**Теория по лабораторной работе 7**

[**Многопоточность. Классы «Thread», интерфейс «Runnable». Модификатор «Sinchronized».** 2](#_Toc117516692)

[**Методы wait(), notify() класса Object, интерфейсы Lock и Condition.** 4](#_Toc117516693)

[**Отличия между synchronized и lock** 4](#_Toc117516694)

[**Классы-синхронизаторы из пакета java.util.concurrent** 5](#_Toc117516695)

[**Модификатор volatile. Атомарные типы данных и операции.** 5](#_Toc117516696)

[**Коллекции из пакета java.util.concurrent.** 6](#_Toc117516697)

[**Интерфейсы Executor, ExecutorService, Callable, Future** 6](#_Toc117516698)

[**Интерфейсы Statement, PreparedStatement, ResultSet, RowSet** 8](#_Toc117516699)

[**Пул потоков.** 8](#_Toc117516700)

[**JDBC. Порядок взаимодействия с базой данных. Класс DriverManager. Интерфейс Connection.** 9](#_Toc117516701)

# **Многопоточность. Классы «Thread», интерфейс «Runnable». Модификатор «Sinchronized».**

**Многопоточность операционной системы** – возможность одновременного выполнения более чем одной программы.

**Число одновременно выполняющих процессов не ограничено количеством процессоров**

Индивидуальные приложения могут выполнять множество задач в одно и тоже время. Каждая задача называется поток - **thread**. Существенная разница между многими процессами и потоками заключается в следующем:

**Процессы** – это экземпляр класса, во время работы которому выделены ресурсы класса. Например, память. Процессы выполняются в отдельном адресном пространстве. То есть один процесс не сможет получить доступ к переменным и структурам данных другого. Вернее не может получить доступ просто так. Чтобы получить доступ к ресурсам других процессов надо использовать межпроцессное взаимодействие. Это могут быть файлы, например.

**Потоки** же могут работать с одной и той же областью памяти, а значит иметь доступ к памяти других потоков. Поток – определенный способ выполнения процесса. Когда один поток изменят ресурс процесса, это изменение сразу же становится видно и другим потокам процесса.

**Создание потока:**

1. Чтобы создать новый поток выполнения, необходимо создать новый объект Thread:
   1. Thread worker = new Thread();
2. После того, как объект-поток будет создан, можно задать его конфигурацию и запустить.
3. Когда поток готов к работе, следует вызывать его метод start.
4. Метод start порождает новый выполняемы поток на основе данных объекта класса Thread, после чего завершается.
5. Метод start также вызывает метод run (из интерфейса Runnable) нового потока, что приводит к активизации последнего
6. Выход из метода run – прекращение работы потока.

Класс Thread реализует интерфейс **«Runnable»**. Это обычный интерфейс. И он называется функциональным интерфейсом, который имеет всего лишь один метод – абстрактный метод.

**Второй способ** создания потоков: Через реализацию интерфейса Runnable

1. Поток служит абстракцией понятия исполнителя – субъекта, способного к выполнению каких-либо полезных действий.
2. План работы, подлежащий выполнению, описывается посредством инструкцией метода run.
3. Чтобы некая цель была достигнута, необходимы исполнитель и план работы:
   1. Интерфейс абстрагирует понятие работы и позволяет назначить её исполнителю – объекту потока.
4. В составе интерфейса объявляем единственный метод

Interface Runnable(

Public void run()

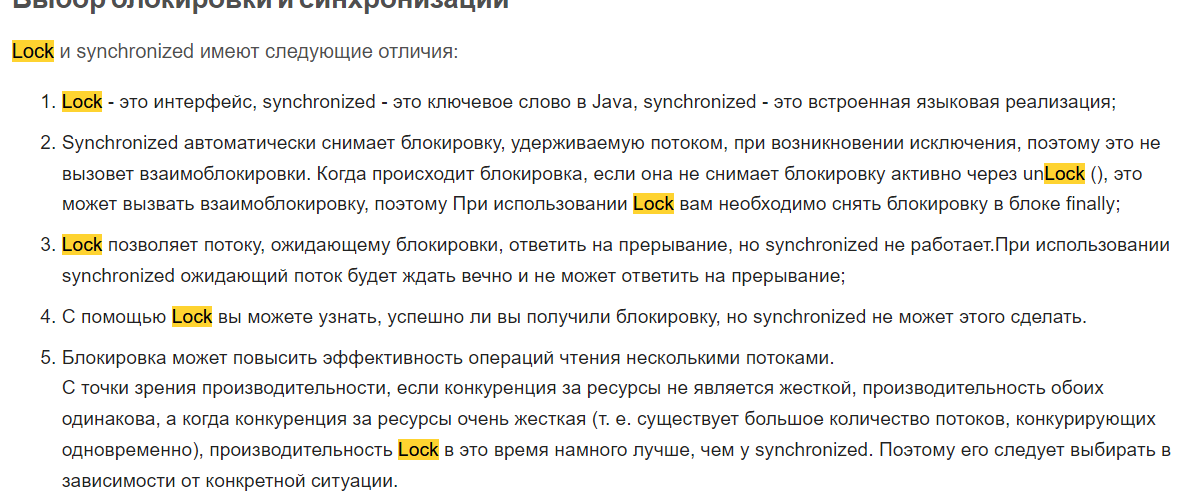
)

**synchronized** перед методом означает, что данный метод может выполнятся только в одном потоке. Есть еще интерфейс **Lock.** У него больше функциональность. Он быстрее работает с большим количеством поток, а вот с маленьким – медленнее. Это если сравнивать с синхронайзд, так у лока скорость работы одинаковое. Так же, почему мы еще используем синхронайзд – у синхронайзда есть автоматическое снятие блокировки, а у лока нет.

# **Методы wait(), notify() класса Object, интерфейсы Lock и Condition.**

Методы wait() и notify() предназначены для управления потоками. Они принадлежат классу Object. Wait() нужен для остановки вызывающего потока. То есть поток, где был вызван этот метод остановится. Так же можно в качестве параметра установить время, на которое остановится поток. И поток будет простаивать до тех пор, пока не будет вновь запущен. Для того, чтобы вновь запустить поток, надо вызвать метод notify() или notifyAll(). Эти методы вызывается в другом потоке. notifyAll() запускает все остановленные потоки. **Lock –** своего рода аналог synchronyzed, только с более широкими возможностями блокировки. Один из плюсов заключается в том, лок мы можем распространить на весь класс. С помощью метода lock() мы блокируем, а с помощью метода unlock() – разблокируем. **Condition.** Не может существовать без lock(). По сути Conditiojn нужен для взаимодейтсвия потоков. Допустим у нас стоит задача: есть два потока. Один отвечает за пополнение баланса, другой – за снятие денег. Соответственно мы не можем снять большую сумму, чем было положено. Для этого в классе мы создаем объект класса Lock, потом создаем объект от Condition: Condition newCondition = lock.newCondition(). В методе, который отвечает за снятие мы пишем – newCondition.await() в цикле определенном, а во втором методе – newCondition.signal()/signalAll(). Эти два метода отличаются тем, что первый – посылает сигнал в один поток, а второй – во все потоки.

# **Отличия между synchronized и lock**



# **Классы-синхронизаторы из пакета java.util.concurrent**

Синхронизатор **semafor** реализует шаблон синхронизации Семафор. В основном они нужны для того, чтобы ограничить одновременный доступ к некоторому ресурсу.Есть два вида конструкторов: Semafor( int permits) и Semafor(int permets, boolean fair). То есть обязательно в конструктор передается количество потоков, которым семафор будет разрешать доступ к ресурсу. Доступ управляется с помощью счетчика, изначальное значение которого мы передаем в конструктор. С каждым потоком значение уменьшается на единицу. И как только значение становится равным нулю, то текущий поток блокируется, пока один поток не выйдет из блока. **CountDownLatch.** Этот синхранизатор 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. Блокировка регулируется счетчиком, значение которого мы устонавливаем. Когда поток прекращает выполнение определенной операции, то значение счетчика уменьшается на один. Как только становится равным нулю, то все потоки разблокируются. **CyclicBarrier** чем то похож на CountDownLatch, но есть главное различие, которое заключается в том, что мы не сможем заново использовать «замок», а вот барьер можем использовать даже после его поломки. **Exchanger<V>** – может понадобится для того, чтобы обменятся данными между двумя потоками в определенной точке работы. Обменник – обощенный класс. Он параметризуется типом объекта передачи.

# **Модификатор volatile. Атомарные типы данных и операции.**

Ключевое слово volatile необходима для того случая, когда одна переменная делится между несколькими потоками и когда один поток пишет в эту переменную, а другой - читает. Тогда с помощью этого слова мы можем настроить взаимодействие потоков. Если мы переменную помечаем словом volatile, то мы говорим, что эта переменная может быть изменена. То есть это слово гарантирует когерентность кешей. То есть если один поток будет изменять значение этой переменной, то эти изменения видят и другие потоки. Для работы с атомарными переменными берется класс java.util.concurent.atomic .Атомарные переменные полезны тем, что во время их изменения никакой другой поток не может вклинится.

# **Коллекции из пакета java.util.concurrent.**

Мы можем сами синхронизовать коллекции класса Collections. Это самый простой способ – использование synchronized. Более продвинутые методы включают в себя алгоритмы, разработанные для одновременного доступа. И такие алгоритмы есть в классах коллекции пакета. Примером такого класса является **«ConcurrentHashMap»**. Это подразделяет всю таблицу на множество небольших разделов, которые имеют свою блокировку. Следовательно, операции записи в эту карту из разных потоков, при условии, что они пишут в разных разделах таблицы, не конкурируют и могут использовать свою собственную блокировку. Также есть еще коллекции: ConcurrentSkipListMap, ConcurrentSkipListSet, CopyOnWriteArrayList и CopyOnWriteArraySet. Класс **CopyOnWriteArrayList** следует использовать вместо ArrayList когда используем большое количество потоков. При использовании обычного ArrayList в многопоточном режиме необходимо либо блокировать целый список во время перебора, либо клонировать его пред перебором. Оба варианта требуют дополнительных ресурсов. А наш класс вместо этого создаёт новую копию списка при выполнении модифицирующей операции и гарантирует, что её итераторы вернут состояние списка на момент создание итератора. Класс **CopyOnWriteArraySet** создан на основе класса CopyOnWriteArrayList. Однако данный класс не поддерживает операцию remove(). Попытка удалить элемент во время итерации приведет к вызову исключения UnsupportedOperationExeption. В общем, если набор данных небольшой и не подвержен изменениям, то лучше использовать CopyOnWriteArraySet.

# **Интерфейсы Executor, ExecutorService, Callable, Future**

**Executor.** Наследником данного интерфейса является **ExecutorService.** Также еще наследниками является **ThreadPoolExecutor.** По итогу у нас всего три класса реализации **ExecutorServixe –** ThreadPoolExecutor, SheduledThreadPoolExecutor и ForkJoinPool. Первая реализация самая часто используемая. 2-я реализация является расширением для запуска задач по расписанию. Executors – фабрика, которая позволяет создавать экземпляры каких-либо классов. Executors – по сути уже готовые потоки, которые мы можем использовать для наших конкретных задач. Первый метод – newCachedThreadPool(). Это уже готовы набор потоков. И чем больше мы запихиваем туда задач, тем больше потоков создается. Однако при завершении задачи, поток не убивается. newFixedThreadPool() – метод для создание фиксированного количества потоков. Еще один интересный метод – newWorkStealingPool(). Это набор потоков, который мы можем не задавать, но он будет соответствовать количеству ядер на компьютере.

**ThreadPoolExecutor.** Int сorePoolSize – это параметр, который говорит, что когда наш es, то у него будут уже запущены именно

**Future.** Класс служит как квитанция. То есть мы получаем квитанцию о том, что наша задача принята на выполнение. И по этой квитанции мы можем получать какую либо информацию. У этого класса есть метод submit, которым запускаем наш поток. Первый метод – **isDone.** Это проверка того, что наша задача завершилась или нет. По сути проверка ситуации. Второй метод – **get().** Этот метод ожидает до тех пор, пока задача не выполнится. Третий метод – cancel(). Это метод пытается остановить выполнение задачи. Но не факт, что она будет прервана. А с помощью четвертого метода is.Cancelled() мы можем получить информацию о том, была ли задача отменена. Есть также метод **invokeAll()**. В него мы можем ппередать список задач. И по сути они запустятся вместе, и метод потом ожидает завершения всех потоков и только потом идет дальше.

Интерфейс Callable позволяет помимо работы в потоке вернуть результат. Сам по себе интерфейс – generic. Мы создаем класс, который имплементим от интерфейса. Переопределяем метод call, где прописываю нужную нам логику и возвращаем результат. В main создаем объект: Callable<Integer> cllable = new MyCallable(). А затем создаем FutureTask futureTask = new FutureTask(callable). А уже futureTask будет иметь множество полезных методов.

# **Интерфейсы Statement, PreparedStatement, ResultSet, RowSet**

**Statement** - этот интерфейс используется для доступа к БД для общих целей. Он крайне полезен, когда мы используем статические SQL – выражения во время работы программы. Этот интерфейс не принимает никаких параметров.

**PreparedStatment –** этот интерфейс может принимать параметры во работы программы. То есть мы его можем использовать для взаимодействия с БД. Например, получить данные по какому либо условию

**CallableStatement –** этот инерфейс становится полезным в случае, когда мы хотим получить доступ к различным процедурам БД. Он также может принимать параметры во время работы.

<https://www.examclouds.com/ru/java/java-core-russian/jdbc-work>

# **Пул потоков.**

**ThreadPool.** Чем больше потокв мы создаём, тем больше мы нагружем компьютер. И java предлагает нам вместо создания, например, 20 потоков, создать 5. И когда эти 5 потоков закончат работу, то они не удалятся, а возьмут новые задачи. Для такой работы мы используем библиотеки ExecutorService и Executors:

ExecutorService executorService = Executors.newFixedThreadPool( count), где count – количество потоков.

Через execute мы создаем объекты. В конце надо указать executorService.shutDown();

Есть еще метод newCashedThreadPool. Он отличается тем, что мы в нем не укзаываем размер, и он наблюдает за тем, чтобы у нас не было очереди потоков. Если приходит объект, то он смотрит, есть ли свободной поток, который простаивает без работы. Если не, то создает новый.

# **JDBC. Порядок взаимодействия с базой данных. Класс DriverManager. Интерфейс Connection.**

DriverManager.*getConnection*(*DB\_URL*, *USER*, *PASS*)

Таким образом у нас происходит соединение с БД. Метод **getConnection()** принимает на вход первым аргументом – url базы данных, потом – имя пользователя, а последним – пароль для БД. И это сохраняется в экземпляр класса **Connection. ПОДКЛЮЧЕНИЕ К БД НЕЛЬЗЯ ОСТАВЛЯТЬ ОТКРЫТЫМ.**