Министерство науки и высшего образования РФ

Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Высшая школа «Киберфизических систем и управления»

**ОТЧЕТ**

по дисциплине «Системный подход в разработке ПО»

Задания по материалу занятия 12

**Выполнил:**

студент гр. 3530902/90201 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ О. В. Позолотин

подпись, дата

**Проверил:**

доцент, к.т.н \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ С. А. Нестеров

подпись, дата

Санкт-Петербург

2023 г.

**Ход работы**

**Подключение c MySQL**

**Изображение выглядит как текст

Автоматически созданное описание**

**Изображение выглядит как текст

Автоматически созданное описание**

**Изображение выглядит как текст

Автоматически созданное описание**

**Пользователь и порт**

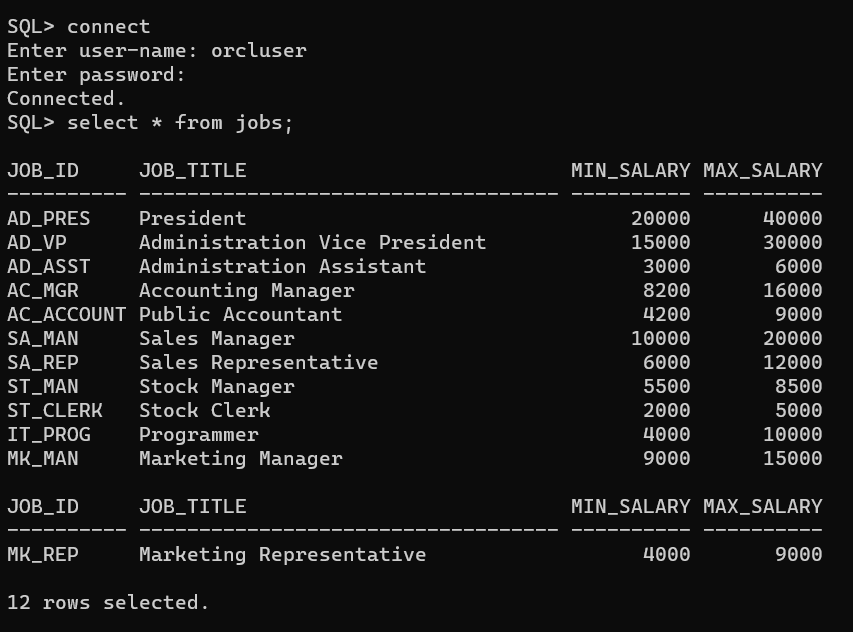
**Изображение выглядит как текст

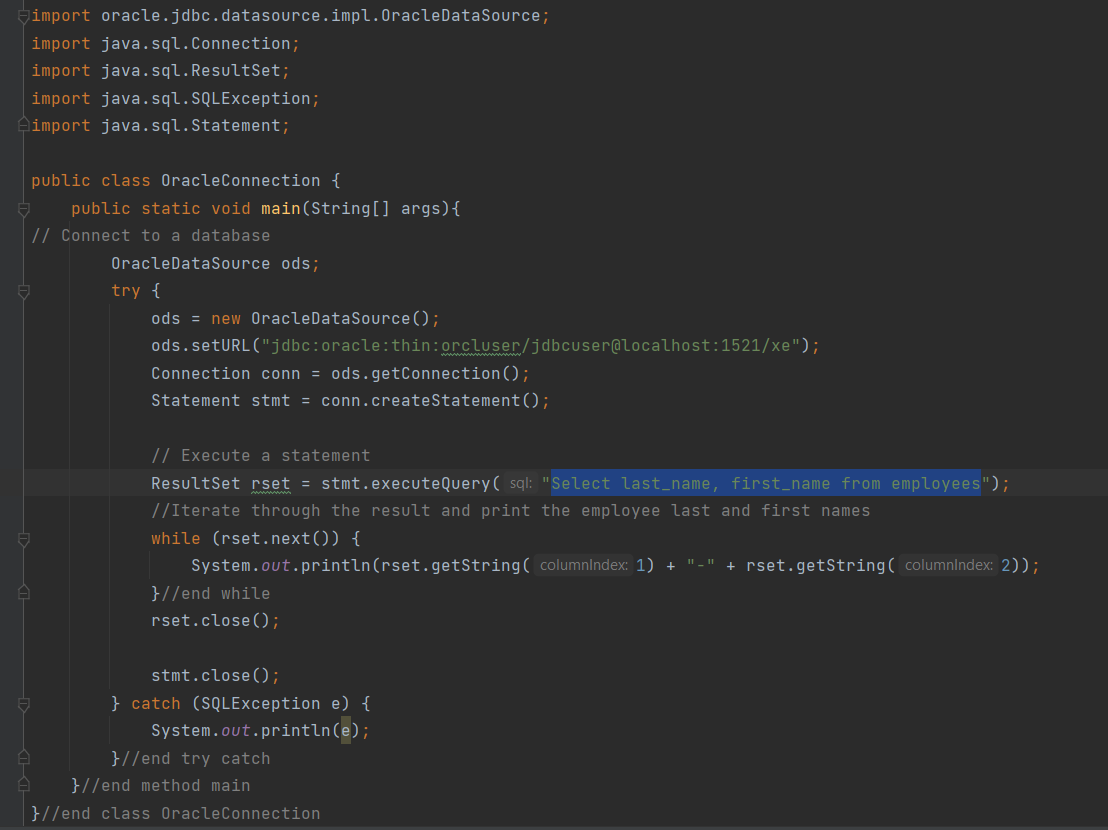
Автоматически созданное описание**

**Изображение выглядит как диаграмма

Автоматически созданное описание**

**Презентация 6.1**

****

****

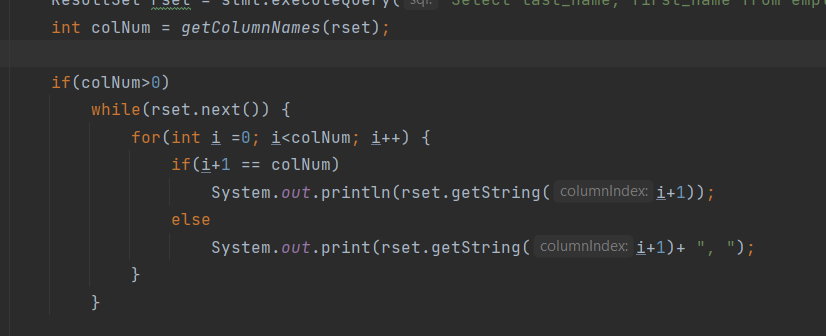
**Результат работы:**

**Изображение выглядит как текст

Автоматически созданное описание**

**Изображение выглядит как текст

Автоматически созданное описание**

****

**Результат:**

**Изображение выглядит как текст

Автоматически созданное описание**

**Создайте переменные для username, password и query используйте имена переменных в коде вместо их значений.**

**Изображение выглядит как текст

Автоматически созданное описание**

**Изображение выглядит как текст

Автоматически созданное описание**

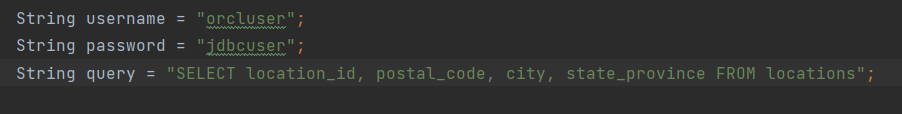
**Измените запрос, чтобы вывести department\_id, department\_name и location\_id для каждого отдела.**

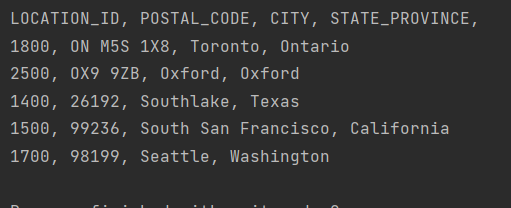
**Изображение выглядит как текст

Автоматически созданное описаниеИзображение выглядит как текст

Автоматически созданное описание**

**Измените запрос, чтобы вывести location\_id, postal\_code, city и state\_province для каждой локации.**

****

****

**Практика**

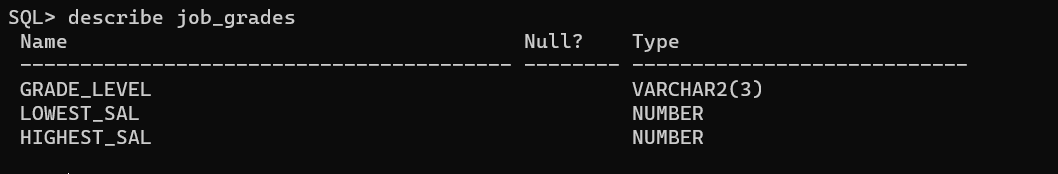
**Vocabulary**

|  |  |
| --- | --- |
|  | The JDBC-ODBC Bridge plus ODBC Driver |
| Statement Interface | The interface used for executing a static SQL statement and returning the results it produces |
| Driver manager | The class which manages a set of JDBC Driver |
| PreparedStatement Interface | The interface which is able to provide information for the tables, stored procedures and so on. |

**2.**

2. Identify the structure of the database tables that you will be working with. a) Open SQLPlus and log in using the orcluser account.

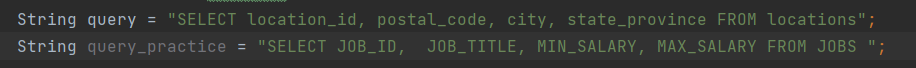
b) Identify the structure of each of the tables by running a DESCRIBE command on the table. Identify column names and data types. The syntax for a DESCRIBE command is: DESCRIBE tablename;

****

**Изображение выглядит как текст

Автоматически созданное описание**

**3.**

****

**Изображение выглядит как текст

Автоматически созданное описание**

**JDBC**

**Драйверы, соединения и запросы**

API JDBC (Java DataBase Connectivity) — стандартный прикладной интерфейс языка Java для организации взаимодействия между приложением и СУБД. Взаимодействие осуществляется с помощью драйверов JDBC, обеспечивающих реализацию общих интерфейсов для конкретных СУБД и конкретных протоколов. В JDBC определяются четыре типа драйверов:

1. Драйвер, использующий другой прикладной интерфейс взаимодействия с СУБД, в частности, ODBC (так называемый JDBC-ODBC — мост). Стандартный драйвер первого типа sun.jdbc.odbc.JdbcOdbcDriver входит в JDK.

2. Драйвер, работающий через внешние native библиотеки клиента СУБД.

3. Драйвер, работающий по сетевому и независимому от СУБД протоколу с промежуточным Java-сервером, который, в свою очередь, подключается к нужной СУБД.

4. Сетевой драйвер, работающий напрямую с нужной СУБД и не требующий установки native-библиотек.

Предпочтение естественным образом отдается второму типу, однако если приложение выполняется на машине, на которой не предполагается установка клиента СУБД, то выбор производится между третьим и четвертым типами.

Последовательность действий для выполнения первого запроса.

1. Подключение библиотеки с классом-драйвером базы данных.

2. Установка соединения с БД.

Connection cn = DriverManager.getConnection("jdbc:oracle:thin:@//localhost:1521:testphones", "system", "pass");

3. Создание объекта для передачи запросов.

Statement st = cn.createStatement();

4. Выполнение запроса.

ResultSet rs = st.executeQuery("SELECT \* FROM phonebook");

5. Обработка результатов выполнения запроса.

6. Закрытие соединения, statement.

st.close(); // закрывает также и ResultSet

cn.close();

**СУБД MySQL**

СУБД MySQL совместима c JDBC и будет применяться для создания учебных баз данных

**Простое соединение и простой запрос**

При создании таблицы следует задавать кодировку UTF-8, поддерживающую хранение символов кириллицы.

В несложном приложении достаточно контролировать закрытие соединения, так как незакрытое или «провисшее» соединение снижает быстродействие системы

Параметры соединения можно задавать несколькими способами: с помощью прямой передачи значений в коде класса, а также с помощью файлов properties или xml. Окончательный выбор производится в зависимости от конфигурации проекта.

**Метаданные**

Существует целый ряд методов интерфейсов ResultSetMetaData и DatabaseMetaData для интроспекции объектов. С помощью этих методов можно получить список таблиц, определить типы, свойства и количество столбцов БД. Для строк подобных методов нет.

String getDatabaseProductName() — возвращает название СУБД;

String getDatabaseProductVersion() — возвращает номер версии СУБД;

String getDriverName() — возвращает имя драйвера JDBC;

String getUserName() — возвращает имя пользователя БД;

String getURL() — возвращает местонахождение источника данных;

ResultSet getTables() — возвращает набор типов таблиц, доступных для данной БД.

**Подготовленные запросы и хранимые процедуры**

Для представления запросов существует еще два типа объектов PreparedStatement и CallableStatement. Объекты первого типа используются при выполнении часто повторяющихся запросов SQL. Такой оператор предварительно готовится и хранится в объекте, что ускоряет обмен информацией с базой данных при многократном выполнении однотипных запросов. Второй интерфейс используется для выполнения хранимых процедур, созданных средствами самой СУБД.

В JDBC также существует механизм batch-команд, который позволяет запускать на исполнение в БД массив запросов SQL вместе, как одну единицу.

Если используется объект PreparedStatement, batch-команда состоит из параметризованного SQL-запроса и ассоциируемого с ним множества параметров.

Метод executeBatch() интерфейса PreparedStatement возвращает массив чисел, причем каждое характеризует число строк, которые были изменены конкретным запросом из batch-команды.

**Транзакции**

Транзакция, или деловая операция, определяется как единица работы, обладающая свойствами ACID:

• Атомарность — две или более операций выполняются все или не выполняется ни одна. Успешно завершенные транзакции фиксируются, в случае неудачного завершения происходит откат всей транзакции.

• Согласованность — при возникновении сбоя система возвращается в состояние до начала неудавшейся транзакции. Если транзакция завершается успешно, то проверка согласованности удостоверяется в успешном завершении всех операций транзакции.

• Изолированность — во время выполнения транзакции все объекты-сущности, участвующие в ней, должны быть синхронизированы.

• Долговечность — все изменения, произведенные с данными во время транзакции, сохраняются, например, в базе данных. Это позволяет восстанавливать систему.

Для фиксации результатов работы SQL-операторов, логически выполняемых в рамках некоторой транзакции, используется SQL-оператор COMMIT.

Для транзакций существует несколько типов чтения:

• грязное чтение (dirty reads) происходит, когда транзакциям разрешено видеть несохраненные изменения данных. Иными словами, изменения, сделанные в одной транзакции, видны вне ее до того, как она была сохранена. Если изменения не будут сохранены, то, вероятно, другие транзакции выполняли работу на основе некорректных данных;

• неповторяющееся чтение (nonrepeatable reads) происходит, когда транзакция А читает строку, транзакция Б изменяет эту строку, транзакция А читает ту же строку и получает обновленные данные;

• фантомное чтение (phantom reads) происходит, когда транзакция А считывает все строки, удовлетворяющие WHERE-условию, транзакция Б вставляет новую или удаляет одну из строк, которая удовлетворяет этому условию, транзакция А еще раз считывает все строки, удовлетворяющие WHERE-условию, уже вместе с новой строкой или недосчитавшись старой. JDBC удовлетворяет уровням изоляции транзакций, опред

**Точки сохранения**

Точки сохранения, представляемые классом java.sql.Savepoint, дают дополнительный контроль над транзакциями, привязывая изменения СУБД к конкретной точке в области транзакции. Установкой точки сохранения обозначается логическая точка внутри транзакции, которая может быть использована для отката данных. Таким образом, если произойдет ошибка, можно вызвать метод rollback(Savepoint point) для отмены всех изменений, которые были сделаны после точки сохранения. Метод boolean supportsSavepoints() интерфейса DatabaseMetaData используется для того, чтобы определить, поддерживает ли точки сохранения драйвер JDBC и сама СУБД. Методы setSavepoint(String name) и setSavepoint() возвращают объект Savepoint интерфейса Connection и используются для установки именованой или неименованой точки сохранения во время текущей транзакции. При этом новая транзакция будет начата, если в момент вызова setSavepoint() не будет активной транзакции.

**Data Access Object**

При создании информационной системы выявляются некоторые слои, которые отвечают за взаимодействие различных частей приложения. Связь с базой данных является важной частью любой системы, поэтому всегда выделяется часть кода, ответственная за передачу запросов в БД и обработку полученных от нее ответов. Общее определение шаблона Data Access Object трактует его как прослойку между приложением и СУБД. DAO абстрагирует бизнессущности системы и отражает их на записи в БД. DAO определяет общие способы использования соединения с БД, моменты его открытия и закрытия или извлечения и возвращения в пул. В общем случае DAO можно определять таким образом, чтобы была возможность подмены одной модели базы данных другой. Например: реляционную заменить на объектную или, что проще, MySQL на Oracle. В практическом программировании такие глобальные задачи ставятся крайне редко, поэтому будет приведено несколько способов организации взаимодействия с БД, отличающихся уровнем использования коннекта к БД и организацией работы с бизнес-сущностями.

**DAO. Уровень метода**

Реализация DAO для конкретного бизнес-объекта имеет шанс выглядеть следующим образом. Часть методов может остаться нереализованной, кроме того, могут добавляться собственные методы, определить которые в более общем классе невозможно из-за узкой области применения. В данном случае это метод Abonent findAbonentByLastName(String name).

Реализация на уровне метода предполагает использование соединения для выполнения единственного запроса, т. е. соединение будет получено из пула в начале работы метода и возвращено по его окончании, что в общем случае не является экономным решением.

**DAO. Уровень класса**

Реализация на уровне класса подразумевает использование одного коннекта к базе данных для вызова нескольких методов конкретного DAO класса. В этом случае вершина иерархии DAO в качестве поля может объявлять сам коннект к СУБД или его оболочку, кроме стандартного набора методов, например:

**DAO. Уровень логики**

На практике чаще всего возникает необходимость при выполнении запроса пользователя обращаться сразу к нескольким ветвям DAO и использовать при этом единственное соединение с БД. В этом случае соединение с БД создается или извлекается из пула до создания экземпляров DAO, а закрывается, соответственно, после выполнения всех обращений к БД.

**Глава 10**

**ЛЯМБДА-ВЫРАЖЕНИЯ**

**Синтаксис лямбда-выражения**

Лямбда-выражение можно рассматривать как некоторый блок кода, опре деляющий действие. Также можно рассматривать лямбда-выражение как определение анонимного метода — то есть метода, у которого нет имени (аналогия не совсем точная, то во многих случаях помогает понять логику происходящего). Используются лямбда-выражения в основном как альтернатива к созданию анонимных классов.

Лямбда-выражение состоит из аргументов и тела. То есть здесь все очень близко к тому, как описывается метод, но только нет названия и идентификатора типа результата. В общем случае аргументы в лямбда-выражении описываются в круглых скобках. Если аргументов несколько, то между собой они разделяются запятыми. Для аргументов указывается тип (хотя может и не указываться — это мы еще обсудим). Команды в теле лямбда-выражения разделяются между собой точкой с запятой, а весь блок команд заключается в фигурные скобки. Между аргументами и блоком команд (телом лямбда-выражения) указывается «стрелка» ->

Есть некоторые правила, которые позволяют упрощать синтаксис описания лямбда-выражений.

• Тип аргумента можно не указывать, если он быть идентифицируется исходя из контекста команды использования лямбда-выражения.

• Если в лямбда-выражении один аргумент, тип которого не указан, то круглые скобки можно не использовать.

• Если в лямбда-выражении нет аргументов, то используются пустые круглые скобки.

• Если в теле лямбда-выражения всего одна команда, то фигурные скобки можно не использовать. В случае, когда единственная команда в теле лямбда-оператора состоит из return-инструкции, ключевое слово return можно не использовать.

**Функциональные интерфейсы**

Функциональный интерфейс — это интерфейс с одним и только одним абстрактным методам. То есть ничего особенного в функциональном интерфейсе нет. Просто должно быть выдержано одно условие: в интерфейсе объявляется всего один абстрактный метод.

Проекта UsingLambdaApplication

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Проект UsingAnonymousClassApplication

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

**Несколько интерфейсов и ссылка на метод**

Одинаковые лямбда-выражения могут присваиваться переменным разных функциональных интерфейсов. Достаточно, чтобы абстрактный метод в каждом таком интерфейсе соответствовал параметрам (таким, как количество и тип аргументов) лямбда-выражения. Ниже приведен небольшой пример, в котором описывается три разных, но однотипных интерфейса, и в каждом из них объявляется метод, который не возвращает результат, и у которого нет аргументов. В главном методе программы объявляются интерфейсные переменные, значением которым присваивается одно и то же (по форме) лямбда выражение ()->System.out.println("Используем лямбда-выражение"), соответствующее методу без аргументов, при выполнении которого в окне вывода отображается сообщение.

Проект LambdaAndInterfacesApplication

**Изображение выглядит как текст

Автоматически созданное описание**

**Изображение выглядит как текст

Автоматически созданное описание**

**Ссылка на метод и конструктор**

Чтобы понять принципы выполнения и использования ссылок на методы, удобно представлять, что значением выражения со ссылкой на метод является инструкция, аналогичная лямбда-выражению, которое соответствует коду метода (на который выполняется ссылка). Проще говоря, ссылка на метод может рассматриваться как лямбда-выражение, соответствующее коду метода, на который выполняется ссылка.

**Ссылка на метод объекта**

Если речь идет о выполнении ссылки на нестатический метод конкретного объекта, то выполняется она просто (и пример выполнения такой ссылки мы видели): указывается имя объекта, и через оператор :: указывается название метода. То есть шаблон выполнения ссылки на нестатический метод следующий : объект::имя\_объекта

Проект ObjMethReferenceApplication

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Мы описываем класс MyClass с закрытым целочисленным полем number, конструктором и двумя метолами: set() для присваивания значения полю и get() для считывания значения поля.

Есть еще два функциональных интерфейса. В интерфейсе MyGetter объявлен метод myget() без аргументов, возвращающий целочисленный результат. В интерфейсе MySetter объявлен метод myset(), у которого целочисленный аргумент и который не возвращает результат. Таким образом, сигнатура метода myget() из интерфейса MyGetter соответствует сигнатуре метода get() из класса MyClass, а сигнатура метода myset() из интерфейса MySetter соответствует сигнатуре метода set() из класса MyClass.

**Ссылка на нестатический метод класса**

Ссылка может выполняться не только на метод объекта, но и на метод класса. В последнем случае имеет значение, является ли метод статическим, или нет. Сначала рассмотрим случай, когда метод нестатический.

Ссылка на нестатический метод некоторого класса выполняется в следующем виде

класс::имя\_метода

Проект MethReferenceApplication

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Результат выполнения программы немного изменился, если сравнивать с предыдущим примером. Но многие блоки кода остались неизменными. Так, совершенно не изменилось описание класса MyClass. Зато изменилось объявление методов myget() из интерфейса MyGetter и myset() из интерфейса MySetter. Дело в том, что в главном методе программы, как и ранее, создается объект obj класса MyClass, но ссылку на методы get() и set() из класса MyClass мы выполняем не через объект, а через класс, соответственно, в формате MyClass::get и MyClass::set. Что означает, например, ссылка MyClass::get? Метод get() описан без аргументов и это нестатический метод, поэтому должен вызываться из объекта. С учетом реального кода метода get(), ссылка MyClass::get соответствует лямбда-выражению вида (MyClass obj)->{obj.get()}.

**Ссылка на статический метод**

Ссылка на статический метод класса формально выполняется так же, как и на нестатический метод класса, но последствия, так сказать, другие. В частности, если используется ссылка на метод вида класс::метод, и при этом метод является статическим и описан с некоторыми аргументами, то указанная выше ссылка эквивалентна лямбда-выражению вида (аргумент)->класс.метод(аргументы). Например, ссылка System.out::println эквивалентна лямбда-выражению t->System.out.println(t). Соответствующим образом данная ссылка и должна использоваться.

Проект StatMethReferenceApplication

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

В главном методе программы есть примеры присваивания интерфейсным переменным в качестве значений ссылок на статические методы. Речь о командах MyShow A=MyClass::show, MySum B=MyClass::sum и MyPrinter P=System. out::println. В результате при выполнении команды A.myshow() на самом деле из класса MyClass вызывается метод show(). При выполнении команды B.mysum(10) из класса MyClass вызывается статический метод sum() с аргументом 10, а вызов метода myprint() из переменной P с аргументом " Сумма чисел: "+B.mysum(10) аналогичен выполнению команды System.out.println("Сумма чисел: "+B.mysum(10)).

**Ссылка на конструктор**

Ссылку можно выполнять не только на метод, но и на конструктор. Синтаксис выполнения ссылки на конструктор такой: после имени класса указывается оператор ::, после которого следует ключевое слово new. Вся конструкция выглядит следующим образом

класс::new

Если использована ссылка на конструктор вида класс::new, то ее эквивалентом является лямбда-выражение вида (аргументы)->new класс(аргументы), где подразумевается, что при создании объекта класса конструктору передаются аргументы.

Проект ConstructorReferenceApplication

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст, апельсин, снимок экрана, близко

Автоматически созданное описание

В главном методе программы командой MyInterface ref=MyClass::new переменной ref интерфейсного типа MyInterface значением присваивается ссылка MyClass::new на конструктор класса MyClass. Интерфейс MyInterface является функциональным. В нем объявлен только один абстрактный метод create(), у которого единственный целочисленный аргумент. Результатом метод возвращает значение типа MyClass. Поэтому значением интерфейсной переменной типа MyInterface может быть присвоена ссылка на метод с целочисленным аргументом, возвращающий значение типа MyClass. Этим критериям удовлетворяет выражение вида new MyClass(число), результатом которого является объектная ссылка типа MyClass. Таким образом, после выполнения команды MyInterface ref=MyClass::new вызов из переменной ref метода create() с целочисленным аргументом эквивалентен вызову (через оператор new) конструктора класса MyClass с таким же аргументом.

**Ссылка на перегруженный метод**

Вполне может сложиться ситуация, когда необходимо выполнить ссылку на метод, который перегружен . В таком случае в классе описано несколько версий метода или, если быть более точными, то несколько методов с одинаковыми названиями. Собственно по ссылке на метод нельзя однозначно определить, о какой версии метода идет речь. Вывод о том, какую версию метода следует использовать, делается на основе интерфейсной переменной, которой в качестве значения присваивается ссылка на метод.

Проект OverloadedMethRefApplication

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

В главном методе программы сначала создается объект obj класса MyClass (команда MyClass obj=new MyClass()), а затем командами Alpha A=obj::set и Bravo B=obj::set одна и та же ссылка obj::set присваивается переменным разного типа. Поскольку в интерфейсе Alpha абстрактный метод описан без аргументов, то при выполнении команды Alpha A=obj::set используется версия метода set() без аргументов. А вот в интерфейсе Bravo абстрактный метод описан с целочисленным аргументом, поэтому в команде Bravo B=obj::set задействована версия метода set() с целочисленным аргументом. Как следствие, при выполнении команды B.one(100) из объекта obj вызывается метод set() с аргументом 100, а при выполнении команды A.none() из объекта obj вызывается метод set() без аргументов.

**Использование лямбда-выражений**

**Передача лямбда-выражения аргументом методу**

Задач, когда функции аргументом передается функция (методу аргументом передается метод) достаточно много. Мы рассмотрим одну, но «классическую» — вычисление интеграла.

Для вычисления интеграла необходимо задать:

• функцию f(x), которая находится под знаком интеграла и называется подынтегральной функцией;

• границы a и b области интегрирования.

Проект IntegralCalcApplication

**Изображение выглядит как текст

Автоматически созданное описание**

Изображение выглядит как текст

Автоматически созданное описание

**Лямбда-выражение и результат метода**

Лямбда-выражение может использоваться при определении результата метода. Формальным результатом в таком случае является значение интерфейсного типа, но собственно в коде метода результат может быть «оформлен» в виде лямбда-выражения. Мы именно такой случай и рассмотрим. Как «учебную» рассмотрим задачу о вычислении производной для функции.

Проект DerivativeCalcApplication

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

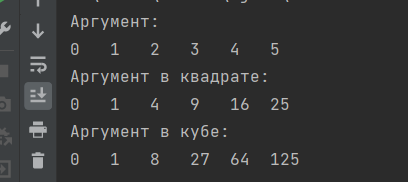
В методе main() командой MyFunction A=Derivative((double x)->{return x\*(3-x);}) объявляется интерфейсная переменная A, значение которой вычисляется вызовом методе Derivative(). Аргументом методу передается лямбда-выражение, которое означает, что «дифференцируется» функция f(x) = x(3 – x). Результатом возвращается ссылка на объект класса, реализующего интерфейс MyFunction, и в этом объекте метод f() определяет производную f'(x) (точное значение для производной f'(x) = 3 – 2x). Аналогично, командой MyFunction B=Derivative((double x)->{return x\*Math.exp(-x);}) определена переменная B, ссылающаяся на объект, метод f() которого соответствует производной от функции f(x) = x · exp(–x) (точное значение для производной f'(x) = exp(–x)(1 – x)). Далее запускаются операторы цикла, с помощью которых значения для производных вычисляются в нескольких точках (для нескольких значений аргумента).

**Лямбда-выражение и поле объекта**

Проект LambdaAsFieldApplication

Изображение выглядит как текст

Автоматически созданное описание



В главном методе командой MyClass obj=new MyClass((int n)->{return n\*n;}) создается объект obj класса MyClass. Аргументом конструктору передано лямбда-выражение. В результате поле ref объекта obj ссылается на объект, в котором метод getNumber() по аргументу n возвращает квадрат аргумента (значение n\*n). Но поскольку метод getNumber() вызывается не напрямую, а через метод get() объекта obj, то «иллюзия» такая, как если бы это метод get() был определен для возведения аргумента в квадрат.

После выполнения команды obj.set((int n)->{return n\*n\*n;}) технически происходит следующее. Создается новый объект класса, реализующего интерфейс MyInterface, в котором метод getNumber() вычисляет третью степень аргумента. Ссылка на этот объект записывается в поле ref объекта obj. Но внешний эффект такой, как будто мы «переопределили» метод get() объекта obj.