с ответами по грейду
- задачи: live-coding, SQL, System Design — с разбором
- чек-лист «за 2 часа до собеса»

@java_jub

t.me/java_jub · подписка бесплатная