**Многопоточное программирование**

!!!Работать с потоками напрямую через создание одиночных *thread-ов* неудобно и чревато ошибками. Поэтому в 2004 году в Java 5 добавили Concurrency API. Он находится в пакете java.util.concurrent и содержит большое количество полезных классов и методов для многопоточного программирования!!!

Содержание

**ОДИНОЧНЫЕ ПОТОКИ**

**Класс Thread** *(Java 1.0)*

**Недостатки при использовании потоков**

**Создание и выполнение потоков**

Наследование от класса Thread

Ожидание завершения потока

Реализация интерфейсов **Runnable** *(Java 1.5)*, **Callable** *(Java 1.8)*

Завершение и прерывание потока

- Завершение потока

- Метод interrupt

**Синхронизация потоков. Оператор synchronized**

**Методы wait и notify**

*Дальше идут три класса из пакета java.util.concurrent.\**

**Семафоры**  java.util.concurrent.Semaphore - Для управления доступом к ресурсу семафор использует счетчик, представляющий количество разрешений. Если значение счетчика больше нуля, то поток получает доступ к ресурсу, при этом счетчик уменьшается на единицу. После окончания работы с ресурсом поток освобождает семафор, и счетчик увеличивается на единицу. Если же счетчик равен нулю, то поток блокируется и ждет, пока не получит разрешение от семафора.

**Обмен между потоками. Класс Exchanger** java.util.concurrent.Exchanger - Для обмена данными между потоками. Первый поток, который вызывает метод exchange(...) заблокируется до тех пор, пока тот же метод не вызовет второй поток. Как только это произойдет, потоки обменяются значениями и продолжат свою работу.

**Класс Phaser** java.util.concurrent.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.

Большинство языков программирования поддерживают такую важную функциональность как многопоточность, и Java в этом плане не исключение. При помощи многопоточности мы можем выделить в приложении несколько потоков, которые будут выполнять различные задачи одновременно. Если у нас, допустим, графическое приложение, которое посылает запрос к какому-нибудь серверу или считывает и обрабатывает огромный файл, то без многопоточности у нас бы блокировался графический интерфейс на время выполнения задачи. А благодаря потокам мы можем выделить отправку запроса или любую другую задачу, которая может долго обрабатываться, в отдельный поток. Поэтому большинство реальных приложений, которые многим из нас приходится использовать, практически не мыслимы без многопоточности.

**Класс Thread**

В Java функциональность отдельного потока заключается в классе **Thread**. И чтобы создать новый поток, нам надо создать объект этого класса. Но все потоки не создаются сами по себе. Когда запускается программа, начинает работать главный поток этой программы. От этого главного потока порождаются все остальные дочерние потоки.

С помощью статического метода **Thread.currentThread()** мы можем получить текущий поток выполнения:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public static void main(String[] args) {        Thread t = Thread.currentThread(); // получаем главный поток      System.out.println(t.getName()); // main  } |

По умолчанию именем главного потока будет main.

Для управления потоком класс Thread предоставляет еще ряд методов. Наиболее используемые из них:

* **getName()**: возвращает имя потока
* **setName(String name)**: устанавливает имя потока
* **getPriority()**: возвращает приоритет потока
* **setPriority(int proirity)**: устанавливает приоритет потока. Приоритет является одним из ключевых факторов для выбора системой потока из кучи потоков для выполнения. В этот метод в качестве параметра передается числовое значение приоритета - от 1 до 10. По умолчанию главному потоку выставляется средний приоритет - 5.
* **isAlive()**: возвращает true, если поток активен
* **isInterrupted()**: возвращает true, если поток был прерван
* **join()**: ожидает завершение потока
* **run()**: определяет точку входа в поток
* **sleep()**: приостанавливает поток на заданное количество миллисекунд
* **start()**: запускает поток, вызывая его метод run()

Мы можем вывести всю информацию о потоке:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public static void main(String[] args) {        Thread t = Thread.currentThread(); // получаем главный поток      System.out.println(t); // main  } |

Консольный вывод:

Thread[main,5,main]

Первое main будет представлять имя потока (что можно получить через t.getName()), второе значение 5 предоставляет приоритет потока (также можно получить через t.getPriority()), и последнее main представляет имя группы потоков, к которому относится текущий - по умолчанию также main (также можно получить через t.getThreadGroup().getName())

**Недостатки при использовании потоков**

Далее мы рассмотрим, как создавать и использовать потоки. Это довольно легко. Однако при создании многопоточного приложения нам следует учитывать ряд обстоятельств, которые негативно могут сказаться на работе приложения.

На некоторых платформах запуск новых потоков может замедлить работу приложения. Что может иметь большое значение, если нам критичная производительность приложения.

Для каждого потока создается свой собственный стек в памяти, куда помещаются все локальные переменные и ряд других данных, связанных с выполнением потока. Соответственно, чем больше потоков создается, тем больше памяти используется. При этом надо помнить, в любой системе размеры используемой памяти ограничены. Кроме того, во многих системах может быть ограничение на количество потоков. Но даже если такого ограничения нет, то в любом случае имеется естественное ограничение в виде максимальной скорости процессора.

**Создание и выполнение потоков**

Для создания нового потока мы можем создать новый класс:

- либо наследуя его от класса **Thread**,

- либо реализуя в классе интерфейс **Runnable**,

- либо реализуя в классе интерфейс **Callable**.

Плюс использования реализации через **Runnable/Callable,** это то, что можно не создавать отдельный класс с какой-то реализацией потока, а используя механизм написания “через лямбду” написать реализацию потока сразу в классе, который будет запускать и использовать этот поток. + если класс уже что-то наследует (extend-ит), то добавить еще механизмы можно только через implement-ирование интерфейсов.

Если требуется выполнить простую операцию в отдельном потоке без возвращения результата и обработки исключений, то лучше использовать **Runnable**. Если же требуется выполнить более сложную операцию с обработкой исключений и возвращением результата, то стоит выбирать **Callable**.

**Наследование от класса Thread**

Создадим свой класс на основе Thread:

|  |  |
| --- | --- |
| 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 | class JThread extends Thread {        JThread(String name){          super(name);      }        public void run(){            System.out.printf("%s started... \n", Thread.currentThread().getName());          try{              Thread.sleep(500);          }          catch(InterruptedException e){              System.out.println("Thread has been interrupted");          }          System.out.printf("%s fiished... \n", Thread.currentThread().getName());      }  }    public class Program {        public static void main(String[] args) {            System.out.println("Main thread started...");          new JThread("JThread").start();          System.out.println("Main thread finished...");      }  } |

Класс потока называется JThread. Предполагается, что в конструктор класса передается имя потока, которое затем передается в конструктор базового класса. В конструктор своего класса потока мы можем передать различные данные, но главное, чтобы в нем вызывался конструктор базового класса Thread, в который передается имя потока.

И также в JThread переопределяется метод **run()**, код которого собственно и будет представлять весь тот код, который выполняется в потоке.

В методе main для запуска потока JThread у него вызывается метод **start()**, после чего начинается выполнение того кода, который определен в методе run:

|  |  |
| --- | --- |
| 1 | new JThread("JThread").start(); |

Консольный вывод:

Main thread started...

Main thread finished...

JThread started...

JThread finished...

Здесь в методе main в конструктор JThread передается произвольное название потока, и затем вызывается метод start(). По сути этот метод как раз и вызывает переопределенный метод run() класса JThread.

Обратите внимание, что главный поток завершает работу раньше, чем порожденный им дочерний поток JThread.

Аналогично созданию одного потока мы можем запускать сразу несколько потоков:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public static void main(String[] args) {        System.out.println("Main thread started...");      for(int i=1; i < 6; i++)          new JThread("JThread " + i).start();      System.out.println("Main thread finished...");  } |

Консольный вывод:

Main thread started...

Main thread finished...

JThread 2 started...

JThread 5 started...

JThread 4 started...

JThread 1 started...

JThread 3 started...

JThread 1 finished...

JThread 2 finished...

JThread 5 finished...

JThread 4 finished...

JThread 3 finished...

**Ожидание завершения потока**

При запуске потоков в примерах выше Main thread завершался до дочернего потока. Как правило, более распространенной ситуацией является случай, когда Main thread завершается самым последним. Для этого надо применить метод **join()**. В этом случае текущий поток будет ожидать завершения потока, для которого вызван метод join:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public static void main(String[] args) {        System.out.println("Main thread started...");      JThread t= new JThread("JThread ");      t.start();      try{          t.join();      }      catch(InterruptedException e){            System.out.printf("%s has been interrupted", t.getName());      }      System.out.println("Main thread finished...");  } |

Метод join() заставляет вызвавший поток (в данном случае Main thread) ожидать завершения вызываемого потока, для которого и применяется метод join (в данном случае JThread).

Консольный вывод:

Main thread started...

JThread started...

JThread finished...

Main thread finished...

Если в программе используется несколько дочерних потоков, и надо, чтобы Main thread завершался после дочерних, то для каждого дочернего потока надо вызвать метод join.

**Реализация интерфейса Runnable, Callable**

Другой способ определения потока представляет реализация интерфейса **Runnable**. Этот интерфейс имеет один метод **run**:

|  |
| --- |
| interface Runnable{      void run();  }  interface Callable{      void call();  } |

В методе run() собственно определяется весь тот код, который выполняется при запуске потока.

После определения объекта Runnable он передается в один из конструкторов класса Thread:

|  |  |
| --- | --- |
| 1 | Thread(Runnable runnable, String threadName) |

Для реализации интерфейса определим следующий класс MyThread:

|  |  |
| --- | --- |
| 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 | class MyThread implements Runnable {         //этот метод идентичен методу run() при создании потока через наследование от класса Thread      public void run(){            System.out.printf("%s started... \n", Thread.currentThread().getName());          try{              Thread.sleep(500);          }          catch(InterruptedException e){              System.out.println("Thread has been interrupted");          }          System.out.printf("%s finished... \n", Thread.currentThread().getName());      }  }    public class Program {        public static void main(String[] args) {            System.out.println("Main thread started...");  //а тут написание создания и старта потока слегка отличается от предыдущего(через наследование от класса Thread)          Thread myThread = new Thread(new MyThread(),"MyThread");          myThread.start();          System.out.println("Main thread finished...");      }  } |

Реализация интерфейса Runnable во многом аналогична переопределению класса Thread. Также в методе run определяется простейший код, который усыпляет поток на 500 миллисекунд.

В методе main вызывается конструктор Thread, в который передается объект MyThread. И чтобы запустить поток, вызывается метод start(). В итоге консоль выведет что-то наподобие следующего:

Main thread started...

Main thread finished...

MyThread started...

MyThread finished...

Поскольку Runnable фактически представляет функциональный интерфейс, который определяет один метод, то объект этого интерфейса мы можем представить в виде лямбда-выражения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class Program {        public static void main(String[] args) {            System.out.println("Main thread started...");          Runnable r = ()->{              System.out.printf("%s started... \n", Thread.currentThread().getName());              try{                  Thread.sleep(500);              }              catch(InterruptedException e){                  System.out.println("Thread has been interrupted");              }              System.out.printf("%s finished... \n", Thread.currentThread().getName());          };          Thread myThread = new Thread(r,"MyThread");          myThread.start();          System.out.println("Main thread finished...");      }  } |

**Завершение и прерывание потока**

Примеры потоков ранее представляли поток как последовательный набор операций. После выполнения последней операции завершался и поток. Однако нередко имеет место и другая организация потока в виде бесконечного цикла. Например, поток сервера в бесконечном цикле прослушивает определенный порт на предмет получения данных. И в этом случае мы также можем предусмотреть механизм завершения потока.

**Завершение потока**

Распространенный способ завершения потока представляет опрос логической переменной. И если она равна, например, false, то поток завершает бесконечный цикл и заканчивает свое выполнение.

Определим следующий класс потока:

|  |  |
| --- | --- |
| 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 | class MyThread implements Runnable {        private boolean isActive;        void disable(){          isActive=false;      }        MyThread(){         isActive = true;      }        public void run(){            System.out.printf("%s started... \n", Thread.currentThread().getName());          int counter=1; // счетчик циклов          while(isActive){              System.out.println("Loop " + counter++);              try{                  Thread.sleep(400);              }              catch(InterruptedException e){                  System.out.println("Thread has been interrupted");              }          }          System.out.printf("%s finished... \n", Thread.currentThread().getName());      }  } |

Переменная isActive указывает на активность потока. С помощью метода disable() мы можем сбросить состояние этой переменной.

Теперь используем этот класс:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public static void main(String[] args) {        System.out.println("Main thread started...");      MyThread myThread = new MyThread();      new Thread(myThread,"MyThread").start();        try{          Thread.sleep(1100);            myThread.disable();            Thread.sleep(1000);      }      catch(InterruptedException e){          System.out.println("Thread has been interrupted");      }      System.out.println("Main thread finished...");  } |

Итак, вначале запускается дочерний поток: new Thread(myThread,"MyThread").start(). Затем на 1100 миллисекунд останавливаем Main thread и потом вызываем метод myThread.disable(), который переключает в потоке флаг isActive. И дочерний поток завершается.

**Метод interrupt**

Еще один способ вызова завершения или прерывания потока представляет метод **interrupt()**. Вызов этого метода устанавливает у потока статус, что он прерван. Сам метод возвращает true, если поток может быть прерван, в ином случае возвращается false.

При этом сам вызов этого метода НЕ завершает поток, он только устанавливает статус: в частности, метод **isInterrupted()**класса Thread будет возвращать значение true. Мы можем проверить значение возвращаемое данным методом и прозвести некоторые действия. Например:

|  |  |
| --- | --- |
| 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 | class JThread extends Thread {        JThread(String name){          super(name);      }      public void run(){            System.out.printf("%s started... \n", Thread.currentThread().getName());          int counter=1; // счетчик циклов          while(!isInterrupted()){                System.out.println("Loop " + counter++);          }          System.out.printf("%s finished... \n", Thread.currentThread().getName());      }  }  public class Program {        public static void main(String[] args) {            System.out.println("Main thread started...");          JThread t = new JThread("JThread");          t.start();          try{              Thread.sleep(150);              t.interrupt();                Thread.sleep(150);          }          catch(InterruptedException e){              System.out.println("Thread has been interrupted");          }          System.out.println("Main thread finished...");      }  } |

В классе, который унаследован от Thread, мы можем получить статус текущего потока с помощью метода **isInterrupted()**. И пока этот метод возвращает false, мы можем выполнять цикл. А после того, как будет вызван метод interrupt, isInterrupted()возвратит true, и соответственно произойдет выход из цикла.

Возможный консольный вывод:

Main thread started...

JThread started...

Loop 1

Loop 2

Loop 3

Loop 4

JThread finished...

Main thread finished...

Если основная функциональность заключена в классе, который реализует интерфейс Runnable, то там можно проверять статус потока с помощью метода **Thread.currentThread().isInterrupted()**:

|  |  |
| --- | --- |
| 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 | class MyThread implements Runnable {        public void run(){            System.out.printf("%s started... \n", Thread.currentThread().getName());          int counter=1; // счетчик циклов          while(!Thread.currentThread().isInterrupted()){                System.out.println("Loop " + counter++);          }          System.out.printf("%s finished... \n", Thread.currentThread().getName());      }  }  public class Program {        public static void main(String[] args) {            System.out.println("Main thread started...");          MyThread myThread = new MyThread();          Thread t = new Thread(myThread,"MyThread");          t.start();          try{              Thread.sleep(150);              t.interrupt();                Thread.sleep(150);          }          catch(InterruptedException e){              System.out.println("Thread has been interrupted");          }          System.out.println("Main thread finished...");      }  } |

Однако при получении статуса потока с помощью метода isInterrupted() следует учитывать, что если мы обрабатываем в цикле исключение **InterruptedException** в блоке catch, то при перехвате исключения статус потока автоматически сбрасывается, и после этого isInterrupted будет возвращать false.

Например, добавим в цикл потока задержку с помощью метода sleep:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public void run(){        System.out.printf("%s started... \n", Thread.currentThread().getName());      int counter=1; // счетчик циклов      while(!isInterrupted()){            System.out.println("Loop " + counter++);          try{              Thread.sleep(100);          }          catch(InterruptedException e){              System.out.println(getName() + " has been interrupted");              System.out.println(isInterrupted());    // false              interrupt();    // повторно сбрасываем состояние          }      }      System.out.printf("%s finished... \n", Thread.currentThread().getName());  } |

Когда поток вызовет метод interrupt, метод sleep сгенерирует исключение InterruptedException, и управление перейдет к блоку catch. Но если мы проверим статус потока, то увидим, что метод isInterrupted возвращает false. Как вариант, в этом случае мы можем повторно прервать текущий поток, опять же вызвав метод interrupt(). Тогда при новой итерации цикла while метода isInterrupted возвратит true, и поизойдет выход из цикла.

Либо мы можем сразу же в блоке catch выйти из цикла с помощью break:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | while(!isInterrupted()){        System.out.println("Loop " + counter++);      try{          Thread.sleep(100);      }      catch(InterruptedException e){          System.out.println(getName() + " has been interrupted");            break;  // выход из цикла      }  } |

Если бесконечный цикл помещен в конструкцию try...catch, то достаточно обработать InterruptedException:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public void run(){        System.out.printf("%s started... \n", Thread.currentThread().getName());      int counter=1; // счетчик циклов      try{          while(!isInterrupted()){              System.out.println("Loop " + counter++);              Thread.sleep(100);          }      }      catch(InterruptedException e){          System.out.println(getName() + " has been interrupted");      }        System.out.printf("%s finished... \n", Thread.currentThread().getName());  } |

**Синхронизация потоков. Оператор synchronized**

При работе потоки нередко обращаются к каким-то общим ресурсам, которые определены вне потока, например, обращение к какому-то файлу. Если одновременно несколько потоков обратятся к общему ресурсу, то результаты выполнения программы могут быть неожиданными и даже непредсказуемыми. Например, определим следующий код:

|  |  |
| --- | --- |
| 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 | public class Program {        public static void main(String[] args) {            CommonResource commonResource= new CommonResource();          for (int i = 1; i < 6; i++){                Thread t = new Thread(new CountThread(commonResource));              t.setName("Thread "+ i);              t.start();          }      }  }    class CommonResource{        int x=0;  }    class CountThread implements Runnable{        CommonResource res;      CountThread(CommonResource res){          this.res=res;      }      public void run(){          res.x=1;          for (int i = 1; i < 5; i++){              System.out.printf("%s %d \n", Thread.currentThread().getName(), res.x);              res.x++;              try{                  Thread.sleep(100);              }              catch(InterruptedException e){}          }      }  } |

Здесь определен класс CommonResource, который представляет общий ресурс и в котором определено одно целочисленное поле x.

Этот ресурс используется классом потока CountThread. Этот класс просто увеличивает в цикле значение x на единицу. Причем при входе в поток значение x=1:

|  |  |
| --- | --- |
| 1 | res.x=1; |

То есть в итоге мы ожидаем, что после выполнения цикла res.x будет равно 4.

В главном классе программы запускается пять потоков. То есть мы ожидаем, что каждый поток будет увеличивать res.x с 1 до 4 и так пять раз. Но если мы посмотрим на результат работы программы, то он будет иным:

Thread 1 1

Thread 2 1

Thread 3 1

Thread 5 1

Thread 4 1

Thread 5 6

Thread 2 6

Thread 1 6

Thread 3 6

Thread 4 6

Thread 4 11

Thread 2 11

Thread 5 11

Thread 3 11

Thread 1 11

Thread 4 16

Thread 1 16

Thread 3 16

Thread 5 16

Thread 2 16

То есть пока один поток не окончил работу с полем res.x, с ним начинает работать другой поток.

Чтобы избежать подобной ситуации, надо синхронизировать потоки. Одним из способов синхронизации является использование ключевого слова **synchronized**. Этот оператор предваряет блок кода или метод, который подлежит синхронизации. Для его применения изменим класс CountThread:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class CountThread implements Runnable{        CommonResource res;      CountThread(CommonResource res){          this.res=res;      }      public void run(){          synchronized(res){              res.x=1;              for (int i = 1; i < 5; i++){                  System.out.printf("%s %d \n", Thread.currentThread().getName(), res.x);                  res.x++;                  try{                      Thread.sleep(100);                  }                  catch(InterruptedException e){}              }          }      }  } |

При создании синхронизированного блока кода после оператора synchronized идет объект-заглушка: synchronized(res). Причем в качестве объекта может использоваться только объект какого-нибудь класса, но не примитивного типа.

Каждый объект в Java имеет ассоциированный с ним **монитор**. Монитор представляет своего рода инструмент для управления доступа к объекту. Когда выполнение кода доходит до оператора synchronized, монитор объекта res блокируется, и на время его блокировки монопольный доступ к блоку кода имеет только один поток, который и произвел блокировку. После окончания работы блока кода, монитор объекта res освобождается и становится доступным для других потоков.

После освобождения монитора его захватывает другой поток, а все остальные потоки продолжают ожидать его освобождения.

В итоге консольный вывод изменится:

Thread 1 1

Thread 1 2

Thread 1 3

Thread 1 4

Thread 3 1

Thread 3 2

Thread 3 3

Thread 3 4

Thread 5 1

Thread 5 2

Thread 5 3

Thread 5 4

Thread 4 1

Thread 4 2

Thread 4 3

Thread 4 4

Thread 2 1

Thread 2 2

Thread 2 3

Thread 2 4

При применении оператора synchronized к методу пока этот метод не завершит выполнение, монопольный доступ имеет только один поток - первый, который начал его выполнение. Для применения synchronized к методу, изменим классы программы:

|  |  |
| --- | --- |
| 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 | public class Program {        public static void main(String[] args) {            CommonResource commonResource= new CommonResource();          for (int i = 1; i < 6; i++){                Thread t = new Thread(new CountThread(commonResource));              t.setName("Thread "+ i);              t.start();          }      }  }    class CommonResource{        int x;      synchronized void increment(){          x=1;          for (int i = 1; i < 5; i++){              System.out.printf("%s %d \n", Thread.currentThread().getName(), x);              x++;              try{                  Thread.sleep(100);              }              catch(InterruptedException e){}          }      }  }    class CountThread implements Runnable{        CommonResource res;      CountThread(CommonResource res){          this.res=res;      }        public void run(){          res.increment();      }  } |

Результат работы в данном случае будет аналогичен примеру выше с блоком synchronized. Здесь опять в дело вступает монитор объекта CommonResource - общего объекта для всех потоков. Поэтому синхронизированным объявляется не метод run() в классе CountThread, а метод increment класса CommonResource. Когда первый поток начинает выполнение метода increment, он захватывает монитор объекта CommonResource. А все потоки также продолжают ожидать его освобождения.

**Методы wait и notify**

Иногда при взаимодействии потоков встает вопрос о извещении одних потоков о действиях других. Например, действия одного потока зависят от результата действий другого потока, и надо как-то известить один поток, что второй поток произвел некую работу. И для подобных ситуаций у класса **Object** определено ряд методов:

* **wait()**: освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод notify()
* **notify()**: продолжает работу потока, у которого ранее был вызван метод wait()
* **notifyAll()**: возобновляет работу всех потоков, у которых ранее был вызван метод wait()

Все эти методы вызываются только из синхронизированного контекста - синхронизированного блока или метода.

Рассмотрим, как мы можем использовать эти методы. Возьмем стандартную задачу из прошлой темы - "Производитель-Потребитель" ("Producer-Consumer"): пока производитель не произвел продукт, потребитель не может его купить. Пусть производитель должен произвести 5 товаров, соответственно потребитель должен их все купить. Но при этом одновременно на складе может находиться не более 3 товаров. Для решения этой задачи задействуем методы wait() и notify():

|  |  |
| --- | --- |
| 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 | public class Program {        public static void main(String[] args) {            Store store=new Store();          Producer producer = new Producer(store);          Consumer consumer = new Consumer(store);          new Thread(producer).start();          new Thread(consumer).start();      }  }  // класс Производитель  class Producer implements Runnable{        Store store;      Producer(Store store){         this.store=store;      }      public void run(){          for (int i = 1; i < 6; i++) {              store.put();          }      }  }  // Класс Потребитель  class Consumer implements Runnable{         Store store;      Consumer(Store store){         this.store=store;      }      public void run(){          for (int i = 1; i < 6; i++) {              store.get();          }      }  }  // Класс Магазин, хранящий произведенные товары  class Store{     private int product=0;     public synchronized void get() {        while (product<1) {           try {              wait();           }           catch (InterruptedException e) {           }        }        product--;        System.out.println("Покупатель купил 1 товар");        System.out.println("Товаров на складе: " + product);        notify(); //будим поток в методе *put().* И правильно будет будить поток в методе *put()*,  //если *product < 3*, иначе не будим. Т.к. если поток метода *put()* будет  //умудряться часто делать два цикла в то время как поток метода get() будет  //успевать делать только один цикл,то мы получим переполнение склада     }     public synchronized void put() {         while (product>=3) {           try {              wait();           }           catch (InterruptedException e) {           }        }        product++;        System.out.println("Производитель добавил 1 товар");        System.out.println("Товаров на складе: " + product);        notify();//будим поток в методе *get()*     }  } |

Итак, здесь определен класс магазина, потребителя и покупателя. Производитель в методе run() добавляет в объект Store с помощью его метода put() 6 товаров. Потребитель в методе run() в цикле обращается к методу get объекта Store для получения этих товаров. Оба метода Store - put и get являются синхронизированными.

Для отслеживания наличия товаров в классе Store проверяем значение переменной product. По умолчанию товара нет, поэтому переменная равна 0. Метод get() - получение товара должен срабатывать только при наличии хотя бы одного товара. Поэтому в методе get проверяем, отсутствует ли товар:

|  |  |
| --- | --- |
| 1 | while (product<1) |

Если товар отсутсвует, вызывается метод wait(). Этот метод освобождает монитор объекта Store и блокирует выполнение метода get, пока для этого же монитора не будет вызван метод notify().

Когда в методе put() добавляется товар и вызывается notify(), то метод get() получает монитор и выходит из конструкции while (product<1), так как товар добавлен. Затем имитируется получение покупателем товара. Для этого выводится сообщение, и уменьшается значение product: product--. И в конце вызов метода notify() дает сигнал методу put() продолжить работу.

В методе put() работает похожая логика, только теперь метод put() должен срабатывать, если в магазине не более трех товаров. Поэтому в цикле проверяется наличие товара, и если товар уже есть, то освобождаем монитор с помощью wait() и ждем вызова notify() в методе get().

И теперь программа покажет нам другие результаты:

Производитель добавил 1 товар

Товаров на складе: 1

Производитель добавил 1 товар

Товаров на складе: 2

Производитель добавил 1 товар

Товаров на складе: 3

Покупатель купил 1 товар

Товаров на складе: 2

Покупатель купил 1 товар

Товаров на складе: 1

Покупатель купил 1 товар

Товаров на складе: 0

Производитель добавил 1 товар

Товаров на складе: 1

Производитель добавил 1 товар

Товаров на складе: 2

Покупатель купил 1 товар

Товаров на складе: 1

Покупатель купил 1 товар

Товаров на складе: 0

Таким образом, с помощью wait() в методе get() мы ожидаем, когда производитель добавит новый продукт. А после добавления вызываем notify(), как бы говоря, что магазин теперь снова пуст, и можно еще добавлять.

А в методе put() с помощью wait() мы ожидаем освобождения места на складе. После того, как место освободится, добавляем товар и через notify() уведомляем покупателя о том, что он может забирать товар.

**Семафоры**

Семафоры представляют еще одно средство синхронизации для доступа к ресурсу. В Java семафоры представлены классом **Semaphore**, который располагается в пакете **java.util.concurrent**.

Для управления доступом к ресурсу семафор использует счетчик, представляющий количество разрешений. Если значение счетчика больше нуля, то поток получает доступ к ресурсу, при этом счетчик уменьшается на единицу. После окончания работы с ресурсом поток освобождает семафор, и счетчик увеличивается на единицу. Если же счетчик равен нулю, то поток блокируется и ждет, пока не получит разрешение от семафора.

Установить количество разрешений для доступа к ресурсу можно с помощью конструкторов класса Semaphore:

|  |  |
| --- | --- |
| 1  2 | Semaphore(int permits)  Semaphore(int permits, boolean fair) |

Параметр permits указывает на количество допустимых разрешений для доступа к ресурсу. Параметр fair во втором конструкторе позволяет установить очередность получения доступа. Если он равен true, то разрешения будут предоставляться ожидающим потокам в том порядке, в каком они запрашивали доступ. Если же он равен false, то разрешения будут предоставляться в неопределенном порядке.

Для получения разрешения у семафора надо вызвать метод **acquire()**, который имеет две формы:

|  |  |
| --- | --- |
| 1  2 | void acquire() throws InterruptedException  void acquire(int permits) throws InterruptedВxception |

Для получения одного разрешения применяется первый вариант, а для получения нескольких разрешений - второй вариант.

После вызова этого метода пока поток не получит разрешение, он блокируется.

После окончания работы с ресурсом полученное ранее разрешение надо освободить с помощью метода **release()**:

|  |  |
| --- | --- |
| 1  2 | void release()  void release(int permits) |

Первый вариант метода освобождает одно разрешение, а второй вариант - количество разрешений, указанных в permits.

Используем семафор в простом примере:

|  |  |
| --- | --- |
| 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 | import java.util.concurrent.Semaphore;    public class Program {        public static void main(String[] args) {            Semaphore sem = new Semaphore(1); // 1 разрешение          CommonResource res = new CommonResource();          new Thread(new CountThread(res, sem, "CountThread 1")).start();          new Thread(new CountThread(res, sem, "CountThread 2")).start();          new Thread(new CountThread(res, sem, "CountThread 3")).start();      }  }  class CommonResource{        int x=0;  }    class CountThread implements Runnable{        CommonResource res;      Semaphore sem;      String name;      CountThread(CommonResource res, Semaphore sem, String name){          this.res=res;          this.sem=sem;          this.name=name;      }        public void run(){            try{              System.out.println(name + " ожидает разрешение");              sem.acquire();              res.x=1;              for (int i = 1; i < 5; i++){                  System.out.println(this.name + ": " + res.x);                  res.x++;                  Thread.sleep(100);              }          }          catch(InterruptedException e){System.out.println(e.getMessage());}          System.out.println(name + " освобождает разрешение");          sem.release();      }  } |

Итак, здесь есть общий ресурс CommonResource с полем x, которое изменяется каждым потоком. Потоки представлены классом CountThread, который получает семафор и выполняет некоторые действия над ресурсом. В основном классе программы эти потоки запускаются. В итоге мы получим следующий вывод:

CountThread 1 ожидает разрешение

CountThread 2 ожидает разрешение

CountThread 3 ожидает разрешение

CountThread 1: 1

CountThread 1: 2

CountThread 1: 3

CountThread 1: 4

CountThread 1 освобождает разрешение

CountThread 3: 1

CountThread 3: 2

CountThread 3: 3

CountThread 3: 4

CountThread 3 освобождает разрешение

CountThread 2: 1

CountThread 2: 2

CountThread 2: 3

CountThread 2: 4

CountThread 2 освобождает разрешение

Семафоры отлично подходят для решения задач, где надо ограничивать доступ. Например, классическая задача про обедающих философов. Ее суть: есть несколько философов, допустим, пять, но одновременно за столом могут сидеть не более двух. И надо, чтобы все философы пообедали, но при этом не возникло взаимоблокировки философами друг друга в борьбе за тарелку и вилку:

|  |  |
| --- | --- |
| 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 | import java.util.concurrent.Semaphore;    public class Program {        public static void main(String[] args) {            Semaphore sem = new Semaphore(2);          for(int i=1;i<6;i++)              new Philosopher(sem,i).start();      }  }  // класс философа  class Philosopher extends Thread  {      Semaphore sem; // семафор. ограничивающий число философов      // кол-во приемов пищи      int num = 0;      // условный номер философа      int id;      // в качестве параметров конструктора передаем идентификатор философа и семафор      Philosopher(Semaphore sem, int id)      {          this.sem=sem;          this.id=id;      }        public void run()      {          try          {              while(num<3)// пока количество приемов пищи не достигнет 3              {                  //Запрашиваем у семафора разрешение на выполнение                  sem.acquire();                  System.out.println ("Философ " + id+" садится за стол");                  // философ ест                  sleep(500);                  num++;                    System.out.println ("Философ " + id+" выходит из-за стола");                  sem.release();                    // философ гуляет                  sleep(500);              }          }          catch(InterruptedException e)          {              System.out.println ("у философа " + id + " проблемы со здоровьем");          }      }  } |

В итоге только два философа смогут одновременно находиться за столом, а другие будут ждать:

Философ 1 садится за стол

Философ 3 садится за стол

Философ 3 выходит из-за стола

Философ 1 выходит из-за стола

Философ 2 садится за стол

Философ 4 садится за стол

Философ 2 выходит из-за стола

Философ 4 выходит из-за стола

Философ 5 садится за стол

Философ 1 садится за стол

Философ 1 выходит из-за стола

Философ 5 выходит из-за стола

Философ 3 садится за стол

Философ 2 садится за стол

Философ 3 выходит из-за стола

Философ 4 садится за стол

Философ 2 выходит из-за стола

Философ 5 садится за стол

Философ 4 выходит из-за стола

Философ 5 выходит из-за стола

Философ 1 садится за стол

Философ 3 садится за стол

Философ 1 выходит из-за стола

Философ 2 садится за стол

Философ 3 выходит из-за стола

Философ 5 садится за стол

Философ 2 выходит из-за стола

Философ 4 садится за стол

Философ 5 выходит из-за стола

Философ 4 выходит из-за стола

**Обмен между потоками. Класс Exchanger**

Класс **Exchanger** предназначен для обмена данными между потоками. Он является типизированным и типизируется типом данных, которыми потоки должны обмениваться.

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

|  |  |
| --- | --- |
| 1  2 | 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.

Данный класс очень просто использовать:

|  |  |
| --- | --- |
| 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 | import java.util.concurrent.Exchanger;    public class Program {        public static void main(String[] args) {            Exchanger<String> ex = new Exchanger<String>();          new Thread(new PutThread(ex)).start();          new Thread(new GetThread(ex)).start();      }  }    class PutThread implements Runnable{        Exchanger<String> exchanger;      String message;        PutThread(Exchanger<String> ex){            this.exchanger=ex;          message = "Hello Java!";      }      public void run(){            try{              message=exchanger.exchange(message);              System.out.println("PutThread has received: " + message);          }          catch(InterruptedException ex){              System.out.println(ex.getMessage());          }      }  }  class GetThread implements Runnable{        Exchanger<String> exchanger;      String message;        GetThread(Exchanger<String> ex){            this.exchanger=ex;          message = "Hello World!";      }      public void run(){            try{              message=exchanger.exchange(message);              System.out.println("GetThread has received: " + message);          }          catch(InterruptedException ex){              System.out.println(ex.getMessage());          }      }  } |

В классе PutThread отправляет в буфер сообщение "Hello Java!":

|  |  |
| --- | --- |
| 1 | message=exchanger.exchange(message); |

Причем в ответ метод exchange возвращает данные, которые отправил в буфер другой поток. То есть происходит обмен данными. Хотя нам необязательно получать данные, мы можем просто их отправить:

|  |  |
| --- | --- |
| 1 | exchanger.exchange(message); |

Логика класса GetThread аналогична - также отправляется сообщение.

В итоге консоль выведет следующий результат:

PutThread has received: Hello World!

GetThread has received: Hello Java!

**Класс Phaser**

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.

Для создания объекта Phaser используется один из конструкторов:

|  |  |
| --- | --- |
| 1  2  3  4 | Phaser()  Phaser(int parties)  Phaser(Phaser parent)  Phaser(Phaser parent, int parties) |

Параметр parties указывает на количество сторон (грубо говоря, потоков), которые должны выполнять все фазы действия. Первый конструктор создает объект Phaser без каких-либо сторон. Второй конструктор регистрирует передаваемое в конструктор количество сторон. Третий и четвертый конструкторы также устанавливают родительский объект Phaser.

Основные методы класса Phaser:

* **int register()**: регистрирует сторону, которая выполняет фазы, и возвращает номер текущей фазы - обычно фаза 0
* **int arrive()**: сообщает, что сторона завершила фазу и возвращает номер текущей фазы
* **int arriveAndAwaitAdvance()**: аналогичен методу arrive, только при этом заставляет phaser ожидать завершения фазы всеми остальными сторонами
* **int arriveAndDeregister()**: сообщает о завершении всех фаз стороной и снимает ее с регистрации. Возвращает номер текущей фазы или отрицательное число, если синхронизатор Phaser завершил свою работу
* **int getPhase()**: возвращает номер текущей фазы

При работае с классом Phaser обычно сначала создается его объект. Далее нам надо зарегистрировать все участвующие стороны. Для регистрации в каждой стороне вызывается метод register(), либо можно обойтись и без этого метода, передав нужное количество сторон в конструктор Phaser.

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

Применим Phaser в приложении:

|  |  |
| --- | --- |
| 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 | import java.util.concurrent.Phaser;    public class Program {        public static void main(String[] args) {            Phaser phaser = new Phaser(1);          new Thread(new PhaseThread(phaser, "PhaseThread 1")).start();          new Thread(new PhaseThread(phaser, "PhaseThread 2")).start();            // ждем завершения фазы 0          int phase = phaser.getPhase();          phaser.arriveAndAwaitAdvance();          System.out.println("Фаза " + phase + " завершена");          // ждем завершения фазы 1          phase = phaser.getPhase();          phaser.arriveAndAwaitAdvance();          System.out.println("Фаза " + phase + " завершена");            // ждем завершения фазы 2          phase = phaser.getPhase();          phaser.arriveAndAwaitAdvance();          System.out.println("Фаза " + phase + " завершена");            phaser.arriveAndDeregister();      }  }    class PhaseThread implements Runnable{        Phaser phaser;      String name;        PhaseThread(Phaser p, String n){            this.phaser=p;          this.name=n;          phaser.register();      }      public void run(){            System.out.println(name + " выполняет фазу " + phaser.getPhase());          phaser.arriveAndAwaitAdvance(); // сообщаем, что первая фаза достигнута            System.out.println(name + " выполняет фазу " + phaser.getPhase());          phaser.arriveAndAwaitAdvance(); // сообщаем, что вторая фаза достигнута            System.out.println(name + " выполняет фазу " + phaser.getPhase());          phaser.arriveAndDeregister(); // сообщаем о завершении фаз и удаляем с регистрации объекты      }  } |

Итак, здесь у нас фазы выполняются тремя сторонами - главным потоком и двумя потоками PhaseThread. Поэтому при создании объекта Phaser ему передается число 1 - главный поток, а в конструкторе PhaseThread вызывается метод register(). Мы в принципе могли бы не использовать метод register, но тогда нам надо было бы указать Phaser phaser = new Phaser(3), так как у нас три стороны.

Фаза в каждой стороне представляет минимальный примитивный набор действий: для потоков PhaseThread это вывод сообщения, а для главного потока - подсчет текущей фазы с помощью метода getPhase(). При этом отсчет фаз начинается с нуля. Каждая сторона завершает выполнение фазы вызовом метода phaser.arriveAndAwaitAdvance(). При вызове этого метода пока последняя сторона не завершит выполнение текущей фазы, все остальные стороны блокируются.

После завершения выполнения последней фазы происходит отмена регистрации всех сторон с помощью метода arriveAndDeregister().

В итоге работа программы даст следующий вывод:

PhaseThread 1 выполняет фазу 0

PhaseThread 2 выполняет фазу 0

PhaseThread 1 выполняет фазу 1

PhaseThread 2 выполняет фазу 1

Фаза 0 завершена

Фаза 1 завершена

PhaseThread 1 выполняет фазу 2

PhaseThread 2 выполняет фазу 2

Фаза 2 завершена

В данном случае получается немного путанный вывод. Так, сообщения о выполнении фазы 1 выводится после сообщения об окончании фазы 0. Что связано с многопоточностью - фазы завершились, но в одном потоке еще не выведено сообщение о завершении, тогда как другие потоки уже начали выполнение следующей фазы. В любом случае все это происходит уже после завершения фазы.

Но чтобы было более наглядно, мы можем использовать sleep в потоках:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | public void run(){            System.out.println(name + " выполняет фазу " + phaser.getPhase());          phaser.arriveAndAwaitAdvance(); // сообщаем, что первая фаза достигнута          try{              Thread.sleep(200);          }          catch(InterruptedException ex){              System.out.println(ex.getMessage());          }            System.out.println(name + " выполняет фазу " + phaser.getPhase());          phaser.arriveAndAwaitAdvance(); // сообщаем, что вторая фаза достигнута          try{              Thread.sleep(200);          }          catch(InterruptedException ex){              System.out.println(ex.getMessage());          }          System.out.println(name + " выполняет фазу " + phaser.getPhase());          phaser.arriveAndDeregister(); // сообщаем о завершении фаз и удаляем с регистрации объекты      } |

И в этом случае вывод будет более привычным, хотя на работу фаз это никак не повлияет.

PhaseThread 1 выполняет фазу 0

PhaseThread 2 выполняет фазу 0

Фаза 0 завершена

PhaseThread 2 выполняет фазу 1

PhaseThread 1 выполняет фазу 1

Фаза 1 завершена

PhaseThread 2 выполняет фазу 2

PhaseThread 1 выполняет фазу 2

Фаза 2 завершена