Concurrency (Многопоточность) в Java

1. Потоки (Threads, Runnable, Callable)

Поток (`Thread`) — это наименьшая единица выполнения в Java. JVM позволяет выполнять несколько потоков одновременно (параллельно или псевдопараллельно на одноядерных CPU).

1.1. Thread (Наследование)

```
class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("Thread is running: " + Thread.currentThread().getName());
    }
}

// Запуск
MyThread thread = new MyThread();
thread.start(); // Запускает новый поток
```

Особенности:

- Наследование от `Thread` ограничивает возможность наследования других классов (т.к. Java не поддерживает множественное наследование).
- Прямой доступ к методам `Thread` (`sleep()`, `interrupt()` и др.).

1.2. Runnable (Интерфейс)

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Runnable is running: " +
Thread.currentThread().getName());
    }
}

// Запуск
Thread thread = new Thread(new MyRunnable());
thread.start();
```

Преимущества:

- Более гибкий подход (можно реализовывать другие интерфейсы).
- Совместим с `ExecutorService` (пулы потоков).

1.3. Callable (Возвращает результат)

```
class MyCallable implements Callable<String> {
    @Override
    public String call() throws Exception {
        return "Result from Callable";
    }
}
```

```
// Использование с ExecutorService
ExecutorService executor = Executors.newSingleThreadExecutor();
Future<String> future = executor.submit(new MyCallable());
System.out.println(future.get()); // Блокирует поток, пока результат не будет получен
executor.shutdown();
```

Отличия от Runnable:

- Может возвращать результат (`Future<T>`).
- Может выбрасывать исключения.

---

2. Синхронизация (synchronized, Lock, volatile)

2.1. `synchronized` (Блокировка на уровне метода или блока)

```
class Counter {
    private int count = 0;

    // Синхронизированный метод
    public synchronized void increment() {
        count++;
    }

    // Синхронизированный блок
    public void decrement() {
        synchronized (this) {
            count--;
        }
    }
}
```

Как работает:

- Только один поток может выполнять синхронизированный метод/блок для одного объекта.
- `synchronized` гарантирует **видимость изменений** между потоками (happens-before).

2.2. `Lock` (ReentrantLock, ReadWriteLock)

```
private final Lock lock = new ReentrantLock();

public void performTask() {
    lock.lock();
    try {
        // Критическая секция
    } finally {
        lock.unlock(); // Всегда в finally!
    }
}
```

Преимущества перед `synchronized`:

- Возможность **прерываемой блокировки** (`lockInterruptibly()`).

- **Таймауты** (`tryLock(1, TimeUnit.SECONDS)`).
- **Честная очередь** (fairness).

2.3. `volatile` (Гарантированная видимость изменений)

```
private volatile boolean flag = false;
```

Когда использовать:

- Когда переменная изменяется одним потоком, а читается многими.
- Гарантирует, что изменения **видны сразу** всем потокам (но не атомарность!).

---

3. Пул потоков (ExecutorService, ThreadPoolExecutor)

3.1. ExecutorService (Интерфейс)

```
ExecutorService executor = Executors.newFixedThreadPool(4); // Пул из 4 потоков
executor.submit(() -> System.out.println("Task running"));
executor.shutdown(); // Остановка после завершения задач
```

Типы пулов:

- `newFixedThreadPool(n)` — фиксированное число потоков.
- `newCachedThreadPool()` — создает потоки по мере необходимости.
- `newSingleThreadExecutor()` — один поток (очередь задач).
- `newScheduledThreadPool(n)` — планировщик задач.

3.2. ThreadPoolExecutor (Гибкая настройка)

```
ThreadPoolExecutor executor = new ThreadPoolExecutor(
    2, // corePoolSize
    4, // maximumPoolSize
    60, TimeUnit.SECONDS, // keepAliveTime
    new ArrayBlockingQueue<>(10) // очередь задач
);
```

---

4. Потокобезопасные коллекции

4.1. ConcurrentHashMap

```
ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();
map.put("key", 1);
```

Особенности:

- Сегментированная блокировка (высокая производительность).
- **Потокобезопасные операции** (`putIfAbsent`, `compute`).

4.2. CopyOnWriteArrayList

```
CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>();
list.add("item");
```

Как работает:

- При изменении создается **новая копия** массива.
- Подходит для **часто читаемых, редко изменяемых** списков.

---

5. Атомарные операции (Atomic-классы)

5.1. AtomicInteger, AtomicLong

```
AtomicInteger counter = new AtomicInteger(0);
counter.incrementAndGet(); // Атомарное увеличение
```

Особенности:

- Основаны на **CAS (Compare-And-Swap)** процессора.
- Нет блокировок (non-blocking).

5.2. AtomicReference

```
AtomicReference<String> ref = new AtomicReference<>("initial");
ref.compareAndSet("initial", "updated"); // Атомарное сравнение и замена
```

---

6. Проблемы многопоточности

6.1. Race Condition (Состояние гонки)

```
if (x == 10) { // Поток 1 проверяет
    x = 0;     // Поток 2 меняет x до выполнения этой строки
}
```

Решение: `synchronized`, атомарные операции.

6.2. Deadlock (Взаимная блокировка)

```
// Поток 1: lock(A), затем пытается lock(B)
// Поток 2: lock(B), затем пытается lock(A)
```

Решение:

- Упорядоченная блокировка (`lock(A) -> lock(B)` везде).
- Таймауты (`tryLock`).

6.3. Livelock (Активная блокировка)

```
// Потоки постоянно меняют состояние, но не прогрессируют
while (!success) {
```

```
    tryAgain(); // Бесконечные попытки
}
```

Решение: Рандомизированные задержки.

Вывод

- **Потоки:** `Thread`, `Runnable`, `Callable` (возвращает результат).
- **Синхронизация:** `synchronized`, `Lock`, `volatile`.
- **Пул потоков:** `ExecutorService`, `ThreadPoolExecutor`.
- **Потокобезопасные коллекции:** `ConcurrentHashMap`, `CopyOnWriteArrayList`.
- **Атомарные операции:** `AtomicInteger`, `AtomicReference`.
- **Проблемы:** Race condition, deadlock, livelock.

Многопоточность требует аккуратной работы с **разделяемыми ресурсами** и правильного выбора механизмов синхронизации.