Класс Phaser, примеры реализации кода в Java

В версии JDК 7 внедрен новый **класс синхронизации под названием Phaser**. Главное его назначение - синхронизировать потоки исполнения, которые выполняют одну или несколько стадий (или фаз) каких то действий *(может быть, что все потоки выполняют одно и тоже действие на одном шаге, а может быть,что разные действия, используя переопределение метода onAdvance())*. Например, в прикладной программе может быть несколько потоков исполнения, реализующих три стадии обработки заказов.

На первой стадии отдельные потоки исполне­ния используются для того, чтобы проверить сведения о клиенте, наличие товара на складе и его цену. По завершении этой стадии остаются два потока исполнения, где на второй стадии вычисляется стоимость доставки и сумма соответствующего налога, а на заключительной стадии подтверждается оплата и определяется ори­ентировочное время доставки.

В прошлом для синхронизации нескольких пото­ков исполнения в такой прикладной программе пришлось бы немало потрудиться. А с появлением класса Phaser этот процесс **значительно упростился**.

Прежде всего следует иметь в виду, что класс Phaser действует подобно описан­ному ранее [**классу CyclicBarrier**](https://pro-java.ru/parallelizm-v-java/klass-cyclicbarrier-primery-realizacii-koda-v-java/), за исключением того, что он поддерживает не­сколько фаз.

В итоге класс Phaser позволяет определить объект синхронизации, ожидающий завершения определенной фазы. Затем он переходит к следующей фазе и снова ожидает ее завершения. Следует также иметь в виду, что **класс Phaser** можно использовать и для синхронизации только одной фазы. В этом отношении он действует подобно классу CyclicBarrier, хотя главное его назначение - син­хронизация нескольких фаз. В классе Phaser определяются четыре конструктора.

В данной статье упоминаются следующие **два конструктора этого класса:**

|  |  |
| --- | --- |
| 1  2 | Phaser()  Phaset(int\_количество сторон) |

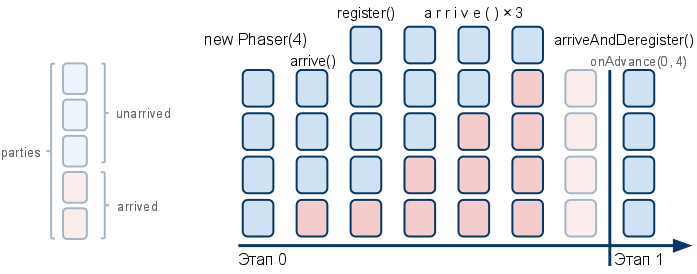
Первый конструктор создает синхронизатор фаз с нулевым регистрационным счетом, а второй устанавливает значение регистрационного счета равным задан­ному **количеству\_сторон**. Объекты, регистрируемые синхронизатором фаз, за­частую обозначаются термином **сторона**.

Обычно имеется полное соответствие количества регистрируемых объектов и синхронизируемых потоков исполне­ния, хотя этого и не требуется. В обоих случаях текущая фаза является нулевой. Поэтому когда создается экземпляр класса Phaser, он первоначально **находится в нулевой фазе**.

Обычно класс Phaser используется следующим образом. Сначала создается новый экземпляр класса Phaser. Затем синхронизатор фаз регистрирует одну или несколько сторон, вызывая метод register() или указывая нужное количе­ство сторон в конструкторе класса Phaser.

Синхронизатор фаз ожидает до тех пор, пока все зарегистрированные стороны не завершат фазу. Сторона извещает об этом, вызывая один из многих методов, предоставляемых классом Phaser, на­пример метод **arrive()** или **arriveAndAwaitAdvance()**.

Как только все стороны достигнут данной фазы, она считается завершенной, и синхронизатор фаз может перейти к следующей фазе (если она имеется) или завершить свою работу. Далее этот процесс поясняется более подробно.

[](http://pro-java.ru/wp-content/uploads/2017/06/klass-phaser-primery-realizacii-koda-v-java-pro-java-ru-2.png)

Для регистрации стороны после создания объекта класса Phaser следует **вы­звать метод register()**. Ниже приведена общая форма этого метода. В итоге он возвратит номер регистрируемой фазы.

|  |  |
| --- | --- |
|  | int register() |

Чтобы сообщить о завершении фазы, сторона должна вызвать метод arrive( ) или какой-нибудь его вариант. Когда количество достижений конца фазы сравня­ется с количеством зарегистрированных сторон, фаза завершится и объект класса Phaser перейдет к следующей фазе (если она имеется). **Метод arrive()** имеет следующую общую форму:

|  |  |
| --- | --- |
|  | int arrive() |

Этот метод сообщает, что сторона (обычно поток исполнения) завершила не­которую задачу (или ее часть) . Он возвращает текущий номер фазы. Если работа синхронизатора фаз завершена, этот метод возвращает отрицательное значение.

Метод arrive() не приостанавливает исполнение вызывающего потока. Это означает, что он не ожидает завершения фазы. Этот метод должен быть вызван толь­ко зарегистрированной стороной.

Если требуется **указать завершение фазы**, а затем ожидать завершения этой фазы всеми остальными зарегистрированными сторонами, следует вызвать метод arriveAndAwaitAdvance(). Ниже приведена общая форма этого метода.

|  |  |
| --- | --- |
|  | int arriveAndAwaitAdvance() |

Этот метод ожидает до тех пор, пока все стороны не достигнут данной фазы, а затем возвращает номер следующей фазы или отрицательное значение, если синхронизатор фаз завершил свою работу. **Метод ariveAndAwaitAdvance()** дол­жен быть вызван только зарегистрированной стороной.

Поток исполнения может достигнуть данной фазы, а затем сняться с регистрации, вызвав метод arriveAndDeregister(). Ниже приведена общая форма этого метода.

|  |  |
| --- | --- |
|  | int arriveAndDeregister() |

Этот метод возвращает номер текущей фазы или отрицательное значение,если синхронизатор фаз завершил свою работу. Он не ожидает завершения фазы. Метод arriveAndDeregister() должен быть вызван только зарегистрированной стороной.

Чтобы получить номер текущей фазы, следует вызвать метод getPhase(). Его общая форма выглядит следующим образом:

|  |  |
| --- | --- |
|  | final int getPhase() |

Когда создается объект класса Phaser, первая фаза получает нулевой номер, вторая фаза - номер 1, третья фаза - номер 2 и т.д. Если вызывающий объект класса Phaser завершил свою работу, возвращается отрицательное значение.

В приведенном ниже примере программы демонстрируется применение класса Phaser. В этой программе создаются три потока, каждый из которых имеет три фазы своего исполнения. Для синхронизации каждой фазы применяется класс Phaser.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84 | **// Пример применения класса Phaser**    import java.util.concurrent.\*;    class PhaserDemo {        public static void main(String args[]) {       Phaser phsr = new Phaser(1);  int curPhase;    System.out.println("Запуск потоков");    new MyThread(phsr, "A");  new MyThread(phsr, "B");  new MyThread(phsr, "C");    // ожидать заверешния всеми потоками исполнения первой фазы  curPhase = phsr.getPhase();  phsr.arriveAndAwaitAdvance();  System.out.println("Фаза " + curPhase + " завершена");    // ожидать завершения всеми потоками исполнения второй фазы  curPhase = phsr.getPhase();  phsr.arriveAndAwaitAdvance();  System.out.println("Фаза " + curPhase + " завершена");    curPhase = phsr.getPhase();  phsr.arriveAndAwaitAdvance();  System.out.println("Фаза " + curPhase + " завершена");       // снять основной поток исполнения с регистрации  phsr.arriveAndDeregister();    if(phsr.isTerminated()) {      System.out.println("Синхронизатор фаз завершен");  }      }  }      **// Поток исполнения, использующий синхронизатор фаз типа Phaser**  class MyThread implements Runnable {      Phaser phsr;      String name;        MyThread(Phaser p, String n) {           phsr = p;  name = n;  phsr.register();  new Thread(this).start();      }        public void run() {    System.out.println("Поток " + name + " начинает первую фазу");  phsr.arriveAndAwaitAdvance(); // известить о достижении фазы    // Небольшая пауза, чтобы не нарушить порядок вывода. Только для иллюстрации, но необязательно для правильного функционирования синхронизатора фаз  try {      Thread.sleep(10);  } catch(InterruptedException e) {      System.out.println(e);  }    System.out.println("Поток " + name + " начинает вторую фазу");  phsr.arriveAndAwaitAdvance(); // известить о достижении фазы    // Небольшая пауза, чтобы не нарушить порядок вывода. Только для иллюстрации, но необязательно для правильного функционирования синхронизатора фаз  try {      Thread.sleep(10);  } catch(InterruptedException e) {      System.out.println(e);  }    System.out.println("Поток " + name + " начинает третью фазу");        // известить о достижении фазы и снять потоки с регистрации          phsr.arriveAndDeregister();      }  } |

Ниже приведен результат, выводимый данной программой.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | Поток B начинает первую фазу  Поток C начинает первую фазу  Поток A начинает первую фазу  Фаза 0 завершена  Поток A начинает вторую фазу  Поток B начинает вторую фазу  Поток C начинает вторую фазу  Фаза 1 завершена  Поток C начинает третью фазу  Поток B начинает третью фазу  Поток A начинает третью фазу  Фаза 2 завершена  Синхронизатор фаз завершен |

Рассмотрим подробнее основные части данной программы. Сначала в методе main() создается объект phsr типа Phaser с начальным счетом сторон, равным 1 (что соответствует основному потоку исполнения).

Затем **создаются три объекта типа MyThread** и запускаются три соответствующих [**потока исполнения**](https://pro-java.ru/category/parallelizm-v-java/). Обратите внимание на то, что объекту типа MyThread передается ссылка на объект phsr синхронизатор фаз. Объекты типа MyThread используют этот синхронизатор фаз для синхронизации своих действий.

Затем в методе main() вызывается метод getPhase(), чтобы **получить номер текущей фазы** (который первоначально явля­ется нулевым), а после этого - метод arriveAndAwaitAdvance(). В итоге выпол­нение метода main() приостанавливается до тех пор, пока не завершится нулевая, по существу, первая фаза.

Но этого не произойдет до тех пор, пока все объекты типа MyThread не вызовут метод arriveAndAwaitAdvance(). Как только это про­изойдет, метод main() **возобновит свое выполнение**, сообщив о завершении нуле­вой фазы, и перейдет к следующей фазе. Этот процесс повторяется до завершения всех трех фаз. Затем в методе main( ) вызывается метод arriveAndDeregister() ,чтобы снять с регистрации все три объекта типа MyThread.

В итоге зарегистриро­ванных сторон больше не остается, а следовательно, перейдя к следующей фазе, синхронизатор фаз завершит свою работу.

Теперь рассмотрим объект типа MyThread. Прежде всего обратите вниманиена то, что конструктору передается ссылка на синхронизатор фаз, в котором новый поток исполнения регистрируется далее как отдельная сторона.

Таким образом, каждый новый объект типа MyThread становится стороной, зарегистрированной синхронизатором фаз, переданным конструктору этого объекта.

Обратите также внимание на то, что у каждого потока исполнения имеются три фазы. В данном примере каждая фаза состоит из метки-заполнителя, которая просто отображает имя потока исполнения и то, что он делает.

Безусловно, реальный код выполнял бы в потоке более полезные действия. Между первыми двумя фазами поток испол­нения **вызывает метод arriveAndAwaitAdvance()**. Таким образом, каждый по­ток исполнения ожидает завершения фазы всеми остальными потоками, включая и основной поток. По завершении всех потоков исполнения, включая и основной, синхронизатор фаз переходит к следующей фазе.

А по завершении третьей фазы каждый поток исполнения снимается с регистрации, вызывая метод arriveAndDeregister(). Как поясняется в комментариях к объектам типа MyThread, метод sleep() вызывается исключительно в иллюстративных целях, чтобы **предотвра­тить нарушение вывода** из-за многопоточности, хотя это и не обязательно для пра­вильного функционирования синхронизатора фаз.

Если удалить вызовы метода sleep(), то выводимый результат может выглядеть немного запутанным, но фазы все равно будут синхронизированы правильно.

Следует также иметь в виду, что в рассматриваемом здесь примере используют­ся три однотипных потока исполнения, хотя это и не обязательное требование. Каждая сторона, пользующаяся синхронизатором фаз, может быть **совершенно непохожей на остальные**, выполняя свою задачу.

Все, что происходит при переходе к следующей фазе, вполне поддается кон­тролю. Для этого следует переопределить метод onAdvance(). Этот метод вызы­вается исполняющей средой, когда синхронизатор фаз переходит от одной фазы к следующей. Ниже приведена общая форма данного метода.

|  |  |
| --- | --- |
| 1 | protected boolean onAdvance(int фаза, int количество\_сторон) |

Здесь **параметр фаза** обозначает текущий номер фазы перед его приращением, а параметр **количество\_сторон** - число зарегистрированных сторон.

Для того чтобы завершить работу синхронизатора фаз, метод onAdvance() должен возвра­тить логическое значение true. А для того чтобы продолжить работу синхрони­затора фаз, метод onAdvance() должен возвратить булевское значение false.

В версии по умолчанию метод onAdvance() возвращает логическое значение true, чтобы завершить работу синхронизатора фаз, если зарегистрированных сторон больше нет. Как правило, переопределяемая версия данного метода долж­на следовать этой практике.

**Метод onAdvance()** может быть, в частности, переопределен для того, что­бы дать синхронизатору фаз возможность выполнить заданное количество фаз, а затем остановиться.

Ниже приведен характерный тому пример. В данном при­мере создается класс MyPhaser, расширяющий класс Phaser таким образом, что­бы выполнять заданное количество фаз. С этой целью переопределяется метод onAdvance().

Конструктор класса MyPhaser принимает один аргумент, задающий количество выполняемых фаз. Обратите внимание на то, что класс MyPhaser **ав­томатически регистрирует одну сторону**. Это удобно для целей данного примера,но у конкретной прикладной программы могут быть другие потребности.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88 | // Расширить класс Phaser и переопределить метод onAdvance()  // таким образом, чтобы было выполнено только определенное  // количество фаз    import java.util.concurrent.\*;    // Расширить класс MyPhaser, чтобы выполнить только  // определенное количество фаз    class MyPhaser extends Phaser {        int numPhases;        MyPhaser(int parties, int phaseCount) {          super(parties);  numPhases = phaseCount - 1;      }        // переопределить метод onAdvance(), чтобы выполнить      // определенное количество фаз        protected boolean onAdvance(int p, int regParties) {          // Следующий оператор println() требуется только для  // целей иллюстрации. Как правило, метод onAdvance()  // не отображает выводимые данные  System.out.println("Фаза " + p + " завершена.\n");    // возратить логическое значение true,  // если все фазы завершены  if(p == numPhases || regParties == 0) return true;    // В противном случает возратить логическое значение false;  return false;      }  }    class PhaserDemo2 {        public static void main(String args[]) {    MyPhaser phsr = new MyPhaser(1, 4);  System.out.println("Запуск потоков\n");    new MyThread(phsr, "A");  new MyThread(phsr, "B");  new MyThread(phsr, "C");    // ожидать завершения определенного количества фаз  while(!phsr.isTerminated()) {      phsr.arriveAndAwaitAdvance();          }            System.out.println("Синхронизатор фаз завершен");      }  }    // Поток исполнения, использующий синхронизатор фаз типа Phaser  class MyThread implements Runnable {        Phaser phsr;      String name;        MyThread(Phaser p, String n) {          phsr = p;          name = n;          phsr.register();          new Thread(this).start();      }        public void run() {            while(!phsr.isTerminated()) {      System.out.println(               "Поток " + name + " начинает фазу " + phsr.getPhase());      phsr.arriveAndAwaitAdvance();        // небольшая пауза, чтобы не нарушить порядок вывода.      // Только для иллюстрации, но необязательно для правильного      // функционирования синхронизатора фаз        try {          Thread.sleep(10);      } catch(InterruptedException e) {  System.out.println(e);      }  }      }  } |

Ниже приведен результат, выводимый данной программой.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | Запуск потоков    Поток B начинает фазу 0  Поток A начинает фазу 0  Поток C начинает фазу 0  Фаза 0 завершена.    Поток C начинает фазу 1  Поток B начинает фазу 1  Поток A начинает фазу 1  Фаза 1 завершена.    Поток A начинает фазу 2  Поток C начинает фазу 2  Поток B начинает фазу 2  Фаза 2 завершена.    Поток B начинает фазу 3  Поток C начинает фазу 3  Поток A начинает фазу 3  Фаза 3 завершена.    Синхронизатор фаз завершен |

В методе main( ) создается один экземпляр класса Phaser. В качестве аргумен­та ему передается значение 4. Это означает, что он будет выполняться в течении четырех фаз, а затем завершится. Затем создаются три потока исполнения и на­чинается следующий цикл, как показано ниже.

|  |  |
| --- | --- |
| 1  2  3  4 | // ожидать завершения определенного количества фаз  while(!phsr.isTerminated()) {      phsr.arriveAndAwaitAdvance();  } |

В этом цикле метод arriveAndAwaitAdvance() вызывается до завершения работы синхронизатора фаз, а этого не произойдет до тех пор, пока не будет вы­полнено определенное количество фаз.

В данном случае цикл будет продолжать­ся до тех пор, пока не завершатся все четыре фазы. Следует также иметь в виду, что метод arriveAndAwaitAdvance() вызывается из потоков исполнения вплоть до завершения работы синхронизатора фаз в данном цикле. Это означает, что по­токи исполняются до завершения заданного количества фаз.

А теперь рассмотрим подробнее исходный код метода onAdvance(). Всякий раз, когда вызывается метод onAdvance(), ему передается текущая фаза и коли­чество зарегистрированных сторон. Если текущая фаза соответствует указанной фазе или количество зарегистрированных сторон равно нулю, метод onAdvance() возвращает логическое значение true, прекращая таким образом работу синхро­низатора фаз.

Это делается в приведенном ниже фрагменте кода. Как можно за­метить, для достижения желаемого результата необходимо совсем немного кода.

|  |  |
| --- | --- |
| 1  2  3 | // возратить логическое значение true,  // если все фазы завершены  if(p == numPhases || regParties == 0) return true; |

Прежде чем завершить эту статью, следует заметить, что расширять класс Phaser совсем не обязательно. Как и в предыдущем примере, для этого достаточно пере­определить метод onAdvance().

В некоторых случаях может быть создан более компактный код с помощью анонимного внутреннего класса, **переопределяющего методы onAdvance() и класс Thread**: (тут будет делаться какая-то хрень с заданным в переменной alphabet алфавитом)

import java.util.concurrent.Phaser;

public class FBIEasterEgg {

static int lines = 10;

static String alphabet = "abcdefghijklmnopqrstuvwxyz";

static StringBuffer randomAlphabet = new StringBuffer();

public static void main(String[] args) {

final Phaser phaser = new Phaser() { //*сразу при создании объекта класса Phaser переопределяется метод onAdvance()*

protected boolean onAdvance(int phase, int parties) {

System.out.println(randomAlphabet);

randomAlphabet = new StringBuffer();

return phase >= lines; // управление количеством циклов здесь

}

};

// всем придется дождаться главного потока main

phaser.register();

for (int i = 0; i < alphabet.length(); i++) {

final char next = alphabet.charAt(i);

phaser.register(); // билет на следующую тему

new Thread() {

public void run() { //*сразу при создании нитей переопределяется метод run()*

do {

randomAlphabet.append(next);

phaser.arriveAndAwaitAdvance(); // checkout

} while ( !phaser.isTerminated() );

}

}.start();

}

System.out.println("Write this by hand and carry in the pocket:");

// какие-то дополнительные приготовления могут быть сделаны здесь

phaser.arriveAndDeregister();

}

}

У класса Phaser имеются дополнительные возможности, которые моrут ока­заться полезными в прикладных программах. В частности, вызвав метод awaitAdvance(), **можно ожидать конкретной фазы**, как показано ниже.

|  |  |
| --- | --- |
| 1 | int awaitAdvance(int фаза) |

Здесь параметр фаза обозначает номер фазы, в течение которой метод awaitAdvance() находится в состоянии ожидания до тех пор, пока не произойдет пере­ход к следующей фазе.

Этот метод завершится немедленно, если передаваемый ему параметр фаза не будет равен текущей фазе. Он завершится сразу и в том слу­чае, если синхронизатор фаз завершит работу.

Но если в качестве параметра фазаэтому методу будет передана текущая фаза, то он будет ожидать перехода к сле­дующей фазе. Этот метод должен быть вызван только зарегистрированной сто­роной. Имеется также прерывающая версия этого метода под названием **awaitAdvanceInterruptibly()**.

Чтобы зарегистрировать несколько сторон, следует вызвать метод bulkRegister( ), чтобы получить количество зарегистрированных сторон - метод getRegisteredParties(), чтобы получить количество сторон, достигших или не достигших своей фазы - метод **getArrivedParties()** или **getUnarrivedParties()** соответственно, а чтобы перевести синхронизатор фаз в завершенное состояние - метод forceTermination().

Класс Phaser позволяет также создать дерево синхронизаторов фаз. Он снаб­жен двумя дополнительными конструкторами, которые позволяют указать роди­тельский синхронизатор фаз и метод getParent().