# **Работа с InterruptedException**

Эта история, наверное, вам знакома: вы пишете тестовую программу, и вам нужна на некоторое время пауза, поэтому вы вызываете Thread.sleep(). Но тут компилятор или IDE отклоняет это, заявляя, что вы не отреагировали на отмеченное InterruptedException. Что такое InterruptedException, и почему вам приходится с ним иметь дело?

Наиболее типичной реакцией на InterruptedException является его игнорирование, когда вы перехватываете его и ничего с ним не делаете (или, возможно, регистрируете его, что вряд ли намного лучше) - как мы увидим позднее в [Листинге 4](https://www.ibm.com/developerworks/ru/library/j-jtp05236/index.html#code4). К сожалению, этот подход отбрасывает важную информацию о том, что произошло прерывание, что может подвергнуть риску способность приложения отменять функции или своевременно прекратить работу.

## Методы блокирования

Когда метод выдает InterruptedException, мы фактически получаем сразу несколько сообщений, в дополнение к тому, что он может сгенерировать отмеченное исключение. Он говорит вам о том, что это метод блокирования, и что он предпримет попытку разблокироваться и вернуться на более ранний этап - если вы хорошо попросите.

Метод блокирования отличается от обычного метода, которому требуется много времени для запуска. Завершение обычного метода зависит только от того, как много работы вы на него возложили, и имеются ли в наличии соответствующие компьютерные ресурсы (циклы центрального процессора и память). Завершение метода блокирования, с другой стороны, также зависит от некоего внешнего события, такого как прекращение действия таймера, завершение ввода-вывода, или действие другого потока (снимающего блокировочный замок, устанавливающего метку или ставящего в очередь задачу). Обычные методы завершаются, как только их работа выполнена, но методы блокирования менее предсказуемы, поскольку они зависят от внешних событий. Методы блокирования снижают время реакции, поскольку бывает трудно предсказать, когда они завершатся.

Поскольку методы блокирования потенциально могут занять сколь угодно много времени, если ожидаемое ими событие никогда не произойдет, то зачастую очень полезно для операций блокирования быть отменяемыми (cancelable). (Зачастую для долго работающих неблокирующих методов также полезно быть отменяемыми.) Отменяемой операцией является та, которая может быть извне доведена до завершения прежде, чем она самостоятельно завершится обычным путем. Механизм прерывания, обеспечиваемый Thread и поддерживаемый Thread.sleep() и Object.wait(), является механизмом отмены; он позволяет одному потоку запрашивать, чтобы другой поток прекратил выполнение того, что он делал ранее. Когда метод выдает InterruptedException, то он говорит вам, что если поток, выполняющий метод, прерван, то он предпримет попытку прекратить то, что он делает, вернуться на ранний этап, и указать его досрочный возврат путем выдачи InterruptedException. Рабочие методы библиотек блокирования должны реагировать на прерывание и выдавать InterruptedException, так, чтобы их можно было использовать в отменяемых функциях без риска для ответной реакции.

### **Прерывание потока**

Каждый поток имеет связанное с ним булево свойство, которое отображает его статус прерывания. Статус прерывания изначально имеет значение false; когда поток прерывается каким-либо другим потоком путем вызова Thread.interrupt(), то происходит одно из двух. Если другой поток выполняет прерываемый метод блокирования низкого уровня, такой как Thread.sleep(), Thread.join() или Object.wait(), то он разблокируется и выдает InterruptedException. Иначе, interrupt() просто устанавливает статус прерывания потока. Код, действующий в прерванном потоке, может позднее обратиться к статусу прерывания, чтобы посмотреть, был ли запрос на прекращение выполняемого действия; статус прерывания может быть прочитан с помощью Thread.isInterrupted(), и может быть прочитан и сброшен за одну операцию при помощи неудачно названного Thread.interrupted().

Прерывание является коллективным механизмом взаимодействия (cooperative mechanism). Когда один поток прерывает другой, прерванный поток не обязательно немедленно прекратит делать то, что он делал. Напротив, прерывание является вежливым способом, чтобы попросить другой поток прекратить то, что он делает, если он не против, когда ему это будет удобно. Некоторые методы, например, Thread.sleep(), серьезно относятся к такой просьбе, но методы не обязаны обращать внимание на прерывание. Неблокирующие методы, выполнение которых занимает много времени, могут положительно отнестись к просьбам о прерывании, путем опроса статуса прерывания, и в случае прерывания вернуться к более раннему этапу. Вы можете проигнорировать запрос о прерывании, но это может нанести ущерб ответной реакции.

Одним из преимуществ коллективной природы прерывания является то, что оно обеспечивает большую гибкость для безопасного построения отменяемых функций. Редко требуется немедленно остановить какую-либо функцию; структуры программных данных могут находиться в противоречивом состоянии, если функции были отменены во время обновления. Прерывание позволяет отменяемой функции завершить любую производящуюся работу, восстановить инварианты, уведомить другие функции об отмене, а затем прекратить работу.

## Работа с InterruptedException

Если выдача InterruptedException означает, что метод является блокирующим, то вызов метода блокирования означает, что ваш метод также является блокирующим, и у вас должна быть стратегия для работы с InterruptedException. Зачастую наиболее простой стратегией является генерирование собственного InterruptedException, как показано в методах putTask() и getTask() в Листинге 1. Выполняя это, вы также делаете ваш метод восприимчивым к прерыванию, и это для этого требуется всего-навсего добавить InterruptedException в вашу конструкцию throws.

##### **Листинг 1. Распространение InterruptedException на вызывающие операторы без его перехвата**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class TaskQueue {      private static final int MAX\_TASKS = 1000;        private BlockingQueue<Task> queue          = new LinkedBlockingQueue<Task>(MAX\_TASKS);        public void putTask(Task r) throws InterruptedException {          queue.put(r);      }        public Task getTask() throws InterruptedException {          return queue.take();      }  } |

Иногда необходимо произвести некоторую очистку, прежде чем распространить исключение. В этом случае вы можете перехватить InterruptedException, выполнить очистку, а затем повторно сгенерировать исключение. В Листинге 2, механизме подбора игроков в онлайновом игровом портале, показана именно такая технология. Метод matchPlayers() ожидает прибытия двух игроков, а затем начинает новую игру. Если она прервана после того, как появился один игрок, но до появления второго, то он ставит игрока в конец очереди, прежде чем повторно выдать InterruptedException, так, чтобы запрос игрока не был утерян.

##### **Листинг 2. Выполнение специализированной (task-specific) очистки перед повторной выдачей InterruptedException**

|  |  |
| --- | --- |
| 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 | public class PlayerMatcher {      private PlayerSource players;        public PlayerMatcher(PlayerSource players) {          this.players = players;      }        public void matchPlayers() throws InterruptedException {          try {               Player playerOne, playerTwo;               while (true) {                   playerOne = playerTwo = null;                   // Wait for two players to arrive and start a new game                   playerOne = players.waitForPlayer(); // could throw IE                   playerTwo = players.waitForPlayer(); // could throw IE                   startNewGame(playerOne, playerTwo);               }           }           catch (InterruptedException e) {               // If we got one player and were interrupted, put that player back               if (playerOne != null)                   players.addFirst(playerOne);               // Then propagate the exception               throw e;           }      }  } |

### **Не поглощайте прерывания**

Иногда генерирование InterruptedException не является опцией, как в случае задачи, определяемой Runnable, которая вызывает прерываемый метод. В этом случае вы не можете повторно выдать InterruptedException, но вообще ничего не делать нельзя. Когда метод блокирования обнаруживает прерывание и выдает InterruptedException, то он очищает статус прерывания. Если вы перехватили InterruptedException, но не можете повторно его сгенерировать, то вы должны сохранить подтверждение того, что прерывание произошло так, чтобы вышестоящий в стеке вызовов код мог узнать о прерывании и среагировать на него, если он хочет это сделать. Эта задача выполняется с помощью вызова interrupt(), чтобы "повторно прервать" текущий поток, как показано в Листинге 3. По крайней мере, когда бы вы ни перехватили InterruptedException и не выдали его повторно, еще раз прервите текущий поток перед возвратом.

##### **Листинг 3. Восстановление статуса прерывания после перехвата InterruptedException**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class TaskRunner implements Runnable {      private BlockingQueue<Task> queue;        public TaskRunner(BlockingQueue<Task> queue) {          this.queue = queue;      }        public void run() {          try {               while (true) {                   Task task = queue.take(10, TimeUnit.SECONDS);                   task.execute();               }           }           catch (InterruptedException e) {               // Restore the interrupted status               Thread.currentThread().interrupt();           }      }  } |

Худшее, что вы можете сделать с InterruptedException - это его поглощение - перехват без повторной его выдачи и без переподтверждения статуса прерывания потока. Стандартный подход к работе с незапланированной ошибкой - перехватить ее и зарегистрировать - также расценивается как поглощение (swallowing) прерывания, поскольку вышестоящий в стеке вызовов код не сможет узнать об этом. (Регистрирование InterruptedException также просто бессмысленно, потому что к тому времени, когда человек прочитает журнал, будет слишком поздно что-либо предпринимать.) В Листинге 4 показан слишком типичный шаблон поглощения прерывания:

##### **Листинг 4. Поглощение прерывания - не делайте этого**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | // Don't do this  public class TaskRunner implements Runnable {      private BlockingQueue<Task> queue;        public TaskRunner(BlockingQueue<Task> queue) {          this.queue = queue;      }        public void run() {          try {               while (true) {                   Task task = queue.take(10, TimeUnit.SECONDS);                   task.execute();               }           }           catch (InterruptedException swallowed) {               /\* DON'T DO THIS - RESTORE THE INTERRUPTED STATUS INSTEAD \*/           }      }  } |

Если вы не можете повторно выдать InterruptedException, независимо от того, планируете ли вы работать с запросом прерывания, вам все-таки нужно повторно прервать текущий поток, поскольку единичный запрос прерывания может иметь множество "получателей." Стандартная реализация рабочего потока пула потоков (ThreadPoolExecutor) реагирует на прерывание, следовательно, прерывание задачи, выполняющейся в пуле потока, может влиять как на отмену задачи, так и на уведомление работающего потока о том, что пул потока завершает работу. Если задачей было поглотить запрос прерывания, рабочий поток может не узнать, что было запрошено прерывание, а это может отложить завершение работы приложения или службы.

## Реализация отменяемых задач

В спецификации языка не дается какой-либо конкретной семантики прерывания, но в больших программах сложно получить какую-нибудь иную семантику прерывания, кроме отмены. В зависимости от функции, пользователь может запросить отмену через GUI или через сетевой механизм, например JMX или Web-сервисы. Она также может запрашиваться логикой программы. Например, Web crawler (поисковый агент) может автоматически прекратить свою работу в случае обнаружения переполнения диска, или параллельный алгоритм может запустить множество потоков для поиска в различных областях пространства решения и отменить их, как только один из них его найдет.

То, что задача является отменяемой, не означает, что ей нужно немедленно среагировать на запрос прерывания. Для задач, выполняющих код в цикле, проверка на прерывание один раз за итерацию цикла является обычной. В зависимости от того, как долго выполняется цикл, это может занять некоторое время, прежде чем код задачи заметит, что поток был прерван (либо с помощью обращения к статусу прерывания с Thread.isInterrupted() или с помощью вызова метода блокирования). Если необходимо, чтобы задача была более "отзывчивой" (responsive), то она может чаще делать обращения к статусу прерывания. Методы блокирования обычно опрашивают статус прерывания немедленно при входе, генерируя InterruptedException, если оно настроено на улучшение "отзывчивости".

Единственный раз, когда допустимо поглотить прерывание, это когда вы знаете, что поток должен вот-вот завершить работу. Этот сценарий имеет место только когда класс, вызывающий прерываемый метод, является частью Thread, а не Runnable или универсального библиотечного кода, как показано в Листинге 5. Он создает поток, который перечисляет простые числа до тех пор, пока он не будет прерван, и позволяет потоку завершить работу после прерывания. Ищущий простые числа цикл проверяет наличие прерывания в двух местах: один раз, опрашивая метод isInterrupted() в заголовке цикла с проверкой условия, и еще раз - при вызове метода блокирования BlockingQueue.put().

##### **Листинг 5. Сообщения interrupt можно поглощать, если вы знаете, что поток скоро завершит работу**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class PrimeProducer extends Thread {      private final BlockingQueue<BigInteger> queue;        PrimeProducer(BlockingQueue<BigInteger> queue) {          this.queue = queue;      }        public void run() {          try {              BigInteger p = BigInteger.ONE;              while (!Thread.currentThread().isInterrupted())                  queue.put(p = p.nextProbablePrime());          } catch (InterruptedException consumed) {              /\* Allow thread to exit \*/          }      }        public void cancel() { interrupt(); }  } |

### **Непрерываемое блокирование**

Не все методы блокирования выдают InterruptedException. Классы входных и выходных потоков могут блокировать ожидание ввода-вывода для завершения, но они не выдают InterruptedException, а также не возвращаются на ранний этап, если их прервать. Тем не менее, в случае сокета I/O, если поток закрывает сокет, блокирование операции ввода-вывода данного сокета в других потоках завершится раньше с исключением SocketException. Неблокирующие I/O-классы в java.nio также не поддерживают прерываемый ввод-вывод, но операции блокирования можно отменить сходным образом, закрыв канала или запросив активацию в Selector. Аналогично, попытка получить внутреннюю блокировку (вводом блока synchronized) не может быть прервана, но ReentrantLock поддерживает прерываемый режим захвата.

### **Неотменяемые задачи**

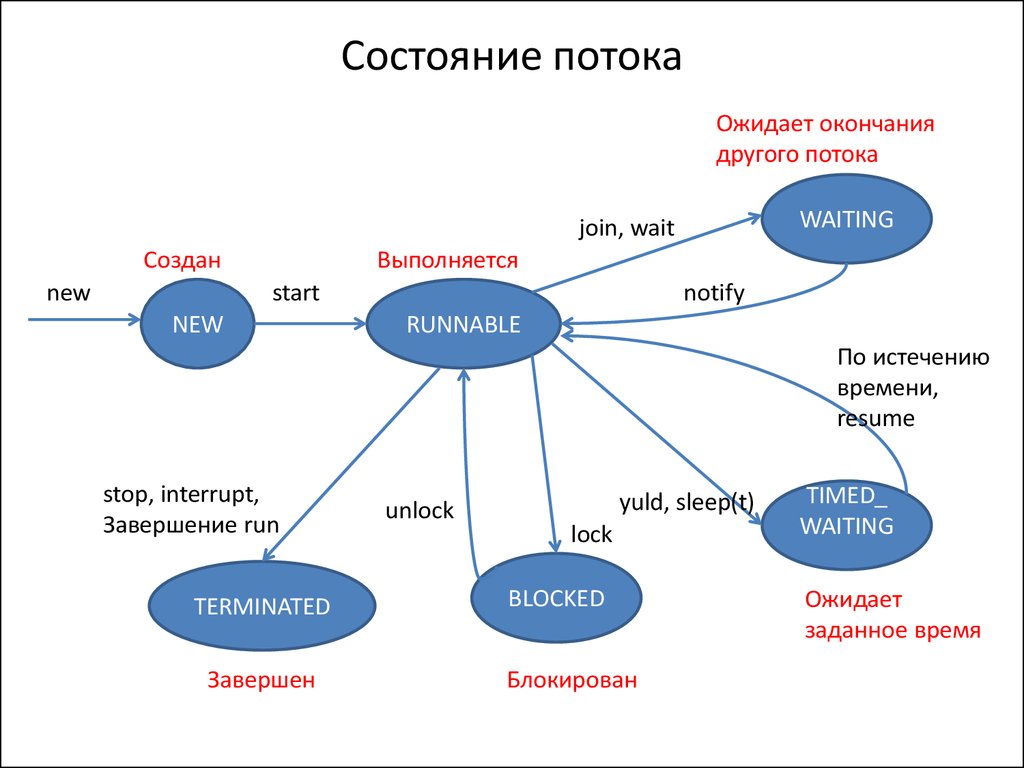
Некоторые задачи просто отказываются быть прерванными, что делает их неотменяемыми. Однако даже неотменяемые задачи должны попытаться сохранить статус прерывания на случай, если вышестоящий в стеке вызовов код захочет воздействовать на прерывание после завершения неотменяемой задачи. В Листинге 6 показан метод, ожидающий в очереди блокировок до тех пор, пока элемент не станет доступным, не обращая внимания на то, был ли он прерван. Как сознательный гражданин, он восстанавливает статус прерывания в финальном блоке после его завершения, так чтобы не лишить вызывающие операторы запроса прерывания. (Он не мог восстановить статус прерывания ранее, поскольку это привело бы к бесконечному циклу - BlockingQueue.take() могло бы опросить статус прерывания немедленно при входе и выдать InterruptedException, если он обнаружит установку статуса прерывания.)

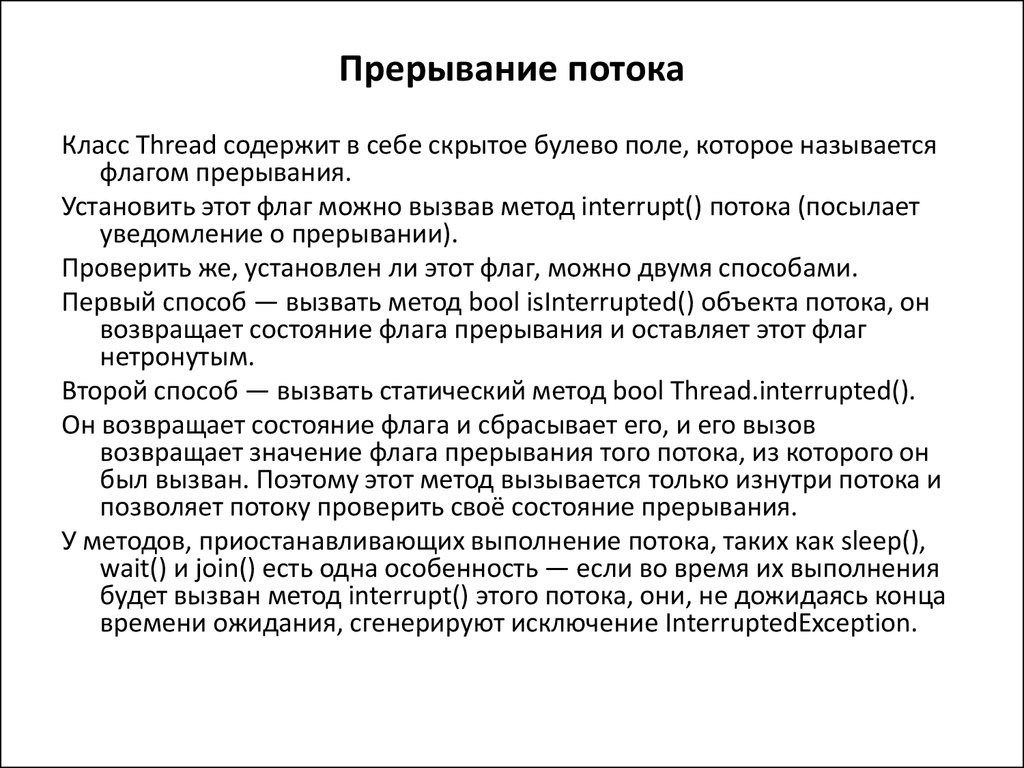
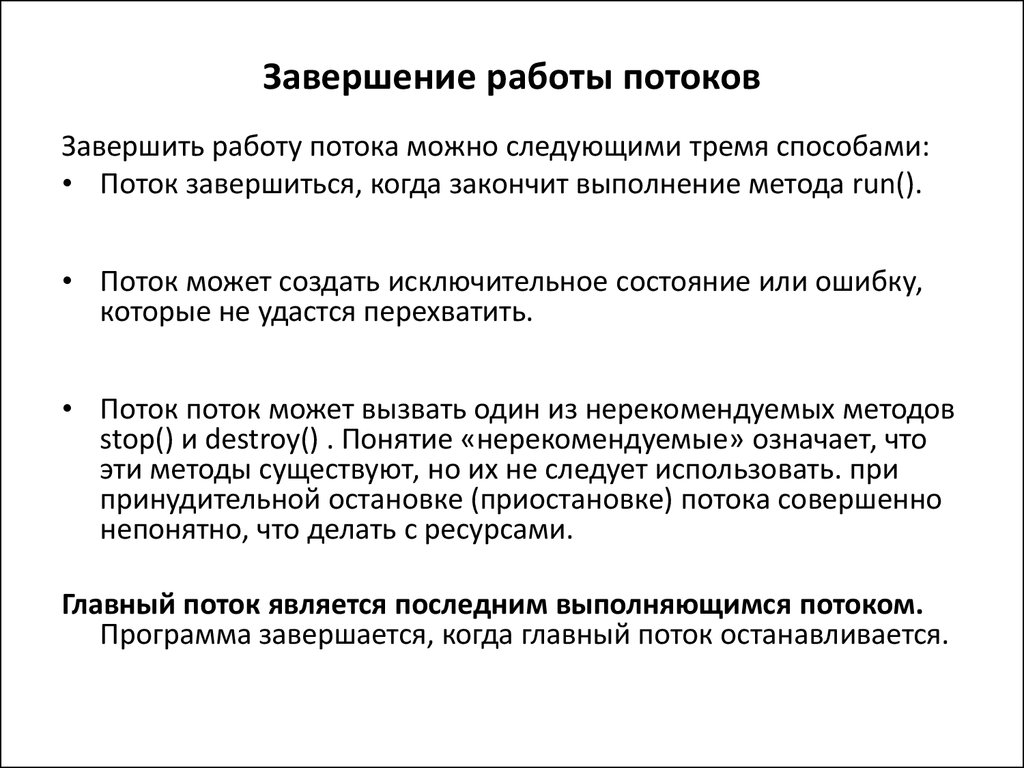
##### **Листинг 6. Неотменяемая задача, восстанавливающая статус прерывания перед возвратом**

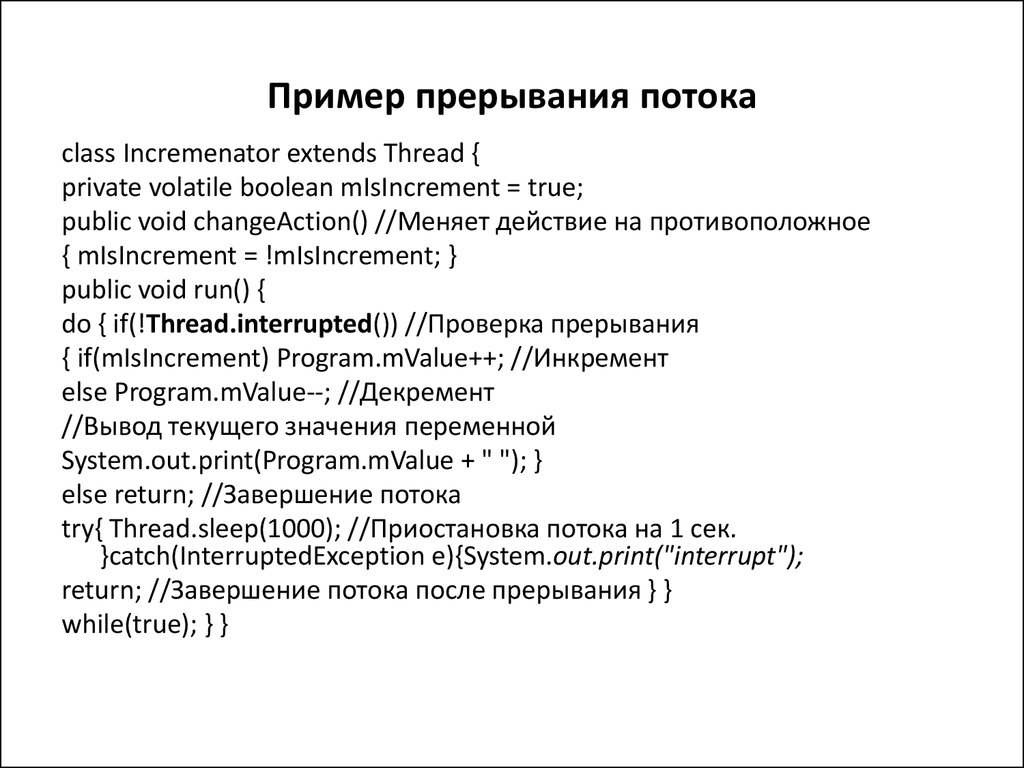
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public Task getNextTask(BlockingQueue<Task> queue) {      boolean interrupted = false;      try {          while (true) {              try {                  return queue.take();              } catch (InterruptedException e) {                  interrupted = true;                  // fall through and retry              }          }      } finally {          if (interrupted)              Thread.currentThread().interrupt();      }  } |

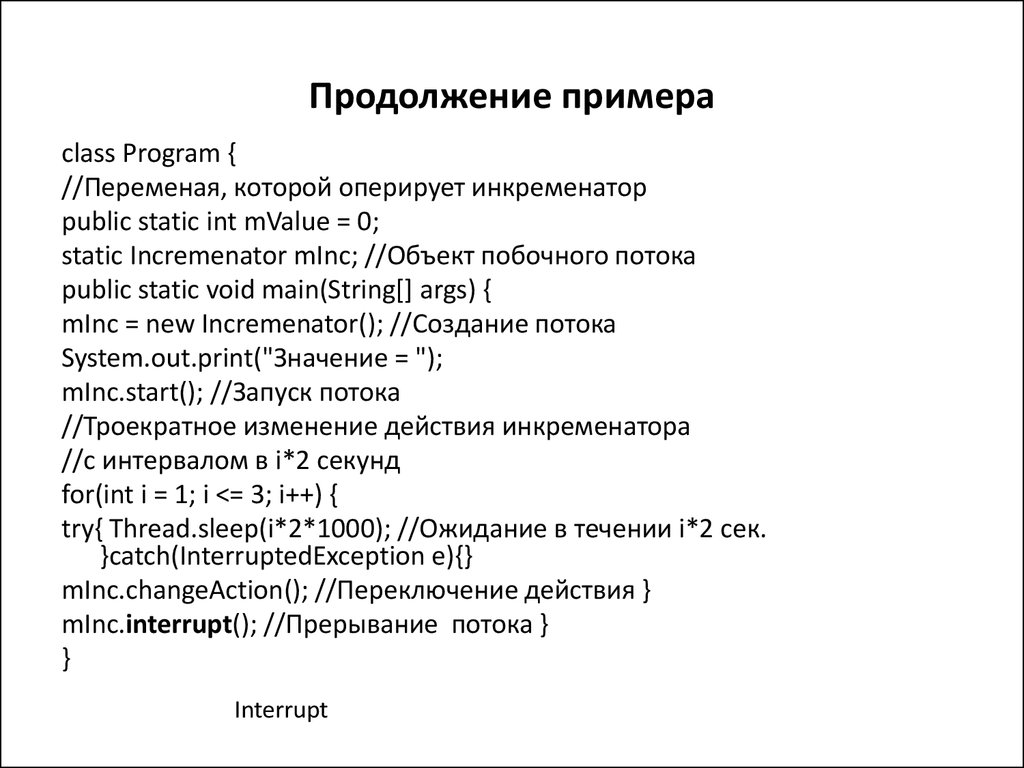
## Резюме

Вы можете использовать коллективный механизм прерывания, обеспечиваемый платформой Java для конструирования гибких алгоритмов отмены. Функции (activities) могут решать, являются ли они отменяемыми или нет, насколько им следует быть восприимчивыми к прерыванию, и они могут задержать прерывание для выполнения специализированной очистки, если немедленный возврат подвергнет риску целостность приложения. Даже если вы захотите полностью проигнорировать прерывание в вашем коде, убедитесь в восстановлении статуса прерывания, если вы перехватили InterruptedException, и не генерируйте его повторно, чтобы вызывающий его код не был лишен информации о том, что произошло прерывание.









#### **Interruption**

На данный момент в Java принят уведомительный порядок остановки потока (хотя JDK 1.0 и имеет несколько управляющих выполнением потока методов, например stop(), suspend() и resume() - в следующих версиях JDK все они были помечены как deprecated из-за потенциальных угроз взаимной блокировки).

Существует два способа которыми JVM уведомляет поток о том, что его прерывают.

Первый — это собственно InterruptedException.

Второй — это флаг потока INTERRUPT, который может быть получен при помощи метода Thread.isInterrupted().

По своей сути, InterruptedException сигнализирует о том, что поток просят завершить его работу. При этом вас не просят немедленно завершить свою работу. Вас просят корректно завершить работу. На это может понадобится некоторое время.

Для корректной остановки потока можно использовать метод класса Thread.interrupt().

Класс Thread содержит в себе скрытое булево поле, которое называется флагом прерывания. Установить этот флаг можно вызвав метод interrupt() потока. В дальнейшем состояние этого флага можно проверить двумя способами.

Первый способ — вызвать метод bool isInterrupted() объекта потока,

второй — вызвать статический метод bool Thread.interrupted() (для текущего потока).

Первый метод возвращает состояние флага прерывания и оставляет этот флаг нетронутым. Второй метод возвращает состояние флага и сбрасывает его. Заметьте что Thread.interrupted() — статический метод класса Thread, и его вызов возвращает значение флага прерывания того потока, из которого он был вызван. Поэтому этот метод вызывается только изнутри потока и позволяет потоку проверить своё состояние прерывания.

Механизм прерывания позволит нам решить проблему с засыпанием потока. У методов, приостанавливающих выполнение потока, таких как sleep(), wait() и join() есть одна особенность — если во время их выполнения будет вызван метод interrupt() этого потока, они, не дожидаясь конца времени ожидания (т.е. выйдут из состояния ожидания или спячки), сгенерируют исключение InterruptedException. Флаг в этом случае не выставляется.

Прерывание потока

■ public void interrupt() Изменяет статус потока на прерванный

■ public static boolean interrupted() Возвращает и очищает статус потока (прерван или нет)

■ public boolean isInterrupted() Возвращает статус потока (прерван или нет)

■ Поток должен в ходе своей работы проверять свой статус и корректно завершать работу, если его прервали

А если поток «спит»?

■ В том случае, если в текущий момент поток выполняет методы wait(), sleep(), join(), а его прерывают вызовом метода interrupt()…

■ метод прерывает свое выполнение с выбросом исключения InterruptedException !

■ Потоку не сообщается, что его прервали!

Схема действия при этом получается следующей:

* Реализовать поток.
* В потоке периодически проводить проверку статуса прерывания через вызов isInterrupted().
* Если состояние флага изменилось или было выброшено исключение во время ожидания/спячки, следовательно поток пытаются остановить извне.
* Принять решение – продолжить работу (если по каким-то причинам остановиться невозможно) или освободить заблокированные потоком ресурсы и закончить выполнение.

Возможная проблема, которая присутствует в этом подходе – блокировки на потоковом вводе-выводе. Если поток заблокирован на чтении данных - вызов interrupt() из этого состояния его не выведет. Решения тут различаются в зависимости от типа источника данных. Если чтение идет из файла – долговременная блокировка крайне маловероятна и тогда можно просто дождаться выхода из метода read(). Если же чтение каким-то образом связано с сетью – стоит использовать неблокирующий ввод-вывод из Java NIO.

Второй вариант реализации метода остановки (а также и приостановки) – сделать собственный аналог interrupt(). Т.е. объявить в классе потока флаги – на остановку и/или приостановку и выставлять их путем вызова заранее определённых методов извне. Методика действия при этом остаётся прежней – проверять установку флагов и принимать решения при их изменении. Недостатки такого подхода:

* Во-первых, потоки в состоянии ожидания таким способом не «оживить».
* Во-вторых, выставление флага одним потоком совсем не означает, что второй поток тут же его увидит. Для увеличения производительности виртуальная машина использует кеш данных потока, в результате чего обновление переменной у второго потока может произойти через неопределенный промежуток времени (хотя допустимым решением будет объявить переменную-флаг как volatile).

**В чем разница между interrupted() и isInterrupted()?**

Механизм прерывания работы потока в Java реализован с использованием внутреннего флага, известного как статус прерывания. Прерывание потока вызовом Thread.interrupt() устанавливает этот флаг. Методы Thread.interrupted() и isInterrupted() позволяют проверить, является ли поток прерванным.

Когда прерванный поток проверяет статус прерывания, вызывая статический метод Thread.interrupted(), статус прерывания сбрасывается.

Нестатический метод isInterrupted() используется одним потоком для проверки статуса прерывания у другого потока, не изменяя флаг прерывания.

Для наглядности протестируйте эти три варианта кода:

Пример Test1 бесконечный цикл

**public** **static** **void** main(String[] args) {

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

**while** (1 == 1) {

System.out.println(Thread.currentThread().isInterrupted());

Thread.currentThread().interrupt();

System.out.println(Thread.currentThread().isInterrupted())

}

}

}).start();

}

Не проверяем в цикле interrupted поток или нет (а он таковым становится при первом же шаге цикла, но поток продолжит работать бесконечно)

Пример Test2 проверка флага прерывания

**public** **static** **void** main(String[] args) {

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

**while** (!Thread.currentThread().isInterrupted()) {

System.out.println(Thread.currentThread().isInterrupted());

Thread.currentThread().interrupt();

System.out.println(Thread.currentThread().isInterrupted());

}

}

}).start();

}

Здесь interrupted - условие цикла, соответственно как только поток становится interrupted цикл завершается, run тоже, поток заканчивает работу.

Пример Test3 проверка флага прерывания с его сбросом в цикле методом interrupted()

**public** **static** **void** main(String[] args) {

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

**while** (!Thread.currentThread().isInterrupted()) {

System.out.println(Thread.currentThread().isInterrupted());

Thread.currentThread().interrupt();

// изменение состояния флага

System.out.println(Thread.currentThread().interrupted());

}

}

}).start();

}

Хотя здесь interrupted - условие цикла, соответственно, как только поток становится interrupted цикл должен завершится, однако метод , interrupted() при проверке соcтаяния меняет флаг прерывания на false, поток заканчивает работу.

Важно понимать, что interrupt адресуется не только вам (коду, который выполняется в потоке), но и владельцу потока. Другими словами, не только вы в этом потоке заинтересованы в том, чтобы определить что наступило прерывание.

**try** **{**

Object o **=** queue**.**take**();**

**}** **catch** **(** InterruptedException e **)** **{}**

Данный код неверен, потому что он “давит” сигнал прерывания. Если этот код выполняется в пуле потоков, то на этой задаче worker (который вызвал вас) должен был бы завершить исполнение задач, так как поток получил прерывание. Но этого не произойдет потому что сигнал прерывания был перехвачен вышеприведенным кодом. Как правило, это выражается в том, что в java машине остаются висеть потоки из-за чего JVM не может завершить свою работу.

Следующий код так же некорректен, потому что мы “маскируем” interrupt в исключительную ситуацию другого типа. Тем самым он не дает возможности вызывающей стороне зафиксировать ситуацию прерывания.

**try** **{**

Object o **=** queue**.**take**();**

**}** **catch** **(** InterruptedException e **)** **{**

**throw** **new** RuntimeException**(**e**);**

**}**

Корректный код будет выглядеть следующим образом.

**try** **{**

Object o **=** queue**.**take**();**

**}** **catch** **(** InterruptedException e **)** **{**

Thread**.**currentThread**().**interrupt**();**

**}**

Здесь мы ловим InterruptedException и выставляем флаг потока сигнализирующий о прерывании.

Политика прерывания метода является частью его контракта. Если вы декларируете что InterruptedException не может произойти во время выполнения метода, но сами при этом вызываете методы генерирующие эту исключительную ситуацию, то вы должны корректно конвертировать InterruptedException во флаг потока о прерывании. То же самое относится и к вызывающей стороне. Если вы вызываете метод, который по контракту генерирует InterruptedException, то вы должны ожидать что метод будет уведомлять вас о прерывании посредством исключительной ситуации. Если нет, то посредством флага.

Так зачем два способа сигнализации? Разве нельзя обойтись одним?”.

## Зачем нужен Thread.isInterrupted()?

InterruptedException бывает неудобен по причине того, что это checked exception. Это значит что компилятор заставляет вас или обработать его по месту, или указать в своем контракте. В случае, если вы не можете указать InterruptedException в своем контракте (например, вы имплементируете интерфейс где в контракте InterruptedException не указан), флаг потока остается единственным способом передать вызывающей стороне информацию о прерывании.

## Зачем нужен InterruptedException?

InterruptedException позволяет прервать поток уже выполняющий блокирующий вызов. В случае, если метод уже выполняется, то существует только один способ прервать его выполнение без возврата какого-либо значения и не нарушая при этом его контракт, — сгенерировать исключительную ситуацию. В этом случае возвращаемое значение метода просто неопределено.

### Потоки-демоны

В Java процесс завершается тогда, когда завершается последний его поток. Даже если метод main() уже завершился, но еще выполняются порожденные им потоки, система будет ждать их завершения.

Однако это правило не относится к особому виду потоков - демонам. Если завершился последний обычный поток процесса, и остались только потоки-демоны, то они будут принудительно завершены и выполнение процесса закончится. Чаще всего потоки-демоны используются для выполнения фоновых задач, обслуживающих процесс в течение его жизни.

Сделать поток демоном очень просто - достаточно установить соответствующий флаг с помощью метода **setDaemon()**.

**public** **class** DaemonThread **extends** Thread {  
  
    @Override  
    **public** **void** run() {  
        **try** {  
            TimeUnit.***SECONDS***.sleep(3);  
            System.out.println(**"Поток-демон завершён"**);  
        } **catch** (InterruptedException e) {  
            Thread.currentThread().interrupt();  
        }  
    }  
  
    **public** **static** **void** main(String[] args) {  
        Thread thread = **new** DaemonThread();  
        thread.setDaemon(**true**);  
        thread.start();  
        System.out.println(**"Основной поток завершён"**);  
    }  
}

Несмотря на то, что поток-демон останавливается на три секунды, мы эти три секунды ждать не будем и программа завершит свою работу. При это исключение **InterruptedException** не возникает.

### Как остановить поток

Остановить поток можно с помощью метода **interrupt()**. В случае, если поток не является демоном, то возникает исключение InterruptedException, о котором мы говорили выше.

**public** **class** ThinkerThread **extends** Thread {  
  
    @Override  
    **public** **void** run() {  
        **try** {  
            TimeUnit.***SECONDS***.sleep(3);  
            System.out.println(**"Второй поток завершён"**);  
        } **catch** (InterruptedException e) {  
            Thread.currentThread().interrupt();  
        }  
    }  
  
    **public** **static** **void** main(String[] args) {  
        Thread thread = **new** ThinkerThread();  
        thread.start();  
        System.out.println(**"Основной поток завершён"**);  
        thread.interrupt();  
    }  
}

Но вызов interrupt() следует толковать не как принудительную остановку, а как вежливый запрос. Немного изменим пример, чтобы показать, как должна обрабатываться остановка. Для имитации тяжёлой задачи сделаем цикл в котором будем вычислять тригонометрическую функцию.

**public** **class** ThinkerThread **extends** Thread {  
  
    @Override  
    **public** **void** run() {  
        **while** (!Thread.currentThread().isInterrupted()) {  
            Math.cos(Integer.***MAX\_VALUE***);  
        }  
        Thread.currentThread().interrupt();  
        System.out.println(**"Второй поток завершён"**);  
    }  
  
    **public** **static** **void** main(String[] args) {  
        Thread thread = **new** ThinkerThread();  
        thread.start();  
        System.out.println(**"Основной поток завершён"**);  
        thread.interrupt();  
    }  
}

Перед каждой итерацией цикла мы проверяем флаг **Thread.currentThread().isInterrupted()**. И если он оказывается установлен, мы прерываем наши вычисления. Таким образом, мы корректно обрабатываем флаг остановки потока.