**1 Какие приоритеты нитей бывают?**  
Ответ на этот вопрос есть в лекциях JavaRush.  
Для оптимизации параллельной работы нитей в Java имеется возможность устанавливать приоритеты нитей. Нити с большим приоритетом имеют преимущество в получении времени процессора перед нитями с более низким приоритетом.  
Работа с приоритетами обеспечивается следующими методами класса Thread:  
public final void setPriority(int newPriority)  
Устанавливает приоритет нити.  
public final int getPriority()  
Позволяет узнать приоритет нити.  
Значение параметра в методе setPriority не может произвольным. Оно должно находиться в пределах от MIN\_PRIORITY до MAX\_PRIORITY. При своем создании нить имеет приоритет NORM\_PRIORITY.  
MIN\_PRIORITY = 1.  
NORM\_PRIORITY =5.  
MAX\_PRIORITY = 10.   
**2 Можно ли остановить нить, снизив ее приоритет до 0?**  
Ответ в статье: «Топ 50 вопросов на собеседовании. Тема: Многопоточность (Multithreading)»  
На форуме нашел.  
Есть вариант этой статьи на английском языке:   
[www.javacodegeeks.com/2014/07/top-50-java-thread-interview-questions-answers-for-freshers-experienced-programmers.html](https://www.javacodegeeks.com/2014/07/top-50-java-thread-interview-questions-answers-for-freshers-experienced-programmers.html)   
Java предоставляет богатые API для всего, но, по иронии судьбы, не предоставляет удобных способов остановки нити. В JDK 1.0 было несколько управляющих методов, например stop(), suspend() и resume(), которые были помечены как deprecated в будущих релизах из-за потенциальных угроз взаимной блокировки, с тех пор разработчики Java API не предприняли попыток представить стойкий, ните-безопасный и элегантный способ остановки нитей. Программисты в основном полагаются на факт того, что нить останавливается сама, как только заканчивает выполнять методы run() или call(). Для остановки вручную, программисты пользуются преимуществом volatile boolean переменной и проверяют её значение в каждой итерации, если в методе run() есть циклы, или прерывают нити методом interrupt() для внезапной отмены заданий.  
Конкретно по вопросу: Нигде не видел, чтобы кто — то приоритет выставлял в 0.  
Если кто знает об этом что-нибудь, то напишите в комментариях.  
  
**3 Зачем нужен класс ThreadGroup?**  
  
ThreadGroup представляет собой набор нитей, которые так же могут содержать в себе другие группы потоков. Группа нитей образует дерево, в котором каждая другая группа нитей имеет родителя (кроме исходной). Поток имеет право доступа к данным из своей группы нитей, но не имеет такого доступа к другим группам или к родительской группе потоков.  
**4 В какой группе нитей состоит main-thread?**  
Нигде не нашел)) Подскажите где это есть))  
**5 Что такое паттерн ThreadPool?**  
На это есть выдержка из статьи на википедии:  
In computer programming, the thread pool pattern (also replicated workers or worker-crew model) is where a number of threads are created to perform a number of tasks, which are usually organized in a queue. The results from the tasks being executed might also be placed in a queue, or the tasks might return no result (for example, if the task is for animation). Typically, there are many more tasks than threads. As soon as a thread completes its task, it will request the next task from the queue until all tasks have been completed. The thread can then terminate, or sleep until there are new tasks available.  
The number of threads used is a parameter that can be tuned to provide the best performance. Additionally, the number of threads can be dynamic based on the number of waiting tasks. For example, a web server can add threads if numerous web page requests come in and can remove threads when those requests taper down. The cost of having a larger thread pool is increased resource usage. The algorithm used to determine when to create or destroy threads will have an impact on the overall performance:  
— create too many threads, and resources are wasted and time also wasted creating any unused threads  
— destroy too many threads and more time will be spent later creating them again  
— creating threads too slowly might result in poor client performance (long wait times)  
В компьютерном программировании есть модель пула потоков, где определенное число потоков создается для выполнения целого ряда задач, которые обычно организуются в очереди. Результаты от выполненных задач также могут быть помещены в очередь, либо задачи могут не возвращать никакого результата (например, если задача для анимации).   
Как правило, существует гораздо больше задач, чем потоков. Как только поток завершит свою задачу, он будет запрашивать следующую задачу из очереди, пока все задачи не будут завершены. Поток может затем прерваться или заснуть. Количество используемых потоков, это параметр, который может быть настроен, для обеспечения наилучшей производительности. Кроме того, число потоков может быть динамическим на основе количества возникающих задач. Например, веб-сервер может добавлять потоки, если запросы многочисленных веб-страниц приходят и может удалить потоки, когда этих запросов становится меньше. С увеличением размера пула потоков увеличивается использование ресурсов компьютера. Алгоритм, используемый для определения того, когда создавать или уничтожать потоки, будет иметь влияние на общую производительность: — Создать слишком много потоков значит тратить ресурсы и время впустую.   
— Уничтожить слишком много потоков и больше времени будет потрачено позже снова для их создания — Создание потоков слишком медленно, может привести к снижению производительности клиента.  
  
**6 Зачем нужен класс ThreadPoolExecutor?**  
public class ThreadPoolExecutor extends AbstractExecutorService  
ExecutorService это выполняет каждую представленную задачу, используя один возможно из нескольких объединенных в пул потоков, обычно сконфигурированное использование Executors методы фабрики.  
Пулы потоков рассматривают две различных проблемы: они обычно обеспечивают улучшенную производительность, выполняя большие количества асинхронных задач, из-за уменьшенных издержек вызова на задачу, и они обеспечивают средство ограничения и управления ресурсами, включая потоки, использованные, выполняя набор задач. Каждый ThreadPoolExecutor также поддерживает немного основной статистики, такой как число завершенных задач.  
Чтобы быть полезным через широкий диапазон контекстов, этот класс обеспечивает много корректируемых параметров и рычагов расширяемости. Однако, программистов убеждают использовать более удобное Executors методы фабрики Executors.newCachedThreadPool() (неограниченный пул потоков, с автоматическим восстановлением потока), Executors.newFixedThreadPool(int) (пул потоков фиксированного размера) и Executors.newSingleThreadExecutor() (единственный фоновый поток), которые предварительно конфигурируют настройки для наиболее распространенных сценариев использования.   
  
**7 Сколько способов создать нить вы знаете?**  
  
На уровне языка есть два способа создания нити. Объект класса java.lang.Thread представляет собой нить, но ей требуется задача для исполнения, которая является объектом, реализующим интерфейс java.lang.Runnable. Так как класс Thread реализует интерфейс Runnable, вы можете переопределить метод run() унаследовав ваш класс от Thread или реализовав в нём интерфейс Runnable.  
  
**8 Для чего используется класс Future?**  
  
Future хранит результат асинхронного вычисления. Вы можете запустить вычисление, предоставив кому-либо объект Future, и забыть о нем. Владелец объекта Future может получить результат, когда он будет готов.  
**9 В чем преимущества Callable над Runnable?**  
Ссылка: [www.ibm.com/developerworks/ru/library/l-java\_universe\_multithreading\_tasks/index.html](http://www.ibm.com/developerworks/ru/library/l-java_universe_multithreading_tasks/index.html)  
Интерфейс Callable гораздо больше подходит для создания задач, предназначенных для параллельного выполнения, нежели интерфейс Runnable или тем более класс Thread. При этом стоит отметить, что возможность добавить подобный интерфейс появилась только начиная с версии Java 5, так как ключевая особенность интерфейса Callable – это использование параметризованных типов (generics), как показано в листинге.

Листинг Создание задачи с помощью интерфейса Callable  
10      1 import java.util.concurrent.Callable;  
11      2 public class CallableSample implements Callable<String>{  
12      3     public String call() throws Exception {  
13      4         if(какое-то условие) {  
14      5             throw new IOException("error during task processing");  
15      6         }  
16      7         System.out.println("task is processing");  
17      8         return "result ";  
18      9     }  
19      10 }

Сразу необходимо обратить внимание на строку 2, где указано, что интерфейс Callable является параметризованным, и его конкретная реализация – класс CallableSample, зависит от типа String. На строке 3 приведена сигнатура основного метода call в уже параметризованном варианте, так как в качестве типа возвращаемого значения также указан тип String. Фактически это означает, что была создана задача, результатом выполнения которой будет объект типа String (см. строку 8). Точно также можно создать задачу, в результате работы которой в методе call будет создаваться и возвращаться объект любого требуемого типа. Такое решение значительно удобнее по сравнению с методом run в интерфейсе Runnable, который не возвращает ничего (его возвращаемый тип –void) и поэтому приходится изобретать обходные пути, чтобы извлечь результат работы задачи.  
Еще одно преимущество интерфейса Callable – это возможность «выбрасывать» исключительные ситуации, не оказывая влияния на другие выполняющиеся задачи. На строке 3 указано, что из метода может быть «выброшена» исключительная ситуация типа Exception, что фактически означает любую исключительную ситуацию, так как все исключения являются потомками java.lang.Exception. На строке 5 эта возможность используется для создания контролируемой (checked) исключительной ситуации типа IOException. Метод run интерфейса Runnable вообще не допускал выбрасывания контролируемых исключительных ситуаций, а выброс неконтролируемой (runtime) исключительной ситуации приводил к остановке потока и всего приложения.  
  
**10 Можно ли отменить выполнение задачи, если использовать класс Future?**  
Исходя из этой дискуссии, поднятой на хабре, выходит, что нельзя.   
[habrahabr.ru/post/133413/](https://habrahabr.ru/post/133413/)  
У Future есть метод Future.cancel(boolean), который должен отменить выполнение задачи. Но если задача уже начала выполняться, вызов Future.cancel(true) на самом деле не остановит ее. В недрах реализацииFutureTask выполняется код:

if (mayInterruptIfRunning) {  
Thread r = runner;  
if (r != null)  
r.interrupt(); }

Т.е. опять потоку, в котором выполняется задача, всего лишь рекомендуется прекратить выполнение. К тому же, мы не имеем даже возможности узнать выполняется ли задача в данный момент или нет. Есть, метод Future.isDone(), но опять мимо, он возвращает true не только когда задача завершила выполнение, а сразу после вызова Future.cancel(), даже если задача все еще выполняется (ведь Future.cancel(true) не останавливает задачу которая уже начала выполняться).   
  
Хорошо, если мы сами пишем весь код, тогда можно в нужных местах аккуратно обрабатывать Thread.isInterrupted() и все будет ОК. Но если мы запускаем сторонний код? Если у нас есть сервер расширяемый с помощью плагинов? Какой-нибудь криво написанный плагин может запросто привести к неработоспособному состоянию весь сервер ведь мы не можем корректно прервать выполнение зависшего плагина.