лаба 7 - многопоточность и бд

Многопоточность. Класс Thread, интерфейс Runnable. Модификатор synchronized

Многопоточность в Java — это одновременное выполнение двух или более потоков для максимального использования центрального процессора (CPU — central processing unit). Каждый поток работает параллельно и не требует отдельной области памяти. К тому же, переключение контекста между потоками занимает меньше времени.

Процесс — это экземпляр выполняющейся программы. Каждый процесс обладает своим собственным адресным пространством и ресурсами, такими как файлы и сетевые соединения.

Поток (или поток выполнения) — более легковесная единица выполнения, существующая внутри процесса. В одном процессе может существовать несколько потоков, которые могут параллельно выполнять инструкции программы. В отличие от процессов, потоки внутри одного процесса совместно используют его ресурсы.

Основные причины применения многопоточности

1. Повышение производительности. Разделение задач на потоки позволяет эффективнее использовать ресурсы процессора и ускоряет выполнение вычислительных задач.

2. Отзывчивость приложения. Задачи, требующие времени, такие как загрузка данных или выполнение сложных вычислений, могут быть вынесены в отдельные потоки, что не блокирует основной поток и обеспечивает отзывчивость приложения.

3. Улучшенный пользовательский опыт. Параллельное выполнение задач позволяет создавать более интерактивные пользовательские интерфейсы.

Thread

- Наследуется от **Object**, реализует **Runnable**.
- **Методы**
  - **start()** — запускает поток (вызывает **run()** асинхронно).
  - **run()** — содержит код для выполнения (переопределяется).
  - **sleep(long millis)** — приостанавливает поток на указанное время.
  - **interrupt()** — прерывает поток.

- **join()** — ожидает завершения потока.
- **getPriority()**: возвращает приоритет потока
- **setPriority(int proirity)**: устанавливает приоритет потока. Приоритет является одним из ключевых факторов для выбора системой потока из кучи потоков для выполнения. В этот метод в качестве параметра передается числовое значение приоритета - от 1 до 10. По умолчанию главному потоку выставляется средний приоритет - 5.
- **isAlive()**: возвращает true, если поток активен
- **isInterrupted()**: возвращает true, если поток был прерван

```
class MyThread extends Thread {
   @Override
   public void run() {
      System.out.println("Поток выполняется"); //код задачи(создаётся экземпляр
                                               //анонимного класса
   }
}
MyThread thread = new MyThread();
thread.start(); //запуск потока
```

Runnable

- Функциональный интерфейс с одним методом `run()`

```
public class MyRunnable implements Runnable {
   public void run() {
      System.out.println("Поток выполняется"); //код задачи
   }
}

new Thread(new MyRunnable()).start(); //запуск потока
```

Наследование класса **Thread** целесообразно применять, когда нужно **дополнить функциональность** самого класса Thread**.**

Использование интерфейса **Runnable** – когда просто нужно одновременно выполнить несколько задач и не требуется вносить изменений в сам механизм многопоточности.

Сравнение 2 подходов:

1. **Наследование:** В Java класс может наследовать только один класс. Следовательно, если класс уже наследует другой класс, он не сможет расширить класс Thread. В этом

случае единственным вариантом будет реализация интерфейса Runnable.

2. **Композиция:** Использование интерфейса Runnable обеспечивает большую гибкость, поскольку позволяет взаимодействовать с несколькими интерфейсами. Кроме того, **интерфейс Runnable можно использовать для создания объекта задачи, который затем можно передать в конструктор Thread(это показано в примере)**

В каком порядке запускать новые потоки решает **Планировщик потоков** – часть **JVM**, которая решает какой поток должен выполнится в каждый конкретный момент времени и какой поток нужно приостановить.

Использование многопоточности может привести к двум ситуациям:

Deadlock (взаимная блокировка) – несколько потоков находятся в состоянии ожидания ресурсов, занятых друг другом, и ни один из них не может продолжать выполнение.

Race Condition (состояние гонки) – ошибка проектирования многопоточной системы или приложения, при которой работа системы или приложения зависит от того, в каком порядке выполняются части кода.

Не все **Race condition** потенциально производят **Deadlock**,
однако, **Deadlock** происходят **только** в **Race condition**.

Последовательность выполнения потоков контролировать **нельзя.**

Если поток был запущен и завершился – **повторно запустить** его **не получится**.

Жизненный цикл потока:

- **New**

Поток находится в состоянии **New**, когда создается новый экземпляр объекта класса **Thread**, но метод `start()` не вызывался.

- **Runnable**

Когда для созданного нового объекта **Thread** был вызван метод `start()`.

Такой поток либо **ожидает**, что **планировщик** заберет его для выполнения, либо уже **запущен**.

- **Non-Runnable** (**Blocked** , **Timed-Waiting**)

Когда поток временно неактивен, то есть объект класса **Thread** существует, но **не выбран планировщиком** для выполнения.

- **Terminated**

Когда поток завершает выполнение своего метода `run()`, он переходит в состояние terminated (завершен). На этом этапе выполнение потока **завершается**.

synchronized

Синхронизация — процесс управления одновременным выполнением нескольких потоков с целью предотвращения конфликтов при доступе к общим ресурсам.

Монитор — механизм синхронизации, связанный с каждым объектом. Он обеспечивает взаимное исключение: в мониторе объекта может находиться только один поток в любой момент времени. Мониторы предотвращают состояния гонки и обеспечивают безопасное взаимодействие между потоками.

> 💡 Проблемы с использованием общих ресурсов решаются синхронизацией потоков (блокировкой ресурсов).

Механизм синхронизации обеспечивает **последовательный доступ** к ресурсам: выполнение программы приостанавливается до освобождения блокировки.

Ключевое слово synchronized используется для блокировки ресурса.

Синхронизированным может быть

- весь **метод**
- отдельный **блок кода** внутри метода.

Примеры:

```
//Синхронизированный метод экземпляра (блокировка на объект)
public class Test {
   public synchronized void test() {
      //код
   }
}
```

```
//Синхронизированный статический метод (блокировка на классе)
public class Test {
   public static synchronized void testStatic() {
      //код
   }
}
```

Важно:

- static метод – в этом случае синхронизация будет осуществляться **по классу** где этот метод объявлен.

- Если у объекта один метод синхронизирован как static, а другой как обычный, они **могут выполняться одновременно**, так как монитор для static метода — это **класс**, а для обычного метода — **экземпляр объекта**.

Про final поля:

- Поля final инициализируются в **конструкторе**.

  Их значения гарантированно видны всем потокам **без необходимости синхронизации**.

Недостаток synchronized:

- Потоки вынуждены ждать освобождения блокировки, что может создать **узкое место (bottleneck)** и ухудшить производительность программы.

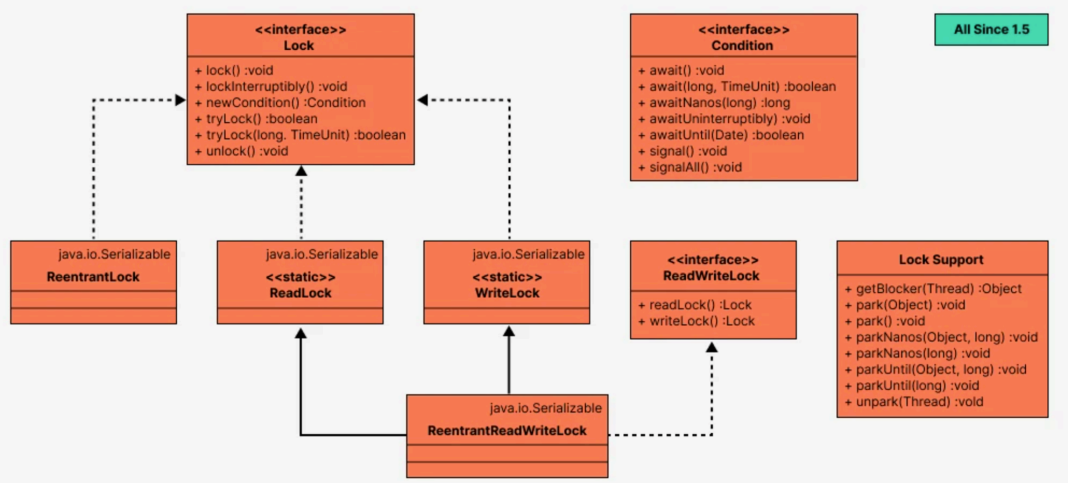
Методы wait(), notify() класса Object, интерфейсы Lock и Condition

> 💡 Методы **wait(), notify(), и notifyAll()** — это инструменты для координации работы между потоками, которые используют общие ресурсы. Они вызываются на объекте класса, который реализует интерфейс Object, и должны использоваться только в синхронизированных блоках или методах.

- wait() заставляет текущий поток ожидать до тех пор, пока другой поток не вызовет notify() или notifyAll() на том же объекте.
- notify() пробуждает один случайно выбранный поток, который ожидает на этом объекте.
- notifyAll() пробуждает все потоки, которые ожидают на этом объекте.

```
class ConditionLoop {
 private boolean condition;
 synchronized void waitForCondition() throws InterruptedException {
  while (!condition) {
   wait();
  }
 }
 synchronized void satisfyCondition() {
  condition = true;
  notifyAll();
 }
}
```

Lock



Пакет java.util.concurrent.locks имеет стандартный интерфейс **Lock**.

Различия между блокировкой и синхронизированным блоком

Есть несколько различий между использованием синхронизированного блока и использованием API блокировки :

- **Синхронизированный блок полностью содержится в методе —** мы можем иметь **операции** lock() и unlock() API Lock в отдельных методах .
- Синхронизированный блок не поддерживает справедливость, любой поток может получить блокировку после освобождения, предпочтения не могут быть указаны. **Мы можем добиться справедливости в API-интерфейсах блокировки , указав свойство справедливости** . Это гарантирует, что самому длинному ожидающему потоку будет предоставлен доступ к блокировке.
- Поток блокируется, если он не может получить доступ к синхронизированному блоку . **API блокировки предоставляет метод tryLock() . Поток получает блокировку только в том случае, если он доступен и не удерживается каким-либо другим потоком.** Это уменьшает время блокировки потока, ожидающего блокировки.
- Поток, который находится в состоянии «ожидания» получения доступа к синхронизированному блоку , не может быть прерван. **API блокировки предоставляет метод lockInterruptably() , который можно использовать для прерывания потока, когда он ожидает блокировки.**

Основные методы

- **void lock()** — получить блокировку, если она доступна; если блокировка недоступна, поток блокируется до тех пор, пока блокировка не будет снята

- **void lockInterruptably()** — аналогичен lock(), но позволяет прервать заблокированный поток и возобновить выполнение с помощью выброшенного исключения java.lang.InterruptedException .
- **boolean tryLock()** – это неблокирующая версия метода lock(), он пытается немедленно получить блокировку, возвращает true, если блокировка успешна
- **boolean tryLock(long timeout, TimeUnit timeUnit)** — это похоже на tryLock(), за исключением того, что он ждет заданный тайм-аут, прежде чем отказаться от попытки получить блокировку
- **void unlock()** — разблокирует экземпляр Lock

Заблокированный экземпляр всегда должен быть разблокирован, чтобы избежать взаимоблокировки. Рекомендуемый блок кода для использования блокировки должен содержать блоки try/catch и finally :

```
Lock lock = ...;
lock.lock();
try {
//доступ к общему ресурсу
} finally {
    lock.unlock();
}
```

В дополнение к интерфейсу `Lock` у нас есть интерфейс `ReadWriteLock` , который поддерживает пару блокировок: одну для операций только для чтения и одну для операции записи. Блокировка чтения может одновременно удерживаться несколькими потоками, пока нет записи.

ReadWriteLock объявляет методы для получения блокировки чтения или записи:

- **Lock readLock()** — возвращает блокировку, которая использовалась для чтения
- **Lock writeLock()** — возвращает блокировку, которая использовалась для записи.

ReentantLock

Класс ReentrantLock реализует интерфейс блокировки . Он предлагает ту же семантику параллелизма и памяти, что и неявная блокировка монитора, доступ к которой осуществляется с помощью синхронизированных методов и операторов, с расширенными возможностями.

Вот как можно использовать ReentrantLock для синхронизации:

```
public class SharedObject {
    //...
```

```
    ReentrantLock lock = new ReentrantLock();
    int counter = 0;

    public void perform() {
        lock.lock();
        try {
            //критическая секция
            count++;
        } finally {
            lock.unlock();
        }
    }
    //...
}

public Stack<Vehicle> removeById(User user, long id) throws DBProviderException {
        locker.lock();
        try {
            if (DBProvider.removeVehicleById(id)){
                collection.removeIf(vehicle -> vehicle.getId() == id);
                return new Stack<>();
            }
            throw new DBProviderException("Произошла ошибка при удалении элемента. Возмо
        } finally {
            locker.unlock();
        }
    }
```

ReentrantReadWriteLock

Класс ReentrantReadWriteLock реализует интерфейс ReadWriteLock .

- **Блокировка чтения** — если ни один поток не получил блокировку записи или не запросил ее, тогда несколько потоков могут получить блокировку чтения.
- **Блокировка записи** — если ни один поток не читает и не пишет, только один поток может получить блокировку записи.

```
public class SynchronizedHashMapWithReadWriteLock {
    Map<String,String> syncHashMap = new HashMap<>();
    ReadWriteLock lock = new ReentrantReadWriteLock();
    // ...
    Lock writeLock = lock.writeLock();
```

```
    public void put(String key, String value) {
        try {
            writeLock.lock();
            syncHashMap.put(key, value);
        } finally {
            writeLock.unlock();
        }
    }
    ...
    public String remove(String key){
        try {
            writeLock.lock();
            return syncHashMap.remove(key);
        } finally {
            writeLock.unlock();
        }
    }
    //...
}

Lock readLock = lock.readLock();
//...
public String get(String key){
    try {
        readLock.lock();
        return syncHashMap.get(key);
    } finally {
        readLock.unlock();
    }
}

public boolean containsKey(String key) {
    try {
        readLock.lock();
        return syncHashMap.containsKey(key);
    } finally {
        readLock.unlock();
    }
```

Condition

Интерфейсное условие ***Condition*** в сочетании с блокировкой *Lock* позволяет заменить методы монитора (wait, notify и notifyAll) объектом, управляющим ожиданием событий. Блокировка Lock заменяет использование synchronized, а Condition — объектные методы монитора.

Методы интерфейса Condition

- await() - переводит поток в состояние ожидания до тех пор, пока не будет выполнено некоторое условие или пока другой поток не вызовет методы signal/signalAll
- await(long time, TimeUnit unit) - переводит поток в состояние ожидания на определенное время пока не будет выполнено некоторое условие или пока другой поток не вызовет методы signal/signalAll
- signal() - сигнализирует потоку, у которого ранее был вызван метод await(), о возможности продолжения работы. Применение аналогично использованию методу notify класса Object
- signalAll() - сигнализирует всем потокам, у которых ранее был вызван метод await(), о возможности продолжения работы. Применение аналогично использованию методу notifyAll класса Object

Условие *Condition (также называют* очередь условия), предоставляет средство управления для одного потока, чтобы приостановить его выполнение до тех пор, пока он не будет уведомлен другим потоком. Объект Condition связывают с блокировкой.

```
//Чтобы получить Condition для блокировки Lock используют метод newCondition()
ReentrantLock locker = new ReentrantLock();
Condition condition = locker.newCondition();

//Чтобы перевести поток в ожидание, если определенное условие не выполняется,
//то используется метод await
while (условие)
    condition.await();

//После завершения всех действий в потоке (при выходе) подается сигнал
//об изменении условия другим потокам
condition.signalAll();
```

Классы-сихронизаторы из пакета java.util.concurrent

Semaphore

Класс Semaphore

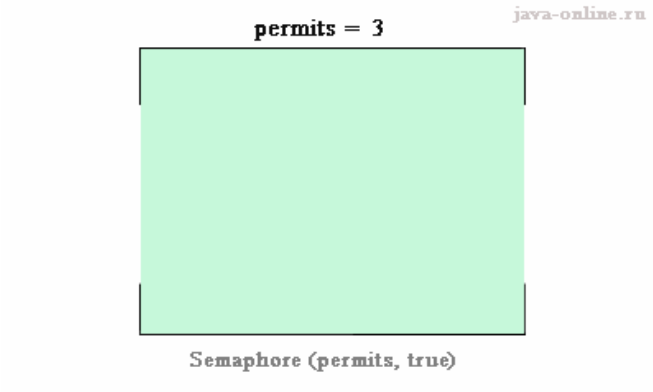
ИТМО

- Управляет несколькими разрешениями на доступ
- **Semaphore**(**int permits**, boolean fair)
- Каждый поток:
  - получает разрешение - semaphore.**acquire**()
    - `while (permits == 0) wait; permits--;`
  - возвращает разрешение - semaphore.**release**()
    - `permits++;`
- **Semaphore(1)** - бинарный семафор (mutex)

Semaphore ограничивает доступ к общему ресурсу, в качестве которого могут выступать программные/аппаратные ресурсы или файловая система.

Для управления доступом к общему ресурсу Semaphore использует **счетчик**.

Если значение счетчика больше 0, то поток исполнения получает разрешение, после чего значение счетчика семафора уменьшается на единицу. При значении счетчика равным 0 очередному потоку исполнения в доступе будет отказано, и он будет заблокирован до тех пор, пока не будут освобождены ресурсы.




```
//2 конструктора
Semaphore(int permits);
Semaphore(int permits, boolean fair);
```

Параметр **permits** определяет исходное значение счетчика разрешений т.е. количество потоков исполнения, которым может быть одновременно предоставлен доступ к общему ресурсу. По умолчанию ожидающим потокам предоставляется разрешение в неопределенном порядке. Если же использовать второй конструктор и параметру

справедливости *fair* присвоить значение true, то разрешения будут предоставляться ожидающим потокам исполнения в том порядке, в каком они его запрашивали.

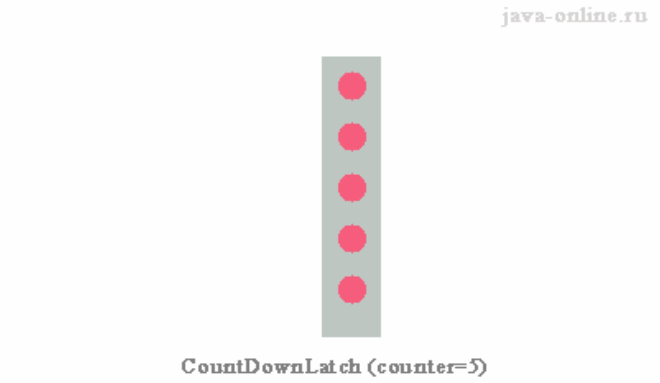
CountDownLatch

Класс CountDownLatch

ИТМО

- Открывает доступ после обратного отсчета
- **CountDownLatch(int count)**
- Каждый поток:
  - извещает о событии - latch.**countDown**()
    - `count--;`
  - ждет разрешения - latch.**await**() :: void
    - `while (count > 0) wait;`

CountDownLatch представляет собой «защелку с обратным отсчетом» : несколько потоков, выполняя определенный код, блокируются до тех пор, пока не будут выполнены заданные условия. Количество условий определяются счетчиком. Как только счетчик обнулится, т.е. будут выполнены все условия, самоблокировки выполняемых потоков снимаются, и они продолжают выполнение кода.




```
CountDownLatch(int number); //конструктор
void await() throws InterruptedException; //метод самоблокировки
void countDown(); //метод уменьшения счётчика
```

CyclicBarrier

Класс CyclicBarrier

ИТМО

- Синхронизация группы потоков
- **CyclicBarrier**(**int parties**, Runnable task)
- Каждый поток:
  - ждет остальных - barrier.**await**() :: int // `--parties`
    - `if (parties > 0) wait;`
  - последний поток открывает барьер - **notifyAll**
    - перед открытием выполняет задачу - task.`run()`
  - сброс барьера - `barrier.reset()`
  - барьер может сломаться - BrokenBarrierException

CyclicBarrier представляет собой барьерную синхронизацию. Особенно эффективно использование барьеров при циклических расчетах. При барьерной синхронизации алгоритм расчета делят на несколько потоков. С помощью барьера организуют точку сбора частичных результатов вычислений, в которой подводится итог этапа вычислений.

Барьер для группы потоков означает, что каждый поток должен остановиться в определенном месте и ожидать прихода остальных потоков группы. Как только все потоки достигнут барьера, их выполнение продолжится.

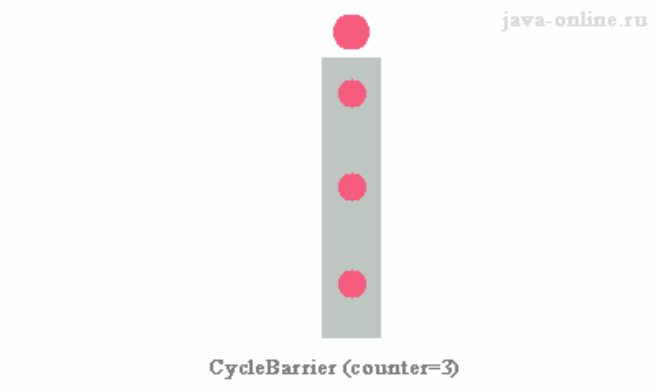
```
//2 конструктора
CyclicBarrier(int count);
CyclicBarrier(int count, Runnable class);

void await() throws InterruptedException
boolean await(long wait, TimeUnit unit) throws InterruptedException;
```

В первом конструкторе задается количество потоков, которые должны достигнуть барьера, чтобы после этого одновременно продолжить выполнение кода.

Во втором конструкторе дополнительно задается реализующий интерфейс Runnable класс, который должен быть запущен после прихода к барьеру всех потоков. Поток запускать самостоятельно НЕ НУЖНО. **CyclicBarrier** это делает автоматически.

Циклический барьер **CyclicBarrier** похож на **CountDownLatch**. Главное различие между ними связано с тем, что «защелку» нельзя использовать повторно после того, как её счётчик обнулится, а барьер можно использовать (в цикле). С точки зрения API циклический барьер CyclicBarrier имеет только метод самоблокировки await и не имеет метода декрементации счетчика, а также позволяет подключить и автоматически запускать дополнительный потоковый класс при достижении барьера всех исполняемых потоков.



Phaser

Класс Phaser

ИТМО

- Универсальный барьер-защелка
- **Phaser**(Phaser parent, int parties)
  - phase = 0 (номер фазы, возвращается методами)
- Действия потоков:
  - **register**() - регистрация
  - **arrive**() - прибытие
    - **arriveAndDeregister**() - и отмена регистрации
    - **arriveAndAwaitAdvance**() - и ожидание остальных
  - все прибыли - phase++ и поехали дальше

Phaser (фазировщик), как и CyclicBarrier, является реализацией объекта синхронизации типа «Барьер» (CyclicBarrier). В отличии от CyclicBarrier, **Phaser** предоставляет больше гибкости.

Важные особенности **Phaser** :

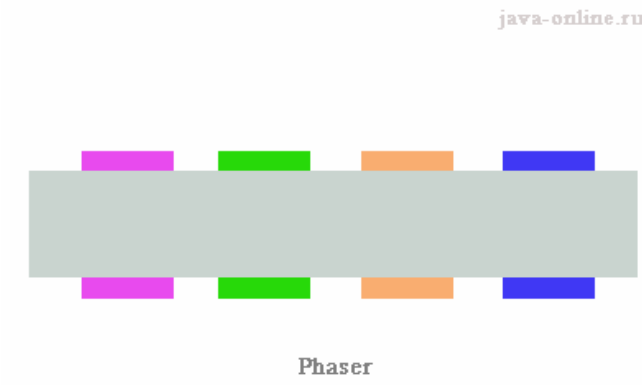
1. Phaser может иметь несколько фаз (барьеров). Если количество фаз равно 1, то плавно переходим к CyclicBarrier (осталось только все исполнительные потоки остановить у барьера).
2. Каждая фаза (цикл синхронизации) имеет свой номер.
3. Количество участников-потоков для каждой фазы жестко не задано и может меняться. Исполнительный поток может регистрироваться в качестве участника и отменять свое участие;
4. Исполнительный поток не обязан ожидать, пока все остальные участники соберутся у барьера. Достаточно только сообщить о своем прибытии.

```
Phaser();
Phaser(int parties);
Phaser(Phaser parent);
Phaser(Phaser parent, int parties);
```

Параметр parties определяет количество участников, которые должны пройти все фазы. Первый конструктор создает объект Phaser без каких-либо участников. Второй конструктор регистрирует передаваемое в конструктор количество участников. Третий и четвертый конструкторы дополнительно устанавливают родительский объект Phaser.

При создании экземпляр класса Phaser находится в нулевой фазе. В очередном состоянии (фазе) синхронизатор находится в ожидании до тех пор, пока все зарегистрированные потоки не завершат данную фазу. Потоки извещают об этом, вызывая один из методов arrive() или arriveAndAwaitAdvance().

Метод	Описание
int register()	Метод регистрирует участника и возвращает номер текущей фазы.
int arrive()	Метод указывает на завершения выполнения текущей фазы и возвращает номер фазы. Если же работа Phaser закончена, то метод вернет отрицательное число. При вызове метода arrive поток не приостанавливается, а продолжает выполняться.
int arriveAndAwaitAdvance()	Метод вызывается потоком/участником, чтобы указать, что он завершил текущую фазу. Это аналог метода CyclicBarrier.await(), сообщающего о прибытии к барьеру.
int arriveAndDeregister()	Метод arriveAndDeregister сообщает о завершении всех фаз участником и снимается с регистрации. Данный метод возвращает номер текущей фазы или отрицательное число, если Phaser завершил свою работу
int getPhase()	Получение номера текущей фазы.



Exchanger<V>

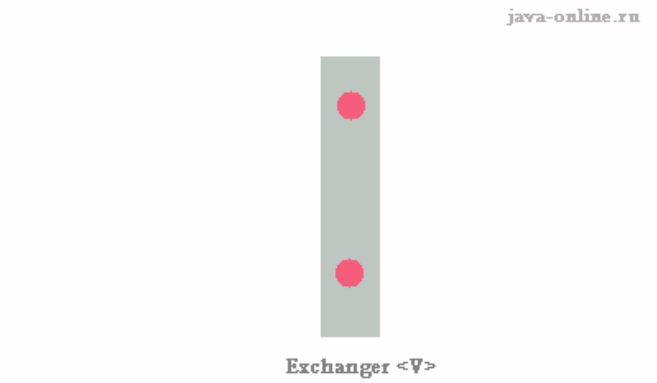
Класс Exchanger<V>

- 2 потока синхронно меняются объектами
- **Exchanger**()
- Потоки:
  - обмен по готовности - V **exchange**(V obj)

Exchanger (обменник) предназначен для упрощения процесса обмена данными между двумя потоками исполнения. Принцип действия класса Exchanger связан с ожиданием того, что два отдельных потока должны вызвать его метод **exchange**. Как только это произойдет, **Exchanger** произведет обмен данными, предоставляемыми обоими потоками. Обменник является обобщенным классом, он параметризируется типом объекта передачи `Exchanger<V>();`

```
//методы класса Exchanger
V exchange(V buffer) throws InterruptedException;
V exchange(V buffer, long wait, TimeUnit unit)
                throws InterruptedException;
```

Параметр buffer является ссылкой на обмениваемые данные. Метод возвращает данные из другого потока исполнения. Вторая форма метода позволяет определить время ожидания.



Модификатор volatile. Атомарные типы данных и операции.

Модификатор volatile и happens-before

- Модификатор **volatile** — переменная может измениться не в текущем потоке
- **Операции чтения-записи** переменной с модификатором volatile должны выполняться **без использования кэша**
- **Порядок операций** чтения-записи переменной с модификатором volatile **не должен меняться** — должно соблюдаться отношение «**happens-before**»
  - Значения переменных, видимые в потоке 1 до записи значения в volatile переменную должна быть видны в потоке 2 после чтения volatile переменной

volatile означает доступ к переменной через память (не через кэш)

Может возникнуть ситуация, что один поток изменит значение общей переменной, а второй поток будет продолжать работать с ее **старым значением** из своего **кэша**.

Также, в отличие от других примитивных типов данных, операции чтения и записи long и double не являются **атомарными** из-за их большого размера (64 бита).

Эти две проблемы решает модификатор **volatile:**

1. Операции чтения и записи **volatile** переменной являются **атомарными**.
2. Переменная не будет помещаться в **кэш**: результат записи значения в **volatile** переменную одним потоком будет **виден всем другим потокам**, которые используют эту переменную для чтения.

Атомарные типы данных и операции

пакет java.util.concurrent.atomic

- Атомарная операция — операция, выполняющаяся без промежуточных состояний.
- Атомарные операции не вызывают состояние гонок
- операции чтения-записи ссылок и примитивных типов (кроме long и double) — атомарные
- Пакет java.util.concurrent.atomic
  - AtomicInteger, AtomicLong,
  - AtomicBoolean, AtomicReference
  - AtomicIntegerArray
  - LongAccumulator, DoubleAccumulator
  - LongAdder, DoubleAdder

```
class Count {
  AtomicInteger counter =
    new AtomicInteger(0);
  public void up() {
    counter.incrementAndGet();
  }
  public void down() {
    counter.decrementAndGet();
  }
}
```

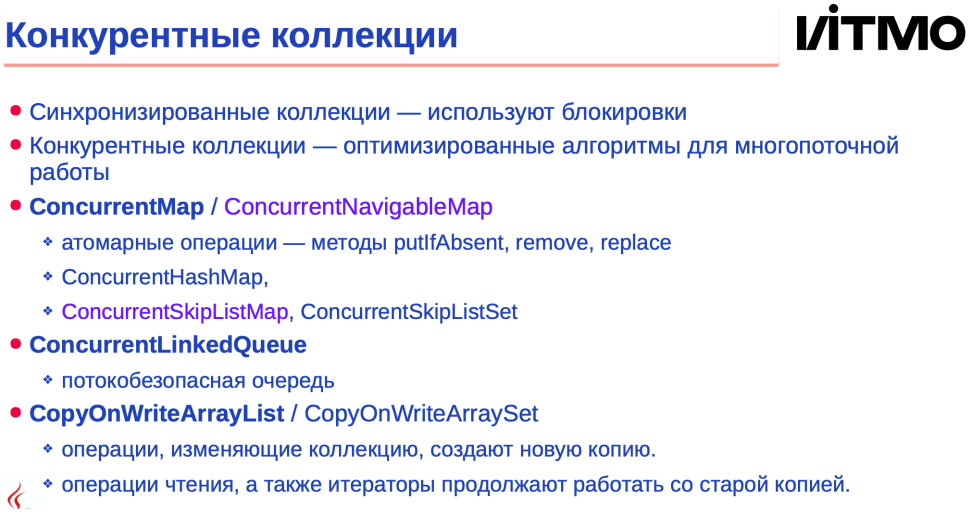
Для работы с переменными в многопоточной среде, когда важна **атомарность операций**, используются специальные классы из пакета java.util.concurrent.atomic, например:

- AtomicInteger
- AtomicLong
- AtomicBoolean
- AtomicReference<T>

Эти классы предоставляют **методы атомарного обновления**, такие как:

- get(), set(value) — атомарные чтение и запись
- incrementAndGet(), decrementAndGet() — атомарное увеличение и уменьшение
- compareAndSet(expected, update) — атомарная замена, если текущее значение равно ожидаемому

Коллекции из пакета java.util.concurrent.



Пакет **java.util.concurrent** предлагает свой набор потокобезопасных классов, допускающих разными потоками одновременное чтение и внесение изменений. Итераторы классов данного пакета представляют данные на определенный момент времени. Все операции по изменению коллекции (add, set, remove) приводят к созданию новой копии внутреннего массива. Этим гарантируется, что при проходе итератором по коллекции не будет **ConcurrentModificationException**. Следует помнить, что при копировании массива копируются только ссылки на объекты.

CopyOnWriteArrayList реализует алгоритм CopyOnWrite и является потокобезопасным аналогом ArrayList. Класс CopyOnWriteArrayList содержит изменяемую ссылку на неизменяемый массив, обеспечивая преимущества потокобезопасности без необходимости использования блокировок. Т.е. при выполнении модифицирующей операции CopyOnWriteArrayList создаёт новую копию списка и гарантирует, что её итераторы вернут состояние списка на момент создания итератора и не вызовут *ConcurrentModificationException*.

CopyOnWriteArrayList следует использовать вместо ArrayList в потоконагруженных приложениях, где могут иметь место нечастые операции вставки и удаления в одних потоках и одновременный перебор в других. Это типично для случая, когда коллекция ArrayList используется для хранения списка объектов.

ConcurrentHashMap<K, V> реализует интерфейс *java.util.concurrent.ConcurrentMap* и отличается от HashMap и Hashtable внутренней структурой хранения пар key-value. ConcurrentHashMap использует несколько сегментов, и данный класс можно рассматривать как группу HashMap'ов. По умолчанию количество сегментов равно 16. Доступ к данным определяется по сегментам, а не по объекту. Итераторы данного класса фиксируют структуру данных на момент начала его использования.

Если пара key-value хранится в 10-ом сегменте, то ConcurrentHashMap заблокирует, при необходимости, только 10-й сегмент, и не будет блокировать остальные 15.

CopyOnWriteArraySet выполнен на основе CopyOnWriteArrayList с реализацией интерфейса Set. Лучше всего CopyOnWriteArraySet использовать для read-only коллекций небольших размеров. Если в данных коллекции произойдут изменения, накладные расходы, связанные с копированием, не должны быть ресурсоёмкими.

ConcurrentNavigableMap расширяет возможности интерфейса NavigableMap для использования в многопоточных приложениях; итераторы класса декларируются как потокобезопасные и не вызывают ConcurrentModificationException.

ConcurrentSkipListMap является аналогом коллекции TreeMap с сортировкой данных по ключу и с поддержкой многопоточности.

ConcurrentSkipListSet выполнен на основе ConcurrentSkipListMap с реализацией интерфейса Set.

Интерфейсы Executor, ExecutorService, Callable, Future

Основные интерфейсы: Executor, ExecutorService и фабрика Executors.

Объекты, которые реализуют интерфейс **Executor**, могут выполнять runnable-задачу.

После вызова этого метода и передачи задачи на выполнение задача будет выполнена асинхронно. Также этот интерфейс разделяет, кто будет выполнять задачу и что будет выполняться, — в отличии от класса Thread.

Executor

- базовый интерфейс для выполнения задач.
- метод: `void execute(Runnable command)`
- **асинхронное выполнение** задачи без возврата результата.

```
Executor executor = Executors.newFixedThreadPool(2);
executor.execute(() → System.out.println("Hello"));
```

ExecutorService

Этот интерфейс наследуется от интерфейса Executor и предоставляет возможности для выполнения заданий Callable, для прерывания выполняемой задачи и завершения работы пула потоков.

Позволяет

- запускать задачи с результатом (`submit()`)
- управлять завершением задач (`shutdown()` / `shutdownNow()`)

```
//1 пример
ExecutorService service = Executors.newFixedThreadPool(3);
service.shutdown();
```

Callable<V> и Future<V>

При работе многопоточного приложения часто необходимо получение от потока результата его деятельности в виде некоторого объекта. Эту задачу можно решить с использованием интерфейсов **Callable**<V> и **Future**<V>

Интерфейс **Callable**<V> очень похож на интерфейс Runnable. Однако, в отличие от Runnable, интерфейс **Callable** использует Generic'и для определения типа возвращаемого объекта. Runnable содержит метод run(), описывающий действие потока во время выполнения, а Callable – метод call().

Интерфейс **Future** также, как и интерфейс Callable, использует Generic'и. Методы интерфейса можно использовать для проверки завершения работы потока, ожидания завершения и получения результата. Результат выполнения может быть получен методом get, если поток завершил работу. Прервать выполнения задачи можно

методом cancel. Дополнительные методы позволяют определить завершение задачи : нормальное или прерванное.

Метод	Описание
cancel (boolean mayInterruptIfRunning)	попытка завершения задачи
V get()	ожидание (при необходимости) завершения задачи, после чего можно будет получить результат
V get(long timeout, TimeUnit unit)	ожидание (при необходимости) завершения задачи в течение определенного времени, после чего можно будет получить результат
isCancelled()	вернет true, если выполнение задачи будет прервано прежде завершения
isDone()	вернет true, если задача завершена

```
ExecutorService executor = Executors.newFixedThreadPool(3);
Callable<Integer> task = () → 10 + 20;
Future<Integer> result = executor.submit(task);
System.out.println("Результат: " + result.get());  // 30
executor.shutdown();
```

Пулы потоков

Создавать потоки для выполнения большого количества задач очень трудоемко: создание потока и освобождение ресурсов — дорогостоящие операции. Для решения проблемы ввели пулы потоков и очереди задач, из которых берутся задачи для пулов.

Пул потоков — своего рода контейнер, в котором содержатся потоки, которые могут выполнять задачи, и после выполнения одной самостоятельно переходить к следующей.

Он позволяет повторно использовать потоки.

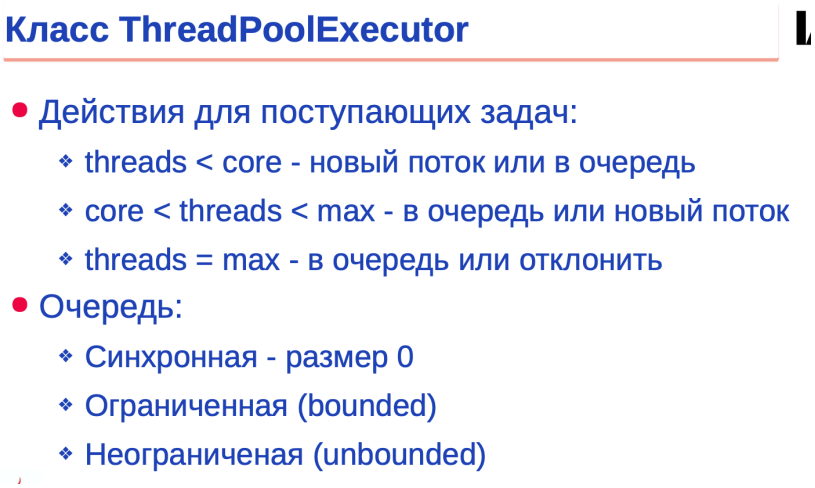
Вторая причина создания пулов потоков — возможность разделить объект, выполняющий код, и непосредственно код задачи, которую необходимо выполнить

Класс Executors — создает классы, которые реализуют интерфейсы **Executor** и **ExecutorService**. Основные реализации пула потоков:

1. **Executors.newFixedThreadPool(n)** - фиксированное кол-во потоков, новые задачи помещаются в очередь, если все потоки заняты
2. **Executors.newCachedThreadPool()** - создаёт новые потоки при необходимости, а неиспользуемые потоки автоматически удаляются спустя 60 секунд простоя
3. **Executors.newSingleThreadExecutor()** - только 1 поток, задачи выполняются строго по очереди
4. Executors.newWorkStealingPool() - использует **ForkJoinPool** с несколькими рабочими потоками, которые **динамически перераспределяют задачи** между собой. Подходит для параллельных задач, особенно если задачи короткие и часто создаются

ThreadPoolExecutor — это **реализация пула потоков**. Она позволяет **точно управлять** параметрами работы пула: количеством потоков, очередью задач, стратегией отклонения и временем жизни потоков.

Все фабрики из класса Executors (newFixedThreadPool(), newCachedThreadPool() и др.) **внутренне используют ThreadPoolExecutor**.




```
ThreadPoolExecutor executor = new ThreadPoolExecutor(
    int corePoolSize,
    int maximumPoolSize,
    long keepAliveTime,
    TimeUnit unit,
    BlockingQueue<Runnable> workQueue
);
```

- **corePoolSize** — минимальное количество потоков, которые всегда остаются активными, даже если нет задач.
- **maximumPoolSize** — максимальное число потоков, которое может быть создано при большой нагрузке.
- **keepAliveTime** — время ожидания перед завершением неосновных потоков (тех, что превышают corePoolSize), если они простаивают.
- **workQueue** — очередь, в которую ставятся задачи, если все corePoolSize потоки заняты.

Существует несколько типов очередей, которые можно использовать:

- **LinkedBlockingQueue** — неограниченная очередь. Подходит для `newFixedThreadPool()`. Если потоки заняты, задачи будут скапливаться в очереди.

- **SynchronousQueue** — очередь без хранения: каждая задача должна быть немедленно передана потоку. Используется в `newCachedThreadPool()`.
- **ArrayBlockingQueue** — очередь с фиксированным размером. Позволяет ограничить количество ожидающих задач.

JDBC. Порядок взаимодействия с базой данных. Класс DriverManager. Интерфейс Connection

Java DataBase Connectivity
JDBC API — высокоуровневый интерфейс для доступа к данным
JDBC Driver API — низкоуровневый интерфейс для создания драйверов

Процесс взаимодействия

ИТМО

```
Connection connection = DriverManager.getConnection(…);

  Statement statement = connection.createStatement();

    ResultSet resultSet = statement.executeQuery("SELECT …");

    while (resultSet.next()) {
    // получение и обработка данных
    }

    resultSet.close();
  statement.close();
connection.close();
```

Driver

- Интерфейс Driver — часть JDBC (Java Database Connectivity).
- Он **отвечает за установку соединения с БД**.
- Метод Connection connect(String url, Properties info) используется для подключения к базе данных по URL.
- Каждый производитель СУБД (например, Oracle, PostgreSQL) **реализует этот интерфейс** в своём JDBC-драйвере.

DriverManager

Управляет списком драйверов **баз данных**, загрузка драйевра: Class.forName

Предоставляет метод `getConnection(...)` для **установки соединения** с БД

Connection

Интерфейс Connection

- Абстракция соединения (сессия)
  - методы:
    - Statement createStatement()
    - PreparedStatement prepareStatement(String sql)
    - CallableStatement prepareCall(String sql)

    - DatabaseMetaData getMetaData()

Интерфейсы Statement, PreparedStatement, ResultSet, RowSet

Семейство интерфейсов Statement

Statement

- Статический SQL-запрос
- Запрос передается как параметр при выполнении

```
String query = "SELECT * FROM table WHERE id = 15";
Statement st = connection.createStatement();
st.executeQuery(query)
```

PreparedStatement

Наследуется от Statement

- Динамический запрос с подстановкой
- Запрос передается как параметр при создании

```
String query = "SELECT * FROM table WHERE id = ?";
PreparedStatement ps = connection.prepareStatement(query);
ps.setInt(1, 15); //1 — номер параметра, 15 — значение

SELECT * FROM table WHERE id = 15
```

Флаги — это специальные **константы**, которые передаются как аргументы в методы, чтобы **влиять на поведение запроса** или соединения.

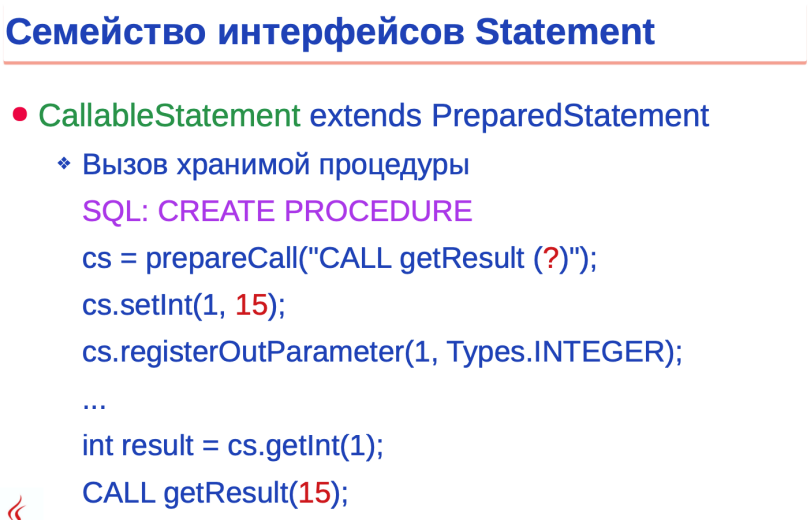
Statement.RETURN_GENERATED_KEYS

- Указывает, что нужно вернуть **автогенерируемые ключи** (например, автоинкрементный id после INSERT).

Используется при создании PreparedStatement:

```
connection.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS);
```

CallableStatement



SQL выражение с возможностью получить возвращаемое значение из хранимых процедур (SQL Stored Procedures).

ResultSet

Этот интерфейс представляет результирующий набор базы данных. Он обеспечивает приложению построчный доступ к результатам запросов в базе данных.

Основные методы:

- next() — переходит к следующей строке (перемещает курсор к строке результата)
- getInt("column"), getString("column") — читают данные из текущей строки

RowSet

- RowSet расширяет интерфейс ResultSet, поэтому его функции более мощные, чем у ResultSet.
- RowSet более гибко перемещается по данным таблицы и может прокручивать назад и вперед (функции `previous()`, `next()`, `first()`, `last()` и т.д)
- RowSet поддерживает кэшированные данные, которые можно использовать даже после закрытия соединения.

- RowSet поддерживает новый метод подключения, ты можешь подключиться к базе данных без подключения. Также он поддерживает чтение источника данных XML.
- RowSet поддерживает фильтр данных.