1. **Многопоточность. Класс Thread, интерфейс Runnable. Модификатор synchronized.**

**📌 1. Что такое многопоточность в Java**

**Многопоточность** — это способность программы выполнять **несколько потоков (threads)** одновременно. Поток — это независимая последовательность выполнения. В Java каждый поток реализуется как экземпляр класса Thread.

Преимущества многопоточности:

* Параллельная обработка задач (например, скачивание файлов, работа с UI, вычисления)
* Улучшение производительности на многоядерных процессорах
* Повышение отзывчивости приложений

**📘 2. Класс Thread**

Java предоставляет встроенный класс java.lang.Thread для создания и управления потоками.

**🔧 Способы создания потока через Thread:**

**Способ 1: Наследование от Thread**

java

КопироватьРедактировать

class MyThread extends Thread {

public void run() {

System.out.println("Поток " + Thread.currentThread().getName() + " запущен");

}

}

public class Main {

public static void main(String[] args) {

MyThread thread1 = new MyThread();

thread1.start(); // запускает новый поток

}

}

⚠️ start() запускает поток асинхронно. Нельзя вызывать run() напрямую — это просто вызов метода, а не запуск потока.

**🧩 3. Интерфейс Runnable**

Более гибкий способ — **реализовать интерфейс Runnable**. Особенно полезно, когда ваш класс уже наследует другой.

java

КопироватьРедактировать

class MyRunnable implements Runnable {

public void run() {

System.out.println("Runnable поток " + Thread.currentThread().getName());

}

}

public class Main {

public static void main(String[] args) {

Thread thread = new Thread(new MyRunnable());

thread.start();

}

}

Преимущество Runnable — можно использовать лямбды, объединять с другими классами и передавать как объект.

**🔐 4. Синхронизация и модификатор synchronized**

Когда несколько потоков обращаются к **общему ресурсу (например, переменной)**, возникает **состояние гонки (race condition)**. Чтобы этого избежать, используют synchronized.

**✅ Пример без synchronized (ошибка):**

java

КопироватьРедактировать

class Counter {

int count = 0;

public void increment() {

count++;

}

}

Если несколько потоков вызовут increment() одновременно, count может быть некорректным.

**🔒 Пример с synchronized:**

java

КопироватьРедактировать

class Counter {

private int count = 0;

public synchronized void increment() {

count++;

}

public int getCount() {

return count;

}

}

Или можно синхронизировать **только блок**:

java

КопироватьРедактировать

public void increment() {

synchronized (this) {

count++;

}

}

🔑 synchronized блокирует объект — только один поток может войти в блок синхронизации в конкретный момент.

**🚧 Пример полной программы с несколькими потоками и synchronized**

java

КопироватьРедактировать

class Counter {

private int count = 0;

public synchronized void increment() {

count++;

}

public int getCount() {

return count;

}

}

public class Main {

public static void main(String[] args) throws InterruptedException {

Counter counter = new Counter();

Runnable task = () -> {

for (int i = 0; i < 1000; i++) {

counter.increment();

}

};

Thread t1 = new Thread(task);

Thread t2 = new Thread(task);

t1.start();

t2.start();

t1.join(); // Ждём завершения

t2.join();

System.out.println("Итоговое значение: " + counter.getCount());

}

}

**🧠 Дополнительно: особенности и подводные камни**

**❗ Нельзя повторно запускать поток:**

java

КопироватьРедактировать

Thread t = new Thread();

t.start(); // ok

t.start(); // IllegalThreadStateException

**⚠️ synchronized может вызывать deadlock — взаимную блокировку. Следи за порядком захвата ресурсов.**

**✔️ Лучше использовать ExecutorService, Callable, Future, ReentrantLock, если нужна более гибкая работа — это входит в java.util.concurrent (если интересно, могу рассказать об этом отдельно).**

**🧾 Резюме**

| **Элемент** | **Назначение** |
| --- | --- |
| Thread | Класс для создания потока, можно переопределить run() |
| Runnable | Интерфейс с методом run(), передаётся в Thread |
| synchronized | Модификатор и блок для синхронизации доступа к ресурсам |

1. **Методы wait(), notify() класса Object, интерфейсы Lock и Condition**

**🧵 Часть 1: wait(), notify(), notifyAll() (класс Object)**

**🔑 Ключевые свойства:**

* Эти методы **унаследованы от Object**, т.к. любой объект может быть монитором.
* Используются **внутри synchronized блока или метода**, иначе: IllegalMonitorStateException.
* Поток **освобождает монитор и уходит в ожидание** (wait()), пока его не разбудит другой поток (notify()/notifyAll()).

**🔁 Пример использования: классическая задача "Производитель-Потребитель"**

java

КопироватьРедактировать

class Store {

private int product = 0;

public synchronized void put() throws InterruptedException {

while (product >= 1) {

wait(); // ждём, пока кто-то не заберёт товар

}

product++;

System.out.println("Производитель добавил товар");

notify(); // будим потребителя

}

public synchronized void get() throws InterruptedException {

while (product < 1) {

wait(); // ждём, пока появится товар

}

product--;

System.out.println("Потребитель забрал товар");

notify(); // будим производителя

}

}

public class Main {

public static void main(String[] args) {

Store store = new Store();

Thread producer = new Thread(() -> {

try {

for (int i = 0; i < 5; i++) store.put();

} catch (InterruptedException e) { e.printStackTrace(); }

});

Thread consumer = new Thread(() -> {

try {

for (int i = 0; i < 5; i++) store.get();

} catch (InterruptedException e) { e.printStackTrace(); }

});

producer.start();

consumer.start();

}

}

**💡 Важные замечания:**

* wait() **освобождает монитор** — поток временно выходит из synchronized блока.
* notify() **пробуждает** один поток, находящийся в wait() на том же объекте.
* notifyAll() — пробуждает **все** потоки, ожидающие на объекте.

**🔒 Часть 2: Lock и Condition (java.util.concurrent.locks)**

Начиная с Java 5, появился более **гибкий и мощный способ** синхронизации — интерфейсы Lock и Condition.

**🔧 Интерфейс Lock**

Аналог synchronized, но с:

* ручным управлением блокировкой (lock() / unlock())
* возможностью прерывания или таймаута
* несколькими Condition на один объект

**Пример:**

java

КопироватьРедактировать

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

class Counter {

private int count = 0;

private final Lock lock = new ReentrantLock();

public void increment() {

lock.lock(); // ручная блокировка

try {

count++;

} finally {

lock.unlock(); // всегда освобождай!

}

}

public int getCount() {

return count;

}

}

**🧭 Интерфейс Condition**

Это **эквивалент wait()/notify()**, но работает с Lock вместо synchronized.

**Полный пример (аналог "Производитель-Потребитель"):**

java

КопироватьРедактировать

import java.util.concurrent.locks.\*;

class Store {

private int product = 0;

private final Lock lock = new ReentrantLock();

private final Condition notEmpty = lock.newCondition();

private final Condition notFull = lock.newCondition();

public void put() throws InterruptedException {

lock.lock();

try {

while (product >= 1)

notFull.await(); // аналог wait()

product++;

System.out.println("Производитель добавил товар");

notEmpty.signal(); // аналог notify()

} finally {

lock.unlock();

}

}

public void get() throws InterruptedException {

lock.lock();

try {

while (product < 1)

notEmpty.await();

product--;

System.out.println("Потребитель забрал товар");

notFull.signal();

} finally {

lock.unlock();

}

}

}

**📊 Сравнение synchronized vs Lock/Condition**

| **Характеристика** | **synchronized + wait()** | **Lock + Condition** |
| --- | --- | --- |
| Автоматическая разблокировка | ✅ Да (по выходу из блока) | ❌ Нет (нужно вручную unlock()) |
| Несколько Condition | ❌ Нет | ✅ Да (можно несколько Condition) |
| Поддержка таймаутов | ❌ Ограничена | ✅ Встроена (await(timeout)) |
| Прерываемость ожидания | ❌ Сложно | ✅ Прямо (lockInterruptibly()) |
| Простой синтаксис | ✅ Очень | ❌ Чуть более сложный |

**🧾 Вывод**

**Когда использовать synchronized / wait() / notify():**

* В простых задачах с одним монитором
* Для краткости и читаемости
* Когда нет необходимости в прерывании или таймаутах

**Когда использовать Lock и Condition:**

* Когда нужна гибкость, таймауты, несколько очередей ожидания
* При высоконагруженной многопоточности
* Для избежания вложенных synchronized-блоков

**🧠 Что такое монитор в Java**

**🔹 Определение:**

**Монитор** — это механизм **синхронизации потоков**, встроенный в каждый объект Java. Он обеспечивает **взаимное исключение (mutual exclusion)** при доступе к критическим участкам кода.

**🔸 Физически: монитор — это встроенный объектный "замок", к которому могут "войти" потоки. Только один поток может удерживать его в данный момент.**

**🔐 Пример:**

java

КопироватьРедактировать

synchronized(obj) {

// критическая секция

}

В этом коде:

* obj — это объект-монитор
* поток "входит" в монитор, если он свободен
* другие потоки **ждут**, пока текущий поток не выйдет

**💡 Связь с wait(), notify():**

Методы wait() / notify() / notifyAll() **работают только в мониторе**:

* wait() — поток временно **покидает монитор** и ждёт сигнала
* notify() — **будит** один ожидающий поток на этом мониторе
* notifyAll() — **будит всех**, кто ждал

**🔒 Теперь подробнее о Lock из java.util.concurrent.locks**

**🔸 Что такое Lock?**

Lock — это **интерфейс**, который предоставляет **ручной контроль** над блокировкой ресурса. Это **альтернатива synchronized**, но **более гибкая**.

Он определён в пакете:

java

КопироватьРедактировать

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

**✅ Основные методы интерфейса Lock**

| **Метод** | **Назначение** |
| --- | --- |
| lock() | Захват блокировки. Блокирует, пока не получит |
| unlock() | Освобождает блокировку. **Обязательно** вызывать! |
| tryLock() | Пытается захватить блокировку, но не ждёт |
| lockInterruptibly() | Захватывает блокировку, но может быть прерван |
| newCondition() | Создаёт объект Condition (альтернатива wait()/notify()) |

**📌 Пример с ReentrantLock**

java

КопироватьРедактировать

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

class Counter {

private int count = 0;

private final Lock lock = new ReentrantLock();

public void increment() {

lock.lock(); // захват блокировки

try {

count++;

} finally {

lock.unlock(); // обязательно!

}

}

public int getCount() {

return count;

}

}

⚠️ Никогда не забудь unlock() в finally — иначе будет **deadlock**.

**🔁 Отличие от synchronized**

| **Свойство** | **synchronized** | **Lock + ReentrantLock** |
| --- | --- | --- |
| Управление | Автоматическое | Ручное (lock() / unlock()) |
| Прерываемость | Нет | Да (lockInterruptibly()) |
| Попытка без ожидания | Нет | Да (tryLock()) |
| Несколько Condition на объект | Нет | Да (newCondition()) |
| Повторный захват (reentrant) | Да | Да (ReentrantLock) |

**🌀 Внутреннее устройство ReentrantLock**

* **Reentrant** = поток может захватить замок **повторно**, если уже держит его.
* Под капотом реализован с использованием **счетчиков захвата**, **очереди ожидания**, **volatile** и **CAS (compare-and-swap)** операций.
* Это низкоуровневая, но **очень быстрая и гибкая альтернатива мониторам**.

**📘 Пример с tryLock()**

java

КопироватьРедактировать

if (lock.tryLock()) {

try {

// критическая секция

} finally {

lock.unlock();

}

} else {

System.out.println("Не удалось получить блокировку — занят!");

}

**✅ Когда использовать Lock, а не synchronized?**

Используй Lock, если:

* Нужна **гибкость** (например, таймаут, отмена ожидания)
* Нужны **несколько условных переменных (Condition)** на одну блокировку
* Нужна **высокая производительность** (Lock масштабируется лучше на многоядерных машинах)
* Тебе важно **не блокироваться навсегда** (через tryLock)

**🧾 Вывод:**

* **Монитор** — это встроенный механизм синхронизации через synchronized.
* wait(), notify() работают **только внутри монитора**.
* Lock/Condition — это **ручной и более гибкий механизм**, аналогичный, но расширенный.

1. **Классы-сихронизаторы из пакета java.util.concurrent.**

**🧩 1. Что такое синхронизаторы?**

**Синхронизаторы** — это специальные классы из пакета java.util.concurrent, предназначенные для координации работы **нескольких потоков**:

* Один ждёт, пока другой что-то сделает
* Несколько потоков должны достигнуть одной точки
* Один поток должен ограничить доступ другим

Эти классы обеспечивают **высокую производительность и безопасность**, в отличие от ручной синхронизации через wait/notify.

**🛠️ 2. Основные синхронизаторы в java.util.concurrent**

| **Класс** | **Назначение** |
| --- | --- |
| CountDownLatch | Потоки ждут, пока не завершится n действий |
| CyclicBarrier | Потоки ждут друг друга на «барьере» |
| Semaphore | Ограничивает количество одновременных потоков |
| Exchanger | Обмен данными между двумя потоками |
| Phaser | Расширенная версия CyclicBarrier с фазами |

**📌 3. Подробное описание и примеры**

**🔻 CountDownLatch**

**Смысл:** Поток ждёт, пока другие потоки вызовут countDown() N раз.

java

КопироватьРедактировать

import java.util.concurrent.CountDownLatch;

public class App {

public static void main(String[] args) throws InterruptedException {

CountDownLatch latch = new CountDownLatch(3); // ждём 3 сигнала

for (int i = 0; i < 3; i++) {

new Thread(() -> {

System.out.println("Поток завершён");

latch.countDown(); // уменьшаем счётчик

}).start();

}

latch.await(); // ждём, пока счётчик станет 0

System.out.println("Все потоки завершились. Продолжаем работу.");

}

}

🟢 Подходит для: старта работы только после подготовки ресурсов, ожидания завершения задач и т.д.

**🔁 CyclicBarrier**

**Смысл:** Все потоки должны достигнуть «барьера» (синхронизируются), прежде чем продолжить.

java

КопироватьРедактировать

import java.util.concurrent.CyclicBarrier;

public class App {

public static void main(String[] args) {

CyclicBarrier barrier = new CyclicBarrier(3, () -> {

System.out.println("Все потоки достигли барьера!");

});

Runnable task = () -> {

System.out.println(Thread.currentThread().getName() + " ждёт");

try {

barrier.await(); // ждём других

System.out.println(Thread.currentThread().getName() + " продолжает работу");

} catch (Exception e) {}

};

new Thread(task).start();

new Thread(task).start();

new Thread(task).start();

}

}

🟢 Подходит для: параллельной обработки данных, фазовых шагов, где все потоки должны быть синхронизированы.

**🔢 Semaphore**

**Смысл:** Ограничивает количество одновременных потоков.

java

КопироватьРедактировать

import java.util.concurrent.Semaphore;

public class App {

public static void main(String[] args) {

Semaphore semaphore = new Semaphore(2); // максимум 2 потока одновременно

Runnable task = () -> {

try {

semaphore.acquire();

System.out.println(Thread.currentThread().getName() + " зашёл");

Thread.sleep(1000); // делает что-то

System.out.println(Thread.currentThread().getName() + " вышел");

} catch (InterruptedException e) {}

finally {

semaphore.release();

}

};

for (int i = 0; i < 5; i++)

new Thread(task).start();

}

}

🟢 Подходит для: ограничения доступа к ресурсам, например к БД, файлам, API.

**🔄 Exchanger<T>**

**Смысл:** Два потока обмениваются объектами.

java

КопироватьРедактировать

import java.util.concurrent.Exchanger;

public class App {

public static void main(String[] args) {

Exchanger<String> exchanger = new Exchanger<>();

new Thread(() -> {

try {

String response = exchanger.exchange("Данные из потока 1");

System.out.println("Поток 1 получил: " + response);

} catch (InterruptedException e) {}

}).start();

new Thread(() -> {

try {

String response = exchanger.exchange("Ответ из потока 2");

System.out.println("Поток 2 получил: " + response);

} catch (InterruptedException e) {}

}).start();

}

}

🟢 Подходит для: **peer-to-peer** обменов, например: передача буфера между потоками.

**🔀 Phaser**

**Смысл:** Как CyclicBarrier, но с возможностью **динамического добавления и удаления потоков** и **нескольких фаз**.

java

КопироватьРедактировать

import java.util.concurrent.Phaser;

public class App {

public static void main(String[] args) {

Phaser phaser = new Phaser(3); // 3 участника

Runnable task = () -> {

System.out.println(Thread.currentThread().getName() + " пришёл на фазу 1");

phaser.arriveAndAwaitAdvance(); // ждём всех

System.out.println(Thread.currentThread().getName() + " перешёл к фазе 2");

phaser.arriveAndAwaitAdvance(); // и снова ждём

};

for (int i = 0; i < 3; i++)

new Thread(task).start();

}

}

🟢 Подходит для: циклов обработки, когда одни потоки отваливаются/добавляются на разных шагах.

**🧾 Таблица-сравнение**

| **Класс** | **Для чего** | **Повторное использование** | **Гибкость** |
| --- | --- | --- | --- |
| CountDownLatch | Подождать выполнения N задач | ❌ Нет | Прост |
| CyclicBarrier | Синхронизировать все потоки | ✅ Да | Средняя |
| Semaphore | Ограничить количество потоков | ✅ Да | Гибкий |
| Exchanger | Обменяться данными | ✅ Да | Узкий |
| Phaser | Синхронизация с фазами | ✅ Да | Очень гибкий |

**🧠 Вывод**

Пакет java.util.concurrent содержит **мощные и гибкие инструменты** для многопоточности, которые:

* Лучше масштабируются, чем synchronized
* Просты в использовании
* Подходят для разных задач: от ожидания завершения до обмена данными

1. **Модификатор volatile. Атомарные типы данных и операции**

**📌 1. Модификатор volatile**

**🔹 Что это такое?**

volatile — это модификатор переменной, который говорит JVM:

«Не кешируй это поле локально потоком. Всегда читай/пиши в **общую память (main memory)**.»

**🔸 Проблема без volatile:**

В многопоточности **потоки могут видеть устаревшие значения переменных**. Это происходит из-за:

* CPU-кэшей
* оптимизаций компилятора и JVM

**Пример:**

java

КопироватьРедактировать

class FlagTest {

private static boolean running = true;

public static void main(String[] args) throws InterruptedException {

Thread t = new Thread(() -> {

while (running) {} // может бесконечно крутиться!

});

t.start();

Thread.sleep(1000);

running = false; // поток t может этого "не увидеть"

}

}

**✅ Решение:**

java

КопироватьРедактировать

private static volatile boolean running = true;

Теперь running = false **гарантированно будет замечено** другим потоком.

**🔐 Что делает volatile:**

1. **Гарантирует видимость** изменений между потоками.
2. **Не даёт переставлять инструкции** (reordering) вокруг чтения/записи.
3. **Не делает переменные атомарными**, если они составные (например: count++).

**❗ Важно:**

java

КопироватьРедактировать

volatile int counter;

counter++; // не атомарно!

Это = read → increment → write, а между ними другой поток может вмешаться.

**🧮 2. Атомарные типы (java.util.concurrent.atomic)**

Если volatile не даёт атомарности, тогда что даёт?

**🔹 Атомарные классы — это обёртки, которые делают операции без блокировок, используя CAS (compare-and-swap).**

**📦 Основные классы:**

| **Класс** | **Назначение** |
| --- | --- |
| AtomicInteger | Атомарный int |
| AtomicLong | Атомарный long |
| AtomicBoolean | Атомарный boolean |
| AtomicReference<T> | Атомарная ссылка на объект |
| LongAdder, DoubleAdder | Высокопроизводительные счётчики |

**🔧 Пример AtomicInteger:**

java

КопироватьРедактировать

import java.util.concurrent.atomic.AtomicInteger;

public class Counter {

private final AtomicInteger count = new AtomicInteger(0);

public void increment() {

count.incrementAndGet(); // атомарно

}

public int get() {

return count.get();

}

}

**🔁 Методы AtomicInteger:**

* get() — получить значение
* set(val) — установить значение
* incrementAndGet() — ++ и вернуть
* getAndIncrement() — вернуть и ++
* compareAndSet(expected, update) — CAS-операция

**Пример CAS:**

java

КопироватьРедактировать

AtomicInteger counter = new AtomicInteger(0);

if (counter.compareAndSet(0, 10)) {

System.out.println("Установлено на 10");

} else {

System.out.println("Кто-то уже изменил");

}

**⚙️ Что под капотом?**

Атомарные классы используют **низкоуровневые инструкции процессора**, а именно:

* compare-and-swap (CAS)
* Unsafe API (внутри JVM)
* Без блокировок — → **отсутствие deadlock**

**🚀 LongAdder / DoubleAdder**

Когда у тебя **много потоков** инкрементируют счётчик, AtomicInteger может стать узким местом.

**🔹 Решение: LongAdder**

* Под капотом хранит **несколько "ячейк" счётчиков**, каждая — для своей группы потоков.
* А потом **суммирует** их.

java

КопироватьРедактировать

import java.util.concurrent.atomic.LongAdder;

public class FastCounter {

private final LongAdder adder = new LongAdder();

public void increment() {

adder.increment(); // быстро и без блокировок

}

public long get() {

return adder.sum();

}

}

🟢 Лучше для **высокой конкуренции** (например, 100+ потоков).

**🧾 Таблица сравнения**

| **Средство** | **Видимость** | **Атомарность** | **Применение** |
| --- | --- | --- | --- |
| volatile | ✅ Да | ❌ Нет | Флаги, одиночные значения |
| synchronized | ✅ Да | ✅ Да | Универсальный способ |
| AtomicInteger и др. | ✅ Да | ✅ Да | Лёгкие атомарные счётчики |
| LongAdder | ✅ Да | ✅ Да | Высоконагруженные счётчики |

**📌 Выводы:**

* **volatile** — для видимости и простых флагов.
* **Атомарные типы** — если нужно выполнять **несколько операций как одну (атомарно)** без synchronized.
* **LongAdder** — когда потоков **много и нужна производительность**.

**Атомарная операция** — это **неделимая операция**, которую **нельзя прервать** или **увидеть в "промежуточном состоянии"** из другого потока.

count++ — это **три операции**:

1. Прочитать count из памяти
2. Увеличить значение
3. Записать результат обратно

Если два потока одновременно выполняют count++, может произойти **состояние гонки (race condition)**: оба читают старое значение и оба записывают одинаковый результат.

**✅ Атомарная версия:**

java

КопироватьРедактировать

AtomicInteger count = new AtomicInteger(0);

count.incrementAndGet(); // атомарно

Метод incrementAndGet() — это **одна цельная операция**. Если два потока её вызовут одновременно:

* Один подождёт окончания другого
* Результат будет корректным

**5. Коллекции из пакета java.util.concurrent**

**📦 1. Что такое коллекции java.util.concurrent**

Java из коробки предоставляет обычные коллекции: ArrayList, HashMap, HashSet и т.д. Они **не потокобезопасны**.

java.util.concurrent — это пакет, содержащий **потокобезопасные коллекции**, которые можно **использовать одновременно из нескольких потоков без synchronized снаружи.**

**🧱 2. Основные коллекции из java.util.concurrent**

| **Тип коллекции** | **Интерфейс** | **Примеры реализаций** |
| --- | --- | --- |
| Очереди | Queue / Deque | ConcurrentLinkedQueue, LinkedBlockingQueue, ArrayBlockingQueue, PriorityBlockingQueue, ConcurrentLinkedDeque |
| Карты | Map | ConcurrentHashMap, ConcurrentSkipListMap |
| Наборы | Set | ConcurrentSkipListSet, CopyOnWriteArraySet |
| Списки | List | CopyOnWriteArrayList |
| Блокирующие очереди | BlockingQueue | LinkedBlockingQueue, ArrayBlockingQueue, DelayQueue |

**🧵 3. Коллекции по группам**

**🔁 CopyOnWriteArrayList / CopyOnWriteArraySet**

* При **каждой модификации** (add/remove) создаётся **новая копия массива**
* Отлично подходит, если **чтений больше, чем записей**

java

КопироватьРедактировать

import java.util.concurrent.CopyOnWriteArrayList;

CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>();

list.add("A");

list.add("B");

for (String item : list) {

System.out.println(item); // безопасно в многопоточке

}

✅ Применяется в: кэшах, конфигурации, обработке событий (много читаем, редко изменяем)

**🗃 ConcurrentHashMap**

* Потокобезопасная версия HashMap
* Работает **без полной блокировки**
* Использует **сегментацию** (в Java 7) или **CAS + дерево** (в Java 8+)

java

КопироватьРедактировать

import java.util.concurrent.ConcurrentHashMap;

ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();

map.put("apple", 1);

map.putIfAbsent("apple", 2); // безопасно

map.compute("apple", (k, v) -> v + 1); // атомарное обновление

✅ Применяется в: кэши, счётчики, карты ресурсов, блокировки по ключу

**⏳ BlockingQueue**

* Интерфейс для **очередей с ожиданием**
* Методы:
  + put() — ждёт, если нет места
  + take() — ждёт, если пусто

java

КопироватьРедактировать

import java.util.concurrent.ArrayBlockingQueue;

ArrayBlockingQueue<String> queue = new ArrayBlockingQueue<>(2);

queue.put("A"); // добавит

queue.put("B"); // добавит

// queue.put("C"); // заблокирует, пока не освободится место

String item = queue.take(); // заберёт элемент, если есть

**Разновидности:**

| **Класс** | **Особенности** |
| --- | --- |
| ArrayBlockingQueue | Фиксированный размер, быстрый |
| LinkedBlockingQueue | Динамический размер, связанный список |
| PriorityBlockingQueue | Очередь по приоритету |
| DelayQueue | Элементы появляются по таймеру |

✅ Применяется в: очередях задач, producer-consumer, потокобезопасных буферах

**🧱 ConcurrentLinkedQueue / Deque**

* **Неблокирующая очередь**, основана на **CAS**
* Отлично подходит для систем с высокой пропускной способностью

java

КопироватьРедактировать

import java.util.concurrent.ConcurrentLinkedQueue;

ConcurrentLinkedQueue<String> queue = new ConcurrentLinkedQueue<>();

queue.offer("task1");

System.out.println(queue.poll()); // task1

✅ Применяется в: обработчиках событий, очередях сообщений, lock-free структурах

**🌲 ConcurrentSkipListMap / Set**

* **Потокобезопасная** сортированная коллекция (аналог TreeMap)
* Использует **skip list** (многоуровневая линейная структура)

java

КопироватьРедактировать

import java.util.concurrent.ConcurrentSkipListMap;

ConcurrentSkipListMap<Integer, String> map = new ConcurrentSkipListMap<>();

map.put(2, "B");

map.put(1, "A");

System.out.println(map.firstEntry()); // 1=A

✅ Применяется в: таймлайнах, структуре данных с упорядоченным доступом

**⚙ Как они работают внутри**

| **Механизм** | **Что делает** |
| --- | --- |
| **CAS (Compare-And-Swap)** | Недорогие атомарные операции вместо блокировок |
| **Сегментация** (в старом ConcurrentHashMap) | Делит данные на сегменты, блокирует по частям |
| **Lock-free структуры** | Без synchronized, с помощью CAS |
| **Копирование на запись** (CopyOnWrite) | Новая копия при изменении, старые читаются без блокировки |

**🧾 Таблица — что выбрать?**

| **Цель** | **Коллекция** |
| --- | --- |
| Много потоков читают, мало пишут | CopyOnWriteArrayList, CopyOnWriteArraySet |
| Словарь / кэш / счётчики | ConcurrentHashMap |
| Потокобезопасная очередь | LinkedBlockingQueue, ArrayBlockingQueue |
| Безблокирующая очередь | ConcurrentLinkedQueue |
| Сортированные коллекции | ConcurrentSkipListMap, Set |

**💡 На заметку**

* Не используй обычные ArrayList, HashMap в многопоточке без внешней синхронизации.
* Collections.synchronizedList(...) — это устаревший способ (работает, но медленно и небезопасно при итерации).
* ConcurrentHashMap **безопасен даже при одновременном чтении/записи/удалении**.
* Для итерации по ConcurrentHashMap — **итератор слабоконсистентный**: не кидает ConcurrentModificationException, но может не видеть самые свежие данные.
  1. **Интерфейсы Executor, ExecutorService, Callable, Future**

**🔷 1. Интерфейс Executor**

**🔸 Что это?**

Executor — это **самый базовый интерфейс** для запуска задач **в потоке**, не управляя потоками напрямую.

**📘 Сигнатура:**

java

КопироватьРедактировать

public interface Executor {

void execute(Runnable command);

}

**Пример:**

java

КопироватьРедактировать

import java.util.concurrent.Executor;

import java.util.concurrent.Executors;

public class App {

public static void main(String[] args) {

Executor executor = Executors.newSingleThreadExecutor();

executor.execute(() -> System.out.println("Выполняется задача"));

}

}

✅ Применяется как **универсальный интерфейс**, на основе которого построены все остальные.

**🔷 2. Интерфейс ExecutorService**

**🔸 Расширение Executor с дополнительными методами:**

* submit() — запускает задачу и возвращает Future
* shutdown() — завершает приём задач
* invokeAll(), invokeAny() — запускает несколько задач

java

КопироватьРедактировать

ExecutorService service = Executors.newFixedThreadPool(4);

**📌 Пример использования:**

java

КопироватьРедактировать

import java.util.concurrent.\*;

public class App {

public static void main(String[] args) throws InterruptedException {

ExecutorService executor = Executors.newFixedThreadPool(2);

Runnable task = () -> System.out.println(Thread.currentThread().getName() + ": Задача выполнена");

executor.execute(task);

executor.execute(task);

executor.shutdown(); // останавливаем приём новых задач

executor.awaitTermination(1, TimeUnit.SECONDS);

}

}

🔥 Под капотом ExecutorService использует **пул потоков** (ThreadPoolExecutor).

**🔷 3. Интерфейс Callable<V>**

**🔸 Аналог Runnable, но с результатом и исключением**

java

КопироватьРедактировать

@FunctionalInterface

public interface Callable<V> {

V call() throws Exception;

}

**Пример:**

java

КопироватьРедактировать

Callable<String> task = () -> {

Thread.sleep(500);

return "Готово!";

};

✅ Используется с submit() в ExecutorService, возвращает Future<V>

**🔷 4. Интерфейс Future<V>**

**🔸 Представляет результат асинхронной задачи.**

Методы:

| **Метод** | **Назначение** |
| --- | --- |
| get() | Блокирует до завершения и возвращает результат |
| get(timeout) | Ждёт заданное время, потом исключение |
| cancel() | Отменяет задачу |
| isDone() | Проверяет, завершена ли задача |
| isCancelled() | Была ли отменена |

**📘 Полный пример: Callable + Future**

java

КопироватьРедактировать

import java.util.concurrent.\*;

public class App {

public static void main(String[] args) throws Exception {

ExecutorService executor = Executors.newSingleThreadExecutor();

Callable<Integer> task = () -> {

Thread.sleep(1000);

return 42;

};

Future<Integer> future = executor.submit(task);

System.out.println("Ожидаем результат...");

Integer result = future.get(); // блокируется

System.out.println("Результат: " + result);

executor.shutdown();

}

}

**🔁 Разница: Runnable vs Callable**

| **Свойство** | **Runnable** | **Callable<V>** |
| --- | --- | --- |
| Возвращает значение | ❌ Нет | ✅ Да (call() → V) |
| Может выбрасывать исключения | ❌ Нет напрямую | ✅ Да (throws Exception) |
| Метод | void run() | V call() |

**🚀 Распространённые реализации ExecutorService**

| **Метод** | **Описание** |
| --- | --- |
| Executors.newFixedThreadPool(n) | Пул из n потоков |
| Executors.newSingleThreadExecutor() | Один поток |
| Executors.newCachedThreadPool() | Неограниченный (растущий) пул |
| Executors.newScheduledThreadPool(n) | Поддержка schedule(), delay, periodic |

**🧠 Почему это круто?**

* ✅ **Не нужно вручную управлять потоками**
* ✅ Легко **управлять пулом**, shutdown'ом
* ✅ Можно **получать результат** выполнения (через Future)
* ✅ Гибко масштабируется

**🧾 Резюме**

| **Интерфейс** | **Назначение** |
| --- | --- |
| Executor | Базовый интерфейс для запуска Runnable |
| ExecutorService | Расширенный: управление задачами, shutdown, submit() |
| Callable<V> | Задача с результатом |
| Future<V> | Объект, представляющий результат задачи |

**⚠ Важно помнить**

* shutdown() останавливает **приём новых задач**, но не завершает уже запущенные
* Используй future.get(timeout) чтобы **не зависнуть навсегда**
* Всегда вызывай shutdown() или shutdownNow() во избежание зависания JVM
  1. **Пулы потоков**

**🔷 Что такое пул потоков?**

**Пул потоков** — это заранее созданный набор потоков, который **переиспользуется для выполнения задач**, вместо создания нового потока каждый раз.

**🔸 Почему это важно:**

* Создание потока = дорого (по ресурсам и времени)
* Много потоков → тормоза, контекстные переключения
* Повторное использование потоков = **высокая производительность**

**📦 Где это реализовано?**

В Java — через **ExecutorService**, особенно через **ThreadPoolExecutor**.

Создаётся через Executors:

java

КопироватьРедактировать

ExecutorService pool = Executors.newFixedThreadPool(4);

**🧱 Виды пулов потоков**

| **Метод Executors** | **Что создаёт** | **Особенности** |
| --- | --- | --- |
| newFixedThreadPool(n) | Пул с фиксированным числом потоков | Потоки не растут, задачи в очереди |
| newCachedThreadPool() | Гибкий пул, растёт по мере надобности | Хорошо для краткоживущих задач |
| newSingleThreadExecutor() | Один поток | Задачи выполняются строго по порядку |
| newScheduledThreadPool(n) | Планировщик задач | Задачи с задержкой или периодом |

**🧠 Как работает пул потоков?**

1. Ты **отправляешь задачу** (Runnable или Callable) через submit() / execute()
2. Задача попадает в **очередь**
3. Поток из пула **берёт задачу и выполняет**
4. После завершения — **поток не умирает**, а ждёт следующую задачу

**⚙ Под капотом: ThreadPoolExecutor**

Это **реальная реализация** пула потоков:

java

КопироватьРедактировать

ThreadPoolExecutor executor = new ThreadPoolExecutor(

corePoolSize, // минимальное количество потоков

maximumPoolSize, // максимум

keepAliveTime, // время «жизни» лишних потоков

TimeUnit.SECONDS,

new LinkedBlockingQueue<>(), // очередь задач

new ThreadPoolExecutor.AbortPolicy() // обработка переполнения

);

Именно его используют все методы Executors.\*().

**📘 Пример: фиксированный пул**

java

КопироватьРедактировать

import java.util.concurrent.\*;

public class App {

public static void main(String[] args) {

ExecutorService pool = Executors.newFixedThreadPool(3);

for (int i = 1; i <= 5; i++) {

final int taskNum = i;

pool.execute(() -> {

System.out.println("Задача " + taskNum + " выполняется потоком " + Thread.currentThread().getName());

try {

Thread.sleep(1000);

} catch (InterruptedException e) {}

});

}

pool.shutdown(); // завершение после выполнения всех задач

}

}

**🧾 Важные методы ExecutorService**

| **Метод** | **Назначение** |
| --- | --- |
| execute(Runnable) | Запустить задачу (без Future) |
| submit(Callable) | Запустить и получить Future |
| shutdown() | Остановить приём задач, ждать завершения |
| shutdownNow() | Прервать всё немедленно |
| awaitTermination() | Ждать завершения заданное время |

**🧪 Пример: submit() + Future**

java

КопироватьРедактировать

ExecutorService pool = Executors.newFixedThreadPool(2);

Future<Integer> future = pool.submit(() -> {

Thread.sleep(500);

return 123;

});

Integer result = future.get(); // блокирует до завершения

System.out.println(result);

**🚦 Что будет, если задач больше, чем потоков?**

* Если потоков 2, а задач 5 → 2 начнут выполняться, 3 — **будут ждать в очереди**
* Если очередь заполнена и нельзя создать больше потоков → **RejectedExecutionException**

**⚠ Проблемы, которые могут возникнуть**

| **Проблема** | **Причина** | **Решение** |
| --- | --- | --- |
| OutOfMemoryError | Слишком много задач, очередь без лимита | Использовать ограниченную очередь |
| RejectedExecutionException | Задача не может быть запущена | Использовать политику отбраков |
| Потоки не завершаются | Не вызван shutdown() | Всегда закрывай пул! |

**🛠 Политики обработки отказов (RejectedExecutionHandler)**

Если пул переполнен, ты можешь задать поведение:

| **Политика** | **Что делает** |
| --- | --- |
| AbortPolicy (по умолчанию) | Бросает исключение |
| CallerRunsPolicy | Выполняет задачу в вызывающем потоке |
| DiscardPolicy | Просто отбрасывает |
| DiscardOldestPolicy | Удаляет самую старую задачу и добавляет новую |

**🧠 Когда и что использовать?**

| **Цель** | **Что использовать** |
| --- | --- |
| Фиксированное число задач | newFixedThreadPool(n) |
| Очень много коротких задач | newCachedThreadPool() |
| Один поток (логика, последовательность) | newSingleThreadExecutor() |
| Задачи по таймеру | newScheduledThreadPool(n) |

**📌 Вывод:**

* Пул потоков = **управляемое, повторно используемое множество потоков**
* Работает через ExecutorService / ThreadPoolExecutor
* Повышает **производительность**, снижает **нагрузку**
* Использовать **shutdown() обязательно**, иначе — зависание JVM
  1. **JDBC. Порядок взаимодействия с базой данных. Класс DriverManager. Интерфейс Connection**

**📌 Что такое JDBC?**

**JDBC** — это **API в Java для взаимодействия с реляционными базами данных**, входящий в стандартную библиотеку (java.sql).

JDBC позволяет:

* подключаться к БД,
* выполнять SQL-запросы,
* получать и обрабатывать результаты.

**🔷 Общий порядок взаимодействия с БД через JDBC**

plaintext

КопироватьРедактировать

1. Загрузить драйвер (иногда не требуется в современных JDBC)

2. Получить соединение через DriverManager

3. Создать объект Statement или PreparedStatement

4. Выполнить SQL-запрос

5. Обработать результат (если он есть)

6. Закрыть ресурсы (ResultSet, Statement, Connection)

**🧩 1. Класс DriverManager**

**📘 Что это?**

DriverManager управляет **установкой соединений с БД**, находя подходящий **драйвер**.

**🔹 Метод:**

java

КопироватьРедактировать

Connection conn = DriverManager.getConnection(url, user, password);

🔑 Где:

* url — строка подключения (jdbc:mysql://localhost:3306/mydb)
* user / password — логин и пароль от БД

**📦 Пример:**

java

КопироватьРедактировать

String url = "jdbc:mysql://localhost:3306/shop";

String user = "root";

String password = "1234";

Connection conn = DriverManager.getConnection(url, user, password);

Важно: нужный драйвер (например, mysql-connector-j) должен быть добавлен в classpath (обычно через Maven или Gradle).

**🔷 2. Интерфейс Connection**

**📘 Что это?**

Объект Connection представляет **открытое соединение с базой данных**.

С его помощью можно:

* создавать запросы (SQL),
* управлять транзакциями,
* закрывать соединение.

**🔹 Примеры основных методов:**

| **Метод** | **Описание** |
| --- | --- |
| createStatement() | Создаёт простой SQL-запрос |
| prepareStatement(sql) | Создаёт параметризованный запрос |
| setAutoCommit(false) | Отключить автокоммит (для транзакций) |
| commit() / rollback() | Завершение / откат транзакции |
| close() | Закрывает соединение |

**🔧 Пример полного подключения и запроса**

java

КопироватьРедактировать

import java.sql.\*;

public class JdbcExample {

public static void main(String[] args) {

String url = "jdbc:mysql://localhost:3306/shop";

String user = "root";

String password = "1234";

try (Connection conn = DriverManager.getConnection(url, user, password);

Statement stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery("SELECT \* FROM products")) {

while (rs.next()) {

int id = rs.getInt("id");

String name = rs.getString("name");

double price = rs.getDouble("price");

System.out.println(id + ": " + name + " → $" + price);

}

} catch (SQLException e) {

e.printStackTrace();

}

}

}

⚠ Важно: используем **try-with-resources** — соединение, Statement и ResultSet будут **автоматически закрыты**.

**🧪 Немного подробнее про SQL-инструкции**

**Statement**

java

КопироватьРедактировать

Statement stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery("SELECT \* FROM table");

Используется для **простых запросов** (без параметров)

**PreparedStatement**

java

КопироватьРедактировать

PreparedStatement ps = conn.prepareStatement("SELECT \* FROM users WHERE id = ?");

ps.setInt(1, 5);

ResultSet rs = ps.executeQuery();

Используется для:

* **параметризации запросов**
* **предотвращения SQL-инъекций**
* **повторного использования**

**🔐 Транзакции через Connection**

java

КопироватьРедактировать

conn.setAutoCommit(false); // вручную управляем транзакцией

try {

// ... операции insert/update/delete

conn.commit();

} catch (SQLException e) {

conn.rollback(); // откат в случае ошибки

}

**📦 Как выглядит JDBC-URL?**

| **БД** | **Пример JDBC URL** |
| --- | --- |
| MySQL | jdbc:mysql://localhost:3306/mydb |
| PostgreSQL | jdbc:postgresql://localhost:5432/mydb |
| SQLite | jdbc:sqlite:myfile.db |
| Oracle | jdbc:oracle:thin:@localhost:1521:xe |

**⚠ Важные замечания:**

* JDBC **не проверяет доступность базы** сам — ошибки надо ловить (SQLException)
* После выполнения запросов **обязательно закрывать соединение**
* Работа с ResultSet возможна **только пока открыты Statement и Connection**

**🧾 Резюме**

| **Компонент** | **Назначение** |
| --- | --- |
| DriverManager | Управляет подключениями к БД |
| Connection | Представляет соединение, управляет транзакциями |
| Statement | Выполняет простой SQL-запрос |
| PreparedStatement | Выполняет параметризованные запросы |
| ResultSet | Таблица результатов SQL-запроса |

* 1. **Интерфейсы Statement, PreparedStatement, ResultSet, RowSet**

**🧩 1. Интерфейс Statement**

**📘 Назначение:**

Выполнение **простых SQL-запросов без параметров** (чаще всего — SELECT, UPDATE, DELETE, INSERT).

**🔧 Пример:**

java

КопироватьРедактировать

Statement stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery("SELECT \* FROM products");

**🔹 Методы:**

| **Метод** | **Назначение** |
| --- | --- |
| executeQuery(sql) | Выполняет SELECT, возвращает ResultSet |
| executeUpdate(sql) | Выполняет INSERT, UPDATE, DELETE, возвращает кол-во строк |
| execute(sql) | Универсальный: возвращает true или false в зависимости от результата |
| close() | Закрывает statement |

**⚠ Недостатки Statement:**

* Не поддерживает параметры (?)
* Подвержен **SQL-инъекциям**:

java

КопироватьРедактировать

String query = "SELECT \* FROM users WHERE login = '" + userInput + "'";

**🧩 2. Интерфейс PreparedStatement**

**📘 Назначение:**

То же, что и Statement, но с поддержкой **параметров (?)**. Предотвращает **SQL-инъекции** и повышает производительность при повторных запросах.

**🔧 Пример:**

java

КопироватьРедактировать

PreparedStatement ps = conn.prepareStatement("SELECT \* FROM users WHERE id = ?");

ps.setInt(1, 5); // подставляем параметр

ResultSet rs = ps.executeQuery();

**🔹 Методы:**

| **Метод** | **Назначение** |
| --- | --- |
| setXxx(index, value) | Задание параметров (setInt, setString, ...) |
| executeQuery() | SELECT |
| executeUpdate() | INSERT, UPDATE, DELETE |
| execute() | Универсальный |

**✅ Преимущества:**

* Защита от **SQL-инъекций**
* Повторное использование с разными параметрами
* Компилируется один раз и может выполняться много раз (внутри драйвера)

**🧩 3. Интерфейс ResultSet**

**📘 Назначение:**

Представляет **таблицу результатов SQL-запроса**. Используется для чтения данных **построчно**.

**🔧 Пример:**

java

КопироватьРедактировать

ResultSet rs = stmt.executeQuery("SELECT id, name FROM products");

while (rs.next()) {

int id = rs.getInt("id");

String name = rs.getString("name");

System.out.println(id + " → " + name);

}

**🔹 Методы:**

| **Метод** | **Назначение** |
| --- | --- |
| next() | Переход к следующей строке |
| getXxx("column") | Получение значения (getInt, getString, ...) |
| wasNull() | Проверка: было ли значение NULL |
| close() | Закрывает ResultSet |

**🔸 Дополнительно:**

* Можно перемещаться по результату (rs.previous(), rs.absolute(n)), если создан с нужным ResultSet.TYPE\_SCROLL\_INSENSITIVE.
* Полезно проверять: rs.next() перед чтением.

**🧩 4. Интерфейс RowSet**

**📘 Назначение:**

**Объектный интерфейс над ResultSet**, с расширенной функциональностью: **дисконнект**, сериализация, биндинги с GUI и т.д.

Он наследует ResultSet, но:

* Можно работать без открытого Connection
* Поддерживает событийную модель (наблюдатели)
* Используется в GUI (например, с Swing)

**🔧 Основные реализации:**

| **Класс** | **Особенность** |
| --- | --- |
| JdbcRowSet | Онлайн, над ResultSet |
| CachedRowSet | **Оффлайн**, можно передавать между потоками |
| FilteredRowSet | Фильтрация данных по условию |
| WebRowSet | XML-совместимый RowSet |
| JoinRowSet | Позволяет делать JOIN внутри Java |

**📘 Пример CachedRowSet:**

java

КопироватьРедактировать

import javax.sql.rowset.CachedRowSet;

import com.sun.rowset.CachedRowSetImpl;

CachedRowSet crs = new CachedRowSetImpl();

crs.setUrl("jdbc:mysql://localhost:3306/shop");

crs.setUsername("root");

crs.setPassword("1234");

crs.setCommand("SELECT \* FROM products");

crs.execute(); // выполняется и сохраняется локально

while (crs.next()) {

System.out.println(crs.getString("name"));

}

⚠ Требуется зависимость: com.sun.rowset.CachedRowSetImpl (в некоторых JDK'ах надо подключать отдельно)

**🧾 Сравнение интерфейсов:**

| **Интерфейс** | **Назначение** | **Поддержка параметров** | **Онлайн/Оффлайн** | **Примеры** |
| --- | --- | --- | --- | --- |
| Statement | Простые SQL-запросы | ❌ Нет | Онлайн | SELECT \* FROM table |
| PreparedStatement | Параметризированные запросы | ✅ Да | Онлайн | WHERE id = ? |
| ResultSet | Представление результата | - | Онлайн | Чтение результатов |
| RowSet | Расширенный ResultSet, событийная модель | ✅ Да | Онлайн или оффлайн | CachedRowSet, JdbcRowSet |

**✅ Когда что использовать?**

| **Задача** | **Лучшее средство** |
| --- | --- |
| Один раз выполнить простой запрос | Statement |
| Выполнить запрос с параметрами | PreparedStatement |
| Безопасно обработать результат | ResultSet |
| Хранить результат оффлайн | CachedRowSet |
| Использовать в GUI | RowSet |

* 1. **Шаблоны проектирования**

**Доп информация**

**SQL-инъекция (SQL Injection)** — это уязвимость, при которой **злоумышленник подставляет SQL-код в пользовательский ввод**, чтобы изменить поведение SQL-запроса