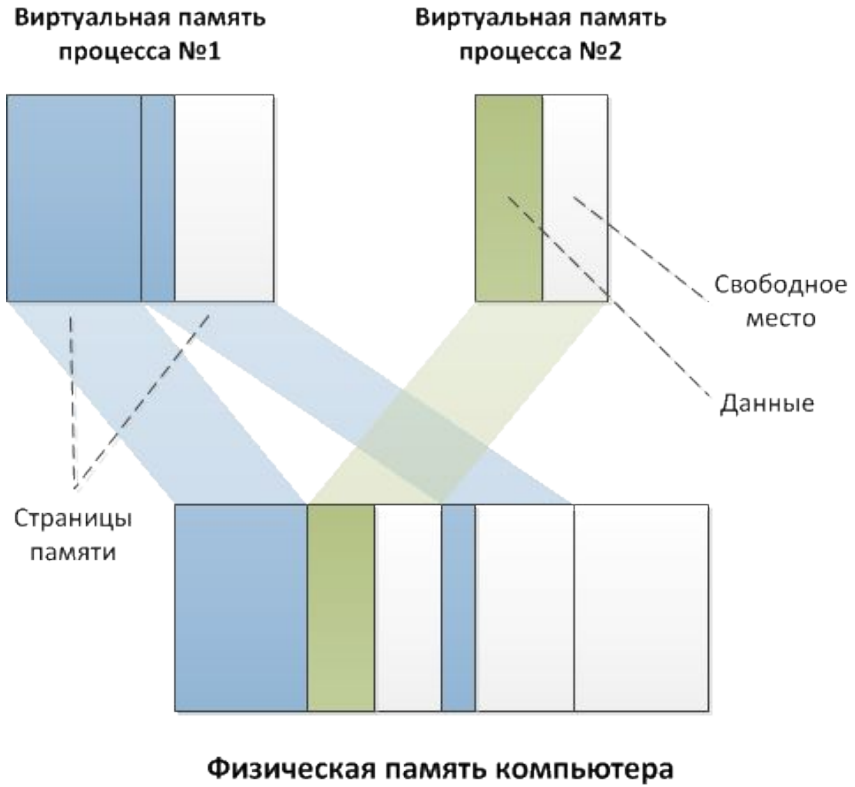
Java

Concurrency in Java

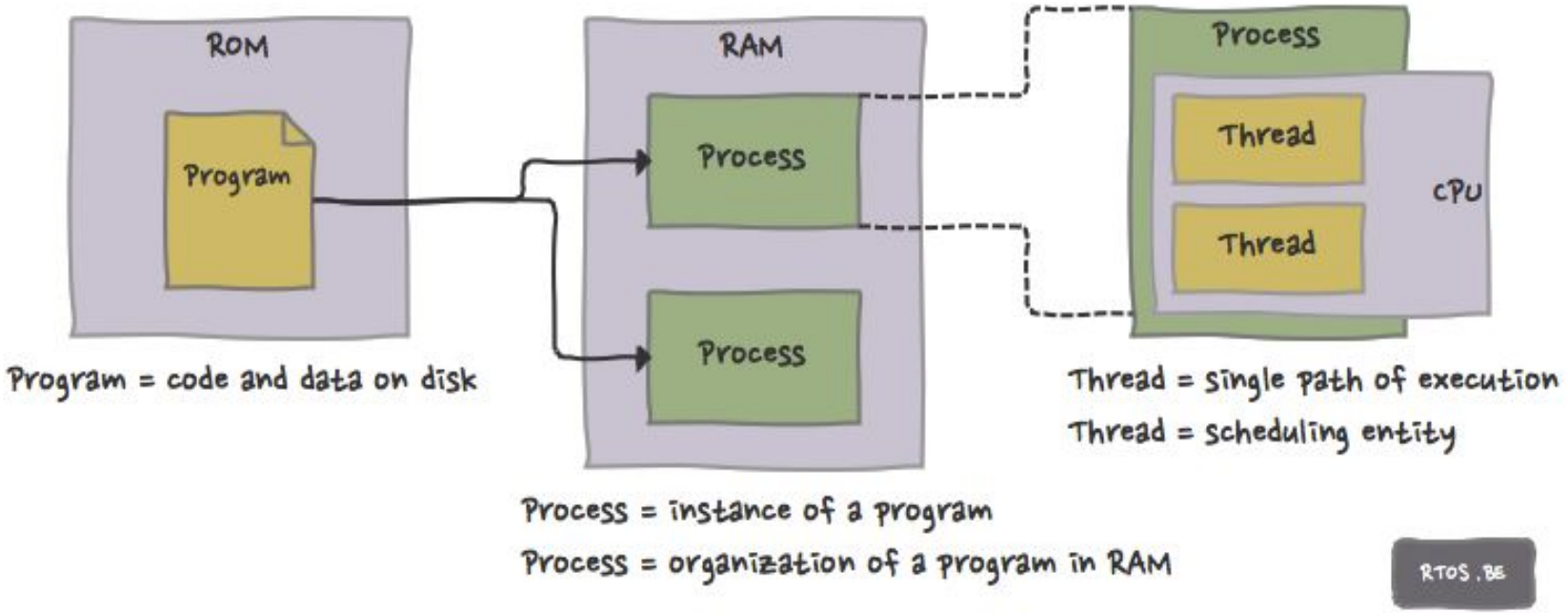
Concurrency in Java

Модель памяти
Создание потоков (Thread, Runnable)
Действия над потоками
Потокобезопасность, синхронизация
Semaphore, Монитор
Синхронизация потоков (synchronized, Lock)
Volatile
Happens-before
Atomic, CAS
ExecutorService, Callable, Future, CompletableFuture

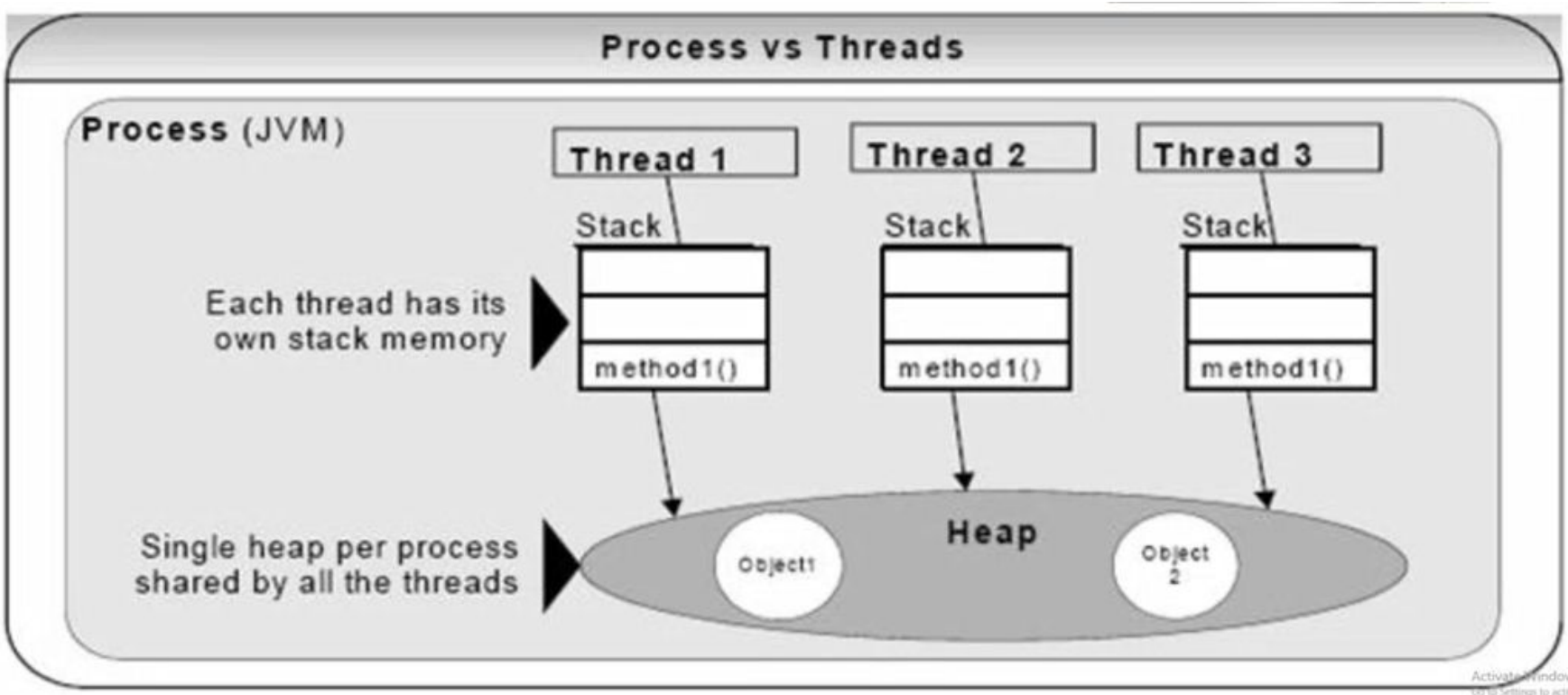
Модель памяти управляемой средствами ОС



Память – процесс - поток



Память – процесс - поток



Создание потока

- Thread : класс - наследуем
- Runnable : интерфейс – имплементируем (метод run())
- Callable<V> : интерфейс – имплементируем (метод call())

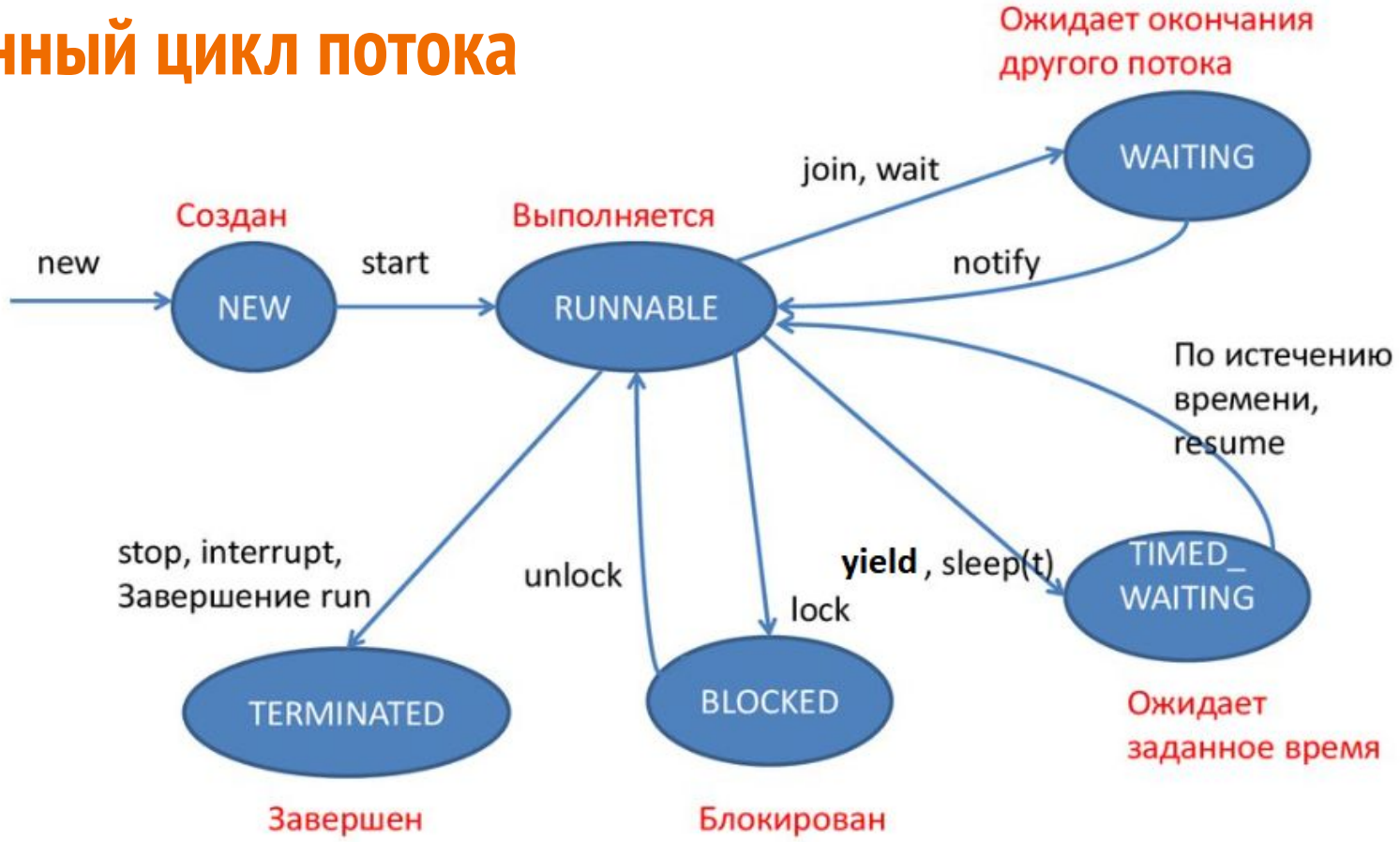
Примеры:

```
Runnable myRunnable = () -> someActions();

Thread myThread = new Thread(myRunnable, “MyRunnable Name”)

Thread myThread1 = new Thread(() -> someActions())

Executors.newSingleThreadExecutor().submit(this::someReturningMethod());

Callable<Integer> intCallable = () -> 1;

Executors.newSingleThreadExecutor().submit(intCallable);
```

Класс Thread

- run() - запуск потока. В нём пишите свой код
- start() - запустить поток
- setName() / getName() - задать /получить имя потока
- setPriority() / getPriority() - задать /получить приоритет потока
- isAlive() - определить, выполняется ли поток
- join() - ожидать завершение потока
- sleep() - приостановить поток на заданное время
- setDaemon(Boolean on) - пометить поток как демон, либо как пользовательский
- yield() - метод приостанавливающий поток на квант времени
- stop(), destroy(), resume(), suspend() - устаревшие (deprecated) методы управления жизненным циклом потока.

Жизненный цикл потока



Потокобезопасность, синхронизация

- Потокобезопасность - свойство объекта или кода, которое гарантирует, что при исполнении или использовании несколькими потоками, код будет вести себя, как предполагается.
- Синхронизация - это процесс, который позволяет выполнять потоки параллельно и согласованно.
  - Общие ресурсы – условие согласованного состояния
  - Critical section - участок исполняемого кода программы, в котором производится доступ к общему ресурсу (данным или устройству), который не должен быть одновременно использован более чем одним потоком исполнения.
- Средства для синхронизации:
  - wait()/notify() на мониторе (мьютексе) через synchronized
  - join()
  - Классы Lock, Semaphore. (java.util.concurrent)

Синхронизация потоков

- synchronized – ключевое слово для объявления синхронизированного блока кода (метода).
- join() - механизм, позволяющий одному потоку ждать завершения выполнения другого
- java.util.concurrent:
  - Lock – интерфейс явных блокировок (ReentrantLock, ReadWriteLock).
  - Объекты синхронизации:
    - Semaphore - ограничивающий количество потоков, которые могут «войти» в заданный участок кода
    - CountDownLatch - разрешающий вход в заданный участок кода при выполнении определенных условий
    - CyclicBarrier - типа «барьер», блокирующий выполнение определенного кода для заданного количества потоков
    - Exchanger - позволяющий провести обмен данными между двумя потоками
    - Phaser - типа «барьер», но в отличие от CyclicBarrier, предоставляет больше гибкости

Lock

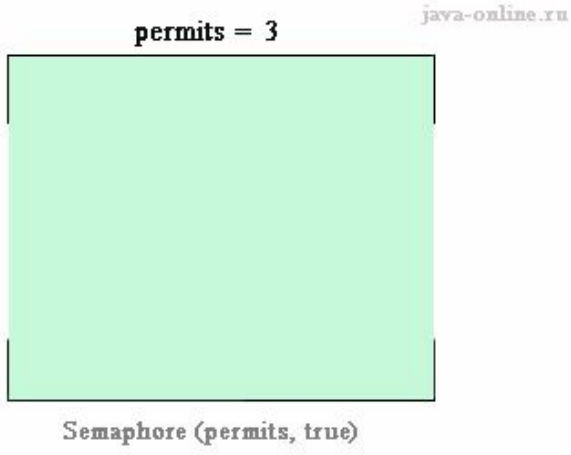
- Lock – интерфейс явных блокировок (ReentrantLock, ReadWriteLock).
  - lock() - получение блокировки
  - lockInterruptibly() - получение блокировки, если текущий поток не прерывается
  - newCondition() - получение нового Condition, связанного с блокировкой Lock
  - tryLock() - получение блокировки, если она свободна во время вызова
  - tryLock(long time, TimeUnit unit) - получение блокировки в течение заданного времени
  - unlock() - освобождение блокировки

```
Lock l = ...;
l.lock();
try
{
    //действия над ресурсом, защищенным данной блокировкой
}
finally
{
    l.unlock() //гарантия того, что блокировка будет отпущена
}
```

Semaphore, Монитор

Semaphore - шаблон синхронизации Семафор: доступ к блоку кода управляется с помощью счётчика.

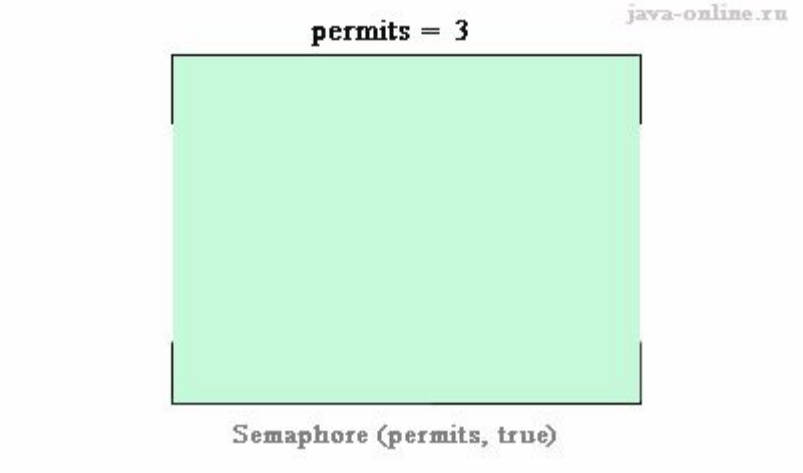
«Объект, ограничивающий количество потоков, которые могут войти в заданный участок кода.» (Э. Дейкстра)



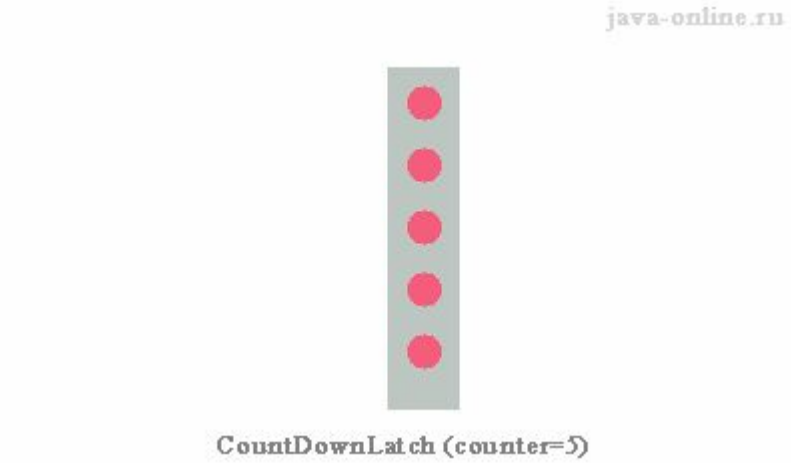
Монитор - инструмент для управления доступа к объекту. По сути - бинарный семафор. Монитор состоит из mutex-а и массива ожидающих очереди потоков.

Объекты синхронизации

Semaphore - ограничивающий количество потоков, которые могут «войти» в заданный участок кода

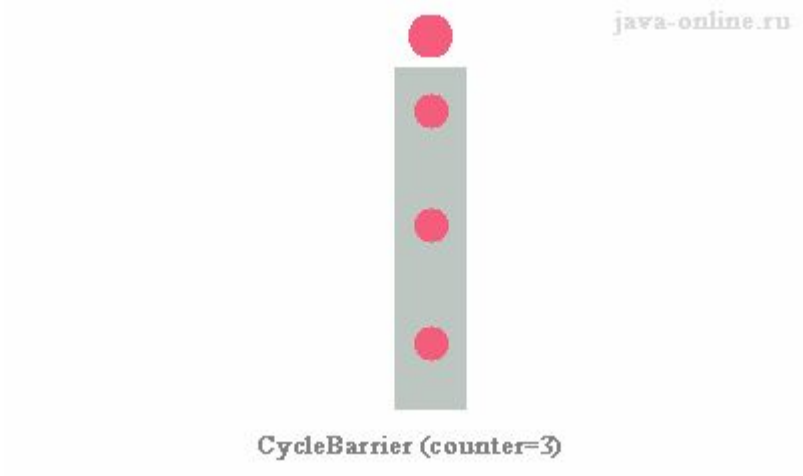


CountDownLatch - разрешающий вход в заданный участок кода при выполнении определенных условий

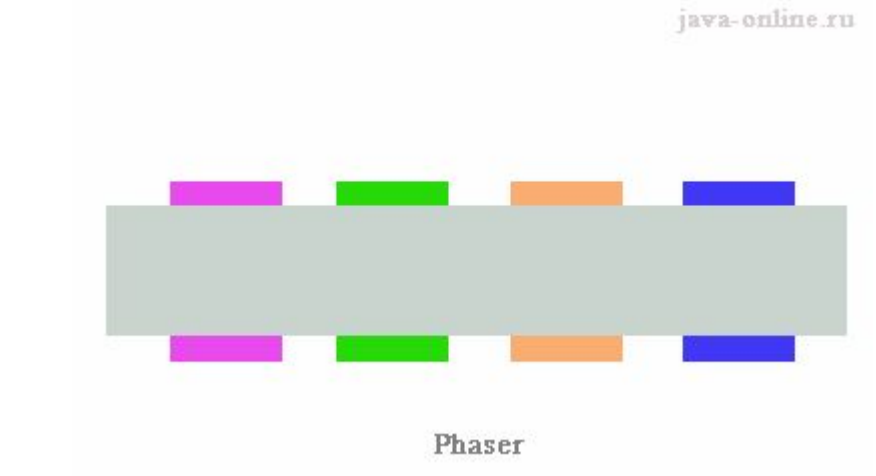


Объекты синхронизации

CyclicBarrier - типа «барьер», блокирующий выполнение определенного кода для заданного количества потоков

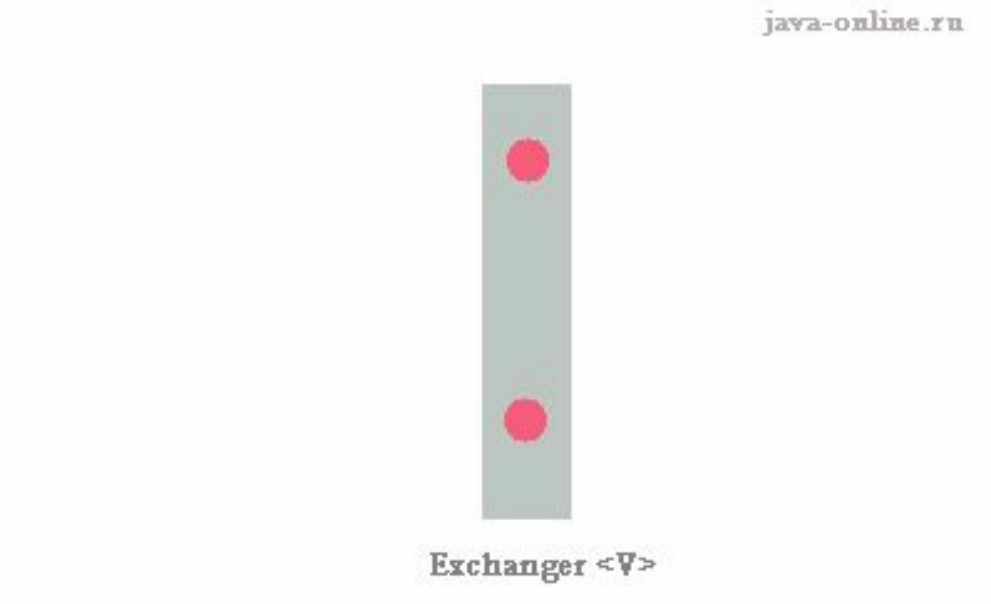


Phaser - типа «барьер», но в отличие от CyclicBarrier, предоставляет больше гибкости



Объекты синхронизации

Exchanger - позволяющий провести обмен данными между двумя потоками



Атомарные операции

- Атомарные операции – такие, которые либо выполняются полностью, либо не выполняются совсем. Не имеют проявлений побочных эффектов, пока операция не завершена.
- Операции чтения и записи атомарны для переменных ссылочных типов и большинства примитивных (кроме long и double)
- Операции чтения и записи атомарны для всех типов переменных, если они отмечены ключевым словом volatile
- Atomic-классы (AtomicInteger, AtomicLong, AtomicReference<V>)
  - Атомарные операции методов Atomic выполняются целиком, их выполнение не может быть прервано планировщиком потоков.
  - Оптимистичная блокировка
  - Аппаратная поддержка - compare-and-swap (CAS)
  - Методы: compareAndSet, getAndSet

Оптимистичная блокировка

Каждый атомарный класс включает метод compareAndSet, представляющий механизм оптимистичной блокировки и позволяющий изменить значение value только в том случае, если оно равно ожидаемому значению (т.е. current)

```
public class SimulatedCAS
{
    private int value;

    public synchronized int getValue() { return value; }

    public synchronized int compareAndSwap(int expectedValue, int newValue)
    {
        int oldValue = value;
        if (value == expectedValue)
        {
            value = newValue;
        }
        return oldValue;
    }
}
```

```
private volatile long value;

public final long get() {
    return value;
}

public final long getAndAdd(long delta) {
    while (true) {
        long current = get();
        long next = current + delta;
        if (compareAndSet(current, next))
            return current;
    }
}
```

???

Что напечатает данный код?

thread1() и thread2() запускаются в разных потоках

```
int x;
volatile int g;

public void thread1() {
    g = 1;
    x = 1;
}

public void thread2() {
    System.out.println(g);
    System.out.println(x);
}
```

Операции, связанные отношением happens-before

- happens-before - логическое ограничение на порядок выполнения инструкций программы. Если указывается, что запись в переменную и последующее ее чтение связаны через эту зависимость, то как бы при выполнении не переупорядочивались инструкции, в момент чтения все связанные с процессом записи результаты уже зафиксированы и видны
  - В рамках одного потока любая операция happens-before любой операцией, следующей за ней в исходном коде
  - Запись в volatile переменную happens-before чтение из той же самой переменной.
  - Освобождение монитора happens-before получения того же самого монитора
  - thread.start() happens-before thread.run()
  - Завершение thread.run() happens-before выход из thread.join()
  - ...

ExecutorService

- Сервис исполнителей – высокоуровневая замена работе с потоками напрямую.
  - Асинхронность
  - Пул потоков
  - Необходимо останавливать явно - shutdown()
- Executors – предоставляет удобные методы-фабрики для создания различных сервисов исполнителей (ThreadPoolExecutor, FixedThreadPool , ForkJoinPool, ScheduledThreadPoolExecutor)
- Работа с Callable<V> - результат возвращает в Future<V>

Future<V>

- Future<V> - специальный объект для запроса результата работы Callable<V>
- методы:
  - 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

Источники

- Lesson: Concurrency (The Java™ Tutorials > Essential Java Classes) (oracle.com)
- Многопоточность Thread, Runnable (java-online.ru)
- Многопоточный пакет util.concurrent (java-online.ru)
- Когда параллельные потоки буксуют / Хабр (habr.com)
- Многопоточное программирование в Java 8. Часть первая. Параллельное выполнение кода с помощью потоков (tproger.ru)
- Многопоточное программирование в Java 8. Часть вторая. Синхронизация доступа к изменяемым объектам (tproger.ru)
- Многопоточное программирование в Java 8. Часть третья. Атомарные переменные и конкурентные таблицы (tproger.ru)
- Java: продвинутая конкурентность / Хабр (habr.com)
- Справочник по синхронизаторам java.util.concurrent.*