Потоки

- Поток демон - представляет некие фоновые услуги при выполнении основной команды
- Поток демон не является необходимой часть программы. С завершением потоков не демонов завершается
- остальная часть программы. Запуск потока не демона - начало программы
- Можно явно
- Установка флага демона через точку setDeamon(true)
- Все статические методы создания перегружаются для получание объекта ThreadFactory
- Чтобы узнать является ли поток демоном вызовите метод isDeamon()
- Если поток демон, то все потоки, которые он создает тоже
- демоны.

- Затем поток переходит в бесконечный цикл, на каждом шаге которого метод yield()
- Потоки-демоны завершают свои методы run() без выполнения блока finally
- Если закомментировать вызов isDaemon() то блок finally будет выполнен

- Затем пооток переходит в бесконечный цикл, на каждом шаге которого метод yield()
- Потоки-демоны завершают свои методы run() без выполнения блока finally
- Если закомментировать вызов stDaemon() то блок finally будет выполнен
- Разновидности реализации
- Во всех примерах все классы задач реализуют Runnable
- Имена объектов Thread задаются вызовом конструктора Thread
- Для обращение имени в toString используется имя getName
- В классе InnerThread2 представлена альтернатива - внутренний субкласс Thread
- Чтоб дождаься завершения другого потока перед своим продолжением
- Поток может вызвать метод join
- Группы потоков
- Группа потоков хранит совокупность потоков
- Не надо использовать
- Перехват исключений
- Нельзя перехватывать исключения, возбужденные из потока
- Проблему можно решить при помощи объектов Executor
- exec.execute(new ExceptionThread())

- Совместное использование ресурсов
- Однопоточная программа выполняет по одной операции
- за один раз
- При многопоточности возможна ситуация в необходимости
- сразу нескольких ресурсов
- Некорректный доступ к ресурсам
- Задача не может зависеть от другой задачи, тк порядок
- завершения задач не гарантирован
- Разрешение спора за разделяемые ресурсы может осуществляться
- за счет блокировки
- Все многопоточне схемы синхронизируют доступ к разделяемым ресурсам
- Доступ к разделяемому ресурсу в один момент времени может получить
- только одна задача
- Создает эффект взаимного исключения называется мьютексом
- Встроенная поддержка для предотвращения спора за ресурсы в
- виде ключевого слова synchronized
- Обычно поля класса закрытые private, доступ к их памяти только
- посредством методов
- Можно предотвратить конфликты, объявив такие методы синхронизированными
- Каждый объект содерит объект простой блокировки - монитор
- автоматически является частью этого объекта
- При пометки метода словом synchronized не позволяет другим
- синхронизированнм методом быть вызванным до завершения задачи

- Существует отдельная блокировка уровня класса следящая за тем, чтоб
- статические синхронизированные методы не использовали общие стат данные класса
- Использование синхронизации
- Если вы записываете данные в переменную, которая может быть прочитана
- другим потоком или читаете данные из переменной, которая могла быть записана другим потоком,
- должны использовать синхронизацию
- Использование объектов Lock
- Операция создания установления и снятия блокировки с объектом Lock выполняется явно
- Такой подход обладает большей гибкость. при решении некоторых видов задач
- Пример переписанный для явного использования объектов Lock
- В блоке try должна находиться команда return которая гарантирует, что команда Unlock не будет выполнена слишком рано
- Если при использовании ключевого слова synchronized произойдет ошибка
- программа выдает исключение
- С ключевым словом synchronized невозможно отработать неудачную попытку получения блокировки
- необходимо использовать библиотеку concurrent
- Атомарная операция - операция, которую не может прервать планировщик потоков
- Атомарные операции не могут прерываться механизмом потоков
- Обозначаются ключевым словом volatile
- Если поле объявляется с ключевым словом volatile сразу поале записи
- значения в это поле, все операции чтения увидт изменение
- Ключевое слово volatile не работает в том случае, когда значение поля
- зависит от его предыдущего значения

- Инкремент не является атомарным и состоит из чтения и хаписи
- Если вы объявите переменную как volatile, то тем самым указываете компилятору
- не проводить оптимизацию, это приведет у удалению операций чтения и записи обеспечивающих синхронизацию
- поля с локальными данными потока
- Поля следует объявлять volatile тогда, когда к нему могут одновременно
- обращаться несколько залач и одно из этих обращений является записью
- Безопасные атомарные операции - чтение и присвоение примитивов
- Атомарные классы
- Специальные классы атомарных переменных
- Предназначены для использования оптимизаций атомарноти машинного
- уровня, доступных на некоторых современных процессорах
- Критические секции
- Чтобы предотвратить доступ нескольких потоков только к части кода
- а не к методу в целом. Фрагмент кода, который изолируется таким способом
- - критическая секция. Для его создание ключевое слово synchronized(тут что-то){может обращаться только одна задача в момент времени}
- - синхронизированная блокировка
- Синхронизация по другим объектам
- Для синхронизированного блока должен быть указан объект, по которому
- осуществляется синхронизация
- самый разумный выбор - текущий объект, для которого вызывается метод -
- synchronized(this)

- Если синхориназцию приходится выполнять по другому объекту, но в этом случае
- должны убедиться в том, что все задейтвованные задачи
- синхронизируют по одному объекту
- Локальная память потоков
- Способ предотвращения конкуренции задач за совместные ресурсы -
- устранение совместного использования
- Локальная память потока - механизм, автоматически создает для
- одной переменной несколько блоков памяти
- Количество потоков использующих объект равно количеству блоков памяти
- Это позволяет связать с каждым потоком некоторое состояние
- Объекты ThreadLocal храняться в статических полях, занимается созданием и управлением
- памяти потоков
- При создании объекта можно обратиться к содержимому методами гет сет

- Завершение задач
- isCanceled - используется при проверке необходимости завершения

- использование фукции yield - позволяет другим потокам получить время выполнение
- потока, у которого вызван yield
- Вызывается в методе run

- *A happens-before B
- - все изменения выполненные операцией а видны при выполнении операции В
- -> в многопоточной программе одни события будут происходить до появления других
- start happens before run
- метод sleep - один из вариантов остановки потока

- Стостояние потока
- Переходное - во время этого состояния поток получает все необходимые
- системные ресурсы и выполняет инициализацию
- Активное - если у процессора есть свободная память для запуска потока
- поток будет выполняться
- Блокировки - поток заблокирован, номожет выполняться
- Завершенное - задача потока завершена и он не может стать активным
- Одним из способов перехода в завершенное состояние - возврат из
- метода run()
- Переход в блокированное состояние
- Блокировка с помозью sleep - бездействие заданное время
- Блокировка методом wait - поток будет простаивать до тех пор, пока не получит
- сообщение о возобновлении работы notify notifyall
- Поток ожидает завершение ввода-вывода
- Пытается вызват синхронизированный метод другого объекта, но его объект блокировки
- недоступен
- Иногда требуется принудительное завершение заблокированной задачи

- Прерывание
- При выходе из заблокированной задачи может возникнуть необходимость в освобождении ресурсов
- -> выход из середины мнтода run - использование исключений
- Чтобы вернуться в заведомо допутимое состояние при завершении задачи
- необходимо тщательно проанализировать пути выполнения кода и написать
- условие catch для освобождения всех ресурсов.
- Для завершения заблокированных задач в класс Thread был включен метод
- interrupt() устанавливающий сотояние прерывания
- при вызове метода если поток находился в состоянии ожидания программа вызовет
- InterruptedException
- Класс, реагирующий на interrput()
- Если поток был в работоспособном состоянии - установка флага, а после
- самостоятельная проверка флагов и завершение
- Поток с установленным состоянием прерывание выдает исключение, если он уже заюлокирован или пытается выполнить
- блокирующую ситуацию
- Чтобы вызвать interrupt необходимо иметь ссылку на объект Thread - способ выхода
- из run без возбуждение исключений
- Чтобы вызвать interrupt необходимо иметь ссылку на объект Thread
- Каждая задача представляет собой разновидность блокировки
- SleepBlock - пример прерываемой блокировки, IObBlocked и SynchronizedBlocked
- - непрерываемая блокировка
- Из выходных данных видно, что вызов sleep можно прервать , но задачу
- пытающуюся прервать синхрониз блокировку или выполнть ввод-вывод прервать нельзя
- Эффективное решение задачи - закрытие ресурса по которому блокировалась
- задача
- прямой вызов метода run не имеет отношения к многопоточности. сначала start
- метод join - гарантия выполнения потоков

- Чтобы взаимодействовать между задачами, необходимо корректно использовать
- русерсы задач. Для этого поведение синхронизируется с помощью
- объекта блокировки
- Для согласование - установления последовательности выполнения задач -
- тоже используют мьютекс, гарантирующий, что только одна задача может
- ответить на сигнал
- Также используют механизм, приастоновки задачи до изменения внешнего
- состояния. Согласование между задачами реализуется с помощью
- wait и notifyAll(). Можно использовать объекты Condition с методами
- await() signal().
- wait notify
- wait - позволяет дождаться изменения внешнего условия
- Активное ожидание - регулярная проверка условия - неэффективно ->
- wait только останавливает задачу, notify - активизирует задачу и проверяет изменения
- wait - механизм синхронизации действий между задачами
- sleep - не освобождает объект блокировки, yeield - тоже
- Когда задача входит в wait внутри метода выполнение потока приостанавливается,
- блокировка снимается
- Блокировка может быть получена другой залачи, остальные потоки могут получиить доступ
- к другим объектам во время wait. -> ожидание означает предоставление доступа к другим синхронизированным
- операциям

- Также используют механизм, приастоновки задачи до изменения внешнего
- состояния. Согласование между задачами реализуется с помощью
- wait и notifyAll(). Можно использовать объекты Condition с методами
- await() signal().
- wait notify
- wait - позволяет дождаться изменения внешнего условия
- Активное ожидание - регулярная проверка условия - неэффективно ->
- wait только останавливает задачу, notify - активизирует задачу и проверяет изменения
- wait - механизм синхронизации действий между задачами
- sleep - не освобождает объект блокировки, yeield - тоже
- Когда задача входит в wait внутри метода выполнение потока приостанавливается,
- блокировка снимается
- Блокировка может быть получена другой залачи, остальные потоки могут получиить доступ
- к другим объектам во время wait. -> ожидание означает предоставление доступа к другим синхронизированным
- операциям
- Есть две формы метода wait - с установленным временем (выход по времени)
- с пустым аргументом - выход при уведомлениии notify
- wait и notify пренадлежат к классу object, а не к потоку
- wait допустимо вызывать в любом синхронизированном методе\блоке
- При вызове wait в не синх блоке - исключение IllegalManitorException ->
- поток, вызывающий методы wait должен владеть объектом блокировки
- Чтобы провести операции блокировки необходимо захватить объект блокировки
- с помощью synchronized(object){object.wait()}
- Вызов wait должен быть заключен в цикл while в условиях короторой - причина блокировки объекта
- while(!условие){do smth}
- Необходимо чтобы программа проверяла интересующее условие
- Пропущенные сигналы
- wait в цикле ожидания
- перед синхронизированным методом вызов другого синх метода, настроивающего
- условия с уведомлением
- Сразу после идет не проверка условия а синх метод ожидания ->
- -> ожидание происходит неопр. время
- Решение -> добавить цикл while в синх метод

- notify() notifyAll()
- notify() - активизирует только одну ожидающую задачу - перед
- вызовом необходимо убедиться в активизации правильной задачи
- при использовании notify все задачи должны ожидать по одному условию
- То же самое для всех субклассов
- Если не то notifyAll
- notifyAll активизируются задачи, ожидающие конкретной блокировки
- Производители и потребители
- Потребитель - использует ресурсы производителя для выполнения задач, ожидает возобновление
- ресурсов у производителя
- Явное использование объектов Lock и Condition
- Condition - использует мьютекс, обеспечивает возможноть await
- при изменение внешнего состояния задача оповещается вызовом signal()
- signalAll()
- использование класса lock - для создание объекта Condition

- Производители-потребители и очереди
- Проблему, решаемую методами wait и notify можно перенести на более абстрактный уровень
- используя синхронизированную очередь, разрешающую выполнение и удаление вставки только
- одной задаче
- Используется очередь неогранниченого размера. Есть ограниенного через Array
- Очереди приостанавливают задачу в случае попытки получение объекта из пустой очереди.
- Возобновляют выполнение при появление элементов
- Пример реализации - извлечение объектов из очереди и их выполнение в потоке.
- Использование каналов для ввода-вывода между потоками
- Организацие взаимодействия потоков посредствам ввода-вывода
- поддержка ввода-вывода в форме каналов
- канал можно сопоставить с блокирующей очередью

- Взаимная блокировка
- deadblock - один поток ожидает освобождение первого, который ожидает
- освобождение второго
- Обстоятельства возникновения взаимной блокировки
- Взаимное исключение - использование одного и того же ресурса несколькими потоками
- Удержание - один из процессов должен удерживать ресурс и ожидать ресурс другого процесса
- Все ресурсы освобождаются естественным путем
- Круговое ожидание
- Для того, чтобы не было вз блок надо исключить одно из условий
- Нет специальных компонентов исключающих тупики
- Новые библиотечные компоненты
- CountDownLatch
- Предназначен для синхронизации одной нескольких задач, ожидающих
- завершение набора операций, выполняемых другими задачами
- Получает исходное значение счетчика
- Любая задача, вызывающая await - блокируется до тех пор, пока счетчик не уменьшится
- до нуля. Вызов countDown у объекта для уменьшения значения счетчика
- CountDownLatch - без сбрасывания счетчика CyclicBarrier - сбрасывает
- задачи, вызывающие countDown - не блокируются, блокируются задачи, вызывающие await
- Пример использование - разбиение задачи на подзадачи, которые необходим выполнить

- Потоковая безопасность не обеспечивается для каждой библиотеки java
- CyclicBarrier
- Используется в тех ситуациях, когда вы хотите создать группу параллельно работающих задача
- PriorityBlockingQueue
- Приоритная очередь с блокирующими операцимя выборки
- Объектами могут быть задачи, извлекаемые из очереди в порядке приоритетов
- Семафор -
- отличается от обычной блокировки наличием счетчика, который показывает, сколько
- задач может обращаться к ресурсу одновременно
- Exchanger - барьер, который меняет местами объекты двух задач
- Когда задача входит в барьер она использует один объект, при выходе -
- использует объект, который раньше удерживался другой задачей
- Объекты Exchanger используются тогда, когда одна задача выполняет
- затратные операции создания ообъектов а другая потребляет -
- возможна ситуация параллельного создания и потребления
- При вызове exchange() у объекта Exchanger - задача блокируется, пока
- другой объект не вызовет такой же метод

- Моделирование - каждый компонент в парралельной модели может быть
- представлен отдельной задачей
- К объектам предназначенным только для чтения не обязатльно применять
- синхронизацию или volatile

- Сравнение мьютексов
- Вызовы синхронизированного метода быстрее, чем вызовы ReentrantLock, в случае
- ситуации микрохронометража - тестирование функции вне контекста -
- синхронизация быстрее при единичном использование. Выводы о
- скорости действия стоит делать учитыва следующее
- Различия в производительности проявляются только при конкуренции
- за мьютексы -> в программе должны использоваться
- множественные задачи
- Встречая слово synchronized компилятор может применять спец оптимизации
- Чтобы полноценно сделать вывод о скорости работы нужно проводить тестирование
- следующим образом
- Создание нескольких задач, не только изменяющих, но и читающих значения
- Тогда видно, что решение Lock - превосходят по эффективности
- решения с synchronized, в широких пределах затраты, направленные
- на synchronized увеличиваются, а с Lock они постоянны
- Использовать synchronized стоит, когда тела защищаемых методов
- чрезвычайно малы + synchronized понятнее, чем установление блокировки
- через try и снятие в finally
- Контейнеры без блокировок
- Разработчик сам решает, стоит ли включать дополнительные
- действия по осуществлению синхронизации при использовании контейнеров
- Появились новые типы, для потоковых операций, использующие
- следующую стратегию
- изменение в контейнерах и чтение происходит только при условии,
- что читатели видят только результат завершенных изменений

- Оптимистическая блокировка
- Прежде чем использовать надо проверить болле простые варианты и убедиться в необходимости оптимизации
- Некоторые классы Atomic - атомарные операции, выполняются за единичное кочисетво памяти,
- Оптимистическая блокировка означает, что мьютекс при вычислениях
- реально не используется, но после завершение вычислений используется метод compareanset
- ему передается старое и новое значение, если старое не согласуется со значением в объекте - операция неудачна - другая задача успела изменить объект
- В общем случае - мьютекс для предотвр одновременной модификации объекта
- В данной ситуации оптимизм - данные свободны
- Все это ради производительности

- Если операция сравн будет неудачна нужно решать самим что делать - возобновлять сравнение или пропустить

- Класс ReadWriteLock
- Оптимизирует ситуацию с относительно редкой записью и частым чтением из структуры данных
- Несколько задач могут читать даннык при условии что ни одна задача не пытается их записать.
- При установлении блокировки записи, чтение не возможно до ее освобождения
- Приимущества проявляются при больших операциях
- ReadWriterList - внутри два класса read и write
- метод set - блокировка записи, чоб вызвать ArrayList.set()
- метод get - блокировка чтенеия
- Сложно, применять в случае необходимости повышения производительности

- Активные объекты
- Альтернатива большому количеству потоков - активные объекты
- Каждый объект поддерживает собственный рабочий поток и очередь сообщений
- Все запросы ставятся в очередь для последовательного выполнения
- При использовании активных объектов - обработка сообщенийа не методов
- При отправке сообщения оно преобр в задачу, которая помещается в очередь
- для дальнейшей релизации
- Активные компоненты
- 1 Каждый имеет собственный рабочий поток
- 2 Каждый полностью контролирует доступ к полям
- 3 Все взаимодействия проходят в форме сообщений
- 4 Все сообщения помещаются в очередь

- Итог
- 1 В програме могут выполняться несколько независимых задач
- 2 Разработчик должен проанализровать все возможные проблемы при завершении задач
- 3 Задачи могут взаимодействовать друг с другом через общие ресурсы. Для предотвращения
- конфлитов использование блокировок
- 4 Взаимные блокировки могут быть в плохо спроектированной программе
- Причины для использования паралелльного программирования
- Возможность управления несколькими подзадачами
- улучшение структуры кода
- удобство для конечного пользования
- Дополниткльно - замена тяжолого переключения контекста процессов легким переключением контекста выполнения
- тк все потоки - одно и то же пространство памяти
- Недостатки многозадачности
- 1 Замедленме программы связанное с ожиданием освобождения блокированных ресурсов
- 2 Доп нагрузка на процессор для управления потоками
- 3 Ненужна сложность как следствие неудачнвх решений
- 4 Голодание - имеет место,когда один или более потокв блокируются в получении доступа к ресурсу
- и не могут вследствиии этого двигаться дальше , соствязание за ресурсы
- взаимные блокировки
- 5 Непоследовательное поведение на различных платформах.
- Одно из самых больших затруднений с потоками возникает когда несколько потоков одновременно
- пытаются использовать один и тотже ресурс
- Количество потоков ограничено (зависит от ОС и JVM)