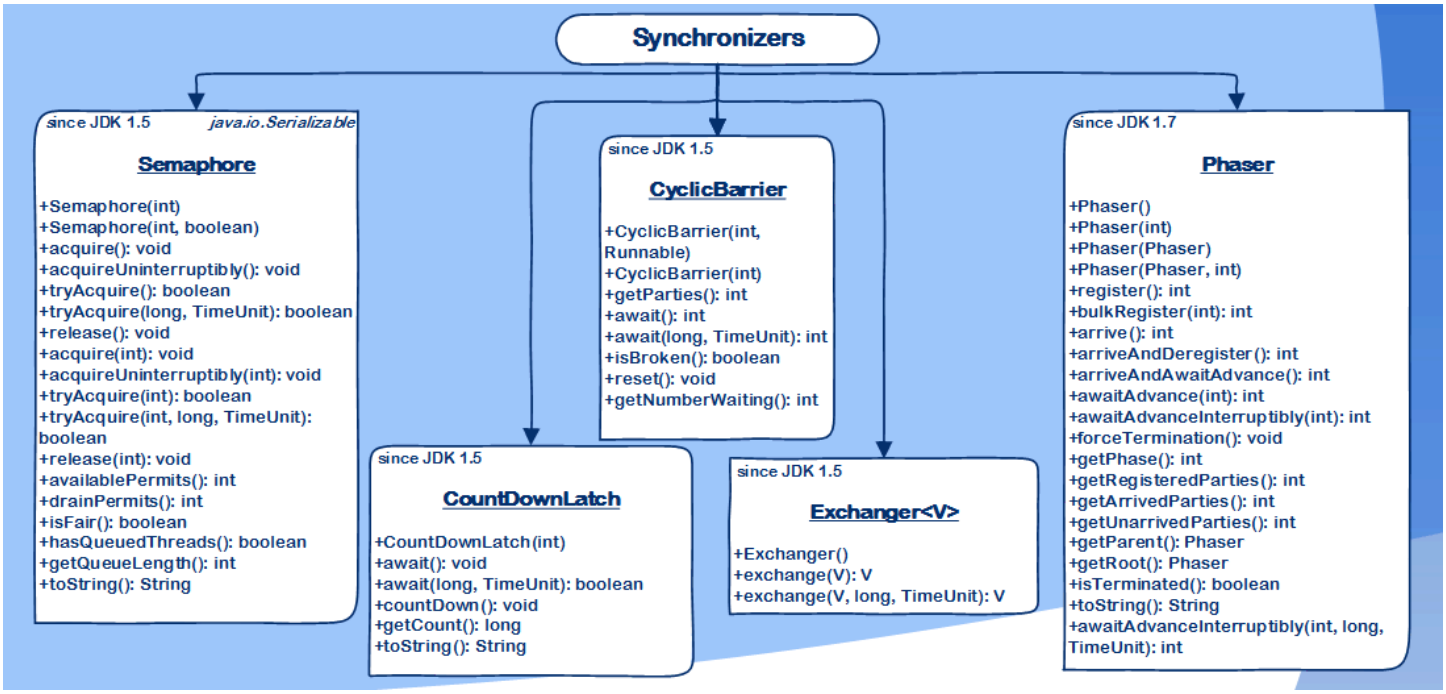
В java.util.concurrent много различных классов, которые по функционалу можно поделить на группы: Concurrent Collections, Executors, Atomics и т.д. Одной из этих групп будет Synchronizers (синхронизаторы).

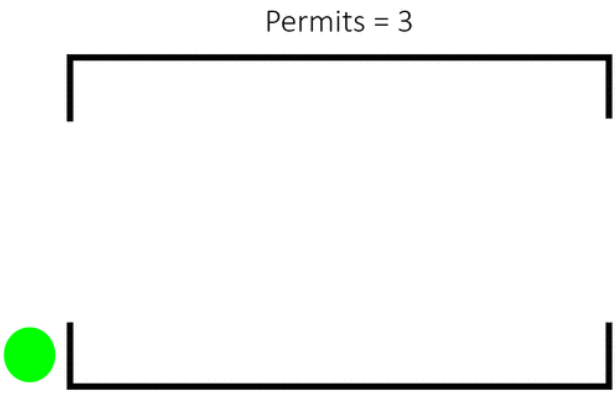


Синхронизаторы – вспомогательные утилиты для синхронизации потоков, которые дают возможность разработчику регулировать и/или ограничивать работу потоков и предоставляют более высокий уровень абстракции, чем основные примитивы языка (мониторы).

Semaphore

Синхронизатор Semaphore реализует шаблон синхронизации Семафор. Чаще всего, семафоры необходимы, когда нужно ограничить доступ к некоторому общему ресурсу. В конструктор этого класса (Semaphore(int permits) или Semaphore(int permits, boolean fair)) обязательно передается количество потоков, которому семафор будет разрешать одновременно использовать заданный ресурс.

Semaphore



Доступ управляется с помощью счётчика: изначально значение счётчика равно int permits, когда поток заходит в заданный блок кода, то значение счётчика уменьшается на единицу, когда поток его покидает, то увеличивается. Если значение счётчика равно нулю, то текущий поток блокируется, пока кто-нибудь не выйдет из блока (в качестве примера из жизни с permits = 1, можно привести очередь в кабинет в поликлинике: когда пациент покидает кабинет, мигает лампа, и заходит следующий пациент).

Пример использования Semaphore:

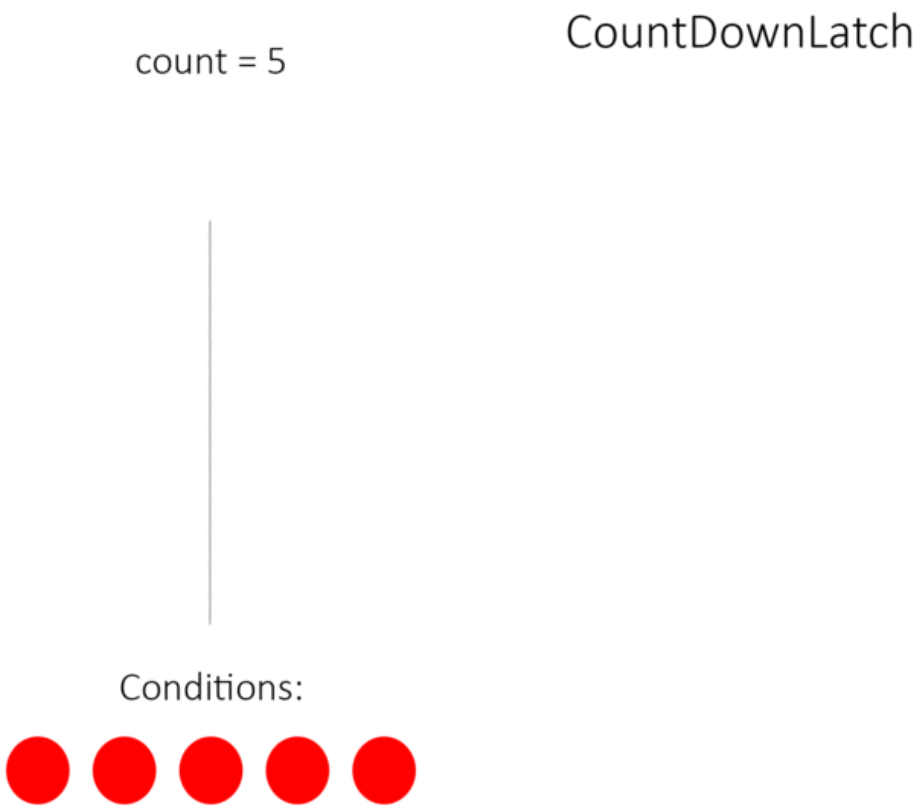
Существует парковка, которая одновременно может вмещать не более 5 автомобилей. Если парковка заполнена полностью, то вновь прибывший автомобиль должен подождать пока не освободится хотя бы одно место. После этого он сможет припарковаться.

```
import java.util.concurrent.Semaphore;

public class Parking {
    //Парковочное место занято - true, свободно - false
    private static final boolean[] PARKING_PLACES = new boolean[5];
    //Устанавливаем флаг "справедливый", в таком случае метод
    //acquire() будет раздавать разрешения в порядке очереди
    private static final Semaphore SEMAPHORE = new Semaphore(5, true);

    public static void main(String[] args) throws InterruptedException {
        for (int i = 1; i <= 7; i++) {
            new Thread(new Car(i)).start();
            Thread.sleep(400);
        }
    }

    public static class Car implements Runnable {
        private int carNumber;

        public Car(int carNumber) {
            this.carNumber = carNumber;
        }
```

```
        @Override
        public void run() {
            System.out.printf("Автомобиль №%d подъехал к парковке.\n", carNumber);
            try {
                //acquire() запрашивает доступ к следующему за вызовом этого метода блоку
кода,
                //если доступ не разрешен, поток вызвавший этот метод блокируется до тех
пор,
                //пока семафор не разрешит доступ
                SEMAPHORE.acquire();

                int parkingNumber = -1;

                //Ищем свободное место и паркуемся
                synchronized (PARKING_PLACES){
                    for (int i = 0; i < 5; i++)
                        if (!PARKING_PLACES[i]) {       //Если место свободно
                            PARKING_PLACES[i] = true;  //занимаем его
                            parkingNumber = i;          //Наличие свободного места,
гарантирует семафор
                            System.out.printf("Автомобиль №%d припарковался на месте
%d.\n", carNumber, i);
                            break;
                        }
                }

                Thread.sleep(5000);       //Уходим за покупками, к примеру

                synchronized (PARKING_PLACES) {
                    PARKING_PLACES[parkingNumber] = false;//Освобождаем место
                }

                //release(), напротив, освобождает ресурс
                SEMAPHORE.release();
                System.out.printf("Автомобиль №%d покинул парковку.\n", carNumber);
            } catch (InterruptedException e) {
            }
        }
    }
}
```

CountDownLatch

CountDownLatch (замок с обратным отсчетом) предоставляет возможность любому количеству потоков в блоке кода ожидать до тех пор, пока не завершится определенное количество операций, выполняющихся в других потоках, перед тем как они будут «отпущены», чтобы продолжить свою деятельность. В конструктор CountDownLatch (CountDownLatch(int count)) обязательно передается количество операций, которое должно быть выполнено, чтобы замок «отпустил» заблокированные потоки.



Блокировка потоков снимается с помощью счётчика: любой действующий поток, при выполнении определенной операции уменьшает значение счётчика. Когда счётчик достигает 0, все ожидающие потоки разблокируются и продолжают выполняться (примером CountDownLatch из жизни может служить сбор экскурсионной группы: пока не наберется определенное количество человек, экскурсия не начнется).

Пример использования CountDownLatch:

Мы хотим провести автомобильную гонку. В гонке принимают участие пять автомобилей. Для начала гонки нужно, чтобы выполнились следующие условия:

- Каждый из пяти автомобилей подъехал к стартовой прямой;

- Была дана команда «На старт!»;

- Была дана команда «Внимание!»;

- Была дана команда «Марш!».

Важно, чтобы все автомобили стартовали одновременно.

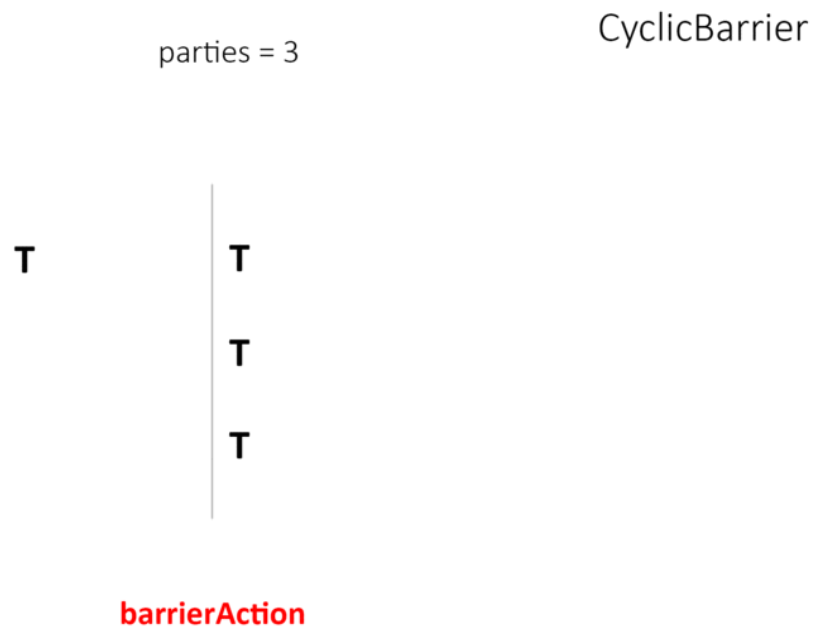
```
import java.util.concurrent.CountDownLatch;

public class Race {
    //Создаем CountDownLatch на 8 "условий"
    private static final CountDownLatch START = new CountDownLatch(8);
    //Условная длина гоночной трассы
    private static final int trackLength = 500000;

    public static void main(String[] args) throws InterruptedException {
        for (int i = 1; i <= 5; i++) {
            new Thread(new Car(i, (int) (Math.random() * 100 + 50))).start();
```

```
            Thread.sleep(1000);
        }

        while (START.getCount() > 3) //Проверяем, собрались ли все автомобили
            Thread.sleep(100);              //у стартовой прямой. Если нет, ждем 100ms

        Thread.sleep(1000);
        System.out.println("На старт!");
        START.countDown();//Команда дана, уменьшаем счетчик на 1
        Thread.sleep(1000);
        System.out.println("Внимание!");
        START.countDown();//Команда дана, уменьшаем счетчик на 1
        Thread.sleep(1000);
        System.out.println("Марш!");
        START.countDown();//Команда дана, уменьшаем счетчик на 1
        //счетчик становится равным нулю, и все ожидающие потоки
        //одновременно разблокируются
    }

    public static class Car implements Runnable {
        private int carNumber;
        private int carSpeed;//считаем, что скорость автомобиля постоянная

        public Car(int carNumber, int carSpeed) {
            this.carNumber = carNumber;
            this.carSpeed = carSpeed;
        }

        @Override
        public void run() {
            try {
                System.out.printf("Автомобиль №%d подъехал к стартовой прямой.\n",
carNumber);
                //Автомобиль подъехал к стартовой прямой - условие выполнено
                //уменьшаем счетчик на 1
                START.countDown();
                //метод await() блокирует поток, вызвавший его, до тех пор, пока
                //счетчик CountDownLatch не станет равен 0
                START.await();
                Thread.sleep(trackLength / carSpeed);//ждем пока проедет трассу
                System.out.printf("Автомобиль №%d финишировал!\n", carNumber);
            } catch (InterruptedException e) {
            }
        }
    }
}
```

CountDownLatch может быть использован в самых разных схемах синхронизации: к примеру, чтобы пока один поток выполняет работу, заставить другие потоки ждать или, наоборот, чтобы заставить поток ждать других, чтобы выполнить работу.

CyclicBarrier

CyclicBarrier реализует шаблон синхронизации Барьер. Циклический барьер является точкой синхронизации, в которой указанное количество параллельных потоков встречается и блокируется. Как только все потоки прибыли, выполняется опционное действие (или не выполняется, если барьер был инициализирован без него), и, после того, как оно выполнено, барьер ломается и ожидающие потоки «освобождаются». В конструктор барьера (CyclicBarrier(int parties) и CyclicBarrier(int parties, Runnable barrierAction)) обязательно передается количество сторон, которые должны «встретиться», и, опционально, действие, которое должно произойти, когда стороны встретились, но перед тем когда они будут «отпущены».



Барьер похож на CountDownLatch, но главное различие между ними в том, что вы не можете заново использовать «замок» после того, как его счётчик достигнет нуля, а барьер вы можете использовать снова, даже после того, как он сломается. CyclicBarrier является альтернативой метода join(), который «собирает» потоки только после того, как они выполнились.

Пример использования CyclicBarrier:

Существует паромная переправа. Паром может переправлять одновременно по три автомобиля. Чтобы не гонять паром лишний раз, нужно отправлять его, когда у переправы соберется минимум три автомобиля.

```
import java.util.concurrent.CyclicBarrier;

public class Ferry {
    private static final CyclicBarrier BARRIER = new CyclicBarrier(3, new FerryBoat());
    //Инициализируем барьер на три потока и таском, который будет выполняться, когда
    //у барьера соберется три потока. После этого, они будут освобождены.

    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < 9; i++) {
            new Thread(new Car(i)).start();
            Thread.sleep(400);
        }
    }

    //Таск, который будет выполняться при достижении сторонами барьера
    public static class FerryBoat implements Runnable {
```

```
        @Override
        public void run() {
            try {
                Thread.sleep(500);
                System.out.println("Паром переправил автомобили!");
            } catch (InterruptedException e) {
            }
        }
    }

    //Стороны, которые будут достигать барьера
    public static class Car implements Runnable {
        private int carNumber;

        public Car(int carNumber) {
            this.carNumber = carNumber;
        }

        @Override
        public void run() {
            try {
                System.out.printf("Автомобиль №%d подъехал к паромной переправе.\n",
carNumber);
                //Для указания потоку о том что он достиг барьера, нужно вызвать метод
await()
                //После этого данный поток блокируется, и ждет пока остальные стороны
достигнут барьера
                BARRIER.await();
                System.out.printf("Автомобиль №%d продолжил движение.\n", carNumber);
            } catch (Exception e) {
            }
        }
    }
}
```

Exchanger<V>

Exchanger (обменник) может понадобиться, для того, чтобы обменяться данными между двумя потоками в определенной точки работы обоих потоков. Обменник — обобщенный класс, он параметризируется типом объекта для передачи. Exchanger (обменник) может понадобиться, для того, чтобы обменяться данными между двумя потоками в определенной точки работы обоих потоков. Обменник — обобщенный класс, он параметризируется типом объекта для передачи.

Exchanger<V>

Обменник является точкой синхронизации пары потоков: поток, вызывающий у обменника метод exchange() блокируется и ждет другой поток. Когда другой поток вызовет тот же метод, произойдет обмен объектами: каждая из них получит аргумент другой в методе exchange(). Стоит отметить, что обменник поддерживает передачу null значения. Это дает возможность использовать его для передачи объекта в одну сторону, или, просто как точку синхронизации двух потоков.

Пример использования Exchanger<V>:

Есть два грузовика: один едет из пункта A в пункт D, другой из пункта B в пункт C. Дороги AD и BC пересекаются в пункте E. Из пунктов A и B нужно доставить посылки в пункты C и D. Для этого грузовики в пункте E должны встретиться и обменяться соответствующими посылками.

```
import java.util.concurrent.Exchanger;

public class Delivery {
    //Создаем обменник, который будет обмениваться типом String
    private static final Exchanger<String> EXCHANGER = new Exchanger<>();

    public static void main(String[] args) throws InterruptedException {
        String[] p1 = new String[]{"{посылка A->D}", "{посылка A->C}"};//Формируем груз
для 1-го грузовика
        String[] p2 = new String[]{"{посылка B->C}", "{посылка B->D}"};//Формируем груз
для 2-го грузовика
        new Thread(new Truck(1, "A", "D", p1)).start();//Отправляем 1-й грузовик из A в D
        Thread.sleep(100);
        new Thread(new Truck(2, "B", "C", p2)).start();//Отправляем 2-й грузовик из B в C
    }

    public static class Truck implements Runnable {
        private int number;
```

```
        private String dep;
        private String dest;
        private String[] parcels;

        public Truck(int number, String departure, String destination, String[] parcels)
{
            this.number = number;
            this.dep = departure;
            this.dest = destination;
            this.parcels = parcels;
        }

        @Override
        public void run() {
            try {
                System.out.printf("В грузовик №%d погрузили: %s и %s.\n", number,
parcels[0], parcels[1]);
                System.out.printf("Грузовик №%d выехал из пункта %s в пункт %s.\n",
number, dep, dest);
                Thread.sleep(1000 + (long) Math.random() * 5000);
                System.out.printf("Грузовик №%d приехал в пункт Е.\n", number);
                parcels[1] = EXCHANGER.exchange(parcels[1]);//При вызове exchange() поток
блокируется и ждет
                //пока другой поток вызовет exchange(), после этого произойдет обмен
посылками
                System.out.printf("В грузовик №%d переместили посылку для пункта %s.\n",
number, dest);
                Thread.sleep(1000 + (long) Math.random() * 5000);
                System.out.printf("Грузовик №%d приехал в %s и доставил: %s и %s.\n",
number, dest, parcels[0], parcels[1]);
            } catch (InterruptedException e) {
            }
        }
    }
}
```

Phaser

Phaser (фазер), как и CyclicBarrier, является реализацией шаблона синхронизации Барьер, но, в отличии от CyclicBarrier, предоставляет больше гибкости. Этот класс позволяет синхронизировать потоки, представляющие отдельную фазу или стадию выполнения общего действия. Как и CyclicBarrier, Phaser является точкой синхронизации, в которой встречаются потоки-участники. Когда все стороны прибыли, Phaser переходит к следующей фазе и снова ожидает ее завершения.

Если сравнить Phaser и CyclicBarrier, то можно выделить следующие важные особенности Phaser:

- Каждая фаза (цикл синхронизации) имеет номер;
- Количество сторон-участников жестко не задано и может меняться: поток может регистрироваться в качестве участника и отменять свое участие;
- Участник не обязан ожидать, пока все остальные участники соберутся на барьере. Чтобы продолжить свою работу достаточно сообщить о своем прибытии;
- Случайные свидетели могут следить за активностью в барьере;

- Поток может и не быть стороной-участником барьера, чтобы ожидать его преодоления;
- У фазера нет опционального действия.

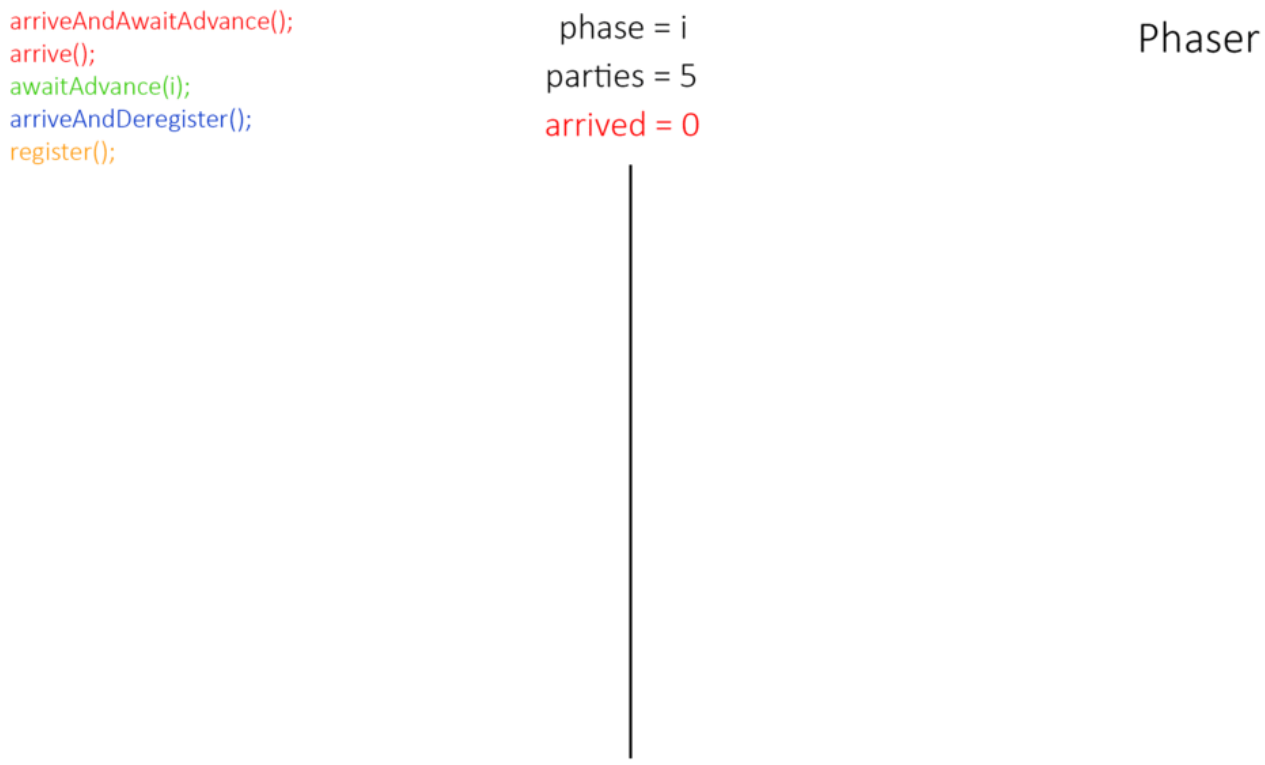
Объект Phaser создается с помощью одного из конструкторов:

```
Phaser()
Phaser(int parties)
```

Параметр parties указывает на количество сторон-участников, которые будут выполнять фазы действия. Первый конструктор создает объект Phaser без каких-либо сторон, при этом барьер в этом случае тоже «закрыт». Второй конструктор регистрирует передаваемое в конструктор количество сторон. Барьер открывается когда все стороны прибыли, или, если снимается последний участник.

Основные методы:

- **int register()** — регистрирует нового участника, который выполняет фазы. Возвращает номер текущей фазы;
- **int getPhase()** — возвращает номер текущей фазы;
- **int arriveAndAwaitAdvance()** — указывает что поток завершил выполнение фазы. Поток приостанавливается до момента, пока все остальные стороны не закончат выполнять данную фазу. Точный аналог CyclicBarrier.await(). Возвращает номер текущей фазы;
- **int arrive()** — сообщает, что сторона завершила фазу, и возвращает номер фазы. При вызове данного метода поток не приостанавливается, а продолжает выполнятся;
- **int arriveAndDeregister()** — сообщает о завершении всех фаз стороной и снимает ее с регистрации. Возвращает номер текущей фазы;
- **int awaitAdvance(int phase)** — если phase равно номеру текущей фазы, приостанавливает вызвавший его поток до её окончания. В противном случае сразу возвращает аргумент.



Пример использования Phaser:

Есть пять остановок. На первых четырех из них могут стоять пассажиры и ждать автобуса. Автобус выезжает из парка и останавливается на каждой остановке на некоторое время. После конечной остановки автобус едет в парк. Нам нужно забрать пассажиров и высадить их на нужных остановках.

```
import java.util.ArrayList;
import java.util.concurrent.Phaser;

public class Bus {
    private static final Phaser PHASER = new Phaser(1);//Сразу регистрируем главный поток
    //Фазы 0 и 6 - это автобусный парк, 1 - 5 остановки

    public static void main(String[] args) throws InterruptedException {
        ArrayList<Passenger> passengers = new ArrayList<>();

        for (int i = 1; i < 5; i++) {           //Сгенерируем пассажиров на остановках
            if ((int) (Math.random() * 2) > 0)
                passengers.add(new Passenger(i, i + 1));//Этот пассажир выходит на
следующей

            if ((int) (Math.random() * 2) > 0)
                passengers.add(new Passenger(i, 5));    //Этот пассажир выходит на
конечной
        }

        for (int i = 0; i < 7; i++) {
            switch (i) {
                case 0:
                    System.out.println("Автобус выехал из парка.");
                    PHASER.arrive();//В фазе 0 всего 1 участник - автобус
                    break;
                case 6:
                    System.out.println("Автобус уехал в парк.");
                    PHASER.arriveAndDeregister();//Снимаем главный поток, ломаем барьер
                    break;
                default:
                    int currentBusStop = PHASER.getPhase();
                    System.out.println("Остановка № " + currentBusStop);

                    for (Passenger p : passengers)          //Проверяем, есть ли
пассажиры на остановке
                        if (p.departure == currentBusStop) {
                            PHASER.register();//Регистрируем поток, который будет
участвовать в фазах
                            p.start();        // и запускаем
                        }

                    PHASER.arriveAndAwaitAdvance();//Сообщаем о своей готовности
            }
        }
    }

    public static class Passenger extends Thread {
        private int departure;
        private int destination;
```

```
        public Passenger(int departure, int destination) {
            this.departure = departure;
            this.destination = destination;
            System.out.println(this + " ждёт на остановке № " + this.departure);
        }

        @Override
        public void run() {
            try {
                System.out.println(this + " сел в автобус.");

                while (PHASER.getPhase() < destination) //Пока автобус не приедет на
нужную остановку(фазу)
                    PHASER.arriveAndAwaitAdvance();     //заявляем в каждой фазе о
готовности и ждем

                Thread.sleep(1);
                System.out.println(this + " покинул автобус.");
                PHASER.arriveAndDeregister();   //Отменяем регистрацию на нужной фазе
            } catch (InterruptedException e) {
            }
        }

        @Override
        public String toString() {
            return "Пассажир{" + departure + " -> " + destination + '}';
        }
    }
}
```