// JavaJub

Разбор задачи

X5 Tech · Senior

4 из 8 багов в сервисе расчёта скидок

Чёрная пятница в Пятёрочке · код-ревью под Senior

Что в этом документе

4 разбора с правильным кодом · тизеры на остальные 4 · главный коварный — в Pro

Бесплатная версия из серии Java Jub Pro · Полный разбор + код + чек-лист в Pro

Содержание

01 Постановка задачи и контекст

Эту задачу дают на Senior-собеседовании в X5 Tech и Farzoom. Снаружи выглядит как обычное «найди баги в коде на parallelStream». Внутри — самая многослойная задача из X5-серии: 8 разных багов в 15 строках, причём главный — не там, где обычно ищут. Если ты Senior и пропустишь главный — это сигнал интервьюеру, что в проде у тебя были бы инциденты на деньги.

В этой бесплатной версии разбираем 4 бага из 8 — техническая база, на которой отсеивают Middle. Оставшиеся 4 — самые провокационные, на них валятся даже Senior, и среди них тот САМЫЙ главный коварный, ради которого вся задача и существует на собесе. Полный разбор каждого с примерами правильного кода — в Java Jub Pro.

Контекст для интервью

- Компания: X5 Tech / Farzoom, Senior Java.
- Бренд внутри X5: Пятёрочка, расчёт скидок для Чёрной пятницы.
- Стек: Java 21, Spring Boot 3, PostgreSQL.
- Масштаб: 50 000 SKU, цены пересчитываются перед открытием акции.
- Тайминг на собесе: 20 минут на разбор.
- Сложность: medium-hard.

Исходный код

Ниже сервис, который прислал автор. Задача — найти баги. По уровню Senior их должно быть 6+, идеально 8. Прочти код медленно. На первый взгляд он выглядит «нормально», компилируется и даже отрабатывает. Запомни это ощущение — оно главный враг на этой задаче.

```
@Service
public class BlackFridayPricingService {

    private Random random = new Random();

    public List<BigDecimal> calculateDiscountPrices(int[] basePrices) {

        return Arrays.stream(basePrices)
            .parallel()
            .filter(price -> price % 2 != 0)
            .mapToObj(price -> {
                try {
                    Thread.sleep(50);
                } catch (InterruptedException e) {
                    // ignore
                }
                double discount = random.nextDouble() * 0.5;
                return new BigDecimal(price * (1 - discount));
            })
            .collect(Collectors.toList());
    }
}
```

Стратегия разбора на собесе

1. Прочитай код 2 раза молча, без комментариев. Первый раз — общее понимание. Второй — примечай узкие места.

2. Начни вслух с типов данных (BigDecimal, double, int). Это покажет интервьюеру, что ты думаешь о финансовой корректности до многопоточки.
3. Дальше — многопоточный блок. Здесь Senior должен идти не «парадного» путём (parallel плох), а глубоким: ForkJoinPool, contention, escape carrier (Java 21).
4. Потом Random — и его две проблемы (contention + детерминизм).
5. В конце — стиль и архитектура: filter, final-поля, DI.

> **ФИШКА · Что показывает Senior на этой задаче**
>
> Задаёт уточняющие вопросы перед тем, как чертить — это первый признак опыта.
>
> Отделяет «нашёл баг» от «обосновал что будет в проде» — второе ценится выше.
>
> Начинает с самого опасного бага, не с самого очевидного.
>
> Знает Goetz «Java Concurrency in Practice» — это видно по тому, как он говорит про InterruptedException.

02 Баг #1 — contention на shared Random

Начнём с самого контринтуитивного бага в коде. Технически java.util.Random thread-safe — это написано в Javadoc. На собесе кандидат говорит «ну, Random безопасен, я проверил». Дальше идут JMH-бенчмарки, которые показывают: parallelStream с shared Random может работать МЕДЛЕННЕЕ, чем тот же код в sequential-режиме. Это не баг производительности, это баг ПОНИМАНИЯ thread-safety.

Где баг

```
private Random random = new Random();              // <-- один на сервис

// внутри parallelStream:
double discount = random.nextDouble() * 0.5;       // <-- 7 потоков дёргают одну ссылку
```

JMH-цифры — увидеть глазами

Реальный замер на 8-ядерной машине, 1 миллион вызовов nextDouble:

Подход	Время	Замечание
sequential (без parallel)	~12 ms	baseline
parallel + shared java.util.Random	~25 ms	× медленнее sequential!
parallel + ThreadLocalRandom.current()	~2 ms	× в 6 раз быстрее sequential

Эти цифры контринтуитивны. Если ты раньше думал, что .parallel() ВСЕГДА ускоряет код — это контрпример. И он не теоретический. Любой может проверить через jmh за 10 минут. На собесе X5 спросят именно это: «почему parallel на shared Random медленнее, чем sequential?». Если ответа нет — Senior-оффер не светит.

Что не так — внутреннее устройство Random

- **Как java.util.Random становится thread-safe?**
  → java.util.Random формально thread-safe — это написано в Javadoc. Но реализация потокобезопасности — через атомарный CAS на одном поле AtomicLong seed. Каждый вызов nextDouble() внутри делает Compare-And-Set операцию на этом seed. Когда 7 потоков параллельно долбят один Random — все 7 бьются за один и тот же CAS на одной переменной. Это называется contention (конкуренция за ресурс). CPU тратит больше времени на координацию между потоками, чем на полезную работу.

- **Почему shared Random может быть медленнее sequential?**
  → В sequential один поток последовательно дёргает nextDouble — CAS всегда успешен, никаких retry. В parallel 7 потоков одновременно пытаются обновить seed. Только один CAS успешен, остальные 6 потоков retry-ят. На следующей итерации картина повторяется: один успешен, шесть retry. CPU занят координацией, прогресса мало. Плюс — сами CPU-кэши проседают, потому что переменная seed постоянно инвалидируется через MESI-протокол между ядрами. Итог: parallel хуже sequential на 30-100%, в зависимости от железа.

Решение: ThreadLocalRandom

```
double discount = ThreadLocalRandom.current().nextDouble() * 0.5;
```

- **Как устроен ThreadLocalRandom?**
  → ThreadLocalRandom использует отдельный seed для каждого потока — хранится в полях класса Thread (это специальная JVM-оптимизация, не обычный ThreadLocal). Метод current() возвращает экземпляр для текущего потока без synchronized и без CAS — просто читает из локального состояния потока. Поэтому нет contention, каждый поток работает со своим seed независимо. Создан специально для concurrent кода. Появился в Java 7 (java.util.concurrent.ThreadLocalRandom). С Java 17 появился общий интерфейс RandomGenerator — рекомендуемый способ работать с ГСЧ в новом коде.

- **Когда использовать что — Random, SecureRandom, ThreadLocalRandom?**
  → Random — для однопоточного кода с обычной случайностью (тесты, моки, простой код). ThreadLocalRandom — для многопоточного кода с обычной случайностью (concurrent workloads, parallel stream). SecureRandom — для криптографически стойких чисел: токены аутентификации, ID транзакций, salt для паролей. SecureRandom медленнее (использует энтропию ОС через /dev/urandom), но непредсказуемо. Никогда не используй Random для session ID или паролей — последовательность вычисляется по seed.

Что должен видеть Senior на code review

- private Random random = new Random() как поле сервиса — автоматический маркер для проверки: «используется ли в parallel-контексте?»
- Если в коде есть .parallel() / parallelStream() рядом с shared Random — это блокирующее замечание на review.
- Правильно: ThreadLocalRandom.current() внутри лямбды или RandomGenerator через DI.
- В JMH-бенчмарке любого concurrent-кода — обязательно проверять с разными RNG, иначе микро-бенчмарк не показывает реальную картину прода.

> **ФИШКА · Что сказать на собесе про этот баг**
> «java.util.Random thread-safe формально, но через CAS на одном seed».
> «В parallelStream это даёт contention — параллельный код медленнее sequential».
> «Решение — ThreadLocalRandom.current() (Java 7+) или RandomGenerator (Java 17+)».
> «SecureRandom — только для криптографии, не для бизнес-логики».
> Бонус — упомянуть JMH как стандарт измерения concurrent-производительности.

03 Баг #2 — parallelStream на общем ForkJoinPool

Технический баг про concurrency. Большинство кандидатов видят .parallel() и думают «ну, многопоточка — это плохо тут». Senior должен идти глубже: какой пул, какой эффект на остальное приложение, как фиксить правильно.

Где баг

```
return Arrays.stream(basePrices)
    .parallel()         // <-- использует общий ForkJoinPool.commonPool()
    .filter(...)
    ...;
```

Что не так

- **Почему .parallel() — проблема?**
  → parallel() и parallelStream() под капотом используют ForkJoinPool.commonPool() — это один общий пул потоков на ВСЁ JVM-приложение. Размер по умолчанию = Runtime.availableProcessors() - 1 (на 8-ядерной машине это 7 потоков). Все parallelStream-ы во всех сервисах используют этот один пул. Если наш сервис запустил обработку 50 000 SKU с тяжёлой операцией внутри — он на минуты блокирует общий пул, и другие параллельные операции в других сервисах встают. Это классический случай global state в Java.

- **Чем это грозит в проде?**
  → Сценарий: микросервис обрабатывает 1000 RPS обычных запросов. Каждый из них где-то использует parallelStream (агрегация, фильтр коллекции). Параллельно стартует background-job на пересчёт скидок: 50 000 SKU × тяжёлая операция / 7 потоков = несколько минут. На эти минуты common pool целиком занят background-задачей. Все обычные parallelStream становятся эффективно sequential — задачи стоят в очереди commonPool, ждут освобождения потоков. p99 latency API в десятки раз вырастает. И никаких алертов: технически ничего не упало.

- **Как исправить?**
  → Два рабочих варианта. Первый — создать свой ForkJoinPool под конкретную задачу и передать параллельный стрим внутрь его submit. Второй (Java 21) — использовать newVirtualThreadPerTaskExecutor: виртуальные потоки решают проблему «пул заканчивается», их может быть миллион. Для batch-задач с blocking IO виртуальные потоки — лучший выбор. Для CPU-bound — ForkJoinPool с разумным числом потоков (обычно число ядер).

Решение 1: свой ForkJoinPool

```
@Service
public class BlackFridayPricingService {

    private final ForkJoinPool pricingPool = new ForkJoinPool(
        Runtime.getRuntime().availableProcessors()     // выделенный пул
    );

    public List<BigDecimal> calculate(int[] basePrices) {
        try {
            return pricingPool.submit(() ->
                Arrays.stream(basePrices)
                    .parallel()
```

```
                    .mapToObj(this::priceFor)
                    .collect(Collectors.toList())
            ).get();
        } catch (InterruptedException | ExecutionException e) {
            Thread.currentThread().interrupt();
            throw new RuntimeException(e);
        }
    }

    @PreDestroy
    void shutdown() {
        pricingPool.shutdown();    // не забыть закрыть!
    }
}
```

Решение 2: virtual threads (Java 21+)

```
public List<BigDecimal> calculate(int[] basePrices) {
    try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
        List<Future<BigDecimal>> futures = new ArrayList<>(basePrices.length);
        for (int price : basePrices) {
            futures.add(executor.submit(() -> priceFor(price)));
        }
        List<BigDecimal> result = new ArrayList<>(basePrices.length);
        for (Future<BigDecimal> f : futures) {
            result.add(f.get());
        }
        return result;
    } catch (InterruptedException | ExecutionException e) {
        Thread.currentThread().interrupt();
        throw new RuntimeException(e);
    }
}
```

- **Когда что выбирать — свой ForkJoinPool или virtual threads?**
  → ForkJoinPool — для CPU-bound задач: математика, парсинг, трансформации без IO. Размер пула = число ядер. Wow-эффекта не даст, но не блокирует common pool. Virtual threads — для IO-bound: блокирующий JDBC, HTTP-вызовы, файловое IO. Их можно создавать миллионы — виртуальный поток ~few КБ против ~1 МБ у платформенного. Когда виртуальный поток блокируется на IO, он снимается с carrier (платформенного) потока, который тут же берёт другой виртуальный. CPU не простаивает.

> **ФИШКА · Что сказать на собесе про parallelStream**
> .parallel() использует ОБЩИЙ ForkJoinPool.commonPool() на всю JVM.
> Один тяжёлый parallelStream может «забить» пул для всего приложения.
> Решение: свой ForkJoinPool через submit, либо virtual threads (Java 21+).
> Для CPU-bound — выделенный FJP. Для IO-bound — virtual threads.

04 Баг #3 — magic filter в стриме

Баг «стилистический», но для Senior он принципиален — это про инженерную культуру, читаемость и поддержку. Если кандидат его не замечает, это знак: код писали без code review.

Где баг

```
.filter(price -> price % 2 != 0)    // <-- что это вообще?
```

Что не так

- **В чём суть проблемы?**
  → Это «магическое» условие — фильтр, смысл которого непонятен из кода. «Берём только нечётные цены» — это бизнес-правило или баг? Если правило — должно быть документировано или вынесено в named-predicate. Если баг — почему оно в коде? На code review такой фрагмент должен заставить рецензента остановиться и задать вопрос автору: «А что должно произойти с чётными ценами? Они не попадают в результат — это намеренно?». Если автор не может объяснить — это явный баг.

- **Какие могут быть «легитимные» причины?**
  → Их немного, и все маловероятны. (1) Тесты на edge cases — фильтр оставили из отладочного кода. (2) Какой-то странный бизнес-юзкейс типа «скидку даём только на товары с нечётной ценой» — но это нонсенс с точки зрения маркетинга. (3) Способ симулировать ~50% выборку через простое условие — тогда нужно правильное sampling. В реальном коде ВСЕГДА должен быть смысл у каждой строки. Условие без объяснения — это долг для будущих разработчиков.

- **Как правильно?**
  → Два варианта в зависимости от ответа на «зачем»: (1) Если фильтр случайный или для отладки — удалить полностью. Если ничего не сломается — фильтр был лишним. (2) Если фильтр действительно нужен бизнесу — вынести в named-метод с понятным именем, добавить Javadoc. Это сразу превращает «магию» в документированный код: filter(this::isEligibleForDiscount). А внутри isEligibleForDiscount — реальная логика с объяснением.

Как должно быть

```
.filter(this::isEligibleForDiscount)

/**
 * Товар eligible для Чёрной пятницы, если он не относится к stop-list:
 * — алкоголь и табак (по закону скидки запрещены)
 * — социальные товары (хлеб, молоко — регулирование цен)
 * — товары собственного производства (нет маржи для скидки)
 */
private boolean isEligibleForDiscount(int price) {
    // конкретная бизнес-логика
    return ...;
}
```

Почему это важно для Senior

- **Что показывает Senior на этом моменте?**

→ Способность отделять «работает» от «правильно». Junior смотрит на filter и думает: «ну это просто фильтр, всё ок». Middle может заметить, что условие странное, но не настоит на изменении. Senior останавливается: «это бизнес-правило или баг? Если правило — опишем явно. Если баг — уберём». Это часть code review культуры. Если в коде нельзя понять «зачем» — это код, который через год никто не поймёт. И через два года его трогать боятся, начинается legacy.

ФИШКА · Правило для code review

На любом code review задавай себе два вопроса: «Это работает правильно?» и «А ЗАЧЕМ это вообще делается?». Второй важнее.

Если на «зачем» нет очевидного ответа — спрашивай у автора.

Идеальный код объясняет себя сам. Если не объясняет — нужен комментарий или рефакторинг в named-метод.

Просто имена методов уже документация: isEligibleForDiscount, isPremiumCustomer, hasActiveSubscription.

// JavaJub

Это только начало гайда.

X5 Tech · Senior

Полная версия — бесплатно в Telegram-канале:

- полный разбор всех багов + правильный код
- чек-лист код-ревью под Senior
- разбор самого коварного бага

@java_jub

t.me/java_jub · подписка бесплатная