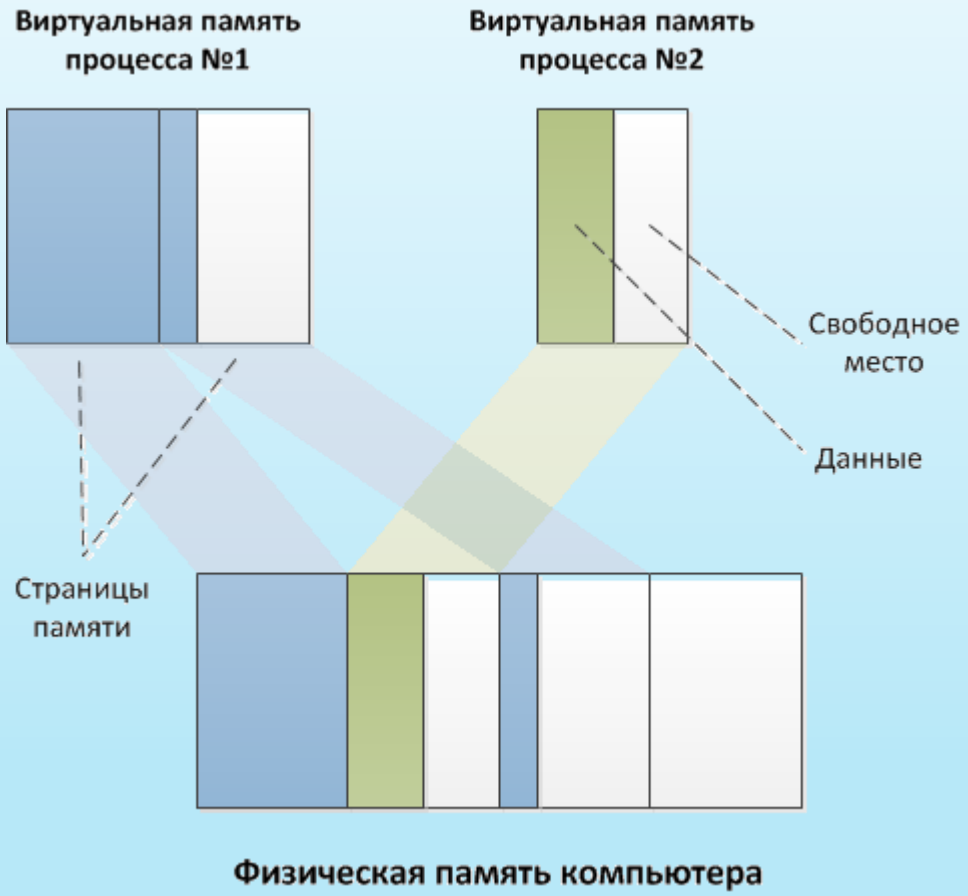
Concurrency in Java

Clevertec : Alexey Pavlyuchenkov

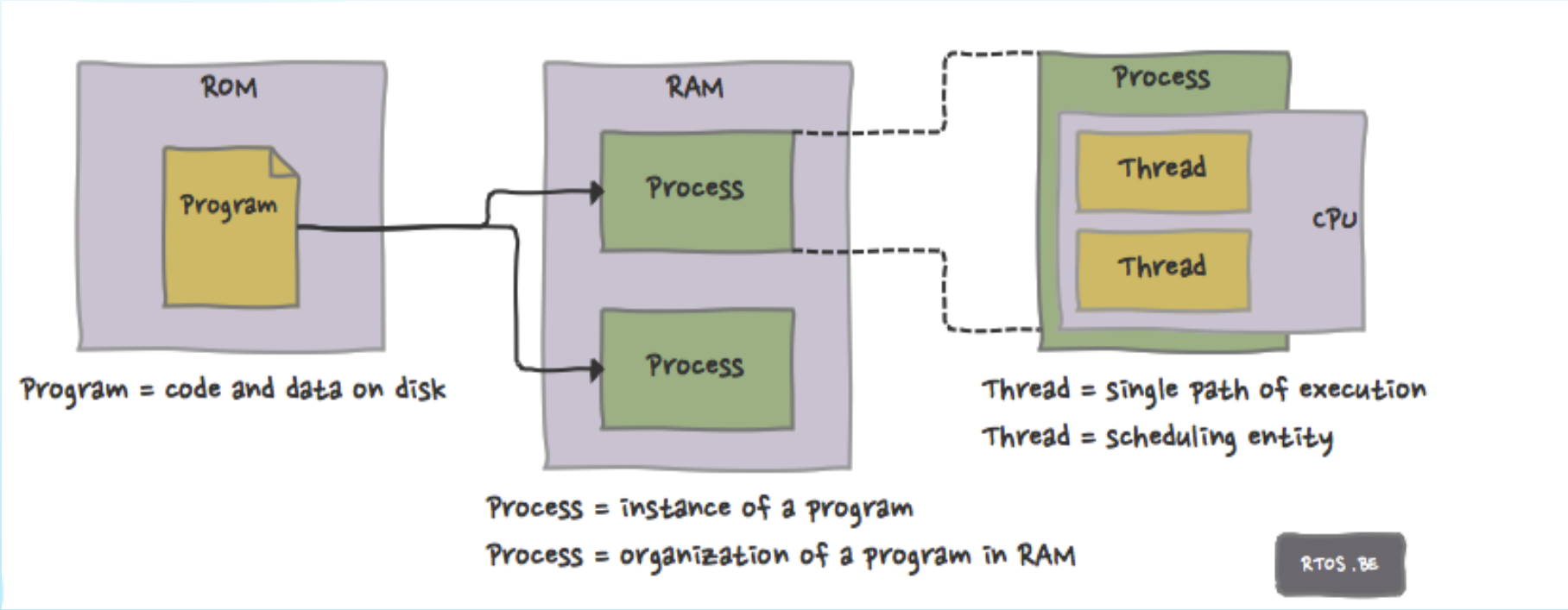
Concurrency in Java

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

Модель памяти управляемой средствами ОС



Память –> процесс –> поток



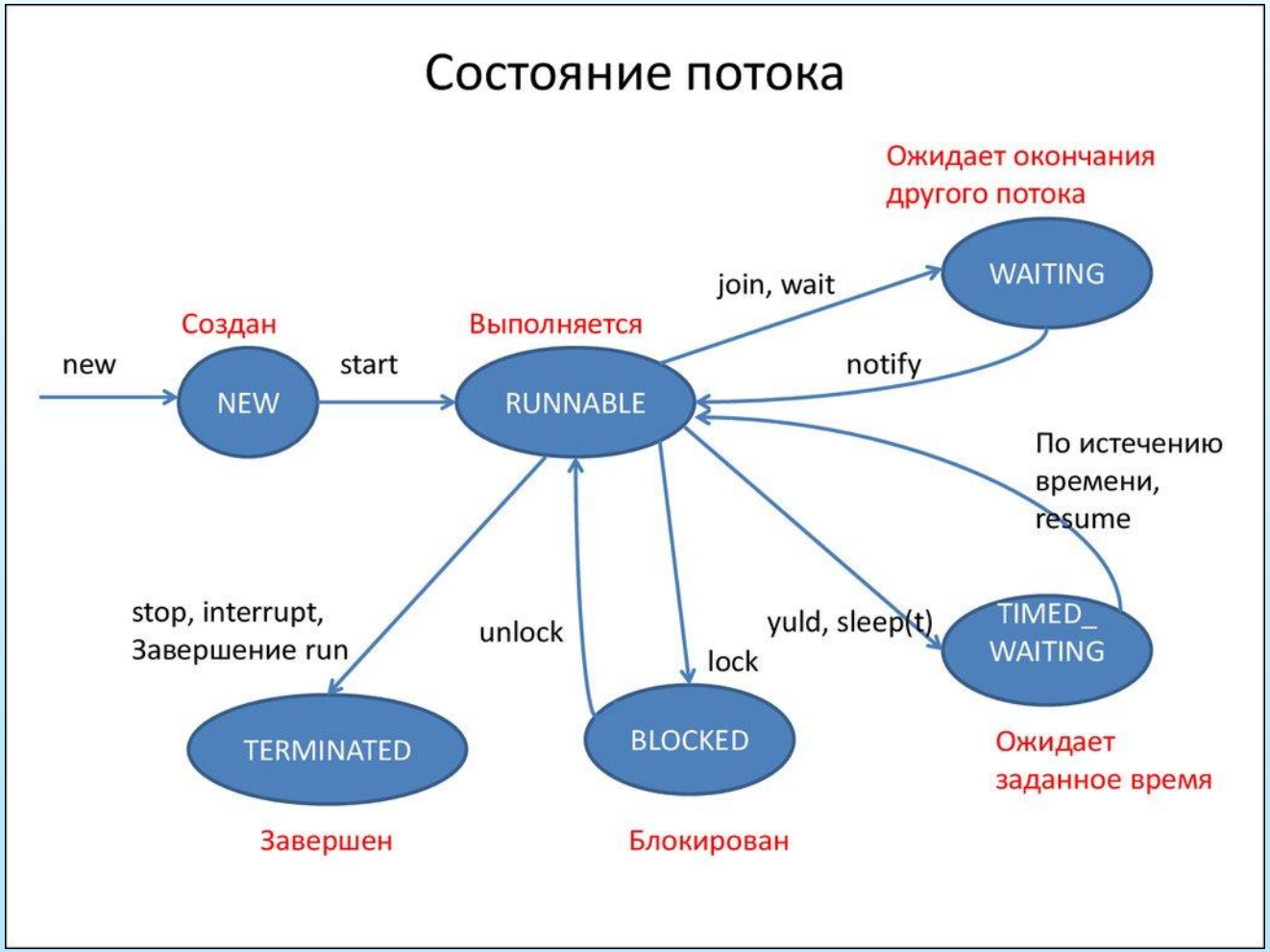
- потоки намного легче процессов поскольку требуют меньше времени и ресурсов
- переключение контекста между потоками намного быстрее, чем между процессами
- намного проще добиться взаимодействия между потоками, чем между процессами

Создание потоков

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

```
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);
```

Жизненный цикл потока



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()** - переводит поток из состояния running в состояние runnable, давая возможность другим потокам активизироваться.
- ~~**stop(), destroy(), resume(), suspend()**~~ - устаревшие (deprecated) методы управления жизненным циклом потока.
- **interrupt()** - сообщает потоку о необходимости остановки.

Модель памяти Java

- **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 с согласованным доступом к общим ресурсам.
  - **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.

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**

???

Что напечатает данный код (варианты) ?

thread1() и thread2() запускаются в разных потоках

```
int x;
int g;

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

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

???

Что напечатает данный код (варианты) ?

thread1() и thread2() запускаются в разных потоках

```
int x;
int g;

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

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

Ответ: [0,0], [0,1], [1,0], [1,1]

???

А такой 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);
}
```

???

А такой код????

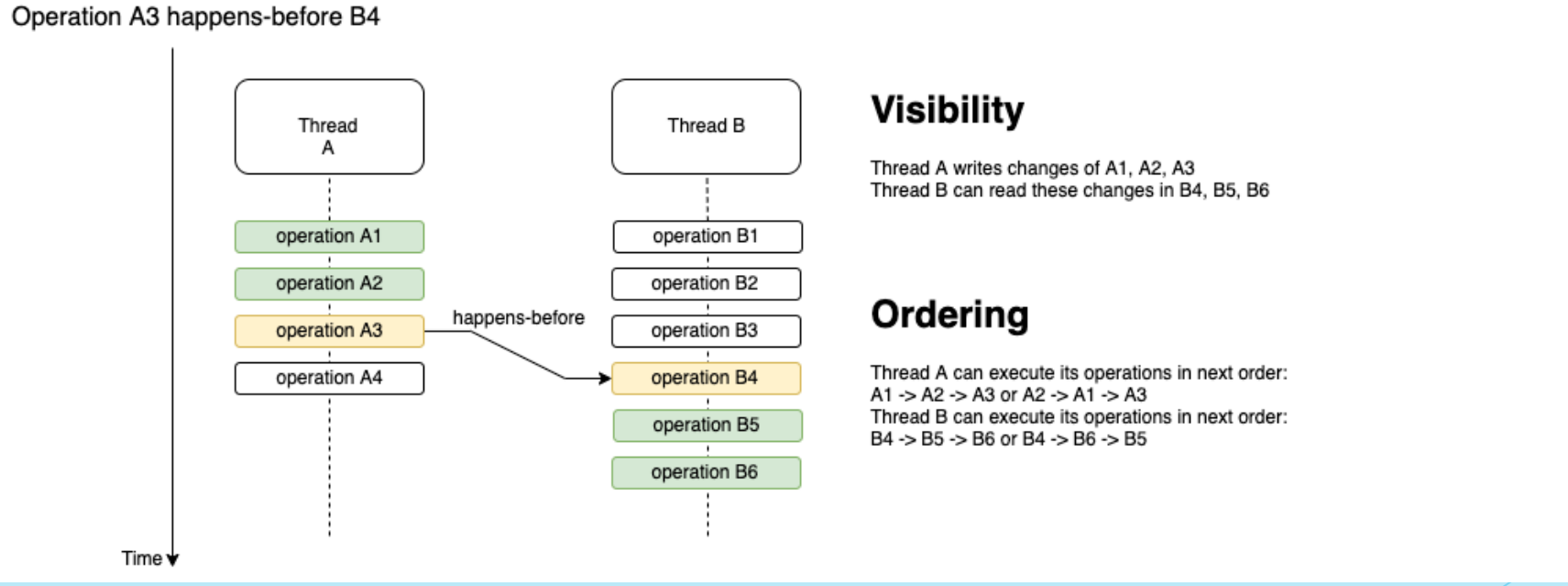
(thread1() и thread2() запускаются в разных потоках)

```
int x;
volatile int g;

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

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

Ответ: [0,0], [1,0], [1,1]

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



Примеры **happens-before**

- 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.
- Захват монитора (начало synchronized, метод lock) и всё, что после него в том же потоке
- Возврат монитора (конец synchronized, метод unlock) и всё, что перед ним в том же потоке
- Освобождение монитора happens-before получения того же самого монитора
- thread.start() happens-before thread.run()
- Завершение thread.run() happens-before выход из thread.join()
- Запись в final-поля в конструкторе и всё, что после конструктора.

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

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.

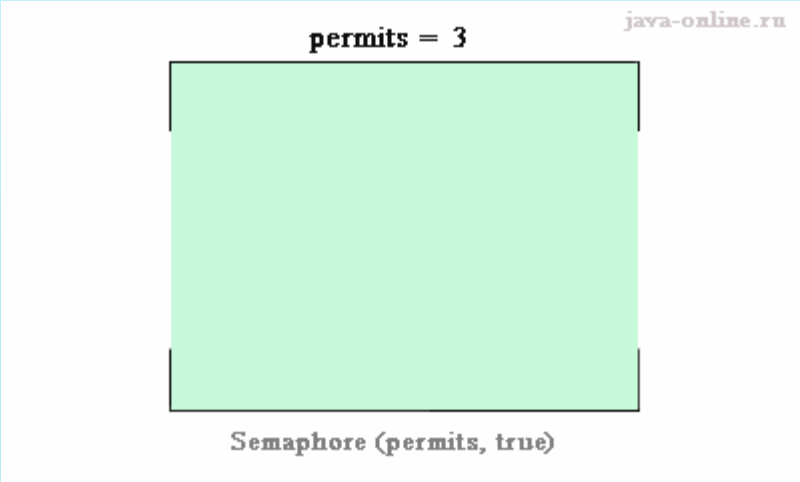
Lock

- **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
}
```

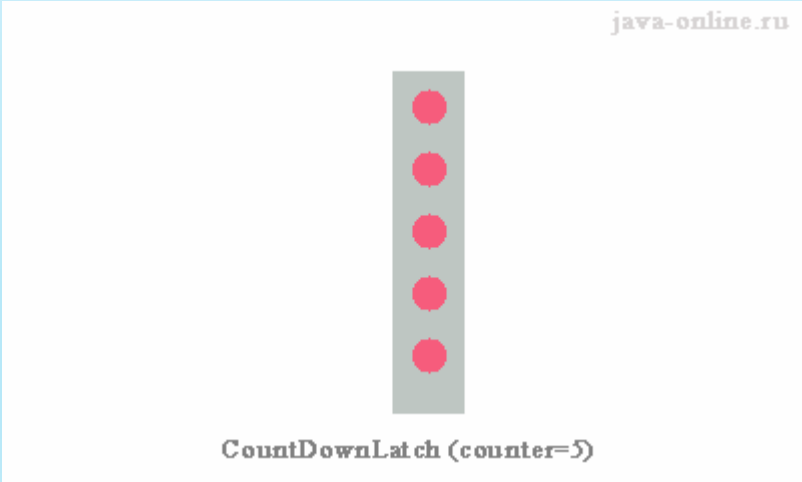
Объекты синхронизации

- **Semaphore** - ограничивающий количество потоков, которые могут «войти» в заданный участок кода
  - `Semaphore(int permits)`
  - `Semaphore(int permits, boolean fair)`
  - **`void acquire()`** или **`acquire(int number)`** – попытка получения разрешения семафора;
  - **`void release()`** или **`acquire(int number)`** – освобождение ранее полученного разрешения;



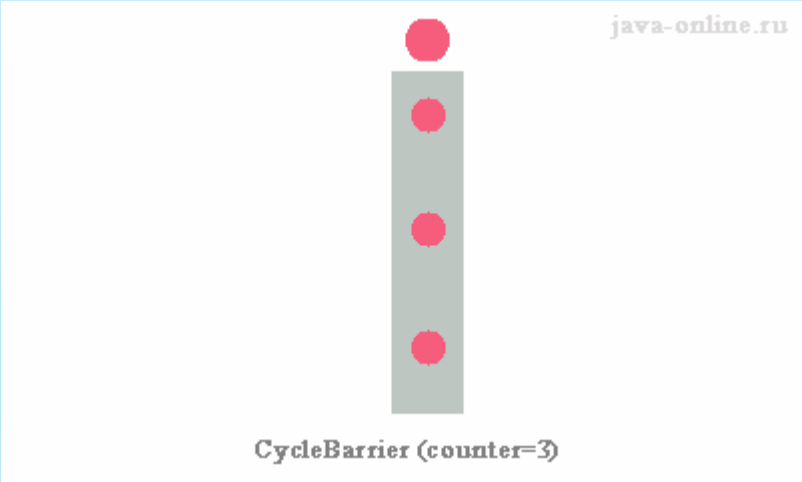
Объекты синхронизации

- **CountDownLatch** – типа «барьер» - «защелка с обратным отсчетом» - блокировка потоков до выполнения некоторых условий, выполнение которых определяется счетчиком.
  - `CountDownLatch(int count)`
  - `void await()` - самоблокировка
  - `boolean await(long wait, TimeUnit unit)` – самоблокировка по времени



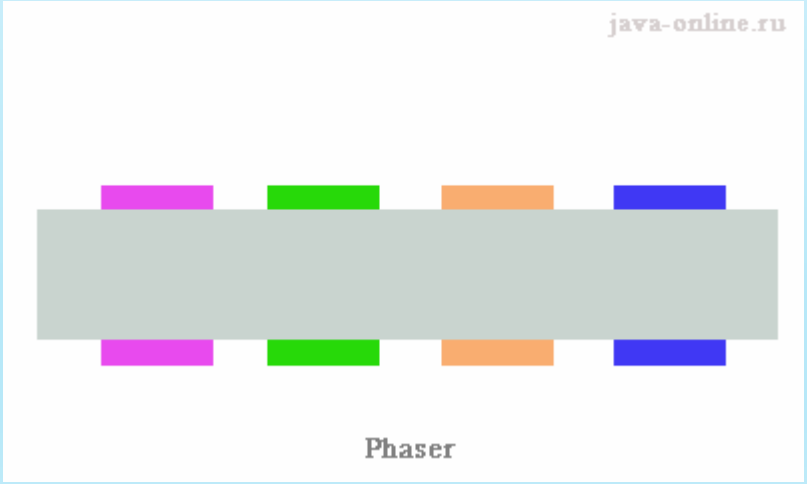
Объекты синхронизации

- **CyclicBarrier** - типа «барьер», блокирующий выполнение определенного кода для заданного количества потоков с опциональным последующим выполнением заданного потока:
  - `CyclicBarrier(int count)`
  - `CyclicBarrier(int count, Runnable class)`
  - `void await()` - самоблокировка
  - `boolean await(long wait, TimeUnit unit)` – самоблокировка по времени



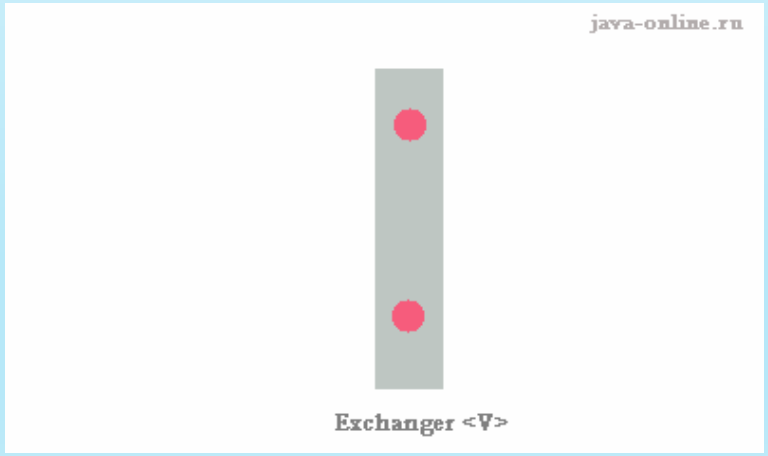
Объекты синхронизации

- **Phaser** - типа «барьер», но в отличие от CyclicBarrier, предоставляет больше гибкости
  - Phaser может иметь несколько фаз (барьеров).
  - Каждая фаза (цикл синхронизации) имеет свой номер.
  - Количество участников-потоков для каждой фазы жестко не задано и может меняться. Исполнительный поток может регистрироваться в качестве участника и отменять свое участие.
  - Исполнительный поток не обязан ожидать, пока все остальные участники соберутся у барьера. Достаточно только сообщить о своем прибытии.



Объекты синхронизации

- **Exchanger** - позволяющий провести обмен данными между двумя потоками
  - `Exchanger<V>()`
  - `V exchange(V buffer)` – метод обмена
  - `V exchange(V buffer, long wait, TimeUnit unit)`
  - Метод exchange, вызванный в одном потоке, не завершится успешно до тех пор, пока он не будет вызван из второго потока исполнения



Виды блокировок при синхронизации

- Пессимистические блокировки (syncronized) - эксклюзивный доступ к данным. Когда один поток получил пессимистическую блокировку на данные, другие потоки не могут читать и изменять эти данные, пока поток не снимет блокировку.
  - - deadlock
- Оптимистические блокировки (CAS) – копирование потоком значения общих данных в свою память, модификация их и попытка записи через проверку соответствия версий (изменились ли общие данные за время модификации).

Atomic-классы

- 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;
    }
}
```

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, ссылки

- 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)