Оглавление

[Модель памяти Java 4](#_Toc48842505)

[Потокобезопасность 11](#_Toc48842506)

[Конкуренция и параллелизм 11](#_Toc48842507)

[Кооперативная многозадачность. Какой тип многозадачности использует Java? Чем обусловлен этот выбор? 12](#_Toc48842508)

[ordering, as-if-serial semantics, sequential consistency, visibility, atomicity, happens-before, mutual exclusion, safe publication? 13](#_Toc48842509)

[Процесс и поток 15](#_Toc48842510)

[Зеленые потоки 17](#_Toc48842511)

[Способы создания потока 18](#_Toc48842512)

[Thread и Runnable 19](#_Toc48842513)

[start() и run() 19](#_Toc48842514)

[Как принудительно запустить поток? 19](#_Toc48842515)

[Монитор 20](#_Toc48842516)

[private мьютекс 21](#_Toc48842517)

[Можно ли создавать новые экземпляры класса, пока выполняется static synchronized метод? 21](#_Toc48842518)

[На каком объекте происходит синхронизация при вызове static synchronized метода? 21](#_Toc48842519)

[Синхронизация 22](#_Toc48842520)

[Способы синхронизации 23](#_Toc48842521)

[Состояния потока 24](#_Toc48842522)

[Приоритет потока 25](#_Toc48842523)

[wait() и notify()/notifyAll() 26](#_Toc48842524)

[notify() и notifyAll() 27](#_Toc48842525)

[Почему методы wait() и notify() вызываются только в синхронизированном блоке? 28](#_Toc48842526)

[Чем отличается работа метода wait() с параметром и без параметра? 28](#_Toc48842527)

[Semaphore 28](#_Toc48842528)

[volatile (synchronized, transient, native) 29](#_Toc48842529)

[Различия volatile и Atomic-переменным 30](#_Toc48842530)

[В чем заключаются различия между java.util.concurrent.Atomic\*.compareAndSwap() и java.util.concurrent.Atomic\*.weakCompareAndSwap(). 30](#_Toc48842531)

[Потоки-демоны 31](#_Toc48842532)

[Можно ли сделать основной поток программы демоном? 31](#_Toc48842533)

[Thread.join() 32](#_Toc48842534)

[Thread.sleep() и Thread.yield() 32](#_Toc48842535)

[Что значит «усыпить» поток? 32](#_Toc48842536)

[Как остановить поток? 33](#_Toc48842537)

[В чем разница между interrupted() и isInterrupted()? 35](#_Toc48842538)

[Почему не рекомендуется использовать метод Thread.stop()? 36](#_Toc48842539)

[Runnable и Callable 37](#_Toc48842540)

[FutureTask 37](#_Toc48842541)

[deadlock (взаимная блокировка) 37](#_Toc48842542)

[livelock 38](#_Toc48842543)

[race condition (состояние гонки) 39](#_Toc48842544)

[Способы решения 39](#_Toc48842545)

[Фреймворк Fork/Join 40](#_Toc48842546)

[Как проверить, удерживает ли поток монитор определенного ресурса? 42](#_Toc48842547)

[!!! Что означает ключевое слово synchronized? Где и для чего может использоваться? 44](#_Toc48842548)

# Модель памяти Java

Модель памяти Java (Java Memory Model, JMM) описывает поведение потоков в среде исполнения Java. Это часть семантики языка Java, набор правил, описывающий выполнение многопоточных программ и правил, по которым потоки могут взаимодействовать друг с другом посредством основной памяти.

Формально модель памяти определяет набор действий межпоточного взаимодействия (эти действия включают в себя, в частности, чтение и запись переменной, захват и освобождений монитора, чтение и запись volatile переменной, запуск нового потока), а также модель памяти определяет отношение между этими действиями -happens-before — абстракции 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.

Существует несколько основных правил для отношения happens-before:

∙ В рамках одного потока любая операция happens-before любой операцией следующей за ней в исходном коде;

∙ Освобождение монитора (unlock) happens-before захват того же монитора (lock);

∙ Выход из synchronized блока/метода happens-before вход в synchronized блок/метод на том же мониторе;

∙ Запись volatile поля happens-before чтение того же самого volatile поля;

∙ Завершение метода run() экземпляра класса Thread happens-before выход из метода join() или возвращение false методом isAlive() экземпляром того же потока;

∙ Вызов метода start() экземпляра класса Thread happens-before начало метода run() экземпляра того же потока;

∙ Завершение конструктора happens-before начало метода finalize() этого класса;

∙ Вызов метода interrupt() на потоке happens-before обнаружению потоком факта, что данный метод был вызван либо путем выбрасывания исключения InterruptedException, либо с помощью методов isInterrupted() или interrupted().

∙ Связь happens-before транзитивна, т. е. если X happens-before Y, а Y happens-before Z, то X happens-before Z.

∙ Освобождение/захват монитора и запись/чтение в volatile переменную связаны отношением happens-before, только если операции проводятся над одним и тем же экземпляром объекта.

∙ В отношении happens-before участвуют только два потока, о поведении остальных потоков ничего сказать нельзя, пока в каждом из них не наступит отношение happens-before с другим потоком.

Можно выделить несколько основных областей, имеющих отношение к модели памяти:

Видимость (visibility). Один поток может в какой‑то момент временно сохранить значение некоторых полей не в основную память, а в регистры или локальный кэш процессора, т. о. второй поток, выполняемый на другом процессоре, 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 главной 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. Надо отметить, что это описание не требование к реализации, а всего лишь модель, которая объясняет поведение программы, так, в качестве локальной памяти не обязательно выступает кэш память, это могут быть регистры процессора или потоки могут вообще не иметь локальной памяти.

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»: т. к. все операции с памятью происходят раньше освобождения монитора и освобождение монитора происходит раньше захвата монитора, то все операции с памятью, которые были сделаны потоком до выхода из synchronized блока должны быть видны любому потоку, который входит в synchronized блок для того же самого монитора. Очень важно, что это правило работает только в том случае, если потоки синхронизируются, используя один и тот же монитор!

Что касается volatile переменных, то запись таких переменных производится в основную память, минуя локальную, и чтение volatile переменной производится также из основной памяти, т. е. значение переменной не может сохраняться в регистрах или локальной памяти потока и операция чтения этой переменной гарантированно вернет последнее записанное в нее значение.

Также модель памяти определяет дополнительную семантику ключевого слова final, имеющую отношение к видимости: 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. Наличие такой семантики для ключевого слова final позволяет создание неизменяемых (immutable) объектов, содержащих только final поля, такие объекты могут свободно передаваться между потоками без обеспечения синхронизации при передаче.

Есть одна проблема, связанная с final полями: реализация разрешает менять значения таких полей после создания объекта (это может быть сделано, например, с использованием механизма reflection). Если значение final поля — константа, чье значение известно на момент компиляции, изменения такого поля могут не иметь эффекта, т. к. обращения к этой переменной могли быть заменены компилятором на константу. Также спецификация разрешает другие оптимизации, связанные с final полями, например, операции чтения final переменной могут быть переупорядочены с операциями, которые потенциально могут изменить такую переменную. Так что 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. Вернее, с точки зрения потока, наблюдающего за выполнением операций в другом потоке, операции могут быть выполнены не в том порядке, в котором они идут в исходном коде. Тот же 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. Еще одна причина reordering, может заключаться в том, что процессор может решить поменять порядок выполнения операций, если, например, сочтет что такая последовательность выполнится быстрее.

Вопрос reordering также регулируется набором правил для отношения «происходит раньше» и у этих правил есть следствие, касающееся порядка операций, используемое на практике: операции чтения и записи volatile переменных не могут быть переупорядочены с операциями чтения и записи других volatile и не volatile переменных. Это следствие делает возможным использование volatile переменной как флага, сигнализирующем об окончании какого-либо действия. В остальном правила, касающиеся порядка выполнения операций, гарантируют упорядоченность операций для конкретного набора случаев (таких как, например, захват и освобождение монитора), во всех остальных случаях оставляя компилятору и процессору полную свободу для оптимизаций.

# Потокобезопасность

Потокобезопасность — свойство объекта или кода, которое гарантирует, что при исполнении или использовании несколькими потоками, код будет вести себя, как предполагается. Например, потокобезопасный счетчик не пропустит ни один счет, даже если один и тот же экземпляр этого счетчика будет использоваться несколькими потоками.

## Конкуренция и параллелизм

Конкуренция — способ одновременного решения множества задач.

Признаки:

∙ Наличие нескольких потоков управления (например Thread в Java, корутина в Kotlin), если поток управления один, то конкурентного выполнения быть не может.

∙ Недетерминированный результат выполнения. Результат зависит от случайных событий, реализации и того как была проведена синхронизация. Даже если каждый поток полностью детерминированный, итоговый результат будет недетерминированным

Параллелизм — способ выполнения разных частей одной задачи.

Признаки:

∙ Необязательно имеет несколько потоков управления

∙ Может приводить к детерминированному результату, так, например, результат умножения каждого элемента массива на число, не изменится, если умножать его по частям параллельно.

# Кооперативная многозадачность. Какой тип многозадачности использует Java? Чем обусловлен этот выбор?

Кооперативная многозадачность — способ деления процессорного времени между потоками, при котором каждый поток обязан отдавать управление следующему добровольно.

Преимущества такого подхода — простота реализации, меньшие накладные расходы на переключение контекста.

Недостатки — если один поток завис или ведет себя некорректно, то зависает целиком вся система и другие потоки никогда не получат управление.

Java использует вытесняющую многозадачность, при которой решение о переключении между потоками процесса принимает операционная система.

В отличие от кооперативной многозадачности управление операционной системе передается вне зависимости от состояния работающих приложений, благодаря чему, отдельные зависшие потоки процесса, как правило, не «подвешивают» всю систему целиком. За счет регулярного переключения между задачами также улучшается отзывчивость приложения и повышается оперативность освобождения ресурсов, которые больше не используются.

В реализации вытесняющая многозадачность отличается от кооперативной, в частности, тем, что требует обработки системного прерывания от аппаратного таймера.

# ordering, as-if-serial semantics, sequential consistency, visibility, atomicity, happens-before, mutual exclusion, safe publication?

ordering механизм, который определяет, когда один поток может увидеть out-of-order (неверный) порядок исполнения инструкций другого потока. CPU для повышения производительности может переупорядочивать процессорные инструкции и выполнять их в произвольном порядке до тех пор, пока для потока внутри не будет видно никаких отличий. Гарантия предоставляемая этим механизмом называется as-if-serial semantics.

sequential consistency — то же, что и as-if-serial semantics, гарантия того, что в рамках одного потока побочные эффекты от всех операций будут такие, как будто все операции выполняются последовательно.

visibility определяет, когда действия в одном потоке становятся видны из другого потока.

happens-before — логическое ограничение на порядок выполнения инструкций программы. Если указывается, что запись в переменную и последующее ее чтение связаны через эту зависимость, то как бы при выполнении не переупорядочивались инструкции, в момент чтения все связанные с процессом записи результаты уже зафиксированы и видны.

atomicity — атомарность операций. Атомарная операция выглядит единой и неделимой командой процессора, которая может быть или уже выполненной или еще невыполненной.

mutual exclusion (взаимоисключающая блокировка, семафор с одним состоянием) — механизм, гарантирующий потоку исключительный доступ к ресурсу. Используется для предотвращения одновременного доступа к общему ресурсу. В каждый момент времени таким ресурсом может владеть только один поток. Простейший пример:

|  |  |
| --- | --- |
| 1.  2.  3. | synchronized(obj) {  ...  } |

safe publication — показ объектов другим потокам из текущего, не нарушая ограничений visibility. Способы такой публикации в Java:

∙ static{} инициализатор;

∙ volatile переменные;

∙ atomic переменные;

∙ сохранение в разделяемой переменной, корректно защищенной с использованием synchronized(), синхронизаторов или других конструкций, создающих read/write memory barrier;

∙ final переменные в разделяемом объекте, который был корректно проинициализирован.

# Процесс и поток

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) — определенный 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. Т. к. потоки расходуют существенно меньше ресурсов, чем процессы, в процессе выполнения работы выгоднее создавать дополнительные потоки и избегать создания новых процессов.

## Зеленые потоки

Зеленые (легковесные) потоки (green threads) — потоки эмулируемые виртуальной машиной или средой исполнения. Создание зеленого потока не подразумевает под собой создание реального потока ОС.

Виртуальная машина Java берет на себя заботу о переключении между разными зелеными потоками, а сама машина работает как один поток ОС. Это дает несколько преимуществ. Потоки ОС относительно дороги в большинстве POSIX-систем. Кроме того, переключение между native threads гораздо медленнее, чем между зелеными потоками.

Это все означает, что в некоторых ситуациях зеленые потоки гораздо выгоднее, чем native threads. Система может поддерживать гораздо большее количество green threads, чем потоков OС. Например, гораздо практичнее запускать новый зеленый поток для нового HTTP-соединения к веб‑серверу, вместо создания нового native thread.

Однако есть и недостатки. Самый большой заключается в том, что вы не можете исполнять два потока одновременно. Поскольку существует только один native thread, только он и вызывается планировщиком ОС. Даже если у вас несколько процессоров и несколько зеленых потоков, только один процессор может вызывать зеленый поток. И все потому, что с точки зрения планировщика заданий ОС все это выглядит одним потоком.

Начиная с версии 1.2 Java поддерживает native threads, и с тех пор они используются по умолчанию.

# Способы создания потока

∙ Создать потомка класса Thread и переопределить его метод run();

∙ Создать объект класса Thread, передав ему в конструкторе экземпляр класса, реализующего интерфейс Runnable. Эти интерфейс содержит метод run(), который будет выполняться в новом потоке. Поток закончит выполнение, когда завершится его метод run().

∙ Вызвать метод submit() у экземпляра класса реализующего интерфейс ExecutorService, передав ему в качестве параметра экземпляр класса реализующего интерфейс Runnable или Callable (содержит метод call(), в котором описывается логика выполнения).

## Thread и Runnable

Thread — класс, некоторая надстройка над физическим потоком.

Runnable — интерфейс, представляющий абстракцию над выполняемой задачей.

Помимо того, что Runnable помогает разрешить наследования, несомненный плюс от его использования состоит от его использования состоит в том, что он позволяет логически отделить логику выполнения задачи от непосредственного управления потоком.

## start() и run()

Несмотря на то, что start() вызывает метод run() внутри себя, это не то же самое, что просто вызов run(). Если run() вызывается как обычный метод, то он вызывается в том же потоке и никакой новый поток не запускается, как это происходит, в случае, когда вы вызываете метод start().

## Как принудительно запустить поток?

Никак. В Java не существует абсолютно никакого способа принудительного запуска потока. Это контролируется JVM и Java не предоставляет никакго API для управления этим процессом.

# Монитор

Монитор, мьютекс (mutex) — средство обеспечения контроля за доступом к ресурсу. У монитора может быть максимум один владелец в каждый текущий момент времени. Следовательно, если кто‑то использует ресурс и захватил монитор для обеспечения единоличного доступа, то другой, желающий использовать тот же ресурс, должен подождать освобождения монитора, захватить его и только потом начать использовать ресурс.

Удобно представлять монитор 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 у каждого экземпляра объекта есть монитор, который контролируется непосредственно виртуальной машиной. Используется он так: любой нестатический synchronized-метод при своем вызове прежде всего пытается захватить монитор того объекта, у которого он вызван (на который он может сослаться как на this). Если это удалось — метод исполняется. Если нет — поток останавливается и ждет, пока монитор будет отпущен.

## private мьютекс

Объект для синхронизации делается private, чтобы сторонний код не мог на него синхронизироваться и случайно получить взаимную блокировку.

## Можно ли создавать новые экземпляры класса, пока выполняется static synchronized метод?

Да, можно создавать новые экземпляры класса, т. к. статические поля не принадлежат к экземплярам класса.

## На каком объекте происходит синхронизация при вызове static synchronized метода?

У синхронизированного статического метода нет доступа к this, но есть доступ к объекту класса Class, он присутствует в единственном экземпляре и именно он выступает в качестве монитора для синхронизации статических методов. Т. о., следующая конструкция:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | public class SomeClass {  public static synchronized void someMethod() {  // код  }  } |

эквивалентна такой:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | public class SomeClass {  public static void someMethod(){  synchronized(SomeClass.class){  // code  }  }  } |

# Синхронизация

Синхронизация — процесс, который позволяет выполнять потоки параллельно.

В Java все объекты имеют одну блокировку, благодаря которой только один поток одновременно может получить доступ к критическому коду в объекте. Такая синхронизация помогает предотвратить повреждение состояния объекта. Если поток получил блокировку, ни один другой поток не может войти в синхронизированный код, пока блокировка не будет снята. Когда поток, владеющий блокировкой, выходит из синхронизированного кода, блокировка снимается. Теперь другой поток может получить блокировку объекта и выполнить синхронизированный код. Если поток пытается получить блокировку объекта, когда другой поток владеет блокировкой, поток переходит в состояние Блокировки до тех пор, пока блокировка не снимется.

## Способы синхронизации

∙ 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, связаный с этим экземпляром, закончит работу.

∙ Использование классов из пакета java.util.concurrent, который предоставляет набор классов для организации межпоточного взаимодействия. Примеры таких классов — Lock, Semaphore и пр. Концепция данного подхода заключается в использовании атомарных операций и переменных.

# Состояния потока

Потоки могут находиться в одном из следующих состояний:

∙ Новый (New). После создания экземпляра потока, он находится в состоянии Новый до тех пор, пока не вызван метод start(). В этом состоянии поток не считается живым.

∙ Работоспособный (Runnable). Поток переходит в состояние Работоспособный, когда вызывается метод start(). Поток может перейти в это состояние также из состояния Работающий или из состояния Блокирован. Когда поток находится в этом состоянии, он считается живым.

∙ Работающий (Running). Поток переходит из состояния Работоспособный в состояние Работающий, когда Планировщик потоков выбирает его как работающий в данный момент.

∙ Живой, но не работоспособный (Alive, but not runnable). Поток может быть живым, но не работоспособным по нескольким причинам:

∙ Ожидание (Waiting). Поток переходит в состояние Ожидания, вызывая метод wait(). Вызов notify() или notifyAll() может перевести поток из состояния Ожидания в состояние Работоспособный.

∙ Сон (Sleeping). Метод sleep() переводит поток в состояние Сна на заданный промежуток времени в миллисекундах.

∙ Блокировка (Blocked). Поток может перейти в это состояние, в ожидании ресурса, такого как ввод/вывод или из‑за блокировки другого объекта. В этом случае поток переходит в состояние Работоспособный, когда ресурс становится доступен.

∙ Мертвый (Dead). Поток считается мертвым, когда его метод run() полностью выполнен. Мертвый поток не может перейти ни в какое другое состояние, даже если для него вызван метод start().

# Приоритет потока

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.

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\_PRlORITY = 5.

Получить текущее значение приоритета потока можно вызвав метод: final int getPriority() у экземпляра класса Thread.

# wait() и notify()/notifyAll()

Эти методы поределены у класса Object и предназначены для взаимодействия потоков между собой при межпоточной синхронизации.

∙ 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() подает 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.

Поток может вызвать методы wait() или notify() для определенного объекта, только если он в данный момент имеет блокировку на этот объект. wait(), notify() и notifyAll() должны вызываться только из синхронизированного кода.

## notify() и notifyAll()

Дело в том, что «висеть» на методе wait() одного монитора могут сразу несколько потоков. При вызове notify() только один из них выходит из wait() и пытается захватить монитор, а затем продолжает работу со следующего после wait() оператора. Какой из них выйдет — заранее неизвестно. А при вызове notifyAll(), все висящие на wait() потоки выходят из wait(), и все они пытаются захватить монитор. Понятно, что в любой момент времени монитор может быть захвачен только одним потоком, а остальные ждут своей очереди. Порядок очереди определяется планировщиком потоков Java.

## Почему методы wait() и notify() вызываются только в синхронизированном блоке?

Монитор надо захватывать в явном виде (через synchronized-блок), потому что методы wait() и notify() не синхронизированы.

## Чем отличается работа метода wait() с параметром и без параметра?

wait()

∙ без параметров освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод notify()/notifyAll(),

∙ с параметрами заставит поток ожидать заданное количество времени или вызова notify()/notifyAll().

# Semaphore

Semaphore — новый тип синхронизатора: семафор со счетчиком, реализующий шаблон синхронизации Семафор. Доступ управляется с помощью счетчика: 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 БД в пуле.

# volatile (synchronized, transient, native)

volatile — этот модификатор вынуждает потоки отключить оптимизацию доступа и использовать единственный экземпляр переменной. Если переменная примитивного типа — этого будет достаточно для обеспечения потокобезопасности. Если же переменная является ссылкой на объект — синхронизировано будет исключительно значение этой ссылки. Все же данные, содержащиеся в объекте, синхронизированы не будут!

synchronized — это зарезервированное слово позволяет добиваться синхронизации в помеченных им методах или блоках кода.

Ключевые слова transient и native к многопоточности никакого отношения не имеют, первое используется для указания полей класса, которые не нужно сериализовать, а второе — сигнализирует о том, что метод реализован в платформо-зависимом коде.

# Различия volatile и Atomic-переменным

volatile принуждает использовать единственный экземпляр переменной, но не гарантирует атомарность. Например, операция count++ не станет атомарной просто потому, что count объявлена volatile. C другой стороны class AtomicInteger предоставляет атомарный метод для 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.

## В чем заключаются различия между java.util.concurrent.Atomic\*.compareAndSwap() и java.util.concurrent.Atomic\*.weakCompareAndSwap().

∙ weakCompareAndSwap() не создает memory barrier и не дает гарантии happens-before;

∙ weakCompareAndSwap() сильно зависит от кэша/CPU, и может возвращать false без видимых причин;

∙ weakCompareAndSwap(), более легкая, но поддерживаемая далеко не всеми архитектурами и не всегда эффективная операция.

# Потоки-демоны

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. Базовое свойство потоков-демонов заключается в возможности основного потока приложения завершить выполнение потока-демона (в отличие от обычных потоков) с окончанием кода метода main(), не обращая внимания на то, что поток-демон еще работает.

## Можно ли сделать основной поток программы демоном?

Нет. Потоки-демоны позволяют описывать фоновые процессы, которые нужны только для обслуживания основных потоков выполнения и не могут существовать без них.

# Thread.join()

Когда поток вызывает join() для другого потока, текущий работающий поток будет ждать, пока другой поток, к которому он присоединяется, не будет завершен:

|  |  |
| --- | --- |
| 1.  2.  3. | void join()  void join(long millis)  void join(long millis, int nanos) |

# Thread.sleep() и Thread.yield()

Метод yield() служит причиной того, что поток переходит из состояния работающий (running) в состояние работоспособный (runnable), давая возможность другим потокам активизироваться. Но следующий выбранный для запуска поток может и не быть другим.

Метод sleep() вызывает засыпание текущего потока на заданное время, состояние изменяется с работающий (running) на ожидающий (waiting).

## Что значит «усыпить» поток?

Это значит приостановить его на определенный промежуток времени, вызвав в ходе его выполнения статический метод Thread.sleep() передав в качестве параметра необходимое количество времени в миллисекундах. До истечения этого времени поток может быть выведен из состояния ожидания вызовом interrupt() с выбрасыванием InterruptedException.

# Как остановить поток?

На данный момент в Java принят уведомительный порядок остановки потока (хотя JDK е.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 некоторый 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 определенных методов извне. Методика действия при этом остается 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.

## 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 переведет 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?

# Runnable и Callable

|  |  |  |
| --- | --- | --- |
|  | Runnable | Callable<V> |
|  | @FunctionalInterface | |
| версии Java | 1.0 | 5.0 |
| методы | run() | call() |
| тип | void | Future |
| исключения | — | throws Exception |

# FutureTask

FutureTask представляет собой отменяемое асинхронное вычисление в параллельном Java приложении. 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 еще не завершено. Объекты FutureTask могут быть использованы для обертки объектов Callable и Runnable. Т. к. FutureTask реализует Runnable, его можно передать в Executor на выполнение.

# deadlock (взаимная блокировка)

Взаимная блокировка (deadlock) — явление при котором все потоки находятся в режиме ожидания. Происходит, когда достигаются состояния:

∙ взаимного исключения: по крайней мере один ресурс занят в режиме неделимости и следовательно только один поток может использовать ресурс в любой данный момент времени.

∙ удержания и ожидания: поток удерживает как минимум один ресурс и запрашивает дополнительные ресурсов, которые удерживаются другими потоками.

∙ отсутствия предочистки: операционная система не переназначивает ресурсы: если они уже заняты, они должны отдаваться удерживающим потокам сразу же.

∙ цикличного ожидания: поток ждет освобождения ресурса другим потоком, который в свою очередь ждет 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.

# 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.

# race condition (состояние гонки)

Состояние гонки (race condition) — ошибка проектирования многопоточной системы или приложения, при которой эта работа напрямую 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.

## Способы решения

Использование локальной копии — копирование разделяемой переменной в локальную переменную потока. Этот способ работает только тогда, когда переменная одна и копирование производится атомарно (за одну машинную команду), использование volatile.

Синхронизация — операции над разделяемым ресурсом происходят в синхронизированном блоке (при использовании ключевого слова synchronized).

Комбинирование методов — вышеперечисленные способы можно комбинировать, копируя «опасные» переменные в синхронизированном блоке. С одной стороны, это снимает ограничение на атомарность, с другой — позволяет избавиться от слишком больших синхронизированных блоков.

Очевидных способов выявления и исправления состояний гонки не существует. Лучший способ избавиться от гонок — правильное проектирование многозадачной системы.

# Фреймворк Fork/Join

Фреймворк Fork/Join, представленный в JDK 7, — 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) идет процесс «свертки» — 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 все меньшие и меньшие диапазоны, пока не вырождается в тривиальный случай из е элемента. Хотя в некотором смысле Join будет необходим и тут, т. к. все равно остается необходимость дождаться пока не закончится выполнение всех подзадач.

Еще одно замечательное 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 все еще заняты.

# Как проверить, удерживает ли поток монитор определенного ресурса?

Метод Thread.holdsLock(lock) возвращает true, когда текущий поток удерживает монитор у определенного объекта.

# В чем заключаются различия между CyclicBarrier и CountDownLatch?

CountDownLatch (замок с обратным отсчетом) предоставляет возможность любому количеству потоков в блоке кода ожидать до тех пор, пока не завершится определенное количество операций, выполняющихся в других потоках, перед тем как они будут «отпущены», чтобы продолжить свою деятельность. В конструктор CountDownLatch(int count) обязательно передается количество операций, которое должно быть выполнено, чтобы замок «отпустил» заблокированные потоки.

Примером CountDownLatch из жизни может служить сбор экскурсионной группы: пока не наберется определенное количество человек, экскурсия не начнется.

CyclicBarrier реализует шаблон синхронизации «Барьер». Циклический барьер является точкой синхронизации, в которой указанное количество параллельных потоков встречается и блокируется. Как только все потоки прибыли, выполняется опционное действие (или не выполняется, если барьер был инициализирован без него), и, после того, как оно выполнено, барьер ломается и ожидающие потоки «освобождаются». В конструкторы барьера CyclicBarrier(int parties) и CyclicBarrier(int parties, Runnable barrierAction) обязательно передается количество сторон, которые должны «встретиться», и, опционально, действие, которое должно произойти, когда стороны встретились, но перед тем когда они будут «отпущены».

CyclicBarrier является альтернативой метода join(), который «собирает» потоки только после того, как они выполнились.

CyclicBarrier похож на CountDownLatch, но главное различие между ними в том, что использовать «замок» можно лишь единожды - после того, как его счётчик достигнет нуля, а «барьер» можно использовать неоднократно, даже после того, как он «сломается».

id="coreMultithreadingMutexPrivateMutex" class="hideContent">

<div class="text">Объект для синхронизации делается private, чтобы сторонний код не мог на него синхронизироваться и случайно получить взаимную блокировку.</div>

</div>

</div>

# !!! Что означает ключевое слово synchronized? Где и для чего может использоваться?

Вап

onClick="window.open('java.util.concurrent.locks.Lock')"><span class="textCode">Lock</span></div>

<div class="text showHide" onClick="window.open('java.util.concurrent.locks.ReadWriteLock')"><span class="textCode">ReadWriteLock</span></div>

<div class="text showHide" onClick="window.open('java.util.concurrent.locks.ReentrantReadWriteLock')"><span class="textCode">ReentrantReadWriteLock</span></div>

<div class="text">

<li>расширяет базовый интерфейс <span class="textCode">Lock</span>;</li>

<li>для улучшения производительности в многопоточном процессе и оперирует парой связанных блокировок (одна - для операций чтения, другая - для записи);</li>

<li>блокировка чтения может удерживаться одновременно несколькими читающими потоками, до тех пор пока не появится записывающий;</li>

<li>блокировка записи является эксклюзивеной.</li>

</div>

<div class="text">Существует реализующий интерфейс <span class="textCode">ReadWriteLock</span> класс <span class="textCode">ReentrantReadWriteLock</span>, который поддерживает до 65535 блокировок записи и до стольки же блокировок чтения.</div>

<script>document.write(codeCoreCultithreadingReadWriteLock.innerHTML)</script>

</div>

<div class="text showHide textCode" onClick="showHide('coreMultithreadingСlassesAndInterfacesSemaphore')">Semaphore</div>

<div id="coreMultithreadingСlassesAndInterfacesSemaphore" class="hideContent">

<div class="text">

<li>как <span class="textCode">synchronize</span> ограничивает одновременный доступ к ресурсу, но одновременно могут работать несколько потоков (не более заданного);</li>

<li><span class="textCode">void acquire()</span> — <span class="textCode">void release()</span>,</li>

<li><span class="textCode">new Semaphore(е0, true)</span> — "справедливая" синхронизация.</li>

</div>

<script>document.write(codeCoreMultithreadingСlassesAndInterfacesSemaphore.innerHTML)</script>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingСlassesAndInterfacesCountdownlatch')"><span class="textCode">CountDownLatch</span> и <span class="textCode">CyclicBarrier</span></div>

<div id="coreMultithreadingСlassesAndInterfacesCountdownlatch" class="hideContent">

<div class="text">

<li>дожидаться друг друга и одновременно стартовать;</li>

<li><span class="textCode">void countDown()</span> и <span class="textCode">void await()</span>;</li>

<li><span class="textCode">CyclicBarrier</span> допускает повторное ожидание.</li>

</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingСlassesAndInterfacesFuturetask')"><span class="textCode">FutureTask</span></div>

<div id="coreMultithreadingСlassesAndInterfacesFuturetask" class="hideContent">

<div class="text"><span class="textCode">FutureTask</span> представляет собой отменяемое асинхронное вычисление в параллельном Java приложении. Этот класс предоставляет базовую реализацию <span class="textCode">Future</span>, 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 еще не завершено. Объекты <span class="textCode">FutureTask</span> могут быть использованы для обертки объектов <span class="textCode">Callable</span> и <span class="textCode">Runnable</span>. Так как <span class="textCode">FutureTask</span> реализует <span class="textCode">Runnable</span>, его можно передать в <span class="textCode">Executor</span> на выполнение.</div>

</div>

<div class="text showHide textCode" onClick="showHide('coreMultithreadingСlassesAndInterfacesReentrantLock')">ReentrantLock</div>

<div id="coreMultithreadingСlassesAndInterfacesReentrantLock" class="hideContent">

<div class="text">

<li><span class="textCode">synchronized</span> на несколько методов;</li>

<li><span class="textCode">lock()</span> и <span class="textCode">unlock()</span>.</li>

</div>

<script>document.write(codeCoreMultithreadingСlassesAndInterfacesReentrantLock.innerHTML)</script>

</div>

<div class="text showHide textCode" onClick="showHide('coreMultithreadingСlassesAndInterfacesCondition')">Condition</div>

<div id="coreMultithreadingСlassesAndInterfacesCondition" class="hideContent">

<div class="text">

<li>аналог <span class="textCode">wait()</span>/<span class="textCode">notify()</span>;</li>

<li>привязка к <span class="textCode">Lock</span>;</li>

<li>у одного <span class="textCode">Lock</span> может быть несколько <span class="textCode">Condition</span>.</li>

</div>

<script>document.write(codeCoreMultithreadingСlassesAndInterfacesCondition.innerHTML)</script>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingСlassesAndInterfacesCollections')">Коллекции</div>

<div id="coreMultithreadingСlassesAndInterfacesCollections" class="hideContent">

<div class="text">

<li><span class="textCode">ConcurrentHashMap</span> — <span class="textCode">HashMap</span>,</li>

<li><span class="textCode">ConcurrentSkipListMap</span> — <span class="textCode">TreeMap</span>,</li>

<li><span class="textCode">ConcurrentSkipListSet</span> — <span class="textCode">TreeSet</span>,</li>

<li><span class="textCode">CopyOnWriteArrayList</span> — <span class="textCode">ArrayList</span>,</li>

<li><span class="textCode">CopyOnWriteArraySet</span>.</li>

<li><span class="textCode">java.util.Collections.synchronizedCollection()</span>.</li>

</div>

<div class="text showHide textCode" onClick="showHide('coreMultithreadingСlassesAndInterfacesCollectionsConcurrentlinkedqueue')">ConcurrentLinkedQueue</div>

<div id="coreMultithreadingСlassesAndInterfacesCollectionsConcurrentlinkedqueue" class="hideContent">

<div class="text">Реализация очереди, поддерживащая одновременный доступ из многих потоков, при этом не использующая блокироки.</div>

<div class="text">

Операции:

<li><span class="textCode">boolean offer(E e)</span></li>

<li><span class="textCode">E poll()</span></li>

<li><span class="textCode">E peek()</span></li>

</div>

</div>

<div class="text showHide textCode" onClick="showHide('coreMultithreadingСlassesAndInterfacesCollectionsBlockingqueue')">BlockingQueue&#60;E></div>

<div id="coreMultithreadingСlassesAndInterfacesCollectionsBlockingqueue" class="hideContent">

<div class="text">Очередь, поддерживающая ограничение по размеру и операции ожидания.</div>

<div class="text">

Операции:

<li><span class="textCode">void put(E e)</span>,</li>

<li><span class="textCode">E take()</span>.</li>

</div>

<div class="text">

Реализации:

<li><span class="textCode">LinkedBlockingQueue</span>,</li>

<li><span class="textCode">ArrayBlockingQueue</span>.</li>

</div>

</div>

</div>

<div class="text showHide textCode" onClick="showHide('coreMultithreadingСlassesAndInterfacesExecutorservice')">ExecutorService и соседи</div>

<div id="coreMultithreadingСlassesAndInterfacesExecutorservice" class="hideContent">

<div class="text">

<li>инфраструктура для выполнения задач в несколько потоков;</li>

<li>инкапсулирует создание потоков, организацию очереди задач, распределение задач по потокам.</li>

</div>

<div class="text">

Методы:

<li><span class="textCode">Future&#60;?> submit(Runnable task)</span>,</li>

<li><span class="textCode">&#60;T> Future&#60;?> submit(Callable&#60;T> task)</span>,</li>

<li><span class="textCode">void shutdown</span>,</li>

<li><span class="textCode">List&#60;Runnable> shutdownNow()</span>.</li>

</div>

<div class="text showHide textCode" onClick="showHide('coreMultithreadingСlassesAndInterfacesExecutorserviceExecutor')">Executor</div>

<div id="coreMultithreadingСlassesAndInterfacesExecutorserviceExecutor" class="hideContent">

<div class="text">

<li><span class="textCode">ExecutorService newSingleThreadExecutor()</span>,</li>

<li><span class="textCode">ExecutorService newFixedThreadPool(int nThreads)</span>,</li>

<li><span class="textCode">ExecutorService newCachedTheadPool()</span>.</li>

</div>

</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingСlassesAndInterfacesForkjoin')">фреймворк Fork/Join</div>

<div id="coreMultithreadingСlassesAndInterfacesForkjoin" class="hideContent">

<div class="text">

<li>вариант ExecutorService, в котором выполняющиеся задачи могут динамически порождать подзадачи;</li>

<li>принимает на исполнение ForkJoinTask.</li>

</div>

<hr>

<div class="text">Одну задачу рекурсивно разбить на маленькие подзадачи (Fork) и решает параллельно (Join).</div>

<div class="text">Потоки, которые завершили выполнение собственных подзадач, могут «украсть» подзадачи у других потоков, которые все еще заняты (work-stealing алгоритм).</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingСlassesAndInterfacesStream')">стримы</div>

<div id="coreMultithreadingСlassesAndInterfacesStream" class="hideContent">

<div class="text">

<span class="textCode">stream.parallel()</span>

<li>возвращает <span class="textCode">stream</span>, дальнейшие операции в котором будут исполняться параллельно;</li>

<li>надо следить за доступом к общим данным из передаваемых в stream операций.</li>

</div>

</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingMethods')">методы</div>

<div id="coreMultithreadingMethods" class="hideContent">

<div class="text showHide" onClick="showHide('coreMultithreadingMethodsStartAndRun')"><span class="textCode">start()</span> и <span class="textCode">run()</span></div>

<div id="coreMultithreadingMethodsStartAndRun" class="hideContent">

<div class="text"><span class="textCode">run()</span> не запусит новый поток.</div>

</div>

<div class="text showHide" onClick="showHide('Можно ли создавать новые экземпляры класса, пока выполняется static synchronized метод?')">Можно ли создавать новые экземпляры класса, пока выполняется <span class="textCode">static</span> <span class="textCode">synchronized</span> метод?</div>

<div id="Можно ли создавать новые экземпляры класса, пока выполняется static synchronized метод?" class="hideContent">

<div class="text">Да (статические поля не принадлежат к объекту).</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingMethodsWaitNotifyAndNotifyall')"><span class="textCode">wait()</span> и <span class="textCode">notify()</span>/<span class="textCode">notifyAll()</span> (<span class="textCode">Object</span>)</div>

<div id="coreMultithreadingMethodsWaitNotifyAndNotifyall" class="hideContent">

<div class="text showHide" onClick="showHide('coreMultithreadingMethodsWaitNotifyAndNotifyallWaitNotifyAndNotifyall')"><span class="textCode">wait()</span> и <span class="textCode">notify()</span>/<span class="textCode">notifyAll()</span></div>

<div id="coreMultithreadingMethodsWaitNotifyAndNotifyallWaitNotifyAndNotifyall" class="hideContent">

<div class="text">

<li>должны вызываться только из синхронизированного кода.</li>

<li>не могут быть вызваны для потоков <span class="hе">блокировка</span>;</li>

</div>

<div class="text">

<li><span class="textCode">wait()</span>: освобождает монитор и <span class="hе">работающий</span> <span th:text="${should}"></span> <span class="hе">ожидание</span>;</li>

<li><span class="textCode">notify()</span>: один из потоков объекта <span class="hе">ожидание</span> <span th:text="${should}"></span> <span class="hе">работоспособный</span> (не определено какой);</li>

<li><span class="textCode">notifyAll()</span>: все потоки объекта <span class="hе">ожидание</span> <span th:text="${should}"></span> <span class="hе">работоспособный</span>.</li>

</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingNotifyAndNotifyall')"><span class="textCode">notify()</span> и <span class="textCode">notifyAll()</span></div>

<div id="coreMultithreadingNotifyAndNotifyall" class="hideContent">

<div class="text">Дело в том, что «висеть» на методе wait() одного монитора могут сразу несколько потоков. При вызове notify() только один из них выходит из wait() и пытается захватить монитор, а затем продолжает работу со следующего после wait() оператора. Какой из них выйдет - заранее неизвестно. А при вызове notifyAll(), все висящие на wait() потоки выходят из wait(), и все они пытаются захватить монитор. Понятно, что в любой момент времени монитор может быть захвачен только одним потоком, а остальные ждут своей очереди. Порядок очереди определяется планировщиком потоков Java.</div>

</div>

<div class="text showHide" onClick="showHide('Почему методы wait() и notify() вызываются только в синхронизированном блоке?')">Почему методы <span class="textCode">wait()</span> и <span class="textCode">notify()</span> вызываются только в синхронизированном блоке?</div>

<div id="Почему методы wait() и notify() вызываются только в синхронизированном блоке?" class="hideContent">

<div class="text"><span class="textCode">wait()</span> и <span class="textCode">notify()</span> не синхронизированы.</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingWait')"><span class="textCode">wait()</span> с параметром и без</div>

<div id="coreMultithreadingWait" class="hideContent">

<div class="text">

<li class="text">без параметров освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод <span class="textCode">notify()</span>/<span class="textCode">notifyAll()</span>,</li>

<li class="text">с параметрами заставит поток ожидать заданное количество времени или вызова <span class="textCode">notify()</span>/<span class="textCode">notifyAll()</span>.</li>

</div>

</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingThreadSleepAndThreadYield')"><span class="textCode">Thread.sleep()</span> и <span class="textCode">Thread.yield()</span></div>

<div id="coreMultithreadingThreadSleepAndThreadYield" class="hideContent">

<div class="text"><span class="textCode">yield()</span> — <span class="hе">работающий</span> <span th:text="${should}"></span> <span class="hе">работоспособный</span> (возможность другим потокам, хотя может и сам).</div>

<div class="text"><span class="textCode">sleep()</span> — <span class="hе">работающий</span> <span th:text="${should}"></span> <span class="hе">ожидающий</span> на заданное время.</div>

<div class="text showHide" onClick="window.open('java.lang.Thread')"><span class="textCode">Thread</span></div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingThreadJoin')"><span class="textCode">Thread.join()</span></div>

<div id="coreMultithreadingThreadJoin" class="hideContent">

<div class="text"><span class="hе">Работающий</span> будет ждатьт завершение вызванного.</div>

<script>document.write(codeCoreMultithreadingThreadJoin.innerHTML)</script>

<div class="text showHide" onClick="window.open('java.lang.Thread')"><span class="textCode">Thread</span></div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingBlockingMethod')">блокирующий метод</div>

<div id="coreMultithreadingBlockingMethod" class="hideContent">

<div class="text">Метод, который блокируется до тех пор, пока задание не выполнится (<span class="textCode">accept()</span> у <span class="textCode">ServerSocket</span> блокируется в ожидании подключения клиента). Здесь блокирование означает, что контроль не вернется к вызывающему методу до тех пор, пока не выполнится задание. Так же существуют асинхронные или неблокирующиеся методы, которые могут завершится до выполнения задачи.</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingSubmitAndExecute')"><span class="textCode">submit()</span> и <span class="textCode">execute()</span> у пула потоков</div>

<div id="coreMultithreadingSubmitAndExecute" class="hideContent">

<div class="text">Оба метода являются способами подачи задачи в пул потоков, но между ними есть небольшая разница.</div>

<div class="text"><span class="textCode">execute(Runnable command)</span> определен в интерфейсе <span class="textCode">Executor</span> и выполняет поданную задачу и ничего не возвращает.</div>

<div class="text"><span class="textCode">submit()</span> – перегруженный метод, определенный в интерфейсе <span class="textCode">ExecutorService</span>. Способен принимать задачи типов <span class="textCode">Runnable</span> и <span class="textCode">Callable</span> и возвращать объект <span class="textCode">Future</span>, который можно использовать для контроля и управления процессом выполнения, получения его результата.</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingMethodsThreadStop')">не рекомендуется использовать <span class="textCode">Thread.stop()</span></div>

<div id="coreMultithreadingMethodsThreadStop" class="hideContent">

<div class="text">При принудительной остановке (приостановке) потока, <span class="textCode">stop()</span> 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 переведет ли принудительное снятие к нарушению консистентности системы? То же самое можно расширить и на случай соединения с базой данных: если</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingInterruptedAndIsInterrupted')"><span class="textCode">interrupted()</span> и <span class="textCode">isInterrupted()</span></div>

<div id="coreMultithreadingInterruptedAndIsInterrupted" class="hideContent">

<div class="text">Механизм прерывания работы потока в Java реализован с использованием внутреннего флага, известного как статус прерывания. Прерывание потока вызовом <span class="textCode">Thread.interrupt()</span> устанавливает этот флаг. Методы <span class="textCode">Thread.interrupted()</span> и <span class="textCode">isInterrupted()</span> позволяют проверить, является ли поток прерванным.</div>

<div class="text">Когда прерванный поток проверяет статус прерывания, вызывая статический метод <span class="textCode">Thread.interrupted()</span>, статус прерывания сбрасывается.</div>

<div class="text">Нестатический метод <span class="textCode">isInterrupted()</span> используется одним потоком для проверки статуса прерывания у другого потока, не изменяя флаг прерывания.</div>

</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingModifiers')">модификаторы</div>

<div id="coreMultithreadingModifiers" class="hideContent">

<div class="text showHide" onClick="showHide('coreMultithreadingModifiersVolatileSynchronizedTransientNative')"><span class="textCode">volatile</span>, <span class="textCode">synchronized</span>, <span class="textCode">transient</span> и <span class="textCode">native</span></div>

<div id="coreMultithreadingModifiersVolatileSynchronizedTransientNative" class="hideContent">

<div class="text">

<span class="textCode">volatile</span> — вынуждает потоки отключить оптимизацию доступа и использовать единственный экземпляр переменной.

<li>переменная примитивного типа – достаточно для обеспечения потокобезопасности;</li>

<li>переменная ссылочного типа – синхронизировано будет значение ссылки (данные объекта синхронизированы не будут).</li>

</div>

<div class="text"><span class="textCode">synchronized</span> — позволяет добиваться синхронизации в методах или блоках кода.</div>

<div class="text">

К многопоточности отношения не имеют:

<li><span class="textCode">transient</span> — поля не сериализуются;</li>

<li><span class="textCode">native</span> — метод реализован в платформо-зависимом коде.</li>

</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingModifiersVolatileAndAtomic')"><span class="textCode">volatile</span> и Atomic переменные</div>

<div id="coreMultithreadingModifiersVolatileAndAtomic" class="hideContent">

<div class="text"><span class="textCode">volatile</span> принуждает использовать единственный экземпляр переменной, но не гарантирует атомарность. Например, операция <span class="textCode">count++</span> не станет атомарной просто потому что <span class="textCode">count</span> объявлена <span class="textCode">volatile</span>. C другой стороны <span class="textCode">class AtomicInteger</span> предоставляет атомарный метод для выполнения таких комплексных операций атомарно, например <span class="textCode">getAndIncrement()</span> — атомарная замена оператора инкремента, его можно использовать, чтобы атомарно увеличить текущее значение на один. Похожим образом сконструированы атомарные версии и для других типов данных.</div>

</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingThread')">поток (thread)</div>

<div id="coreMultithreadingThread" class="hideContent">

<div class="text showHide" onClick="showHide('coreMultithreadingThreadProcessAndThread')">процесс и поток (thread)</div>

<div id="coreMultithreadingThreadProcessAndThread" class="hideContent">

<div class="text">

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingThreadSafety')">потокобезопасность</div>

<div id="coreMultithreadingThreadSafety" class="hideContent">

<div class="text">Гарантия, что многопоточная программа поведет себя как предполагается (счетчик не пропустит ни один счет).</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingThreadCreate')">создать поток</div>

<div id="coreMultithreadingThreadCreate" class="hideContent">

<li class="text">у потомка <span class="textCode">Thread</span> переопределить <span class="textCode">run()</span>.</li>

<li class="text">в конструктор объекта <span class="textCode">Thread</span> передать объект класса <span class="textCode">implements Runnable</span> (либо лямбда).</li>

<div class="text showHide" onClick="showHide('coreMultithreadingThreadCreateThreadAndRunnable')"><span class="textCode">Thread</span> и <span class="textCode">Runnable</span></div>

<div id="coreMultithreadingThreadCreateThreadAndRunnable" class="hideContent">

<script>document.write(coreMultithreadingСlassesAndInterfacesThreadAndRunnable.innerHTML)</script>

</div>

<hr>

<li class="text"><span class="textCode">submit()</span> у объекта <span class="textCode">implements ExecutorService</span> с параметром объекта <span class="textCode">implements Runnable</span> или <span class="textCode">implements Callable</span>.</li>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingThreadForceStart')">принудительно запустить поток?</div>

<div id="coreMultithreadingThreadForceStart" class="hideContent">

<div class="text">Не можем.</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingThreadConditions')">состояния потока</div>

<div id="coreMultithreadingThreadConditions" class="hideContent">

<div class="text">

<li><span class="hе">новый</span> (new):</li>

<li class="li2">создание — <span class="textCode">start()</span>,</li>

<li class="li2">не живой;</li>

</div>

<div class="text">

<li><span class="hе">работоспособный</span> (runnable):</li>

<li class="li2">после <span class="textCode">start()</span>,</li>

<li class="li2">живой;</li>

</div>

<div class="text">

<li><span class="hе">работающий</span> (running):</span>

<li class="li2">из <span class="hе">работоспособный</span> планировщиком потоков;</li>

</div>

<div class="text">

<li><span class="hе">ожидание</span> (waiting):</li>

<li class="li2"><span class="textCode">wait()</span> — <span class="textCode">notify()</span>/<span class="textCode">notifyAll()</span> (<span class="textCode">interrupt()</span>),</li>

<li class="li2">либо <span class="textCode">wait()</span> на заданное время (<span class="textCode">interrupt()</span>);</li>

</div>

<div class="text">

<li><span class="hе">сон</span> (sleeping).</li>

<li class="li2"><span class="textCode">sleep()</span> на заданное время (<span class="textCode">interrupt()</span>);</li>

</div>

<div class="text">

<li><span class="hе">блокировка</span> (blocked):</li>

<li class="li2">в ожидании ресурса (ввод/вывод или из-за блокировки другого объекта);</li>

</div>

<div class="text">

<li><span class="hе">мертвый</span> (dead):</li>

<li class="li2">после выполнения <span class="textCode">run()</span> или неперехваченное исключение,</li>

<li class="li2">не пригоден к использованию (даже после <span class="textCode">start()</span>).</li>

</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingThreadPriority')">приоритет потока</div>

<div id="coreMultithreadingThreadPriority" class="hideContent">

<div class="text">Какому из потоков будет разрешено работать (планировщиком потоков).</div>

<div class="text"><span class="textCode">final void setPriority(int level)</span> (<span class="textCode">Thread</span>). <span class="textCode">е</span> <= <span class="textCode">level</span> <= <span class="textCode">е0</span>, по умолчанию — <span class="textCode">5</span>.</div>

<div class="text"><span class="textCode">final int getPriority()</span> (<span class="textCode">Thread</span>).</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingThreadDemons')">потоки-демоны</div>

<div id="coreMultithreadingThreadDemons" class="hideContent">

<div class="text showHide" onClick="showHide('coreMultithreadingThreadDemonsThreadDemons')">что такое?</div>

<div id="coreMultithreadingThreadDemonsThreadDemons" class="hideContent">

<div class="text">

<li>работают в фоновом режиме;</li>

<li>не являются частью программы;</li>

<li>обслуживают основные потоки;</li>

<li><span class="textCode">setDaemon(boolean value)</span>;</li>

<li><span class="textCode">boolean isDaemon()</span>;</li>

<li>после завершения <span class="textCode">main()</span> завершают работу.</li>

</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingThreadDemonsMainThreadDemon')">основной поток демоном?</div>

<div id="coreMultithreadingThreadDemonsMainThreadDemon" class="hideContent">

<div class="text">Нет. Потоки-демоны позволяют описывать фоновые процессы, которые нужны только для обслуживания основных потоков выполнения и не могут существовать без них.</div>

</div>

</div>

<div class="text showHide" onClick="showHide('Что значит «усыпить» поток?')">«усыпить» поток</div>

<div id="Что значит «усыпить» поток?" class="hideContent">

<div class="text"><span class="textCode">Thread.sleep()</span> с указанием времени (<span class="textCode">interrupt()</span>).</div>

<div class="text showHide" onClick="window.open('java.lang.Thread')"><span class="textCode">Thread</span></div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingThreadStop')">остановить поток</div>

<div id="coreMultithreadingThreadStop" class="hideContent">

<div class="text"><span class="textCode">stop()</span>, <span class="textCode">suspend()</span> и <span class="textCode">resume()</span> — <span class="textCode">deprecated</span>.</div>

<div class="text">

<span class="textCode">interrupt()</span> (<span class="textCode">Thread</span>) — выставляет некоторый внутренний флаг-статус прерывания.

<li>В дальнейшем состояние этого флага можно проверить с помощью метода <span class="textCode">isInterrupted()</span> или <span class="textCode">Thread.interrupted()</span> (для текущего потока).</li>

<li>способен вывести поток из состояния ожидания или спячки (<span class="textCode">sleep()</span> или <span class="textCode">wait()</span>) – текущее состояние прервется и будет выброшено исключение <span class="textCode">InterruptedException</span> (флаг не выставляется).</li>

</div>

<div class="text">

Схема действия при этом получается следующей:

<li>Реализовать поток.</li>

<li>В потоке периодически проводить проверку статуса прерывания через вызов <span class="textCode">isInterrupted()</span>.</li>

<li>Если состояние флага изменилось или было выброшено исключение во время ожидания/спячки, следовательно поток пытаются остановить извне.</li>

<li>Принять решение – продолжить работу (если по каким-то причинам остановиться невозможно) или освободить заблокированные потоком ресурсы и закончить выполнение.</li>

</div>

<div class="text">Возможная проблема, которая присутствует в этом подходе – блокировки на потоковом вводе-выводе. Если поток заблокирован на чтении данных — вызов <span class="textCode">interrupt()</span> 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 <span class="textCode">read()</span>. Если же чтение каким-то образом связано с сетью – стоит использовать неблокирующий ввод-вывод из Java NIO.</div>

<div class="text">Второй вариант реализации метода остановки (а также и приостановки) – сделать собственный аналог <span class="textCode">interrupt()</span>. Т.е. объявить в классе потока флаги – на остановку и/или приостановку и выставлять их путем вызова заранее определенных методов извне. Методика действия при этом остается 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 <span class="textCode">volatile</span>).</div>

</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingDeadlock')">взаимная блокировка (deadlock), livelock</div>

<div id="coreMultithreadingDeadlock" class="hideContent">

<div class="text">Все потоки находятся в режиме ожидания.</div>

<div class="text">

<li>взаимного исключения: по крайней мере один ресурс занят в режиме неделимости и следовательно только один поток может использовать ресурс в любой данный момент времени.</li>

<li>удержания и ожидания: е-й удерживает ресурс и запрашивает ресурсов, удерживаемый другими.</li>

<li>отсутствия предочистки: ОС не переназначивает ресурсы (если они уже заняты, они должны отдаваться удерживающим потокам сразу же).</li>

<li>цикличного ожидания: е-й поток ждет освобождения ресурса 2-м, 2-й ждет освобождения ресурса заблокированного е-ым.</li>

</div>

<div class="text">Избежать цикличное ожидание — получать мониторы разделяемых ресурсов в определенном порядке, освобождая их в обратном порядке.</div>

<div class="text">livelock – взаимная блокировка, при котором несколько потоков выполняют бесполезную работу, попадая в зацикленность.</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingJmm')">JMM (Java Memory Model, модель памяти Java)</div>

<div id="coreMultithreadingJmm" class="hideContent">

<li class="text showHide" onClick="showHide('coreMultithreadingJmmJmm')">описывает поведение потоков в JVM.</li>

<div id="coreMultithreadingJmmJmm" class="hideContent">

<div class="text">Часть семантики языка Java, набор правил, описывающий выполнение многопоточных программ и правил, по которым потоки могут взаимодействовать друг с другом посредством основной памяти.</div>

</div>

<div class="text">

Определяет

<li>набор действий (чтение и запись переменной, захват и освобождений монитора, чтение и запись volatile переменной, запуск нового потока),</li>

<li class="showHide" onClick="showHide('coreMultithreadingJmm2')">отношение между этими действиями (-happens-before).</li>

<div id="coreMultithreadingJmm2" class="hideContent">

<script>document.write(linkHappensbefor.innerHTML)</script>

</div>

</div>

<div class="text">

Основные области JMM:

<li class="showHide" onClick="showHide('coreMultithreadingJmmVisibility')">видимость (visibility),</li>

<div id="coreMultithreadingJmmVisibility" class="hideContent">

<div class="text">Один поток может в какой-то момент временно сохранить значение некоторых полей не в основную память, а в регистры или локальный кэш процессора, таким образом второй поток, выполняемый на другом процессоре, 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.</div>

<div class="text showHide" onClick="showHide('coreMultithreadingJmmVariables')">переменные (кроме локальных)</div>

<div id="coreMultithreadingJmmVariables" class="hideContent">

<li>в главной памяти (доступна всем потокам);</li>

<li>Каждый поток имеет локальную—рабочую—память (хранит копии переменных и работает с копиями);</li>

<li>Надо отметить, что это описание не требование к реализации, а всего лишь модель, которая объясняет поведение программы, так, в качестве локальной памяти не обязательно выступает кэш память, это могут быть регистры процессора или потоки могут вообще не иметь локальной памяти.</li>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingJmmVisibilitySynchronized')"><span class="textCode">synchronized</span> метод или блок</div>

<div id="coreMultithreadingJmmVisibilitySynchronized" class="hideContent">

<div class="text">

<li>при входе в <span class="textCode">synchronized</span> метод или блок поток обновляет содержимое локальной памяти;</li>

<li>при выходе из <span class="textCode">synchronized</span> метода или блока поток записывает изменения из локальной памяти в главную;</li>

<li>т. к. операции с памятью происходят раньше освобождения монитора и освобождение монитора происходит раньше захвата монитора, то все операции с памятью, которые были сделаны потоком до выхода из <span class="textCode">synchronized</span> блока должны быть видны любому потоку, который входит в <span class="textCode">synchronized</span> блок для того же самого монитора (правило работает, если потоки синхронизируются и используют один и тот же монитор).</li>

</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingJmmVisibilityVolatile')"><span class="textCode">volatile</span> переменные</div>

<div id="coreMultithreadingJmmVisibilityVolatile" class="hideContent">

<div class="text">

<li>запись в основную памят;</li>

<li>чтение из основной памяти (гарантированно вернет последнее записанное в нее значение).</li>

</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingJmmVisibilityFinal')"><span class="textCode">final</span> поля</div>

<div id="coreMultithreadingJmmVisibilityFinal" class="hideContent">

<div class="text">

<li>после создания объекта значения <span class="textCode">final</span> поля видет любой поток без дополнительной синхронизации;</li>

<li>«корректно создан» — ссылка на создающийся объект не должна использоваться до тех пор, пока не завершился конструктор объекта;</li>

<li>наличие такой семантики для ключевого слова <span class="textCode">final</span> позволяет создание неизменяемых (immutable) объектов, содержащих только <span class="textCode">final</span> поля, такие объекты могут свободно передаваться между потоками без обеспечения синхронизации при передаче.</li>

<li>проблема: реализация разрешает менять значения <span class="textCode">final</span> полей после создания объекта (reflection) (рекомендуется изменять только внутри конструктора).</li>

</div>

</div>

</div>

<li class="showHide" onClick="showHide('coreMultithreadingJmmReordering')">переупорядочивание (reordering).</li>

<div id="coreMultithreadingJmmReordering" class="hideContent">

<div class="text">Порядок выполнения операций может не соответствовать тому, как он представлен в исходном коде.</div>

<div class="text">чтения и записи <span class="textCode">volatile</span> переменных не могут быть переупорядочены с операциями чтения и записи другихe переменных (<span class="textCode">volatile</span> и не-<span class="textCode">volatil</span>) (можно использовать <span class="textCode">volatile</span> переменную как флага, сигнализирующем об окончании какого-либо действия).</div>

<div class="text">В остальном правила, касающиеся порядка выполнения операций, гарантируют упорядоченность операций для конкретного набора случаев (таких как, например, захват и освобождение монитора), во всех остальных случаях оставляя компилятору и процессору полную свободу для оптимизаций.</div>

</div>

</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingCompetitionAndConcurrency')">конкуренция и параллелизм</div>

<div id="coreMultithreadingCompetitionAndConcurrency" class="hideContent">

<div class="text">

Конкуренция — одновременное решение нескольких задач.

<li>несколько потоков;</li>

<li>недетерминированный результат.</li>

</div>

<div class="text">

Параллелизм — выполнения разных частей одной задачи.

<li>может быть и один;</li>

<li>детерминированный (определенный) результат.</li>

</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingMultitasking')">многозадачность (кооперативная (КМ) и вытесняющая (ВМ))</div>

<div id="coreMultithreadingMultitasking" class="hideContent">

<div class="text">Многозадачность — решение задач несколькими потоками. Разница — кто принимает решение о переключении между потоками процесса.</div>

<div class="text showHide" onClick="showHide('coreMultithreadingMultitaskingCooperative')">КМ — потоки.</div>

<div id="coreMultithreadingMultitaskingCooperative" class="hideContent">

<div class="text">

преимущества:

<li>простота реализации,</li>

<li>меньшие накладные расходы на переключение контекста.</li>

</div>

<div class="text">

недостатки:

<li>завис один поток или ведет себя некорректно — зависла вся система.</li>

</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingMultitaskingPreemptive')">ВМ (Java) — ОС.</div>

<div id="coreMultithreadingMultitaskingPreemptive" class="text hideContent">

<div class="text">

преимущества:

<li>отдельные потоки процесса не «подвешивают» всю систему,</li>

<li>регулярного переключения между задачами также улучшается отзывчивость приложения и повышается оперативность освобождения ресурсов.</li>

</div>

<div class="text">

недостатки:

<li>требует обработки системного прерывания от аппаратного таймера.</li>

</div>

</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingOrderingAsifserialSemanticssequentialconsistencyVisibilityAtomicityHappensbeforeMutualExclusionSafePublication')">ordering, as-if-serial semantics, sequential consistency, visibility, atomicity, happens-before, mutual exclusion, safe publication?</div>

<div id="coreMultithreadingOrderingAsifserialSemanticssequentialconsistencyVisibilityAtomicityHappensbeforeMutualExclusionSafePublication" class="hideContent">

<div class="text">

<div class="text"><span class="hе">ordering</span> — один поток может увидеть неверный порядок исполнения инструкций другого потока.</div>

<div class="text"><span class="hе">as-if-serial semantics</span> и <span class="hе">sequential consistency</span> — гарантии ordering.</div>

<div class="text"><span class="hе">visibility</span> действия в одном потоке становятся видны из другого потока.</div>

<div class="text showHide" onClick="showHide('coreMultithreadingHappensbefore')"><span class="hе">happens-before</span> — логическое ограничение на порядок выполнения инструкций программы (если указывается, что запись в переменную и последующее ее чтение связаны через эту зависимость, то как бы при выполнении не переупорядочивались инструкции, в момент чтения все связанные с процессом записи результаты уже зафиксированы и видны).</div>

<div id="coreMultithreadingHappensbefore" class="hideContent">

<script>document.write(linkHappensbefor.innerHTML)</script>

</div>

<div class="text"><span class="hе">atomicity</span> — операций атомарная (единая и неделимая; либо выполнена, либо нет).</div>

<div class="text"><span class="hе">mutual exclusion</span> (взаимоисключающая блокировка, семафор с одним состоянием) — доступ к ресурсу только одном потоку (<span class="textCode">synchronized(obj) { … }</span>).</div>

<div class="text">

<span class="hе">safe publication</span> — показ объектов другим потокам из текущего (без не нарушения ограничений visibility. Способы:

<li><span class="textCode">static{}</span> инициализатор;</li>

<li><span class="textCode">volatile</span> переменные;</li>

<li><span class="textCode">atomic</span> переменные;</li>

<li>сохранение в разделяемой переменной, корректно защищенной с использованием <span class="textCode">synchronized()</span>, синхронизаторов или других конструкций, создающих read/write memory barrier;</li>

<li><span class="textCode">final</span> переменные в разделяемом объекте, который был корректно проинициализирован.</li>

</div>

</div>

</div>

<div class="text showHide" onClick="showHide('Как проверить, удерживает ли поток монитор определенного ресурса?')">Как проверить, удерживает ли поток монитор определенного ресурса?</div>

<div id="Как проверить, удерживает ли поток монитор определенного ресурса?" class="hideContent">

<div class="text"><span class="textCode">Thread.holdsLock(lock)</span> возвращает <span class="textCode">true</span>, когда текущий поток удерживает монитор у определенного объекта.</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingSynchronizationWhenCallingTheStaticSynchronizedMethod')">На каком объекте происходит синхронизация при вызове <span class="textCode">static synchronized</span> метода?</div>

<div id="coreMultithreadingSynchronizationWhenCallingTheStaticSynchronizedMethod" class="hideContent">

<div class="text">У синхронизированного статического метода нет доступа к <span class="textCode">this</span>, но есть доступ к объекту класса <span class="textCode">Class</span>, он присутствует в единственном экземпляре и именно он выступает в качестве монитора для синхронизации статических методов. Таким образом, следующая конструкция:</div>

<script>document.write(codeCoreMultithreadingSynchronizationWhenCallingTheStaticSynchronizedMethod.innerHTML)</script>

<div class="text">эквивалентна такой:</div>

<script>document.write(codeCoreMultithreadingSynchronizationWhenCallingTheStaticSynchronizedMethod2.innerHTML)</script>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingCompareandswapAndWeakcompareandswap')"><span class="textCode">java.util.concurrent.Atomic\*.compareAndSwap()</span> и <span class="textCode">java.util.concurrent.Atomic\*.weakCompareAndSwap()</span></div>

<div id="coreMultithreadingCompareandswapAndWeakcompareandswap" class="hideContent">

<div class="text">

<span class="textCode">weakCompareAndSwap()</span>:

<li>не создает memory barrier и не дает гарантии happens-before;</li>

<li>сильно зависит от кэша/CPU, и может возвращать <span class="textCode">false</span> без видимых причин;</li>

<li>более легкая, но поддерживаемая далеко не всеми архитектурами и не всегда эффективная операция.</li>

</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingRaceCondition')">Состояние гонки (race condition)</div>

<div id="coreMultithreadingRaceCondition" class="hideContent">

<div class="text showHide" onClick="showHide('coreMultithreadingRaceConditionRaceCondition')">Состояние гонки (race condition)</div>

<div id="coreMultithreadingRaceConditionRaceCondition" class="hideContent">

<div class="text">Ошибка проектирования многопоточной системы или приложения, при которой эта работа напрямую 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.</div>

</div>

<div class="text showHide" onClick="showHide('Существует ли способ решения проблемы race condition?')">Существует ли способ решения проблемы race condition?</div>

<div id="Существует ли способ решения проблемы race condition?" class="hideContent">

<div class="text">Распространенные способы решения:</div>

<li class="text">Использование локальной копии — копирование разделяемой переменной в локальную переменную потока. Этот способ работает только тогда, когда переменная одна и копирование производится атомарно (за одну машинную команду), использование <span class="textCode">volatile</span>.</li>

<li class="text">Синхронизация — операции над разделяемым ресурсом происходят в синхронизированном блоке (при использовании ключевого слова <span class="textCode">synchronized</span>).</li>

<li class="text">Комбинирование методов — вышеперечисленные способы можно комбинировать, копируя «опасные» переменные в синхронизированном блоке. С одной стороны, это снимает ограничение на атомарность, с другой — позволяет избавиться от слишком больших синхронизированных блоков.</li>

<div class="text">Очевидных способов выявления и исправления состояний гонки не существует. Лучший способ избавиться от гонок — правильное проектирование многозадачной системы.</div>

</div>

</div>

<div class="text showHide" onClick="showHide('Что происходит, когда в потоке выбрасывается исключение?')">Что происходит, когда в потоке выбрасывается исключение?</div>

<div id="Что происходит, когда в потоке выбрасывается исключение?" class="hideContent">

<li class="text">Если исключение не поймано – поток «умирает» (переходит в состяние мертв (dead)).</li>

<li class="text">Если установлен обработчик непойманных исключений, то он возьмет управление на себя. <span class="textCode">Thread.UncaughtExceptionHandler</span> — интерфейс, определенный как вложенный интерфейс для других обработчиков, вызываемых, когда поток внезапно останавливается из-за непойманного исключения. В случае, если поток собирается остановиться из-за непойманного исключения, JVM проверяет его на наличие <span class="textCode">UncaughtExceptionHandler</span>, используя <span class="textCode">Thread.getUncaughtExceptionHandler()</span>, и если такой обработчик найдет, то вызовет у него метод <span class="textCode">uncaughtException()</span>, передав этот поток и исключение в виде аргументов.</li>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingThreadPool')">пул потоков</div>

<div id="coreMultithreadingThreadPool" class="hideContent">

<div class="text showHide" onClick="showHide('coreMultithreadingThreadPoolThreadPool')">пул потоков</div>

<div id="coreMultithreadingThreadPoolThreadPool" class="hideContent">

<div class="text">Создание потока является затратной по времени и ресурсам операцией. Количество потоков, которое может быть запущено в рамках одного процесса также ограниченно. Чтобы избежать этих проблем и в целом управлять множеством потоков более эффективно в Java был реализован механизм пула потоков (thread pool), который создается во время запуска приложения и в дальнейшем потоки для обработки запросов берутся и переиспользуются уже из него. Таким образом, появляется возможность не терять потоки, сбалансировать приложение по количеству потоков и частоте их создания.</div>

<div class="text">

Начиная с Java е.5 Java API предоставляет фреймворк <span class="textCode">Executor</span>, который позволяет создавать различные типы пула потоков:

<li><span class="textCode">Executor</span> — упрощенный интерфейс пула, содержит один метод для передачи задачи на выполнение;</li>

<li><span class="textCode">ExecutorService</span> — расширенный интерфейс пула, с возможностью завершения всех потоков;</li>

<li><span class="textCode">AbstractExecutorService</span> — базовый класс пула, реализующий интерфейс <span class="textCode">ExecutorService</span>;</li>

<li><span class="textCode">Executors</span> — фабрика объектов связанных с пулом потоков, в том числе позволяет создать основные типы пулов;</li>

<li><span class="textCode">ThreadPoolExecutor</span> — пул потоков с гибкой настройкой, может служить базовым классом для нестандартных пулов;</li>

<li><span class="textCode">ForkJoinPool</span> — пул для выполнения задач типа <span class="textCode">ForkJoinTask</span>;</li>

<li>... и другие.</li>

</div>

<div class="text">

Методы <span class="textCode">Executors</span> для создания пулов:

<li><span class="textCode">newCachedThreadPool()</span> — если есть свободный поток, то задача выполняется в нем, иначе добавляется новый поток в пул. Потоки не используемые больше минуты завершаются и удалются и кэша. Размер пула неограничен. Предназначен для выполнения множество небольших асинхронных задач;</li>

<li><span class="textCode">newCachedThreadPool(ThreadFactory threadFactory)</span> — аналогично предыдущему, но с собственной фабрикой потоков;</li>

<li><span class="textCode">newFixedThreadPool(int nThreads)</span> — создает пул на указанное число потоков. Если новые задачи добавлены, когда все потоки активны, то они будут сохранены в очереди для выполнения позже. Если один из потоков завершился из-за ошибки, на его место будет запущен другой поток. Потоки живут до тех пор, пока пул не будет закрыт явно методом <span class="textCode">shutdown()</span>.</li>

<li><span class="textCode">newFixedThreadPool(int nThreads, ThreadFactory threadFactory)</span> — аналогично предыдущему, но с собственной фабрикой потоков;</li>

<li><span class="textCode">newSingleThreadScheduledExecutor()</span> — однопотоковый пул с возможностью выполнять задачу через указанное время или выполнять периодически. Если поток был завершен из-за каких-либо ошибок, то для выполнения следующей задачи будет создан новый поток.</li>

<li><span class="textCode">newSingleThreadScheduledExecutor(ThreadFactory threadFactory)</span> — аналогично предыдущему, но с собственной фабрикой потоков;</li>

<li><span class="textCode">newScheduledThreadPool(int corePoolSize)</span> — пул для выполнения задач через указанное время или переодически;</li>

<li><span class="textCode">newScheduledThreadPool(int corePoolSize, ThreadFactory threadFactory)</span> — аналогично предыдущему, но с собственной фабрикой потоков;</li>

<li><span class="textCode">unconfigurableExecutorService(ExecutorService executor)</span> — обертка на пул, запрещающая изменять его конфигурацию;</li>

</div>

</div>

<div class="text showHide" onClick="showHide('Какого размера должен быть пул потоков?')">Какого размера должен быть пул потоков?</div>

<div id="Какого размера должен быть пул потоков?" class="hideContent">

<div class="text">Настраивая размер пула потоков, важно избежать двух ошибок: слишком мало потоков (очередь на выполнение будет расти, потребляя много памяти) или слишком много потоков (замедление работы всей систему из-за частых переключений контекста).</div>

<div class="text">Оптимальный размер пула потоков зависит от количества доступных процессоров и природы задач в рабочей очереди. На N-процессорной системе для рабочей очереди, которая будет выполнять исключительно задачи с ограничением по скорости вычислений, можно достигнуть максимального использования CPU с пулом потоков, в котором содержится N или N+е поток. Для задач, которые могут ждать осуществления I/O (ввода - вывода) - например, задачи, считывающей HTTP-запрос из сокета – может понадобиться увеличение размера пула свыше количества доступных процессоров, потому, что не все потоки будут работать все время. Используя профилирование, можно оценить отношение времени ожидания (<span class="textCode">WT</span>) ко времени обработки (<span class="textCode">ST</span>) для типичного запроса. Если назвать это соотношение <span class="textCode">WT/ST</span>, то для N-процессорной системе понадобится примерно <span class="textCode">N \* (е + WT/ST)</span> потоков для полной загруженности процессоров.</div>

<div class="text">Использование процессора — не единственный фактор, важный при настройке размера пула потоков. По мере возрастания пула потоков, можно столкнуться с ограничениями планировщика, доступной памяти, или других системных ресурсов, таких, как количество сокетов, дескрипторы открытого файла, или каналы связи базы данных.</div>

</div>

<div class="text showHide" onClick="showHide('Что будет, если очередь пула потоков уже заполнена, но подается новая задача?')">Что будет, если очередь пула потоков уже заполнена, но подается новая задача?</div>

<div id="Что будет, если очередь пула потоков уже заполнена, но подается новая задача?" class="hideContent">

<div class="text">Если очередь пула потоков заполнилась, то поданная задача будет «отклонена». Например - метод <span class="textCode">submit()</span> у <span class="textCode">ThreadPoolExecutor</span> выкидывает <span class="textCode">RejectedExecutionException</span>, после которого вызывается <span class="textCode">RejectedExecutionHandler</span>.</div>

</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingStackAndHeap')">Стек (stack) и куча (heap) с точки зрения многопоточности</div>

<div id="coreMultithreadingStackAndHeap" class="hideContent">

<div class="text">Cтек – участок памяти, тесно связанный с потоками. У каждого потока есть свой стек, которые хранит локальные переменные, параметры методов и стек вызовов. Переменная, хранящаяся в стеке одного потока, не видна для другого.</div>

<div class="text">Куча – общий участок памяти, который делится между всеми потоками. Объекты, неважно локальные или любого другого уровня, создаются в куче. Для улучшения производительности, поток обычно кэширует значения из кучи в свой стек, в этом случае для того, чтобы указать потоку, что переменную следует читать из кучи используется ключевое слово volatile.</div>

</div>

<div class="text showHide" onClick="showHide('Как поделиться данными между двумя потоками?')">Как поделиться данными между двумя потоками?</div>

<div id="Как поделиться данными между двумя потоками?" class="hideContent">

<div class="text">Использовать общий объект или параллельные структуры данных (<span class="textCode">BlockingQueue</span>).</div>

</div>

<div class="text showHide" onClick="showHide('Какой параметр запуска JVM используется для контроля размера стека потока?')">Какой параметр запуска JVM используется для контроля размера стека потока?</div>

<div id="Какой параметр запуска JVM используется для контроля размера стека потока?" class="hideContent">

<div class="text"><span class="textCode">-Xss</span></div>

</div>

<div class="text showHide" onClick="showHide('Как получить дамп потока?')">Как получить дамп потока?</div>

<div id="Как получить дамп потока?" class="hideContent">

<div class="text">Среды исполнения Java на основе HotSpot генерируют только дамп в формате HPROF. В распоряжении разработчика имеется несколько интерактивных методов генерации дампов и один метод генерации дампов на основе событий.</div>

<div class="text">

Интерактивные методы:

<li>Использование <span class="textCode">Ctrl+Break</span>: если для исполняющегося приложения установлена опция командной строки <span class="textCode">-XX:+HeapDumpOnCtrlBreak</span>, то дамп формата HPROF генерируется вместе с дампом потока при наступлении события <span class="textCode">Ctrl+Break</span> или <span class="textCode">SIGQUIT</span> (обычно генерируется с помощью kill -3), которое инициируется посредством консоли. Эта опция может быть недоступна в некоторых версиях. В этом случае можно попытаться использовать следующую опцию: <span class="textCode">-Xrunhprof:format=b,file=heapdump.hprof</span>.</li>

<li>Использование инструмента jmap: утилита jmap, поставляемая в составе каталога <span class="textCode">/bin/</span> комплекта JDK, позволяет запрашивать дамп HPROF из исполняющегося процесса.</li>

<li>Использование операционной системы: Для создания файла ядра можно воспользоваться неразрушающей командой gcore или разрушающими командами kill -6 или kill -ее. Затем извлечь дамп кучи из файла ядра с помощью утилиты jmap.</li>

<li>Использование инструмента JConsole. Операция <span class="textCode">dumpHeap</span> предоставляется в JConsole как MBean-компонент <span class="textCode">HotSpotDiagnostic</span>. Эта операция запрашивает генерацию дампа в формате HPROF.</li>

</div>

<div class="text">

Метод на основе событий:

<li>Событие <span class="textCode">OutOfMemoryError</span>: Если для исполняющегося приложения установлена опция командной строки <span class="textCode">-XX:+HeapDumpOnOutOfMemoryError</span>, то при возникновении ошибки <span class="textCode">OutOfMemoryError</span> генерируется дамп формата HPROF. Это идеальный метод для «production» систем, поскольку он практически обязателен для диагностирования проблем памяти и не сопровождается постоянными накладными расходами с точки зрения производительности. В старых выпусках сред исполнения Java на базе HotSpot для этого события не устанавливается предельное количество дампов кучи в пересчете на одну JVM; в более нов</li>

</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingThreadLocalVariable')">ThreadLocal-переменная</div>

<div id="coreMultithreadingThreadLocalVariable" class="hideContent">

<div class="text"><span class="textCode">ThreadLocal</span> — класс, позволяющий имея одну переменную, иметь различное ее значение для каждого из потоков.</div>

<div class="text">У каждого потока — т.е. экземпляра класса <span class="textCode">Thread</span> — есть ассоциированная с ним таблица ThreadLocal-переменных. Ключами таблицы являются cсылки на объекты класса <span class="textCode">ThreadLocal</span>, а значениями - ссылки на объекты, «захваченные» ThreadLocal-переменными, т.е. ThreadLocal-переменные отличаются от обычных переменных тем, что у каждого потока свой собственный, индивидуально инициализируемый экземпляр переменной. Доступ к значению можно получить через методы <span class="textCode">get()</span> или <span class="textCode">set()</span>.</div>

<div class="text">Например, если мы объявим ThreadLocal-переменную: <span class="textCode">ThreadLocal&#60;Object> locals = new ThreadLocal&#60;Object>();</span>. А затем, в потоке, сделаем <span class="textCode">locals.set(myObject)</span>, то ключом таблицы будет ссылка на объект <span class="textCode">locals</span>, а значением - ссылка на объект <span class="textCode">myObject</span>. При этом для другого потока существует возможность «положить» внутрь <span class="textCode">locals</span> другое значение.</div>

<div class="text">Следует обратить внимание, что <span class="textCode">ThreadLocal</span> изолирует именно ссылки на объекты, а не сами объекты. Если изолированные внутри потоков ссылки ведут на один и тот же объект, то возможны коллизии.</div>

<div class="text">Так же важно отметить, что т.к. ThreadLocal-переменные изолированы в потоках, то инициализация такой переменной должна происходить в том же потоке, в котором она будет использоваться. Ошибкой является инициализация такой переменной (вызов метода <span class="textCode">set()</span>) в главном потоке приложения, потому как в данном случае значение, переданное в методе <span class="textCode">set()</span>, будет «захвачено» для главного потока, и при вызове метода <span class="textCode">get()</span> в целевом потоке будет возвращен <span class="textCode">null</span>.</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingDoubleCheckedLockingSingleton')">double checked locking Singleton</div>

<div id="coreMultithreadingDoubleCheckedLockingSingleton" class="hideContent">

<div class="text">Один из способов создания потокобезопасного класса, реализующего шаблон Одиночка. Данный метод пытается оптимизировать производительность, блокируясь только случае, когда экземпляр одиночки создается впервые.</div>

<script>document.write(codeCoreMultithreadingDoubleCheckedLockingSingleton.innerHTML)</script>

<div class="text">Следует заметить, что требование <span class="textCode">volatile</span> обязательно. Проблема Double Checked Lock заключается в модели памяти Java, точнее в порядке создания объектов, когда возможна ситуация, при которой другой поток может получить и начать использовать (на основании условия, что указатель не нулевой) не полностью сконструированный объект. Хотя эта проблема была частично решена в JDK е.5, однако рекомендация использовать <span class="textCode">voloatile</span> для Double Cheсked Lock остается в силе.</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingThreadSafeSingleton')">создание потокобезопасного Singleton</div>

<div id="coreMultithreadingThreadSafeSingleton" class="hideContent">

<div class="text">Static field</div>

<script>document.write(codeCoreMultithreadingThreadSafeSingleton.innerHTML)</script>

<div class="text">Enum</div>

<script>document.write(codeCoreMultithreadingThreadSafeSingleton2.innerHTML)</script>

<div class="text">Synchronized Accessor</div>

<script>document.write(codeCoreMultithreadingThreadSafeSingleton3.innerHTML)</script>

<div class="text">Double Checked Locking & volatile</div>

<script>document.write(codeCoreMultithreadingThreadSafeSingleton4.innerHTML)</script>

<div class="text">On Demand Holder Idiom</div>

<script>document.write(codeCoreMultithreadingThreadSafeSingleton5.innerHTML)</script>

</div>

<div class="text showHide" onClick="showHide('Чем полезны неизменяемые объекты?')">Чем полезны неизменяемые объекты?</div>

<div id="Чем полезны неизменяемые объекты?" class="hideContent">

<div class="text">Неизменяемость (immutability) помогает облегчить написание многопоточного кода. Неизменяемый объект может быть использован без какой-либо синхронизации. К сожалению в Java нет аннотации <span class="textCode">@Immutable</span>, которая делает объект неизменяемым, для этого разработчикам нужно самим создавать класс с необходимыми характеристиками. Для этого необходимо следовать некоторым общим принципам: инициализация всех полей только конструкторе, отсутствие методов <span class="textCode">setX()</span> вносящих изменения в поля класса, отсутствие утечек ссылки, организация отдельного хранилища копий изменяемых объектов и т.д.</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingBusySpin')">busy spin</div>

<div id="coreMultithreadingBusySpin" class="hideContent">

<div class="text">Техника, которую программисты используют, чтобы заставить поток ожидать при определенном условии. В отличие от традиционных методов <span class="textCode">wait()</span>, <span class="textCode">sleep()</span> или <span class="textCode">yield()</span>, которые подразумевают уступку процессорного времени, этот метод вместо уступки выполняет пустой цикл. Это необходимо, для того, чтобы сохранить кэш процессора, т.к. в многоядерных системах, существует вероятность, что приостановленный поток продолжит свое выполнение уже на другом ядре, а это повлечет за собой перестройку состояния процессорного кэша, которая является достаточно затратной процедурой.</div>

</div>

<div class="text showHide" onClick="showHide('Перечислите принципы, которым вы следуете в многопоточном программировании?')">Перечислите принципы, которым вы следуете в многопоточном программировании?</div>

<div id="Перечислите принципы, которым вы следуете в многопоточном программировании?" class="hideContent">

<div class="text">При написании многопоточных программ следует придерживаться определенных правил, которые помогают обеспечить достойную производительность приложения в сочетании с удобной отладкой и простотой дальнейшей поддержки кода.</div>

<li class="text">Всегда давайте значимые имена своим потокам. Процесс отладки, нахождения ошибок или отслеживание исключения в многопоточном коде – довольно сложная задача. <span class="textCode">OrderProcessor</span>, <span class="textCode">QuoteProcessor</span> или <span class="textCode">TradeProcessor</span> намного информативнее, чем <span class="textCode">Threadе</span>, <span class="textCode">Thread2</span> и <span class="textCode">Thread3</span>. Имя должно отражать задачу, выполняемую данным потоком.</li>

<li class="text">Избегайте блокировок или старайтесь уменьшить масштабы синхронизации. Блокировка затратна, а переключение контекста еще более ресурсоемко. Пытайтесь избегать синхронизации и блокировки насколько это возможно, и организуйте критическую секцию в минимально необходимом объеме. Поэтому синхронизированный блок всегда предпочительней синхронизированного метода, дополнительно наделяя возможностью абсолютного контроля над масштабом блокировки.</li>

<li class="text">Обрабатывайте прерывание потока с особой тщательностью. Нет ничего хуже оставшегося заблокированным ресурса или системы в неконстистентном, по причине неподтвержденной транзакции, состоянии.</li>

<li class="text">Помните об обработке исключений. Выброшенные <span class="textCode">InterruptedException</span> должны быть адекватно обработаны, а не просто подавлены. Так же не стоит пренебрегать <span class="textCode">Thread.UncaughtExceptionHandler</span>. При использовании пула потоков необходимо помнить, что он зачастую просто «проглатывает» исключения. Так, если вы отправили на выполнение <span class="textCode">Runnable</span> нужно обязательно поместить код выполнения задачи внутрь блока <span class="textCode">try-catch</span>. Если в очередь пула помещается <span class="textCode">Callable</span>, необходимо удостоверится, что результат выполнения всегда изымается помощью блокирующего <span class="textCode">get()</span>, чтобы в случае возникновения существовала возможнотсь заново выбросить произошедшее исключение.</li>

<li class="text">Между синхронизаторами и <span class="textCode">wait()</span> и <span class="textCode">notify()</span> следует выбирать синхронизаторы. Во-первых, синхронизаторы, типа <span class="textCode">CountDownLatch</span>, <span class="textCode">Semaphore</span>, <span class="textCode">CyclicBarrier</span> или <span class="textCode">Exchanger</span> упрощают написание кода. Очень сложно реализовывать комплексный управляющий поток, используя <span class="textCode">wait()</span> и <span class="textCode">notify()</span>. Во-вторых, эти классы написаны и поддерживаются настоящими мастерами своего дела и есть шанс, что в последующих версиях JDK они будут оптимизированы изнутри или заменены более производительной внешней реализацией.</li>

<li class="text">Почти всегда использование Concurrent сollection выгоднее использования Synchronized сollection, т.к. первые более современны (используют все доступные на момент их написания новшества языка) и масштабируемы, чем их синхронизированые аналоги.</li>

</div>

<div class="text showHide" onClick="showHide('Какое из следующих утверждений о потоках неверно?')">Какое из следующих утверждений о потоках неверно?</div>

<div id="Какое из следующих утверждений о потоках неверно?" class="hideContent">

<li class="text">Если метод <span class="textCode">start()</span> вызывается дважды для одного и того же объекта <span class="textCode">Thread</span>, во время выполнения генерируется исключение.</li>

<li class="text">Порядок, в котором запускались потоки, может не совпадать с порядком их фактического выполнения.</li>

<li class="text">Если метод <span class="textCode">run()</span> вызывается напрямую для объекта <span class="textCode">Thread</span>, во время выполнения генерируется исключение.</li>

<li class="text">Если метод <span class="textCode">sleep()</span> вызывается для потока, во время выполнения синхронизированного кода, блокировка не снимается.</li>

<div class="text">Правильный ответ: 3. Если метод <span class="textCode">run()</span> вызывается напрямую для объекта <span class="textCode">Thread</span>, во время выполнения исключение не генерируется. Однако, код, написанный в методе <span class="textCode">run()</span> будет выполняться текущим, а не новым потоком. Таким образом, правильный способ запустить поток – это вызов метода <span class="textCode">start()</span>, который приводит к выполнению метода <span class="textCode">run()</span> новым потоком.</div>

<div class="text">Вызов метода <span class="textCode">start()</span> дважды для одного и того же объекта <span class="textCode">Thread</span> приведет к генерированию исключения <span class="textCode">IllegalThreadStateException</span> во время выполнения, следовательно, утверждение е верно. Утверждение 2 верно, так как порядок, в котором выполняются потоки, определяется Планировщиком потоков, независимо от того, какой поток запущен первым. Утверждение 4 верно, так как поток не освободит блокировки, которые он держит, когда он переходит в состояние Ожидания.</div>

</div>

<div class="text showHide" onClick="showHide('Даны 3 потока Те, Т2 и Т3? Как реализовать выполнение в последовательности Те, Т2, Т3?')">Даны 3 потока Те, Т2 и Т3? Как реализовать выполнение в последовательности Те, Т2, Т3?</div>

<div id="Даны 3 потока Те, Т2 и Т3? Как реализовать выполнение в последовательности Те, Т2, Т3?" class="hideContent">

<div class="text">Такой последовательности выполнения можно достичь многими способами, например просто воспользоваться методом <span class="textCode">join()</span>, чтобы запустить поток в момент, когда другой уже закончит свое выполнение. Для реализации заданной последовательности, нужно запустить последний поток первым, и затем вызывать метод <span class="textCode">join()</span> в обратном порядке, то есть Т3 вызывает <span class="textCode">Т2.join</span>, а Т2 вызывает <span class="textCode">Те.join</span>, таким образом Те закончит выполнение первым, а Т3 последним.</div>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingTasks')">задачи</div>

<div id="coreMultithreadingTasks" class="hideContent">

<div class="text showHide" onClick="showHide('coreMultithreadingTasksMinimalNonblockingStack')">Напишите минимальный неблокирующий стек (всего два метода — <span class="textCode">push()</span> и <span class="textCode">pop()</span>).</div>

<div id="coreMultithreadingTasksMinimalNonblockingStack" class="hideContent">

<script>document.write(codeCoreMultithreadingMinimalNonblockingStack.innerHTML)</script>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingMinimalNonblockingStackSemaphore')">Напишите минимальный неблокирующий стек (всего два метода — <span class="textCode">push()</span> и <span class="textCode">pop()</span>) с использованием <span class="textCode">Semaphore</span>.</div>

<div id="coreMultithreadingMinimalNonblockingStackSemaphore" class="hideContent">

<script>document.write(codeCoreMultithreadingMinimalNonblockingStackSemaphore.innerHTML)</script>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingMinimalNonblockingArraylist')">Напишите минимальный неблокирующий ArrayList (всего четыре метода — <span class="textCode">add()</span>, <span class="textCode">get()</span>, <span class="textCode">remove()</span>, <span class="textCode">size()</span>).</div>

<div id="coreMultithreadingMinimalNonblockingArraylist" class="hideContent">

<script>document.write(codeCoreMultithreadingMinimalNonblockingArraylist.innerHTML)</script>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingThreadSafeClassImplementation')">Напишите потокобезопасную реализацию класса с неблокирующим методом <span class="textCode">BigInteger next()</span>, который возвращает элементы последовательности: <span class="textCode">[е, 2, 4, 8, е6, ...]</span>.</div>

<div id="coreMultithreadingThreadSafeClassImplementation" class="hideContent">

<script>document.write(codeCoreMultithreadingThreadSafeClassImplementation.innerHTML)</script>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingSimplestMultithreadedLimitedBuffer')">Напишите простейший многопоточный ограниченный буфер с использованием <span class="textCode">synchronized</span>.</div>

<div id="coreMultithreadingSimplestMultithreadedLimitedBuffer" class="hideContent">

<script>document.write(codeCoreMultithreadingSimplestMultithreadedLimitedBuffer.innerHTML)</script>

</div>

<div class="text showHide" onClick="showHide('coreMultithreadingSimplestMultithreadedLimitedBufferReentrantlock')">Напишите простейший многопоточный ограниченный буфер с использованием <span class="textCode">ReentrantLock</span>.</div>

<div id="coreMultithreadingSimplestMultithreadedLimitedBufferReentrantlock" class="hideContent">

<script>document.write(codeCoreMultithreadingSimplestMultithreadedLimitedBufferReentrantlock.innerHTML)</script>

</div>

</div>

</div>