Оглавление

Многопоточность	
1.Чем процесс отличается от потока?	**Процесс — экземпляр программы во время выполнения, независимый объект, которому выделены системные ресурсы** (например, процессорное время и память). **Каждый процесс выполняется в отдельном адресном пространстве: один процесс не может получить доступ к переменным и структурам данных другого**. Если процесс хочет получить доступ к чужим ресурсам, необходимо использовать межпроцессное взаимодействие. Это могут быть конвейеры, файлы, каналы связи между компьютерами и многое другое. Для каждого процесса ОС создает так называемое «виртуальное адресное пространство», к которому процесс имеет прямой доступ. Это пространство принадлежит процессу, содержит только его данные и находится в полном его распоряжении. Операционная система же отвечает за то, как виртуальное пространство процесса проецируется на физическую память. **Поток**(thread) — **способ выполнения процесса, определяющий последовательность исполнения кода в процессе**. **Потоки всегда создаются в контексте какого-либо процесса, и вся их жизнь проходит только в его границах**. Потоки могут исполнять один и тот же код и манипулировать одними и теми же данными, а также совместно использовать описатели объектов ядра, поскольку таблица описателей создается не в отдельных потоках, а в процессах. Так как потоки расходуют существенно меньше ресурсов, чем процессы, в процессе выполнения работы выгоднее создавать дополнительные потоки и избегать создания новых процессов.
2.Чем Thread отличается от Runnable? Когда нужно использовать Thread, а когда Runnable? (Ответ что тред - это класс, а ранбл интерфейс - считается не полным, нужно рассказать подробно)	Thread - это класс, некоторая надстройка над физическим потоком. Runnable - это интерфейс, представляющий абстракцию над выполняемой задачей. Помимо того, что Runnable помогает разрешить проблему множественного наследования, несомненный плюс от его использования состоит в том, что он позволяет логически отделить логику выполнения задачи от непосредственного управления потоком. В классе Thread имеется несколько методов, которые можно переопределить в порожденном классе. Из них обязательному переопределению подлежит только метод run(). Этот же метод, безусловно, должен быть определен и при реализации интерфейса Runnable. Некоторые программисты считают, что создавать подкласс, порожденный от класса Thread, следует только в том случае, если нужно дополнить его новыми функциями. Так, если переопределять любые другие методы из класса Thread не нужно, то можно ограничиться только реализацией интерфейса Runnable. Кроме того, реализация интерфейса Runnable позволяет создаваемому потоку наследовать класс, отличающийся от Thread
3.Что такое монитор? Как монитор реализован в java?	**Монитор** - механизм синхронизации потоков, обеспечивающий доступ к неразделяемым ресурсам. Частью монитора является mutex, который встроен в класс Object и имеется у каждого объекта. Удобно представлять mutex как id захватившего его объекта. Если этот id равен 0 – ресурс свободен. Если не 0 – ресурс занят. Можно встать в очередь и ждать его освобождения. В Java **монитор реализован** с помощью ключевого слова **synchronized**.
4.Что такое синхронизация? Какие способы синхронизации существуют в java?	Синхронизация это **процесс**, **который позволяет выполнять потоки параллельно.** В Java все объекты имеют блокировку, благодаря которой только один поток одновременно может получить доступ к критическому коду в объекте. Такая

	синхронизация помогает предотвратить повреждение состояния объекта. Способы синхронизации в Java: **Системная синхронизация с использованием wait()/notify().** Поток, который ждет выполнения каких-либо условий, вызывает у этого объекта метод wait(), предварительно захватив его монитор. На этом его работа приостанавливается. Другой поток может вызвать на этом же самом объекте метод notify() (опять же, предварительно захватив монитор объекта), в результате чего, ждущий на объекте поток «просыпается» и продолжает свое выполнение. В обоих случаях монитор надо захватывать в явном виде, через synchronized-блок, потому как методы wait()/notify() не синхронизированы! **Системная синхронизация с использованием join()**. Метод join(), вызванный у экземпляра класса Thread, позволяет текущему потоку остановиться до того момента, как поток, связанный с этим экземпляром, закончит работу. **Использование классов из пакета java.util.concurrent.Locks -** механизмы синхронизации потоков, альтернативы базовым synchronized, wait, notify, notifyAll: **Lock, Condition, ReadWriteLock.**

5. Как работают методы wait(), notify() и notifyAll()?	**wait(): освобождает монитор** и **переводит вызывающий поток** в состояние **ожидания** до тех пор, **пока другой поток не вызовет метод notify()/notifyAll();** **notify(): продолжает** работу **потока**, у которого ранее был вызван метод wait(); **notifyAll(): возобновляет** работу **всех потоков**, у которых ранее был вызван метод wait(). Когда вызван метод wait(), поток освобождает блокировку на объекте и переходит из состояния Работающий (Running) в состояние Ожидания (Waiting). Метод **notify() подаёт сигнал** одному из **потоков**, **ожидающих на объекте**, **чтобы перейти в состояние Работоспособный (Runnable)**. При этом невозможно определить, какой из ожидающих потоков должен стать работоспособным. Метод notifyAll() заставляет все ожидающие потоки для объекта вернуться в состояние Работоспособный (Runnable). **Если ни один поток не находится в ожидании на методе wait(), то при вызове notify() или notifyAll() ничего не происходит.** wait(), notify() и notifyAll() **должны** вызываться **только из синхронизированного кода.**

6. В каких состояниях может находиться поток?	**New** - объект класса Thread создан, но еще не запущен. Он еще не является потоком выполнения и естественно не выполняется.
	Runnable - поток готов к выполнению, но планировщик еще не выбрал его.
	Running – поток выполняется.
	Waiting/blocked/sleeping - поток блокирован или поток ждет окончания работы другого потока.
	Dead - поток завершен. Будет выброшено исключение при попытке вызвать метод start() для dead потока.
	public enum State (У класса Thread есть внутренний класс State - состояние, а также метод public State getState().)
	{
	NEW, — поток создан, но еще не запущен;
	RUNNABLE, — поток выполняется;
	BLOCKED, — поток блокирован;
	WAITING, — поток ждет окончания работы другого потока;
	TIMED_WAITING, — поток некоторое время ждет окончания другого потока;
	TERMINATED; — поток завершен.
	}

7. Что такое семафор? Как он реализован в Java?	**Semaphore – это новый тип синхронизатора: семафор со счётчиком, реализующий шаблон синхронизации Семафор**. Доступ управляется с помощью счётчика: изначальное значение счетчика задается в конструкторе при создании синхронизатора, когда поток заходит в заданный блок кода, то значение счетчика уменьшается на единицу, когда поток его покидает, то увеличивается. Если значение счетчика равно нулю, то текущий поток блокируется, пока кто-нибудь не выйдет из защищаемого блока. Semaphore используется для защиты дорогих ресурсов, которые доступны в ограниченном количестве, например подключение к базе данных в пуле.
8. Что означает ключевое слово volatile? Почему операции над volatile переменными не атомарны?	**Переменная volatile является атомарной для чтения, но операции над переменной НЕ являются атомарными. Поля, для которых неприемлемо увидеть «несвежее» (stale) значение в результате кэширования или переупорядочения.** Если происходит какая-то операция, например, инкремент, то атомарность уже не обеспечивается, потому что сначала выполняется чтение(1), потом изменение(2) в локальной памяти, а затем запись(3). Такая операция не является атомарной и в неё может вклиниться поток по середине. Атомарная операция выглядит единой и неделимой командой процессора. **Переменная volatile находится в хипе, а не в кэше стека .**
9. Для чего нужны	volatile не гарантирует атомарность. Например, операция count++ не станет атомарной просто потому что count объявлена volatile. С другой стороны class AtomicInteger предоставляет атомарный метод для

Atomic типы данных? Чем отличаются от volatile?	выполнения таких комплексных операций атомарно, например getAndIncrement() – атомарная замена оператора инкремента, его можно использовать, чтобы атомарно увеличить текущее значение на один. Похожим образом сконструированы атомарные версии и для других типов данных.
10. Что такое потоки демоны? Для чего они нужны? Как создать поток-демон?	Потоки-демоны **работают в фоновом режиме вместе с программой**, но не являются неотъемлемой частью программы**. Если какой-либо процесс может выполняться на фоне работы основных потоков выполнения и его деятельность заключается в обслуживании основных потоков приложения,** то такой процесс может быть запущен как поток-демон с помощью метода **setDaemon(boolean value)**, вызванного у потока до его запуска. Метод boolean isDaemon() позволяет определить, является ли указанный поток демоном или нет. **Основной поток приложения может завершить выполнение потока-демона** (в отличие от обычных потоков) **с окончанием кода метода main(), не обращая внимания**, **что поток-демон еще работает.** Поток демон можно сделать только если он еще не запущен. Пример демона - GC.
11. Что такое приоритет потока? На что он влияет? Какой приоритет у потоков по умолчанию?	Приоритеты потоков используются планировщиком потоков для принятия решений о том, когда какому из потоков будет разрешено работать. Теоретически высокоприоритетные потоки получают больше времени процессора, чем низкоприоритетные. Практически объем времени процессора, который получает поток, часто зависит от нескольких факторов помимо его приоритета(является ли поток демоном). Чтобы установить приоритет потока, используется метод класса Thread: final void setPriority(int level). Значение level изменяется в пределах от Thread.MIN_PRIORITY = 1 до Thread.MAX_PRIORITY = 10. Приоритет по умолчанию - Thread.NORM_PRIORITY = 5. Получить текущее значение приоритета потока можно вызвав метод: final int getPriority() у экземпляра класса Thread. Метод yield() можно использовать для того чтобы принудить планировщик выполнить другой поток, который ожидает своей очереди.
12. Как работает Thread.join()? Для чего он нужен?	Когда поток вызывает join(), он будет ждать пока поток, к которому он присоединяется, будет завершён, либо отработает переданное время: void join() void join(long millis) - с временем ожидания void join(long millis, int nanos) Применение: при распараллелили вычисления, вам надо дождаться результатов, чтобы собрать их в кучу и продолжить выполнение.
13. Чем отличаются методы wait() и sleep()?	метод sleep() - приостанавливает поток на указанное время. Состояние меняется на WAITING, по истечению - RUNNABLE. метод wait() - меняет состояние потока на WAITING. Может быть вызван только у объекта владеющего блокировкой, в противном случае выкинется исключение IllegalMonitorStateException
14. Можно ли вызвать start() для одного потока дважды?	Нельзя стартовать поток больше, чем единожды. В частности, поток не может быть перезапущен, если он уже завершил выполнение. Выдает: IllegalThreadStateException

15. Как правильно остановить поток? Для чего нужны методы **.stop(), .interrupt(), .interrupted(), .isInterrupted().**	Как остановить поток? 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). **Для корректной остановки потока** можно использовать метод класса Thread - **interrupt().** Этот метод выставляет внутренний флаг-статус прерывания. В дальнейшем состояние этого флага можно проверить с помощью метода isInterrupted() или Thread.interrupted() (для текущего потока). **Метод interrupt() также способен вывести поток из состояния ожидания или спячки**. Т.е. если у потока были вызваны методы sleep() или wait() – текущее состояние прервется и будет выброшено исключение InterruptedException. Флаг в этом случае не выставляется. **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).
	Почему не рекомендуется использовать метод Thread.stop()? **При принудительной остановке (приостановке) потока, stop() прерывает поток в недетерменированном месте выполнения, в результате становится совершенно непонятно, что делать с принадлежащими ему ресурсами. Поток может открыть сетевое соединение - что в таком случае делать с данными, которые еще не вычитаны?** Где гарантия, что после дальнейшего запуска потока (в случае приостановки) он сможет их дочитать? Если поток блокировал разделяемый ресурс, то как снять эту блокировку и не переведёт ли принудительное снятие к нарушению консистентности системы? То же самое можно расширить и на случай соединения с базой данных: если поток остановят посередине транзакции, то кто ее будет закрывать? Кто и как будет разблокировать ресурсы?
	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.**

16. Чем Runnable отличается от Callable?	Интерфейс **Runnable** появился в Java 1.0, а интерфейс **Callable** был введен в Java 5.0 в составе библиотеки java.util.concurrent; **Классы, реализующие** интерфейс **Runnable** для выполнения задачи **должны реализовывать метод run(). Классы, реализующие интерфейс Callable - метод call();** Метод Runnable.**run() не возвращает** никакого значения, Callable - это параметризованный функциональный интерфейс. Callable.**call() возвращает Object, если он не параметризован, иначе указанный тип.** Метод **run() НЕ может выбрасывать проверяемые исключения, в то время как метод call() может.**
17. Что такое FutureTask?	FutureTask представляет собой **отменяемое асинхронное вычисление** в **параллельном потоке**. Этот **класс** предоставляет **базовую реализацию Future**, с методами для запуска и остановки вычисления, методами для запроса состояния вычисления и извлечения результатов. **Результат может быть получен только когда вычисление завершено**, **метод получения будет заблокирован, если вычисление ещё не завершено**. Объекты FutureTask могут быть использованы для обёртки объектов Callable и Runnable. Так как **FutureTask помимо Future реализует Runnable**, его можно передать в Executor на выполнение.
18. Что такое deadlock?	Взаимная блокировка (deadlock) - **явление** при котором **все потоки** находятся **в режиме ожидания и своё состояние не меняют.** Происходит, когда достигаются состояния: **взаимного исключения:** по крайней мере **один** ресурс занят в режиме неделимости и следовательно только один поток может использовать ресурс в данный момент времени. **удержания и ожидания:** поток удерживает как минимум один ресурс и запрашивает дополнительные ресурсы, которые удерживаются другими потоками. **отсутствия предочистки:** операционная система не переназначает ресурсы: если они уже заняты, они должны отдаваться удерживающим потокам сразу же. **цикличного ожидания:** поток ждет освобождения ресурса другим потоком, который в свою очередь ждёт освобождения ресурса заблокированного первым потоком. Простейший способ **избежать взаимной блокировки – не допускать цикличного ожидания.** Этого **можно достичь**, **получая мониторы разделяемых ресурсов в определенном порядке и освобождая их в обратном порядке.**
19. Что такое livelock?	livelock – **тип взаимной блокировки**, при котором **несколько потоков выполняют бесполезную работу**, **попадая** в **зацикленность при попытке получения** каких-либо **ресурсов**. При этом **их состояния** постоянно **изменяются** в **зависимости друг от друга.** Фактической **ошибки не возникает**, но КПД системы падает до 0. Часто возникает в результате попыток предотвращения deadlock. Реальный **пример** livelock, – когда два человека встречаются в узком коридоре и каждый, пытаясь быть вежливым, отходит в сторону, и так они бесконечно двигаются из стороны в сторону, абсолютно не продвигаясь в нужном им направлении.
20. Что такое race condition?	Состояние гонки (race condition) - **ошибка проектирования многопоточной системы или приложения**, **при которой работа зависит** от того**, в каком порядке выполняются потоки.** Состояние гонки **возникает когда поток, который должен исполнится в начале, проиграл гонку и первым исполняется другой поток:** поведение кода изменяется, из-за чего возникают недетерменированные ошибки. **DataRace** - это свойство выполнения программы. Согласно JMM, выполнение считается содержащим гонку данных, если оно содержит по крайней мере два конфликтующих доступа (чтение или запись в одну и ту

	же переменную), которые не упорядочены отношениями «happens before». **Starvation** - потоки не заблокированы, но есть нехватка ресурсов из-за чего потоки ничего не делают. Самый простой способ решения — копирование переменной в локальную переменную. Или просто синхронизация потоков методами и sync-блоками.
21. Что такое Фреймворк fork/join? Для чего он нужен?	Фреймворк Fork/Join, представленный в JDK 7, - это набор классов и интерфейсов позволяющих использовать преимущества многопроцессорной архитектуры современных компьютеров. **Он разработан для выполнения задач, которые можно рекурсивно разбить на маленькие подзадачи, которые можно решать параллельно.** **Этап Fork**: **большая задача разделяется** на **несколько меньших подзадач,** которые в свою очередь также разбиваются на меньшие. **И так до тех пор, пока задача не становится тривиальной и решаемой последовательным способом.** **Этап Join:** далее (опционально) идёт процесс «свёртки» - **решения подзадач некоторым образом объединяются пока не получится решение всей задачи.** Решение всех подзадач (в т.ч. и само разбиение на подзадачи) происходит параллельно. Для **решения некоторых** задач **этап Join не требуется. Например**, для параллельного QuickSort — массив рекурсивно делится на всё меньшие и меньшие диапазоны, пока не вырождается в тривиальный случай из 1 элемента. Хотя в некотором смысле Join будет необходим и тут, т.к. всё равно остаётся необходимость дождаться пока не закончится выполнение всех подзадач. Ещё одно **преимущество** этого фреймворка заключается в том, что он **использует work-stealing алгоритм**: **потоки**, которые **завершили** выполнение **собственных подзадач**, могут «**украсть**» **подзадачи у** других **потоков**, **которые всё ещё заняты.**
22. Что означает ключевое слово synchronized? Где и для чего может использоваться?	Зарезервированное слово позволяет добиваться синхронизации в помеченных им методах или блоках кода.
23. Что является монитором у статического synchronized-метода?	Объект типа Class, соответствующий классу, в котором определен метод.
24. Что является монитором у нестатического synchronized-метода?	Объект this
25. util. Concurrent поверхностно.	http://java-online.ru/concurrent.xhtml Классы и интерфейсы пакета java.util.concurrent объедининены в несколько групп по функциональному признаку: **collections** - Набор эффективно работающих в многопоточной среде

	коллекций. CopyOnWriteArrayList(Set), ConcurrentHashMap. Итераторы классов данного пакета представляют данные на определенный момент времени. Все операции по изменению коллекции (add, set, remove) приводят к созданию новой копии внутреннего массива. Этим гарантируется, что при проходе итератором по коллекции не будет ConcurrentModificationException.
	Отличие **ConcurrentHashMap** связано с внутренней структурой хранения пар key-value. ConcurrentHashMap использует несколько сегментов, и данный класс нужно рассматривать как группу HashMap'ов. Количество сегментов по умолчанию равно 16. Если пара key-value хранится в 10-ом сегменте, то ConcurrentHashMap заблокирует, при необходимости, только 10-й сегмент, и не будет блокировать остальные 15.
	CopyOnWriteArrayList: -volatile массив внутри -lock только при модификации списка, поэтому операции чтения очень быстрые -новая копия массива при модификации -fail-fast итератор -модификация через iterator невозможна - UnsupportedOperationException
	synchronizers - Объекты синхронизации, позволяющие разработчику управлять и/или ограничивать работу нескольких потоков. Содержит пять объектов синхронизации: semaphore, countDownLatch, ciclycBarrier, exchanger, phaser.
	CountDownLatch - объект синхронизации потоков, блокирующий один или несколько потоков до тех пор, пока не будут выполнены определенные условия. Количество условий задается счетчиком. При обнулении счетчика, т.е. при выполнении всех условий, блокировки выполняемых потоков будут сняты и они продолжат выполнение кода. Одноразовый. **CyclicBarrier** — барьерная синхронизация останавливает поток в определенном месте в ожидании прихода остальных потоков группы. Как только все потоки достигнут барьера, барьер снимается и выполнение потоков продолжается. Как и CountDownLatch, использует счетчик и похож на него. Отличие связано с тем, барьер можно использовать повторно(в цикле). **Exchanger** — объект синхронизации, используемый для двустороннего обмена данными между двумя потоками. При обмене данными допускается null значения, что позволяет использовать класс для односторонней передачи объекта или же просто, как синхронизатор двух потоков. Обмен данными выполняется вызовом метода exchange, сопровождаемый самоблокировкой потока. Как только второй поток вызовет метод exchange, то синхронизатор Exchanger выполнит обмен данными между потоками. **Phaser** — объект синхронизации типа «Барьер», но, в отличие от CyclicBarrier, может иметь несколько барьеров (фаз), и количество участников на каждой фазе может быть разным.
	atomic - Набор атомарных классов для выполнения атомарных операций. Операция является атомарной, если её можно безопасно выполнять при параллельных вычислениях в нескольких потоках, не используя при этом ни блокировок, ни синхронизацию synchronized. **Queues** - содержит классы формирования неблокирующих и блокирующих очередей для многопоточных приложений. Неблокирующие очереди «заточены» на скорость выполнения, блокирующие очереди приостанавливают потоки при работе с очередью. **Locks** - Механизмы синхронизации потоков, альтернативы базовым synchronized, wait, notify, notifyAll: Lock, Condition, ReadWriteLock.
	Lock — базовый интерфейс, предоставляющий более гибкий подход при ограничении доступа к ресурсам/блокам по сравнению с использованием synchronized. Так, при использовании нескольких блокировок, порядок их освобождения может быть произвольный. Имеется возможность перехода к альтернативному сценарию, если блокировка уже захвачена. Condition — интерфейсное условие в сочетании с блокировкой Lock

	позволяет заменить методы монитора/мьютекса (wait, notify и notifyAll) объектом, управляющим ожиданием событий. Объект с условием чаще всего получается из блокировок с использованием метода lock.newCondition(). Таким образом можно получить несколько комплектов wait/notify для одного объекта. Блокировка Lock заменяет использование synchronized, а Condition — объектные методы монитора. ReadWriteLock — интерфейс создания read/write блокировок, который реализует один единственный класс ReentrantReadWriteLock. Блокировку чтение-запись следует использовать при длительных и частых операциях чтения и редких операциях записи. Тогда при доступе к защищенному ресурсу используются разные методы блокировки, как показано ниже : ReadWriteLock rwl = new ReentrantReadWriteLock(); Lock readLock = rwl.readLock(); Lock writeLock = rwl.writeLock();
	Executors - включает средства, называемые сервисами исполнения, позволяющие управлять потоковыми задачами с возможностью получения результатов через интерфейсы Future и Callable.
	ExecutorService служит альтернативой классу Thread, предназначенному для управления потоками. В основу сервиса исполнения положен интерфейс Executor, в котором определен один метод : void execute(Runnable thread); При вызове метода execute исполняется поток thread.
26. Stream API & ForkJoinPool. Как связаны, что это такое.	В Stream API есть простой способ распараллеливания потока методом parallel() или parallelStream(), чтобы получить выигрыш в производительности на многоядерных машинах. По-умолчанию parallel stream используют ForkJoinPool.commonPool. Этот пул создается статически и живет пока не будет вызван System::exit. Если задачам не указывать конкретный пул, то они будут исполняться в рамках commonPool. По-умолчанию, размер пула равен на 1 меньше, чем количество доступных ядер. Когда некий тред отправляет задачу в common pool, то пул может использовать вызывающий тред (caller-thread) в качестве воркера. ForkJoinPool пытается загрузить своими задачами и вызывающий тред.
27. Java Memory Model	**Описывает как потоки должны взаимнодействовать через общую память.** Определяет набор действий межпоточного взаимодействия. В частности, чтение и запись переменной, захват и освобождений монитора, чтение и запись volatile переменной, запуск нового потока. JMM определяет отношение между этими действиями "happens-before" - абстракцей обозначающей, что если операция X связана отношением happens-before с операцией Y, то весь код следуемый за операцией Y, выполняемый в одном потоке, видит все изменения, сделанные другим потоком, до операции X.
	Можно выделить несколько основных областей, имеющих отношение к модели памяти: **Видимость** (visibility). Один поток может временно сохранить значения некоторых полей не в основную память, а в регистры или локальный кэш процессора, таким образом второй поток, читая из основной памяти, может не увидеть последних изменений поля. И наоборот, если поток на протяжении какого-то времени работает с регистрами и локальными кэшами, читая данные оттуда, он может сразу не увидеть изменений, сделанных другим потоком в основную память. К вопросу видимости имеют отношение следующие ключевые слов языка Java: synchronized, volatile, final.
	С точки зрения Java все переменные (за исключением локальных переменных, объявленных внутри метода) хранятся в heap памяти, которая доступна всем потокам. Кроме этого, каждый поток имеет локальную—рабочую—память, где он хранит копии переменных, с которыми он работает, и при выполнении программы поток работает только с этими копиями.

	synchronized - При входе в synchronized метод или блок поток обновляет содержимое локальной памяти, а при выходе из synchronized метода или блока поток записывает изменения, сделанные в локальной памяти, в главную. Такое поведение synchronized методов и блоков следует из правил для отношения «происходит раньше»
	volatile - запись volatile-переменных производится в основную память, минуя локальную. и чтение volatile переменной производится также из основной памяти, то есть значение переменной не может сохраняться в регистрах или локальной памяти потока и операция чтения этой переменной гарантированно вернёт последнее записанное в неё значение.
	final - после того как объект был корректно создан, любой поток может видеть значения его final полей без дополнительной синхронизации. «Корректно создан» означает, что ссылка на создающийся объект не должна использоваться до тех пор, пока не завершился конструктор объекта. Рекомендуется изменять final поля объекта только внутри конструктора, в противном случае поведение не специфицировано.
	Переупорядочивание (Reordering). Для увеличения производительности процессор/компилятор могут переставлять местами некоторые инструкции/операции. Процессор может решить поменять порядок выполнения операций, если, например, сочтет что такая последовательность выполнится быстрее. Эффект может наблюдаться, когда один поток кладет результаты первой операции в регистр или локальный кэш, а результат второй операции попадает непосредственно в основную память. Тогда второй поток, обращаясь к основной памяти может сначала увидеть результат второй операции, и только потом первой, когда все регистры или кэши синхронизируются с основной памятью. Также регулируется набором правил «happens-before»: операции чтения и записи volatile переменных не могут быть переупорядочены с операциями чтения и записи других volatile и не-volatile переменных.
	https://habr.com/ru/company/golovachcourses/blog/221133/