1. **Многопоточность. Класс Thread, интерфейс Runnable. Модификатор synchronized.**

Класс Thread: для того, чтобы создать поток, необходимо создать объект этого класса. При создании собственного потока – необходимо переопределить метод run где вы пропишете то, что хотите от данного потока. Запустить поток после создания – метод start у объекта.

* Thread.currentThread() – получаем текущий поток выполнения.
* getName() – возвращает имя потока
* setName(String name) – устанавливает имя потока
* getPriority() – возвращает приоритет потока
* setPriority(int proirity) – устанавливает приоритет потока. Приоритет является одним из ключевых факторов для выбора системой потока из кучи потоков для выполнения. В этот метод в качестве параметра передается числовое значение приоритета - от 1 до 10. По умолчанию главному потоку выставляется средний приоритет - 5.
* isAlive(): возвращает true, если поток активен
* isInterrupted(): возвращает true, если поток был прерван
* join(): ожидает завершение потока
* run(): определяет точку входа в поток
* sleep(): приостанавливает поток на заданное количество миллисекунд
* start(): запускает поток, вызывая его метод run()

Интерфейс Runnable: этот интерфейс имеет один метод run().

При создании синхронизированного блока кода после оператора synchronized идет объект-заглушка: synchronized(res). Причем в качестве объекта может использоваться только объект какого-нибудь класса, но не примитивного типа.Синхронизация потоков. для работы с общими ресурсами.

Оператор synchronised()

Синхронизированный блок:         synchronized(res){...}

Синхронизированный метод:     synchronized void increment(){...}

Мьютекс — это специальный объект для синхронизации потоков. Он «прикреплен» к каждому объекту в Java. Неважно, пользуешься ли ты стандартными классами или создал собственные классы, скажем, Cat и Dog: у всех объектов всех классов есть мьютекс. Задача мьютекса — обеспечить такой механизм, чтобы доступ к объекту в определенное время был только у одного потока.

Монитор — это дополнительная «надстройка» над мьютексом. Фактически **монитор — это «невидимый» для программиста кусок кода**. По сути, **монитор в Java выражен с помощью слова** synchronized. Весь код, который появился вместо слова synchronized в последнем примере, — это и есть монитор. Благодаря ему создается защитный механизм, который проверяет – захвачен ли мьютекс или нет. И реализует дальнейшую работу.

Семафор — это средство для синхронизации доступа к какому-то ресурсу. **Его особенность заключается в том, что при создании механизма синхронизации он использует счетчик.** Счетчик указывает нам, сколько потоков одновременно могут получать доступ к общему ресурсу. Разница только в том, что мьютекс объекта может захватить одновременно только один поток, а в случае с семафором используется счетчик потоков, и доступ к ресурсу могут получить сразу несколько из них. И это не просто случайное сходство :) На самом деле **мьютекс — это одноместный семафор**. То есть, это семафор, счетчик которого изначально установлен в значении 1.

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

Методы класса Object, предназначенные для многопоточности:

* wait(): освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод notify()
* notify(): продолжает работу потока, у которого ранее был вызван метод wait()
* notifyAll(): возобновляет работу всех потоков, у которых ранее был вызван метод wait()

Все эти методы вызываются только из синхронизированного контекста - синхронизированного блока или метода.

Lock - Вначале поток пытается получить доступ к общему ресурсу. Если он свободен, то на него накладывает блокировку. После завершения работы блокировка с общего ресурса снимается. Если же ресурс не свободен и на него уже наложена блокировка, то поток ожидает, пока эта блокировка не будет снята.

* void lock(): ожидает, пока не будет получена блокировка
* void unlock(): снимает блокировку

По сути мы просто в самом начале создаем объект этого интерфейса, далее в самом начале метода пишем lock.lock, а затем на выходе lock.unlock. Lock работает с мьютексом объекта.

Condition - Применение условий в блокировках позволяет добиться контроля над управлением доступом к потокам. Условие блокировки представлет собой объект интерфейса Condition из пакета java.util.concurrent.locks. Условие Condition, иначе именуемое как очередь условия, предоставляет средство управления для одного потока, чтобы приостановить его выполнение до тех пор, пока он не будет уведомлен другим потоком.

Блокировка Lock заменяет использование synchronized, а Condition — объектные методы монитора.

* await(): поток ожидает, пока не будет выполнено некоторое условие и пока другой поток не вызовет методы signal/signalAll. Во многом аналогичен методу wait класса Object
* signal(): сигнализирует, что поток, у которого ранее был вызван метод await(), может продолжить работу. Применение аналогично использованию методу notify класса Object
* signalAll(): сигнализирует всем потокам, у которых ранее был вызван метод await(), что они могут продолжить работу. Аналогичен методу notifyAll() класса Object

Condition condition = lock.newCondition();

1. **Классы-сихронизаторы из пакета java.util.concurrent.**

* **Semaphore** - чаще всего используются для ограничения количества потоков при работе с аппаратными ресурсами или файловой системой. Доступ к общему ресурсу управляется с помощью счетчика. Если он больше нуля, то доступ разрешается, а значение счетчика уменьшается. Если счетчик равен нулю, то текущий поток блокируется, пока другой поток не освободит ресурс.
* **CountDownLatch** - Позволяет одному или нескольким потокам ожидать до тех пор, пока не завершится определенное количество операций, выполняющих в других потоках. Классический пример с драйвером довольно неплохо описывает логику класса: Потоки, вызывающие драйвер, будут висеть в методе await (с таймаутом или без), пока поток с драйвером не выполнит инициализацию с последующим вызовом метода countDown.
* **CyclicBarrier** - Может использоваться для синхронизации заданного количества потоков в одной точке. Барьер достигается когда N-потоков вызовут метод await(...) и заблокируются. После чего счетчик сбрасывается в исходное значение, а ожидающие потоки освобождаются.
* **Exchanger**<V> - основное предназначение данного класса — это обмен объектами между двумя потоками. При этом, также поддерживаются null значения, что позволяет использовать данный класс для передачи только одного объекта или же просто как синхронизатор двух потоков. Первый поток, который вызывает метод exchange(...) заблокируется до тех пор, пока тот же метод не вызовет второй поток.
* **Phaser** - Улучшенная реализация барьера для синхронизации потоков, которая совмещает в себе функционал CyclicBarrier и CountDownLatch, вбирая в себя самое лучшее из них. Так, количество потоков жестко не задано и может динамически меняться. Класс может повторно переиспользоваться и сообщать о готовности потока без его блокировки.

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

Ключевое слово volatile:

* Она всегда будет атомарно читаться и записываться. Даже если это 64-битные double или long.
* Java-машина не будет помещать ее в кэш. Так что ситуация, когда 10 потоков работают со своими локальными копиями исключена.

Атомарные типы данных – типы данные, которые могут иметь либо начальное значение до изменения, либо конечное. Никакого промежуточного значения быть не может.

Атомарные классы пакета java.util.concurrent.atomic можно разделить на 4 группы :

|  |  |
| --- | --- |
| • AtomicBoolean • AtomicInteger • AtomicLong • AtomicReference | Atomic-классы для boolean, integer, long и ссылок на объекты. Классы этой группы содержат метод **compareAndSet**, принимающий 2 аргумента : предполагаемое текущее и новое значения. Метод устанавливает объекту новое значение, если текущее равно предполагаемому, и возвращает true. Если текущее значение изменилось, то метод вернет false и новое значение не будет установлено. Кроме этого, классы имеют метод **getAndSet**, который безусловно устанавливает новое значение и возвращает старое. Классы AtomicInteger и AtomicLong имеют также методы инкремента/декремента/добавления нового значения. |
| • AtomicIntegerArray • AtomicLongArray • AtomicReferenceArray | Atomic-классы для массивов integer, long и ссылок на объекты. Элементы массивов могут быть изменены атомарно. |
| • AtomicIntegerFieldUpdater • AtomicLongFieldUpdater • AtomicReferenceFieldUpdater | Atomic-классы для обновления полей по их именам с использованием [reflection](http://java-online.ru/java-reflection.xhtml). Смещения полей для CAS операций определяется в конструкторе и кэшируются. Сильного падения производительности из-за *reflection* не наблюдается. |
| • AtomicStampedReference • AtomicMarkableReference | Atomic-классы для реализации некоторых алгоритмов, (точнее сказать, уход от проблем при реализации алгоритмов). Класс AtomicStampedReference получает в качестве параметров ссылку на объект и int значение. Класс AtomicMarkableReference получает в качестве параметров ссылку на объект и битовый флаг (true/false). |

Эффект ключевого слова volatile приблизительно таков, что каждая отдельная операция чтения или записи по этой переменной является атомарной. Примечательно, однако, что операция, которая требует более одного чтения/записи - например, i++, который эквивалентен i = i + 1, который делает одно чтение и один - не является атомарным, поскольку другой поток может напишите в i между чтением и записью.Классы Atomic, такие как AtomicInteger и AtomicReference, предоставляют более широкий спектр операций атомарно, в частности, включая приращение для AtomicInteger.

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

Простые операции типа get (интерфейс List) и put (интерфейс Map) могут выполняться безопасно без дополнительной синхронизации. Но существует несколько распространённых операций, связанных с итератором Iterator<E> и методом add (put-if-absent), которые всё же нуждаются во внешней синхронизации, чтобы избежать конкуренции при обращении к данным.

Рассмотрим метод addValue реализующий принцип добавить-если-отсутствует с коллекцией типа Map. Если элемент отсутствует в коллекции map, то его необходимо добавить. Несмотря на то, что map синхронизирован, у параллельного потока существует возможность вставить значение с повторяющимся ключом между моментами возврата текущего потока из метода containsKey() и вызовом им метода put(). Чтобы гарантировать невозможность «одновременной» вставки в массив из параллельного потока, необходимо заключить пару строк метода в [синхронизированный блок](http://java-online.ru/java-thread.xhtml#synchronized) или синхронизировать весь метод.

Итераторы также подвержены сбоям при работе в многопоточном приложении. Так, если один поток изменяет содержимое коллекции, а второй поток обрабатывает ее итератором Iterator, то при вызове метода Iterator.hasNext() или Iterator.next() будет вызвано исключение ConcurrentModificationException (см. код ниже). Чтобы обезопасить приложение от вызова исключения также, как и в предыдущем примере, необходимо целиком блокировать List на время перебора.

* Класс ConcurrentHashMap

СoncurrentHashMap использует несколько сегментов, и данный класс нужно рассматривать как группу HashMap’ов. Количество сегментов по умолчанию равно 16. Если пара key-value хранится в 10-ом сегменте, то ConcurrentHashMap заблокирует, при необходимости, только 10-й сегмент, и не будет блокировать остальные 15.

* Класс CopyOnWriteArrayList

Класс CopyOnWriteArrayList следует использовать вместо ArrayList в потоконагруженных приложениях, где могут иметь место нечастые операции вставки и удаления в одних потоках и одновременный перебор в других. Это типично для случая, когда коллекция ArrayList используется для хранения списка объектов. Он создаёт новую копию списка при выполнении модифицирующей операции и гарантирует, что её итераторы вернут состояние списка на момент создания итератора и не выкинут ConcurrentModificationException. Нет необходимости клонировать список до перебора или блокировать его во время перебора, т.к. используемая итератором копия списка изменяться не будет. Другими словами, CopyOnWriteArrayList содержит изменяемую ссылку на неизменяемый массив, поэтому до тех пор, пока эта ссылка остаётся фиксированной, вы получаете все преимущества потокобезопасности от неизменности без необходимости блокировок.

* CopyOnWriteArraySet

Создан на основе класса CopyOnWriteArrayList, т.е. использует все его возможности. Он добавлен в JDK 1.5 как и остальные коллекции пакета java.util.concurrent. Лучше всего CopyOnWriteArraySet использовать для read-only коллекций небольших размеров. Если в данных коллекции произойдут изменения, накладные расходы, связанные с копированием, не должны быть ресурсоёмкими. Необходимо помнить, что итераторы класса CopyOnWriteArraySet не поддерживают операцию remove(). Попытка удалить элемент во время итерирации приведет к вызову исключения UnsupportedOperationException. В своей работе итераторы используют «моментальный снимок» массива, который был сделан на момент создания итератора. Таким образом, если набор данных небольшой и не подвержен изменениям, то лучше использовать

Collections.synchronizedList  (List)  
Collections.synchronizedSet   (Set)  
Collections.synchronizedMap (Map)  
Написанные выше коллекции являются потокобезопасными, однако все методы в них выполняются в синхронизированном виде, поэтому данные коллекция являются немасштабируемыми. Такие методы как get, put могут же выполняться безопасно без дополнительной синхронизации.

1. **Интерфейсы Executor, ExecutorService, Callable, Future**

**Executor** – при создании данного объекта у него есть всего 1 метод – execute(thread), данный метод запускает исполнение указанного потокаПри вызове метода execute исполняется поток thread. То есть, метод execute запускает указанный поток на исполнение. Следующий код показывает, как вместо обычного старта потока Thread.start() можно запустить поток с использованием сервиса исполнения.

**ExecutorService** - автоматически предоставляет пул потоков и API для назначения ему задач. может выполнять задачи Runnable и Callable.

* Execute() метод является недействительным и не дает никакой возможности получить результат выполнения задачи или для проверки состояния выполнения задачи: executorService.execute(runnableTask);
* submit () отправляет задачу Callable или Runnable в ExecutorService и возвращает результат типа Future

Future<String> future = executorService.submit(callableTask);

* invokeAny () назначает набор задач ExecutorService , заставляя каждую из них запускаться, и возвращает результат успешного выполнения одной задачи (если было успешное выполнение)

String result = executorService.invokeAny(callableTasks);

* invokeAll () назначает набор задач ExecutorService , заставляя каждую из них запускаться, и возвращает результат выполнения всех задач в виде списка объектов типа Future :

List<Future<String>> futures = executorService.invokeAll(callableTasks)

**Callable** - Интерфейс Callable<V> очень похож на интерфейс Runnable. Объекты, реализующие данные интерфейсы, исполняются другим потоком. Однако, в отличие от Runnable, интерфейс Callable использует Generic'и для определения типа возвращаемого объекта. Runnable содержит метод run(), описывающий действие потока во время выполнения, а Callable – метод call(). Интерфейс Java Callable(java.util.concurrent.Callable) представляет асинхронную задачу, которая может быть выполнена отдельным потоком. Например, можно передать объект Callable в Java ExecutorService, который затем выполнит его асинхронно. Метод call() вызывается для выполнения асинхронной задачи. Имеет всего один метод – call. Если задача выполняется асинхронно, результат обычно передается обратно через Java Future. Это тот случай, когда Callable передается в ExecutorService для одновременного выполнения. Runnable изначально был разработан для длительного параллельного выполнения, например, одновременный запуск сетевого сервера или просмотр каталога на наличие новых файлов. Интерфейс Callable больше предназначен для одноразовых задач, которые возвращают один результат.

**Future** - Методы интерфейса можно использовать для проверки завершения работы потока, ожидания завершения и получения результата. Результат выполнения может быть получен методом get, если поток завершил работу. Прервать выполнения задачи можно методом cancel, is done – проверка завершения потока. Callable задачи возвращают объект java.util.concurrent.Future. Используя объект Java Future, мы можем узнать состояние задачи Callable и получить возвращенный объект. Он предоставляет метод get(), который может ожидать завершения Callable и затем возвращать результат.

1. **Пулы потоков**

Большинство реализаций исполнителя в java.util.concurrent используйте пулы потоков, которые состоят из рабочих потоков. Этот вид потока существует отдельно от Runnable и Callable задачи это выполняется и часто используется, чтобы выполнить многократные задачи.

Используя рабочие потоки минимизирует издержки, должные распараллеливать создание. Объекты потока используют существенное количество памяти, и в крупномасштабном приложении, выделяя и освобождение многих объектов потока создают существенные издержки управления памятью.

* Fixed thread pool – мы говорим максимальное количество потоков и executorservise создает их такое количество и они начинают ждать задачи, которые приходят через метод submit
* Single thread pool – пул потоков с 1 потоком, который разгребает все задачи
* Cached Thread Pool – Динамический пул потоков, который позволяет именть фиксированное количество потоков всегда в ожидании и масштабироваться до большего числа потоков при необходимости
* Scheduled Thread Pool – для периодических задач. Мы хотим чтобы данная задача выполнялась раз в 10 минут.

1. **JDBC**

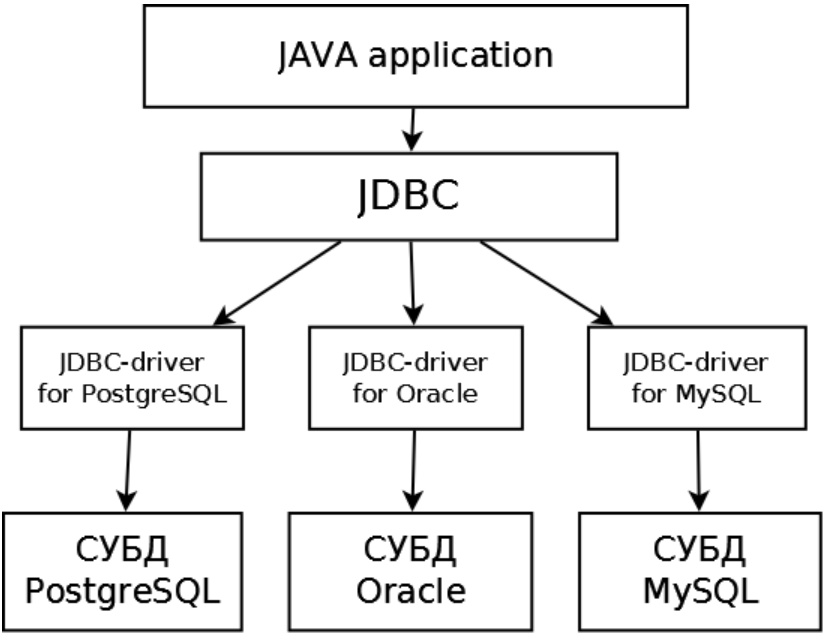
JDBC – средство для работы с хранилищем данных. Java DataBase Connectivity – соединение с базами данных.

JDBC – набор интерфейсов и некоторых классов, которые позволяют работать с базами данных. Главный принцип архитектуры – универсальный способ общения с разными базами данных.

java.sql, javax.sql – пакеты JDBC.

Data Tier – уровень хранения данных.

Persistence Layer – некоторый уровень абстракции для работы с данными из хранилища с уровня Data Tier.



Этапы:

1) Вызов Class.forName() – загрузка класса. Например, Driver.

2) DriverManager.getConnection(url, login, password) – подключение к бд. jdbc:postgresql://localhost:5432/contactdb

Большинство классов в момент своей загрузки выполняют очень важный шаг — они РЕГИСТРИРУЮТСЯ у класса DriverManager. Первый метод как раз и позволяет классу DriverManager пройти по всему списку зарегистрированных у него драйверов и у каждого спросить — “ты умеешь работать с этим URL”. Т.е. первая часть параметра url, о которой мы говорили немного раньше, как раз и позволяет классу DriverManager выбрать драйвер для определенного типа СУБД. Первый шаг сделан — мы выбрали нужный драйвер.

И вот тут приходит очередь второго метода — именно он позволяет создать соединение — возвращает экземпляр класса, который реализует еще один важный интерфейс — java.sql.Connection.

java.sql.Connection — это реальное соединение с конкретным экземпляром СУБД определенного типа. Наше соединение готово.

1. **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.

Методы:

* boolean execute (String SQL) – этот метод возвращает логическое значение true, если объект ResultSet может быть получен. В противном случае он вовращает false. Он используется для выполнения DDL SQL – запросов ил в случаях, когда мы используем динамический SQL.
* int executeUpdate (String SQL) – этот метода возвращает количесство столбцов в таблице, на которое повлиял наш SQL – запрос. Мы используем этот метод для выполнения SQL – запросов, когда хотим получить количество задействованных столбцов, например количество данных по определённому запросу.
* ResultSet executeQuery (String SQL) – этот мтеод возвращает нам экземпляр ResultSet. Мы используем этот метод в случаях, когда мы рассчитываем получить множество объектов в результате выполнения нашего SQL – запроса. Например, при получении списка элементов, которые удовлетворяют опредлённым условиям.

PreparedStatement

Этот интерфейс используется в случае, когда мы планируем использовать SQL – выражения множество раз. Он принимает параметры во время работы программы. Также он нужен в случае, если вы не хотите обрабатывать кучу инфы по типу кавычек, дат и т.п.

String SQL = "Update developers SET salary WHERE specialty = ?";

Все параметры, которые отмечены символом ? называются маркерами параметра. Это означает, что они должны быть переданы через параметры метода.

Каждый параметр ссылается на свой порядковый номер в сигнатуре метода. Т.е. первый маркер находится на первом месте, второй – на втором и т.д. В отличие от массивов, здесь отсчёт идёт с 1. Это связано с особенностями реляционной модели, на которой и основана работа реляционных БД.

Для выполнения SQL – запросов используются методы с такими же названиями (execute(), executeQuery, executeUpdate), которые несколько модифицированы.

Закрытие экземпляра PreparedStatement

Когда мы закрываем наше соединение (Connection) для сохранения результатов в БД мы таким же образом закрываем и экземпляр PreparedStatement.

Для этого мы испольузем метод close().

CallableStatement

Этот интерфейс становится полезным вслучае, когда мы хотим получить доступ к различным процедурам БД. Он также может принимать параметры во время работы программы.

Существует три типа параметров: IN, OUT, INOUT. PreparedStatement использует только IN, а CallableStatement, в свою очередь, использует все три.

* IN - Параметр, значение которого известно в момент, когда создаётся запрос. Мы назначем параметр IN с помощью метода типа setXXX().
* OUT - Параметр, значение которого возвращается SQL – запросом. Мы получаем значения из OUT с помощью методов типа getXXX().
* INOUT - Параметр, который использует входные и выходные значения. Мы назначем параметр с помощью метода типа setXXX(), а получаем значения, с помощью метода типа getXXX().

Когда мы используем параметры типа OUT и INOUT, нам необходимозадействовать дополнительный метод registerOutParameter(). Этот метод устанавливает тип данных JDBC в тип данных процедуры.

После того, как мы вызвали процедуру, мы получаем значение из параметра OUT с помощью соответствующего метода getXXX(). Этот метод преобразует полученное значение из типа данных SQL в тип данных Java.

Когда мы закрываем наше соединение (Connection) для сохранения результатов в БД мы таким же образом закрываем и экземпляр Statement.

Для этого мы испольузем метод close().