МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Чернігівська політехніка»

JAVA ТЕХНОЛОГІЇ ПРИКЛАДНОГО ПРОГРАМУВАННЯ

МЕТОДИЧНІ ВКАЗІВКИ
до лабораторного практикуму та самостійної роботи з дисципліни
«Технології прикладного програмування»
для студентів спеціальності 123 – "Комп'ютерна інженерія"

ЗАТВЕРДЖЕНО
на засіданні кафедри
інформаційних та комп'ютерних систем
протокол № 1 від 27.08.21

Чернігів 2021

Java технології прикладного програмування. Методичні вказівки до лабораторного практикуму та самостійної роботи з дисципліни «Технології прикладного програмування» для студентів спеціальності 123 – “Комп’ютерна інженерія” /Укл.: Бивойно П.Г., Бивойно Т.П. – Чернігів: НУ «ЧЕРНІГІВСЬКА ПОЛІТЕХНІКА», 2021. – 148с.

Укладачі: Бивойно Павло Георгійович, канд. техн. наук, доцент;
Бивойно Тарас Павлович, ст. викладач

Відповідальний за випуск: В.М. Базилевич, зав. кафедрою інформаційних і комп'ютерних систем, канд. екон. наук

Рецензент: О.А. Пріла, доцент кафедри інформаційних та комп’ютерних систем, канд. техн. наук, доцент

ЗМІСТ

ВСТУП

Корпоративний додаток (enterprise application) являє собою програму, призначену для обробки великого обсягу даних по бізнес правилам, впровадження якої дозволяє надати певні переваги корпорації (підприємству).

Корпоративним додатком не є засоби обробки тексту, системи регулювання витрати палива в автомобільному двигуні або автоматичного контролю хімічних процесів. Не є ними також і операційні системи, компілятори, ігри, тощо.

Корпоративний додаток зазвичай передбачає необхідність довготривалого (іноді протягом десятиліть) зберігання даних. Дані часто здатні пережити кілька поколінь прикладних програм, призначених для їх обробки, апаратних засобів, операційних систем і компіляторів.

Безліч користувачів звертаються до даних паралельно. Як правило, їх кількість не перевищує сотні, але для систем, розміщених в середовищі Web, цей показник зростає на кілька порядків.

У таких додатках повинен бути розроблений і потужний користувацький інтерфейс.

Корпоративні додатки не існують в ізоляції. Зазвичай вони вимагають інтеграції з іншими системами, побудованими в різний час із застосуванням різних технологій.

У даних методичних вказівках розглядаються різні Java-технології, що дозволяють спростити розробку складних програмних систем:

- Java reflection;
- JDBC;
- JAXB, JSON;
- JPA;
- Java Servlet, Java Server Pages;
- JAX-RS (Web Services).

Лабораторні роботи виконуються відповідно до вибраного варіанту РГР із переліку завдань, що наведені нижче. Варіант студент обирає сам і реєструється у відповідній папці ресурсу «мудл».

По кожній роботі студент повинен створити java-проєкт, провести дослідження та оформити електронний звіт у відповідності з вимогами ДСТУ 3008:2015 Файли проєктів та звіти завантажуюься у відповідні папки ресурсу «мудл». Завдання, що виконуються в межах самостійної роботи, є складовими РГР.

За лабораторну роботу студент може отримати до 100 балів, з урахуванням своєчасності та якості виконання всіх складових роботи. Складовими є: звіт, результати виконання досліджень та завдання до самостійної роботи і відповіді на контрольні питання під час співбесіди. Оцінки, отримані за лабораторні роботи, враховуються при виставленні підсумкової оцінки. Для отримання заліку всі роботи повинні бути виконані і кожна з них оцінена не менше ніж в 60 балів.

ВАРІАНТИ ТЕМ РГР			
Варіант	Таблиці		
1.Залізниця	місто	вокзал	рейс
2.Автобусні перевезення	вокзал	напрям	рейс
3. Аеропорт	місто	аеропорт	рейс
4. Виробництво комп'ютерів	країна	фірма	комп'ютер
5. Продаж оргтехніка	місто	магазин	товар
6. Торгова база	склад	група товарів	товар
7. Демографія	регіон	область	місто/село
8. Екологія	область	район/місто	екопроблеми
9. Місто	вулиця	будинок	квартира
10. Комунальні служби	жред	професія	робітник
11. МОК	олімпіада	вид спорту	спортсмен
12. УПЛ	клуб	амплуа	спортсмен
13. Ігрові чемпіонати	що, де, коли	команди	учасники
14. Комендант	корпус	аудиторія	меблі
15. Фармацевтична фірма	країна	відділ	ліки
16.Торгова мережа	товар	постачальники	поставки
17. Підприємство	вироби	вузли	деталі
18. Бібліотека	кафедра	предмет	література
19. Університет	факультет	кафедра	викладач
20. Кафедра	предмет	викладачі	студенти
21. Ресторан	кухня	блюдо	продукти
22. Лікарня	відділення	хворий	показники
23. Космос	зірки	планети	супутники
24. Мої витрати	дата(3 поля)	вид, ліміт	що, де, сума
25. Географія	континенти	регіони	держави
26. Гуртожиток	блок	кімната	мешканець
27. Мережева торгівля	продавець	товар	реалізація
28. Поліклініка	лікар	пацієнт	рецепт
29. Музика	жанр	група	пісня
30. Відпочинок	країна	тур	учасники
31. Суд	суддя	справа	підсудні

1 ЛАБОРАТОРНА РОБОТА № 1.
ДОСЛІДЖЕННЯ ДИНАМІЧНОГО ВИЗНАЧЕННЯ ТИПІВ

1.1 Мета роботи

В результаті виконання лабораторної роботи студент має отримати такі знання і навички:

– ознайомитися з механізмом рефлексії в Java;
– навчитися створювати об’єкти класу Class і використовувати їх методи;
– познайомитися з класами, що забезпечують роботу із jar-файлами;
– закріпити набуті знання і набути навички реалізації механізму рефлексії під час реалізації завдання для самостійної роботи.

1.2 Короткі теоретичні відомості

У Java вбудована можливість динамічного завантаження довільного класу по заданому імені, скорочено RTTI (Run-time type information). Така можливість розширює можливості поліморфізму, бо реалізації інтерфейсів можна визначати та змінювати на етапі виконання програми, без переписування коду, завантажуючи потрібні класи із jar-файлів, які можуть знаходитися будь де. Це є однією з основних особливостей платформи Java.

1.2.1 **Файли з розширенням .jar**

Як відомо, програма на Java являє собою сукупність класів. Такі сукупності класів найчастіше зберігаються у так званих jar-файлах. Це звичайні архівні файли, але вони містять відкомпільований байт-код класів у файлах з розширенням .class. Окрім того jar-файли можуть містити і вихідні тексти класів у файлах з розширенням .java. Jar-файли можуть використовуватися як бібліотеки класів, але їх можна і запускати для виконання на віртуальній машині Java. В останньому випадку у jar-архіві обов’язково має бути файл MANIFEST, який містить інформацію про клас, де знаходиться метод main(), з якого починається робота програми, та іншу мета-інформацію.

Для роботи з jar-файлами у Java-програмах використовуються класи JarFile та JarEntry, що знаходяться у пакеті java.util.jar.

Клас JarFile використовується для читання вмісту jar-файлу. Цей клас розширює клас java.util.zip.ZipFile і забезпечує підтримку читання файлу маніфесту. Ось деякі конструктори класу JarFile:

– JarFile(String name) throws IOException;
– JarFile(File file) throws IOException.

Методи об’єктів класу JarFile забезпечують доступ до його складових за допомогою ітератора або стриму:

– public Enumeration<JarEntry> entries()
– public Stream<JarEntry> stream()

Клас JarEntry використовуються для створення об’єктів, що надають інформацію про складові jar-архіву.

Методи об’єктів класу JarEntry дозволяють отримати характеристики цих складових. Ось деякі з них:

– getName();
– getSize();
– isDirectory() .

1.2.2 **Механізм завантаження класів**

Для того, щоб програма почала працювати, байт-код класів має бути завантажений у пам’ять комп’ютера. Класи завантажуються за потребою, але деякі базові класи, зокрема класи пакета java.lang завантажуються при старті програми.

Для завантаження класів використовується декілька видів завантажувачів.

Стандартні завантажувачі для пошуку класів використовуючи Buid Path.

Але якщо ім’я jar-файлу стає відомим тільки на етапі виконання програми, то можна скористатися власним завантажувачем.

Для того, щоб створити такий завантажувач, необхідно успадкувати клас java.lang.ClassLoader.

Базовою інформацією для об’єктів цього класу має бути ім’я jar-файлу, в якому знаходяться потрібні класи. Це дозволить створити об’єкт типу JarFile, який допоможе отримати із jar-файлу байткод класів.

У новоствореному класі слід також передбачити сховище для посилань на класи, що вже були завантажені, щоб уникнути повторного завантаження того самого класу.

Далі слід реалізувати абстрактний метод

public Class<?> findClass(String name)

У цей метод як параметр передається назва класу, а сам метод має завантажити клас і повернути об’єкт типу Class. Про клас Class мова буде йти у наступному пункті.

Метод findClass реалізується за таким алгоритмом:

– спочатку робиться спроба завантажити клас із сховища, або із системи;
– якщо ця спроба невдала, то байткод класу отримується через файлову систему;
– далі ім’я класу і відповідний байткод передається до методу defineClass, що визначений у класі ClassLoader. Цей метод завантажує байткод та повертає об’єкт класу Class.

Саме за цією схемою реалізовано клас JarClassLoader, що забезпечує завантаження класів із заданого jar-файлу. Текст цього класу наведено у додатку А до лабораторної роботи і його можна використовувати для виконання завдання для самостійної роботи.

1.2.3 **Клас Class**

Під час виконання java-програми вся інформація про клас зберігається в спеціальному об'єкті типу Class, який існує для кожного класу і створюється при завантаженні класу.

Є різні способи отримання доступ до об’єкту типу Class. Розглянемо деякі з них.

Перший спосіб полягає у використанні літералу class, який додається через крапку до назви класу. Наприклад, щоб отримати такий обїєкт для класу String, можна скористатися таким кодом:

```
Class clazz = String.class;
```

Це проста, але досить дивна, нестандартна конструкція, бо класи не мають публічного статичного поля class. Кажуть, що у цьому випадку ми просто додаємо до імені класу суфікс .class.

Другий спосіб полягає у зверненні до об’єкта будь якого класу через метод getClass(), що визначений у класі Object. Попередній приклад можна переписати так:

```
Class clazz = “qwerty”.getClass();
```

Результат буде той самий.

Ще один зі способів полягає у використанні статичного методу forName() класу Class. Цей метод повертає екземпляр Class для класу, ім’я якого передаємо через параметр. Але виклик цього методу потребує обробки виключення.

Наприклад:

```
try {
        Class clazz = Class.forName ("java.lang.String");
} catch (ClassNotFoundException e) {
        e.printStackTrace();
}
```

Слід зазначити, що об’єкти типу Class можна отримати також для масивів і примітивних типів. Для цього теж можна використовувати суфікс .class, наприклад:

```
Class clazz = String[].class;
Class clz  = int.class;
```

Окрім того, кожний клас обгортка для примітивних типів має публічну константу TYPE, яка містить посилання на об’єкт типу Class для відповідного примітивного типу. Наприклад,

```
Class clz  =  Integer.TYPE;
```

1.2.4 **Використання об’єктів класу Class**

Об’єкти класу Class дозволяють отримати повну інформацію про клас, створювати об’єкти відповідних класів, викликати методи і таке інше.

1.2.4.1 Інформація про структуру класу

Інформація про структуру класу надається у вигляді об’єктів типів Constructor, Method, Field та інших з пакета java.lang.reflect. Доступ до цих об’єктів можна отримати через методи об’єтів класу Class. Перелік таких методів, що повертають інформацію про методи класу (об’єкти типу Method), наведено у таблиці 1.1. Аналогічні методи існують і для інших складових класу.

Складова Declared у назві метода означає, що надається доступ до усіх методів класу, навіть приватних, але не методів суперкласів. Відсутність цієї складової вказує на те, що надається доступ тільки до публічних методів, включаючи методи суперкласів.

Параметрами методів, що повертають об’єкт типу Method, є назва метода та перелік типів його параметрів у вигляді об’єктів типу Class. Наприклад,

```
Class clz = String.class;
Method mtd = clz.getMethod("lastIndexOf",String.class, int.class);
```

У цьому прикладі ми отримуємо об’єкт типу Method для методу lastIndexOf класу String. Метод має два параметри. Перший типу String і другий типу int.

Познайомитися з усіма методами доступу до складових класу можна відкривши текст класу Class.

Таблиця 1.1 – Типи та методи, що характеризують клас Method

Тип результату	Метод об’єкту класу Class
Method	getMethod(String name, Class … param)
	getDeclaredMethod(String name, Class … param)
Method[]	getMethods()
	getDeclaredMethods()

1.2.4.2 Тестування різновиду класу

У класі Class представлена також велика група методів, що дозволяють протестувати наявність тих, чи інших ознак класу, що представлений об’єктом типу Class. У таблиці 1.2 наведено деякі з цих методів. Повний перелік можна знайти у тексті класу.

Таблиця 1.2 – Методи тестування об’єктів типу Class

Метод	Коментар
boolean isArray()	Визначає, чи відповідає даний об’єкт масиву
boolean isInterface()	Визначає, чи представляє даний об’єкт інтерфейс
boolean isPrimitive()	Визначає, чи представляє даний об’єкт примітивний тип

1.2.4.3 Визначення модифікаторів

Як клас, так і його складові зазвичай характеризуються сукупністю модифікаторів. Для визначення сукупності модифікаторів використовують метод int getModifiers(), що повертає ціле число, біти якого вказують на наявність або відсутність того чи іншого модифікатора. Константи, що визначають біти для кожного модифікатора завантажені у клас java.lang.reflect.Modifier. Цей же клас містить статичні методи, які дозволяють визначити, чи закодований у цілому числі той, чи інший модифікатор. Наприклад, метод Modifier.isPublic(int) повертає true, якщо біт, відповідальний за модифікатор public містить одиничку. Аналогічно виглядають методи для перевірки наявності інших модифікаторів, наприклад, isAbstract(int), isStatic(int).

Якщо потрібно отримати перелік модифікаторів у вигляді рядка символів, можна скористатися статичним методом toString(int) класу Modifier. Метод перетворює ціле число у перелік модифікаторів. Наприклад, щоб отримати перелік модифікаторів константи PI з класу java.lang.Math, можна скористатися кодом, що наведено нижче.

```
Class clz = Class.forName("java.lang.Math");
Field f = clz.getDeclaredField("PI");
String sModif = Modifier.toString(f.getModifiers());
```

Після його виконання змінна sModif прийме значення

```
public static final.
```

У версії Java 8 для об’єктів типу Class з’явився метод toGenericString(), який повертає у вигляді рядка символів докладну інформацію про клас, включаючи і модифікатори. Такий самий метод можна викликати і для окремих складових класу, якщо є необхідність отримати інформацію у вигляді тексту.

Наприклад, попередній приклад можна переписати так:

```
Class clz = Class.forName("java.lang.Math");
Field f = clz.getDeclaredField("PI");
String s = f.toGenericString();
```

Після його виконання отримаємо такий рядок символів:

```
public static final double java.lang.Math.PI
```

1.2.4.4 Створення об’єктів

Об’єкт класу Class надає можливість створювати нові екземпляри відповідних класів. Для цього використовується метод newInstance(). Метод повертає посилання на об’єкт, який приведено до типу Object.

Наприклад:

```
Class clzz = String.class;
String s = (String) clzz.newInstance();
```

Метод newInstance() для об'єктів класу Class можна використовувати, якщо клас має пустий конструктор. Для виклику конструктора з параметрами слід скористатися об’єктом типу Constructor, і через нього викликати конструктор,

скориставшись методом newInstance():

```
Class clazz = Double.class;
Constructor cstr = clazz.getConstructor(double.class);
Object obj = cstr.newInstance(123.234);
```

1.2.4.5 Звернення до методів

Для того, щоб через механізм reflection викликати метод, треба створити об'єкт типу Method для цього методу, а також створити об'єкт класу, для якого буде викликатися метод, якщо метод не статичний. Після цього для об'єкту типу Method викликається метод invoke(). Першим параметром цього методу має бути об'єкт, для якого викликається метод. Якщо метод статичний, то цей параметр буде null. Наступними мають бути параметри, яких потребує сам метод.

В якості прикладу наведемо виклик статичного методу valueOf(int) класу String для числа 35.

```
Class clzz = String.class;
Method mth = clzz.getDeclaredMethod("valueOf", int.class);
String str = (String) mth.invoke(null,35);
```

Наступний приклад демонструє непрямий виклик методу indexOf(String) для об'єкту класу String.

```
String str = "abcd";
Class clzz = str.getClass();
Method mth = clzz.getDeclaredMethod("indexOf", String.class);
int pos = (int) mth.invoke(str,"bc");
```

1.2.4.6 Звернення до полів

Технологія reflection надає можливість непрямого звертання і до полів об'єктів, навіть приватних. Хай, наприклад, є клас із приватним полем:

```
public class TestPrivateField {
     private int x = 10;
     public int getX(){
        return x;
     }
}
```

Використовучи технологію reflection можна з іншого класу поміняти значення приватного поля, додавши до методу main такий код:

```
try {
     TestPrivateField obj = new TestPrivateField();
     Field fld = obj.getClass().getDeclaredField("x");
     fld.setAccessible(true);
     fld.set(obj, 20);
     System.out.println(obj.getX());
} catch (Exception e1) {
        e1.printStackTrace();
}
```

1.3 Порядок виконання роботи

В роботі будемо створювати два проєкти.

Перший проєкт буде містити клас діалогу та клас тестування цього діалогу звичайним способом. Для цього проєкту далі буде також створено jar-файл, класи якого будуть досліджуватися і використовуватися за допомоою технології рефлексії.

Другий проєкт буде містити візуальний застосунок, який дозволить досліджувати jar-файл, що було створено у першому проєкті, та використовувати класи цього jar-файлу. Застосунок також має надавати інформацію про розробника проєкту.

В якості прикладу далі буде розглядатися варіант, який складається з трьох таблиць – кафедра, випускник, робота.

Студент мають реалізовувати класи відповідно до своїх варіантів.

1.3.1 **Створення проєкту з діалогом**

Створити новий java-проєкт і пакет.

В якості класу, що буде досліджуватися за допомогою технології reflection, ствоімо візуальний клас діалогу для введення інформації для першої таблиці варіанта. У нашому прицладі – це кафедра. Відповідно клас діалогу назвемо DlgCathedra.

Назви класів у проєктах студентів мають відповідати вибраному варіанту. Кількість полів у діалозі має бути не менше 3. Типи даних мають бути різні, але логічні поля зберігати як ціле число, а дати у вигляді рядків. Це пов’язано з тим, що у різних фреймворках специфічні дані мають свої особливості.

1.3.1.1 Візуальна композиція діалогу

На рисунку 1.1 наведено вигляд цього діалогу у режимі дизайну.

Не забувайте, що діалог має бути модальним.

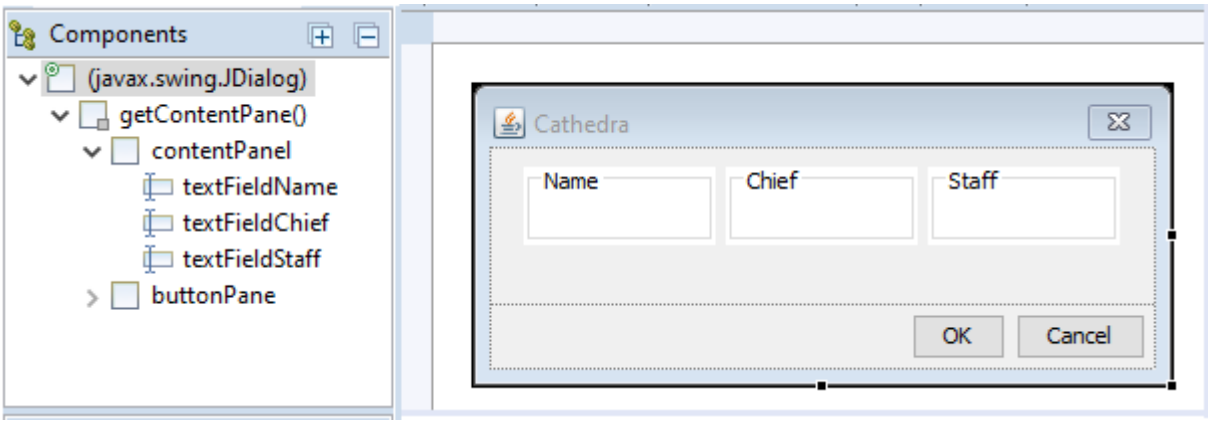

Рисунок 1.1 – Діалог DlgCathedra в режимі дизайну

1.3.1.2 Програмування діалогу

Завдання діалогу – приймати інформацію через графічний інтерфейс

користувача і перетворювати її у зручний для подальшого використання об’єкт. Окрім того, діалог має приймати і відображати відповідну інормацію.

В якості об’єкта, який повертає та приймає діалог, будемо використовувати асоціативний масив (карту) типу java.util.Map, з реалізацією у класі LinkedHashMap.

Пропишемо у класі два поля:

```
private Map<String,Object> map;
private int idCathedra;
```

Друге поле потрібно для зберігання у діалозі унікального ідентифікатора поля у базі даних, який користувач не створює. Значення цього ідентифікатора не виводиться у вікні діалогу, але зберігається в карті.

Для створення діалогу, що відображає надану інформацію, створімо конструктор з параметром:

```
public DlgCathedra(Map<String, Object> map) {
   this();
   textFieldName.setText((String) map.get("name"));
   textFieldChief.setText((String) map.get("chief"));
   textFieldStaff.setText(String.valueOf(map.get("staff")));
   idCathedra = (int) map.get("id");
}
```

Для очистки полів діалогу створімо метод clear():

```
public void clear() {
   textFieldName.setText("");
   textFieldChief.setText("");
   textFieldStaff.setText("0");
   idCathedra = 0;
   map = null;
}
```

Для обробки події натискання на кнопку ОК створімо метод onOk(). Цей метод зчитує інформацію з полів діалогу і, в разі успішного перетворення, створює карту і заносить туди дані про кафедру:

```
private void onOk() {
   String name = textFieldName.getText();
   String chief = textFieldChief.getText();
   String staffTxt = textFieldStaff.getText();
   int staff = 0;
   try {
      staff = Integer.parseInt(staffTxt);
   } catch (NumberFormatException e) {
      JOptionPane.showMessageDialog(this, "\""
            + staffTxt +"\" is not correct number");
      return;
   }
```

```
        map = new LinkedHashMap<>();
        map.put("id", idCathedra);
        map.put("name", name);
        map.put("chief", chief);
        map.put("staff", staff);
        setVisible(false);
    }
```

Для обробки події натискання на кнопку Cancel створімо метод onCancel():

```
    private void onCancel() {
        map = null;
        setVisible(false);
    }
```

На завершення створімо метод доступу до карти, яку створює діалог:

```
    public Map<String, Object> getMap(){
        return map;
    }
```

1.3.1.3 Тестування діалогу в звичайному режимі

Для перевірки працездатності діалогу створімо у проєкті клас TestDlgCathedra з методом main. У цьому методі створений діалог активізується, а після його закриття карта виводиться на консоль.:

```
    public class TestDlgCathedra {
        public static void main(String[] args) {
            DlgCathedra dlg = new DlgCathedra();
            dlg.setVisible(true);
            System.out.println(dlg.getMap());
            dlg.dispose();
        }
    }
```

1.3.2 **Дослідження можливостей технології reflection**

Створімо клас TestReflection з методом main, в якому слід написати код для отримання характеристик класу діалогу.

Результати виконання виводити на консоль. У звіті зафіксувати код класів і результати тестування.

Зокрема, слід отримати:

– характеристику самого класу скориставшись методами toGenericString(), getName(), getSimpleName(), getSuperclass();
– перелік його полів з характеристиками;
– перелік його конструкторів з характеристиками;
– перелік методів з характеристиками;
– через технологію reflection створити об’єкт класу, що досліджується;
– використовуючи технологію reflection отримати доступ до приватних полів типу JTextField та заповнити їх інформацією;

– через технологію reflection викликати його метод setVisible(true) . В результаті діалог має з’явитися на екрані і поля мають бути заповнені;

– через технологію reflection викликати метод getMap() для діалогу. Результат виконання методу вивести на консоль.

Нижче, для прикладу, наведено фрагменти методу main, що реалізує завдання. Вам треба доопрацювати цей код відповідно до свого варіанту і наведеного вище переліку завдань:

```
public static void main(String[] args) {
  try {
     Class clz = Class.forName("lab1.DlgCathedra");

     System.out.println("\nРезультат методу toGenericString()");
     System.out.println(clz.toGenericString());
     //...

     System.out.println("\nРезультати getDeclaredFields()");
     Field[] fld = clz.getDeclaredFields();
     for (Field field : fld) {
        System.out.println(field.toGenericString());
     }
     //...

     System.out.println("\nРезультат створення об'єкта");
     DlgCathedra dlg = (DlgCathedra) clz.newInstance();
     System.out.println(dlg);
     // Занесення даних у приватні поля діалогу
      Field field = clz.getDeclaredField("textFieldStaff");
      field.setAccessible(true);
      JTextField fieldStaff = (JTextField) field.get(dlg);
      fieldStaff.setText("5");
     // ...

     // Тут потрібно через рефлексію
     // викликати метод setVisible(true) для об’єкта dlg

     System.out.println("\nРезультат виклику метода getMap()");
     // Тут потрібно через рефлексію
     // викликати метод getMap() для об’єкта dlg
     // і отриману карту вивести на консоль
  } catch (Exception e) {
     e.printStackTrace();
  }
}
```

1.3.3 **Створення jar-файлу проєкту**

На завершення роботи з проєктом, треба засобами Eclipse створити jar-файл проєкту. Для зручності перевірки назва jar-файлу має починатися з префікса dlg, далі має йти прізвище студента, наприклад, dlgByvoino.

1.4 Завдання для самостійної роботи

На попередньому етапі ми використовували можливості рефлекії для класу, що знаходився у проєкті. Тому завдання, що були реалізовані, можна було виконати і без використання рефлексії.

На даному етапі ми маємо створити проєкт, що дозволить отримати в процесі роботи програми доступ до класу стороннього jar-файлу, який не прописано в BuidPath.

В якості сторонього jar-файлу будемо використовувати jar-файл проєкту, що було створено на попередньому етапі лабораторної роботи.

В ході самостійної роботи необхідно реалізувати візуальний застосунок Вимоги до застосунку такі:

– шлях до jar-файлу визначати через діалог пошуку файлу, який активізується опцією меню;

– після вибору jar-файлу у вікні застосунку має з’явитися перелік класів, що знаходяться в архіві;

– друга опція меню має дозволити перегляд характеристик класу діалогу (наприклад, перелік полів або конструкторів) а також створення діалога і виведення у вікно введеної інформації;

– у меню має бути також опція виклику довідки про розробника з фотографією.

1.5 Рекомендації з виконання завдання для самостійної роботи

1.5.1 **Створення проєкту**

Створімо новий проєкт і в ньому два пакети, наприклад, lab1 і resource. У другий пакет завантажимо фото розробника. Завантаження можна зробити через контекстне меню проєкту, викликавши функції Import... → FileSystem → Next. Далі слід знайти папку із зображенням і виділити потрібний файл у переліку. Щоб побачити файл у пакеті після завантаження, треб зробити refresh проєкту.

1.5.2 **Підключення jar-файлу JarClassLoader**

Оскільки jar-файл, з яким ми збираємось працювати, знаходиться за межами нашого проєкту, для завантаження класів із цього jar-файлу доведеться використовувати свій ClassLoader. Текст відповідного класу JarClassLoader наведено у додатку А. Бажано ознайомитися з кодом цього класу, щоб мати уяву про процес завантаження класу.

Jar-файл з класом завантажувача можна скачати до проєкту зі сторінки курсу на ресурсі «мудл» і підключити його через функцію контекстного меню BuildPath.

Можна також скопіювати код класу із додатка А і включити до пакету. Але в цьому випадку слід мати на увазі, що обмін файлами java-коду з Word може призвести до зміни деяких символів. Найчастіше – це лапки.

1.5.3 **Інтерфейс користувача**

Інтерфейс користувача додатку кожен може створити на свій розсуд, але нижче розглядається реалізація на основі Swing Application Window, що показана на рисунку 1.2. В будь-якому випадку у титулі застосунку має бути прізвище судента і група.

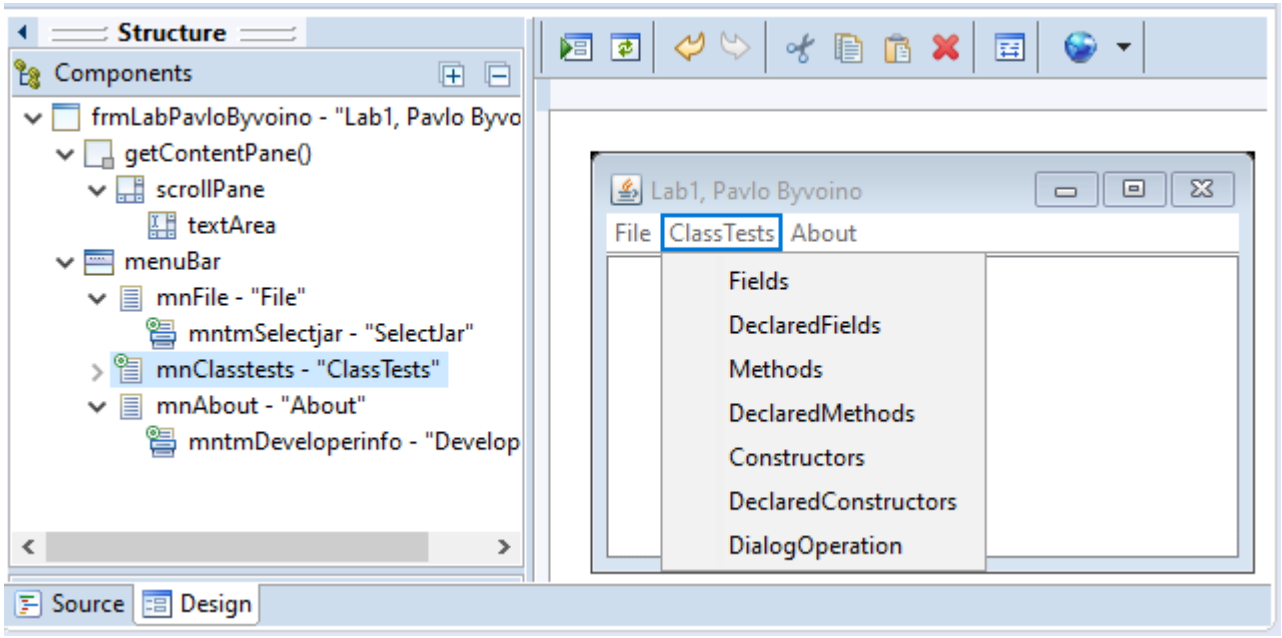

Рисунок 1.2 – Можливий інтерфейс користувача додатку

1.5.4 **Метод для пошуку імені jar-файлу**

Питання, пов’язані з використанням компонентів для пошуку файлів детально розглядалися у лабораторній роботі “Потоки введення-виведення і серіалізація об’єктів” по курсу ООП. Аналогічний діалог використовувася і в розрахунково графічній роботі.

Нижче наведено приклад такого методу, де використовується компонент FileDialog. Його зручно використовувати, тому що він запам’ятовує шлях до файлу із попереднього сеансу. Окрім того, у метод в якості параметра передаємо рядок символів, який задає розширення файлу і використовується для налаштування фільтру вибору файлів діалогу.

```
String getFileName(String filter) {
   FileDialog fileDialog = new FileDialog(frame);
   fileDialog.setMode(FileDialog.LOAD);
   fileDialog.setFile(filter);
   fileDialog.setVisible(true);
   String dr = fileDialog.getDirectory();
   String fn = fileDialog.getFile();
   if (dr == null || fn == null)
      return null;
   return dr + fn;
}
```

1.5.5 **Метод для відкриття та обробки jar-файлу**

Цей метод аналізує вибраний jar-файл і завантажує потрібий нам клас. Для зберігання посилання на завантажений клас створімо приватне поле clz:

```
private Class clz;
```

Далі пов’яжемо з кнопкою меню File→SelectJar наступний метод:

```
protected void onSelectJar() {
   String fileName = getFileName("*.jar");
   //JarFile object creation
   JarFile jarFile = null;
   try {
      jarFile = new JarFile(fileName);
   } catch (Exception e) {
      JOptionPane.showMessageDialog(frame, "Problem with .jar file");
      e.printStackTrace();
      return;
   }
   // ClassLoader creation
   JarClassLoader  loader = new JarClassLoader(fileName);
   //Extract classes from jar-file
   Enumeration<JarEntry> enm = jarFile.entries();
   clz = null;
   textArea.setText("Entry list from jar "+ fileName + "\n");
   // We are only interested in one class from jar
   String searchName = "lab1/DlgCathedra.class";
   while (enm.hasMoreElements()) {
      JarEntry entry = enm.nextElement();
      String name = entry.getName();
      textArea.append(name +"\n");
      if(name.equals(searchName)) {
         //File name correction.
         String cs = name.replace('/', '.');
         cs = cs.substring(0, cs.lastIndexOf(".class"));
         try {
            // Class loading
            clz = loader.findClass(cs);
            textArea.append(clz + " was loaded\n");
         } catch (ClassNotFoundException e) {
            e.printStackTrace();
            return;
         }
      }
   }
}
```

У першому рядку цього метода для отримання імені jar-файлу використовується метод getFileName, який було розглянуто вище.

Далі, на основі цього імені створюється об’єкт jarFile, який забезпечить обробку вибраного jar-файлу.

Створюється також об’єкт loader для завантаження класів з вибраного jar-файлу.

Для опрацювання jar-файлу використовується ітератор, який надає об’єкт jarFile через метод entries().

За допомогою цього ітератора послідовно друкуються та аналізуються усі складові файлу. Якщо складова jar-файлу є класом, який нам потрібен, то назва класу редагується у відповідності з вимогами Java і цей клас завантажується. Посилання на класс зберігається у полі clz.

1.5.6 **Методи реалізації функцій меню ClassTests**

Осільки об’єкт класу Class вже сформовано, то методи реалізації функцій меню ClassTests можуть використовувати фрагменти коду методу main класу TestReflection, який реалізовано на попередньому етапі.

Студенти мають реалізувати ці методи самостійно.

На рисунку 1.3 показано результат виклику функції меню «Declared Fields».

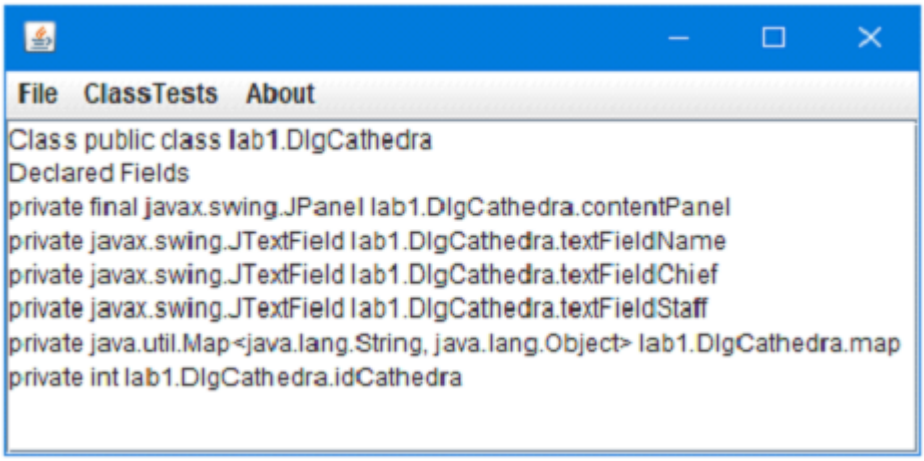

Рисунок 1.3 – Можливий ваіант виведення результатів

1.5.7 **Реалізація функції меню About→DeveloperInfo**

Ця функція меню має надавати інформацію про розробника проєкту. Після вибору цієї функції на екрані має з’являтися віконце, де має бути повне ім’я студента, група, номер індивідуального плану, адреса електронної пошти і номер телефону.

Окрім того, зважаючи на можливість дистанційного навчання, треба мати фото студента, щоб у викладача була більш повна уява про автора проєкту.

Реалізуємо це завдання.

1.5.7.1 **Створення вікна з інформацією про студента**

Згенеруємо у пакеті візуальний клас Info на основі класу JFrame.

Переходимо у режим дизайну і формуємо заголовок вікна «Developer information».

Налаштовуємо GridBagLayout для головної панелі contentPane.

У другому рядку контейнера розміщуємо JtextArea.

У першому рядку розміщуємо звичайну панель JPanel.

Далі записуємо текстову інформацію про студента відкривши редактор тексту для властивості text компонента textArea, рисунок 1.4.

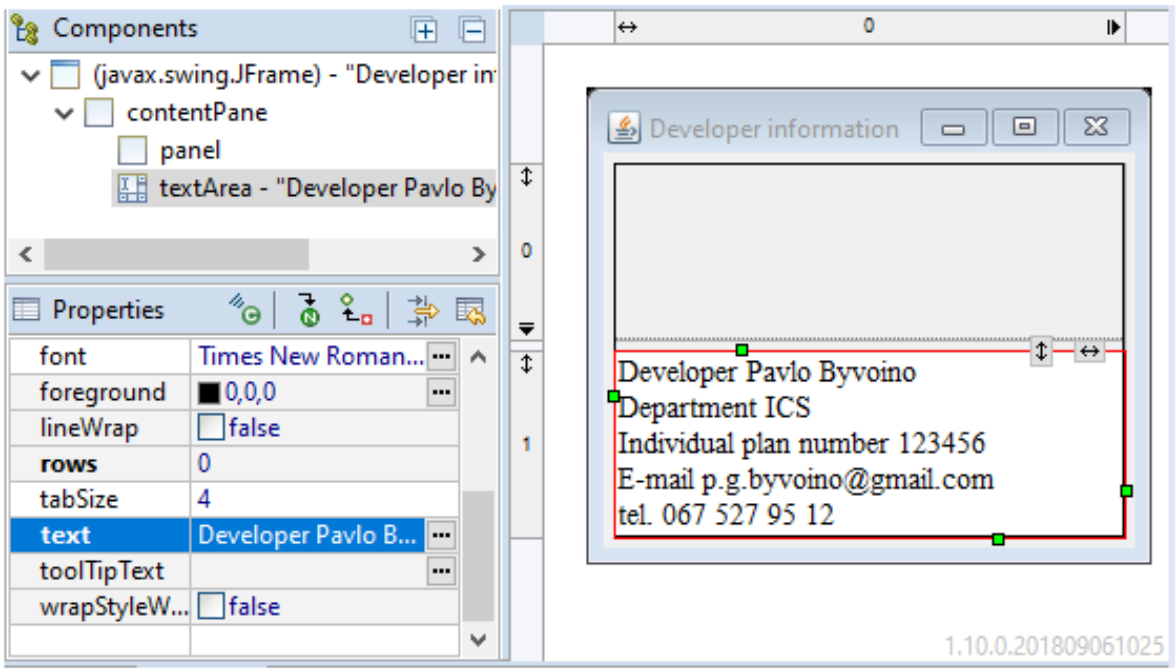

Рисунок 1.4 – Запис інформації про студента

Можна також налаштувати шрифт повідомлення, поле editable…

На завершення налаштуйте властивість менеджера компоновки Grow = none для нижнього рядка, а для верхнього і колонки налаштуйте Grow = 1.

1.5.7.2 **Розміщення фото студента**

Найпростіший спосіб вставити фото – це скористатися компонентом JLabel. У цього компонента є властивість icon, яку можна пов’язати з зображенням за допомогою редактора цієї властивості. Але у цьому випадку доведеться підлаштовувати розмір компонента під розмір фото, що не завжди зручно.

У нашому вікні для розміщення фото ми будемо використовувати компонент JPanel, точніше його нащадка.

Кожен компонент Swing має вбудований об’єкт типу Graphics за допомогою якого можна намалювати що завгодно на поверхні компонента (дивіться останню лабораторну роботу з ООП). Але проблема полягає в тому, що система постійно перемальовує компонент, тому зображення зникає. Нам потрібно змінити панель таким чином, щоб вона перемальовувала себе використовуючи наше зображення.

Для початку треба мати фото. Імпортуйте його у папку resource проєкту, якщо не зробили цього раніше.

Далі змінемо ім’я панелі на panelPhoto та зробимо її доступною через функцію контекстного меню Expose Component.

В результаті у класі буде створено поле panelPhoto і метод getPanelPhoto. який слід переписати відповідно з наведеним нижче кодом. У наведеному тексті мається на увазі, що файл з фото має назву «photo.jpg» і знаходиться у папці resource проєкту.

Доведеться також зробити імпорт класів, що використовуються, зокрема, java.net.URL.

```
private JPanel getPanelPhoto() {
  if (panelPhoto == null) {
    //Create panel as anonymous class object
    panelPhoto = new JPanel() {
      //override method paintComponent in anonymous class
      public void paintComponent(Graphics g){
        super.paintComponent(g);
        Graphics2D g2d = (Graphics2D) g;
        BufferedImage img;
        // set path to photo
        URL url = getClass().getResource("/resource/photo.jpg");
        try {
          //reading photo to image
          img = ImageIO.read(url);
        } catch (IOException e) {
          e.printStackTrace();
          return;
        }

        //image scaling according to panel size
        double k = (double)img.getHeight() / img.getWidth();
        int width = getWidth();
        int height = getHeight();
        if((double)height / width > k)
          height = (int) (width *k);
        else
          width = (int) (height /k);
        Image scaledImg = img.getScaledInstance(
            width, height, Image.SCALE_SMOOTH);
        //show photo
        g2d.drawImage(scaledImg,0,0,null);
      };
    };
  }
  return panelPhoto;
}
```

У методі створюється панель, для якої перевизначено метод paintComponent.

У перевизначеному методі спочатку викликається код неперевизначеного методу через super.

Далі фото зчитується в об'єкт img типу BufferedImage.

Об'єкт img трансформується у відповідності з розміром панелі, після чого трансформоване зображення відмальовується на поверхні панелі.

Наступним кроком поміняємо в конструкторі класу спосіб створення об’єкту panelPhoto таким чином, щоб панель відображала фото одразу після сворення.

Для цього у конструкторі класу Info замість рядка

```
panelPhoto = new JPanel();
```

вводимо

```
panelPhoto = getPanelPhoto();
```

На завершення роботи з фреймом треба в конструкторі класу Info змінити рядок, в якому програмується реакція на закриття фрейму.

Цей рядок має такий вигляд:

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

його треба замінити на такий:

```
setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
```

Якщо цього не зробити, то одночасно із закриттям фрейму буде закриватися і головне вікно, через яке буде викликатися фрейм.

Тепер можна наш фрейм запустити і подивитися, що вийшло.

1.5.7.3 **Виклик інформації із головного вікна**

Перейдемо до класу головного вікна і пов’язуємо з функцією меню About→DeveloperInfo такий метод:

```
protected void onDeveloperInfo() {
  textArea.setText("");
  new Info().setVisible(true);
}
```

1.5.8 **Створення jar-файлу другого проєкту**

Цей jar-файл треба створити за допомогою функції контекстного меню проєкту Export→Java→RannableJarFile. У діалозі, що з’являється, треба вибрати клас, в якому знаходиться метод main, через який запускається проєкт, та визначитися з ім’ям і розташуванням файлу, що створюється.

У назві файлу має бути префікс lab1, після чого має йти прізвище студента, наприклад, lab1Byvoino.jar.

1.6 Вимоги до звіту

У звіті має бути представлено таку інформацію:

– назва роботи як заголовок розділу;
– мета роботи як підрозділ;

– підрозділ з результатами розробки першого проєкту (скрншоти візуальних компонентів та код, що було написано власноручно) та результатів тестування окремими пунктами;

– підрозділ з результатами розробки другого проєкту (скрншоти візуальних компонентів та код, що було написано власноручно) та результатів тестування окремими пунктами;

– підрозділ висновків.

На «мудл» завантажувати zip-файл з файлом зіту та jar-файлами першого та другого проєктів.

1.7 Контрольні питання

1. Яке призначення класу Class.
2. Як отримати об’єкт типу Class, відповідний даному класу.
3. Назвіть основні класи пакету java.lang..
4. Як отримати перелік структурних елементів класу.
5. Яким чином здійснюється розкриття модифікаторів класу.
6. Як викликати потрібний метод через рефлексію.
7. Як відбувається завантажування класу із jar-файлу.
8. Reflection і безпека.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Java Reflection Tutorial from Sun Microsystems. [Електронний ресурс]. – Режим доступу:https://docs.oracle.com/javase/tutorial/reflect/index.html
2. Reflection in object-oriented programming. [Електронний ресурс]. – Режим доступу: https://www.worldscientific.com/doi/abs/10.1142/S0218213092000156
3. Reflection in logic, functional and object-oriented programming: a short comparative study. [Електронний ресурс]. – Режим доступу: https://www.researchgate.net/publication/2732177_Reflection_in_logic_functional_and_object-oriented_programming_a_Short_Comparative_Study
4. Forman, Ira R. and Forman, Nate. Java Reflection in Action. — Manning Publications Co., 2004. — ISBN 1932394184.
5. Jonathan M. Sobel and Daniel P. Friedman. An Introduction to Reflection-Oriented Programming (1996), Indiana University.

ДОДАТОК А

Лістинг А.1 – Клас JarClassLoader

```
package jtLab1;

import java.io.IOException;
import java.io.InputStream;
import java.util.HashMap;
import java.util.Map;
import java.util.jar.JarEntry;
import java.util.jar.JarFile;

/**
 * Клас використовується для створення об'єктів,
 * що дозволяють завантажувати класи із jar-файлу,
 * ім'я якого передаємо через конструктор.
 * Для завантаження класу викликаємо метод findClass(String),
 * до якого передаємо назву класу, як параметр.
 * Роздільниками у назві класу мають бути крапки.
 */
public class JarClassLoader extends ClassLoader {
   private JarFile jar;
   private Map<String, Class<?>> loadedMap = new HashMap<>();

   // До конструктора передаємо назву jar-файлу
   public JarClassLoader(String jarFileName) {
      super(JarClassLoader.class.getClassLoader());
      // JarFile objct creation
      try {
         jar = new JarFile(jarFileName);
      } catch (Exception e) {
         System.err.println("Open jar file error");
         return;
      }
   }
    /**
    * Метод для завантаження класу із jar-файлу.
    * name - це повна назва класу в пакеті,
    * наприклад "pack1.pack2.MyClass".
    */
   @Override
   public Class<?> findClass(String name) throws ClassNotFoundException {
      // Спроба взяти клас з карти(можливо вже завантажувався)
      Class<?> c = loadedMap.get(name);
      if (c != null)
         return c;
      // Спроба завантажити клас через SystemClassLoader
      try {
         return findSystemClass(name);
      } catch (Exception e) {
```

```
      }
      // Якщо не вийшло, отримуємо байкод класу із файлу "name"
      //і завантажуємо його
      byte[] b; //Посилання на масив для байтів класу

      // Отримуємо байт код класу,
      // текст приватного методу loadClassData наведено далі
      b = loadClassData(name);
      // Завантажуємо клас
      try {
         c = defineClass(name, b, 0, b.length);
         // Зберігаємо клас у карті
         loadedMap.put(name, c);
      } catch (Throwable e) {
         throw new ClassNotFoundException(e.getMessage());
      }
      return c;
   }
   /**
   * Метод повертає байткод класу name із jar-файлу jar
   */
   private byte[] loadClassData(String name) throws
                                    ClassNotFoundException {
      // Формування ім'я entry з імені класу
      String entryName = name.replace('.', '/') + ".class";
      // Створення об'єкту entry
      JarEntry entry = jar.getJarEntry(entryName);
      if (entry == null)
         throw new ClassNotFoundException(name);
      // Створння масиву для байт коду
      int size = new Long(entry.getSize()).intValue();
      byte[] buf = new byte[size];
      // Спроба введення даних
      try {
         //Метод getInputStream розпаковує файл на який посилається entry
         InputStream input = jar.getInputStream(entry);
         int count = input.read(buf);
         // Контроль кількості введених байтів
         if (count < size)
            throw new ClassNotFoundException("Size? Error reading class '"
                  + name + "' from jar: " + jar.getName());
         // Close the input stream
         input.close();
      } catch (IOException e1) {
         throw new ClassNotFoundException(e1.getMessage());
      }
      // return bytes
      return buf;
   }
}
```

2 ЛАБОРАТОРНА РОБОТА 2. ВИКОРИСТАННЯ JDBC

2.1 Мета роботи

В результаті виконання лабораторної роботи студент має отримати такі знання і навички:

– ознайомитися із засобами Java, що використовуються для підключення до реляційних баз даних;

– навчитися виконувати запити на створення бази даних, модифікацію даних та запити на отримання інформації з бази в класах Java;

– отримати практичні навички створення java застосунку для роботи з базою даних.

2.2 Короткі теоретичні відомості

2.2.1 **Вступ до JDBC**

JDBC (Java DateBase Connectivity) – це сукупність класів та інтерфейсів, що визначають правила доступу до систем управління реляційними базами даних (СУБД) із Java-програм, використовуючи SQL.

JDBC фактично має дві складові.

Перша складова – це класи та інтерфейси пакета java.sql, що визначають інтерфейс роботи з базою даних і використовуються розробниками застосунків.

Друга складова (нижчого рівня) – це інтерфейси для розробників драйверів, які забезпечують зв’язок з базою даних.

Архітектура системи, що використовує JDBC наведена на рисунку 2.1.

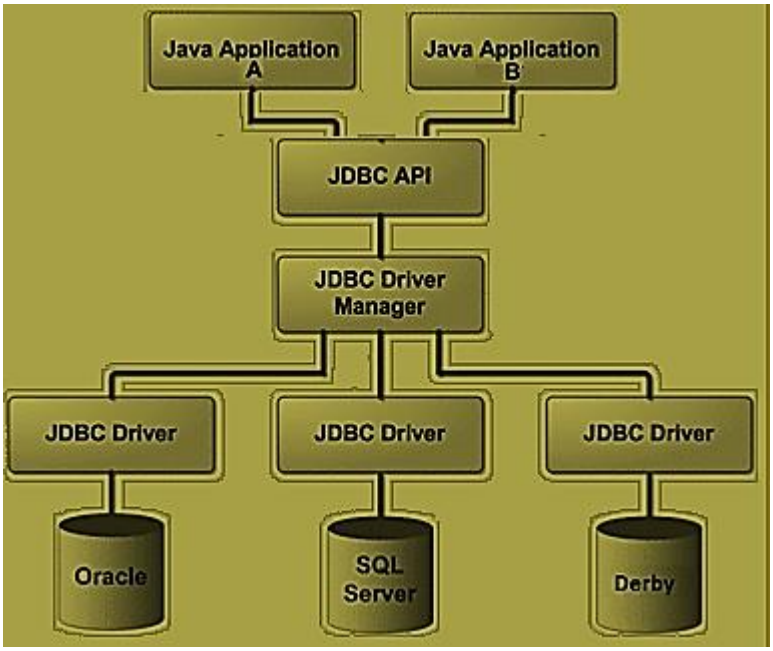

Рисунок 2.1 – JDBC архітектура

З рисунку 2.1 видно, що Java-додатки (Application A та Application B) спілкуються із СУБД через посередника, яким є JDBC. Наявність посередника дозволяє використовувати стандартний інтерфейс спілкування із СУБД незалежно від її типу.

2.2.2 **Драйвери JDBC**

Драйвер JDBC – це сукупність класів, що реалізують інтерфейс JDBC та забезпечують зв'язок з базою даних. Але слід розуміти, що постачальники драйверів не завжди повністю реалізують вимоги інтерфейсів JDBC.

Існуючі драйвери бувають чотирьох типів (рисунок 2.2).

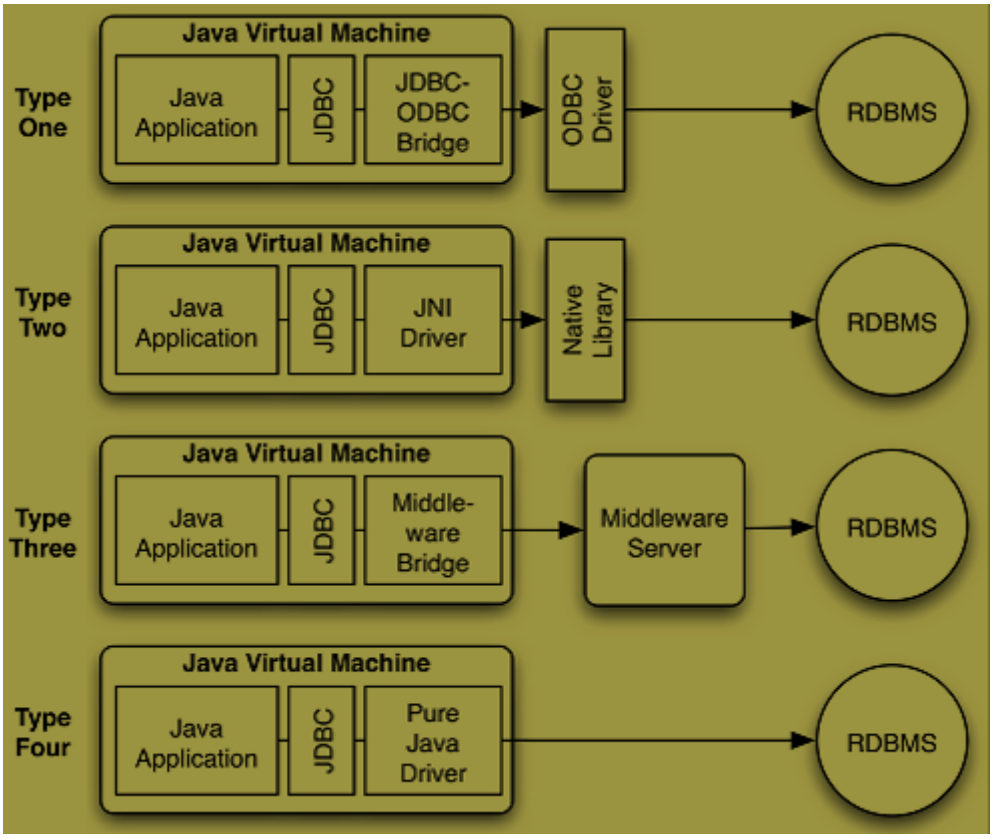

Рисунок 2.2 – Типи JDBC драйверів

Тип 1. Драйвери, що реалізують API JDBC як відображення в інший API доступу до даних. Така технологія отримала назву "bridge". Прикладом драйверів цього типу є JDBC-ODBC міст, що поставлявся з JDK до версії Java8. Така технологія потребує встановлення додаткового програмного забезпечення на клієнтській системі.

Тип 2. Більш швидкі драйвери, написані частково на мові програмування Java і містять Java-код, який викликає методи на C або C ++, що забезпечують доступ до баз даних. Прикладом драйвера такого типу є драйвер Oracle OCI (Oracle Call Interface). Це рішення також вимагає встановлення відповідної бібліотеки на клієнтській системі, яку надає розробник бази дагих.

Тип 3. Драйвери цього типу використовуються для спілкування з базою даних через проміжний сервер, використовуючи незалежний від бази даних протокол. Проміжний сервер передає запити клієнтів до джерела даних.

Тип 4. Драйвери цього типу написані на Java для певного джерела даних. Клієнт за допомогою такого драйвера підключається безпосередньо до джерела даних. Оскільки реалізація драйверу цього типу залежить від особливостей конкретної СУБД, то ці драйвери практично завжди поставляються розробниками баз даних. Такі драйвери використовуються для роботи через JDBC із СУБД JavaDB(Derby), MySQL, PostreSQL. Саме їх ми будемо використовувати в лабораторній роботі.

2.2.3 **Основні класи та інтерфейси JDBC API**

Діаграма основних класів та інтерфейсів JDBC наведена на рисунку 2.3.

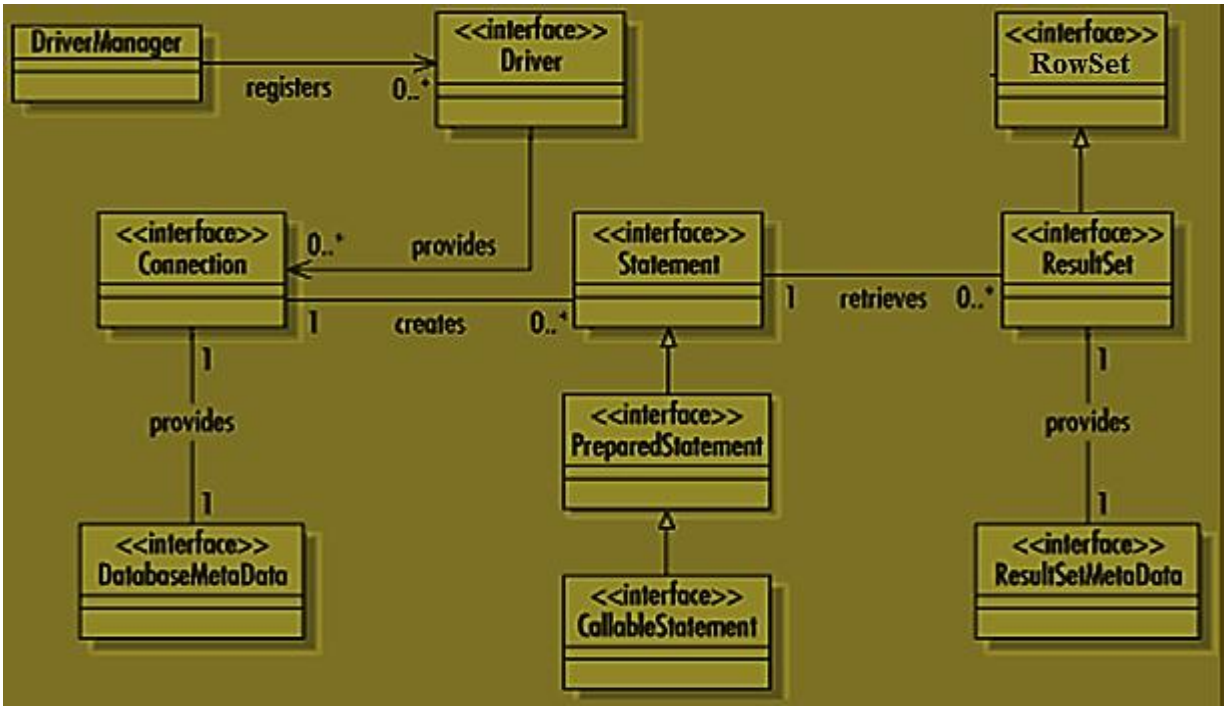

Рисунок 2.3 – Основні класи та інтерфейси JDBC

Як видно з рисунку 2.3, більшість складових JDBC є інтерфейсами. Ці інтерфейси мають бути реалізовані у драйверах, які надають розробники баз даних.

2.2.4 **Встановлення драйвера JDBC**

Для встановлення драйвера JDBC, як правило, потрібно скопіювати jar-файл з класами драйвера на комп'ютер і додати шлях до розташування цього файлу у ClassPath або BuildPath. Крім того, драйвери перших трьох типів вимагають встановлення на стороні клієнта додаткового API.

JDBC драйвери для Derby, MySQL та PostgreSQL додаткового API не потребують.

Драйвери СУБД JavaDB(Derby) поставляються з інсталяцією СУБД. Драйвер для роботи у режимі embeded знаходяться у jar-файлі, що має назву lib.derby.jar. Для роботи з окремим сервером використовується драйвер, що знаходиться у файлі lib.derbyclient.jar.

Драйвер для MySQL входить до складу інсталяційного пакета СУБД, але знаходиться у окремому jar-файлі, що має назву на зразок такої: mysql-connector-java-….jar та міститься у папці Connector.J... .

Драйвер для PostgreSQL до стандартного інсталяційного пакета не входить, але може бути отриманий з Інтернет мережі. Файл може мати назву postgresql-9.4-1201.jdbc41.jar.

Якщо шлях до драйверу вказано правильно, то DriverManager після виклику методу getConnection знайде його і завантажить.

2.2.4.1 Клас java.sql.DriverManager

Клас DriverManager використовується для з'єднання додатку з базою даних шляхом виклику одного із статичних методів getConnection, причому способи формування параметрів деяких з цих методів залежать від СУБД, з якою встановлюється зв'язок. Через параметри цих методів передається інформація про СУБД, що використовується, адреса бази даних та інша службова інформація.

Найбільш універсальним є метод з такою сигнатурою:

```
getConnection(String url, java.util.Properties info)
```

Першим параметром цього методу є рядок url, який надає інформацію про базу даних, її розташування, спосіб підключення та засоби її обслуговування.

У разі використання JDBC-драйвера для підключення до бази даних url має такий вигляд:

```
jdbc:<ім'я драйвера>:[//<хост>[:<порт>/]]<ім'я БД>
```

Наприклад, для підключення до бази даних dbfpo, яка знаходиться за інтернет-адресою dec-fpo.fsay.net і обслуговується СУБД MySQL, URL буде мати такий вигляд:

```
String url = "jdbc:mysql:// dec-fpo.fsay.net:3306/dbfpo";
```

Якщо ж база даних bibl обслуговується сервером PostgreSQL, який встановлено на комп'ютері клієнта, то URL буде мати такий вигляд:

```
String url = "jdbc:postgresql://localhost:5432/bibl"
```

У випадку, коли база myderbydb обслуговується СУБД DERBY у режимі EMBEDED, URL буде мати такий вигляд:

```
String url = "jdbc: derby: myderbydb";
```

Другий параметр методу getConnection, що розглядається, являє собою асоціативний масив типу Properties, який може містити різноманітну інформацію. Кожна окрема властивість заноситься у масив, як пара ключ-значення. Слід мати на увазі, що переліки властивостей для різних СУБД відрізняються, а однакові властивості можуть мати різні ключи.

Наприклад, щоб для СУБД Derby задати ім'я користувача, пароль та вказівку на створення бази у разі її відсутності, пишемо такий код:

```
Properties prop = new Properties();
```

```
prop.setProperty("user", user);
prop.setProperty("password", pass);
prop.setProperty("create", "true");
```

де user та pass – це рядки символів, що задають ім’я та пароль.
Такі самі налаштування для СУБД MySQL будуть виглядати так:

```
Properties prop = new Properties();
prop.setProperty("user", user);
prop.setProperty("password", pass);
prop.setProperty("createDatabaseIfNotExist", "true");
```

А для PostgreSQL ключ для передачі останнього параметру взагалі відсутній.

Після того, як визначені URL та Properties можна встановити з'єднання з базою даних:

```
try {
conn = DriverManager.getConnection(url, prop);
} catch (SQLException e) {
     System.err.println("Проблеми із підключенням до бази " + url);
     e.printStackTrace();
}
```

Виконуючи метод getConnection, DriverManager намагається знайти потрібний драйвер і базу даних. Якщо пошук завершується успішно і з’єднання встановлено, DriverManager повертає об’єкт типу Connection, рисунок 2.3. Зі створенням об’єкту типу Connection починається сеанс зв’язку з базою даних (сесія).

2.2.4.2 Інтерфейс java.sql.Connection

Інтерфейс Connection визначає багато методів, з призначенням яких можна ознайомитися, відкривши вихідний текст інтерфейсу. Тут ми згадаємо тільки про деякі з них. Реалізація цих методів знаходиться у класах драйвера.

Метод getMetaData() повертає об’єкт типу DatabaseMetaData, який містить докладну інформацію про базу даних.

Метод createStatment() повертає об’єкт типу Statement, через який до бази даних передаються SQL оператори.

З об’єктом Connection також пов'язано поняття транзакція. Це умовний пакет інформації, що надсилається до бази даних через один або декілька об'єктів Statement і знаходиться у якомусь буфері. Цей пакет можна або передати базі даних за допомогою методу commit() об'єкту Connection, або відмовитися від передачі за допомогою методу rollback(). За замовчуванням встановлено режим AutoCommit, який передбачає автоматичний виклик методу commit() після завершення виконання кожного SQL оператора.

2.2.4.3 Інтерфейс java.sql.Statement

Головне призначення інтерфейсу Statement – визначити методи, за допомогою яких можна передати SQL оператори до бази даних.

Метод int executeUpdate (String sql) використовується для передачі запитів на створення таблиць та оновлення їх змісту. Цей метод повертає число, що дорівнює кількості вдало виконаних оновлень.

Метод executeQuery(String sql) використовується для отримання результату запиту до БД на вибірку інформації. Результат вибірки повертається у вигляді об’єкту типу ResultSet.

Метод boolean execute (String sql) можна використовувати для передачі будь яких операторів SQL. Метод повертає true, якщо перший результат має тип ResultSet. Для доступу до результатів виконання цього методу можна використовувати методи getResultSet або getUpdateCount.

Наприклад, так може виглядати реалізація запиту на створення таблиці student з полями id та name, якщо об’єкт conn типу Connection вже створено:

```
Statement stmt = null;
String sql = "CREATE TABLE student ("
          + "id INTEGER generated always as identity,"
          + "name VARCHAR(30), "
          + "PRIMARY KEY (id))";
try {
        stmt = conn.createStatement();
        stmt.executeUpdate(sql);
} catch (SQLException e1) {
        e1.printStackTrace();
}
```

Для реалізації запиту до БД на вибірку інформації можна використовувати метод executeQuery (String sql). Наприклад, щоб отримати усі дані з таблиці student, можна написати такий код:

```
sql = "SELECT * FROM student";
ResultSet rs;
try {
     stmt = conn.createStatement();
     rs = stmt.executeQuery(sql);
} catch (SQLException e1) {
        e1.printStackTrace();
}
```

Дані будуть знаходитися у об’єкті ResultSet, що розглядається у 2.1.3.8.

2.2.4.4 Інтерфейс java.sql.PreparedStatement

Об’єкт типу PreparedStatement можна створити за допомогою методу prepareStatement(String), що надсилається об’єкту Connection. В якості параметру до цього методу передається рядок із SQL-оператором.

SQL-оператор компілюється і зберігається в об'єкті PreparedStatement. Цей об'єкт може бути використаний для виконання декілька разів.

Для виконання підготовлених запитів використовуються методи executeUpdate () або executeQuery() без параметрів.

Якщо SQL-оператор містить параметри, то для того, щоб призначити їм

значення, використовують сеттери – методи, що мають префікс set, які вибираються відповідно до типу параметру, що встановлюється. В інших випадках слід використовувати метод setObject. Першим параметром у цих методах є номер параметру, а другим – значення.

Приклад, що наведено нижче, демонструє використання об'єкту PreparedStatement. Змінна con у цьому прикладі представляє активний об'єкт типу Connection:

```
PreparedStatement pstmt = con.prepareStatement(
"UPDATE EMPLOYEES SET SALARY = ? WHERE ID = ?");
pstmt.setBigDecimal(1, 153833.00);
pstmt.setInt(2, 110592);
pstmt.executeUpdate();
```

2.2.4.5 Пакетне виконання SQL запитів

Іноді потрібно виконати кілька SQL запитів один за іншим. Для таких випадків прийнято використовувати механізм пакетного (batch) виконання SQL запитів. Пакет запитів створюється і наповнюється методом addBatch(String) об'єкту Statment. Для виконання пакета використовується метол executeBatch, який повертає масив цілих чисел, де кожний елемент є результатом виконання окремого запиту. Нижче наведено приклад використання пакета.

```
st = con.createStatement();
st.addBatch("INSERT INTO CUSTOMER VALUES (10, 'John', 1000)");
st.addBatch("UPDATE CUSTOMER SET SALARY = 250 WHERE ID = 1");
st.addBatch("UPDATE CUSTOMER SET SALARY = 350 WHERE ID = 2");
int[] results = st.executeBatch();
```

Щоб визначити, чи підтримує драйвер JDBC пакетні оновлення, можна скористатися методом DatabaseMetaData.supportsBatchUpdates ().

2.2.4.6 Інтерфейс java.sql.CallableStatement

Цей інтерфейс надає можливість використання збережених процедур і функцій, які виконуються на стороні сервера бази даних. Об'єкт типу CallableStatment можна отримати через об'єкт Connection за допомогою методу collableStatment(String). В якості параметру до цього методу передається рядок символів із зверненням до процедури або функції:

```
CallableStatement cst = con.callableStatement(
   "{? = call getSumSalary(?)}"
   );
cst.setInt(2, 10);
cst.registerOutParameter(1, java.sql.Types.DOUBLE);
cst.execute();
System.out.println("Sum salary for id < 10 = " + cst.getDouble(1));
```

2.2.4.7 Інтерфейс java.sql.ResultSet

Інтерфейс визначає перелік методів, що дозволяють обробляти результати запиту до БД. Результат запиту, що представлений об'єктом типу ResultSet можна уявляти собі як таблицю, що містить один, або декілька рядків, і кожний з цих рядків може складатися з одного, або декількох стовпчиків.

Початкова позиція об'єкту ResultSet знаходиться перед першим рядком таблиці. Тобто для того, щоб прочитати дані першого рядка потрібно викликати метод next().

Метод boolean next () позиціонує об'єкт ResultSet на наступний рядок, і повертає false, якщо рядки закінчилися. Це дозволяє використовувати цикл while для обробки результатів запиту, наприклад, while(resultSet.next()){…}.

Як приклад, розглянемо виведення на консоль результатів запиту, що наводився як приклад у пункті 2.1.3.4:

```
try {
     while (rs.next())
          System.out.println(rs.getInt("id") + "  " +
rs.getString(2));
} catch (SQLException e) {
     e.printStackTrace();
}
```

Як свідчить приклад, до елементів ResultSet можна звертатися як за порядковим номером, так і використовуючи ім'я поля.

За замовчуванням позицію об'єкту ResultSet можна переміщувати тільки вперед і використовувати результати тільки раз, і не можна змінювати.

Методи доступу до елементів рядка (полів таблиці) мають наступний формат: Type getType (int | String). Параметром цих методів може бути або номер стовпчика або його ім'я. Можливості перетворення типів даних визначаються JDBC специфікацією.

Використання номеру стопця є більш ефективним. Стовпці нумеруються з одиниці. Колонки рядка слід читати зліва направо і тільки один раз.

Використання різних методів доступу до полів дозволяє отримувати дані різного типу з одного і того ж поля. Наприклад, якщо перше поле має тип Date, то для того, щоб отримати його значення у вигляді об'єкту типу Date, слід використовувати метод getDate(1). Якщо ж результат потрібен у вигляді рядка символів, необхідно використовувати метод getString(1).

Можливо також отримати об'єкт ResultSet, який дозволяє скролінг та зміни. Наступний приклад з використанням об'єкту con типу Connection демонструє, як це зробити.

```
Statement stmt = con.createStatement(
ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
  ResultSet rs = stmt.executeQuery("SELECT a, b FROM TABLE");
  // rs will be scrollable, will not show changes made by others,
  // and will be updatable
```

Якщо об’єкт ResultSet дозволяє скролінг, можна використовувати метод previous(), що позиціонує об’єкт ResultSet у попередню позицію, або метод beforeFirst(), що повертає об’єкт ResultSet у початковий стан. Методи first() та last() забезпечують перехід до першої та останньої позицій. Метод absolute(int) дозволяють перейти до рядка із вказаним номером.

Якщо об’єкт ResultSet дозволяє оновлення, то можна використовувати методи оновлення, які мають префікс update. Першим параметром цих методів може бути або номер стовпчика або його ім'я, а другим нове значення поля.

Для того, щоб зафіксувати відповідні зміни у джерелі даних слід викликати метод updateRow(). А для відмови від змін можна викликати метод cancelRowUpdates().

```
rs.absolute(5); // moves the cursor to the fifth row of rs
rs.updateString("NAME", "Петренко"); // updates the
   // NAME column of row 5 to be Петренко
rs.updateRow(); // updates the row in the data source
```

У режимі оновлення можна також додавати рядки. Додатковий рядок створюється методом moveToInsertRow(). Дані до колонок заносяться методами update. Для збереження оновлення використовується метод insertRow().

Нижче наведено приклад додавання рядка.

```
rs.moveToInsertRow(); // moves cursor to the insert row
rs.updateString(1, "Хрущ"); // updates the
// first column of the insert row to be Хрущ
rs.updateDouble(2,4.5); // updates the second column to be 4.5
rs.updateBoolean(3, true); // updates the third column to true
rs.insertRow();
rs.moveToCurrentRow();
```

Для вилучення рядка кожна скористатися методом deleteRow().

Але підтримка постійного з’єднання з базою даних може призвести до затримки обробки запитів від інших користувачів.

Більш докладну інформацію можна знайти к коментарях до класу та його методів.

2.2.4.8 Інтерфейс javax.sql.RowSet

Інтерфейс RowSet є спадкоємцем інтерфейсу ResultSet, тому функціонально ці інтерфейс и мають багато спільного.

Об’єкти RowSet відрізняються перш за все тим, що їх потрібно створювати, викликаючи конструктри класів, що реалізують інтерфейс RowSet, або його спадкоємців.

Друга особливість полягає у тому, що об’єкти RowSet завжди мають здатність до скролінгу і змін.

Окрім того, ці об’єкти здатні формувати події, пов’язані із зміною рядка, зміною значення, тощо. Тож ці об’єкти розглядаються як JavaBeans компоненти.

Об’єкти RowSet можна створювати як приєднаними до бази даних так і від’єднаними від неї.

Від’єднаний RowSet можна створити, наприклад, за допомогою конструктора класу com.sun.rowset.CachedRowSetImpl. Табличку ResultSet можна передати цьому об’єкту через метод populate().

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM whatever");
RowSet rws = = new CachedRowSetImpl();
((CachedRowSetImpl)rws).populate(rs);
```

Приєднаний RowSet можна створити за допомогою конструктора класу com.sun.rowset.JdbcRowSetImpl з параметром типу ResultSet . Але у цьому випадку об’єкт ResultSet має бути створено з налаштуваннями, що дозволяють скролінг та оновлення.

Ось приклад створення такого об’єкту:

```
Statement stm = conn.createStatement(
     ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stm.executeQuery(sql);
RowSet rws = null;
rws = new JdbcRowSetImpl(rs);
```

Можливі і прямі шляхи створення об’єкту RowSet шляхом передачі йому інформації про базу даних та запиту SQL. Відповідні методи можна знайти у вихідному тексті інтерфейсу та класів.

2.2.4.9 Закриття об'єктів

Всі три класи Connection, Statement і ResultSet мають метод close (). Не всі реалізації JDBC вимагають закривати об'єкти перед повторним використанням, але є реалізації, що вимагають закривати об'єкти, тим самим звільняючи ресурси сервера БД. Тому прийнято закривати об'єкти після закінчення роботи з ними. Важливо пам'ятати, що якщо закривається об'єкт Connection, то автоматично закриваються всі пов'язані з ним об'єкти Statement. Аналогічно при закритті об'єкта Statement закриваються всі пов'язані з ним об'єкти ResultSet. При закритті з'єднання все незбережені дані останньої відкритої транзакції будуть втрачені.

2.3 Порядок виконання роботи

2.3.1 **Створення проєкту**

Створімо новий Java-проєкт. В межах цього проєкту ми будемо виконувати лабораторну роботу і завдання для самостійної роботи, тому одразу додаймо до проєкту чотири пакети – controller, query, test, view.

2.3.2 **Підготовка до роботи з базою даних**

В лабораторній роботі ми будемо працювати із СУБД Derby. Ця СУБД є складовою частина JDK8 і може використовуватися як у серверному варіанті, так і у варіанті для одного користувача (embedded варіант). Файл derby.jar знаходиться у папці db.lib JDK8. Якщо нема JDK, то Derbi можна скачати з

інтернет або встановити відповідний плагін в eclipse. Можна також файл derby.jar взяти на сторінці курсу ресурсу мудл.

Щоб мати можливість працювати з базою даних, підключіть derby.jar до Build Path проєкту.

Слід такж враховувати, що різні версії драйверів можуть бути несумісні. Тому для роботи з базою бажано використовувати той самий драйвер, що використовувався для створення цієї бази.

2.3.3 **Створення класу DbConnector**

Для того, щоб спростити процес з'єднання з базою даних, доцільно створити клас, що буде реалізовувати з'єднання з нашою базою даних та надавати доступ до об'єкту Connection.

Створімо у пакеті controller клас DbConnector, скориставшись текстом, що наведено нижче. Майте на увазі, що внаслідок копіювання тексту деякі символи, наприклад лапки, можуть змінюватися. Будьте кважні з імпортом.

Цей клас розрахований на роботу із СУБД Derby, а сама база буде називатися dbByv і зберігатися в папці проєкту.

У своїх проєктах для назви бази обов'язково використовуйте своє прізвище, щоб не було плутанини під час перевірки. Для властивості user також використовуйте своє прізвище.

```
public class DbConnector {
   private static String dbFullName = "dbByv";
   private static String url = "jdbc:derby:" + dbFullName;;
   private static Properties prop = new Properties();
   static private  Connection conn;
   static{
      prop.setProperty("user", "byvoino");
      prop.setProperty("password","12345");
      prop.setProperty("create", "true");
   }
   public static Connection getConnection(){
      if(conn == null)
         try {
            //This string need when class is using by server
            //If driver was download nothing to do
            Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
         } catch (ClassNotFoundException e1) {}
         try {
            conn = DriverManager.getConnection(url, prop);
         } catch (SQLException e) {
            JOptionPane.showMessageDialog(null, e.getMessage() +
                  " or May by connection use another project?");
            e.printStackTrace();
         }
      return conn;
   }
   public static String getDbFullName() {
```

```
        return dbFullName;
    }
    public static void setDbFullName(String dbFullName) {
        DbConnector.url = "jdbc:derby:" + dbFullName;
    }
    public static void setPropertyUser(String str) {
        prop.setProperty("user", str);
    }
    public static void setPropertyPassword(String str) {
        prop.setProperty("password", str);
    }
    public static void setPropertyCreate(String str) {
        prop.setProperty("create", str);
    }
}
```

2.3.4 **Створення бази даних**

Перед виконанням цього етапу слід зауважити, що база даних – це не таблиці, а інструмент для збереження таблиць та маніпуляцій з ними. Таблиці до бази ми будемо додавати пізніше. Нагадуємо, що у назві бази має бути присутнім прізвище студента.

Для створення бази виконайте такі дії:

– у пакеті test створюємо клас CreateDB з методом main;

– у методі main написати код, який за допомогою класу DbConnector забезпечить створення бази даних та з'єднання з нею у вигляді об'єкта типу Connection. Після створення бази та встановлення з'єднання на консоль буде виведено інформацію про з'єднання. Текст методу main наведено нижче. Будьте уважні з імпортом. Інтерфейс Connection потрібен із пакета java.sql.

```
public static void main(String[] args) {
    Connection conn = DbConnector.getConnection();
    System.out.println(conn);
}
```

– запустити на виконання створений клас і перевірити, чи було створено папку з базою даних. База із заданим простим ім'ям має бути розташована у папці проєкту. Для того, щоб вона з'явилася у панелі Package Explorer, треба викликати функцію Refresh контекстного меню цієї панелі.

2.3.5 **Створення таблиці бази даних**

Увага! У лабораторній роботі Ви маєте створювати таблиці, що відповідають вибраному варіанту.

Ми в якості прикладу будемо використовувати базу даних, схему якої наведено на рисунку 2.4. Для побудови схеми використовувався інтернет ресурс https://dbdiagram.io/d [1]. Можна також скористатися ресурсом draw.io [2].

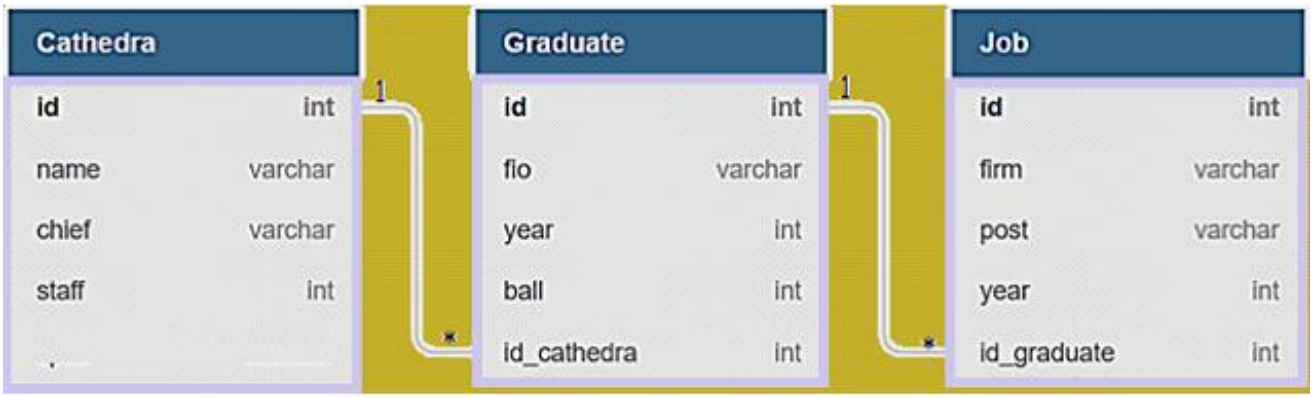

Рисунок 2.4 – Схема бази даних

2.3.5.1 SQL запит для створення першої таблиці бази даних

SQL запит – це рядок символів, і з таблицею бази даних може бути пов'язано декілька таких запитів. Тому для зберігання цих запитів можна створити клас, який буде повертати тексти запитів як відповідь на виклик статичних методів. Тобто цей клас буде виступати як бібліотека запитів до таблиці.

Створімо для нашого прикладу в пакеті query клас з назвою QueryCathedra. Ця назва свідчить про те, що у класі зберігаються запити, які пов'язані з таблицею Cathedra. У проєктах студентів класи слід називати у відповідності з назвами таблиць вибраного варіанту.

Нижче, для прикладу, наведено текст цього класу, в якому запит на створення таблиці повертає статичний метод queryCreate():

```
public class QueryCathedra {

   public static String queryCreate() {
      String sql = "create table Cathedra ("
            + "ID integer generated always as identity,"
            + "NAME varchar(30) default '' not null, "
            + "CHIEF varchar(30) default '', "
            + "STAFF integer default 0, "
            + "primary key (ID))";
      return sql;
   }
}
```

Код класу можна скопіювати, але після цього відредагувати відповідно до свого варіанту. Поле id має бути обов'язково.

Зверніть увагу, рядки символів, які входять до рядка query, обмежуються одинарними лапками.

2.3.5.2 Клас CreateTable1

Створімо у пакеті test клас CreateTable1 з методом main, який забезпечить створення таблиці бази даних. Нижче наведено код методу, що створює таблицю Cathedra:

```
public static void main(String[] args) {
  try {
     Connection conn = DbConnector.getConnection();
     Statement st = conn.createStatement();
     String query = QueryCathedra.queryCreate();
     st.executeUpdate(query);
  } catch (SQLException e) {
     e.printStackTrace();
  }
}
```

Будьте уважні з імпортом. Інтерфейс Statement потрібен із пакету java.sql.

Запустіть на виконання створений код і, якщо нема помилок, спробуйте виконати його повторно. Як наслідок має виникнути виключна ситуація і з'явитися повідомлення, що таблиця вже існує.

2.3.6 **Додавання даних до таблиці бази даних**

Для додавання записів до нашої таблиці у пакеті tests створюємо клас AddToTable1 з методом main. Нижче наведено приклад такого методу, який потрібно переробити з урахуванням особливостей свого варіанту:

```
public static void main(String[] args) {
  Connection conn = DbConnector.getConnection();
  String query = "insert into CATHEDRA(NAME, CHIEF, STAFF)"
       + " values(?,?,?)";
  try {
     PreparedStatement pst = conn.prepareStatement(query);
     pst.setString(1,"ICS");
     pst.setString(2,"Basilevich");
     pst.setInt(3,10);
     pst.executeUpdate();

     pst.setString(1,"Language");
     pst.setString(2, "Duo");
     pst.setInt(3,7);
     pst.executeUpdate();
  } catch (SQLException e) {
     e.printStackTrace();
  }
}
```

Після виконання коду таблиця бази даних має оновитися, а щоб переглянути вміст таблиці переходимо до наступного пункту.

2.3.7 **Виведення вмісту таблиці на консоль**

Для того, щоб переглянути вміст таблиці у пакеті test створюємо клас PrintTable1 з методом main, що схожий на наступний, але з урахуванням особливостей свого варіанта:

```
public static void main(String[] args) {
```

```
        Connection conn = DbConnector.getConnection();
        try {
           Statement st = conn.createStatement();
           String query = "select * from Cathedra";
           ResultSet rs = st.executeQuery(query);
           while (rs.next())
              System.out.println(rs.getString(1) + "   "
                    + rs.getString("NAME") + "   "
                    + rs.getString("CHIEF")+ "   "
                    + rs.getInt("STAFF"));
        } catch (SQLException e) {
           e.printStackTrace();
        }
     }
```

Після виконання коду на консолі має з'явитися вміст таблиці. Його слід зафіксувати для звіту.

2.3.8 **Виконання пакетного запиту до бази даних**

Пакетний запит зручно робити, наприклад, у випадку, коли до таблиці потрібно занести багато даних. Для прискорення цього процесу можна створити діалог, через який будуть заноситися дані до пакета. Діалог буде повертати дані, що відповідають полям таблиці бази даних, а на підставі цих даних можна створювати запити на поповнення бази і формувати з них пакет. Після завершення роботи з діалогом пакет запитів надсилається до бази даних.

2.3.8.1 Діалог для реалізації пакетного запиту

В якості такого діалогу можна використовувати діалог, що було створено на попередній лабораторній роботі. В нашому прикладі це клас DlgCathedra. Його треба скопіювати у проєкт до папки view.

2.3.8.2 Клас для реалізації пакетного запиту

Створімо у пакеті test клас AddBatchFromDlg з методом main.

Діалог повертає дані у вигляді карти. Тому рядок для запиту на додавання будемо формувати за допомогою статичного методу format класу String.

Нижче наведено реалізацію методу main для пакетного заповнення таблиці Cathedra:

```
     public static void main(String[] args){
        try {
           Connection conn = DbConnector.getConnection();
           String query ="insert into CATHEDRA(NAME, CHIEF, STAFF) "
                 + "values(?,?,?)";
           PreparedStatement pst = conn.prepareStatement(query );
           DlgCathedra dlg = new DlgCathedra();
           while(true){
              dlg.clear();
              dlg.setVisible(true);
```

```
            Map<String, Object> map = dlg.getMap();
            if(map == null) break;
            pst.setString(1,(String) map.get("name"));
            pst.setString(2,(String) map.get("chief"));
            pst.setInt(3,(int) map.get("staff"));
            pst.addBatch();
         }
         dlg.dispose();
         pst.executeBatch();
      } catch (Exception e) {
         e.printStackTrace();
      }
      PrintTable1.main(null);
   }
```

2.3.9 **Виведення таблиці БД у візуальний компонент**

Для відображення таблиць бази даних створімо у пакеті view клас DbTableView, який буде успадковувати клас javax.swing.JTable. За рахунок успадкування ми розширимо можливості стандартного класу. Об'єкти цього класу зможуть приймати об'єкти типу List<Map<String, Object>>. Це список з картами, які відповідають рядкам таблиці бази даних і повертати карту Map<String, Object>, що відповідає вибраному рядку.

Списки з картами, які відповідають рядкам таблиці бази даних, ми будемо розглядати, як модель бази даних.

Код класу DbTableView наведено нижче:

```
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

import javax.swing.JOptionPane;
import javax.swing.JTable;
import javax.swing.table.DefaultTableModel;

public class DbTableView extends JTable {

   public  void setDbTableModel(List<Map<String, Object>> mapList) {
      Object[][] values = null;
      Object[] header = new Object[]{"EmptyTable"};
      if(mapList.size() > 0) {
         header = mapList.get(0).keySet().toArray(header);
         values = new Object[mapList.size()][header.length];
         int i = 0;
         for (Map<String, Object> map : mapList)
            values[i++] = map.values().toArray();
      }
      setModel(new DefaultTableModel(values, header));
   }
```

```
    public Map<String, Object> getSelectedRowMap() {
        int row = getSelectedRow();
        if (row == -1) {
            JOptionPane.showMessageDialog(this, "Row is not selected");
            return null;
        }
        Map<String, Object> map = new LinkedHashMap<>();
        for (int i = 0; i < getModel().getColumnCount(); i++) {
            String key = getModel().getColumnName(i);
            Object value = getModel().getValueAt(row, i);
            map.put(key, value);
        }
        return map;
    }
}
```

2.3.9.1 Метод для формування моделі таблиці БД на основі об’єкта типу ResultSet

JDBC передбачає повернення результатів запитів до бази даних у вигляді об’єктів типу ResultSet, а клас DbTableView, який ми створили для відображення таблиці, приймає модель таблиці бази даних у вигляді списоку карт. Тому нам потрібен метод, для перетворення об’єкта ResultSet у список карт.

За операції з базою даних у нашому проєкті буде відповідати клас Controller, тому зараз доцільно створити такий клас у пакеті controller і в ньому реалізувати потрібний нам метод.

```
public class Controller {
    public static List<Map<String,Object>> rsToMapList(ResultSet rs) {
        List<Map<String, Object>> list = new ArrayList<>();
        try {
            ResultSetMetaData metadata = rs.getMetaData();
            int columnCount = metadata.getColumnCount();
            while(rs.next()) {
                Map<String, Object> map = new LinkedHashMap<>();
                for (int i = 1; i <= columnCount; i++) {
                    String columnName = metadata.getColumnName(i);
                    map.put(columnName.toLowerCase(), rs.getObject(i));
                }
                list.add(map);
            }
            rs.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return list;
    }
}
```

2.3.9.2 Створення візуального компонента з відображенням таблиці бази даних

В якості такого компонента у пакеті view створімо діалог, який дозволить відображати таблицю бази даних та вибирати потрібний запис із цієї таблиці.

Ми будемо використовувати цей діалог і в наступних роботах.

Модедь таблиці, що слід відображати, будемо передавати через конструктор як об’єкт типу List<Map<String, Object>>.

Через метод getMap() діалог буде повертати карту, у якої ключами будуть назви колонок таблиці, а значеннями – відповідні значення вибраного рядка.

Діалог створімо на основі компоненту JDialog. Назвемо клас для нього DlgSelect.

Вигляд діалогу у режимі дизайну наведено на рисунку 2.5.

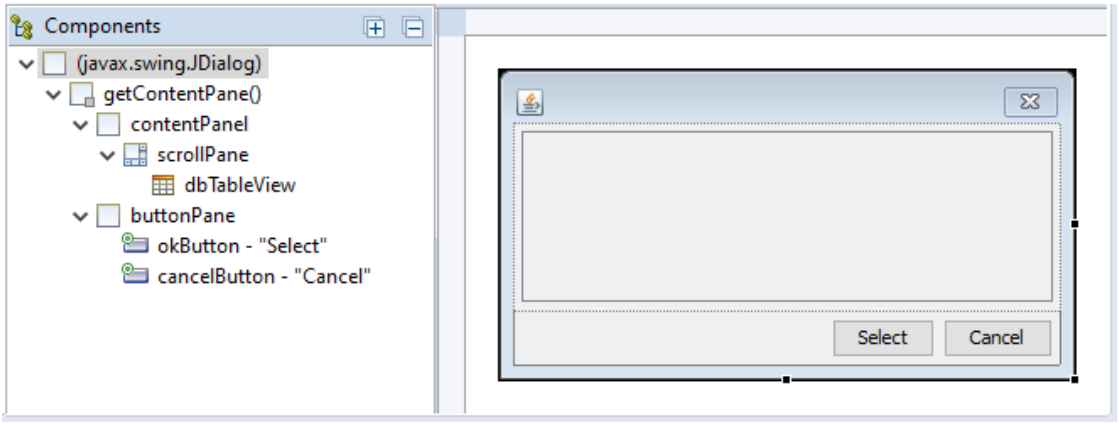

Рисунок 2.5 – Вікно дизайну діалогу для вибору рядка таблиці БД

Для панелі contentPane, що використовується для розміщення компонентів налаштуємо менеджер компоновки CardLayout. У цьому випадку таблиця буде займати всю площу панелі.

Після цього на панелі contentPane розташуємо ScrollPane а на цій панелі компонент класу DbTableView, який ми створили для відображення таблиці.

ScrolPane потрібна, перш за все, для того, щоб у її верхній частині відображався заголовок таблиці.

У режимі дизайну доступ до цього компонента можна отримати через кнопку Choose Components, що знаходиться у верхній частині палетти компонентвів. Компонент треба зробити доступним через функцію ExposeComponent та назвати його dbTtableview.

Далі налаштувати значення true для властивості modal діалогу і створити обробники подій Action Performed для кнопок Select і Cancel з викликом методів on Select() та onCancel().

Далі запрограмуємо діалог.

Перш за все визначимо у діалозі поле для карти, яка буде зберігати результат вибору:

```
private Map<String, Object> map;
```

Далі створімо конструктор діалогу з параметром, через який буде

передаватися модель таблиці бази даних, яку треба відобразити.

```
public DlgSelect(List<Map<String, Object>> mapList) {
  this();
  dbTableView.setDbTableModel(mapList);
}
```

Після виклику конструктора модель одразу буде відображено в таблиці.

Тепер реалізуймо методи, що будуть викликатися після натискання на кнопки:

```
protected void onSelect() {
     map = dbTableView.getSelectedRowMap();
     setVisible(false);
}


protected void onCancel(ActionEvent e) {
  map = null;
  setVisible(false);
}
```

Після цього залишається написати метод, що повертає створену карту:

```
public Map<String, Object> getMap() {
     return map;
}
```

2.3.9.3 Тестування візуального відображення таблиці БД

Для тестування створімо у пакеті test клас TestSelectDialog з методом main. Для таблиці Cathedra код методу main буде виглядати так:

```
public static void main(String[] args) {
  Connection conn = DbConnector.getConnection();
  try {
     Statement st = conn.createStatement();
     String sql = "select * from CATHEDRA";
     ResultSet rs = st.executeQuery(sql);
     List<Map<String, Object>> list =
          Controller.rsToMapList(rs);
     DlgSelect ds = new DlgSelect(list);
     ds.setVisible(true);
     System.out.println(ds.getMap());
     ds.dispose();
  } catch (SQLException e) {
     e.printStackTrace();
  }
}
```

Далі можна тестувати клас. Скриншот результату зафіксувати для звіту.

2.4 Завдання для самостійної роботи

У ході самостійної роботи необхідно створити java застосунок, що забезпечує візуалізацію роботи з базою даних, реалізацію CRUD операцій з таблицями, та виконання наперед визначених та довільного запиту до бази. У даній лабораторній роботі ми обмежимося роботою з першими двома таблицями обраного варіанту.

Частину роботи з розробки такого застосунку ми вже зробили під час виконання лабораторної роботи, тому новий проєкт створювати не треба, натомість будемо продовжувати реалізацію проєкту лабораторної роботи.

2.4.1 **Архитетура застосунку**

Як вже наголошувалось, кожен студент може створювати застосунок на свій розсуд. А в методичних вказівках ми наводимо в якості прикладу один із можливих варіантів.

Для того, щоб студенти мали повне уявлення про варіант застосунку, що ми пропонуємо, розглянемо його архітектуру, рисунок 2.6.

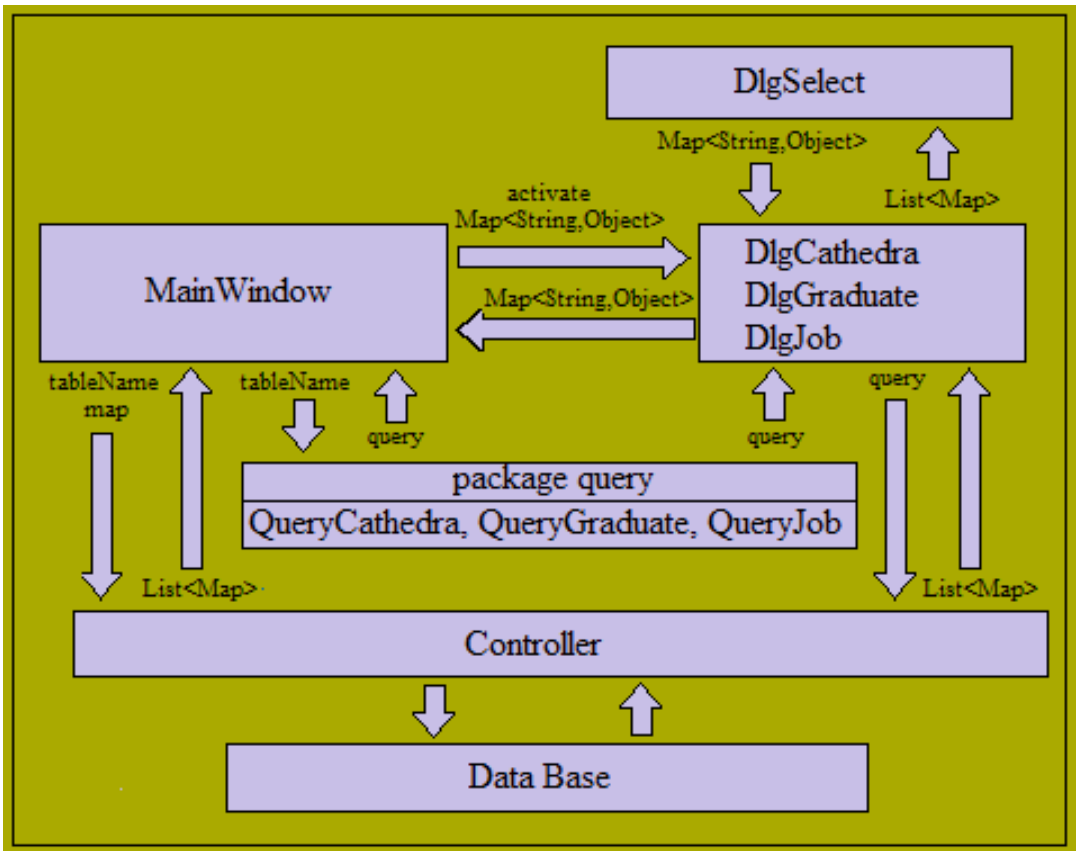

Рисунок 2.6 – Архітектура застосунку, що пропонується як приклад

До застосунку входять такі складові:

– головне вікно, де будуть відображатися результати запитів до бази даних;
– діалоги, через які користувач передає інформацію до програми;
– контролер, що відповідає за взаємодію з базою даних;

– пакет query з класами, що містять тексти стандартних запитів до відповідних таблиць бази даних.

Обмін інформацією між складовими частинами затосунку відбувається за допомогою карт, або списків таких карт. Це зроблено з метою універсалізації візуальної частини, що складається з головного вікна та діалогів. Ці елементи застосунка ми зможемо використовувати без змін і лабораторній роботі 4.

2.5 Рекомендації з виконання завдання для самостійної роботи

2.5.1 Реалізація запитів до бази

2.5.1.1 Стандартні запити до таблиць бази

Один із таких звпитів ми вже створили у класі query.QueryCathedra. Решту методів для першої таблиці студенти мають написати самі.

В якості прикладу тут наводено текст класу query.QueryGraduate для другої таблиці.

Цей клас можна використовувати як зразок і для першої таблиці. Єдина відмінність полягає в тім, що останній метод (вибір переліку випувкників для заданої кафедри) для першої таблиці не має сенсу.

```
public class QueryGraduate  {

   //Запити на створення таблиці БД
   public static String queryCreate() {
      String sql ="create table GRADUATE ("
            + "ID int generated always as identity,"
            + "FIO varchar(30) default '' not null,"
            + "FINISH integer,"
            + "BALL int,"
            + "IDCATHEDRA int constraint FK_GRAD_CTH "
            + "references CATHEDRA on delete cascade, "
            + "primary key (ID))";
      return sql;
   }

   //Запит на отримання вмісту таблиці БД
   public static String queryGetAll() {
      return "select GRADUATE.ID, GRADUATE.FIO, GRADUATE.FINISH,"
            + " GRADUATE.BALL, CATHEDRA.NAME AS CATHEDRA, "
            + " CATHEDRA.ID AS IDCATHEDRA"
            + " from GRADUATE, CATHEDRA "
            + "WHERE CATHEDRA.ID=GRADUATE.IDCATHEDRA "
            + " ORDER BY CATHEDRA, GRADUATE.FIO";
   }

   //Запит на додавння
   public static String queryAdd(Map<String, Object> map) {
      return String.format("insert into GRADUATE"
```

```
            + "(FIO, FINISH, BALL, IDCATHEDRA) "
            + "values('%s', %d, %d, %d)",
            map.get("fio"), map.get("finish"),
            map.get("ball"), map.get("idcathedra"));
    }

    //Запит на редагування
    public static String queryEdit(Map<String, Object> map) {
       return String.format("update GRADUATE"
            + " set  FIO = '%s', FINISH = %d, BALL = %d, "
            + "IDCATHEDRA = %d where ID = %d",
            map.get("fio"), map.get("finish"), map.get("ball"),
            map.get("idcathedra"), map.get("id"));
    }

    //Запит на видалення
    public static String queryDelById(int id) {
       return "DELETE FROM GRADUATE WHERE ID = " + id;
    }

    //Запит на отримання переліку випускників для заданої кафедри
    public static String queryGetForCathedra(int idCathedra) {
       return String.format("select GRADUATE.ID, GRADUATE.NAME, "
            + "GRADUATE.FINISH, GRADUATE.BALL from GRADUATE "
            + "WHERE GRADUATE.IDCATHEDRA = %d "
            + " ORDER BY GRADUATE.NAME", idCathedra);
    }
}
```

2.5.1.2 Наперед визначені запити до бази

Методи, які повертають тексти наперед визначених запитів до бази, можна розташувати в класі Query пакета query. Такими можуть бути запити, що виконуються у відповідь на натискання кнопок Query1 та Query2 візуальної частини.

Як приклад розглянемо запит на отримання переліку випускників деякої кафедри якогось року:

```
public static String query1(int finish, int idCathedra) {
   return String.format("select fio, ball from graduate"
        + " where finish = %d and idcathedra = %d ",
        finish, idCathedra);
}
```

Рік закінчення та id кафедри передаються в метод як параметри.

Студенти мають написати запити до своїх таблиць.

2.5.2 **Розробка контролера**

Цей клас являє собою сполучну ланку між базою даних і візуальною частиною проєкту. Основне завдання цього класу - отримувати тексти запитів до

бази даних, передавати їх базі і повертати результати цих запитів.

Ми вже створили цей клас у пакеті controller і реалізували там метод перетворення об’єкта ResultSet у карту.

Тепер реалізуймо інші методи.

2.5.2.1 Методи загального призначення

Метод TableExist використовується для перевірки наявності таблиці у базі даних по її назві. Можливі різні варіанти виконання такої перевірки, але спеціального метода у JDBC нема. Тому ми скористаємось методом getTables для об’єктів типу DatabaseMetaData. Цей метод повертає ResultSet властивостей таблиць з назвами, що задовольняють шаблону, який передається третім параметром.

Ми в якості цього шаблону передамо ім’я таблиці.

Якщо таблиця існує, то перший виклик метода next у будь-якому випадку поверне true. Якщо таблиці нема, то отримаємо виняткову ситуацію. Ось цим ми і скористаємось.

```
public static boolean tableExist(String tableName) {
  try {
     Connection conn = DbConnector.getConnection();
     DatabaseMetaData md = conn.getMetaData();
     String name = tableName.toUpperCase();
     return md.getTables(null, null, name, null).next();
  } catch (SQLException e) {
     e.printStackTrace();
     return false;
  }
}
```

Далі створімо метод для обробки довільного запиту на вибірку даних з бази, який буде приймати рядок символів з SQL запитом і повертати результат у вигляді списку карт.

У першому рядку методу додано виведення тексту запиту на консоль. Це допоможе пошуку помилок у запитах.

```
public static List<Map<String, Object>> executeQuery(String query){
  System.out.println(query);
  List<Map<String, Object>> list = null;
  try {
     Connection conn = DbConnector.getConnection();
     Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery(query);
     list = rsToMapList(rs);
  } catch (SQLException e) {
     e.printStackTrace();
  }
  return list;
}
```

Створімо також метод на оновлення вмісту таблиці бази даних. який буде

приймати рядок символів з SQL запитом на оновлення і повертати число, що відповідає кількості оновлень.

```
public static int executeUpdate(String query) {
   System.out.println(query);
   try {
      Connection conn = DbConnector.getConnection();
      Statement stmt = conn.createStatement();
      return stmt.executeUpdate(query);
   } catch (SQLException e) {
      e.printStackTrace();
      return 0;
   }
}
```

2.5.2.2 Методи для реалізації CRUD запитів до таблиць БД

Ці методи будуть приймати назву таблиці в якості параметра, а в разі необхідності і карту з даними. По назві таблиці метод може визначити клас із пакета query, вибрати потрібний запит із цього класу і викликати метод executeUpdate контролера.

У методах використовується технологія рефлексії для виклику методів класів пакета query.

```
public static int createTable(String tableName) {
   try {
      String queryClass = "query.Query" + tableName;
      Class<?> clz = Class.forName(queryClass);
      Method mtd = clz.getMethod("queryCreate");
      String sql = (String) mtd.invoke(null);
      int n = Controller.executeUpdate(sql);
      return n;
   } catch (Exception e1) {
      e1.printStackTrace();
      return 0;
   }
}

public static void add(String tableName, Map<String, Object> map) {
   try {
      String queryClass = "query.Query" + tableName;
      Class<?> clz = Class.forName(queryClass);
      //Отримуємо текст sql апиту на додавання
      Method mtd = clz.getMethod("queryAdd",Map.class);
      String sql = (String) mtd.invoke(null,map);
      //Передаємо запит контролеру
      Controller.executeUpdate(sql);
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

```
public static void edit(String tableName, Map<String, Object> map) {
   try {
      String queryClass = "query.Query" + tableName;
      Class<?> clz = Class.forName(queryClass);
      Method mtd = clz.getMethod("queryEdit", Map.class);
      String sql = (String) mtd.invoke(null,map);
      //Передаємо запит контролеру
      Controller.executeUpdate(sql);
   } catch (Exception e) {
      e.printStackTrace();
   }
}

public static void delete(String tableName, Map<String, Object> map)
{
   try {
      String queryClass =  "query.Query" + tableName;
      Class<?> clz = Class.forName(queryClass);
      //Формуємо запит на видалення запису
      Method mtd = clz.getMethod("queryDelById", int.class);
      int id = (Integer)map.get("id");
      String sql = (String) mtd.invoke(null,id);
      //Передаємо запит контролеру
      Controller.executeUpdate(sql);
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

2.5.3 **Діалоги для введення та редагування інформації в базі даних**

Діалог для першої таблиці було створено раніше і він вже використовувався в лабораторній роботі. Але він вимагає доопрацювання. Справа в тім, що для інших таблиць теж потрібні аналогічні діалоги, і вони мають бути одного типу, щоб можна було використовувати переваги поліморфізму. Тому треба визначитися із загальним інтерфейсом для цих діалогів.

2.5.3.1 Інтерфейс для діалогів з інформацією про рядки таблиць

Створімо інтерфейс IDlg, який розташуємо в пакеті vew.

```
public interface IDlg {
   public Map<String, Object> getMap();
   public void clear();
   public void setVisible(boolean b);
   public void dispose();
}
```

Після створення цього інтерфейсу його слід заявити у класі даалогу для першої таблиці.

2.5.3.2 Розробка діалогу для таблиці другого рівня

Особливсть діалогів наступних рівнів полягає в тім, що вони мають надати можливість вибору посилання на об'єкти попереднього рівня. Так, наприклад, для випускника слід визначити, яку кафедру він закінчив.

В якості приклада розглянемо діалог для введення та редагування інформації про випускника, рисунку 2.7.

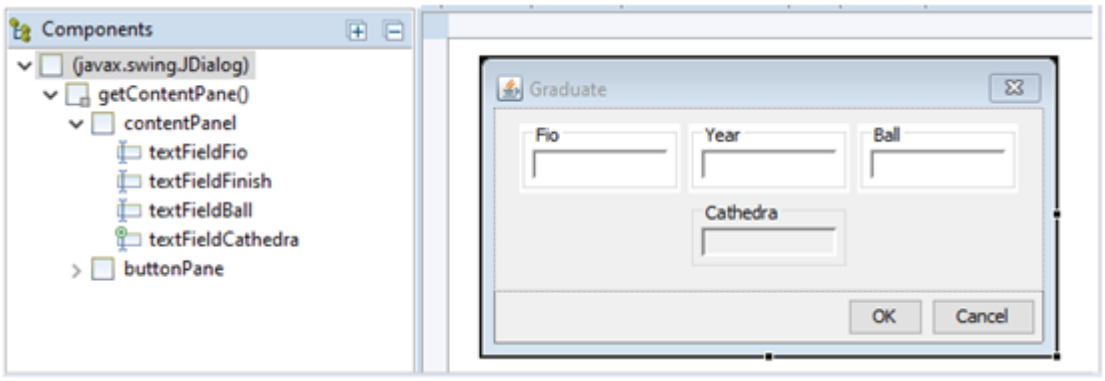

Рисунок 2.7 – Діалог введення та редагування інформації про випускника

Не забувайте, що діалог має реалізовувати інтерфейс IDlg і бути модальним.

Як і для діалогу першого рівня визначимо поля для зберігання карти з інформацією, що введено через діалог, унікального ідентифікатора для рядка таблиці, що створюється або редагується, та для посилання на id кафедри:

```
private Map<String, Object> map;
private int idGraduate;
private int idCathedra;
```

Далі можна реалізувати методи інтерфейсу IDlg за зразком аналогічних методів діалогу першого рівня.

Реалізуймо також конструктор з параметром типу Map. Цей конструктор буде використовуватися для редагування записів таблиці Graduate.

```
public DlgGraduate(Map map) {
  this();
  idGraduate = (Integer) map.get("id");
  idCathedra = (Integer) map.get("idcathedra");
  textFieldFio.setText((String) map.get("fio"));
  textFieldFinish.setText(String.valueOf(map.get("finish")));
  textFieldBall.setText(String.valueOf(map.get("ball")));
  textFieldCathedra.setText((String) map.get("cathedra"));
}
```

Далі слід реалізувати методи обробки подій натискання на кнопки. Тут ми розглянемо тільки кнопку Ok, для кнопки Cancel метод такий самий, як і у діалозі для першого рівня.

```
protected void onOkButton(ActionEvent e) {
  if(idCathedra == 0) {
```

```
      JOptionPane.showMessageDialog(textFieldCathedra,
"Cathedra not selected");
      return;
    }

    String fio = textFieldFio.getText();
    String finishTxt = textFieldFinish.getText();
    String ballTxt = textFieldBall.getText();

    int finish = 0, ball = 0;
    try {
      finish = Integer.parseInt(finishTxt);
    } catch (NumberFormatException e2) {
      JOptionPane.showMessageDialog(this, "\""
          + finishTxt +"\" is not correct number");
      return;
    }
    try {
      ball = Integer.parseInt(ballTxt);
    } catch (NumberFormatException e1) {
      JOptionPane.showMessageDialog(this, "\""
          + ballTxt +"\" is not correct number");
      return;
    }

      map = new LinkedHashMap<>();
    map.put("id", idGraduate);
    map.put("fio", fio);
    map.put("finish", finish);
    map.put("ball", ball);
    map.put("idcathedra", idCathedra);
    setVisible(false);
}
```

Реалізуймо також метод вибору кафедри для даного випускника.

```
protected void mouseClickedTextFieldCathedra() {
    String query = QueryCathedra.queryGetAll();
    List<Map<String, Object>> listMap =
        Controller.executeQuery(query);
    DlgSelect ds = new DlgSelect(listMap);
    ds.setTitle("QueryCathedra selection");
    ds.setVisible(true);
    Map map = ds.getMap();
    ds.dispose();
    idCathedra = (int) map.get("id");
    textFieldCathedra.setText((String) map.get("name"));
}
```

2.5.4 **Розробка дизайгу головного вікна**

Кожен студент може розробляти як проєкт в цілому, так і візуальну

частину на свій розсуд. Тут буде розглянуто варіант візуальної частини, який наведено на рисунку 2.7.

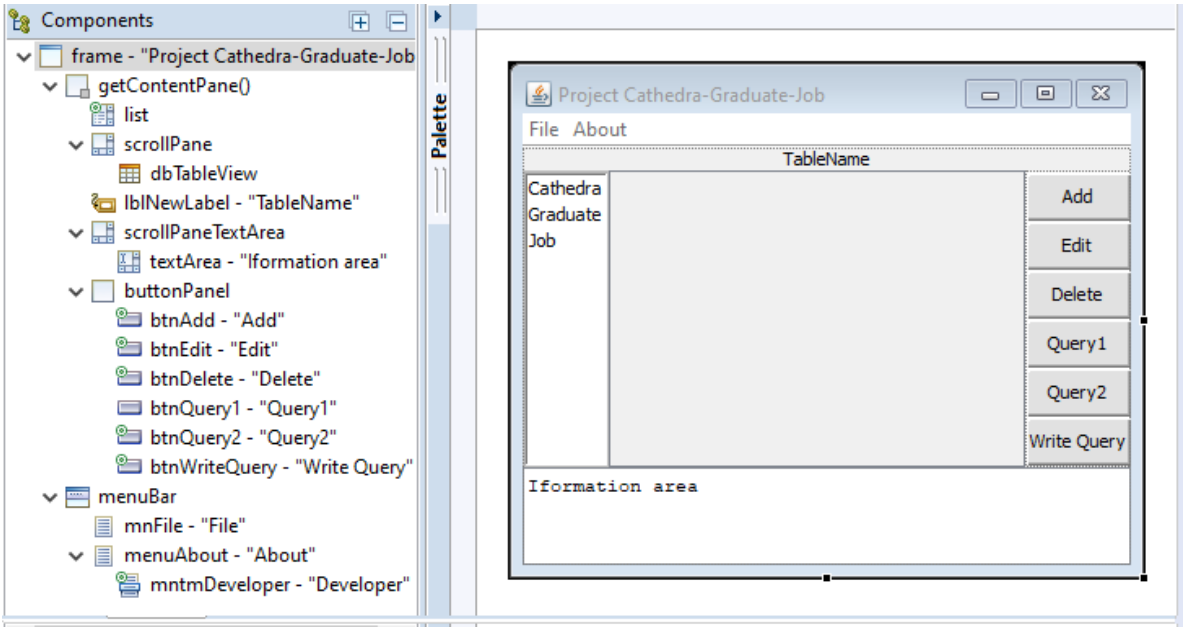

Рисунок 2.7 – Головне вікно в режимі дизайну

Клас для головного вікна MainWindow розташуємо у пакеті view.

Для основної панелі головного вікна налаштовано BorderLayout.

Ліву частину панелі займає список типу JList. В якості моделі списку використовується перелік назв таблиць бази даних. Домовимось, що назви будуть починатися з великої літери.

Праворуч додано панель buttonPanel, яка містить кнопки активізації операцій з базою даних.

По центру головної панелі розташовано компонент dbTableView типу DbTableView. Цей клас було створено під час виконання лабораторної роботи. Компонент буде використовуватися для виведення вмісту таблиць бази та результатів запитів до бази.

У верхній частині панелі розташовано компонент JLabel, у якому буде відображатися назва таблиці або запиту.

Нижню частину займає scrollPane з компонентом textArea, куди будемо виводити службові повідомлення в разі потреби. Висоту цієї частини вікна можна налаштувати у властивості (з розширеного списку) preferedSize компонента scrollPaneTextArea.

До складу головного вікна входить також компонент menuBar, елементи якого будуть надавати інформацію про розробника проєкту і змогу визначитися із розташуванням бази даних.

З виведенням інформації про розробника ми вже розібралися у попередній роботі, а для визначення розташування бази даних треба з функцією меню File→setDbFullName зв’язати метод, що схожий на наступний:

```
protected void onSetDbFullName() {
  String dbName = JOptionPane.showInputDialog(
```

```
            "Enter DB full name", "d:/1/dbBiv");
        DbConnector.setDbFullName(dbName);
    }
```

2.5.5 **Програмування головного вікна**

2.5.5.1 Допоміжний метод головного вікна

Реалізуймо метод, який дозволяє визначити назву таблиці, яку вибрано у списку.

```
    private String getSelectedTable() {
        String tableName = list.getSelectedValue();
        if(tableName == null) {
            JOptionPane.showMessageDialog(frame,
                "Table was not Selected.");
        }
        return tableName;
    }
```

Далі доцільно розглянути методи реакції на дії користувача застосунку. Результатом запитів на перегляд, додавання, редагування та вилучення даних з таблиці буде поява на екрані вмісту таблиці БД. Але для виведення вмісту таблоці нам потрібна її модель у вигляді списку карт. Тому спочатку напишемо метод отримання такої моделі по назві таблиці.

2.5.5.2 Метод отримання моделі таблиці таблиці БД

Тут за допомогою рефлексії ми вкликаємо метод queryGetAll класу, що відповідає потрібній таблиці БД. Цей метод повертає текст sql запиту, який передається контролеру на виконання:

```
    private List<Map<String, Object>>  getDbData(String tableName){
        try {
            String queryClass  = "query.Query" + tableName;
            Class<?> clz = Class.forName(queryClass);
            Method mtd = clz.getMethod("queryGetAll");
            String sql = (String) mtd.invoke(null);
            List<Map<String, Object>> model = Controller.executeQuery(sql);
            return model;
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
```

2.5.5.3 Метод, що обробляє подію вибору таблиці із списку

Цей метод, перш за все, з'ясовує, чи існує таблиця, назва якої вибрана із списку. Якщо таблиця не існує, контролеру надсилається запит на створення таблиці.

Після цього контролеру надсилається запит на отримання вмісту таблиці

БД і результат відображається у компоненті dbTableView.

```
protected void onMouseClickedList(MouseEvent e) {
   getTextArea().setText("");
   String tableName = getSelectedTable();
   if(tableName == null) return;
   if(!Controller.tableExist(tableName))
      Controller.createTable(tableName);
   List<Map<String, Object>>  model = getDbData(tableName);
   dbTableView.setDbTableModel(model);
}
```

2.5.5.4 Метод додавання запису до таблиці БД

Цей метод спочатку створює і активізує діалог введення даних для вибраної таблиці (зауважимо, що поки ми маємо такий діалог тільки для перших двох таблиць). Діалог повертає карту з даними, яку ми відправляємо контролерові. Після цього отримуємо від контролера нову версію таблиці і відображаємо її.

```
protected void onBtnAdd(ActionEvent e) {
   String tableName = getSelectedTable();
   if(tableName == null) return;
   try {
      //Create an activate dialog for selected dbTableView
      Class<?> clz = Class.forName("view.Dlg"
            + getSelectedTable());
      IDlg dlg = (IDlg)clz.newInstance();
      dlg.setVisible(true);
      //Get data from dialog
      Map<String, Object> map = dlg.getMap();
      //Send data to controller
      Controller.add(tableName, map);
      //Show results of operation
      getTextArea().setText("Added " +map.toString());
      dbTableView.setDbTableModel(getDbData(tableName));
   } catch (Exception e1) {
      e1.printStackTrace();
   }
}
```

2.5.5.5 Метод редагування запису в таблиці БД

Цей метод схожий на попередній, але відрізняється тим, що до діалогу передається карта з даними, що відповідають вибраному рядку таблиці БД, який потрібно відредагувати.

```
protected  void onBtnEdit(ActionEvent e) {
   String tableName = getSelectedTable();
   if(tableName == null) return;
   //Map for selected row
   Map<String, Object> map = dbTableView.getSelectedRowMap();
```

```
    if(map == null) return;
    try {
       //Create an activate dialog for selected dbTableView row
       Class clz = Class.forName("view.Dlg" + getSelectedTable());
       Constructor<IDlg> cns = clz.getConstructor(Map.class);
       IDlg dlg = cns.newInstance(map);
       dlg.setVisible(true);
       //Get new data from dialog
       Map<String, Object> newMap = dlg.getMap();
       //Send data to controller
       Controller.edit(tableName, newMap);
       //Show results of operation
       getTextArea().setText("OLd data " +map.toString());
       dbTableView.setDbTableModel(getDbData(tableName));
    } catch (Exception e1) {
       e1.printStackTrace();
    }
}
```

2.5.5.6 Метод вилучення запису із таблиці БД

У цьому методі інформація про вибраний рядок, який потрібно видалити з таблиці, пересилається контролеру.

```
protected  void onBtnDelete(ActionEvent e) {
   String tableName = getSelectedTable();
   if(tableName == null) return;
   //Отримуємо карту для обраного рядка таблиці
   Map<String, Object> map = dbTableView.getSelectedRowMap();
   if(map == null) return;
   //Фіксуємо id вибраного рядка
   Controller.delete(tableName, map);
   dbTableView.setDbTableModel(getDbData(tableName));
}
```

2.5.5.7 Метод реалізації запиту Query1

Як приклад розглянемо реалізацію запиту на отримання переліку випускників деякого року для вибраної кафедри.

У класі Query було створено метод query1, який потребує двох параметрів. Можна створити діалог для отримання значень цих параметрів. Зовнішній вигляд такого діалогу показано на рисунку 2.8.

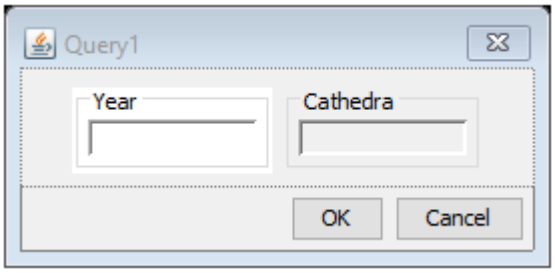

Рисунок 2.8 – Діалог для введення даних до заптиу 1

Приклади програмування схожих диалогів розглядалися вище.

Будемо вважати, що діалог має два публічних методи, які повертають значення, що потрібнв для запиту.

Після цього можа написати метод візуальної частини, який буде викликатися після натискання на кнопку Query1.

```
protected void onBtnQuery1() {
   DlgQuery1 dlg = new DlgQuery1();
   dlg.setVisible(true);
   int finish = dlg.getYear();
   int idCathedra = dlg.getIdCathedra();
   String query = Query.query1(finish, idCathedra);
   dbTableView.setDbTableModel(Controller.executeQuery(query));
}
```

2.5.5.8 Реалізація діалога для довільного запита до БД

Для реалізаці довільного запиту до бази створімо діалог DlgWriteQuery. Вигляд цього діалогу в режимі дизайну показано на рисунку 2.9.

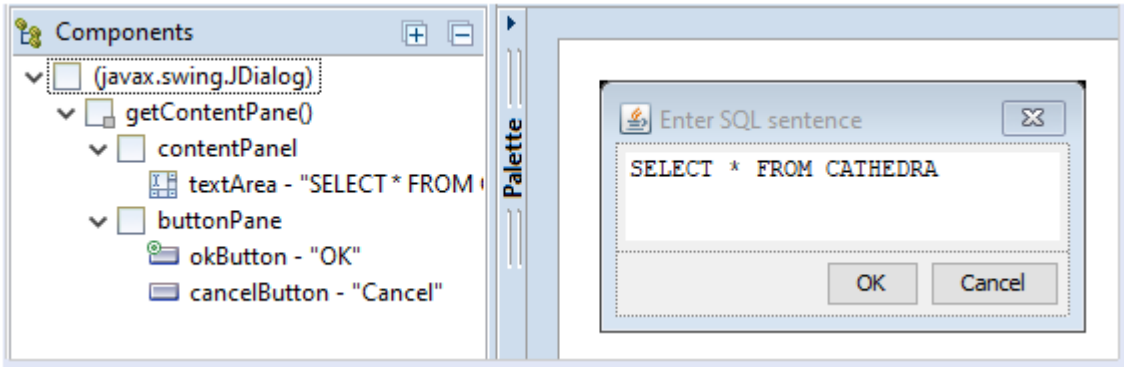

Рисунок 2.9 – Діалог для введення довільного запиту

Для введення тексту запиту у діалзі використовується компонент JTextArea.

Для збереження тексту використовується поле:

```
private String query;
```

З кнопкою Ok та Cancel пов'язані методи:

```
protected void onOk() {
   query = textArea.getText();
```

```
    setVisible(false);
}
protected void onCancel() {
    query = null;
    setVisible(false);
}
```

Для доступу до тексту запиту використовуємо метод:

```
public String getQuery() {
    return query;
}
```

2.5.5.9 Метод візуальної частини для реалізації довільного запита до БД

Цей метод пов'язаний з кнопкою WriteQuery. Після натискання на кнопку на екрані з’являється діалог з полем для введення запиту. Після закриття діалогу запит надсилається контролеру. Результат запиту відображаєиться у таблиці.

```
protected void onWriteQuery() {
    DlgWriteQuery dlg = new DlgWriteQuery();
    dlg.setVisible(true);
    String query = dlg.getQuery();
    dlg.dispose();
    if(query.toLowerCase().indexOf("select")>=0) {
          dbTableView.setDbTableModel(Controller.
          executeQuery(query));
          textArea.setText(query);
    }
    else {
       int res = Controller.executeUpdate(query);
       textArea.setText("Updated " + res);
    }
}
```

2.6 Вимоги до звіту

У звіті має бути предствалено таку інформацію:

– назва роботи;
– мета роботи;
– схема бази даних;
– підрозділ з кодом класів, які досліджувався, та результатами досліджень окремими пунктами;
– підрозділ з результатами виконання самостійної роботи, де окремими пунктами наведено вигляд візуальної частини проєкту, тексти класів або методів, які створювалися або доопрацьовувалися, та результати тестування у вигляді копій екранів. Для класів візуальних частин наводити тільки зображення та методи обробки подій;
– підрозділ висновків.

2.7 Контрольні питання

1. Розповісти про JDBC архітектуру.
2. Охарактеризуйте види JDBC драйверів.
3. Основні класи та інтерфейси JDBC.
4. Назвіть основні функції Driver Manager.
5. Призначення та основні методи інтерфейсу Connection.
6. Призначення та основні методи інтерфейсу Statement.
7. Призначення та основні методи інтерфейсу ResultSet.
8. Метод getConnection() та його параметри.
9. Назвіть порядок підключення до джерела даних.
10. Назвіть порядок встановлення з'єднання з джерелом даних.
11. Архітектура застосунку. Основні складові та комунікація між ними.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Quick and simple free tool to help you draw your database relationship diagrams –Доступу до ресурсу: https://dbdiagram.io/d
2. Онлайн діаграми – Доступу до ресурсу: https://chrome.google.com/webstore/detail/drawio-diagrams/onlkggianjhjenigcpigpjehhpplldkc
3. What is Apache Derby? –Доступу до ресурсу: http://db.apache.org/derby/
4. How to Get Started with Apache Derby (JavaDB). – Доступу до ресурсу: https://www.codejava.net/java-se/jdbc/how-to-get-started-with-apache-derby-javadb
5. Swing Application Window Wizard. –Доступу до ресурсу: https://help.eclipse.org/2020-03/index.jsp?topic=%2Forg.eclipse.wb.swing.doc.user%2Fhtml%2Fwizards%2Fswing%2Fapplication_window.html
6. Java Database Connectivity. –Доступу до ресурсу: https://uk.wikipedia.org/wiki/Java_Database_Connectivity
7. JDBC Tutorial. –Доступу до ресурсу: https://www.tutorialspoint.com/jdbc/jdbc-like-clause.htm

3 ЛАБОРАТОРНА РОБОТА № 3.
ТЕХНОЛОГІЯ ЗВ'ЯЗУВАННЯ XML ДАНИХ

3.1 Мета роботи

В результаті виконання лабораторної роботи студент має отримати такі знання і навички:

– засвоїти основні поняття, що пов'язані з XML документами;
– ознайомитися з можливостями Eclipse для роботи з XML документами;
– ознайомитися з технологією JAXB, та деякими класами пакета javax.xml.bind;
– ознайомитися з поняттям JSON-документ;
– створити проєкт, в якому реалізувати роботу з XML та JSON документами.

3.2 Короткі теоретичні відомості

3.2.1 **XML документи**

XML *(Extensible Markup Language)* – це стандарт запису ієрархічно структурованих даних для обміну між різними застосунками, зокрема, через Інтернет. XML-документ складається із текстових знаків, і придатний до читання як людиною, так і комп'ютером.

Мова XML не залежить від платформи. Це означає, що будь-яка програма, розроблена для використання XML, може читати та обробляти XML-дані незалежно від устаткування або операційної системи. Завдяки такій сумісності мова XML стала однією з найпопулярніших технологій обміну даними.

3.2.1.1 Приклад XML-документа

Прикладом такого документа може бути результат запиту до бази даних. Деякі СУРБД дозволять отримувати результати запитів безпосередньо у вигляді XML-файлу, у інших випадках доводиться виконувати перетворення, щоб отримати XML-файл. Проблема перетворення полягає у тому, що результат запиту до СУРБД має вигляд таблиці, а XML-документ має ієрархічну структуру, тобто вигляд дерева.

XML-документи зберігають у файлах з розширенням .xml.

Нижче наведено вміст XML-файлу, що містить інформацію про оцінки студентів.

```
<?xml version="1.0" encoding="UTF-8"?>
<STUDENTS_MARKS_LIST>
    <STUDENT id="7" name="Kot I.F.">
        <MARK mark="67">
            <SUBJECT id="4" name="OOP" teacher="Byvoino P.G."/>
        </MARK>
        <MARK mark="89">
            <SUBJECT id="5" name="TPCS" teacher="Prila O.A."/>
        </MARK>
```

```
    </STUDENT>
    <STUDENT id="6" name="Zub P.G.">
        <MARK mark="34">
            <SUBJECT id="4" name="OOP" teacher="Byvoino P.G."/>
        </MARK>
        <MARK mark="77">
            <SUBJECT id="5" name="TPCS" teacher="Prila O.A."/>
        </MARK>
    </STUDENT>
</STUDENTS_MARKS_LIST>
```

Перший рядок цього документа являє собою так звану XML-декларація, де вказується версія XML, спосіб кодування документа та інша допоміжна інформація, що використовується у процесі обробки документа. Ознакою того, що це є інструкція з обробки, є тег <? ?>.

Далі йде **кореневий елемент**, що позначений тегом `<STUDENTS_MARKS_LIST>`. У нашому прикладі він не має атрибутів, але це не обов'язково.

До складу кореневого елемента ієрархічно входять інші елементи документа.

Закінчується документ закриваючим тегом `</STUDENTS_MARKS_LIST>`. Коса лінія перед назвою тегу є ознакою того, що тег – закриваючий.

У третьому рядку нашого XML документа починається опис елемента «студент». Його відкриває тег <STUDENT ...>, який містить назви атрибутів студента та їх значення: id="7" name="Kot I.F."

До складу елемента «студент» входить елемент «оцінка» з тегом <MARK …>, у якого є атрибут mark (кількість балів) та елемент «предмет». Цей елемент позначається тегом <SUBJECT…> і має два атрибути – назву предмету та ім'я викладача.

Таким чином, ми бачимо, що документ складається з елементів. У нашому випадку до складу документа входить два елементи «студент». До складу елементів можуть входити інші елементи. У нашому документі кожний студент має по два елементи «оцінка». Оцінка, у свою чергу має елемент «предмет». Елементи можуть мати атрибути.

3.2.1.2 Синтаксичні правила XML

Нижче наведено правила, яких необхідно дотримуватися при складанні XML документів:

- коренем може бути лише один елемент.
- непорожні елементи мають бути розмічені початковим та кінцевим тегами (наприклад, `<пункт>Пункт1</пункт>`). Порожні елементи (що містять тільки атрибути) можуть позначатися «закритим» тегом, наприклад `<IAmEmpty />`. Така пара еквівалентна `<IAmEmpty></IAmEmpty>`.
- один елемент не може мати декілька атрибутів з однаковою назвою. Значення атрибутів розміщуються або в одинарних ('), або у подвійних (") лапках.

– теги можуть бути вкладені, але не можуть перекриватись. Кожен некореневий елемент мусить повністю перебувати в іншому елементі.

– документ має складатися тільки з правильно закодованих дозволених символів Юнікоду. Єдиними симолами кодуваннями, які обов'язково має розуміти XML-процесор, є UTF-16 та UTF-8. Фактичне та задекларованe кодування (*encoding*) документа мають збігатись. Кодування може бути задекларовано на самому початку документа. У разі відсутності інформації про кодування, документ має бути в кодуванні UTF-8 (або його підмножині ASCII).

3.2.2 **XML схема**

Між даними та їх представленням у вигляді XML документа немає однозначної відповідності. Так, у вище наведеному прикладі, можна було і не створювати елемент «предмет», а назву предмету та ім’я викладача представити як атрибути елемента «оцінка», що, можливо, було б і коректніше. Окрім того, для однакових елементів даних у різних XML документах можуть використовуватися різні назви тегів. Ці особливості ускладнюють обробку XML документів. Тому XML документ прийнято пов’язувати з XML схемою, яка являє собою ніби шаблон для створення і контролю коректності XML документа.

3.2.2.1 Графічне редставлення XML схеми

XML схема може мати графічний вигляд. Для XML документа, що розглядався вище як приклад, XML схема може мати такий вигляд, рисунок 3.1.

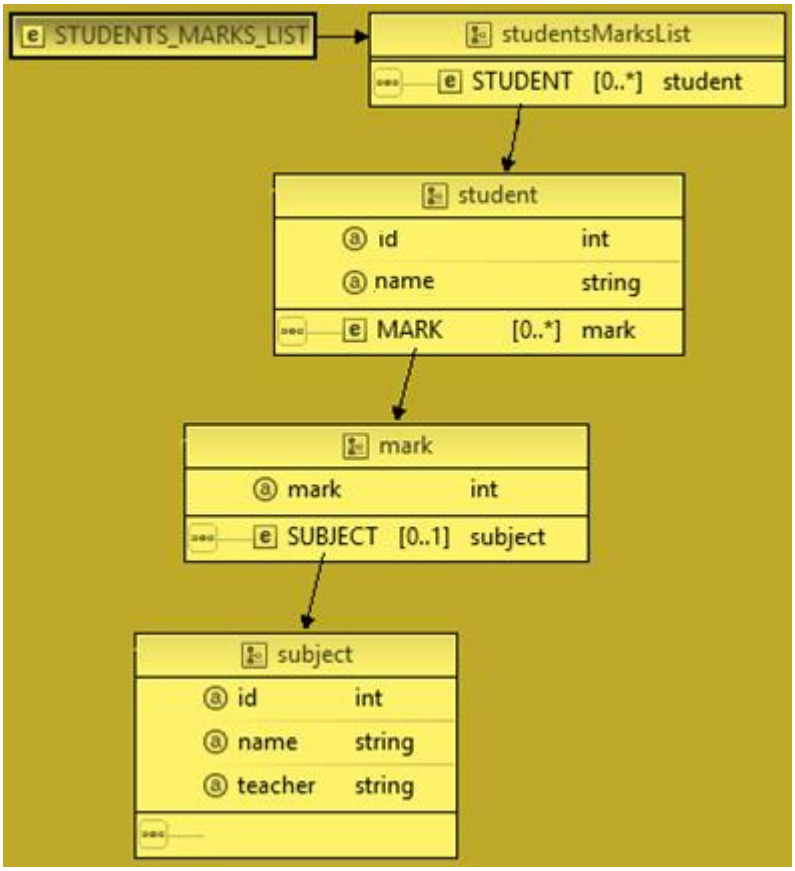

Рисунок 3.1 – Графічне представлення схеми XML документа

3.2.2.2 Представлення схеми XML документа у вихляді .xsd файла

Графічне представлення схеми досить добре сприймається людиною, але для комп'ютерної програми таке представлення не є дуже зручним. Тому схеми XML документів описують за допомогою XML документів. Файли з XML представленням схем документів мають розширення .xsd.

Нижче, для прикладу, наведено вміст xsd схеми, що наведена на рисунку 3.1.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="STUDENTS_MARKS_LIST" type="studentsMarksList"/>

  <xs:complexType name="studentsMarksList">
    <xs:sequence>
      <xs:element name="STUDENT" type="student" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="student">
    <xs:sequence>
      <xs:element name="MARK" type="mark" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:int" use="required"/>
    <xs:attribute name="name" type="xs:string"/>
  </xs:complexType>
```

```
  <xs:complexType name="mark">
     <xs:element name="SUBJECT" type="subject" minOccurs="0"/>
    <xs:attribute name="mark" type="xs:int" use="required"/>
  </xs:complexType>

  <xs:complexType name="subject">
    <xs:attribute name="id" type="xs:int" use="required"/>
    <xs:attribute name="name" type="xs:string"/>
   <xs:attribute name="teacher" type="xs:string"/>
   </xs:complexType>
</xs:schema>
```

Розглянемо деякі особливості побудови XML схем.

У схемі нема довільних тегів. Назви елементів визначені простором імен `xmlns:xs="http://www.w3.org/2001/XMLSchema"`, і ці назви будуть ідентифікуватися за допомогою префіксів «xs:». Використовують також префікс «xsd:».

Визначення елемента у схемі XML-документа полягає в наданні цьому елементові імені та типу. Назви імен елементів та їх типів можуть бути довільними, але типи надалі мають бути визначені:

```
<xs:element name="STUDENTS_MARKS_LIST"
              type="studentsMarksList"/>
<xs:element name="STUDENT" type="student"/>
<xs:element name="MARK" type="mark"/>
<xs:element name="SUBJECT" type="subject"/>
```

Визначення атрибуту також вимагає надання імені та типу.

```
<xs:attribute name="stud_name" type="xs:string"/>
<xs:attribute name="mark" type="xs:int"/>
```

У наведеному прикладі для атрибутів використовуються типи, що вже визначені у просторі імен.

3.2.2.3 Прості типи для атрибутів схеми

Простими називають типи для атрибутів. Існує великий перелік вже визначених простих типів. Деякі з них наведені у таблиці 3.1.

Таблиця 3.1 – Перелік деяких визначених простих типів

Прості типи	Приклад / пояснення
string	"це рядок символів"
boolean	true, false
double	Дійсні числа з плаваючою точкою подвійної точності
decimal	Дійсні числа з фіксованою точкою, наприклад123.45
int	123456789
integer	-12345; 0; 12345
negative-integer	-12345
date	2000-02-16

Окрім вбудованих простих типів, можна створювати нові прості типи, використовуючи тег `xs:simpleType` і додаткові спеціальні теги. Спектр цих типів - від тексту у певному форматі (такого, як телефонний номер або артикул товару) до числового ряду або списку. Для прикладу створимо тип, який обмежує значення для цілого числа між 1 і 100 000.

```
<xs:simpleType name="idNumber">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="1" />
    <xs:maxInclusive value="100000" />
  </xs:restriction>
</xs:simpleType>
```

Можна використовувати інший тип обмеження, створюючи список допустимих значень.

```
<xs:simpleType name="resultType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="excellent" />
    <xs:enumeration value="good" />
    <xs:enumeration value="pass" />
  </xs:restriction>
</xs:simpleType>
```

3.2.2.4 Складні типи для елементів схеми

Складні типи використовують для визначення типів елементів, використовуючи тег `xs:complexType`. Опис складного типу можна розмістити після опису елемента цього типу.

```
<xs:element name="media">
  <xs:complexType>
    <xs:attribute name="mediaid" type="xs:integer" />
  </xs:complexType>
```

Додавання до елемента нащадків також вимагає використання складних типів. Найпростіше перерахувати один або більше дочірніх елементів, використовуючи елемент sequence:

```
<xs:complexType name="studentsList">
    <xs:sequence>
      <xs:element name="STUDENT" type="student" />
    </xs:sequence>
</xs:complexType>
```

Елемент sequence визначає всі можливі дочірні елементи для даного елемента. У деяких випадках, однак, необхідно вибрати один елемент зі списку альтернатив. У цьому разі слід використовувати елемент choice:

```
<xs:complexType name="locationType">
  <xs:choice>
    <xs:element name="description" type="xs:string" />
```

```
      <xs:element name="place" type="xs:string" />
    </xs:choice>
  </xs:complexType>
```

У попередніх прикладах всі елементи і атрибути, що додаються в схему, повинні були з'являтися тільки один раз. Зрозуміло, що це не завжди бажано. Використовуючи minOccurs і maxOccurs, можна управляти тим, чи повинен елемент з'являтися обов'язково і чи може він повторюватися. У даному прикладі схема визначає, що елемент mark не є обов’язковим, але може зустрічатися скільки завгодно разів:

```
<xs:element name="MARK" type="mark"
minOccurs="0" maxOccurs="unbounded"/>
```

Інколи корисним може бути елемент all, який обмежує вміст елемента, але не порядок, в якому з'являються нащадки. Атрибути minOccurs і maxOccurs у цьому випадку можуть приймати значення або 0, або 1.

```
<xs:element name="subject">
  <xs:complexType >
    <xs:all>
      <xs:element name="i" minOccurs="0" maxOccurs="1"
                  type="xs:string" />
      <xs:element name="b" minOccurs="0" maxOccurs="1"
                  type="xs:string" />
    </xs:all>
  </xs:complexType>
</xs:element
```

3.2.3 **Створення XML документів за допомогою Eclipse**

Eclipse дозволяє створювати та опрацьовувати XML документи та схеми. Для того, щоб створити новий документ або схему використовуйте послідовність викликів функцій меню File->new->Other->XML.

Для роботи з документами можна використовувати як режим редагування тексту, так і режим графічно-текстового дизайну.

Ці редактори автоматично відкриваються при спробі відкриття XML або XSD файлу.

Але вручну створювати ці файли незручно. На щастя, існують технології автоматичного створення таких документів.

3.2.4 **Технологія JAXB**

Технологія JAXB (Java API for XML Binding) пропонує швидкий і зручний спосіб створення двостороннього перетворення між XML документами і класами та об’єктами мови Java, а також надає інструменти для верифікації XML документів. Відповідні класи, інтерфейси та анотації знаходяться у пакеті javax.xml.bind. Будьте уважні, імпортуючи класи. Треба вибирати саме цей пакет, javax.xml.bind.

Процес перетворення даних, що знаходяться в пам'яті комп’ютера, у формат їх зберігання називають маршалізацією. Для технологій Java і XML,

маршалізація являє собою перетворення деякого набору Java-об'єктів в XML-документ.

Зворотній процес, тобто процес перетворення даних з формату зберігання у формат представлення в пам'яті, називають демаршалізацією. У нашому випадку це перетворення XML-документа в Java класи та об’єкти.

3.2.4.1 Вимоги до класів об’єктів, що підлягають маршалізації

До класів, об’єкти яких є складовими XML-документа, слід застосувати анотації пакета javax.xml.bind.annotation.

Клас, що визначає кореневий елемент XML-документа має бути проанотований за допомогою анотації XmlRootElement.

Наприклад:

```
@XmlRootElement
public class ConnectionData {
```

Для полів класів, що є елементами або атрибутами XML-документа, мають бути визначені геттери. За замовчуванням відповідні поля вважаються елементами. Для уточнення ролі полів в XML-документі для геттерів використовуються анотації XmlAttribute, XmlRootElement.

Далі ви побачите і інші анотації.

3.2.4.2 Маршалізація даних

Звичайно, можна написати потрібний XML-документ вручну. Але краще для його створення скористатися технологією JAXB.

У пакеті javax.xml.bind знаходиться клас JAXBContext, який надає доступ до можливостей JAXB API. Екземпляр цього класу можна отримати через статичний метод newInstance цього самого класу. Параметром методу має бути об’єкт типу Class для класу, що відповідає за кореневий елемент XML-файлу. Наприклад:

```
JAXBContext jaxbContext = JAXBContext.newInstance(ConnectionData.class);
```

За вирішення завдань маршалізації відповідає об’єкт типу Marshaller пакета javax.xml.bind. Такий об’єкт можна створити за допомогою методу createMarshaller() класу JAXBContext, наприклад:

```
Marshaller jaxbMarshaller = jaxbContext.createMarshaller();
```

В інтерфейсі Marshaller визначено декілька статичних констант для визначення назв властивостей об’єктів цього типу. Ці константи використовуються як параметри методу setProperty.

У наведеному нижче прикладі властивість ***JAXB_FORMATTED_OUTPUT*** об’єкту jaxbMarshaller отримує значення true і об’єкт налаштовується на форматоване виведення XML-файлу.

```
jaxbMarshaller.setProperty( Marshaller.JAXB_FORMATTED_OUTPUT, true );
```

Для створення XML-файлу використовується метод marshall з двома параметрами, що має декілька варіантів. Першим параметром цього методу є

кореневий об’єкт XML-документа (cd), а другий визначає напрямок виведення XML-файлу. Нижче наведено приклад, де XML-файл виводиться спочатку у файл і потім на консоль:

```
jaxbMarshaller.marshal( cd, new File( "connectionData.xml" ) );
jaxbMarshaller.marshal( cd, System.out );
```

3.2.4.3 Демаршалізація даних

Для демаршалізації, так само як і у разі маршалізації, нам знадобиться об'єкт класу JAXBContext. За його допомогою можна створити об’єкт типу Unmarshaller, скориставшись методом createUnmrshaller(), наприклад:

```
Unmarshaller jaxbUnmarshaller = jaxbContext.createUnmarshaller();
```

Отримати демаршалізований об’єкт можна за допомогою методу unmarshall, який має декілька варіантів. У наведеному нижче прикладі, для отримання об’єкту типу ConnectionData використовується метод з параметром типу java.io.File.

```
ConnectionData cd = (ConnectionData) jaxbUnmarshaller
                            .unmarshal(new File("connectionData.xml"));
```

3.2.4.4 Створення XSD схеми

І у цьому випадку нам знадобиться об'єкт класу JAXBContext, який може згенерувати схему за допомогою методу generateSchema. Цей метод потребує доволі заплутаного параметра у вигляді об’єкту анонімного класу, що успадковує абстрактний клас javax.xml.bind.SchemaOutputResolver і потребує реалізації абстрактного методу createOutput, який повертає об’єкт інтерфейсного типу Result з пакета javax.xml.transform. Незважаючи на заплутаність коду нам достатньо врешті решт вказати тільки ім’я файлу, наприклад:.

```
jaxbContext.generateSchema(new SchemaOutputResolver() {
   @Override
   public Result createOutput(String namespaceUri,
                                    String suggestedFileName) {
         return new StreamResult(new File("schem.xsd"));
   }
});
```

3.2.4.5 XJC компілятор

XJC компілятор використовується для генерації Java класів ієрархічної об’єктної моделі на основі XSD-схеми. Цей JAXB компілятор перетворює схему у пакет класів Java, які відповідають структурі, що описана в XSD схемі. Ці класи мають спеціальні анотації JAXB, які забезпечують можливість створення XML-документів.

Для виклику компілятора XJC використовують різні підходи. Ви можете ознайомитися з ними в Internet. Ми тут розглянемо виклик компілятора через Eclipse. Щоб створити класи, потрібно виконати наступну послідовність дій.

1. Викликати контекстне меню для .xsd файлу і вибрати функцію меню Generate → JAXB Classes.
2. Вибрати проєкт та вести назву пакета до якого будуть занесені створені класи.

Слід зауважити, що пакет з’явиться тільки після виклику функції меню refresh для проєкту у якому генерувалися класи.

Далі слід дописати анотацію @XmlRootElement для кореневого класу, яка там чомусь не прописується.

Після цього можна використовувати створену об’єктну ієрархічну модель для операцій демаршалізації та маршалізації.

3.2.5 **JSON документи**

JSON (JavaScript Object Notation) це ще один текстовий формат обміну даними. Цей формат часто застосовується у веб-програмуванні. Вважається, що перевагою JSON перед XML є те, що він дозволяє використовувати складні структури в атрибутах, займає менше місця і прямо інтерпретується за допомогою JavaScript в об'єкти.

JSON оперує з такими поняттями:

– об'єкт (JSONObject), що являє собою невпорядковану множину пар ім'я/значення. Об'єкт починається з символу { і закінчується символом }. Кожне значення слідує за : . Пари ім'я/значення відділяються комами;

– масив (JSONArray), це впорядкована множина значень. Масив починається символом [і закінчується символом]. Значення відділяються комами;

– значення. Може бути рядком символів в подвійних лапках, або числом, або логічними true чи false, або null, або об'єктом, або масивом. Ці структури можуть бути вкладені одна в одну;

– рядок. Це впорядкована множина з нуля або більше символів юнікоду, обмежена подвійними лапками, з використанням escape-послідовностей, що починаються із зворотної косої риски (backslash). Символи представляються теж рядком.

Тип рядок (String) дуже схожий на String в мовах C і Java. Число теж дуже схоже на C- або Java-число, за винятком того, що вісімкові та шістнадцяткові формати не використовуються. Пропуски можуть бути вставлені між будь-якими двома лексемами.

Нижче наведено приклад JSON документа, що отримано з XML документа, який розглядався вище як приклад.

```
{"studentsMarksList": {"STUDENT": [
  {
    "name": "Kot I.F.", "id": 7,
    "MARK": [
      {
        "subject": {
          "teacher": "Byvoino P.G.", "name": "OOP", "id": 4
```

```
        },
        "mark": 63
      },
      {
        "subject": {
          "teacher": "Prila O.A.", "name": "TPCS", "id": 5
        },
        "mark": 92
      }
    ]
  },
  {
    "name": "Zub P.G.", "id": 6,
    "MARK": [
      {
        "subject": {
          "teacher": "Byvoino P.G.", "name": "OOP", "id": 4
        },
        "mark": 72
      },
      {
        "subject": {
          "teacher": "Prila O.A.", "name": "TPCS", "id": 5
        },
        "mark": 29
      }
    ]
  }
]}}
```


Пряме і зворотне перетворення між XML та JSON документами можна реалізувати за допомогою спеціально розробленого програмного забезпечення. Зокрема можна скористатися файлом org.json-chargebee-1.0.jar. Цей файл можна скачати з інтернету або завантажити з мудла.

Можливо, є вже щось більш нове. Отриманий .jar слід додати до свого проєкту через файлову систему і включити у Build Path.

3.2.6 **Бібліотека JSON.org**

JSON.org містить класи для розбору і створення JSON зі звичайних Java-рядків. Також вона має можливість перетворювати JSON в XML, HTTP header, Cookies і багато іншого.

Основу цієї бібліотеки складають наступні класи:

– клас org.json.JSONObject - зберігає невпорядковані пари типу ключ - значення. Значення можуть бути типу String, JSONArray, JSONObject.NULL, Boolean і Number. Клас JSONObject також містить конструктори для конвертації Java-рядки в JSON і подальшого її розбору в послідовність ключ-значень;

– org.json.JSONTokener використовується для розбору JSON рядки, а також використовується всередині класів JSONObject і JSONArray;

– org.json.JSONArray зберігає впорядковану послідовність значень у вигляді масиву JSON елементів;

– org.json.JSONWriter представляє можливість отримання Json. Він містить такі корисні в роботі методи, як append (String) - додати рядок в JSON текст, key (String) і value (String) методи для додавання ключа і значення в JSON рядок. Також org.json.JSONWriter вміє записувати масив;

– org.json.CDL - цей клас містить методи для перетворення значень, розділених комами, в об'єкти JSONArray і JSONArray;

– org.json.Cookie розпорядженні методами для перетворення файлів cookie веб-браузера в JSONObject і назад;

– org.json.CookieList допомагає перетворити список кукі в JSONObject і назад.

3.2.7 **Перетворення XML файлу у JSON документ**

Для перетворення XML-документа у JSON файл використовуються класи JSONObject та XML із пакета org.json вищезгаданої бібліотеки.

Клас XML має статичний метод toJSONObject(String), який приймає в якості параметру рядок символів з текстом XML-документа, а повертає об’єкт типу JSONObject. У свою чергу об’єкт JSONObject можна перетворити у рядок символів за допомогою методу toString(). Параметром цього методу є ціле число, що визначає розмір табуляції. Якщо параметр дорівнює 0, файл виводиться одним рядком.

Наприклад:

```
JSONObject jo = XML.toJSONObject(sringXml);
String str = jo.toString(4);
```

3.3 Порядок виконання роботи

3.3.1 **Створення проєкта**

За основу можна взяти копію проєкта з попередньої роботи. Пакет test можна почистити, решту пакетів залишити без змін.

3.3.2 **Приклад використання XML документа**

У попередній лабораторній роботі інформація, що дозволяла ідентифікувати базу даних та отримати з’єднання з нею зберігалася у класі DbConnector. Але, якщо нам потрібно буде щось змінити у цих даних, доведеться переписувати текст класу. Це не завжди зручно.

Альтернативою є зберігання даних, що ідентифікують базу даних, у XML файлі. Такий варіант збереження службових даних є достатньо розповсюдженим.

У такому випадку об’єкт класу DbConnector повинен буде звернутися до XML-документа, демаршалізувати його і отримати об’єкт, який буде містити необхідну інформацію.

Щоб не писати XML-документ вручну, ми скористаймося технологією JAXB, яка дозволяє перетворити об’єкт у XML-документ, але для створення об’єкту потрібен клас.

3.3.2.1 Створення класу ConnectionData

Створімо в пакеті controller клас ConnectionData.

Об’єкти класу ConnectionData будуть містити інформацію, що потрібна для ініціалізації бази даних. Нижче наведено повний текст класу, але ж не забувайте, що конструктор суперкласу, геттери та сеттери можна створити засобами автогенерації коду.

Найбільш цікавим елементом класу є анотації, які вказують, що у XML-документі буде кореневим елементом, що звичайним елементом, а що атрибутом. Як бачимо, анотації для атрибутів прописуються перед геттерами. Якщо їх не ставити, відповідні поля будуть вважатися елементами і вигляд XML-документа буде іншим.

```
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class ConnectionData {

   private  String dbFullName;
   private  String user;
   private String password;
   private String create;

   public ConnectionData() {
      super();
   }

   @XmlAttribute
   public String getDbFullName() {
      return dbFullName;
   }
   @XmlAttribute
   public String getUser() {
      return user;
   }
   @XmlAttribute
   public String getPassword() {
      return password;
   }
   @XmlAttribute
   public String getCreate() {
      return create;
   }

   public void setDbFullName(String path) {
      this.dbFullName = path;
   }
   public void setUser(String user) {
      this.user = user;
```

```
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public void setCreate(String create) {
        this.create = create;
    }
}
```

Тепер можна спробувати створити об’єкт цього класу і перетворити його у XML-документ (маршалізувати).

3.3.2.2 Маршалізація даних

Створімо у пакеті test клас TestMarshall з методом main, текст якого наведено нижче. Параметри бази даних у прикладі взяті умовні, Ви маєте написати ті, що використовували у другій роботі. Це ідентифікує ваш звіт.

```
public static void main(String[] args) {
    //Створення об’єкту
    ConnectionData cd = new ConnectionData();
    cd.setDbFullName("dbFullName");
    cd.setUser("user");
    cd.setPassword("password");
    cd.setCreate("true");
    //Створення XML документа
    try {
        JAXBContext jaxbContext =
                    JAXBContext.newInstance(ConnectionData.class );
        Marshaller jaxbMarshaller = jaxbContext.createMarshaller();
        jaxbMarshaller.setProperty( Marshaller.JAXB_FORMATTED_OUTPUT, true );
        jaxbMarshaller.marshal( cd, new File( "connectionData.xml" ) );
        jaxbMarshaller.marshal( cd, System.out );
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Після запуску програми ми маємо отримати текст XML-документа на консолі. Також має з’явитися XML-файл connectionData.xml. Щоб його побачити, не забудьте оновити проєкт (refresh).

Поміняйте анотації у класі ConnectionData, знову виконайте програму і порівняйте результати.

Текст методу та копії консолі для обох варіантів зафіксуйте у звіті.

3.3.2.3 Демаршалізація даних

Тепер спробуємо відновити об’єкт із XML-файлу. Для цього створімо у пакеті test клас TestUnmarshall з методом main, текст якого наведено нижче.

```
public static void main(String[] args) {
    try {
        // Демаршалізація об'єкту із XML-файлу
        JAXBContext jaxbContext =
```

```
                JAXBContext.newInstance(ConnectionData.class);
        Unmarshaller jaxbUnmarshaller = jaxbContext.createUnmarshaller();
        ConnectionData cd = (ConnectionData) jaxbUnmarshaller
              .unmarshal(new File("connectionData.xml"));
        // Виведення параметрів об'єкту на консоль
        System.out.println(cd.getDbFullName());
        System.out.println(cd.getUser());
        System.out.println(cd.getPassword());
        System.out.println(cd.getCreate());
     } catch (Exception e) {
        e.printStackTrace();
     }
  }
```

Запустимо програму на виконання і маємо отримати на консолі параметри об’єкту.

Поміняйте значення параметрів об’єкта у XML-файлі і знову виконайте програму. Отримаємо нові значення. Тобто ми маємо можливість налаштовувати об’єкти програми, міняючи вміст XML-файлу.

Результати тестування варіантів зафіксуйте у звіті.

3.3.3 **Створення xsd моделі**

Використовуючи засоби JAXB можна створити і xsd модель на основі класу ConnectionData.

Для реалізації цього завдання у пакеті test створімо клас TestCreateXSD з методом main, повний текст якого наведено нижче. Будьте уважні з імпортом.

```
import java.io.File;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.SchemaOutputResolver;
import javax.xml.transform.Result;
import javax.xml.transform.stream.StreamResult;
import controller.ConnectionData;

public class TestCreateXSD {

   public static void main(String[] args) {
      try {
         JAXBContext jaxbContext =
                   JAXBContext.newInstance(ConnectionData.class);
         jaxbContext.generateSchema(new SchemaOutputResolver() {
            @Override
            public Result createOutput(
                   String namespaceUri, String fileName) {
               return new StreamResult(new File("schema.xsd"));
            }
         });

      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Після запуску додатку у проєкті має з’явитися файл schema.xsd, потрібно тільки зробити refresh проєкту. Після цього можна переглянути файл і зафіксувати його текст звіті.

3.3.4 **Перетворення XML-файлу у JSON**

Для того, щоб мати можливість виконати таке перетворення, треба до build path проєкту підключити jar-файл org.json-chargebee-1.0.jar. або його аналог. Про це йшлося у теоретичних відомостях.

Далі слід створити у пакеті test клас TestXMLtoJSON з методом main. У цьому методі відкрити XML-файл, зчитати його у рядок символів, після чого викликати статичний метод toJSONObject класу XML і передати йому сформований рядок символів. Клас XML поверне об’єкт типу JSONObject, який можна перетворити у рядок символів, роздрукувати, або переписати у текстовий файл.

Нижче наведено текст прикладу:

```
import org.json.JSONObject;
import org.json.XML;

public class TestXMLtoJSON {

   public static void main(String[] args) {
      try {
         // Визначаємося з XML-файлом
         File f = new File("connectionData.xml");
         // Переписуємо XML-файл у рядок символів
         Scanner sc = new Scanner(f);
         StringBuilder sb = new StringBuilder();
         while (sc.hasNextLine()) {
            sb.append(sc.nextLine());
         }
         sc.close();
         // Перетворюємо XML рядок у JSON об'єкт
         JSONObject jo = XML.toJSONObject(sb.toString());
         // Виводимо JSON на консоль
         String str = jo.toString(4);
         System.out.println(str);
         // Виводимо JSON у файл
         FileWriter fr = new FileWriter("connectionData.json");
         fr.write(str);
         fr.close();
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Результат тестування зафіксуйте у звіті

3.4 Завдання до самостійної роботи

Доопрацювати попередній проєкт таким чином, щоб для налаштування параметрів бази даних у класі DbConnector використовувався XML-файл, який створено на основі об’єкта типу ConnectionData.

Додати до контролера метод, який буде повертати результат запиту у вигляді об’єкта JSONArray.

У візуальній частині проєкта надати можливість вибирати шлях до бази даних та доповнити методи відображення таблиць кодом, який забезпечить виведення результатів запиту в компонент textArea у вигляді переліку об’єктів JSONObject.

3.5 Рекомендації до виконання самостійної роботи

За основу можна взяти проєкт з попередньої роботи і додати функцію меню File→ConnectionByXml

Після доопрацювання візуальна частина може виглядати так, як показано на рисунку 3.2.

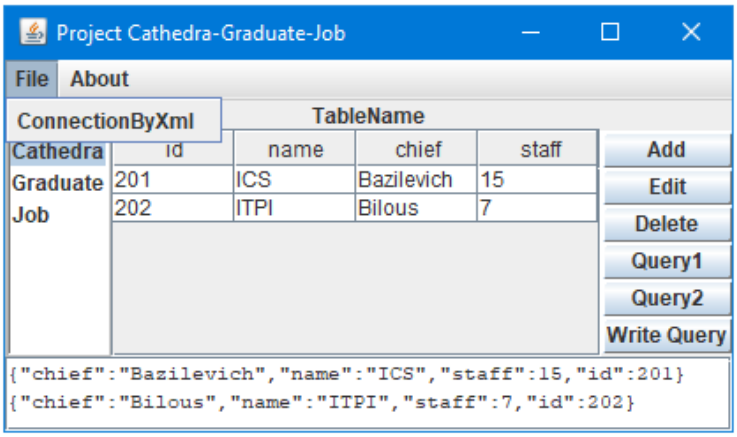

Рисунок 3.2 – Вигляд доопрацьованого інтефейсу користувача

3.5.1 Реалізація налаштування параметрів бази даних з XML файлу

XML файл з налаштуваннями параметрів доступу до БД у нас вже є. Це завдання було реалізовано класом TestMarshall із пакета test.

Далі треба створити метод, який на підставі заданого xml файлу налаштовує клас DbConnector.

```
protected void setPropertyFromXML(String xmlFileName) {
  try {
    // Демаршалізація об'єкту із XML-файлу
    JAXBContext jaxbContext =
              JAXBContext.newInstance(ConnectionData.class);
    Unmarshaller jaxbUnmarshaller =
                        jaxbContext.createUnmarshaller();
    ConnectionData cd = (ConnectionData) jaxbUnmarshaller
         .unmarshal(new File(xmlFileName));
```

```
            //Налаштування параметрів підключення до БД
            DbConnector.setDbFullName(cd.getPath());
            DbConnector.setPropertyUser(cd.getUser());
            DbConnector.setPropertyPassword(cd.getPassword());
            DbConnector.setPropertyCreate(cd.getCreate());
         } catch (Exception e) {
            JOptionPane.showMessageDialog(frame,
                                          "Problem with " + xmlFileName);
            e.printStackTrace();
         }
      }
```

Після цього з кнопкою меню ConnectionByXml пов'язуємо метод:

```
protected void onConnectionByXml() {
   String fName = getFileName(".xml");
   setPropertyFromXML(fName);
}
```

У цьому методі використовується метод getFileName(), який викристовувався в першій лабораторній роботі. В якості параметра в метод передається розширення файлів, серед яких потрібно робити вибір.

Скопіюйте цей метод до класу візуальної частини

Наступним кроком буде поєднання події windowOpened головного вікна з методом, який налаштовує DbConnector, використовуючи XML файл із стандартними налаштуваннями:

```
protected void onWindowOpened() {
   setPropertyFromXML("connectionData.xml");
}
```

Після цього блок статичної ініціалізації параметрів БД у класі DbConnector можна видалити.

Перевірте, чи буде після цього працювати проєкт.

3.5.2 **Реалізація відображення JSON об'єктів**

Для виведення елементів JSONArray У компонент textArea треба до методу обробки події вибору назви таблиці із списку додати такий код:

```
try {
   JSONArray jsar = new JSONArray(model);
   for (int i = 0; i < jsar.length(); i++) {
      textArea.append(jsar.get(i).toString() +'\n');
   }
} catch (Exception e1) {
   e1.printStackTrace();
}
```

3.6 Вимоги до звіту

У звіті має бути предствалено таку інформацію:

– назва роботи;
– мета роботи як окремий підрозділ;
– підрозділ з результатами виконання роботи окремими пунктами. У кожному пункті наведити коду класів, які досліджувався та результати дослідження;
– підрозділ з результатами виконання самостійної роботи, де окремими пунктами наведено вигляд візуальної частини проєкту, тексти класів або методів з власним або переробленим кодом та результати тестування у вигляді копій екранів. Для класів візуальних частин наводити тільки методи обробки подій;
– підрозділ висновків.

На мудл завантажувати zip файл, в якому мають бути звіт, jar файл проєкту (runnable), папка з базою даних, xml файл. Проєкт має працювати, якщо ці файли знаходяться в одному місці.

3.7 Контрольні питання

1. Що таке XML документ і правила його створення.
2. Що таке XSD схема і правила її створення.
3. Що таке JSON документ і правила його створення.
4. Що таке маршалізація і демаршалізація.
5. Які вимоги JAXB до класів стосовно генерації xml та xsd файлів.
6. Яким чином виконати перетворення XML документа у JSON документ

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Introduction to JSON-Java. [Електронний ресурс]. – Режим доступу: https://www.baeldung.com/java-org-json
2. XML and Java: A powerful combination. [Електронний ресурс]. – Режим доступу: https://www.infoworld.com/article/2076453/xml-and-java--a-powerful-combination.html

4 ЛАБОРАТОРНА РОБОТА № 4.
ВИВЧЕННЯ ОБ'ЄКТНО-РЕЛЯЦІЙНОГО ВІДОБРАЖЕННЯ

4.1 Мета роботи

В результаті виконання лабораторної роботи студент має отримати такі знання і навички:

– ознайомитися з технологією об'єктно-реляційного відображення;
– навчитися працювати з базою даних на рівні об'єктів та створювати запити до бази, використовуючи мову запитів JPQL;
– отримати навички створення програми для роботи з реляційною базою даних через об'єктно-реляційне відображення.

4.2 Короткі теоретичні відомості

4.2.1 Об'єктно-реляційне відображення (ORM)

Об'єктно-реляційне відображення (ORM) - це техніка програмування, яка пов'язує реляційну базу даних з концепціями об'єктно-орієнтованого програмування. Як правило, у цій об'єктній моделі кожній таблиці бази даних відповідає клас, а об'єкти цих класів представляють рядки таблиці реальної бази.

Правила об'єктно-реляційного відображення в Java визначаються стандартизованою специфікацією, що має назву JPA (Java persistence API), і являє собою сукупність класів та інтерфейсів.

На сьогоднішній день існує багато різних фреймворків (провайдерів), що реалізують інтерфейси JPA. Ось деякі з них: Hibernate, OpenJPA, EclipseLink..

4.2.2 Архітектура JPA

На рисунку 4.1 наведені основні компоненти JPA і показані взаємозв'язки між ними.

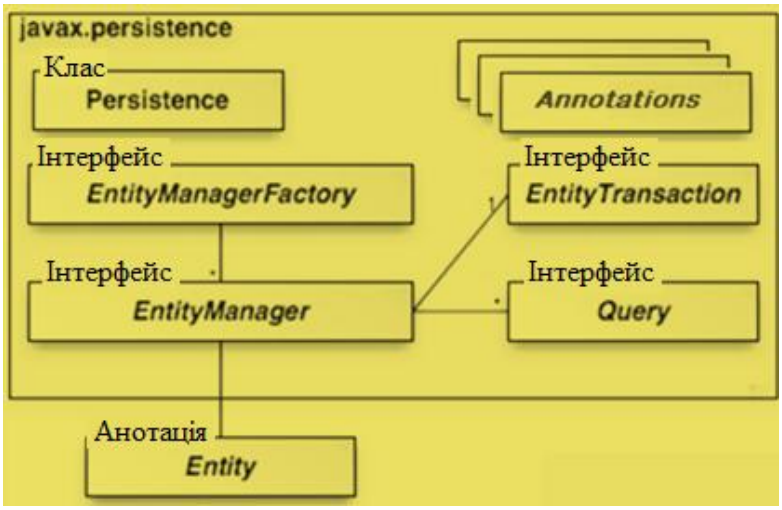

Рисунок 4.1 – Архітектура JPA

Класи та інтерфейси, що визначають JPA знаходяться у пакеті javax.persistence.

Розглянемо їх призначення.

Annotations – сукупність анотацій для класів сутностей, що відповідають таблицям бази даних;

Persistence – клас містить допоміжні статичні методи для отримання об’єкту EntityManagerFactory незалежним від провайдера способом;

EntityManagerFactory – інтерфейс, реалізація якого є фабрикою для створення об'єктів EntityManager;

EntityManager – це основний JPA інтерфейс, що використовується у додатках. Його об’єкти керують набором збережених об'єктів і забезпечують реалізацію API для вставки нових об'єктів і видалення існуючих. З кожним EntityManager пов'язаний свій EntityTransaction, окрім того, EntityManager виступає фабрикою для об'єктів Query.

Entity – сутність, яка є збереженим об'єктом.

EntityTransaction – об'єкт, який керує транзакціями при виконанні операцій із збереженими об'єктами Entity. Операції об’єднуються у групу і, або виконуються повністю або ні, залишаючи сховище даних в незмінному стані.

Query – інтерфейс для виконання запитів по знаходженню збережених об'єктів, які задовольняють заданим критеріям. JPA підтримує запити як на об'єктній мові JPQL (Java Persistence Query Language), так і стандартному SQL. Отримати екземпляри Query можна за допомогою об'єкта EntityManager.

4.2.3 **Метаінформація для процедур ORM**

До складу об’єктної моделі, окрім класів для сутностей, обов’язково має входити XML файл з назвою persistence.xml, що розміщується у папці META-INF проєкту.

Цей файл буде використовуватися при створенні фабрики менеджерів сутностей.

У файлі, окрім обов’язкової інформації, що ідентифікує xml документ, у тезі persistence-unit обов’яково має бути вказано ім’я persistence модуля. Це ім’я може будь яким, наприклад, назвою проєкту. І воно використовується під час створення EntityManagerFactory.

В одному такому файлі можна визначити кілька persistence модулів. Потрібний модуль буде вибиратися при створенні EntityManagerFactory.

У вкладених тегах може бути розташована додаткова інформація.

Класи моделі можуть бути перелічені у тегах class.

У розділі properties можуть бути перелічені властивості, необхідні для з’єднання з базою даних.

У тезі provider може бути також вказаний провайдер JPA. Це клас, який буде створювати фабрику менеджерів сутностей, використовуючи свою реалізацію специфікації JPA. Для різних модулів зберігання можна використовувати різні провайдери.

Нижче наведено приклад такого файлу.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
  xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence

          http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <!–Визначення імені persistence модуля -->
   <persistence-unit name="JpaByvoino">
     <!–Визначення класів моделі -->
      <class>model.Mark</class>
      <class>model.Student</class>
      <class>model.Subject</class>
      <properties>
       <!–Визначення властивостей, необхідних для підключення до БД -->
        <property name="javax.persistence.jdbc.driver"
                  value="com.mysql.jdbc.Driver" />
        <property name="javax.persistence.jdbc.url"
                          value="jdbc:mysql://localhost:3306/mysqldbJpa"
/>
        <property name="javax.persistence.jdbc.user" value="taras" />
        <property name="javax.persistence.jdbc.password" value="1971" />
        <property name="javax.persistence.jdbc.createDatabaseIfNotExist"
                  value="true" />
      </properties>
   </persistence-unit>
</persistence>
```

Слід зазначити, що в каталозі META-INF можна ще розмістити файл orm.xml, у якому вказують параметри відображення для сутностей і їх полів, використовуючи спеціальні теги. У цьому випадку анотації будуть не потрібні. Але цей спосіб не набув широкої популярності у програмістів. Анотаціями користуватися зручніше.

4.2.4 **Реалізація ORM**

Об’єктно-реляційне відображення передбачає наявність двох складових – бази даних та об’єктної моделі цієї бази у вигляді сукупності Java класів, які дозволяють створювати звичайні Java об’єкти, так звані POJO (Plain Old Java Object). Окрім того, ці класи мають бути відповідним чином проанотовані за допомогою анотацій пакету javax.persistence.

Кожна із названих складових може бути отримана із іншої за допомогою відповідних засобів. Тобто, маючи базу даних, можна отримати відповідну їй об’єктну модель. І навпаки, на основі об’єктної моделі можна згенерувати базу даних.

4.2.5 **Особливості створення класів об’єктної моделі**

Як вже було сказано, класи об’єктної моделі – це звичайні класи, які мають бути відповідним чином проанотовані. Окрім того ці класи мають відповідати

умовам, що перелічені у таблиці 4.1.

Таблиця 4.1 – Вимоги до класів об’єктної моделі

1. Наявність конструктора за замовчуванням (конструктор без параметрів)	Специфікація JPA вимагає, щоб класи моделі мали конструктор за замовчуванням, тобто конструктор без параметрів, який може бути public або protected. Враховуючи те, що компілятор JAVA автоматично створює конструктор за замовчуванням, коли не визначене інших конструкторів, це обмеження діє тільки на класи, які визначають і додаткові конструктори.
2. Класи не final	Класи моделі не можуть бути final. Також не можуть бути final методи цих класів. Це пов’язано з тим, що провайдер, що забезпечує відображення, створює свої власні класи, що успадковують класи моделі.
3. Наявність полів ідентифікації	Всі класи моделі повинні містити одне або декілька полів, які разом формують унікальний ключ для сутності. Ці поля називаються identity або primary key.
4. Обмеження спадкування	Класи моделі не можуть успадковуватися від деяких класів, які залежні від операційної системи, таких, наприклад, як java.net.Socket або java.lang.Thread. Якщо клас моделі успадковується від класу, що не належіть моделі, тоді поля цього суперкласу не можуть бути збережені. Всі класи в дереві спадкування повинні мати один і той же тип ідентифікації.

4.2.6 **Анотації для класів моделі**

Усі анотації можна поділити на анотації для класів сутностей та анотації для полів. Основні анотації для класів сутностей розглядаються нижче.

4.2.6.1 Анотація **@Entity**

Ця анотація свідчить, що даний клас відповідає сутності, яка зберігається в базі даних. Необхідно вказувати дану анотацію для кожного такого класу.

Властивість name, що використовується для звернення до сутності у запитах можна не вказувати. За замовчуванням ім'ям сутності є коротке ім'я класу, наприклад:

```
@Entity
public class Group {
   …
}

@Entity(name="Student")
public class Student {
   …
}
```

4.2.6.2 Анотація **@Table**

Вказує ім'я таблиці, де будуть зберігатися сутності. Також можна вказати і ім'я схеми в базі даних. Не є обов'язковою анотацією. За замовчуванням використовується ім’я таблиці, яке відповідає короткому імені класу.

Приклад:

```
@Table(name="TABLE_GROUP", schema="DEFAULT_SCHEMA")
public class Group {
   …
}
```

4.2.7 **Анотації для полів моделі**

За замовчуванням, в JPA усі поля сутностей, які є примітивними типами (int, float, ...), або будь-яким іншим об'єктом, що реалізує інтерфейс Serializable, будуть зберігатися в базі даних в колонках, що відповідають іменам полів. Поля, визначені як static, transient, final заноситися до бази не будуть.

Для кожного поля додатково можна вказати назву колонки або спосіб отримання даних. Можна визначити що поле є ключем (id), чи може його взагалі не потрібно зберігати в базі даних. Також за допомогою анотацій можна вказати на зв'язки об'єкта з іншими об'єктами.

Нижче перелічені основні анотації полів.

4.2.7.1 Анотація @Transient

Вказує, що дане поле не буде зберігатися в базі даних, наприклад:

```
@Transient
private int someInfo;
```

4.2.7.2 Анотація **@Id**

Вказує, що дане поле є ключовим, ідентифікатором сутності (primary key), наприклад:

```
@Id
private int id;
```

4.2.7.3 Анотація @Generated

Вказує, що значення ключового поля буде автоматично генеруватися. Спосіб генерації можна вказати у властивості strategy. Допустимі значення:

AUTO – за замовчуванням. Автоматичний вибір стратегії генерації ключів;
IDENTITY – використання стратегії бази даних;
SEQUENCE – використання джерела даних для алгоритму послідовності;
TABLE – використання окремої таблиці для алгоритму послідовності.

Наприклад:

```
@Id
@GeneratedValue (strategy = GenerationType.IDENTITY)
private int id;
```

4.2.7.4 Анотація **@Temporal**

Ця анотація може використовуватися для полів типу java.util.Date, java.util.Calendar з метою уточнення представлення часу. Дані можуть бути представлені як DATE, TIME або TIMESTAMP (тобто тільки дата, тільки час або і те і те), наприклад:

```
@Temporal(TIMESTAMP)
private Calendar startDate;

@Temporal(DATE)
private Calendar birthDate;
```

4.2.7.5 Анотація **@Basic**

Вказує, що дане поле буде зберігатися в базу даних. Зазвичай, дану анотацію не вказують, крім тих випадків, коли необхідно вказати додаткові параметри відображення. Параметр fetch дозволяє вказати стратегію вибірки даних (EAGER - за замовчуванням, дані вибираються відразу в запиті; LAZY - дані вибираються при першому зверненні до них). Властивість optional дозволяє вказати, чи дозволені null значення в стовпці таблиці і в полі відповідно. За замовчуванням optional = true, що означає, що null значення дозволені.

Наприклад:

```
@Basic
private String fio;

@Basic(fetch=LAZY,optional = false)
private String name;
```

4.2.7.6 Анотація **@OneToOne**

Якщо об'єкт А посилається на один об'єкт В, а той у свою чергу тільки на один об'єкт А, тоді це відношення один-до-одного. Для вказівки такого зв'язку з обох сторін відносини використовується дана анотація. Можливі властивості даної анотації cascade, fetch, mappedBy, optional такі ж, як і в попередніх і наступних анотаціях.

Наприклад:

```
@OneToOne (mappedBy="user", cascade=REMOVE, optional = false)
private Address addr;
```

4.2.7.7 Анотація @ManyToOne

Коли об'єкт А1 містить посилання на об'єкт В, і об'єкт А2 теж може посилатися на В, тоді отримуємо ставлення багато-до-одного. Для вказівки такого зв'язку з боку «багато» використовується дана анотація. Властивості fetch і optional такі ж, як і для попередньої анотації. Властивість cascade, яке вказує поведінка при каскадному доступі до об'єкта, може приймати наступні значення:

PERSIST - зберігати зв'язаний об'єкт при збереженні поточного;

MERGE - приєднувати зв'язаний об'єкт при приєднанні поточного до менеджера;

REMOVE - видаляти зв'язаний об'єкт при видаленні поточного;

REFRESH - оновлювати стан пов'язаного об'єкта при оновленні стан поточного;

ALL - виконувати всі каскадні зміни зі зв'язковим об'єктом.

Наприклад:

```
@ManyToOne(cascade={ PERSIST, MERGE, REMOVE }, fetch = LAZY)
private Group group;
```

4.2.7.8 Анотація @OneToMany

Якщо об'єкт А посилається на кілька об'єктів В, а кожен В посилається лише на один об'єкт А, тоді ми маємо зв'язок один-до-багатьох. Для позначення такого зв'язку з боку «один» використовується дана анотація. Якщо з іншого боку теж є зв'язок (тобто в об'єкті В є посилання на А що визначена анотацією ManyToOne), тоді необхідно вказати, що даний зв'язок двунаправленная. Для цього використовується властивість mappedBy, в якому вказується ім'я поля на іншій стороні (тобто в об'єкті В), яке посилається на даний клас. Так само як і для попередньої анотації тут можливі властивості cascade і fetch.

Наприклад:

```
@OneToMany(mappedBy="group", cascade=ALL)
private Set<Student> students;
```

4.2.7.9 Анотація @ManyToMany

Якщо об'єкт А посилається на кілька об'єктів В, а кожен В може посилається на кілька об'єктів А, тоді це відносини багато до багатьох. Для вказівки такого зв'язку з обох сторін відносини використовується дана анотація. Можливі властивості даної анотації cascade, fetch, mappedBy такі ж, як і в попередніх анотаціях.

Наприклад:

```
@ManyToMany (mappedBy="st", fetch = LAZY)
private List<Student>;
```

4.2.7.10 Анотація **@ OrderBy**

За замовчуванням, результат запиту до бази даних не впорядковується. Для сортування результатів використовується дана анотація. За замовчуванням відбувається сортування по первинному ключу. Але у параметрі до даної анотації можна вказати назву поля, за якоим треба сортувати, а також напрямок сортування (ASC - зростаючий порядок; DESC - регресний порядок).

Наприклад:

```
@OneToMany
@OrderBy("fio ASC")
private Set<Student> students;
```

4.2.7.11 Анотація @NamedQuery

Ця анотація визначає ім’я та текст запиту до таблиці на мові JPQL.
Наприклад:

```
@NamedQuery(name = "Stud.findAll", query = "SELECT s FROM Stud s")
```

Далі, у програмі до цього запиту можна звертатися, використовуючи його ім’я, наприклад:

```
Query q = em.createNamedQuery("Stud.findAll");
```

4.2.8 **Створення об’єктної моделі бази даних**

Об’єктну модель можна створити двома способами.

Якщо таблиці бази даних ще не створено, тоді слід створити класи, що відповідають таблицям БД і прописати у цих класах необхідні анотації. Цю роботу можна більш ефективно виконати за допомогою Eclipse, створюючи класи JPA Entity. Маючи модель, можна згенерувати таблиці бази даних.

Якщо базу даних вже створено, то можна за допомогою JPA згенерувати об’єктну модель.

Розглянемо, як приклад, модель для бази, де ми маємо таблицю Graduate (випускник), яка посилається на таблицю Cathedra (кафедра).

Нижче наведено тексти класів, які згенеровано для таблиць цієї бази даних. У текстах, для скорочення, не наводяться пустий конструктор та геттери і сеттери для полів, що відповідають колонкам таблиць таблиць.

У класі Cathedra було згенеровано додаткове поле graduates для збереження списку випускників цієї кафедри

```
/**
 * The persistent class for the CATHEDRA database table.
 *
 */
@Entity
@NamedQuery(name="Cathedra.findAll",
             query="SELECT c FROM Cathedra c")
public class Cathedra implements Serializable {
  private static final long serialVersionUID = 1L;

  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  private int id;

  private String chief;

  private String name;

  private Object spec;

  private int staff;
```

```
   //bi-directional many-to-one association to Graduate
   @OneToMany(mappedBy="cathedra")
   private List<Graduate> graduates;

   public List<Graduate> getGraduates() {
      return this.graduates;
   }

   public void setGraduates(List<Graduate> graduates) {
      this.graduates = graduates;
   }

   public Graduate addGraduate(Graduate graduate) {
      getGraduates().add(graduate);
      graduate.setCathedra(this);
      return graduate;
   }

   public Graduate removeGraduate(Graduate graduate) {
      getGraduates().remove(graduate);
      graduate.setCathedra(null);
      return graduate;
   }
}

/**
 * The persistent class for the GRADUATE database table.
 *
 */
@Entity
@NamedQuery(name="Graduate.findAll",
            query="SELECT g FROM Graduate g")
public class Graduate implements Serializable {
   private static final long serialVersionUID = 1L;

   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   private int id;

   private int ball;

   private int finish;

   private String fio;

   //bi-directional many-to-one association to Cathedra
   @ManyToOne(cascade={CascadeType.ALL})
   @JoinColumn(name="IDCATHEDRA")
   private Cathedra cathedra;
```

```
}
```

4.2.9 **Робота з persistence об’єктами**

Як уже наголошувалося раніше, JPA дозволяє замінити роботу із записами бази даних на роботу з об’єктами, які прийнято називати persistence об’єктами.

Для управління цими об'єктами використовується менеджер сутностей EntityManager. Примірник даного менеджера можна отримати з класу-фабрики менеджерів. Саму фабрику можна отримати за допомогою статичного методу класу Persistence. До методу створення фабрики потрібно передати ім’я persistence unit, з файлу persistence.xml. Другим, необов’язковим параметром може бути посилання на об’єкт типу Properties, у якому вказані властивості, необхідні для підключення до бази даних. Зазвичай другий параметр не використовується, а параметри підключення до бази данихвизначаються в xml файлі persistence.xml.

Приклад створення Entity менеджера наведено нижче:

```
EntityManagerFactory emf =
          Persistence.createEntityManagerFactory("MyJpaProject");
EntityManager em = emf.createEntityManager();
```

За допомогою менеджера сутностей можна створювати об'єкти виконання запитів (Query) та об'єкти управління транзакціями (EntityTransaction). Він також забезпечує збереження, оновлення та видалення persistence об’єктів.

Інтерфейс менеджера сутностей наведено у вигляді рисунку 4.2.

Менеджер містить кілька методів, які змінюють стан сутностей. Слід зауважити, що після виклику цих методів, зміни не набувають чинності одразу, і дані в базі даних не змінюються. Всі зміни набудуть чинності після підтвердження транзакції, тобто виклику методу commit на об'єкті EntityTransaction. Відповідно, для скасування всіх змін викликається метод rollback.

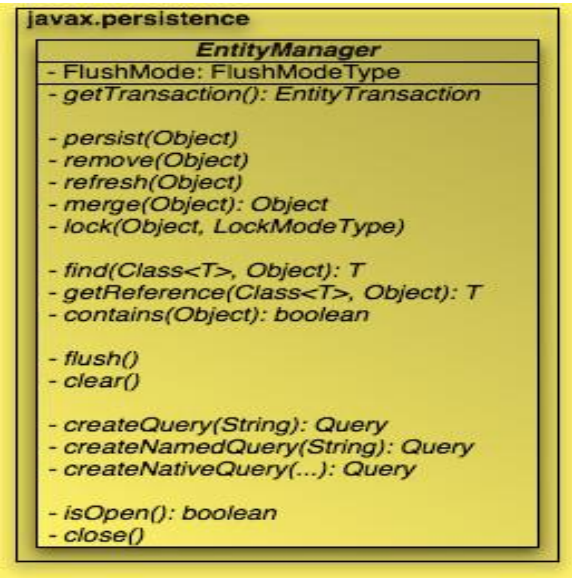

Рисунок 4.2 – Інтерфейс менеджера сутностей

4.2.9.1 Методи менеджера сутностей

Метод **public void persist(Object entity)** забезпечує збереження нового об'єкта в базі даних. Після цього об'єкт переходить зі стану тимчасовий (transient) у стан керований (managed). Якщо об'єкт перебував у стані від'єднаний (detached), тоді буде викинуто виняток IllegalArgumentException. У стан від'єднаний об'єкт потрапляє після збереження, підтвердження транзакції і закриття сеансу взаємодії з менеджером сутностей (тобто виклику методу close).

Метод **public void remove(Object entity)** забезпечує видалення об'єкту з бази даних. Після цього об'єкт переходить зі стану керований (managed) у стан видалений (removed). Якщо об'єкт перебував у стані від'єднаний (detached), тоді буде викинуто виняток IllegalArgumentException.

Метод **public void refresh(Object entity)** перевіряє, чи дані об'єкта відповідають тим, що записані в базі даних. Якщо дані в базі змінилися за час життя даного об'єкта, то вони будуть оновлені в об'єкті. Даний метод зберігає об'єкт у стані керований (managed). Якщо об'єкт перебував у стані від'єднаний (detached), тоді буде викинуто виняток IllegalArgumentException.

Метод **public void merge(Object entity)** приєднує об'єкт, який знаходиться в стані від'єднаний (detached) до поточного менеджеру сутностей і переводить об'єкт у стан керований (managed). Відповідно, якщо дані в об'єкті були змінені, тоді ці зміни набудуть чинності при підтвердженні транзакції. Якщо об'єкт перебував у стані видалений (removed), тоді буде викинуто виняток IllegalArgumentException.

Метод **public void lock(Object entity, LockModeType mode)** блокує об'єкт, використовуючи заданий режим. Режими блокування READ дозволяє іншим транзакціям читати об'єкт, але не змінювати його. Режим WRITE забороняє іншим транзакціям не тільки змінювати об'єкт, але й читати.

4.2.9.2 Життєвий цикл об’єктів сутностей JPA

В процесі життєвого циклу JPA сутність може перебувати у чотирьох станах - новий, керований, видалений та відокремлений

Діаграма станів, у яких може перебувати об’єкт під час виконання перелічених операцій наведена на рисунку 4.3.

Коли об'єкт сутності створюється, його стан – New (у Hibernet - Transient). У такому стані об’єкт не пов’язаний з менеджером сутностей і не має представлення в базі даних.

Об'єкт сутності стає керованим (Managed), коли він зберігається в базі даних за допомогою методу persiste.

У цьому стані об'єкт асоціюється з контекстом постійності (persistence context) менеджера сутності.

Об’єкти сутності, отримані з бази даних методом find менеджера сутностей, також перебувають у керованому стані.

Якщо об'єкт керованої сутності змінюється в межах активної транзакції, оновлення розповсюджується в базу даних при фіксації транзакції.

Об'єкт сутності змінює свій стан з керований на видалений (Deleted) за допомогою методу remove менеджера сутності в межах активної транзакції. Він фізично видаляється з бази даних під час фіксації транзакції.

Останній стан, відокремлений (Detached), представляє об’єкти сутності, які були від’єднані від контексту постійності менеджера сутності.

Від’єднання може відбутися після того, як об’єкт замінили на інший з таким самим id. Змінений об’єкт можна повернути у контекст методом merge.

Метод clear менеджера сутностей також від’єднує усі об’єкти контекст постійності. Від’єднання відбувається і в разі закриття фабрики менеджерів стностей.

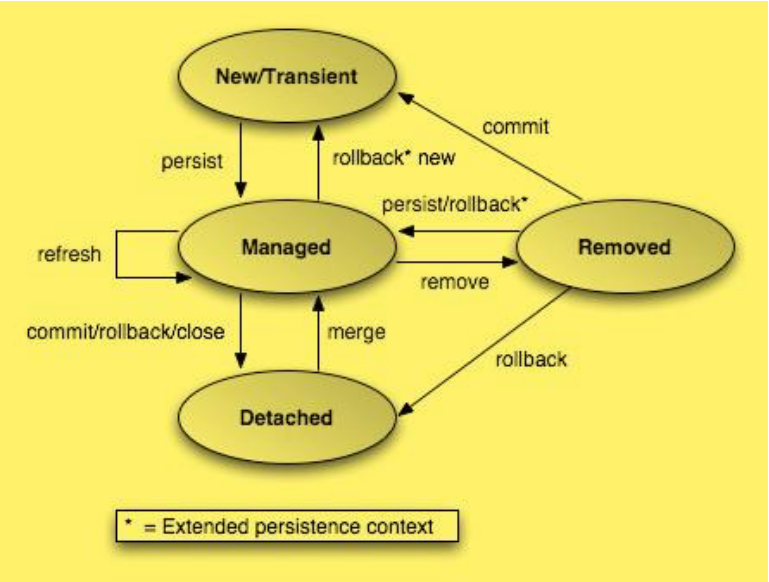

Рисунок 4.3 – Діаграма станів сутностей

4.2.10 **Інтерфейс javax.persistence.Query**

Менеджер сутностей має декілька методів, що починаються словом create і місстять слово Query. В якості параметра до цих методів передається рядок символів, що містить формулювання запиту до бази даних. Усі ці методи повертають об’єкти типу javax.persistence.Query.

JPA дозволяє використовувати декілька варіантів мов формулювання запитів, зокрема SQL.

Щоб отримати результат запиту слід скористатися методом getResultList, який повертає об’єкт типу java.util.List.

Робити запити на мові SQL не завжди зручно, бо результати повертаються у вигляді списку, до складу якого входять не об’єкти моделі, а масиви, що містять значення атрибутів об’єктів.

4.2.10.1 Запити на SQL

У наведеному нижче прикладі для створення об’єкту Query використовується метод createNativeQuery, якому передається запит на мові

SQL. Змінна em містить посилання на об’єкт EntityManaager.

```
Query q = em.createNativeQuery("* from Student");
List<Object[]> list = q.getResultList();
for (Object[] obj : list) {
     System.out.println(Arrays.toString(arr));
}
```

В даному випадку ми отримаємопросто переліки значень атрибутів об’єктів і не будемо знати, яким полям відповідають ці значення.

Більш прийнятний результат можна отримати, якщо налаштувати запит на представлення результатів у вигляді карти. Але це можливо тідьки для запитів, де вказується конкретний перелік полів, а не зірочка.

```
Query q = em.createNativeQuery("select id, name from Student");
q.setHint(QueryHints.RESULT_TYPE, ResultType.Map);
List<Map<String, Object>> list = q.getResultList();
for (Map<String, Object> map : list) {
     System.out.println(map);
}
```

Як результат тут ми отримаємо перелік карт, де в якості ключа буде назва поля.

4.2.10.2 Запити на JPQL

Для формування запитів краще використовувати мову JPQL. Нижче наведено приклад реалізації запиту на цій мові. Тут для створення запиту використовується метод createQuery, а запит повертає список об’єктів Student. Зверніть увагу, що JPQL оперує поняттям класу, а не таблиці бази даних. У запиті визначається об’єкт (Student s) і пропонується вибрати усі такі об’єкти із бази.

```
Query q = em.createQuery("select s from Student s");
List<Student> list = q.getResultList();
for (Student obj : list) {
   System.out.println("id=" + obj.getId() + ", name= " + obj.getName());
}
```

4.2.11 **Короткі відомості про JPQL**

У наведених вище класах, які були згенеровані JPA, є цікава особливість. Там, після анотації @NamedQuery, наведено текст запиту до бази на вибір усіх записів з відповідної таблиці та визначено його ім’я. Запит написано мовою JPQL (Java persistence query language) – це мова створення запитів до бази даних через об’єктну модель. Такий запит має назву іменований.

Запити можна створювати і безпосередньо у тексті програми за допомогою методу createQuery() менеджера сутностей.

Докладний опис JPQL можна знайти у посібнику за адресою:
http://docs.oracle.com/javaee/6/tutorial/doc/index.html
Інформація про JPQL знаходиться у пункті 34 розділу Part VI – Persistence.
Деякі приклади використання JPQL можна знайти за адресою

https://www.tutorialspoint.com/jpa/jpa_jpql.htm

Тут ми дамо тільки коротку довідку про цю мову.

Запит до бази даних на мові JPQL є об'єктом типу Query і може бути одним з трьох типів:

– запит на вибірку даних;
– запит на оновлення даних;
– запит на видалення даних.

4.2.11.1 Запит на вибірку

Запит на вибірку може включати шість частин, або розділів: SELECT, FROM, WHERE, GROUP BY, HAVING і ORDER BY. Перші дві, SELECT і FROM, є обов'язковими, інші ні.

Частина SELECT містить ідентифікаційну змінну (умовну назву результату, що повертається запитом), наприклад, SELECT x. Можна також використовувати виклик операції, яка буде обробляти результат запиту, наприклад, SELECT UPPER(x.name).

Частина FROM оголошує тип ідентифікаційної змінної, яка визначена у частині SELECT. Можуть бути оголошені і інші змінні на які буде посилатися частина WHERE.

Частина WHERE може використовувати параметри двох типів – по індексу і по імені. Значення параметрам присвоюються за допомогою методу setParameter(), який надсилається запиту.

Індексований параметр позначається знаком питання і номером.

```
Query q = em.createQuery("select m from Mark m where m.student.name = ?1");
q.setParameter(1, "stud3");
```

Іменований параметр позначається двокрапкою, після чого йде назва параметру.

```
Query q = em.createQuery("select m from Mark m where m.student.name = :a");
q.setParameter("a", "stud3");
```

Частини GROUP BY, HAVING і ORDER BY по суті схожі с такими ж у мові SQL.

Отримати результат запиту можна за допомогою методу getResultList(), що надсилається запитові. Результат повертається у вигляді списку.

4.2.11.2 Запит на видалення даних

Запит на видалення забезпечує групову операцію з видалення сутностей. Запит може включати дві частини – DELETE FROM та WHERE. Друга частина не є обов'язковою і її структура розглядалася у попередньому пункті.

Частина DELETE FROM містить назву класу та (можливо) назву змінної), яка буде використовуватися у частині WHERE.

Активізувати запит можна викликом методу executeUpdate(). Наприклад:

```
Query q = em.createQuery("DELETE from Mark x WHERE
x.student.name=?1");
```

```
q.setParameter(1, "stud2");
q.executeUpdate();
```

4.2.11.3 Запит на оновлення даних

Запит на оновлення забезпечує групову операцію із оновлення сутностей. Запит може включати три частини – UPDATE, SET та WHERE. Третя частина не є обов'язковою і вже розглядалася.

Частина UPDATE містить назву класу та назву змінної, яка буде використовуватися у частинах SET та WHERE. Наприклад:

```
Query q = em.createQuery("Update Mark x
SET x.mark = 99 WHERE x.student.name=?1");
q.setParameter(1, "stud1");
q.executeUpdate();
```

4.2.12 **Транзакції**

Зв'язок між контекстом постійності і базою даних реалізується за допомогою транзакцій.

Для визначення ділянки коду, що має розглядатися як транзакція використовують об'єкт типу EntityTransaction, який можна отримати за допомогою EntityManager через метод getTransaction().

Початок транзакції позначається методом begin(), який надсилається об'єкту типу EntityTransaction.

Можливі два шляхи завершення транзакції. Нормально, транзакція має завершуватися викликом методу commit(). У цьому випадку змінені дані заносяться до бази даних. Якщо ж під час виконання транзакції виникла виключна ситуація, то транзакція має завершуватися викликом методу rollback(). У цьому випадку усі зміни даних скасовуються.

Нижче наведено приклад використання транзакції, у межах якої змінюється ім'я студента.

```
//Створюємо менеджера за допомогою фабрики
EntityManager em = emfactory.createEntityManager();
//Створюємо транзакцію
EntityTransaction  transaction = em.getTransaction();
try {
//Починаємо транзакцію
  transaction.begin();
  //Знаходимо у базі студента за його id
  Student obj = em.find(Student.class, student.getId());
  //Змінюємо ім'я студента
  obj.setName(newName);
  //Закриваємо транзакцію
  transaction.commit();
} catch (Exception e) {
  e.printStackTrace();
  //Відміна транзації у разі виключної ситуації
  transaction.rollback();
```

```
} finally {
   em.close();
}
```

4.3 Порядок виконання лабораторної роботи

Для виконання цієї лабораторної роботи потрібен Eclipse IDE for Java EE Developers. Скачати Eclipse можна за такою адресою:

https://www.eclipse.org/downloads/packages/

Слід враховувати, що нові версії Eclipse містять плагін з пакетами Java останніх версій, які потрібні Eclipse для роботи. За замовчуванням цей варіант jre використовується і для проєктів, що створюються. Але з цими версіями Eclipse не зовсім коректно працюють деякі бібліотеки та фреймворки.

Тому для цієї та наступних лабораторних робіт слід скачати Eclipse версії не пізніше 2018-12.

І навіть у цій версії того, щоб використовувати Java 8, треба під час створення проєкту поміняти у властивостях проєкту, надаштуванн jre і компілятор.

4.3.1 Відкриття JPA перспективи

Після відкриття Eclipse доцільно одразу відкрити «перспективу» JPA. Для цього можна скористатися іконкою у правому верхньому куті робочої області, або скористатися функцією головного меню Window→ Perspective→ OpenPerspective →JPA.

На рисунку 4.4 показано можливий вигляд перспективи JPA. Зверніть увагу на появу нових панелей у робочій області.

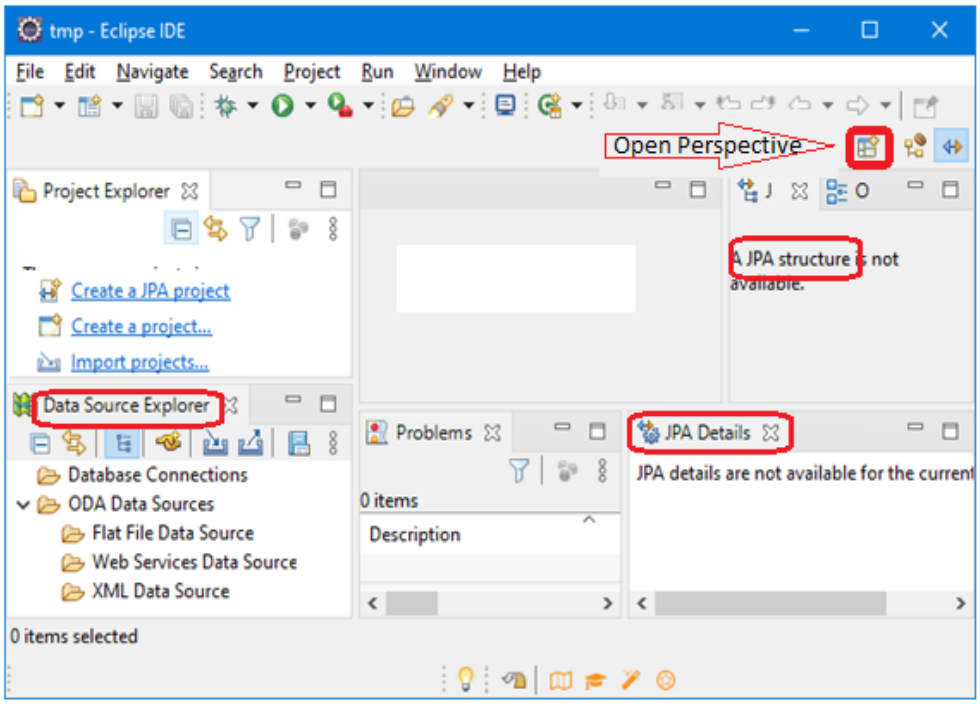

Рисунок 4.4 – Вигляд перспективи JPA

Ліворуч знизу знаходиться панель Data Source Explorer. Ця панель містить інструменти, які дозволяють створити з'єднання з базою даних і працювати з нею. Зокрема можна надсилати SQL запити, переглядати результати цих запитів, переглядати та редагувати вміст таблиць.

Панелі JPA structure та JPA Details забезпечують налаштування класів сутностей, з якими ми далі познайомимося.

4.3.2 **Створення JPA проєкту**

Далі створімо новий JPA проєкт, рисунок 4.5.

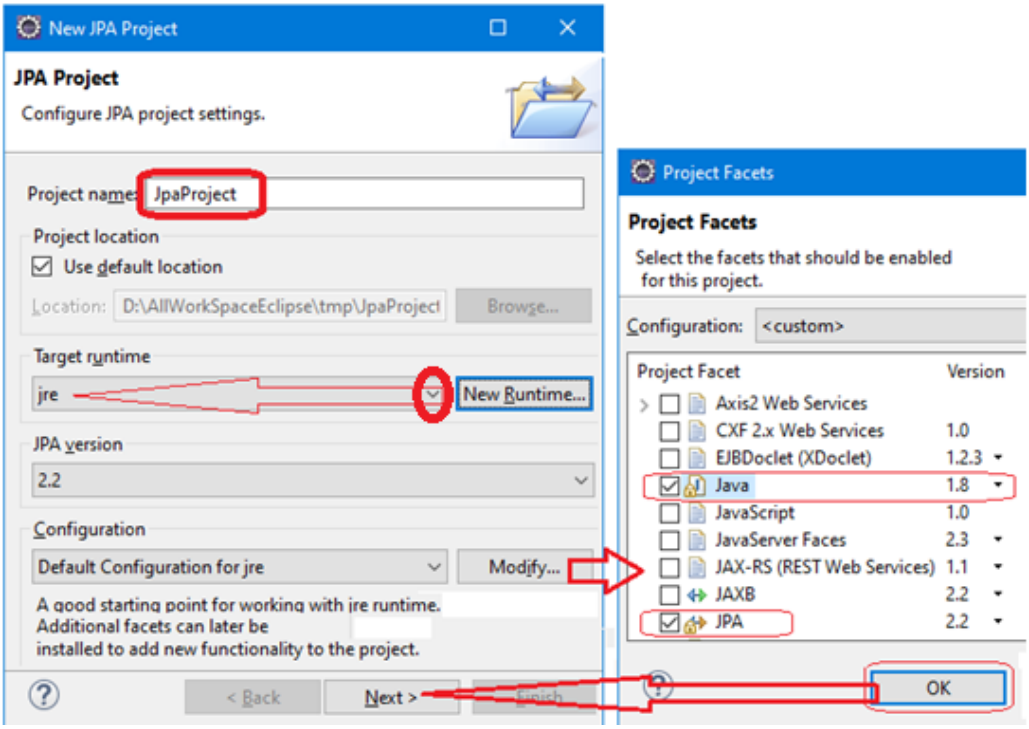

Рисунок 4.5 – Діалог створення JPA проєкту

У діалозі створення проєкту вводимо його ім'я. **До назви проєкту має входити прізвище студента, наприклад, JpaByvoino**.

Із списку Target runtime вибираємо jre або jdk.

Далі через кнопку Modify змінюємо конфігурацію jre, вибравши із переліку версій Java версію 1.8. Після цього натискаємо OK.

Далі у діалогах вибираємо Next , ще раз Next і потрапляємо у завершальний діалог, рисунок 4.6.

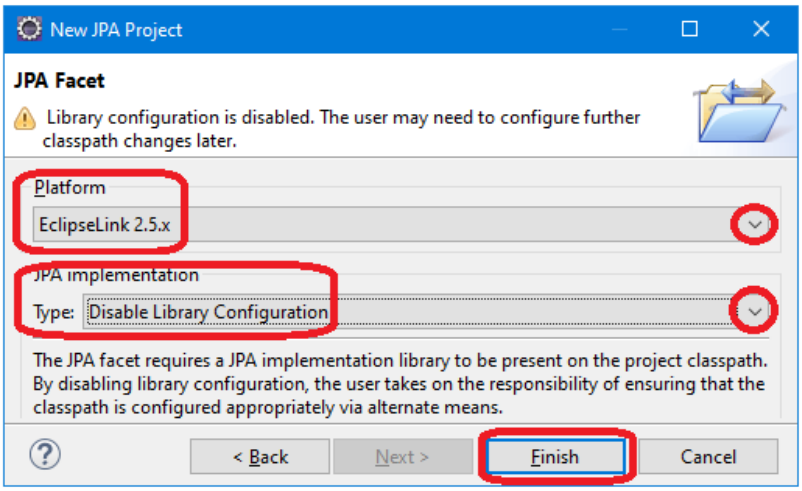

Рисунок 4.6 – Фрагмент завершального діалогу створення JPA проєкта

Із списку Platform вибираємо EclipseLink. Цей вибір означає, що ми будемо створювати проєкт на платформі EclipseLink, яка надає більше можливостей ніж базова Generic.

Із списку типів реалізації JPA виираємо Disable Library Connection. Цей вибір означає, що необхідні файли бібліотек будуть включені до Build Path проєкту пізніше.

Інші елементи діалогу не використовуємо.

Натискаємо OK. Проєкт має з'явитися на закладці проєктів.

4.3.3 **Файл persistence.xml**

Відкрийте папку src.META-INF або JPA Content і познайомтесь з файлом persistence.xml. Для цього треба клікнути по файлу і відкрити закладку source.

Цей файл використовується для зберігання налаштувань проєкта.

Зверніть увагу на ім'я persistence-unit. Це ім'я знадобиться для створення фабрики менеджерів сутностей. Воно, зазвичай, співпадає з назвою проєкту.

4.3.4 **Копіювання бази даних у проєкт і створення з'єднання з нею**

Скопіюємо або імпортуємо до нашого проєкту папку з базою даних із проєкту до лабораторної роботи 2. Взагалі база може бути де завгодно. Але для зручності перенесення проєкту і бази з одного комп'ютера на інший і спрощення перевірки ми розташуємо базу в проєкті.

Додаймо до buildPath проєкту посилання на драйвер derby.jar, той самий, що використовувався для створення бази у другій лабораторній роботі. Інакше можуть виникнути проблеми із з'єднанням.

Далі на закладці DataSourceExplorer налаштуємо з'єднання з базою даних.

З'єднання з базою даних створюється і існує незалежно від проєкту. Але його можна буде використовувати для налаштування проєкту.

На рисунку 4.7 схематично показані перші кроки підключення до бази даних.

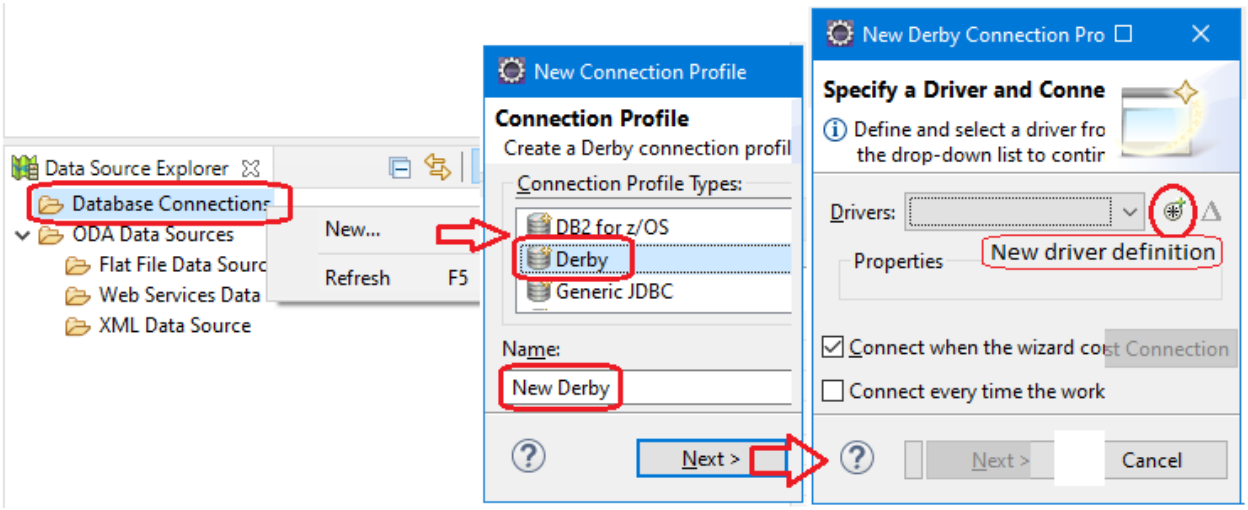

Рисунок 4.7 – Перші кроки створення з’єднання з БД

Через функцію New контекстного меню пункту Database Connection потрапляємо діалог створення з’єднання.

У цьому діалозі слід вибрати СУБД (не базу даних) з якою налаштовується з’єднання і задати ім’я цього з’єднання. У нашому випадку в якості СУБД виступає Derby, ім’я цього з’єднання залишимо таким, яке пропонує система – New Derby, але може бути і іншим

Далі потрапляємо у діалог вибору драйвера для СУБД. Його можна вибрати із списку, ящо драйвер вже створювався. А якщо драйверів нема, то треба перейти у діалог визначення драйвера (іконка New driver difinition).

Наступні кроки показані на рисунку 4.8.

У діалозі, що з’явився, на закладці Name/Type вибираємо тип драйвера та налаштовуємо його ім’я і переходимо на закладку Jar List.

На закладці Jar List за допомогою кнопки Clear All прибираємо ім’я файлу драйвера, що встановлено за замовчуванням (derby.jar).

Далі, через кнопку Add JAR/Zip заходимо у діалог пошуку файлу драйвера. Там має з’явитися стандартний діалог для вибору файла. Після вибору файлу його повне ім’я має з’явитися на панелі і у списку драйверів. Драйвер має бути той самий з яким створювалася база і який підключено до проєкту.

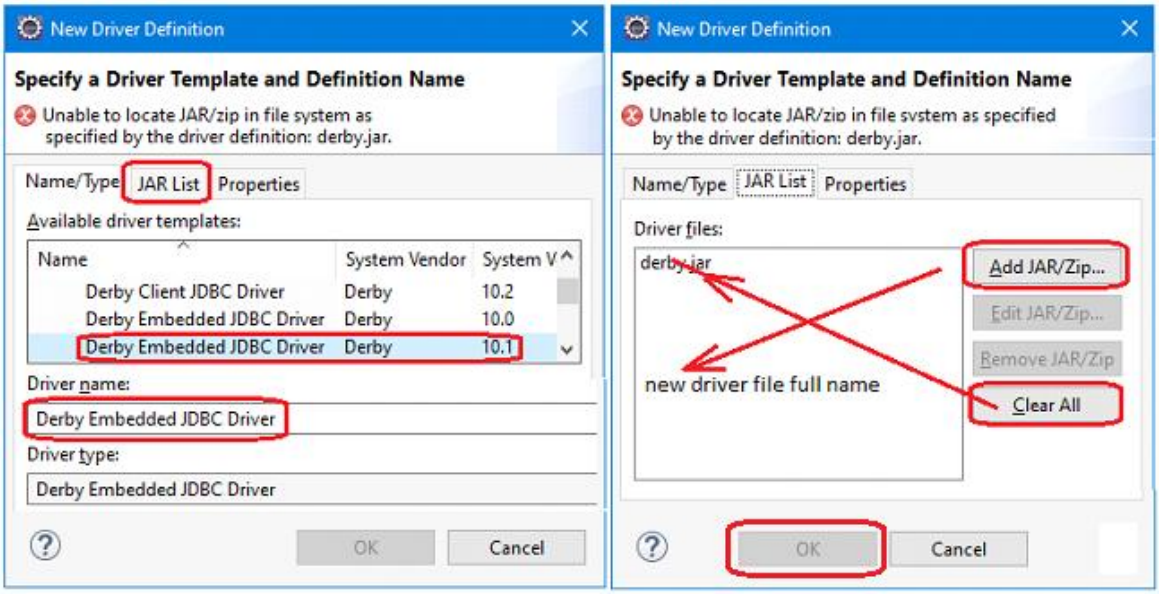

Рисунок 4.8 – Кроки вибору драйвера для з’єднання з БД

Після натискання кнопки OK Eclipse показує діалог налаштування параметрів бази даних і пропонує свій варіант створення нової бази, яка буде використовувати цей драйвер, рисунок 4.9. Але нова база нам не потрібна. Тому скористаймося кнопкою Browse і знайдемо шлях до теки db з нашою базою даних (вона має бути у проєкті).

Тут слід мати на увазі, що для Data Source Explorer за замовчуванням використовує шлях до папки з Eclipse, а не проєкту. Тому тут треба вказувати повний шлях до бази.

Після цього введемо ім’я користувача і пароль.

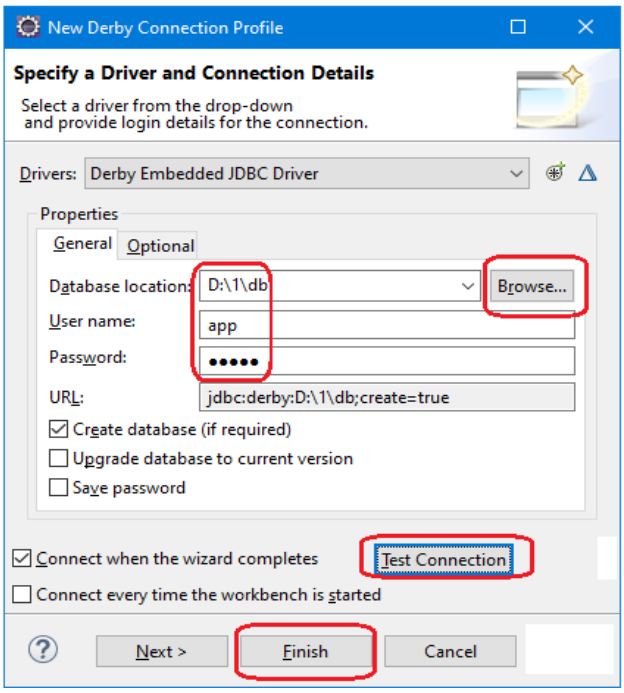

Рисунок 4.9 – Налаштування параметрів БД

Далі можна перевірити коректність з'єднання з базою через кнопку Test

Connection.

Якщо з’єднання було успішним, то після закриття діалогу на панелі Database Connection має з’явитися закладка з ім’ям з’єднання. В нашому випадку це New Derby. Розкривши цю закладку можна зайти у базу і переглядати та редагувати її, рисунок 4.10.

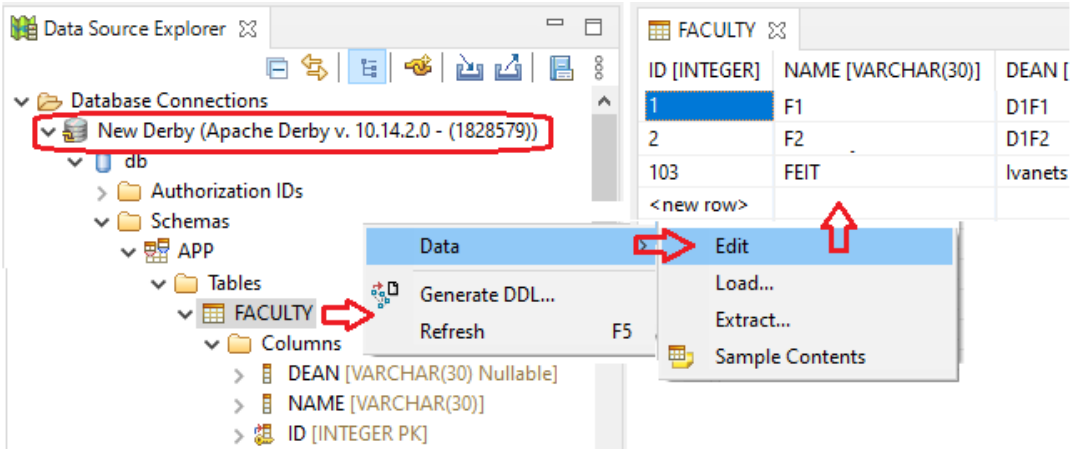

Рисунок 4.10 – Перегляд і редагування БД

Інформацію про з’єднання Eclipse зберігає в рабочій області і після виключення.

У контекстному меню з’єднання, рисунок 4.11, ми бачимо можливості використання цього з’єднання.

З’єднання можна включати, виключати, тестувати. Можна зайти у вікно ScrapBook і там надсилати SQL запити до базі.

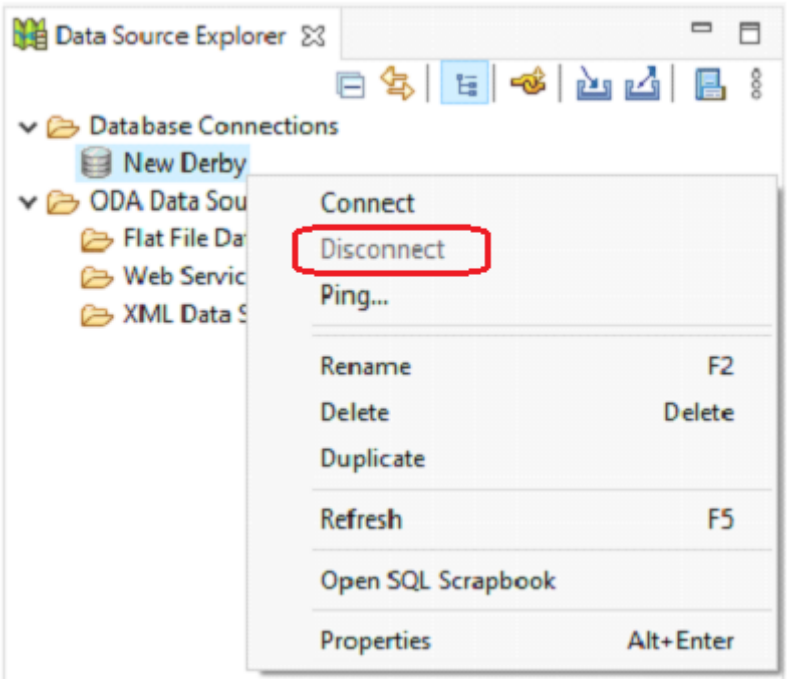

Рисунок 4.11 – Перелік операцій із з'днанням з БД

4.3.5 **Внесення інформації про БД у файл percistence.xml**

Додаймо до файлу percistence.xml інформацію, що необхідна для підключення до бази даних. Ця інформація зберігається у частині xml-файлу з тегом properties.

Розділ properties можна створити у діалозі перегляду та налаштування xml-файлу, рисунок 4.12, на закладці Connection.

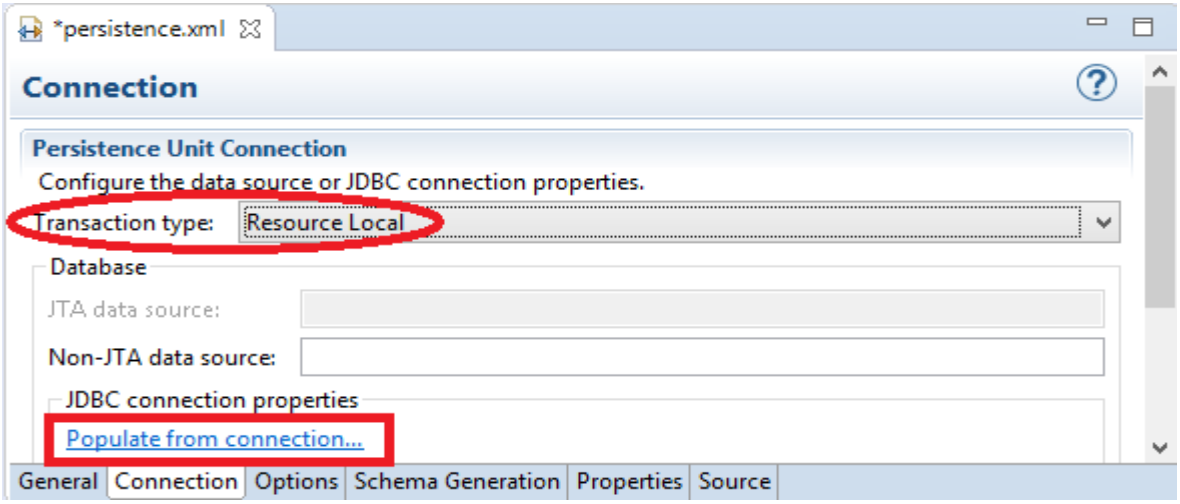

Рисунок 4.12 – Діалог налаштування xml-файлу

Спочатку треба налаштувати поле «Transaction type» так, як показано на рисунку 4.12. Далі натиснути посилання «Populate from connection». Як результат до xml-файлу буде перенесено параметри з'єднання з базою даних.

```
<properties>
<property name="javax.persistence.jdbc.url"
                value="jdbc:derby:D:\1\db;create=true"/>
  <property name="javax.persistence.jdbc.user" value="app"/>
  <property name="javax.persistence.jdbc.password" value="12345"/>
  <property name="javax.persistence.jdbc.driver"
                value="org.apache.derby.jdbc.EmbeddedDriver"/>
</properties>
```

Після цього вся необхідна інформація вже зберігається в xml файлі. Але доцільно ще раз перевірити усі параметри xml файлу.

4.3.6 **Підключення до проєкту JSP бібліотек**

Об'єктна модель бази даних – це сукупність класів з JPA анотаціями, що відповідають таблицям бази даних. Для створення і роботи з об'єктною моделлю до проєкту необхідно підключити дві бібліотеки.

Бібліотека javax.persistence-api-2.2.jar забезпечує підтримку об'єктної моделі даних.

Бібліотека eclipselink-2.7.jar забезпечує створення та підтримку роботи об'єктів типу EntityManager, які реалізують роботу з базою даних.

Скачати ці бібліотеки можна з мудла, або за адресами із інтернет.

http://www.java2s.com/Code/Jar/p/Downloadpersistenceapi20jar.htm

http://www.java2s.com/example/jar/e/download-eclipselink270jar-file.html

4.3.7 **Створення об’єктної моделі бази даних**

Після підключення бібліотек до build path проєкта можна перейти до створення моделі.

Якщо з’єднання з базою відключено, його слід відновити.

Далі, у контекстному меню проєкту вибрати функцію JPA Tools→Generate Entity from Tables.

Результатом буде поява діалогу вибору таблиць бази даних, рисунок 4.13.

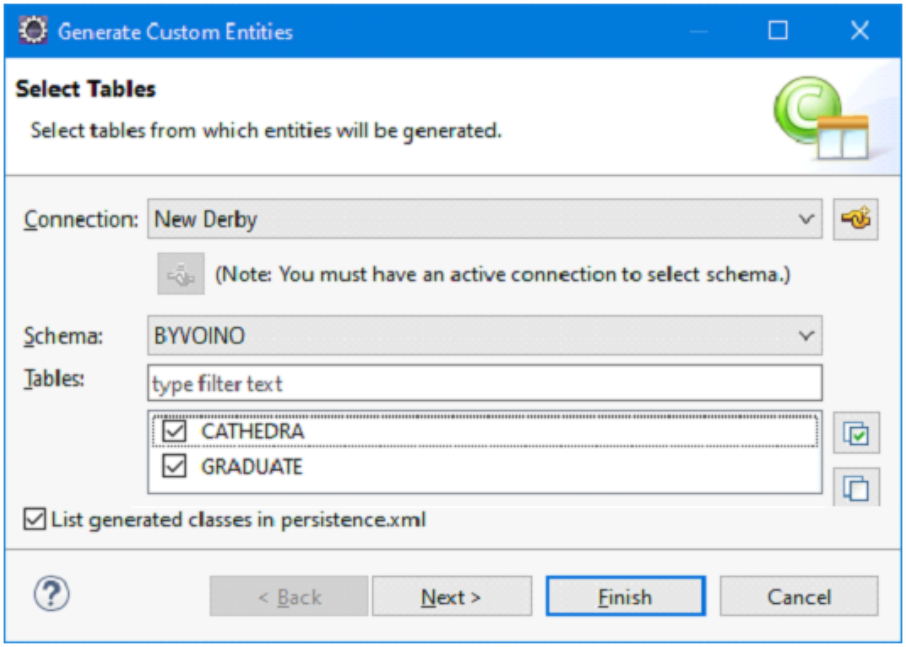

Рисунок 4.13 – Діалог вибору таблиць БД для створення об’єктної моделі

У наступному діалозі налаштуємо каскадне видалення випускників в разі видалення кафедри, рисунок 4.14.

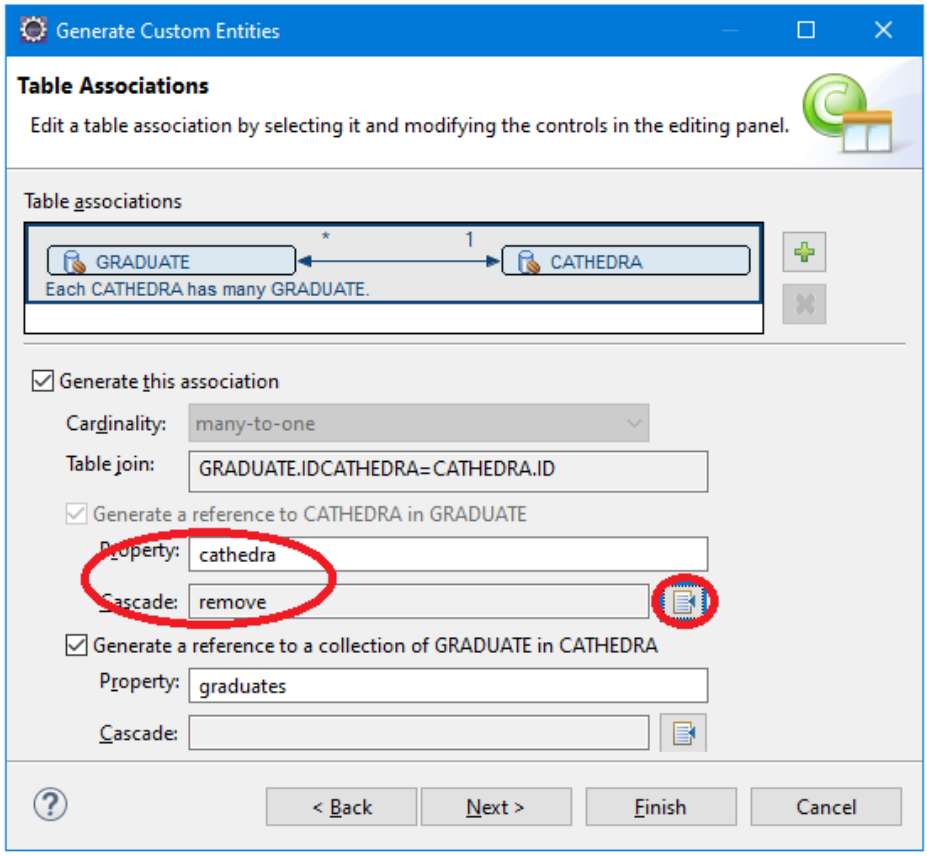

Рисунок 4.13 – Діалог налаштування параметрів зв’язку між таблицями

В наступному діалозі налаштуйте спосіб генерації ключів так, щоб це було identity.

Після завершення налаштувань вибираємо кнопку Finish і отримуємо об’єктну модель. За замовчуванням класи будуть розташовані у папці src.model проєкту.

Посилання на класи мають з’явитися і у xml файлі, у цьому можна пересвідчитися відкривши вікно перегляду xml фалйу.

Після цього connection з базою даних на панелі Data Source Explorer слід відключити, бо може виникнути конфлікт в разі запуску проєкту. Необхідна інформація тепер є в xml-файлі і проєкт буде намагатися створити своє з’єднання..

4.3.8 **Знайомство з об’єктною моделлю**

Після отримання моделі слід докладно проаналізувати створені класи.

Зверніть увагу на анотації перед класом, полями і методами.

У вигляді анотації також реалізовано запити на отримання вмісту відповідної таблиці БД.

В класі для першої таблиці згенеровано методи доступу до списку нащадків класу, що зберігаються у другій таблиці, а також методи додавання і вилученя нащадків.

Зверніть увауу, на панелях JPA Structure та JPA Detail відображається інформація про класи сутностей.

Але у класах не реалізовані методи toString(). Їх слід реалізувати, використовуючи поля, які характеризують даний об'єкт.

Можна створити конмтруктори з полями.

4.3.9 **Робота з базою даних**

Для дослідження можливостей роботи з базою даних через JPA створімо пакет test і розташуймо у ньому декілька тестових класів з методами main().

Слід уважно відноситися до імпорту класів. Класи, що пов'язані з JPA, мають належати до пакету javax.percistence. Списки до пакету java.util.

Нижче наведені приклади реалізації запитів до бази, що складається поки що з двох таблиць – кафедра та випускник.

Вам необхідно реалізувати аналогічні запити до своєї бази даних.

4.3.9.1 Використання іменованого запиту з класу моделі

Як можна побачити, у класах об'єктної моделі автоматично створюються іменовані запити, що повертають усі рядки таблиці. Щоб протестувати такий запит, створімо у пакеті test клас TestNamedQuery з методом main().

Нижче наведено приклад такого методу.

Спочатку створюється фабрика менеджерів. У методі створеня фабрики ім'я persistence unit має точно співпадати з його ім'ям у xml файлі. Значення мають навіть пробіли. У прикладі це «JpaByvoino».

Далі, у методі створюється JPA-менеджер.

Після цього можна створювати запит і реалізовувати його.

Уданому прикладі ми отримуємо список об'єктів типу Cathedra, обробляємо цей список у циклі for, і виводимо на консоль інформацію про випускників кафедри.

Перед запуском метода відключіть з'єднання з базою даних на панелі Database Connection. Програма буде сама підключатися до бази використовуючи файл persistence.xml.

```
public static void main(String[] args) {
   EntityManagerFactory emf =
     Persistence.createEntityManagerFactory("JpaByvoino");
   EntityManager em = emf.createEntityManager();
   Query q = em.createNamedQuery("Cathedra.findAll");
   List<Cathedra> list = q.getResultList();
   for (Cathedra cth : list) {
     System.out.println(cth);
     for(Graduate gr : cth.getGraduates()) {
       System.out.println("\t"+gr);
     }
   }
}
```

4.3.9.2 Використання запиту з параметром

Для цього тесту створімо клас TestParamemterQuery. Метод main цього

класу наведено нижче. У цьому запиті ми отримуємо у вигляді списка інформацію із таблиці Graduate про студентів, які закінчили кафедру у 2011 році.

```
public static void main(String[] args) {
   EntityManagerFactory emf =
   Persistence.createEntityManagerFactory("JpaByvoino");
   EntityManager em = emf.createEntityManager();
   String sq = "select gr from Graduate gr where gr.finish = ?1";
   Query q = em.createQuery(sq);
   q.setParameter(1, 2011);
   List<Graduate> list = q.getResultList();
   for (Graduate gr : list) {
      System.out.println(gr);
   }
}
```

4.3.9.3 Оновлення інформації у базі даних

Створімо клас TestUpdate, у якому реалізуємо зміну ім'я студента із заданим id.

```
public static void main(String[] args) {
   EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("JpaByvoino");
   EntityManager em = emf.createEntityManager();
   Query q = em.createNamedQuery("Graduate.findAll");
   //Output graduates list before update
   List<Graduate> list = q.getResultList();
   for (Graduate gr : list) System.out.println(gr);
   //Update student name with id = 2
   em.getTransaction().begin();
   Graduate grd = em.find(Graduate.class, 2);
   grd.setFio("NewName");
   em.getTransaction().commit();
   //Output graduates list after update
   list = q.getResultList();
   for (Graduate gr : list) System.out.println("\t"+ gr);
}
```

4.3.9.4 Створення нового запису у таблиці

Створімо у пакеті test новий клас TestAddCathedra і в ньому реалізуймо додавання нового запису до таблиці Cathedra без прямого використання запиту.

```
//Створюємо EntityManager
//Виводимо список кафедр до зміни таблиці
. . .
//Транзакція створення нової кафедри
Cathedra cth = new Cathedra();
cth.setName("NewCathedra");
cth.setChief("NewChief");
cth.setStaff(99);
```

```
em.getTransaction().begin();
  em.persist(cth);
em.getTransaction().commit();
//Виводимо список кафедр після зміни таблиці
. . .
```

У цьому прикладі ми не налаштовуємо id кафедри, тому що у базі визначено автоматичну генерацію первинних клюічів, а в моделі для поля id прописано анотацію `@GeneratedValue(strategy = GenerationType.IDENTITY)`

4.3.9.5 Вилучення запису із таблиці по id

Створімо новий клас TestDelById і у ньому реалізуймо вилучення запису про кафедру із таблиці Cathedra по заданому id.

Значення id вибирайте відповідно до даних отриманих в попередньому тесті.

```
//Створюємо EntityManager
//Виводимо список кафедр до зміни таблиці
   . .
//Транзакція видалення кафедри
em.getTransaction().begin();
  Cathedra st = em.find(Cathedra.class, 8);
  em.remove(st);
em.getTransaction().commit();
//Виводимо список кафедр після зміни таблиці
. . .
```

4.3.9.6 Вилучення запису із таблиці з використанням JPQL

Створімо новий клас TestDelByName і у ньому реалізуймо вилучення запису про кафедру з ім’ям NewCathedra із таблиці Cathedra.

```
//Створюємо EntityManager
//Виводимо список студентів до зміни таблиці
   . .
//Транзакція видалення нового студента
String sq = "DELETE FROM Cathedra cth WHERE cth.name = ?1";
Query qDel = em.createQuery(sq);
qDel.setParameter(1, "NewCathedra");
em.getTransaction().begin();
  qDel.executeUpdate();
em.getTransaction().commit();
//Виводимо список студентів після зміни таблиці
. . .
```

4.3.9.7 Використання запиту на SQL

Створімо клас TestSql, у якому реалізуємо SQL запит до таблиці GRADUATE.

Список, який ми отримаємов результаті запиту, буде містити масиви значень атрибутів об’єктів типа Graduate. Елементи масиву розташовуються у

такому порядку, як прописані у базі.

```
//Створюємо EntityManager
   . . .
//SQL query
String sql = "SELECT * FROM GRADUATE  WHERE GRADUATE.BALL >= 0";
Query qSql = em.createNativeQuery(sql);
em.getTransaction().begin();
List<Object[]> list = qSql.getResultList();
em.getTransaction().commit();
//Output list
for (Object[] arr : list) {
   for (int i = 0; i < arr.length; i++) {
      System.out.print(arr[i] + "\t");
   }
   System.out.println();
}
```

Як бачимо, використовувати SQL запити можна, але працювати з результатами запитів не дуже зручно.

4.3.10 **Створення таблиць БД на основі об'єктної моделі**

У проєкті, який було створено на попередньому етапі, ми працювали з об'єктною моделлю, яка була згенерована на основі існуючої бази даних, у складі якої було дві таблиці.

Тепер ми спробуємо створити ще одну таблицю, за допомогою засобів JPA.

4.3.10.1 Створення jpa-моделі для нової таблиці

Створення класу об'єктної моделі розглянемо на прикладі сутності Job, яка має атрибути id, firm, post, start, gaduate.

Створімо у пакеті model клас, використовуючи пункт меню New→JPA Entity.

У діалозі, що з'явився, визначимо назву класу Job і натиснемо кнопку «Next».

Далі скористуймося кнопкою «Add» діалогу і визначимо поля класу.

Для поля id визначимо тип int і позначимо його як ключове (треба поставити галочку у колонці Key).

Для полів firm та post визначимо тип java.lang.String.

Для поля start вибираємо тип int.

Для поля graduate тип Graduate

Після цього натискаємо кнопку «Finish».

На панелі JPA Structure має з'явитися перелік атрибутів сутності Student, а на панелі JPA Details маємо побачити діалог налаштувань сутності.

Якщо клацнути мишкою по атрибуту id, то на панелі JPA Details відкривається діалог додаткових налаштувань цього атрибуту.

Відкриємо налаштування Primary Key Generation і включимо відповідний

прапорець. Після цього відкриємо список Strategy і виберемо стратегію Identity.

Для поля graduate за допомогою контекстного меню атрибута Map As вибираємо анотацію Many to One, а на панелі JPA Details для TargetEntity вибираємо клас – model.Graduate (із довгого списку). Також виставляємо галочку Cascade/Remove.

Далі можна додати до класу Graduatet поле jobs типу java.util.List<Job> і додати анотацію One to Many. Також створити для нього геттер, сеттер та методи додавання і вилучення роботи.

Можна також включити деякі запити до сутностей. Для цього на панелі JPA Structure слід клацнути по назві сутності і у діалозі на панелі JPA Details вибрати функцію Queries. Після натискання кнопки «Add» з'являється діалог, у якому слід визначити назву запиту (наприклад, Job.findAll), його тип (NamedQuery) та ввести текст запиту(наприклад select x from Job x). Як результат, перед назвою класу має з'явитися відповідна анотація, наприклад, така:

```
@NamedQuery(name = "Job.findAll",
                                query = "select x from Job x")
```

4.3.10.2 Генерація таблиць БД на основі моделі

Для створення таблиць бази даних на основі об'єктної моделі можна скористатися функцією контекстного меню проєкту JPA_Tools->Generate Tables from Entity, рисунок 4.15.

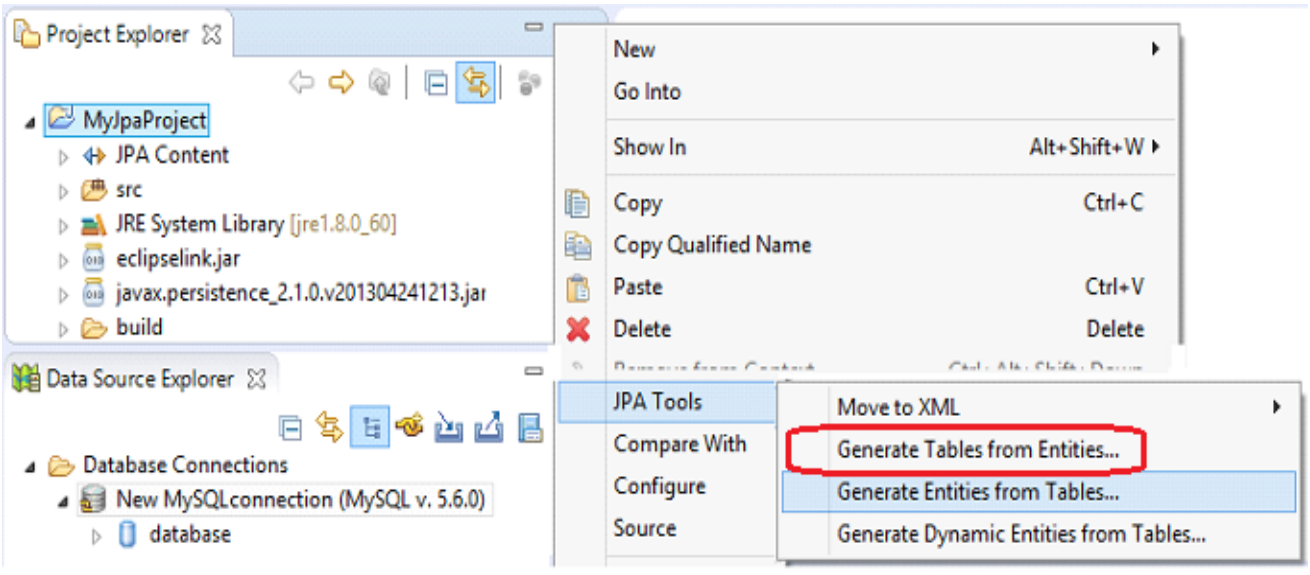

Рисунок 4.15 – Вибір функції генерації таблиць бази даних

Результатом буде поява попередження, що всі таблиці будуть створені заново.

Після створення таблиць інформацію про шлях до бази даних в xml файлі доцільно представити у вигляді тільки назви бази. У цьому випадку наш проєкт буде шукати базу там де він працює, що спрощує перенесення проєкту і бази в інші місця.

Рядок з цією інформацією в xml файлі має виглядати приблизно так:

```
<property name="javax.persistence.jdbc.url"
          value="jdbc:derby:dbByvoino;create=true"/>
```

4.4 Завдання для самостійної роботи

В якості завдання для самостійної роботи слід створити візуальний застосунок для роботи із своєю базою даних подібний до того, що був створений на другій лабораторній роботі, але для роботи з базою даних слід використовувати JPA.

4.5 Рекомендації до виконання завдання

4.5.1 Створення проєкта

За основу беремо проєкт, який було створено під час виконання лабораторної роботи.

Як результат маємо необхідні підключення у buld path і з'єднання з базою даних.

Маємо також пакет model з класами сутностей.

Класи візуальної частини візьмемо з лабораторної роботи 2. До проєкту можна скопіювати весь пакет view.

4.5.2 Виправлення помилок у пакеті view

Після копіювання пакету view у деяких його класах з'являються помилки. Це пов'язано з тим, що ми ще не реалізували класи пакетів controller та view. Ці класи з тими ж назвами нам доведеься створити, але вміст цих класів буде інший, бо доступ до бази треба реалізувати через JPA.

Можна позбутись повідомлень про помилки генеруючи пакети, класи та методи шляхом автоматичного виправлення помилок. Починати краще з виправлення помилок імпорту.

Але цього можна і не робити, а просто створити пакети та реалізувати потрібні класи.

4.5.3 Реалізація контролера

Створімо пакет controller і в ньому клас Controler.

```
public class Controller {

   private static EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("JpaByvoino");

   public static boolean tableExist(String tableName) {
      // If table exist, it is possible to count number of records.
      // If table does not exist, exception occurs.
      String testQuery = String.format(
           "select count(*) from %s ", tableName);
```

```
        EntityManager em = emf.createEntityManager();
        Query q = em.createNativeQuery(testQuery);
        try {
            q.getSingleResult();
            return true;
        } catch (Exception e) {
            return false;
        }finally {
            em.close();
        }
    }

    public static List<Map<String,Object>> executeQuery(String jpql) {
        //Output for debuging
        System.out.println(jpql);
        EntityManager em = emf.createEntityManager();
        Query query = em.createQuery(jpql);
        query.setHint(QueryHints.RESULT_TYPE, ResultType.Map);
        List<Map<String,Object>> list = query.getResultList();
        em.close();
        return list;
    }

    public static void createTable(String queryClass) {
        //This method was not realize, because tables already exist
    }

    public static void add(String tableName,Map<String,Object> map) {
        Object obj = null;
        try {
            String queryClass  = "query.Query" + tableName;
            Class<?> clz = Class.forName(queryClass  );
            Method mth = clz.getDeclaredMethod("createObject",Map.class);
            obj = mth.invoke(null, map);
        } catch (Exception e1) {
            e1.printStackTrace();
            JOptionPane.showMessageDialog(null,"Object create problem");
            return;
        }
        EntityManager em = emf.createEntityManager();
        System.out.println(obj);
        try {
            em.getTransaction().begin();
            em.persist(obj);
            em.getTransaction().commit();
        } catch (Exception e) {
            em.getTransaction().rollback();
            e.printStackTrace();
        } finally {
            em.close();
        }
```

```
}

public static void edit(String tableName,Map<String,Object> map) {
   EntityManager em = emf.createEntityManager();
   Object obj = null;
   try {
      Class<?> clzTable = Class.forName("model." + tableName);
      obj = em.find(clzTable, map.get("id"));
      Class<?> clzQuery = Class.forName("query.Query" + tableName);
      Method mtd = clzQuery.getMethod(
                              "editObject",Object.class, Map.class);
      obj = mtd.invoke(null, obj, map);
   } catch (Exception e1) {
      e1.printStackTrace();
      return;
   }
   if(obj != null)
      try {
         em.getTransaction().begin();
         em.merge(obj);
         em.getTransaction().commit();
      } catch (Exception e) {
         em.getTransaction().rollback();
         e.printStackTrace();
      } finally {
         em.close();
      }
}

public static void delete(
                String tableName, Map<String, Object> map) {

   int id = (int)map.get("id");
   Class<?> clazz = null;
   try {
      clazz = Class.forName("model." + tableName);
   } catch (Exception e1) {
      e1.printStackTrace();
      return;
   }
   if(clazz != null) {
      EntityManager em = emf.createEntityManager();
      try {
         em.getTransaction().begin();
         Object delObj = em.find(clazz, id);
         em.remove(delObj);
         em.getTransaction().commit();
      } catch (Exception e) {
         e.printStackTrace();
         em.getTransaction().rollback();
      } finally {
```

```
            em.close();
        }
    }
}

public static Object getObjectById(String tableName, int id) {
    EntityManager em = emf.createEntityManager();
    Object obj = null;
    try {
        Class<?> clz = Class.forName("model." + tableName);
        obj = em.find(clz, id);
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
    return obj;
}

public static int executeUpdate(String query) {
    EntityManager em = emf.createEntityManager();
    Query q = em.createQuery(query);
    try {
        em.getTransaction().begin();
        int n = q.executeUpdate();
        em.getTransaction().commit();
        return n;
    } catch (Exception e) {
        e.printStackTrace();
        em.getTransaction().rollback();
    } finally {
        em.close();
    }
    return 0;
}
}
```

4.5.4 **Класи пакету Query**

Так само як і в другій роботі нам потрібно створити класи для таблиць і клас Query для довільних запитів.

Для прикладу наведемо тут клас QueryGraduate для другої таблиці.

```
public class QueryGraduate  {

//Запити на створення таблиці БД
public static String queryCreate() {
    //This method was not realize, because tables already exist
    return null;
}

//Запит на отримання вмісту таблиці БД
public static String queryGetAll() {
```

```
        return  "select g.id, g.fio, g.finish, g.ball,"
              + " g.cathedra.name AS cathedra,"
              + " g.cathedra.id AS idcathedra"
              + " FROM model.Graduate g "
              + " ORDER BY cathedra, g.fio";
    }

    //Запит на отримання переліку випускників для заданої кафедри
    public static String queryGetForCathedra(int idCathedra) {
        return String.format("select grd.id, grd.fio, "
              + "grd.finish, grd.ball from model.Graduate grd "
              + "WHERE grd.cathedra.id = %d"
              + " ORDER BY grd.fio", idCathedra);
    }

    //Запит на створення нового випускника
    public static Graduate createObject(Map<String,Object>map) {
        Graduate grd = new Graduate();
        grd.setFio((String) map.get("fio"));
        grd.setFinish((int) map.get("finish"));
        grd.setBall((int) map.get("ball"));
        int idCathedra = (int) map.get("idcathedra");
        Cathedra cathedra = (Cathedra) Controller.
              getObjectById("Cathedra",idCathedra);
        grd.setCathedra(cathedra);
        return grd;
    }

    //Запит на редагування інформації про випускника
    public static Object editObject(Object obj,
          Map<String, Object> map) {
        Graduate grd = (Graduate)obj;
        grd.setFio((String) map.get("fio"));
        grd.setFinish((int) map.get("finish"));
        grd.setBall((int) map.get("ball"));
        int idCathedra = (int) map.get("idcathedra");
        Cathedra cathedra = (Cathedra) Controller.
              getObjectById("Cathedra",idCathedra);
        grd.setCathedra(cathedra);
        return grd;
    }
}
```

4.5.5 **Створення класу DlgJob**

Цей діалог для введення даних та редагування даних таблиці Job можна створити за зразком діалогу DlgGraduate.

4.6 Вимоги до звіту

– Назва роботи.
– Мета роботи.
– Тексти класів моделі.
– Текст файлу persistene.xml
– Тексти класів тестування запитів до бази та результати тестування їх у вигляді копій консолі.
– Тексти класів пакету query проєту для самостійної роботи.
– Тексти методів для реалізації довільних запитів до своєї бази та результати тестування цих запитів.
– Висновки.

4.7 Контрольні питання

1. Що таке об'єктно-реляційне відображення.
2. Причини та переваги використання ORM.
3. Назвіть основні компоненти архітектури JPA і для чого вони.
4. Розкажіть послідовність взаємодії компонентів JPA.
5. Для чого потрібен файл persistence.xml.
6. Назвіть обмеження, що накладаються на збережені об'єкти.
7. Як вказати первинний ключ і його автогенерація за допомогою анотацій.
8. Які є способи каскадної обробки об'єктів.
9. Назвіть основні стани збережених об'єктів і як вони в них потрапляють.
10. Напишіть запит для отримання об'єктів.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. The Java EE 6 Tutorial. [Електронний ресурс]. – Режим доступу: http://docs.oracle.com/javaee/6/tutorial/doc/index.html

5 ЛАБОРАТОРНА РОБОТА № 5.
ЗНАЙОМСТВО З JAVA ТЕХНОЛОГІЯМИ SEVLET ТА JSP

Мета роботи:

– Ознайомитися з технологіями для реалізації інтерфейсу користувача.
– Створити програму для роботи з реляційною базою даних через WEB інтерфейс.

5.1 Короткі теоретичні відомості

5.1.1 **Технологія Servlet**

Сервлети - це особливим чином написані (згідно специфікації) Java-програми, що призначені для обробки інформації та формування веб-сторінок. Вони виконуються на сервері, а виклик їх здійснюється дистанційно через web-браузер за допомогою HTTP протоколу.

Докладну інформацію про сервлети можна знайти за адресою:

https://docs.oracle.com/javaee/5/tutorial/doc/bnafd.html

Програму, що призначена для збереження сервлетів, системної підтримки та забезпечення їх життєвого циклу, називають контейнером. Контейнер може працювати як повноцінний самостійний веб-сервер, або бути постачальником сторінок для іншого веб-сервера. В лабораторній роботі ми будемо використовувати в якості контейнера Appache Tomcat.

Сервлети повинні реалізувати інтерфейс Servlet. Цей інтерфейс потребує реалізації 5 методів:

– void init(ServletConfig config) - метод автоматично викликається один раз при виконанні сервлета для його ініціалізації. Параметр ServletConfig надається контейнером сервлетів, який виконує сервлет;
– ServletConfig getServletConfig() - метод повертає посилання на об'єкт, який реалізує інтерфейс ServletConfig. Цей об’єкт надає доступ до інформації про конфігурацію сервлету;
– String getServletInfo() - метод повертає рядок з інформацією про сервлет;
– void service(ServletRequest request, ServlerResponse response) - контейнер сервлета викликає цей метод для відповіді на клієнтський запит до сервлету;
– void destroy() - метод викликається після завершення виконання сервлета контейнером сервлетів. Даний метод слід застосовувати для звільнення ресурсів, використовуваних сервлетом, такі як відкриті файли або відкриті з'єднання з БД.

Але сервлети, що працюють з клієнтом через Web, зазвичай реалізують інтерфейс Servlet не напряму, а через клас HttpServlet. У цьому класі метод service перевизначається, щоб мати можливість розрізнити стандартні методи запитів, одержувані від Web-браузера клієнта. Типи запитів HTTP є get, post,

delete, options, put, trace. Найбільш поширені - get і post. У класі HttpServlet визначені методи doGet і doPost для обробки реакції на ці запити.

Про різницю між doGet та doPost можна почитати в інтернеті. За замовчуванням, Eclipse створює шаблон сервлету, у якому метод doPost викликає метод doGet.

Створюючи шаблон сервлету, Eclipse додає до заголовку класу анотацію @WebServlet, яка визначає ім'я сервлету, що буде використовуватися для звертання до сервлету з інших компонентів проєкту, наприклад, із html сторінки. За замовчування ім'я сервлету співпадає з ім'ям класу, без розширення. Наприклад, @WebServlet("/MyServlet").

У сервлеті можна сформувати HTML-код веб-сторінки за допомогою об'єкта типу PrintWriter, який можна отримати через параметр response методу doGet. Приклад реалізації сервлету можна подивитися у пункті 5.2.6

Але такий спосіб не дуже зручний, тому формування html тексту у сервлетах зазвичай реалізується інакше.

5.1.2 **Технологія JSP**

Технологія JSP-сторінок (Java Server Pages - JSP) дозволяє без зусиль створювати web-сторінки, у яких є як статична, так і динамічна компоненти.

Докладну інформацію про JSP сторінки можна знайти за адресою:

https://docs.oracle.com/javaee/5/tutorial/doc/bnagy.html

JSP-сторінка обслуговує запити відповідно до технологіїї Java Servlet. Коли запит вперше відображається на JSP-сторінку, то вона автоматично перетворюється у клас сервлету, який компілюється і зберігається. При повторних зверненнях до цієї сторінки використовується вже готовий сервлет. У разі оновлення сторінки, сервлет перекомпілюється.

Сторінка JSP являє собою текстовий документ, який містить два типи тексту: статичні дані та елементи JSP, які ще називають елементами сценарію.

Статичні дані можуть використовувати різні текстові формати, наприклад html.

Елементи сценарію містять код Java, або іншої мови, і позначаються спеціальними тегами <% %>. Якщо замість мови java, що приймається за замовчуванням, потрібно використовувати іншу мову, слід визначити цю мову у директиві page на початку JSP- сторінки:

<% @ page language = "scripting language"%>

У директиві page можна також визначити перелік пакетів та класів, що мають бути імпортовані, наприклад:

<%@ page import="java.util.*, jtap.lab4.model.*" %>

5.1.2.1 Елементи сценарію

Оголошення

JSP-оголошення використовується для оголошення полів та методів на мові створення сценаріїв сторінки. Коли мовою створення сценаріїв є Java, змінні і методи в JSP-оголошеннях стають оголошеннями класу сервлета JSP-

сторінки. Зверніть увагу на знак оклику, як ознаку оголошення, у наведеному нижче прикладі:

```
<%!
  private String name;
  public toString(){
     return name;
  }
%>
```

Скріптлети

JSP-скріплет використовується для зберігання будь-якого фрагменту коду, що відповідає вимогам мови сценаріїв сторінки. Синтаксис для скріптлета виглядає наступним чином:

```
<%
 scripting language statements
%>
```

Змінні, що створені в скріптлеті, доступні з любого місця JSP-сторінки.

Вирази

JSP-вираз використовується для вставки результатів обчислень виразів у потік даних, що повертається клієнту. При цьому значення виразу конвертується у рядок. Коли мовою написання сценаріїв є Java, вираз перетворюється в оператор, що перетворює значення виразу в об'єкт String и вставляєте в неявний об'єкт out.

Синтаксис такого виразу виглядає наступним чином.

Символ = є частиною відкриваючого тегу.

```
<%=  scripting language expression %>
```

Зверніть увагу, що використання крапки з комою в JSP-виразах (саме у виразах, а не фрагментах коду) заборонено, навіть у тих випадках, коли у такого ж виразу в скріптлеті, вона присутня.

Приклад створення jsp сторінки наведено в пункті 5.2.7.

5.1.3 **Способи обміну інформацією між компонентами web-застосунку**

Зазвичай web-застосунок складається з декількох компонентів, що обмінюються інформацією між собою.

Якщо джерелом інформації є html сторінка, то інформація може бути передана через форму, а саме, через атрибути тегів, таких як input, select, button. Кожен параметр, що передається, повинен мати назву і значення. Приймати цю інформацію можуть сервлети або jsp-сторінки через параметр **request** методів doPost або doGet. Для цього використовується метод getParameter(String). До методу передається ім’я параметру, а метод повертає значення параметру у вигляді рядка символів.

Сервлети і jsp-сторінки можуть також обмінюватися між собою атрибутами також через параметр request. Для цього використовуються методи setAttribute(String, Object) та getAttribute(String). Атрибут – це об’єкт будь якого

класу. А для його ідентифікації використовується рядок символів, що є ім'ям атрибуту.

Якщо ж зі сторінкою пов'язано декілька запитів, то request вже не підходить для збереження спільних даних. У цьому випадку слід використовувати об'єкт класу HttpSession, що створюється, коли відкривається сторінка у браузері. У цей момент на сервері відбувається відкриття сесії. Термін «сесія» застосовується до взаємодії сервера і користувача від моменту перегляду першої сторінки на сервері до закриття браузера або ж до закінчення часу відведеного для сесії з користувачем від останнього його звернення до сервера. Сервер підтримує сесію за посередництвом кукі.

Доступ до об'єкту «сесія» на jsp-сторінках можливий через змінну session. У сервлетах цей об'єкт можна отримати через параметр request за допомогою методу getSession(). Для обміну даними з сесією використовуються методи setAttribute() та getAttribute(), що аналогічні методам параметру request.

Для прикладу розглянемо веб-застосунок, що проводить тестування методу random() класу Math. Цей застосунок обчислює середнє значення, дисперсію та середнє квадратичне відхилення для декількох вибірок заданого розміру. Результат роботи цього застосунку показано на рисунку 5.1.

Веб-сторінка застосунку складається з двох іфреймів.

Верхній іфрейм використовується для введення обсягу вибірки та кількості повторних тестів.

Нижній іфрейм використовується для відображення таблиці результатів, яка з'являється після натискання кнопки «Тестувати».

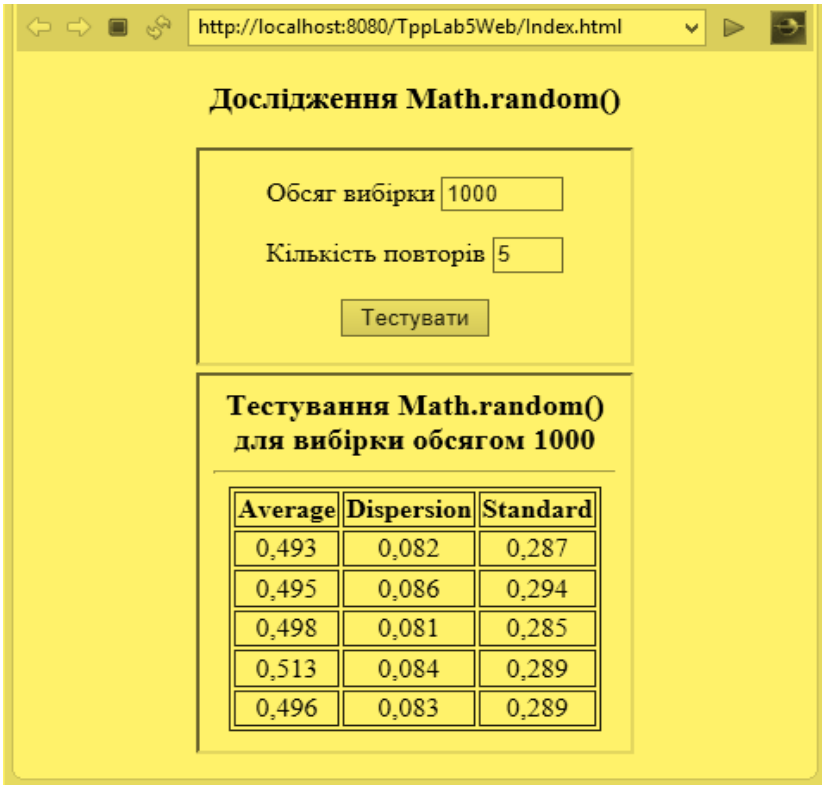

Рисунок 5.1 – Результат тестування веб-застосунку

До складу застосунку входить декілька компонентів. Ці компоненти та зв’язки між ними показано на рисунку 5.2. Потоки даних показано червоними (більш товстими) лініями.

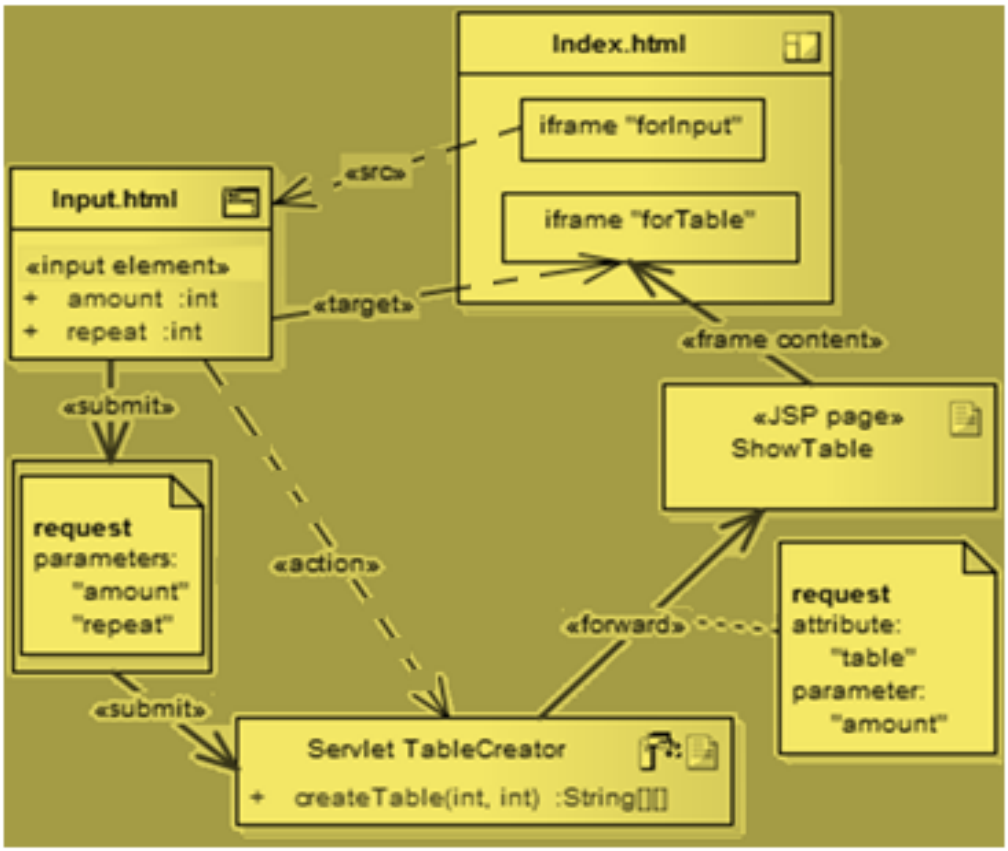

Рисунок 5.2 – Схема взаємодії компонентів веб-застосунку

Далі наводимо короткий опис цих компонентів та їх код.

5.1.3.1 Index.html

Це файл стартової сторінки, на якій розташовано два фрейми – forInput та forTable.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>MathRandomTest</title>
</head>
<body>
  <center>
    <h3>Дослідження Math.random()</h3>
    <iframe name="forInput" src="Input.html" width="250"
height="130">
    </iframe>
    <br>
    <iframe name="forTable" width="250" height="230"> </iframe>
  </center>
```

```
</body>
</html>
```

5.1.3.2 Input.html

Це файл сторінки для введення даних та запуску тестування. Ця сторінка одразу відображується у фреймі forInput і пов’язана з ним через атрибут src тегу input фрейму forInput. Атрибут action форми, яка визначена у цьому файлі, вказує на сервлет TableCreator. Метод doPost цього сервлету буде виконано після натискання на кнопку «Тестувати». Атрибут target форми визначає, що результат обробки даних форми буде надіслано у фрейм forTable.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title></title>
</head>
<body>
  <form action="TableCreator" target="forTable" method="POST">
    <center>
      <p>
        Обсяг вибірки <input type="text" name="amount" size="6"
value="1000">
      <p>
        Кількість повторів <input type="text" name="repeat"
size="1"
          value="5">
      <p>
        <input type="submit" value="Тестувати">
    </center>
  </form>
</body>
</html>
```

5.1.3.3 TableCreator

Сервлет, що забезпечує проведення потрібної кількості тестів, обчислення параметрів вибірок, та формування двовимірного масиву результатів тестування. Дані з форми сервлет отримує через параметри об’єкту request. А результат обробки у вигляді атрибуту table перенаправляє у jsp-сторінку ShowTable через об’єкт request.

```
@WebServlet("/TableCreator")
public class TableCreator extends HttpServlet {
  private static final long serialVersionUID = 1L;

  public TableCreator() {
    super();
  }
```

```
  protected void doPost(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException {
     doGet(request, response);
  }

  protected void doGet(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException {
     int amount = Integer.parseInt(request.getParameter("amount"));
     int rowCount =
Integer.parseInt(request.getParameter("repeat"));
     String[][] table = createTable(amount,rowCount);
     request.setAttribute("table", table);
     request.getRequestDispatcher("ShowTable.jsp").forward(request,
response);
  }

  private String[][] createTable(int amount, int rowCount) {
     String[][] table = new String[rowCount][3];
     DecimalFormat dcfr = new DecimalFormat("0.000");
     for (int r = 0; r < rowCount; r++) {
        Collection<Double> coll = new Vector();
        double sr = 0;
        for (int i = 0; i < amount; i++) {
           double x = Math.random();
           sr += x;
           coll.add(x);
        }
        sr /= amount;
        table[r][0] = dcfr.format(sr);
        double d = 0;
        for (double x : coll) {
           d += (x - sr) * (x - sr);
        }
        d /= coll.size();
        table[r][1] = dcfr.format(d);
        table[r][2] = dcfr.format(Math.sqrt(d));
     }
     return table;
  }

}
```

5.1.3.4 ShowTable.jsp

Це jsp-файл, що відображає результати дослідження у фреймі forTable. Необхідну інформацію отримує від об’єкта request через параметр amount та атрибут table.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
```

```
   pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title></title>
</head>
<body>
   <%
      String[][] table = (String[][]) request.getAttribute("table");
      String amount = request.getParameter("amount");
   %>
   <center>
      <font size="+1"> <strong>
           Тестування Math.random()<br>
           для вибірки обсягом <%=amount%>
      </strong>
      </font>
      <hr>
      <table border="1">
         <tr>
            <th>Average</th>
            <th>Dispersion</th>
            <th>Standard</th>
         </tr>
         <%
            for (String[] row : table) {
         %>
         <tr>
            <td align="center"><%=row[0]%></td>
            <td align="center"><%=row[1]%></td>
            <td align="center"><%=row[2]%></td>
         </tr>
         <%
            } // for
         %>
      </table>
   </center>
</body>
</html>
```

5.1.4 **Сервер Apache Tomcat**

Apache Tomcat – це контейнер сервлетів, розроблений Apache Software Foundation. Tomcat повністю написаний мовою програмування Java та реалізує специфікацію сервлетів і Java Server Pages від Sun Microsystems, що є стандартом для розробки веб-застосунків на Java.

Tomcat можна використовувати і як HTTP – сервер, що ми і будемо

робити. Але продуктивність його у цій ролі гірша, ніж у спеціалізованих веб-серверів.

Проєкти, що мають виконуватися під управлінням Tomcat розташовують у папці webapps.

Папки з проєктом має таку структуру.

Усі .html та .jsp файли знаходяться у кореневій папці проєкту і доступні клієнту.

Натоміть папка WEB.INF, що знаходиться у папці проєкту, закрита для клієнтів.

Java класи знаходяться в папці WEB.INF\classes, а папка WEB.INF\lib зберігає бібліотечні jar-файли проєкту.

Скомпільовані файли JSP сторінок знаходяться за межами папки проєкту і заховані в папці workТомкату. Наприклад, шлях до цих файлів проєкту WebProject, що виконується на локальному хості буде таким: work\Catalina\localhost\WebProject\ org\apache\jsp.

5.2 Порядок виконання роботи

Для виконання роботи потрібна версія Eclipse IDE for Java EE Developers. Вимоги до Eclipse цієї версії докладно розглядалися у минулій лабораторній роботі.

5.2.1 **Підготовка сервера Tomcat**

5.2.1.1 Завантаження та розготання сервера

Сервер Apache Tomcat можна отримати за адресою http://jakarta.apache.org/tomcat

Скачайте і розгорніть сервер у якійсь папці.

Далі доцільно перевірити працездатність сетвера. Для цього слід активізувати сервер скориставшись командним файлом bin/startup.bat з папки, де розгорнуто сервер.

Запуск краще робити через панель cmd, щоб бачити системні повідомлення.

Можливо доведеться створити змінні середовища JRE-HOME та CATALINA_HOME через виклики Комп'ютер→ Властивості системи→ Додаткові параметри→ Змінні середовища→ шлях до папки jre або apache-tomcat.

Майте на увазі, що після налаштувань змінних оточення, cmd требе перезапустити.

Можливо також, що доведеться поміняти порт 8080, а можливо і інші. Це можна зробити відкривши файл config/server.xml.

Відкривати xml файли краще черезEclipse (File→Open) і переглядати в режимі дизайну. Наприклад, шлях до порта 8080 такий: Server→ Service→ Connector.

Після перевірки працездатності сервера його слід виключити за допомогою командного файлу bin/shutdown.bat/

5.2.1.2 Підключення Tomcat до Eclipse

Після того, як Tomcat розгорнуто треба передати інформацію про нього до Eclipse. Для цього слід використати такий ланцюжок викликів меню:

File → New →[Other]→ Server → Server→Next

Як результат отримуємо діалог, рисунок 5.3.

У діалозі вибираємо, наприклад, сервер Apache → Tomcat v 8.0, саме той що було розгорнуто. Інші параметри можна залишити без змін.

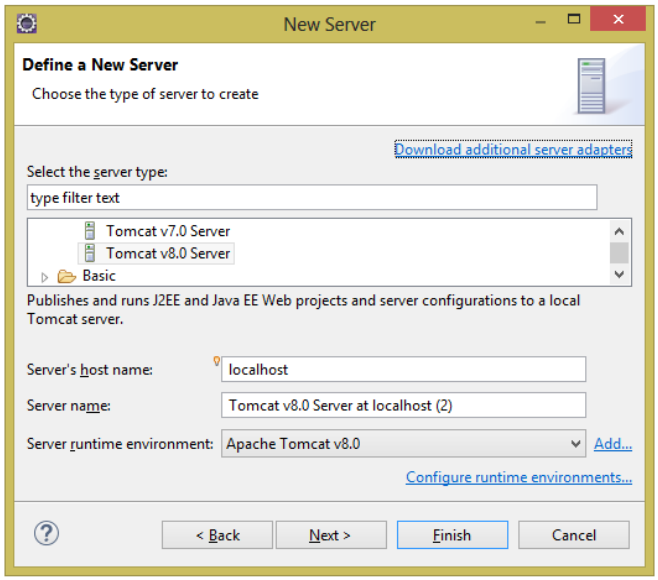

Рисунок 5.3 – Діалог вибору сервера

Далі натискаємо Next. З'являється діалог, рисунок 5.4, у якому слід вказати шлях до папки, де розгорнуто сервер: Tomcat installation directoty→Browse.

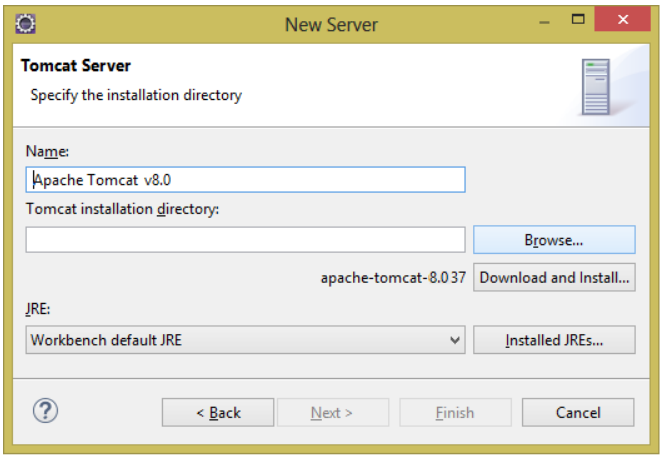

Рисунок 5.4 – Діалог налаштування шляху до сервера

У наступному вікні пропонується вибрати проєкти, які будуть запускатися на цьому сервері. Просто натискаємо Finish.

Після підключення сервера Eclipse створює каталог Servers, в якому зберігає файли налаштувань Tomcat. Якщо потрібно буде змінити налаштування сервера (наприклад, поміняти порт або визначити пул підключень до БД), то це можна зробити у файлах налаштувань.

Окрім того сервер має з'явився на вкладці Servers внизу. Якщо такої вкладки немає, то слід перевірити, чи створено перспективу (Java EE) або можна включите view вручну: Window → Show View → Servers, якщо немає в списку запропонованих пошукати в пункті Others.

Подвійний клік по рядку сервера на цій вкладці відкриває його налаштування. Найважливіші, це Timeouts - час очікування старту сервера. Пишуть, що дане значення краще збільшити, щоб не було проблем з налагодженням старту програми.

На рисунку 5.5 показано вигляд робочого вікна Eclipse з розкритою папкою Servers та діалогом налаштувань сервера.

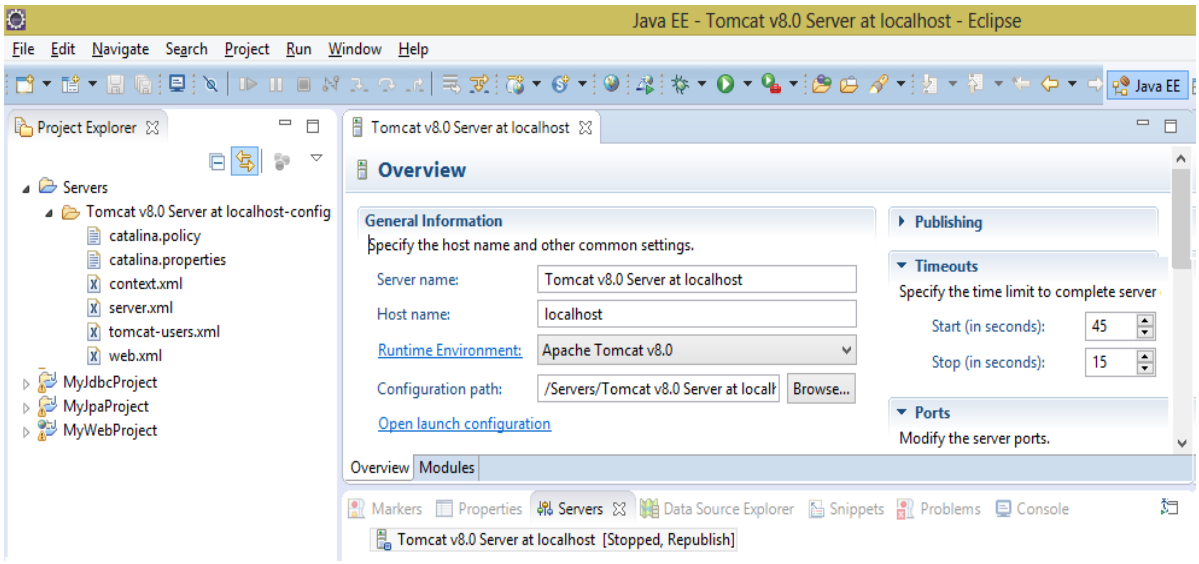

Рисунок 5.5 – Діалог налаштування сервера

5.2.2 **Створення проєкту**

Після того як Tomcat підключено до Eclipse можна перейти до створення WEB проєкту. Для цього слід використати такий ланцюжок викликів меню:

File → New → [Other]→Web→Dynamic Web Project→Next

Як результат отримуємо діалог, рисунок 5.6.

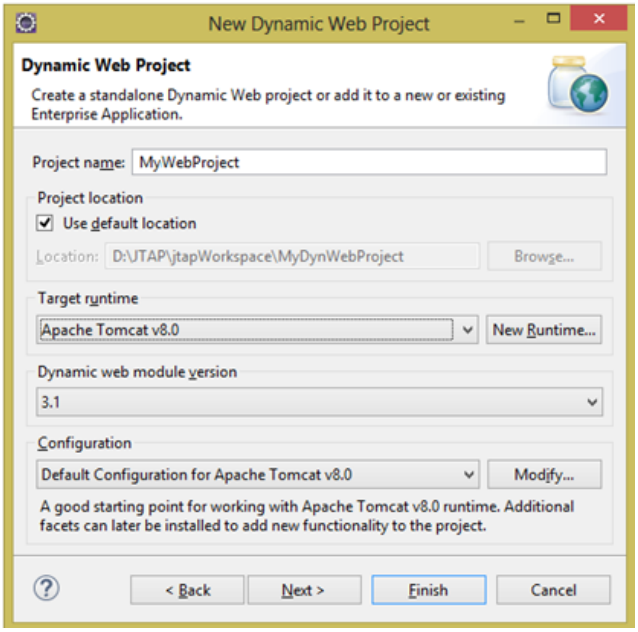

Рисунок 5.6 – Діалог створення WEB проєкту

У діалозі вказуємо ім'я проєкту. В якості Target runtime вбираємо створений на попередньому етапі сервер. У цьому випадку проєкт буде мати посилання на бібліотеку jar-файлів Tomcat.

Наступне вікно – це стандартні налаштування Java-додатку.

Далі з'являється діалог Web Module, рисунок 5.7. Поля діалогу мають таке призначення:

– Context root - контекст, за яким буде доступним додаток в браузері. Повний шлях буде http://localhost:8080/MyWeb, де MyWeb - значення поля;

– Content directory - коренева папка для war-архіву. У ній будуть створені WEB-INF , META-INF. Файли прєкту з неї будуть доступні клієнтам;

– Generate web.xml – вказівка, чи створювати файл web.xml. Починаючи зі специфікації Servlets 3.0 web-додатки можуть обходиться без цього файлу. Але спеціалісти рекомендують створювати цей файл завжди. Тож ставимо галочку.

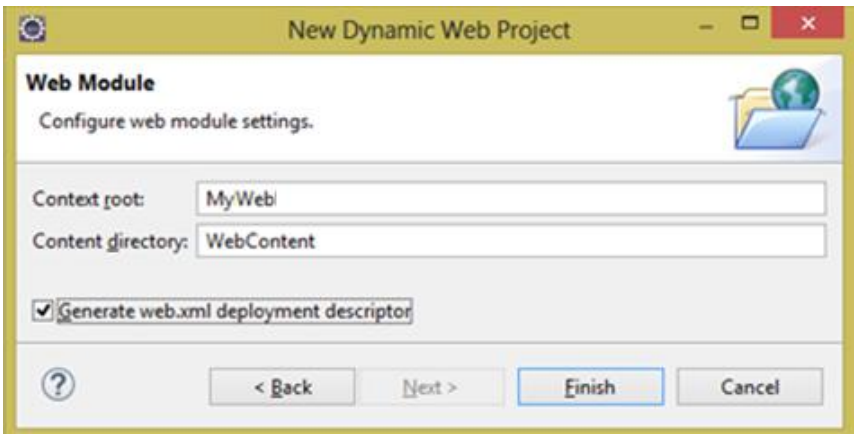

Рисунок 5.7 – Діалог створення WEB проєкту

До проекту треба також додати драйвер derby.jar, який потрібно

скопіювати до розділу Web Content у папку WEB_INF.lib

Після створення проєкту слід додати його до сервера. Для цього у контекстному меню сервера на вкладці Servers (у нижній частині робочої області) слід вибрати функцію «Add and Remove...» і скористатися діалогом, рисунок 5.8.

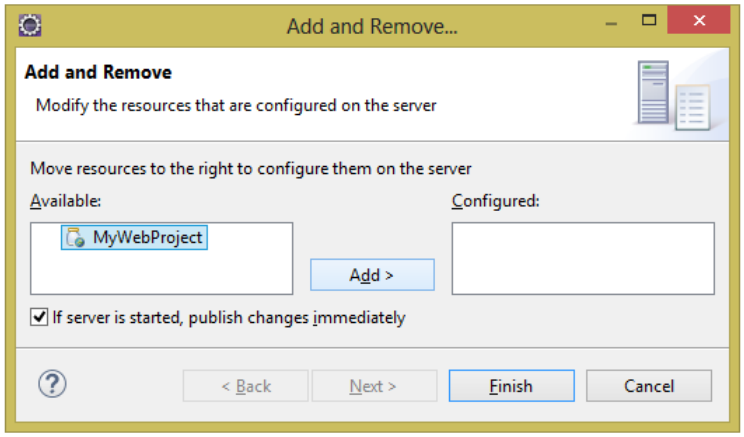

Рисунок 5.8 – Діалог передачі WEB проєкта серверу

Той самий результат можна отримати запустивши проєкт на виконання функцією контекстного меню проєкту Run as→Run on Server. Але у цьому випадку ми ще отримаємо і повідомлення про помилку, бо проєкт пустий.

За замовчуванням, результати виконання проєкту з'являються у вбудованому браузері. Якщо є бажання, щоб воно запускалося в іншому браузері, необхідно налаштувати Eclipse: Window → Preferences → General → Web Browser і визначити браузер, рисунок 5.9.

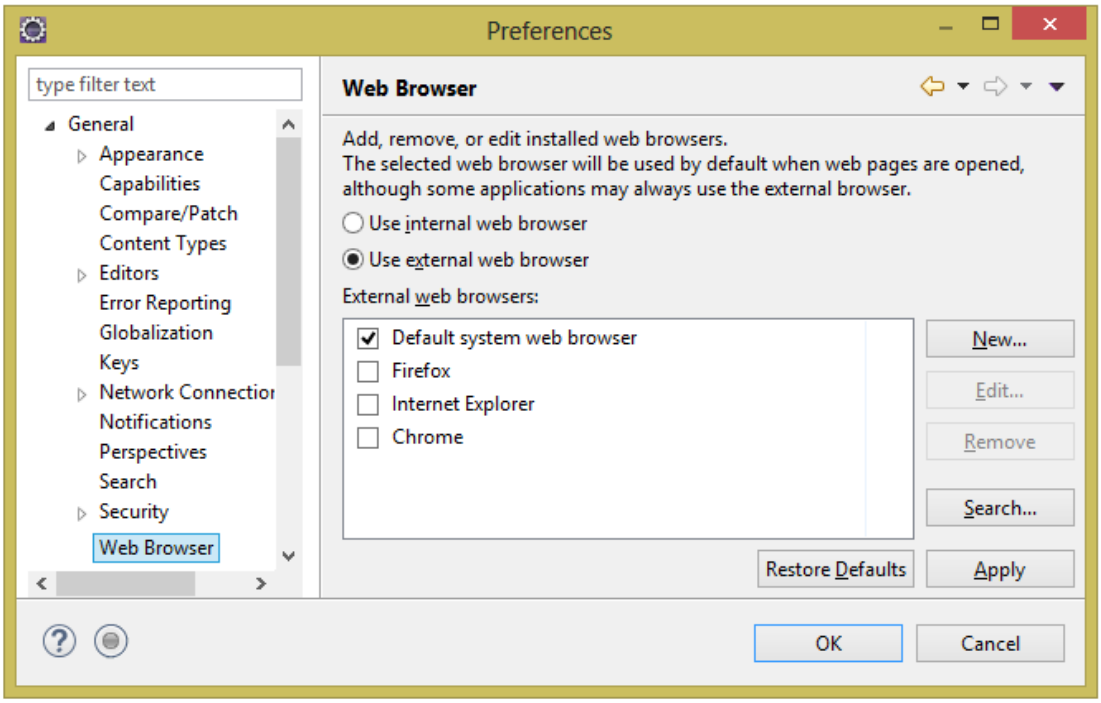

Рисунок 5.9 – Діалог вибору браузера

5.2.3 **Призначення проєкту**

В межах проєкту ми маємо створит веб сторінку через яку користувач зможе надсилати SQL заптити до бази даних і отримувати результати у вигляді таблички. Для реалізації завдання ми розробимо три файли: html, сервлет та jsp.

Виходячи з того, що web не є предметом нашого курсу, для реалізації web будемо використовувати найпростіші рішення.

5.2.4 **Підготовка бази даних і контролера**

Базу даних і контролер візьмемо з другої лабораторної роботи.

Скопіюємо базу даних до папки, шлях до якої не буде дуже довгим, тому що його доведеться вводити на сторінці.

Папку controller скопіюємо в папку src розділу Java Resources ghпроєкту.

Крім того, в папці WebContent проєкту створіть теку resource і туди завантажте файл з фото.

5.2.5 **Створення HTML сторінки**

Створімо стартову сторінку нашого проєкту, яка має виглядати приблизно так,як показано на рисуноку 5.10.

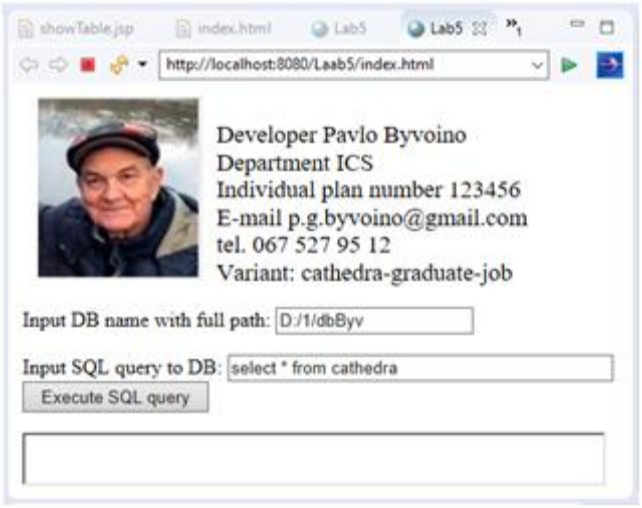

Рисунок 5.10 – Стартова сторінка проєкту

Зрозуміло, що на сторінках, які створюють студенти має бути інформація про студента і відповідати вибраному варіанту.

5.2.5.1 Рекомендації по створенню сторінки

У контекстному меню папки Web Content проєкту виберіть функцію New→ Html File, та введіть назву файлу index. У наступному вікні виберіть відповідний шаблон сторінки, наприклад, New HTML File (5) і натисніть Finish.

В результаті отримаємо шаблон файлу.

Далі доповнимо шаблон текстом відповідно з рисунком 5.11.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Lab5</title>
</head>
<body>
<img src="resource/photo.jpg" align="left" hspace=10 img>
<big><br>
  Developer Pavlo Byvoino <br>
  Department ICS<br>
  Individual plan number 123456<br>
  E-mail p.g.byvoino@gmail.com<br>
  tel. 067 527 95 12<br>
  Variant: cathedra-graduate-job<br>
</big>
<form action="TestServlet" target="forTable" method="POST">
  <p>Input DB name with full path:
  <input type="text" name="path" value="D:/1/dbByv"> </p>
  <p>Input SQL query to DB:
  <input type="text" name="query" size ="45"
  value = "select * from cathedra"><br>
  <input type="submit" value=" Execute SQL query "> </p>
</form>
<iframe name="forTable" width="450" height="100"> </iframe>
</body>
</html>
```

Рисунок 5.11 – Текст файлу HTML для стартової сторінки

Після підготовки тексту можна запустити файл на виконання функцією контекстного меню проєкту Run as→Run on Server.

5.2.6 **Створення сервлета**

Перше завдаання сервлета - налаштувати з’єднання з базою даних використовуючи параметр path.

Друге завдання сервлета – передати контролеру запит, що визначений параметром query, і отримати результат у вигляді списку карт.

Третє завдання – передати отримані результати на web сторінку.

Перші два завдання реалізуються так само як і у другій лабораторній роботі звичайними засобами Java.

Для реалізації третього завдання будемо використовувати об’єкт класу PrintWriter, який надає об'єкт response.

Створімо у папці src розділу Java Resources пакет servlet, де буде зберігатися наш сервлет.

Для створення сервлета в контекстному меню проєкту виберіть функцію New → [Other]→Web→Servlet. Результатом буде поява діалогу, рисунок 5.12.

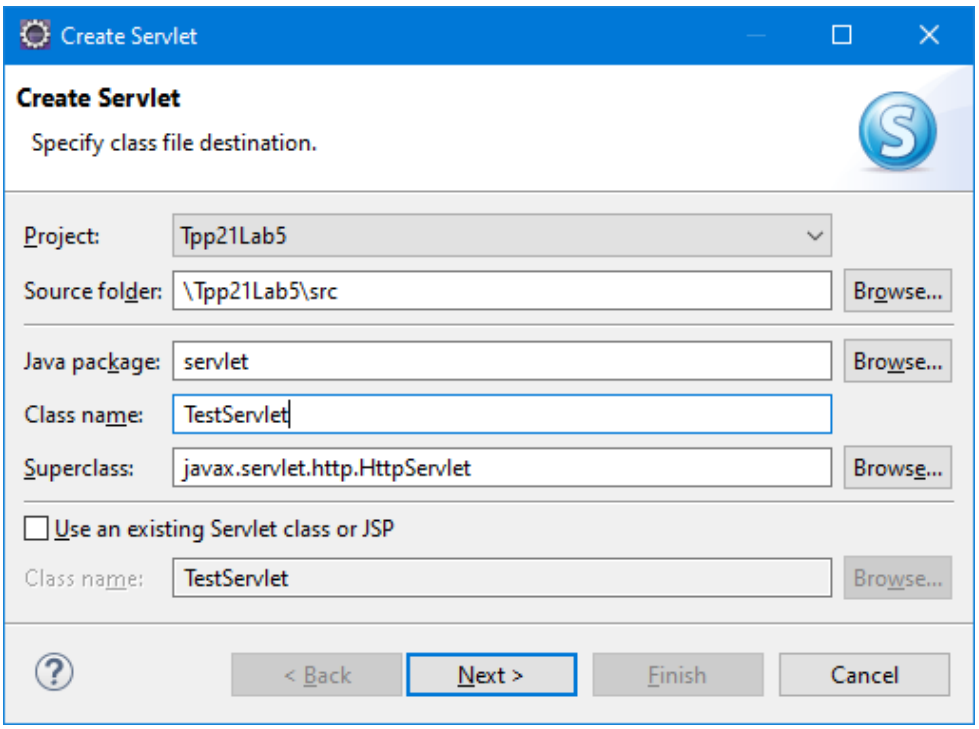

Рисунок 5.12 – Діалог створення сервлету

У діалозі визначаємо назви пакету та назву класу TestSevlet і натискаємо Finish. В пакеті має з'явитися клас сервлету.

У класі визначено пустий конструктор і методи doGet() та doPost() з параметрами request та responce, причому метод doPost() просто викликає метод doGet().

Таким чином нам достатньо довизначити метод doGet(), що спрацює після написання кнопки «Execute SQL query».

На рисунку 5.13 наведено код методу doGet() без заголовку.

```
String path = request.getParameter("path");
DbConnector.setPathToDb(path);

String query = request.getParameter("query");
List<Map<String, Object>>mapList = Controller.executeQuery(query);

// Data output
PrintWriter writer = response.getWriter();
for (Map<String, Object> map : mapList)
  writer.append(map.toString() + "<br>");
```

Рисунок 5.13 – Код методу doGey сервлету

Додайте цей код до класу сервлету і запустіть на виконання index.html. Після введення даних і натискання кнопки маємо отримати рядки із стандартною роздруковкою карт (ключ = значення).

Недолік представлення результатів полягає в тім, що вони не представлені

табличкою, як це планувалося.

Звичайно, за допомогою PrintWriter можна було б сформувати html код для формування таблички і у сервлеті, але ми це зробимо за допомогою jsp файлу.

Для того, щоб сервлет викликав jsp сторінку, змінимо частину коду сервлета, що йде за коментарем Output result і відповідає за виведення результатів.

Заменіть цю частину коду сервлета на такий, що наведено на рисунку 5.14:

```
request.setAttribute("mapList", mapList);
// goto showTable
request.getRequestDispatcher("showTable.jsp").
        forward(request, response);
```

Рисунок 5.13 – Код звернення до jsp сторінки

Тут створюється атрибут mapList для об’єкта request, значенням якого є список з картами.

Другий рядок активізує jsp сторінку ShowTable і передає їй об’єкти request та response.

5.2.7 **Створення JSP сторінки**

Jsp сторінка має створити табличку використовуючи атрибут mapList, що містить список з картами.

Код тегу body такої сторінки наведено на рисунку5.14.

```
<%
   List<Map<String, Object>> mapList =
      (List<Map<String, Object>>)request.getAttribute("mapList");
   Set<String> tableHeaderSet = mapList.get(0).keySet();
%>
<table border="1">
      <!-- Create table header -->
      <tr>
      <% for (String str : tableHeaderSet){ %>
           <th width="100"><%= str %></th>
      <% } %>
      </tr>
      <% for(Map<String, Object> map : mapList){ %>
      <!-- Output rows of table -->
        <tr>
        <%for (Object obj : map.values()) {
          String str = "null";
          if (obj != null)
          str = obj.toString();
        %>
          <td width="100" align="center"><%=str%></td>
        <%} %>
        </tr>
      <%} %>
</table>
```

Рисунок 5.14 – Фрагмент коду jsp сторінки для виведення таблиці

Перші рядки сторінки містять java код, де із списку беремо першу карту, а з неї отримуємо колекцію ключів, які будуть заголовками колонок таблиці.

Далі, в межах тегу <tr> у циклі for формуємо заголовок таблиці.

Після формування заголовку, в подвійному циклі for послідовно виводимо рядки таблиці, використовуючи колекцію значень карти.

Якщо якесь значення не визначено (null), виводимо тектом null.

Щоб створити таку стоінку в контекстному меню папки Web Content проєкту виберіть функцію New→JSP File і назвіть новий файл showTable.jsp.

У наступному вікні виберіть шаблон сторінки, наприклад, New JSP File (html) і натисніть Finish.

В результаті отримаємо шаблон JSP файлу. Якщо у шаблоні використовується таблиця кодування ISO-8859-1, треба змінити на UTF-8.

Далі, між тегами body сформуймо Java та html тексти таким чином, щоб отримати результати у вигляді таблиці, рисунок 5.14.

Коли будете вводити назви класів List, Map, Set, то використовуйте автодоповнення. В цьому випадку оголошення імпрту створяться автоматично.

Після створення сторінки можна знов запустити проект і отримати результат, який слід зафіксувати у звіті.

5.2.8 **Завантаження проєкту на сервер**

Щоб проєкт можна було запускати без використання Eclipse, слід створити war-архів цього проєкту і додати цей архів у папку webapps серверу Apache

Tomcat.

Для цього слід скористатися послідовністю виклику функцій контекстного меню проєкту Export->WAR file.

Далі, у діалозі рисунок 5.15, через кнопку Browse знайти папку webapps та завантажити туди war-файл.

Далі слід активізувати сервер. Для цього використовується bat-файл startup.bat з папки bin, де рогорнуто сервер. Запуск краще робити від імені адміністратора. Щоб бачити системні повідомлення, можна запускати сервер через панель cmd. Можливо доведеться створити змінну середовища JRE-HOME через виклики Комп'ютер->Властивості системи->Додаткові параметри->Змінні середовища>шлях до jre бібліотеки.

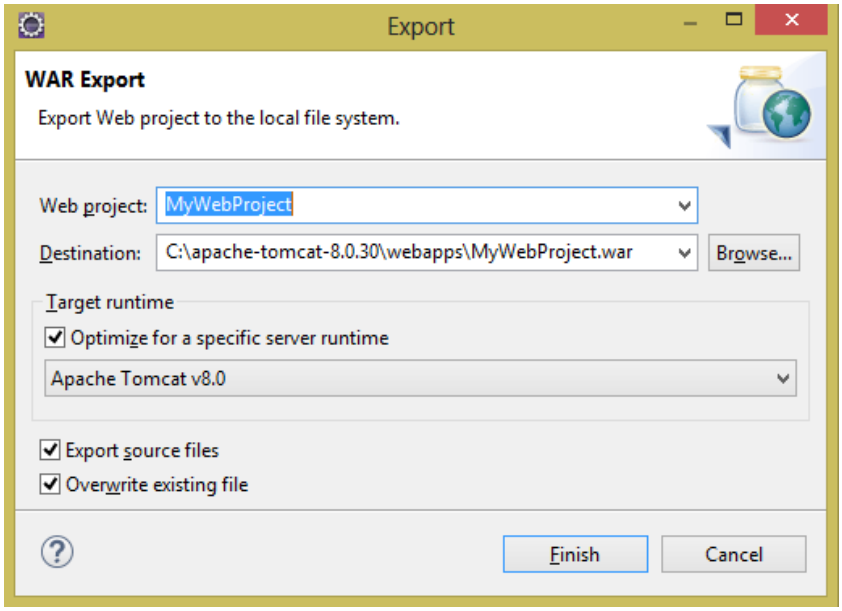

Рисунок 5.15 – Діалог завантаження war-файлу

Після того, як сервер стартує, можна запускати застосунок, набравши у браузері адресу http://localhost:8080/MyWebProject

5.3 Завдання для самостійної роботи

У межах самостійної роботи слід створити web-додаток для тестування Math.random(), що розглядався у теоретичних відомостях, рисунок 5.1.

Для тестування додатку використовувати такі параметри:

– n = 5 + остання цифра номера заліковки;
– обсяг вибірки = n*1000;
– кількість повторів = n.

5.4 Вимоги до звіту

– Назва роботи.
– Мета роботи.

– Тексти html файлу, сервлету і jsp сторінки, створених під час виконання лабораторної роботи.

– Результати тестових запусків для різних запитів (мінімум 2).

– Зовнішній вигляд web-сторінки проєкту, створеної у межах самостійної роботи, з результатами обчислень.

– Схема взаємодії компонентів web-інтерфейсу.

– Тексти сервлетів, jsp та html файлів.

– Результати тестування застосування у вигляді копій браузера.

– Висновки.

5.5 Контрольні питання

1. Що таке сервлет. Створити простий сервлет і активізувати його.
2. Що таке jsp-сторінка. Створити просту сторінку та активізувати її.
3. Елементи сценаріїв jsp-сторінок.
4. Способи обміну між компонентами web-інтерфейсу.
5. Як розгорнути web-застосунок.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Servlets. [Електронний ресурс]. – Режим доступу: https://www.wideskills.com/servlets/servlets-table-contents
2. Apache Tomcat. [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/Apache_Tomcat

6 ЛАБОРАТОРНА РОБОТА № 6.
ВЕБСЕРВІСИ

Мета роботи:

– Ознайомитися з технологіями для реалізації веб-сервісів.
– Створити програму для роботи з веб-сервісом.

6.1 Короткі теоретичні відомості

Веб-сервіс – це програмний ресурс, що знаходиться у мережі і призначений для обміну інформацією з іншими програмами. Цей термін в основному стосується серверів та клієнтів, що взаємодіють за допомогою певних інтернет-протоколів. Кажуть, що веб-сервіси – це реалізація інтернет-технологій для програм.

Веб-сервіс ідентифікується уніфікованим ідентифікатором ресурсів (URI), та переліком публічних інтерфейсів. Інформація, якою обмінюються між собою веб-сервіс та програмний додаток, зазвичай представляється у вигляді XML або JSON документів.

В межах Java-технологій, вебсервіс – це Java-програма, що зберігається на веб-сервері і за запитом надає якусь інформацію у вигляді xml або json документу.

Існують різні сервіси, які надають якісне рішення таких задач як надання інформації про погоду, ведення календаря, відправлення повідомлень, пошук, переклад і т. п.

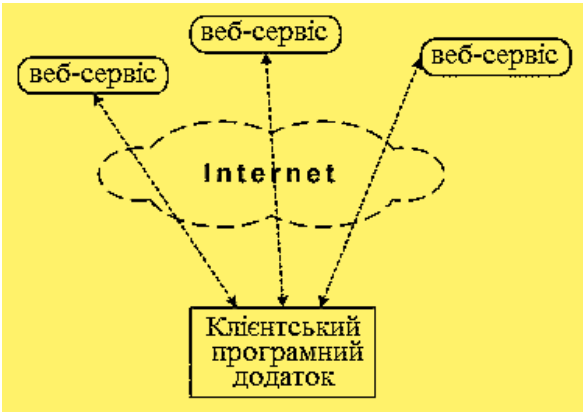

Рисунок 6.1 – Схема використання веб-сервісів

Існують різні технології створення web-сервісів, зокрема RPC, SOAP, REST.

В даній лабораторній роботі ми розглянемо REST технологію, яка сьогодні, отримала широке поширення в Web програмуванні.

6.1.1 **REST технології**

На даний момент дуже поширена думка, що REST це навіть не технологія, а архітектура.

В архітектурі REST (Representational State Transfer) зазвичай присутні сервер REST і клієнт REST. Сервер забезпечує доступ до ресурсів, а клієнт звертається до ресурсів, або змінює їх.

REST дозволяє, щоб ресурси мали різні представлення, наприклад, текст, XML, JSON. Доступ до ресурсу здійснюється через загальний інтерфейс, що заснований на стандартах протоколу HTTP. Кожен ресурс повинен підтримувати загальні операції HTTP. Ресурси ідентифікуються глобальними ідентифікаторами (як правило, URI). Клієнт REST може запросити конкретне представлення ресурсу через протокол HTTP.

6.1.1.1 HTTP методи Rest

Типовим для REST архітектури є використання методів PUT, GET, POST і DELETE.

- GET визначає читання ресурсу без побічних ефектів. Ресурс ніколи не змінюється за допомогою запиту GET.
- PUT створює новий ресурс.
- DELETE видаляє ресурси.
- POST оновлює існуючий ресурс або створює новий.

6.1.1.2 RESTful веб-сервіси

RESTful веб-сервіси реалізують концепцію REST і вимагають дотримання певних правил, зокрема, незалежність сервера від стану клієнта та вимоги до побудови інтерфесу сервіса, зокрема корректне використання http операцій (GET, PUT, …).

Підтримка REST у Java визначена специфікацією Java Specification Request (JSR) 311, що має назву JAX-RS (Java-API для RESTful веб-сервісів).

Прикладом реалізації JAX-RS є Jersey, що надає бібліотеку для реалізації RESTful веб-сервісів в контейнері сервлетів Java, а також клієнтську бібліотеку для зв'язку з RESTful веб-сервісом. Сервіс реалізується на стороні сервера у вигляді класів Jersey сервлетів.

Jersey сервлет аналізує вхідний запит HTTP і вибирає клас і метод, щоб відповісти на цей запит. Цей вибір ґрунтується на JAX-RS анотаціях, що використовуються у класах і методах.

6.1.1.3 JAX-RS аннотации

Нижче наведено найбільш важливі JAX-RS анотації.

@PATH(parm) – значення параметеру parm цієї анотації є унікальним ідентифікатором сервлету або методу. Ці ідентифікатори є складовими частинами URL, що використовується для звернення до сервісу і мають починатися символом косої риски. Наприклад, @PATH(“/students”).

@GET – визначає, що метод буде відповідати на запит HTTP GET.

@PUT – визначає, що метод буде відповідати на запит HTTP PUT.

@DELETE – визначає, що метод буде відповідати на запит HTTP DELETE.

@POST – визначає, що метод буде відповідати на запит HTTP POST.

@Produces(mimetype) – значення параметру mimetype визначає MIME тип результату, що повертається методом з анотацією @GET. Прикладами значень цього параметру можуть бути MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML, MediaType.TEXT_PLAIN.

@Consumes(mimetype) – значення параметру mimetype визначає MIME тип параметра, що приймає метод.

@PathParm(parmName) та @QueryParam(parmName) – використовується у списку параметрів метода для визначення назви параметра, що передається до методу через URL.

6.2 Порядок виконання роботи

6.2.1 **Створення REST веб-сервісу**

З метою отримання навичок реалізації REST-технологій створімо веб-сервіс, який забезпечує виконання CRUD операцій з базою даних, що відповідає вибраному варіанту.

6.2.1.1 Створення проєкту

Створімо новий Dynamic Web Project, наприклад, Lab6, скориставшись рекомендаціями попередньої лабораторної роботи і не забувши згенерувати файл web.xml. У цьому проєкті він нам знадобиться.

6.2.1.2 Підключення бібіліотеки Jersey до проєкту

Завантажте Jersey бібліотеку з сайту https://jersey.java.net/download.html або зі сторінки курсу на ресурсі мудл, папка jersey.

Скопіюйте із отриманого архіву усі jar-файли з папок api, ext та lib у папку WEB-INF/lib створеного проєкту.

Додайте файли бібліотеки до Build Path проєкту.

6.2.1.3 Підключення драйвера бази даних

Додайте до бібліотеки WEB-INF/lib драйвер для роботи з базою даних, у нашому випадку derby.jar. Драйвер має бути той самий, за допомогою якого ствоювалася база даних.

Додайте драйвер до Build Path проєкту.

6.2.1.4 Створення пакетів для Java класів

Створімо у розділі Java Resources проєкту пакет service для Java класу сервіса і скопіюйте до папки src пакет controller із другої лабораторної роботи. Там мають бути класи Controller та DbConnector. Сервіс буде використовувати ці

класи для роботи з базою даних..

6.2.1.5 Доопрацювання файлу web.xml

Відкрийте вихідний текст файлу web.xml, що має бути розташований у папці WEB-INF розділу Web Content проєкту.

Елемент <display-name> документу має містити назву вашого проєкту. Але його можна змінити і саме це ім'я буде входити до складу URI, через який буде викликатися наш сервіс. У нашому прикладі це буде Lab6.

Дані елемента <welcome-file-list> ми не будемо поки що використовувати, але вони нам не заважатимуть.

Далі до документу слід додати такі елементи:

```
<servlet>
  <servlet-name>Jersey RESTful Application</servlet-name>
  <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
  <init-param>
    <param-name>jersey.config.server.provider.packages</param-name>
    <param-value>service</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>Jersey RESTful Application</servlet-name>
  <url-pattern>/rest/*</url-pattern>
</servlet-mapping>
```

У фрагменті жовтим кольором позначено назву папки (service), де знаходиться клас, що визначає функціонал сервісу, та назву складової URI, після якої має знаходитися ім'я сервісу(/rest). Решта тексту є стандартною.

Після цього можна зберегти файл.

Таким чином, початкова частина URI, що буде ідентифікувати наш сервіс буде виглядати так: http://localhost:8080/Lab6/rest/...

6.2.1.6 Створення початкової версії класа Lab6Service

Цей клас буде реалізовувати повний функціонал сервіса, але для початку створімо коротку версію класу, яка дозволить познайомитися з методикою тестування сервіса.

Текст початкової версії наведено нижче. Для реалізації імпорту вибирайте класи з пакетів javax.ws.rs та com.fasterxml.jackson.

```
@Path("/lab6")
public class Lab6Service {
  @GET
  @Path("/test")
  @Produces(MediaType.TEXT_PLAIN)
  public String test() {
     return "Test done!";
  }
}
```

Анотація `@Path("/lab6")` перед заголовком класа означає, що початкова частина URI, через яку забезпечується звертатання до методів класу буде буде

виглядати так: http://localhost:8080/Lab6/rest/lab6/...

У класі реалізовано тільки один метод test(), який повертає “Test done!”.

Перед методом прописані анотації, які визначають, що метод можна викликати через запит HTTP GET. Для виклику використовується шлях `"/test".` Результатом запита буде простий текст.

URI для звертання до цього метода буде виглядати так:
http://localhost:8080/Lab6/rest/lab6/test

6.2.1.7 Створення war-файла і завантаження його до веб-сервера

За допомогою функції меню File → export → Web → War File перейти до діалогу створення та завантаження war-файлу проєкта, рисунок 6.2.

У діалозі треба визначити ім’я файлу та визначитися з місцем його завантаження. Ім’я файлу співпадає з ім’ям проєкту а завантажити його слід у папку webapps сервера apache-tomcat.

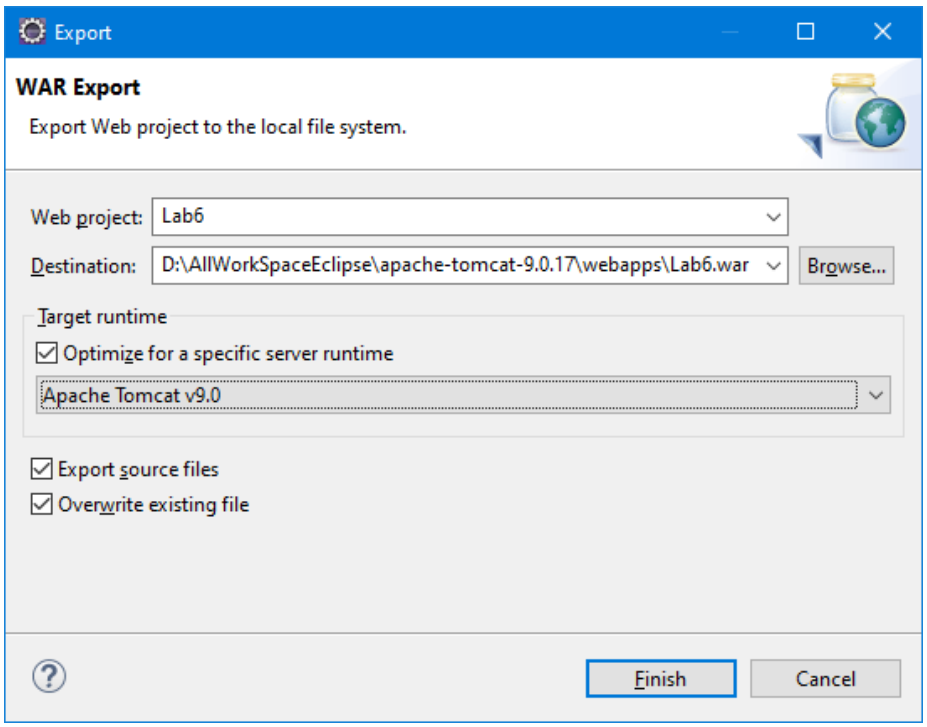

Рисунок 6.2 – Діалог створення та завантаження war-файлу проєкту

Далі слід активізувати сервер, викликавши файл startup.bat через вікно cmd.

Можливо знадобиться налаштувати змінні оточення CATALINA_HOME та JRE_HOME, у яких треба вказати папки, де знаходяться apache-tomcat та jre.

Майте на увазі, що після налаштувань змінних оточення, cmd требе перезапустити.

Можливо також, що доведеться поміняти порт 8080.

Після старту сервер має розгорнути web-архів і створити папку з назвою проєкту у папці webapps.

Перевірте наявність такої папки. Якщо папку не створено, почитайте

повідомлення сервера у папці log. Можливо версії сервера і бібліотек несумісні.

6.2.1.8 Тестування сервіса

У нашому сервісі використовується http операція GET. Такі сервіси можна викликати через звичайний браузер, або навіть із цього тексту, звернувшись за посиланням (якщо співпадає порт):

http://localhost:8080/Lab6/rest/lab6/test

Але інші http методи браузер не пропускає. Тому для тестування сервісу скористаймося додатком Postman до браузера Chrome, який можна скачати за адресою

http://www.getpostman.com/

Зовнішній вигляд частини вікна додатку наведено на рисунку 6.3.

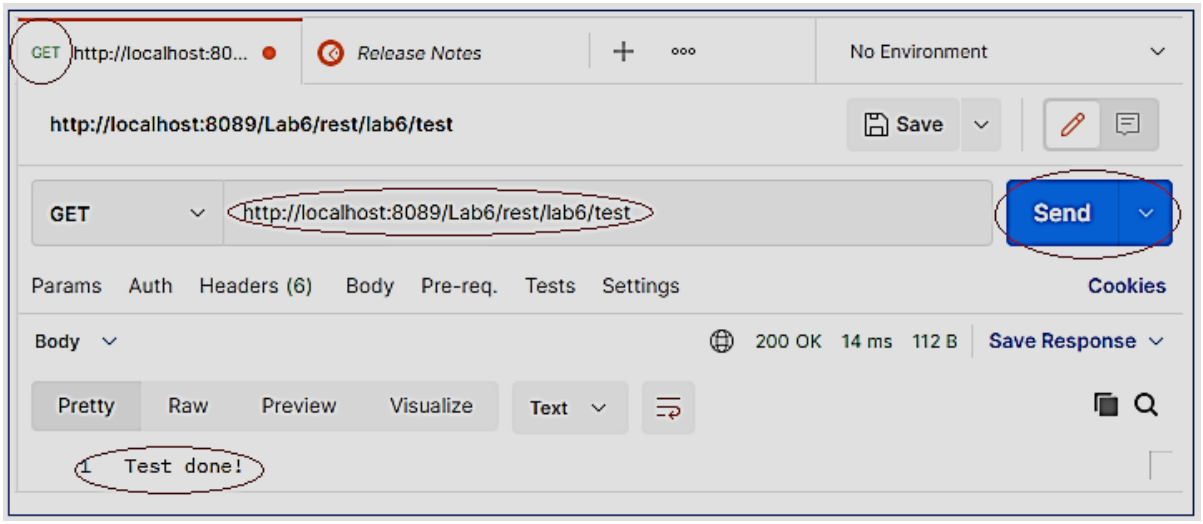

Рисунок 6.3 – Імітатор клієнта Postman

Postman дозволяє вибрати метод, ввести URI, передати сервісу повідомлення та переглянути результат.

Відповідні елементи управління, поле введення URI та результат звернення до нашого метода показані на рисунку 6.3.

Якщо імітація запита пройшла успішно, це означає, що сервер працює нормально, необхідні бібліотекі підключені і можна розширювати функціонал сервіса.

6.2.1.9 Функціонал сервісу

Реалізуймо у сервісі функціонал, який дозволить звертатися до методів класу Controller, який було створено у другій лабораторній роботі і скопіювано до цього проєкту. Для цього треба забезпечити виклик через веб сервіс таких методів:

- tableExist(String tableName);
- createTable(String tableName);
- executeQuery(String query);
- executeUpdate(String query);

Врахуємо також необхідність передачі сервісу в якості параметра повної назви бази даних. Інакше нам доведеться визначати це ім’я в коді сервісу, що не зручно для перевірки.

6.2.1.10 Реалізація запиту про існування таблиці БД

Створімо у класі Lab6Service метод existTable для реалізації відповідного запиту:

```
@GET
@Path("/exist/{table}")
@Produces(MediaType.TEXT_PLAIN)
public String tableExist(
     @PathParam("table") String table,
     @QueryParam("db") String db  ) {

   DbConnector.setDbFullName(db);
   Boolean b = Controller.tableExist(table.toUpperCase());
   return b.toString();
}
```

.Метод буде викликатися через GET запит і має два параметри. Перший параметр передається як елемент шляху (@PathParam) і використовується як параметр методу tableExist контролера.

Другий параметр має передаватися через ім’я параметру (@QueryParam) і використовується для з’єднання з базою даних. Цей параметр містить символи ‘/’, тому передати його як елемент шляху проблематично.

Після створення нового методу у класі проєкт слід заново експротувати на сервер. Не забувайте робити це і після створення інших методів.

Після завантаження war файлу на сервер нову функцію можна протестувати через Postmen/

Виклик цієї функції сервісу для перевірки наявості таблиці job у базі даних d:/1/dbByv буде виглядати так:

http://localhost:8080/Lab6/rest/lab6/exist/job/?db=d:/1/dbByv

Протестуйте цю функцію для своєї бази даних і результат зафіксуйте у звіті.

6.2.1.11 Реалізація запита на створення таблиці у базі даних

Відповідний метод має такі самі параметри, але має використовувати метод createTable контролера і повертати ціле число.

Подумайте, яку htpp операцію тут використати і реалізуйте відповідний метод самостійно. Відповідний елемент шляху назвіть create.

Протестуйте метод створивши нову базу і результат наведіть у звіті.

6.2.1.12 Реалізація передачі sql запиту до бази даних

Текст відповідного метода наведено нижче.

```
@GET
```

```
@Path("/execute/")
@Produces(MediaType.TEXT_PLAIN)
public String execute(
        @QueryParam("db") String db,
        @QueryParam("query") String query) {

   DbConnector.setDbFullName(db);
   List<Map<String, Object>> mapList =
        Controller.executeQuery(query);
   return mapList.toString();
}
```

Для тестування цієї функції сервіса слід використати метод GET з URI, який подібний до наступного:

http://localhost:8080/Lab6/rest/lab6/execute/?db=d:/1/dbByv&query=select * from job

Тут робимо запит на отримання усіх записів із таблиці job у базі d:\1\dbByv.

Студенти мають реалізувати запит до своєї бази и результати додати до звіту.

6.2.1.13 Реалізація запиту на оновлення інформації в базі

Відповідний метод має такі самі параметри як і попередній, але має використовувати метод executeUpdate контролера і повертати ціле число.

Подумайте, яку htpp операцію тут використати і реалізуйте відповідний метод самостійно. Відповідний елемент шляху назвіть update.

Протестуйте метод для запитів на додавання, видалення та редагування і результати тестування через Postmen наведіть у звіті.

6.2.2 **Створення клієнта для REST сервісу**

Клієнтом веб сервісу може бути будь який застосунок. Ми реалізуємо клієнта використовуючи візуальну частину і класи пакету query з другої лабораторної роботи.

6.2.2.1 Створення проєкту

Створімо звичайний Java проєкт.

Скопіюймо до проєкту пакети controller, query, resource та view з другої лабораторної роботи.

6.2.2.2 Підключення бібліотек для реалізації REST клієнта

Для реалізації цього завдання і підключимо до нього такі бібліотеки:
Apache HttpClient (**httpclient-*.jar**), адреса для отримання:
http://mvnrepository.com/artifact/org.apache.httpcomponents/httpclient
Apache HttpCore (**httpcore-*.jar**), адреса для отримання:
http://mvnrepository.com/artifact/org.apache.httpcomponents/httpcore
Apache Commons (**commons-loggin-*.jar**), адреса для отримання:

http://mvnrepository.com/artifact/commons-logging/commons-logging/1.2
Ці jar файли можна також знайти на сторінці курса на мудлі.

6.2.2.3 Підключення бібліотеки для роботи з JSON

Для роботи з JSON скористаймось файлом org.json-chargebee-1.0.jar, що використовувався в третій лабораторній роботі. Цей файл можна скачати з інтернету або завантажити з мудла.

Після підключення бібліотек можна перейти до створення REST клієнта.

6.2.2.4 Підготовка до реалізація клієнта у класі Controller

Виходячі з того, що ми використовуємо візуальну частину другої лабораторної роботи, новий Controller повинен мати той самий інтерфейс, що і в другій роботі. Але методи, які використовують зв'язок з базою даних мають бути іншими, тому що наш клієнт буде користуватися веб сервісом.

Перш за все видалимо у пакеті controller клас dbConnector. Як наслідок у проєкті мають з'явитися помилки у методах, які використовували dbConnector. Саме ці методи нам потрібно буде модіфікувати.

6.2.2.5 Визначення назви бази даних

Створюючи веб сервіс ми вирішил, шо повне ім'я бази даних буде передаватися сервісу як параметр. Тому визначимо у контролері таке поле і методи доступу до нього. Геттер та сеттер можна згенерувати, але геттер після цього треба доопрацювати.

```
private static String dbFullName = null;

public static void setDbFullName(String dbFllName) {
   RestController.dbFullName =dbFllName;
}
private static String getDbFullName() {
   if(dbFullName==null) {
      dbFullName = JOptionPane.showInputDialog(null,
           "Enter DB full name", "d:/1/dbByv");
   }
   return dbFullName;
}
```

6.2.2.6 Доопрацювання методу візуальної частини

Після видалення DbConnector у методі візуальної частини з'явилась помилка. Код методу поміняємо на такий:

```
protected void onSetDbFullName() {
   String dbName = JOptionPane.showInputDialog(
        "Enter DB full name", "d:/1/dbBiv");
   Controller.setDbFullName(dbName);
}
```

6.2.2.7 Реалізація допоміжного методу

Старий контролер використовував метод для перетворення ResultSet у список карт. Цей метод нам не потрібен і його можна видалити. Натомість нам потрібен метод для перетворення рядка формату JSON у список карт. Створімо його:

```
private static List<Map<String, Object>>
                stringToMapList(String result) {
  List<Map<String, Object>> list = new ArrayList<>();
  try {
     JSONArray ar = new JSONArray(result);
     for (int i = 0; i < ar.length(); i++) {
        JSONObject obj = (JSONObject) ar.get(i);
        Map<String, Object> map = new LinkedHashMap<>();
        Iterator<String> itr = obj.keys();
        while (itr.hasNext()) {
           String key =  itr.next();
           map.put(key,obj.get(key));
        }
        list.add(map);
     }
  } catch (JSONException e) {
     e.printStackTrace();
  }
  return list;
}
```

6.2.2.8 Створення об’єкта HTTPClient

Створімо об’єкт класу `HttpClient` і статичний метод, який буде приймати URI запити та повертати результат у вигляді рядка символів. Метод також прийме на себе обробку виключнх ситуацій.

Створімо також посилання на об’єкт URIBuilder, Який ми будемо використовувати для формування uri.

```
private static HttpClient httpClient
             = HttpClientBuilder.create().build();
private static URIBuilder bd;

public static String executeRequest(HttpRequestBase request){
  try {
     HttpResponse response = httpClient.execute(request);
     int code = response.getStatusLine().getStatusCode();
     if (code != 200)
        return "ERROR! "+response.getStatusLine().toString();
     HttpEntity entity = response.getEntity();
     return EntityUtils.toString(entity);
  } catch (Exception e) {
     return "ERROR! "+e.getStackTrace()[0].toString();
  }
```

```
}
```

6.2.2.9 Нова реалізаія методу tableExist

У цьому методі перший параметр передається значенням, а другий через ім’я.

```
public static boolean tableExist(String tableName) {
   String db = getDbName();
   String uri = "http://localhost:8080/Lab6"
        + "/rest/lab6/exist/"+ tableName +"/";
   try {
      bd = new URIBuilder(uri);
      bd.addParameter("db", db);
      String res = executeRequest(new HttpGet(bd.build()));
      return res != null && res.equals("true");
   } catch (URISyntaxException e) {
      e.printStackTrace();
      return false;
   }
}
```

6.2.2.10 Нова реалізація методу executeQuery

Тут два параметри передаються через ім’я.

```
public static List<Map<String, Object>> executeQuery(String query) {
   String db = getDbName();
   String uri = "http://localhost:8080/Lab6"
        + "/rest/lab6/execute/";
   String result="";
   try {
      bd = new URIBuilder(uri);
      bd.addParameter("db", db);
      bd.addParameter("query", query);
      result = executeRequest(new HttpGet(bd.build()));
      return stringToMapList(result);
   } catch (URISyntaxException e) {
      e.printStackTrace();
      return new ArrayList<>();
   }
}
```

6.2.2.11 Нова реалізація методу executeUpdate

```
public static int executeUpdate(String query) {
   String db = getDbName();
   String uri = "http://localhost:8080/Lab6"
        + "/rest/lab6/update/";
   try {
      bd = new URIBuilder(uri);
      bd.addParameter("db", db);
      bd.addParameter("query", query);
```

```
            String res = executeRequest(new HttpPost(bd.build()));
            return Integer.parseInt(res);
        } catch (Exception e) {
            e.printStackTrace();
            return 0;
        }
    }
```

Тестування клієнта

Решта методів контролера залишається без змін

Після створення клієнта можна протестувати застосунок. Протестуйте усі операції з базою. Все має працювати.

6.3 Вимоги до звіту

– Назва роботи.
– Мета роботи.
– Тексти класів сервісу та результати їх тестування через Postmen..
– Результати тестування довільних запитів до своєї бази через клієнта.
– Висновки.

6.4 Контрольні питання

1. Що таке веб-сервіс.
2. Характеристика REST-технології.
3. Призначення web.xml.
4. JAX-RS аннотации.
5. Формування запитів до сервісу з разними типами параметрва.
6. В чому різниці між контролерами у другій та шостій роботах.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. REST. [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/REST
2. Fielding, Roy Thomas (2000). "Representational State Transfer (REST)". Architectural Styles and the Design of Network-based Software Architectures (PhD). University of California, Irvine.
3. Building a RESTful Web Service. [Електронний ресурс]. – Режим доступу: https://spring.io/guides/gs/rest-service/
4. JAX-RS: Advanced Topics and an Example. [Електронний ресурс]. – Режим доступу: https://docs.oracle.com/javaee/7/tutorial/jaxrs-advanced.htm
5. Jersey (JAX-RS) Tutorials. [Електронний ресурс]. – Режим доступу: https://howtodoinjava.com/jersey-jax-rs-tutorials/