**Statement**

* Statement - используется для выполнения SQL-запросов. Он крайне полезен, когда мы используем статические SQL – выражения во время работы программы. Этот интерфейс не принимает никаких параметров. Для создания объекта данного интерфейса мы пишем:   
  Connection conn = DriverManager.getConnection(url, username, password);

Statement stmt = conn.createStatement();

Имеет три метода

* + Execute(String SQL) - Этот метод возвращает логическое значение true, если объект ResultSet может быть получен. В противном случае он вовращает false.
  + 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.
  + ExecuteQuery(String SQL) - 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.
* PrepareStatement - Этот интерфейс используется в случае, когда мы планируем использовать SQL – выражения множество раз. Он принимает параметры во время работы программы. Т.е. предварительно компилирует запросы, которые могут содержать входные параметры обозначенные символом '?'.
* CallableStatement - используется для выполнения процедур, непосредоственно в самой БД(т.е. созданных в БД!). 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 необходимо задействовать 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.

* ResultSet - это объект Java, возвращаемый при запросе (query) данных. Используйте ResultSet.next() для перемещения курсора к следующим записям (перемещается строка ). В определенной записи вы используете методы ResultSet.getXxx () для получения значений в столбцах. Виды ResultSet
  + TYPE\_FORWARD\_ONLY - ResultSet только позволяет просматривать сверху вниз и слева направо. Это по умолчанию ResultSet.
  + TYPE\_SCROLL\_INSENSITIVE - ResultSet позволяет прокручивать вперед, назад, налево, направо. но не чувствителен к изменениям данных ниже DB. То есть при просмотре записи и повторного просмотра записи, он не будет брать новейшие данные записи, которые могли быть изменены кем-то.
  + TYPE\_SCROLL\_SENSITIVE - ResultSet позволяет прокручивать вперед, назад, налево, направо, и чувствителен к изменениям данных.
  + CONCUR\_READ\_ONLY - При просмотре данных с ResultSet этого вида вы можете только читать данные.
  + CONCUR\_UPDATABLE - При просмотре данных с ResultSet этого вида вы можете только менять данные где стоит курсор, например обновить значения определенного столбца.

**JDBC**

JDBC - стандартный API для взаимодействия с реляционными базами данных. JDBC имеет набор классов и интерфейсов, которые могут использоваться для Java-приложения и разговаривать с базой данных.

Основные классы и интерфейсы

* DriverManager: - Это класс, использующийся для управления списком Driver (database drivers).
* Driver - Это интерфейс, использующийся для соединения коммуникации с базой данных, управления коммуникации с базой данных. Когда загружается Driver, программисту не нужно конкретно вызывать его.
* Connection - Интерфейс со всеми методами связи с базой данных. Он описывает коммуникационный контекст. Вся связь с базой данных осуществляется только через объект соединения (connection).
* Statement -Это интерфейс, включающий команду SQL отправленный в базу данных для анализа, обобщения, планирования и выполнения.
* ResultSet - представляет набор записей, извлеченных из-за выполнения запроса.

Подключение происходит так: мы скачиваем нужный нам драйвер, вызвав class.forName() мы загружаем наш драйвер и он регистрируется у DriverManager, далее с помощью DriverManager.getConnection мы создаем соединение с нашей БД.

**Шаблоны проектирования**

* Decorator - позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки». Декоратор имеет альтернативное название — обёртка. Оно более точно описывает суть паттерна: вы помещаете целевой объект в другой объект-обёртку, который запускает базовое поведение объекта, а затем добавляет к результату что-то своё. Оба объекта имеют общий интерфейс, поэтому для пользователя нет никакой разницы, с каким объектом работать — чистым или обёрнутым. Вы можете использовать несколько разных обёрток одновременно — результат будет иметь объединённое поведение всех обёрток сразу. - Блины с добавками в Теремке
* Iterator - даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления. Идея паттерна Итератор состоит в том, чтобы вынести поведение обхода коллекции из самой коллекции в отдельный класс. Объект-итератор будет отслеживать состояние обхода, текущую позицию в коллекции и сколько элементов ещё осталось обойти. Одну и ту же коллекцию смогут одновременно обходить различные итераторы, а сама коллекция не будет даже знать об этом. К тому же, если вам понадобится добавить новый способ обхода, вы сможете создать отдельный класс итератора, не изменяя существующий код коллекции.
* Factory method - определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов. Паттерн Фабричный метод предлагает создавать объекты не напрямую, используя оператор new, а через вызов особого фабричного метода. Не пугайтесь, объекты всё равно будут создаваться при помощи new, но делать это будет фабричный метод. На первый взгляд, это может показаться бессмысленным: мы просто переместили вызов конструктора из одного конца программы в другой. Но теперь вы сможете переопределить фабричный метод в подклассе, чтобы изменить тип создаваемого продукта. Чтобы эта система заработала, все возвращаемые объекты должны иметь общий интерфейс. Подклассы смогут производить объекты различных классов, следующих одному и тому же интерфейсу.
* Command - превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций. В реальности это выглядит так: один из объектов интерфейса напрямую вызывает метод одного из объектов бизнес-логики, передавая в него какие-то параметры. Паттерн Команда предлагает больше не отправлять такие вызовы напрямую. Вместо этого каждый вызов, отличающийся от других, следует завернуть в собственный класс с единственным методом, который и будет осуществлять вызов. Такие объекты называют командами. К объекту интерфейса можно будет привязать объект команды, который знает, кому и в каком виде следует отправлять запросы. Когда объект интерфейса будет готов передать запрос, он вызовет метод команды, а та — позаботится обо всём остальном.
* Flyweight - позволяет вместить бóльшее количество объектов в отведённую оперативную память. Легковес экономит память, разделяя общее состояние объектов между собой, вместо хранения одинаковых данных в каждом объекте. Паттерн Легковес предлагает не хранить в классе внешнее состояние, а передавать его в те или иные методы через параметры. Таким образом, одни и те же объекты можно будет повторно использовать в различных контекстах. Но главное — понадобится гораздо меньше объектов, ведь теперь они будут отличаться только внутренним состоянием, а оно имеет не так много вариаций.
* Singleton - гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа. Все реализации одиночки сводятся к тому, чтобы скрыть конструктор по умолчанию и создать публичный статический метод, который и будет контролировать жизненный цикл объекта-одиночки. Если у вас есть доступ к классу одиночки, значит, будет доступ и к этому статическому методу. Из какой точки кода вы бы его ни вызвали, он всегда будет отдавать один и тот же объект.
* Strategy - определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы. Вместо того, чтобы изначальный класс сам выполнял тот или иной алгоритм, он будет играть роль контекста, ссылаясь на одну из стратегий и делегируя ей выполнение работы. Чтобы сменить алгоритм, вам будет достаточно подставить в контекст другой объект-стратегию. Важно, чтобы все стратегии имели общий интерфейс. Используя этот интерфейс, контекст будет независимым от конкретных классов стратегий. С другой стороны, вы сможете изменять и добавлять новые виды алгоритмов, не трогая код контекста.
* Adapter позволяет объектам с несовместимыми интерфейсами работать вместе. Вы можете создать адаптер. Это объект-переводчик, который трансформирует интерфейс или данные одного объекта в такой вид, чтобы он стал понятен другому объекту. При этом адаптер оборачивает один из объектов, так что другой объект даже не знает о наличии первого. Например, вы можете обернуть объект, работающий в метрах, адаптером, который бы конвертировал данные в футы.
* Facade - предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку. Фасад — это простой интерфейс для работы со сложной подсистемой, содержащей множество классов. Фасад может иметь урезанный интерфейс, не имеющий 100% функциональности, которой можно достичь, используя сложную подсистему напрямую. Но он предоставляет именно те фичи, которые нужны клиенту, и скрывает все остальные. Фасад полезен, если вы используете какую-то сложную библиотеку со множеством подвижных частей, но вам нужна только часть её возможностей.
* Proxy - позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то до или после передачи вызова оригиналу. Паттерн Заместитель предлагает создать новый класс-дублёр, имеющий тот же интерфейс, что и оригинальный служебный объект. При получении запроса от клиента объект-заместитель сам бы создавал экземпляр служебного объекта и переадресовывал бы ему всю реальную работу.