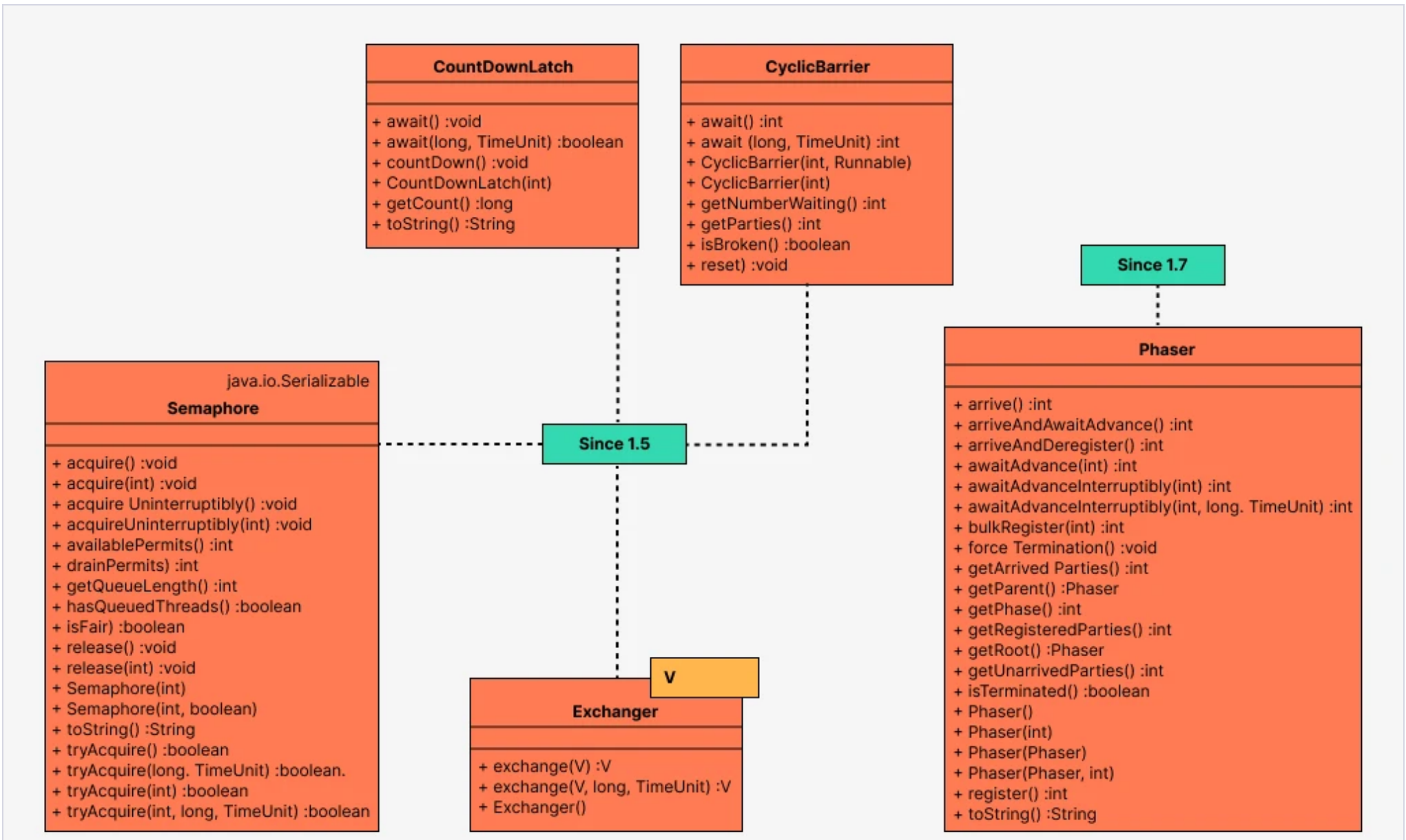
Synchronizers: синхронизация доступа к ресурсам в Java

JSP & Servlets
19 уровень, 4 лекция

ОТКРЫТА

Semaphore



Семафоры, как правило, используются, когда надо ограничить количество потоков при работе с файловой системой. Доступ к файлу или другому общему ресурсу управляется через счетчик. Если его значение больше нуля, доступ разрешен, но в тот же момент времени показания счетчика будут уменьшаться.

В тот момент, когда счетчик вернет ноль, текущий поток будет заблокирован до момента освобождения ресурса другим потоком. Параметр количества разрешений необходимо задавать через конструктор.

Подбирать этот параметр нужно индивидуально, в зависимости от мощности твоего компьютера или ноутбука.

```
1  public class Main {
2
3     public static void main(String[] args) {
4         Semaphore sem = new Semaphore(1);
5         CommonResource res = new CommonResource();
6         new Thread(new MyThread(res, sem, "MyThread_1")).start();
7         new Thread(new MyThread(res, sem, "MyThread_2")).start();
8         new Thread(new MyThread(res, sem, "MyThread_3")).start();
9     }
10 }
11
12 class CommonResource {
```

```
    int value = 0;
}

class MyThread implements Runnable {
    CommonResource commonResource;
    Semaphore semaphore;
    String name;
    MyThread(CommonResource commonResource, Semaphore sem, String name) {
        this.commonResource = commonResource;
        this.semaphore = sem;
        this.name = name;
    }

    public void run() {

        try {
            System.out.println(name + " ожидает разрешение");
            semaphore.acquire();
            commonResource.value = 1;
            for (int i = 1; i < 7; i++) {
                System.out.println(this.name + ": " + commonResource.value);
                commonResource.value++;
                Thread.sleep(150);
            }
        } catch (InterruptedException e) {
            System.out.println(e.getMessage() + " " + name);
            Thread.currentThread().interrupt();
        }
        System.out.println(name + " освобождает разрешение");
        semaphore.release();
    }
}
```

CountDownLatch и другие

`CountDownLatch` — позволяет нескольким потокам ожидать, пока не завершится определенное количество операций, выполняемых в других потоках. В качестве примера можно представить установку приложения: она не начнется, пока ты не примешь правила пользования, пока не выберешь папку, куда устанавливать новую программу и так далее. Для этого есть специальный метод `countDown()` — этот метод уменьшает счетчик count down на единицу.

Как только счетчик становится равным нулю, все ожидающие потоки в await продолжат свою работу, а все последующие вызовы await будут проходить без ожиданий. Счетчик count down одноразовый и не может быть сброшен в первоначальное состояние.

`CyclicBarrier` — используется для синхронизации заданного количества потоков в одной точке. Барьер достигается в тот момент времени, когда N-потоков вызовут метод await(...) и блокируются. После чего счетчик сбрасывается в исходное значение, а ожидающие потоки будут освобождены. Дополнительно, если нужно, существует возможность запуска специального кода до разблокировки потоков и сброса счетчика. Для этого через конструктор передается объект с реализацией интерфейса ***Runnable***.

`Exchanger<V>` — класс `Exchanger` предназначен для обмена данными между потоками. Он является типизированным и типизирует тип данных, которыми потоки должны обмениваться.

Обмен данными производится с помощью единственного метода этого класса `exchange()`:

```
V exchange(V x) throws InterruptedException
V exchange(V x, long timeout, TimeUnit unit) throws InterruptedException, TimeoutException
```

Параметр x представляет буфер данных для обмена. Вторая форма метода также определяет параметр timeout — время ожидания и unit — тип временных единиц, применяемых для параметра timeout.

Класс `Phaser` позволяет синхронизировать потоки, представляющие отдельную фазу или стадию выполнения общего действия. `Phaser` определяет объект синхронизации, который ждет, пока не завершится определенная фаза. Затем `Phaser` переходит к следующей стадии или фазе и снова ожидает ее завершения.

При работе с классом `Phaser` обычно сначала создается его объект. Далее нам надо зарегистрировать всех участников. Для регистрации для каждого участника вызывается метод `register()`, либо можно обойтись и без этого метода, передав нужное количество участников в конструктор `Phaser`.

Затем каждый участник выполняет некоторый набор действий, составляющих фазу. А синхронизатор `Phaser` ждет, пока все участники не завершат выполнение фазы. Чтобы сообщить синхронизатору, что фаза завершена, участник должен вызвать метод `arrive()` или `arriveAndAwaitAdvance()`. После этого синхронизатор переходит к следующей фазе.

‹ Предыдущая лекция

Следующая лекция ›