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

[Многопоточность в Java (javarush.ru)](https://javarush.ru/groups/posts/1992-mnogopotochnostjh-v-java-sutjh-pljusih-i-chastihe-lovushki-)

[Параллелизм против многопоточности против асинхронного программирования: разъяснение / Хабр (habr.com)](https://habr.com/ru/post/337528/)

Deadlock — ситуация, при которой несколько потоков находятся в состоянии ожидания ресурсов, занятых друг другом, и ни один из них не может продолжать выполнение.

Состояние гонки — ошибка проектирования многопоточной системы или приложения, при которой работа системы или приложения зависит от того, в каком порядке выполняются части кода.

Класс Thread: для того, чтобы создать поток, необходимо создать объект этого класса. Планировщик потоков.

Методы:

* 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()

В методе run() описывается весь код, который будет выполняться потоком.

Создание потоков:

1) Наследование от класса Thread: extends Thread, создание конструктора, @override void run()

Thread t = new JThread("JThread").start();

Ожидание завершения потока: метод join(). В этом случае текущий поток будет ожидать завершения потока, для которого вызван метод join().

t.join();

2) Реализация интерфейса Runnable: этот интерфейс имеет один метод run().

После определения объекта Runnable он передается в конструктор класса Thread(Runnable runnable, String threadName).

class MyThread implements Runnable { public void run(){...}}

Thread myThread = new Thread(new MyThread(),"MyThread");

Поскольку этот интерфейс является функциональным, то также можно записать используя лямбда-выражения:

Runnable r = ()->{...};

Thread myThread = new Thread(r,"MyThread");

Для завершения потока можно создать булевую переменнную, которая меняет свое значение в случае, когда вызывается определенный метод.

[Java | Завершение и прерывание потока (metanit.com)](https://metanit.com/java/tutorial/8.4.php)

При выбрасывании, например, методом Thread.sleep() InterruptedException сам поток будет возвращать isInterrupted() = false, поэтому можно вызвать метод interrupt() и опять проверить на isInterrupted(), чтобы выйти из цикла.

Синхронизация потоков. для работы с общими ресурсами.

Оператор synchronised()

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

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

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.

Каждый объект в Java имеет ассоциированный с ним монитор. Монитор представляет своего рода инструмент для управления доступа к объекту. Когда выполнение кода доходит до оператора synchronized, монитор объекта res блокируется, и на время его блокировки монопольный доступ к блоку кода имеет только один поток, который и произвел блокировку. После окончания работы блока кода, монитор объекта res освобождается и становится доступным для других потоков.

После освобождения монитора его захватывает другой поток, а все остальные потоки продолжают ожидать его освобождения.

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

Методы класса Object, предназначенные для многопоточности:

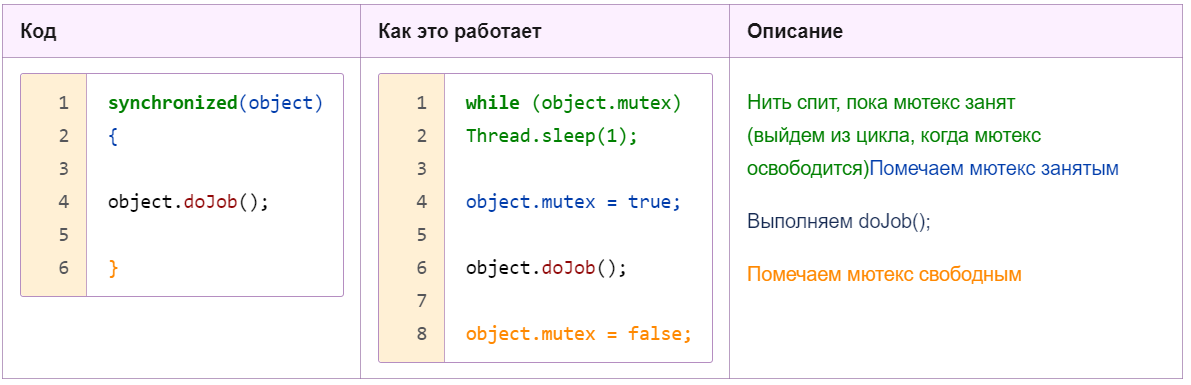
* wait(): освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод notify()
* notify(): продолжает работу потока, у которого ранее был вызван метод wait()
* notifyAll(): возобновляет работу всех потоков, у которых ранее был вызван метод wait()

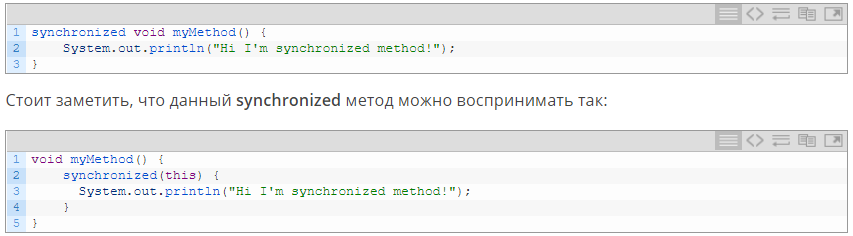
Все эти методы вызываются только из синхронизированного контекста - синхронизированного блока или метода.

[Мьютекс, монитор и семафор в Java (javarush.ru)](https://javarush.ru/groups/posts/2174-v-chem-raznica-mezhdu-mjhjuteksom-monitorom-i-semaforom)

Мютекс – это специальный объект для синхронизации нитей/процессов. Он может принимать два состояния – занят и свободен.

Монитор – это специальный механизм (кусок кода) – надстройка над мютексом, который обеспечивает правильную работу с ним.





Пока поток находится внутри синхронизированного метода объекта, все остальные потоки, которые хотят выполнить этот синхронизированный метод или любой другой синхронизированный метод объекта, должны будут ждать. Это ограничение не распространяется на поток, который уже имеет блокировку и выполняет синхронизированный метод объекта. Такой метод может вызывать другие синхронизированные методы объекта без блокировки. Несинхронизированные методы объекта можно, конечно, вызывать в любой момент любым потоком.

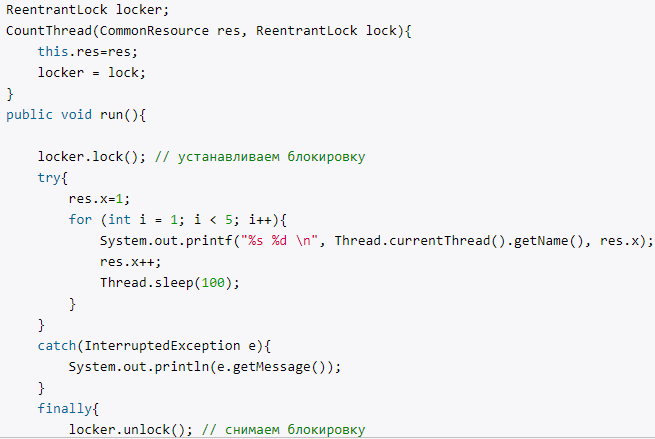
Для управления доступом к общему ресурсу в качестве альтернативы оператору synchronized мы можем использовать блокировки. Функциональность блокировок заключена в пакете java.util.concurrent.locks.

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.

Интерфейс Lock: [Java | Блокировки. ReentrantLock (metanit.com)](https://metanit.com/java/tutorial/8.9.php)

* void lock(): ожидает, пока не будет получена блокировка
* void lockInterruptibly() throws InterruptedException: ожидает, пока не будет получена блокировка, если поток не прерван
* boolean tryLock(): пытается получить блокировку, если блокировка получена, то возвращает true. Если блокировка не получена, то возвращает false. В отличие от метода lock() не ожидает получения блокировки, если она недоступна
* void unlock(): снимает блокировку
* Condition newCondition(): возвращает объект Condition, который связан с текущей блокировкой

Обычно используется класс ReentrantLock (implements Lock):



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.

[Java | Условия в блокировках ReentrantLock (metanit.com)](https://metanit.com/java/tutorial/8.10.php)

* 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

**3) Классы-сихронизаторы из пакета java.util.concurrent.**

|  |  |
| --- | --- |
| [Semaphore](http://java-online.ru/concurrent-synchronizers.xhtml#semaphore) | объект синхронизации, ограничивающий количество потоков, которые могут «войти» в заданный участок кода; |
| [CountDownLatch](http://java-online.ru/concurrent-synchronizers.xhtml#countdownlatch) | объект синхронизации, разрешающий вход в заданный участок кода при выполнении определенных условий; |
| [CyclicBarrier](http://java-online.ru/concurrent-synchronizers.xhtml#cyclicbarrier) | объект синхронизации типа «барьер», блокирующий выполнение определенного кода для заданного количества потоков; |
| [Exchanger](http://java-online.ru/concurrent-synchronizers.xhtml#exchanger) | объект синхронизации, позволяющий провести обмен данными между двумя потоками; |
| [Phaser](http://java-online.ru/concurrent-synchronizers.xhtml#phaser) | объект синхронизации типа «барьер», но в отличие от CyclicBarrier, предоставляет больше гибкости. |

Семафор:

Для управления доступом к ресурсу семафор использует счетчик, представляющий количество разрешений. Если значение счетчика больше нуля, то поток получает доступ к ресурсу, при этом счетчик уменьшается на единицу. После окончания работы с ресурсом поток освобождает семафор, и счетчик увеличивается на единицу. Если же счетчик равен нулю, то поток блокируется и ждет, пока не получит разрешение от семафора.

Semaphore(int permits)

Semaphore(int permits, boolean fair)

Для получения разрешения у семафора надо вызвать метод acquire(), который имеет две формы:

void acquire() throws InterruptedException

void acquire(int permits) throws InterruptedВxception – несколько разрешений

После окончания работы с ресурсом полученное ранее разрешение надо освободить с помощью метода release():

void release()

void release(int permits)

Объект синхронизации CountDownLatch:

Объект синхронизации потоков CountDownLatch представляет собой «защелку с обратным отсчетом» : несколько потоков, выполняя определенный код, блокируются до тех пор, пока не будут выполнены заданные условия. Количество условий определяются счетчиком. Как только счетчик обнулится, т.е. будут выполнены все условия, самоблокировки выполняемых потоков снимаются, и они продолжают выполнение кода.

CountDownLatch(int number);

void await() throws InterruptedException;

boolean await(long wait, TimeUnit unit) throws InterruptedException;

Чтобы уменьшить счетчик объекта CountDownLatch следует вызвать метод countDown:

void countDown();

Объект синхронизации CyclicBarrier:

В исходном коде барьер для группы потоков означает, что каждый поток должен остановиться в определенном месте и ожидать прихода остальных потоков группы. Как только все потоки достигнут барьера, их выполнение продолжится.

CyclicBarrier(int count);

CyclicBarrier(int count, Runnable class);

В первом конструкторе задается количество потоков, которые должны достигнуть барьера, чтобы после этого одновременно продолжить выполнение кода. Во втором конструкторе дополнительно задается реализующий интерфейс Runnable класс, который должен быть запущен после прихода к барьеру всех потоков. Поток запускать самостоятельно НЕ НУЖНО. CyclicBarrier это делает автоматически.

Для указания потоку о достижении барьера нужно вызвать один из перегруженных методов await :

void await() throws InterruptedException

boolean await(long wait, TimeUnit unit) throws InterruptedException;

Циклический барьер CyclicBarrier похож на CountDownLatch. Главное различие между ними связано с тем, что «защелку» нельзя использовать повторно после того, как её счётчик обнулится, а барьер можно использовать (в цикле).

Объект синхронизации Exchanger:

Класс Exchanger (обменник) предназначен для упрощения процесса обмена данными между двумя потоками исполнения. Принцип действия класса Exchanger связан с ожиданием того, что два отдельных потока должны вызвать его метод exchange. Как только это произойдет, Exchanger произведет обмен данными, предоставляемыми обоими потоками.

Обменник является обобщенным классом, он параметризируется типом объекта передачи :

Exchanger<V>();

обменник поддерживает передачу NULL значения, что дает возможность использовать его для передачи объекта в одну сторону или места синхронизации двух потоков.

V exchange(V buffer) throws InterruptedException;

V exchange(V buffer, long wait, TimeUnit unit)

throws InterruptedException

Параметр buffer является ссылкой на обмениваемые данные. Метод возвращает данные из другого потока исполнения. Вторая форма метода позволяет определить время ожидания. Параметры wait и unit описаны выше. Метод exchange, вызванный в одном потоке, не завершится успешно до тех пор, пока он не будет вызван из второго потока исполнения.

Объект синхронизации Phaser

Phaser (фазировщик), как и CyclicBarrier, является реализацией объекта синхронизации типа «Барьер» (CyclicBarrier). В отличии от CyclicBarrier, Phaser предоставляет больше гибкости. Чтобы лучше понять Phaser, можно привести два наглядно демонстрирующих его использование примера.

*В качестве первого примера можно рассмотреть несколько потоков исполнения, реализующих процесс обработки заказов из трех стадий. На первой стадии отдельные потоки исполнения проверяют сведения о клиенте, наличие товара на складе и их стоимость. На второй стадии вычисляется стоимость заказа и стоимость доставки. На заключительной стадии подтверждается оплата и определяется ориентировочное время доставки. Во втором примере несколько потоков реализуют перевозку пассажиров городским транспортом. Пассажиры ожидают транспорт на разных остановках. Транспорт, останавливаясь на остановках, одних пассажиров «сажает», других «высаживает».*

В этих примерах общим является то, что один объект синхронизации Phaser, исполняющий роль заказа и транспорта, играет главную роль, а другие потоки вступают в работу при определенном состоянии Phaser. Таким образом, класс Phaser позволяет определить объект синхронизации, ожидающий завершения определенной фазы. После этого он переходит к следующей фазе и снова ожидает ее завершения.

Особенности Phaser:

* Phaser может иметь несколько фаз (барьеров). Если количество фаз равно 1, то плавно переходим к CyclicBarrier (осталось только все исполнительные потоки остановить у барьера).
* Каждая фаза (цикл синхронизации) имеет свой номер.
* Количество участников-потоков для каждой фазы жестко не задано и может меняться. Исполнительный поток может регистрироваться в качестве участника и отменять свое участие;
* Исполнительный поток не обязан ожидать, пока все остальные участники соберутся у барьера. Достаточно только сообщить о своем прибытии.

Phaser();

Phaser(int parties);

Phaser(Phaser parent);

Phaser(Phaser parent, int parties);

Параметр parties определяет количество участников, которые должны пройти все фазы. Первый конструктор создает объект Phaser без каких-либо участников. Второй конструктор регистрирует передаваемое в конструктор количество участников. Третий и четвертый конструкторы дополнительно устанавливают родительский объект Phaser.

При создании экземпляр класса Phaser находится в нулевой фазе. В очередном состоянии (фазе) синхронизатор находится в ожидании до тех пор, пока все зарегистрированные потоки не завершат данную фазу. Потоки извещают об этом, вызывая один из методов arrive() или arriveAndAwaitAdvance().

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| int register() | Метод регистририрует участника и возвращает номер текущей фазы. |
| int arrive() | Метод указывает на завершения выполнения текущей фазы и возвращает номер фазы. Если же работа Phaser закончена, то метод вернет отрицательное число. При вызове метода arrive поток не приостанавливается, а продолжает выполняться. |
| int arriveAndAwaitAdvance() | Метод вызывается потоком/участником, чтобы указать, что он завершил текущую фазу. Это аналог метода CyclicBarrier.await(), сообщающего о прибытии к барьеру. |
| int arriveAndDeregister() | Метод arriveAndDeregister сообщает о завершении всех фаз участником и снимается с регистрации. Данный метод возвращает номер текущей фазы или отрицательное число, если Phaser завершил свою работу |
| int getPhase() | Получение номера текущей фазы. |

**4) Модификатор volatile. Атомарные типы данных и операции.**

[Ключевое слово volatile и метод yield() (javarush.ru)](https://javarush.ru/groups/posts/1998-upravlenie-potokami-metodih-volatile-i-yield)

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

Атомарные типы данных: int

Неатомарные типы данных: long, double (т.к java кроссплатформенна: в некоторых системах эти примитивы могут иметь размер 32 бит, а где-то 64). Поэтому такие переменные читаются-записываются в 2 стадии: сначала первые 32 бита, затем последние. В промежутке можно суметь считать только “половину”.

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.

Метод yield() класса Thread:

Мы последовательно создаем и запускаем три потока — Thread-0, Thread-1 и Thread-2.

Thread-0 запускается первым и сразу уступает место другим. После него запускается Thread-1, и тоже уступает. После — запускается Thread-2, который тоже уступает.

Больше потоков у нас нет, и после того, как Thread-2 последним уступил свое место, планировщик потоков смотрит: «Так, новых потоков больше нет, кто у нас там в очереди? Кто уступал свое место последним, перед Thread-2? Кажется, это был Thread-1? Окей, значит пусть он и выполняется».

Thread-1 выполняет свою работу до конца, после чего планировщик потоков продолжает координацию: «Окей, Thread-1 выполнился. Есть у нас кто-то еще в очереди?». В очереди есть Thread-0: он уступал свое место сразу до Thread-1. Теперь дело дошло до него, и он выполняется до конца.

После чего планировщик заканчивает координацию потоков: «Ладно, Thread-2, ты уступил место другим потокам, они все уже отработали. Ты уступал место последним, так что теперь твоя очередь». После этого отрабатывает до конца поток Thread-2.

Некоторые правила happens before:

1. Освобождение мьютекса happens before происходит раньше захвата этого же монитора другим потоком.
2. Метод Thread.start() happens before Thread.run().
3. Завершение метода run() happens before выход из метода join().
4. Запись в volatile переменную happens-before чтение из той же переменной.

[Атомарные классы пакета util.concurrent (java-online.ru)](http://java-online.ru/concurrent-atomic.xhtml)

Операция называется атомарной, если её можно безопасно выполнять при параллельных вычислениях в нескольких потоках, не используя при этом ни блокировок, ни синхронизацию synchronized.

Атомарные классы пакета util.concurrent, операции:

private volatile long value;

public final long get() {

return value;

}

public final long getAndAdd(long delta) {

while (true) {

long current = get();

long next = current + delta;

if (compareAndSet(current, next))

return current;

}

}

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). |

**5) Коллекции из пакета java.util.concurrent.**

Структура пакета java.util.concurrent

|  |  |
| --- | --- |
| [collections](http://java-online.ru/concurrent.xhtml#collections) | Набор более эффективно работающих в многопоточной среде коллекций нежели стандартные универсальные коллекции из *java.util* пакета |
| [synchronizers](http://java-online.ru/concurrent.xhtml#synchronizers) | Объекты синхронизации, позволяющие разработчику управлять и/или ограничивать работу нескольких потоков. |
| [atomic](http://java-online.ru/concurrent.xhtml#atomics) | Набор атомарных классов, позволяющих использовать принцип действия механизма *оптимистической блокировки* для выполнения атомарных операций. |
| [Queues](http://java-online.ru/concurrent.xhtml#queues) | Объекты создания блокирующих и неблокирующих очередей с поддержкой многопоточности. |
| [Locks](http://java-online.ru/concurrent.xhtml#locks) | Механизмы синхронизации потоков, альтернативы базовым *synchronized, wait, notify, notifyAll* |
| [Executors](http://java-online.ru/concurrent.xhtml#locks) | Механизмы создания пулов потоков и планирования работы асинхронных задач |

[Многопоточный пакет util.concurrent (java-online.ru)](http://java-online.ru/concurrent.xhtml)

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.  
Один из подходов к улучшению масштабируемости коллекции при сохранении потокобезопасности состоит в том, чтобы обходиться без общей блокировки всей таблицы, а использовать блокировки для каждого hash backet (или, в более общем случае, пула блокировок, где каждая блокировка защищает несколько бакетов). Это позволяет нескольким потокам обращаться к различным частям коллекции одновременно, без соперничества за единственную на всю коллекцию блокировку. Данный подход улучшает масштабируемость операций вставки, извлечения и удаления.

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.

public interface ConcurrentMap<K,V> extends Map<K,V>

{

// добавить, если нет объекта value по ключу K

V putIfAbsent(K key, V value);

// удалить, если имеется объект value с ключом K

boolean remove(K key, V value);

// заменить oldValue новым newValue объекта с ключом K

boolean replace(K key, V oldValue, V newValue);

// заменить новым значением newValue объект с ключом K

V replace(K key, V newValue);

}

[ConcurrentHashMap (Java Platform SE 8 ) (oracle.com)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html)

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.

При использовании обычной ArrayList в многопоточном приложении необходимо либо блокировать целый список во время перебора, либо клонировать его перед перебором; оба варианта требуют дополнительных ресурсов. CopyOnWriteArrayList вместо этого создаёт новую копию списка при выполнении модифицирующей операции и гарантирует, что её итераторы вернут состояние списка на момент создания итератора и не выкинут ConcurrentModificationException. Это так называемый алгоритм CopyOnWrite. 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.

[CopyOnWriteArrayList (Java Platform SE 8 ) (oracle.com)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CopyOnWriteArrayList.html)

CopyOnWriteArraySet

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 CopyOnWriteArraySet.

[CopyOnWriteArraySet (Java Platform SE 8 ) (oracle.com)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CopyOnWriteArraySet.html)

... + очереди.

**6) Интерфейсы Executor, ExecutorService, Callable, Future**

[Concurrent интерфейсы Callable, Future (java-online.ru)](http://java-online.ru/concurrent-callable.xhtml)

Callable<V>:

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().

The Callable interface is similar to Runnable, in that both are designed for classes whose instances are potentially executed by another thread. A Runnable, however, does not return a result and cannot throw a checked exception.

Метод: V call()

Future<V>

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. Дополнительные методы позволяют определить завершение задачи : нормальное или прерванное. Если задача завершена, то прервать ее уже невозможно.

|  |  |
| --- | --- |
| cancel (boolean mayInterruptIfRunning) | попытка завершения задачи |
| V get() | ожидание (при необходимости) завершения задачи, после чего можно будет получить результат |
| V get(long timeout, TimeUnit unit) | ожидание (при необходимости) завершения задачи в течение определенного времени, после чего можно будет получить результат |
| isCancelled() | вернет true, если выполнение задачи будет прервано прежде завершения |
| isDone() | вернет true, если задача завершена |

FutureTask<V>

Класс-оболочка FutureTask базируется на конкретной реализации интерфейса Future. Чтобы создать реализацию данного класса необходим объект Callable; после этого можно использовать Java Thread Pool Executor для асинхронной обработки. Таким образом, FutureTask представляет удобный механизм для превращения Callable одновременно в Future и Runnable, реализуя оба интерфейса. Объект класса FutureTask может быть передан на выполнение классу, реализующему интерфейс Executor, либо запущен в отдельном потоке, как класс, реализующий интерфейс Runnable.

|  |  |
| --- | --- |
| V get() | получение результата выполнения потока; вызов метода блокирует дальнейшее выполнения до окончания вычислений |
| V get(long timeout, TimeUnit unit) | получение результата до окончания вычислений или до истечения указанного интервала времени; если в течение указанного времени вычисления не завершились, то вызывается исключение TimeoutException |
| boolean cancel(boolean mayInterrupt) | отмена выполнения задачи; если задача уже стартована и параметр mayInterrupt равен true, то она прерывается, в противном случае, если вычисления еще не начаты, то они и не начнутся. При успешной отмене выполнения задачи метод возвращает значение true |
| boolean isCancelled() | метод возвращает true, если задача была отменена до ее нормального завершения |
| boolean isDone() | метод возвращает true, если выполнение задачи завершено, прервано или если в процессе ее выполнения возникло исключение |

Интерфейс Executor: [Описание и пример ExecutorService (java-online.ru)](http://java-online.ru/concurrent-executor.xhtml)

void execute(Runnable thread);

При 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.

// Вместо следующего кода

new Thread(new RunnableTask()).start();

// можно использовать

ExecutorService executor;

. . .

executor.execute(new CallableSample1());

Future<String> f1 = executor.submit(new CallableSample2());

Методы интерфейса ExecutorService:

|  |  |
| --- | --- |
| boolean **awaitTermination**(long timeout, TimeUnit unit) | Блокировка до тех пор, пока все задачи не завершат выполнение после запроса на завершение работы или пока не наступит тайм-аут или не будет прерван текущий поток, в зависимости от того, что произойдет раньше |
| List<Future<T>> **invokeAll** (Collection<? extends Callable<T>> tasks) | Выполнение задач с возвращением списка задач с их статусом и результатами завершения |
| List<Future<T>> **invokeAll** (Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit) | Выполнение задач с возвращением списка задач с их статусом и результатами завершения в течение заданного времени |
| T invokeAny(Collection<? extends Callable<T>> tasks) | Выполнение задач с возвращением результата успешно выполненной задачи (т. е. без создания исключения), если таковые имеются |
| T invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit) | Выполнение задач в течение заданного времени с возвращением результата успешно выполненной задачи (т. е. без создания исключения), если таковые имеются |
| boolean isShutdown() | Возвращает true, если исполнитель сервиса остановлен (shutdown) |
| boolean isTerminated() | Возвращает true, если все задачи исполнителя сервиса завершены по команде остановки (shutdown) |
| void shutdown() | Упорядоченное завершение работы, при котором ранее отправленные задачи выполняются, а новые задачи не принимаются |
| List<Runnable> shutdownNow() | Остановка всех активно выполняемых задач, остановка обработки ожидающих задач, возвращение списка задач, ожидающих выполнения |
| Future<T> submit(Callable<T> task) | Завершение выполнения задачи, возвращающей результат в виде объекта Future |
| Future<?> submit(Runnable task) | Завершение выполнения задачи, возвращающей объект Future, представляющий данную задачу |
| Future<T> submit(Runnable task, T result) | Завершение выполнения задачи, возвращающей объект Future, представляющий данную задачу |

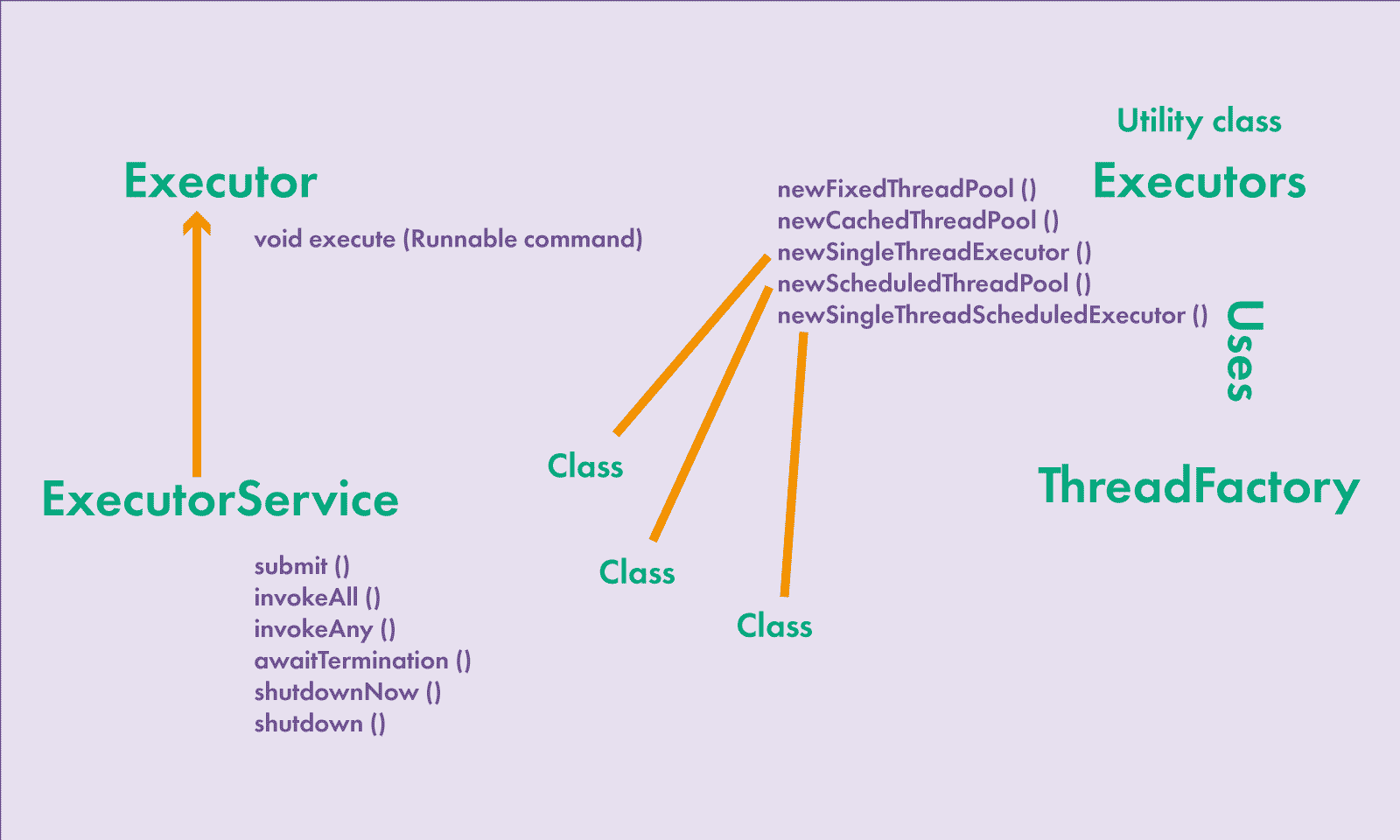
**7) Пулы потоков**

[Многопоточность в Java. Лекция 4: пулы потоков (dataart.team)](https://dataart.team/ru/articles/mnogopotochnost-v-java-lekfiya-4-puly-potokov/)

Пул потоков — своего рода контейнер, в котором содержатся потоки, которые могут выполнять задачи, и после выполнения одной самостоятельно переходить к следующей.

Вторая причина создания пулов потоков — возможность разделить объект, выполняющий код, и непосредственно код задачи, которую необходимо выполнить. Использование пула потоков обеспечивает лучший контроль создания потоков и экономит ресурсы создания потоков.

За созданием и управлением пулом потоков отвечают несколько классов и интерфейсов, которые называются Executor Framework in Java.



Основные интерфейсы: Executor, ExecutorService, фабрика Executors.

Класс Executors – утилитный класс, как Collections:\

Пул потоков использует класс, который реализует интерфейс ThreadFactory для создания потоков, чтобы один поток мог выполнять несколько Runnable- или Callable-объектов. Поток, который выполняет несколько объектов Runnable, называется Worker.

Цепочка выполнения такая: ThreadPoolExecutor -> Thread -> Worker -> YourRunnable.

По умолчанию используется класс Executors$DefaultThreadFactory. ThreadFactory — интерфейс с один методом Thread newThread(Runnable r).

Пул потоков управляет группой рабочих потоков, а также включает очередь для размещения задач, ожидающих выполнения. Пул потоков позволяет избежать частого создания и уничтожения потоков, снизить потребление ресурсов и повысить скорость отклика системы. java.util.concurrent.Executors предоставляет несколько реализаций интерфейса java.util.concurrent.Executor для создания пулов потоков, которые в основном включают четыре роли:

* Пул потоков: исполнитель
* Рабочий поток: рабочий поток, метод run () объекта Worker выполняет метод run () объекта Job.
* Задание задачи: запускаемое и вызываемое
* Очередь блокировки: BlockingQueue

FixedThreadPool

Используется для создания ThreadPool, который использует фиксированное количество потоков, corePoolSize = maxPoolSize = n (фиксированное значение), а очередь блокировки - LinkedBlockingQueue.

SingleThreadExecutor

Используется для создания однопоточного пула потоков, corePoolSize = maxPoolSize = 1, очередь блокировки - LinkedBlockingQueue.

CachedThreadPool

Используется для создания пула кэшируемых потоков, corePoolSize = 0, maxPoolSize = Integer.MAX\_VALUE, очередь блокировки - SynchronousQueue (блокирующая очередь без емкости, каждая операция вставки должна ждать соответствующей операции удаления другого потока, и наоборот), Is to создавать новые потоки в соответствии с требованиями, когда спрос высокий, создается больше, когда спрос низкий, JVM будет медленно освобождать дополнительные потоки.

ScheduledThreadPoolExecutor

Используемый для создания пула потоков неограниченного размера, этот пул потоков поддерживает синхронизацию и периодическое выполнение задач.

[Что такое пул потоков? Как создать пул потоков Java? - Русские Блоги (russianblogs.com)](https://russianblogs.com/article/70101423318/)

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

[JDBC средства для работы с хранилищем данных (javarush.ru)](https://javarush.ru/groups/posts/2172-jdbc-ili-s-chego-vsje-nachinaetsja)

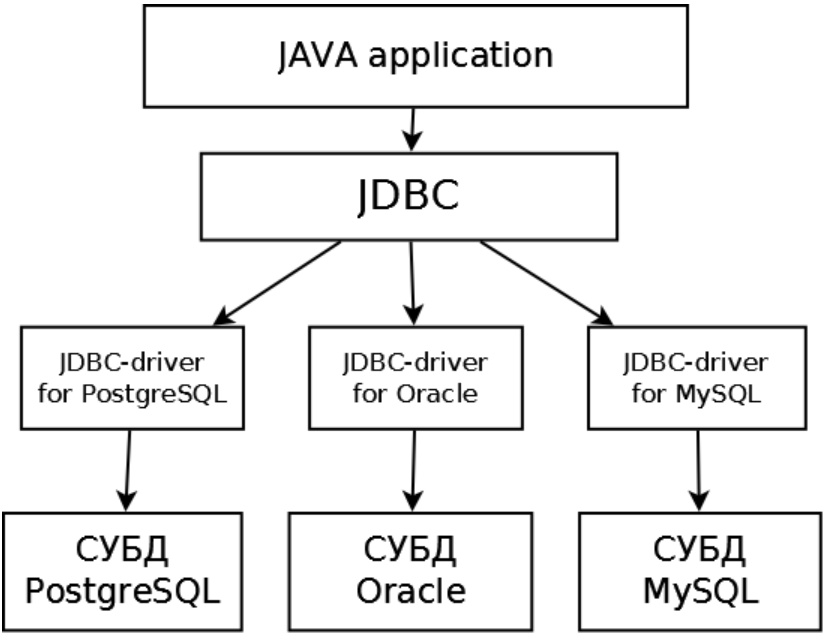
JDBC – средство для работы с хранилищем данных. Java DataBase Connectivity – соединение с базами данных.

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. Среди методов мы можете найти этот: registerDriver(Driver driver). Причем метод статический и создавать экземпляр DriverManager не надо. Таким образом драйвер под конкретный тип СУБД регистрируется у DriverManager. У этого класса (можно глянуть в исходники) создается список драйверов, каждый из которых реализует интерфейс java.sql.Driver.

1. boolean acceptsURL(String url)
2. Connection connect(String url, Properties info)

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”. Отметим, что драйвер под конкретный тип СУБД работает с уникальным набором — MySQL принимает строку “jdbc:mysql:”, PostgreSQL — “jdbc:postgresql:” и т.д. 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. Второй метод использует вторую часть url с адресом, портом и именем базы, а также используется логин и пароль. Снова обращаю ваше внимание на тот факт, что реальный класс будет какой-то специальный, под конкретный тип СУБД, но он обязательно должен реализовать интерфейс java.sql.Connection.

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

Statement stmt = con.createStatement(); - создание запроса

ResultSet rs = stmt.executeQuery("SELECT \* FROM JC\_CONTACT"); - запрос в таблицу

[Базы данных на Java — первые шаги (java-course.ru)](https://java-course.ru/begin/database01/)

SQL-запросы можно условно разделить на две группы:

* Получение данных — к ним относится оператор SELECT (executeQuery) – возвращает ResultSet
* Изменение данных — к ним относятся операторы INSERT, UPDATE и DELETE (executeUpdate) – целое число, кол-во строк.

**9) Интерфейсы Statement, PreparedStatement, ResultSet, RowSet**

[Руководство по JDBC. Утверждения (Statements). – PROSELYTE](https://proselyte.net/tutorials/jdbc/statements/)

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().

[Руководство по JDBC. Result Set. – PROSELYTE](https://proselyte.net/tutorials/jdbc/result-set/)

Данные, полученные в результате SQL – запроса возвращаются в виде множетсва результатов, которые хранятся в сущности под названием Result Set.

Интерфейс java.sql.ResultSet представляет собой множетсво результатов, запроса в БД.

Все методы интерфейса java.sql.ResultSet мы можем разделить на три большие группы:

* Методы получения данных. Эти методы используются для просмотра данных конкетной записи, на которую ссылается указатель.
* Методы изменения данных. Методы этой группы используются для изменения данных текущей записи. Эти изменения передаются в используемую БД.
* Методы навигации. Эти методы используются для перемещения указателя.

Для определения этих свойств используются следующие методы:

* createStatement (int RSType, int RSConcurrency);
* prepareStatement (String SQL, int RSType, int RSConcurrency);
* prepareCall (String SQL, int RSType, int RSConcurrency);

... хватит, я устал.

[JDBC ResultSet и RowSet | EasyJava](https://easyjava.ru/java/jdbc-resultset-i-rowset/)

RowSet расширяет ResultSet и делает его совместимым с концепцией JavaBean (то есть с конструктором по умолчанию, сериализуемым и т.д.). Поскольку интерфейс RowSet расширяет интерфейс ResultSet, весь вышеперечисленный функционал, разумеется, остаётся доступным и в RowSet. Главными отличиями RowSet от ResultSet является тот факт, что RowSet есть JavaBean, со свойствами и нотификациями. Кроме того, RowSet можно строить напрямую из соединения с базой, пропуская отдельно создание запроса.

CachedRowSet, оправдывая своё название, сохраняет работоспособность и тогда, когда соединение с базой уже закрыто, сразу кэшируя в память все данные, которые вернул запрос.

JoinRowSet. Делает слияние таблиц в памяти. Конечно, с точки зрения эффективности выгоднее делать join непосредственно на стороне базы, но не всякая база умеет join (вспоминаем FoxPro опять, ага). Чтобы слить таблицы, вначале необходимо получить две таблицы для слияния и потом добавить их в JoinRowSet, указав по какому полю сливать их.

FilteredRowSet. RowSet который умеет сам себя фильтровать. На первый вгляд кажется бесполезной вещью (даже FoxPro умеет в условия), но на самом деле очень удобен, так как фильтры в коде могут быть гораздо гибче, чем условия выборки в базе. Да и фильтровать какой-нибудь постоянно висящий в памяти словарь становится выгоднее, чем постоянно его перезапрашивать.

WebRowSet умеет сам сохранять себя в XML и создавать себя из XML же. В том году, когда его изобретали, это было удивительным достижением конечно, а сейчас выглядит слегка архаично.

**10) Шаблоны проектирования.**

Наследование – статическое отношение.

Делегирование – динамическое отношение.

Порождающие шаблоны:

\* Factory Method – фабричный метод

\* Abstract Factory – абстрактная фабрика

\* Builder - строитель

\* Prototype - прототип

\* Singleton - одиночка

\* Object Poll – пул объектов

Структурные шаблоны:

\* Adapter

\* Bridge

\* Composite - компоновщик

\* Decorator

\* Facade - фасад

\* Flyweight - приспособленец

\* Proxy – заместитель

Поведенческие шаблоны:

\* Chain of Responsibility – цепочка обязанностей

\* Command

\* Interpreter

\* Iterator

\* Mediator - посредник

\* Memento - хранитель

\* Observer - наблюдатель

\* State - состояние

\* Strategy

\* Template Method – шаблонный метод

\* Visitor