



Java course

- [Начало Java](#)
- [Проект «Отдел кадров»](#)
- [Курсы](#)
- [Статьи](#)
- [Контакты/Вопросы](#)

- [Введение](#)
- [Установка JDK](#)
- [Основные шаги](#)
- [Данные](#)
- [Порядок операций](#)
- [IDE NetBeans](#)
- [ООП](#)
- [Инкапсуляция](#)
- [Наследование](#)
- [Пакеты](#)
- [Переопределение и перегрузка](#)
- [Полиморфизм](#)
- [Статические свойства и методы](#)
- [Отношения между классами](#)
- [Визуализация работа](#)
- [Пример — очередь объектов](#)
- [Массивы — знакомство](#)
- [Многомерные массивы](#)
- [Абстрактные классы](#)
- [Интерфейсы](#)

Список контактов — работаем с базой данных

В этом разделе мы объединим большую часть накопленных знаний по технологиям и сделаем новый вариант нашего списка контактов. Я собираюсь сделать несколько вещей:

1. Реализовать многоязычную версию для нашего приложения
2. Создать DAO, который будет взаимодействовать с базой данных
3. Использовать понятие reflection для возможности конфигурировать DAO

Многоязычная версия

Хочу сразу предупредить — мой вариант не надо рассматривать, как единственный в своем роде. Вы можете найти и другие интересные решения, которые вполне вероятно, вам могут понравиться больше моего.

То, что я хочу реализовать, имеет следующую идею — создать класс, у которого можно будет спросить строку для компонента по определенной системе именования. В нашем случае у нас есть две формы, в которых мы выводим сообщения. Также у нас есть модель для таблицы. Логично, если у нас имя компонента будет складываться из двух частей:

1. Имя (идентификатор) формы/модели
2. Имя (идентификатор) компонента на этой форме/модели

- [Расширенное описание классов](#)
- [Исключения](#)
- [Решения на основе классов](#)
- [Список контактов — начало](#)
- [Коллекции — базовые принципы](#)
- [Коллекции — продолжение](#)
- [Список контактов — GUI приложение](#)
- [Что такое JAR-файлы](#)
- [Многопоточность — первые шаги](#)
- [Многопоточность и синхронизация](#)
- [Работаем с XML](#)
- [Reflection — основы](#)
- [Установка СУБД PostgreSQL](#)
- [Базы данных на Java — первые шаги](#)
- [Возможности JDBC — второй этап](#)
- [JDBC — групповые операции](#)
- [Список контактов — работаем с БД](#)
- [Переезжаем на Maven](#)
- [Потоки ввода-вывода](#)
- [Сетевое взаимодействие](#)
- [С чего начинается Web](#)

Также есть смысл сразу создать небольшой конфигурационный файл, который нам пригодится для нескольких целей. Во-первых — это установка языка. Во-вторых — нам наверняка потребуется набор параметров для соединения с базой данных. Поэтому имеет смысл сразу предусмотреть такой файл. Что мы и сделаем. Для его обозначения будем использовать термин «глобальный». Может слишком громко для такого небольшого файла, но в данном случае «глобальный» означает — он для всех компонентов нашей программы и не говорит о его размерах.

Давайте приступим. Первым мы сделаем класс, который будет отвечать за нашу «глобальную» конфигурацию. Его задача — загрузить данные из конфигурационного файла и потом отдавать нужные параметры по запросу. Он вряд ли у нас будет сложный — вот такой:

```

1 package edu.javacourse.contact.config;
2
3 import java.io.FileReader;
4 import java.io.IOException;
5 import java.util.Properties;
6
7 public class GlobalConfig
8 {
9     private static final String CONFIG_NAME = "contacts.properties";
10    private static final Properties GLOBAL_CONFIG = new Properties();
11
12    // Сделать начальную загрузку параметров из файла по умолчанию
13    public static void initGlobalConfig() throws IOException {
14        initGlobalConfig(null);
15    }
16
17    // Сделать загрузку данных из конфигурационного файла, имя которого передано в виде параметра
18    // Если имя null или пустое - берем файл по умолчанию.
19    public static void initGlobalConfig(String name) throws IOException {

```

```

20     if (name != null && !name.trim().isEmpty()) {
21         GLOBAL_CONFIG.load(new FileReader(name));
22     } else {
23         GLOBAL_CONFIG.load(new FileReader(CONFIG_NAME));
24     }
25 }
26
27 // Получить значение параметра из глобальной конфигурации по имени
28 public static String getProperty(String property) {
29     return GLOBAL_CONFIG.getProperty(property);
30 }
31 }

```

Наверно самое время понять, что же я тут написал. Потому как некоторые моменты могут вас заинтересовать. Наверняка сразу возник вопрос — почему я сделал два метода **initGlobalConfig**. Резонно. Ответ простой — я сделал это для удобства.

Рассуждал я следующим образом: во-первых, не иметь возможности подставить программе свой конфигурационный файл — не очень хорошая идея. Во всяком случае мне не нравится. Поэтому появился метод с входным параметром — **initGlobalConfig(String name)**. Как можно видеть, если передали имя, то его и будем использовать. Если же параметр равен **null** или пустой, то берем файл по умолчанию.

Во-вторых — передавать в метод **null** выглядит не очень красиво. Поэтому и появился второй метод без параметров. Его задача — вызывать основной метод и передать туда тот самый «некрасивый» **null**.

Что касается остального кода, то он достаточно простой и если вы все-таки не понимаете, то рекомендую посмотреть раздел [Коллекции — продолжение](#). Там есть раздел про класс **Properties**.

Итак, класс, который загружает «глобальную конфигурацию», мы написали. Настало время для создания класса, который будет отдавать нам строки для наших графических компонентов. Его задача — использовать механизм ресурсов (о котором мы говорили опять же в разделе [Коллекции — продолжение](#)).

Наши ресурсы — это набор файлов с базовым именем **ContactResources**. Я не стал создавать много файлов — сделал всего два. Один по умолчанию с русскими именами, второй — с английскими.

1. ContactResources.properties
2. ContactResources_en.properties

В конце я конечно же дам ссылку на код всего приложения, но тем не менее, хочу напомнить, что русские буквы в файлах properties заменяются на их код в Юникоде, поэтому выглядеть это будет не очень красиво. Вот такой вот некрасивый у нас файл **ContactResources.properties**

```

1 frame.refresh=\u041e\u0431\u0434\u043e\u0432\u0438\u0432\u0438\u0442\u044c
2 frame.add=\u0410\u0434\u0434\u0430\u0432\u0438\u0432\u0438\u0432\u0438\u0442\u044c
3 frame.update=\u0410\u0437\u0434\u0430\u0432\u0438\u0432\u0438\u0442\u044c
4 frame.delete=\u0414\u0435\u043b\u0438\u0432\u0438\u0442\u044c
5
6 dialog.givenname=\u0418\u043c\u044f

```

```
7 dialog.surname=\u0424\u0430\u043c\u0438\u043b\u0438\u044f
8 dialog.phone=\u0422\u0435\u043b\u0435\u0444\u043e\u043d
9 dialog.email=e-mail
10
11 model.id=ID
12 model.givenname=\u0418\u043c\u044f
13 model.surname=\u0424\u0430\u043c\u0438\u043b\u0438\u044f
14 model.phone=\u0422\u0435\u043b\u0435\u0444\u043e\u043d
15 model.email=e-mail
```

Содержимое на самом деле будет интерпретироваться вот так:

```
1 frame.refresh=Обновить
2 frame.add=Добавить
3 frame.update=Изменить
4 frame.delete=Удалить
5
6 dialog.givenname=Имя
7 dialog.surname=Фамилия
8 dialog.phone=Телефон
9 dialog.email=e-mail
10
11 model.id=ID
12 model.givenname=Имя
13 model.surname=Фамилия
14 model.phone=Телефон
15 model.email=e-mail
```

Наши ресурсы разбиты на три группы. Первая (frame) касается основной формы со списком контактов. Вторая (dialog) — диалога для ввода данных. Ну и третья (model) — для модели отображения наших данных в таблице. Теперь посмотрим на код нашего класса для работы с этими именами.

```
1 package edu.javacourse.contact.gui;
2
3 import edu.javacourse.contact.config.GlobalConfig;
4 import java.util.Locale;
5 import java.util.PropertyResourceBundle;
6
7 public class GuiResource
8 {
9     private static final String RESOURCES = "edu.javacourse.contact.gui.ContactResources";
```

```

10     private static final String LANGUAGE = "language";
11
12     private static PropertyResourceBundle components = null;
13
14     // Загрузка ресурсов для компонентов
15     public static void initComponents() {
16         String lang = GlobalConfig.getProperty(LANGUAGE);
17         if (lang != null && !lang.trim().isEmpty()) {
18             components = (PropertyResourceBundle) PropertyResourceBundle.getBundle(RESOURCES, new Locale(lang));
19         } else {
20             components = (PropertyResourceBundle) PropertyResourceBundle.getBundle(RESOURCES);
21         }
22     }
23
24     // Получение строки для отображения компонента
25     public static String getLabel(String formId, String componentId) {
26         return components.getString(formId + "." + componentId);
27     }
28 }

```

У нас здесь два метода. Первый — **initComponentResources** — предназначен для загрузки наших ресурсов. В первой строке метода мы получаем из «глобальной конфигурации» язык, который собираемся использовать и потом загружаем ресурсы для выбранного языка. Если язык не указан, то мы загружаем ресурсы по умолчанию (без указания локализации).

Теперь мы должны во всех местах, где раньше жестко прописывали строки, заменить их на вызовы **GuiResource.getLabel(formId, componentId)**. Эти изменения мы должны сделать в наших файлах **ContactFrame.java**, **RditContactDialog.java** и **ContactModel.java**.

Т.к. код этих классов достаточно громоздкий, а изменения небольшие, я не стану приводить его в тексте статьи. Вы сможете посмотреть в полном примере. Просто поищите по коду строку **GuiResource.getLabel** и сможете разобраться что мы поменяли.

DAO для работы с базой данных

Разобравшись с многоязычностью, перейдем к написанию DAO для работы с базой данных. Всю необходимую информацию, которая потребуется нам для написания кода, мы уже разбирали в статьях, посвященных работе с базами данных. Так что в данном разделе мы будем просто писать код. Хотя кое-какие моменты нам все-таки потребуется разобрать. Если вы какие-то моменты вы уже не помните — рекомендую вам их освежить в памяти.

Для начала нам надо написать несложный скрипт для создания таблицы для наших контактов. Я сделал этот скрипт для PostgreSQL — если вам захочется попробовать другую базу данных, то придется что-то поменять. В этом скрипте две части. первая удаляет таблицу в случае, если она есть. Вторая часть — создает. Мы уже встречались с этим скриптом в разделе, посвященном PostgreSQL — [Установка PostgreSQL](#).

```

1 DROP TABLE IF EXISTS JC_CONTACT;
2

```

```
3 CREATE TABLE JC_CONTACT
4 (
5     CONTACT_ID SERIAL,
6     FIRST_NAME VARCHAR(50) NOT NULL,
7     LAST_NAME VARCHAR(50) NOT NULL,
8     PHONE VARCHAR(50) NOT NULL,
9     EMAIL VARCHAR(50) NOT NULL,
10    PRIMARY KEY (CONTACT_ID)
11 );
```

Теперь мы можем приступить к созданию нашего DAO для работы с базой данных. Как вы возможно помните, нам необходимо реализовать интерфейс **ContactDAO**. Но прежде чем мы погрузимся в создание нашего класса, я хочу заострить ваше внимание на одном очень непростом моменте. Это работа с соединением с базой данных. Точнее, это реализация интерфейса **java.sql.Connection**. Приведу для демонстрации пример получения коннекта:

```
1 Class.forName("org.postgresql.Driver");
2 String url = "jdbc:postgresql://localhost:5432/contactdb";
3 String login = "postgres";
4 String password = "postgres";
5 Connection con = DriverManager.getConnection(url, login, password);
```

С одной стороны может показаться, что нет никаких подводных камней. Но давайте задумаемся вот над каким вопросом: **Когда мы должны создавать наше соединение ?**.

Как вы возможно слышали, получение соединения — операция достаточно дорогая. Сразу возникает мысль — а давайте создадим один коннект и будем им пользоваться. На первый взгляд идея кажется вполне здоровой. Но не торопитесь хвалить себя. Идея крайне опасна. И опасность заключается в одном слове — **МНОГОПОТОЧНОСТЬ**.

Пока наше приложение работает как однопоточное (а на данный момент оно так и есть), то проблем не возникает. Но что может случиться, если соединением начнут пользоваться два и более потоков одновременно? Даже если предположить, что все компоненты для работы с базой данных потокобезопасны, то остается проблема транзакций. Представьте себе, что произойдет, если один поток начнет транзакцию и поставит свойство **autoCommit** в **false**, а второй чуть позже сделает установку на **true**? Будет такая неразбериха, что попытка понять что и как у вас не так отнимет кучу времени.

Поэтому мой вам совет: **Не пытайтесь использовать одно соединение на все приложение**.

Но что же тогда делать, спросите вы? Как все-таки добиться более производительного решения? Ответ на этот вопрос существует — это так называемый пул соединений (**ConnectionPool**), но я вернусь к нему позже. А пока сделаем не совсем оптимально — каждый раз будем создавать соединение. Чтобы мы могли легко модифицировать наш код после знакомства с **ConnectionPool**, я сделаю отдельный метод для получения коннекта. Также я использую нашу «глобальную конфигурацию» для настройки параметров соединения. Вопрос в общем спорный — правильно ли прописывать параметры соединения в глобальную конфигурацию, но я думаю, что в рамках учебного проекта можно так поступить. Хотя если вам захочется усовершенствовать — дерзайте.

Кроме отдельного метода для получения коннекта, я собираюсь сделать также отдельный класс для создания соединения. Причина банальна — мы

чуть выше говорили, что соединение можно создавать по-разному. Значит для этого нам потребуются разные реализации. А где есть разные реализации одной и той же функциональности — возникают шаблоны проектирования. Для начала мы будем просто каждый раз создавать новое соединение. Класс в общем не сложный — чуть ниже вы его увидите. Но т.к. я хочу в будущем использовать пул соединений, который улучшит производительность, то есть смысл подумать над более легко расширяемой реализацией. Мы можем воспользоваться шаблоном «Абстрактная фабрика». Для этого создадим интерфейс **ConnectionFactory**, простую реализацию этого интерфейса **SimpleConnectionFactory** и наконец класс-фабрику **ConnectionFactoryFactory**. Давайте смотреть код.

Первым мы делаем интерфейс **ConnectionFactory**. Здесь все очень просто:

```
1 package edu.javacourse.contact.dao;
2
3 import java.sql.Connection;
4 import java.sql.SQLException;
5
6 public interface ConnectionBuilder
7 {
8     Connection getConnection() throws SQLException;
9 }
```

Реализация будет несколько сложнее, но тоже не займет много места. Загружаем драйвер и используем конфигурацию для соединения.

```
1 package edu.javacourse.contact.dao;
2
3 import edu.javacourse.contact.config.GlobalConfig;
4 import java.sql.Connection;
5 import java.sql.DriverManager;
6 import java.sql.SQLException;
7
8 public class SimpleConnectionFactory implements ConnectionBuilder
9 {
10     public SimpleConnectionFactory() {
11         try {
12             Class.forName(GlobalConfig.getProperty("db.driver.class"));
13         } catch (ClassNotFoundException ex) {
14             ex.printStackTrace();
15         }
16     }
17
18     @Override
19     public Connection getConnection() throws SQLException {
20         String url = GlobalConfig.getProperty("db.url");
21         String login = GlobalConfig.getProperty("db.login");
```

```

22     String password = GlobalConfig.getProperty("db.password");
23     return DriverManager.getConnection(url, login, password);
24 }
25 }

```

Как можно видеть, в методе **getConnection** мы получаем данные из нашей глобальной конфигурации и создаем новое соединение с базой данных. В принципе, мы это уже видели при разборе способов работы с базой данных на Java — ничего нового. И наконец фабрика:

```

1 package edu.javacourse.contact.dao;
2
3 public class ConnectionBuilderFactory
4 {
5     public static ConnectionBuilder getConnectionBuilder() {
6         return new SimpleConnectionBuilder();
7     }
8 }

```

На данный момент мы сделали очень простую фабрику — я думаю, что после того, как мы закончим подключение нашего нового DAO к проекту, у вас появятся идеи по ее усовершенствованию.

Кроме вышеприведенных изменений, мы должны сделать еще кое-какие действия. Раньше мы с вами объявили интерфейс **ContactDAO** и не сделали достаточно важный шаг — не позволили порождать исключения при работе с источником данных. В начале нам это не сильно мешало, но теперь мы работаем с базой данных и эта работа порождает исключения. Игнорировать их было бы не самым хорошим решением. Поэтому я изменил наш интерфейс, добавив туда исключения. Выглядит это так:

```

1 package edu.javacourse.contact.dao;
2
3 import edu.javacourse.contact.entity.Contact;
4 import edu.javacourse.contact.exception.ContactDaoException;
5 import java.util.List;
6
7 /**
8  * Интерфейс для определения функций хранения данных о контактах
9  */
10 public interface ContactDAO
11 {
12     // Добавление контакта - возвращает ID добавленного контакта
13     public Long addContact(Contact contact) throws ContactDaoException;
14     // Редактирование контакта
15     public void updateContact(Contact contact) throws ContactDaoException;
16     // Удаление контакта по его ID

```



```

17     public void deleteContact(Long contactId) throws ContactDaoException;
18     // Получение контакта
19     public Contact getContact(Long contactId) throws ContactDaoException;
20     // Получение списка контактов
21     public List<Contact> findContacts() throws ContactDaoException;
22
23 }

```

Данное изменение порождает дальнейшие шаги. Во-первых, нам надо создать класс исключения. Мы сделаем сразу два класса — для DAO и для бизнес-логики. Пока они вот такие — не будем усложнять себе жизнь.

```

1 package edu.javacourse.contact.exception;
2
3 public class ContactDaoException extends Exception {
4     {
5         public ContactDaoException() {
6         }
7
8         public ContactDaoException(String message) {
9             super(message);
10        }
11
12        public ContactDaoException(Throwable cause) {
13            super(cause);
14        }
15
16        public ContactDaoException(String message, Throwable cause) {
17            super(message, cause);
18        }
19    }

```

```

1 package edu.javacourse.contact.exception;
2
3 public class ContactBusinessException extends Exception {
4     {
5         public ContactBusinessException() {
6         }
7
8         public ContactBusinessException(String message) {
9             super(message);
10        }

```

```

11
12     public ContactBusinessException(Throwable cause) {
13         super(cause);
14     }
15
16     public ContactBusinessException(String message, Throwable cause) {
17         super(message, cause);
18     }
19 }

```

Т.к. теперь наш DAO порождает исключения, то наш класс **ContactManager** тоже слегка изменится — мы теперь обрабатываем исключения от DAO и порождаем исключения **ContactBusinessException**. Теперь он выглядит вот так:

```

1  package edu.javacourse.contact.business;
2
3  import edu.javacourse.contact.dao.ContactDAO;
4  import edu.javacourse.contact.dao.ContactDAOFactory;
5  import edu.javacourse.contact.entity.Contact;
6  import edu.javacourse.contact.exception.ContactBusinessException;
7  import edu.javacourse.contact.exception.ContactDaoException;
8  import java.util.List;
9
10 /**
11  * Класс для реализации функций над списком контактов
12  */
13 public class ContactManager
14 {
15     private final ContactDAO dao;
16
17     public ContactManager() {
18         dao = ContactDAOFactory.getContactDAO();
19     }
20
21     // Добавление контакта - возвращает ID добавленного контакта
22     public Long addContact(Contact contact) throws ContactBusinessException {
23         try {
24             return dao.addContact(contact);
25         } catch (ContactDaoException ex) {
26             throw new ContactBusinessException(ex);
27         }
28     }
29
30     // Редактирование контакта
31     public void updateContact(Contact contact) throws ContactBusinessException {

```

```

32     try {
33         dao.updateContact(contact);
34     } catch (ContactDaoException ex) {
35         throw new ContactBusinessException(ex);
36     }
37 }
38
39 // Удаление контакта по его ID
40 public void deleteContact(Long contactId) throws ContactBusinessException {
41     try {
42         dao.deleteContact(contactId);
43     } catch (ContactDaoException ex) {
44         throw new ContactBusinessException(ex);
45     }
46 }
47
48 // Получение одного контакта
49 public Contact getContact(Long contactId) throws ContactBusinessException {
50     try {
51         return dao.getContact(contactId);
52     } catch (ContactDaoException ex) {
53         throw new ContactBusinessException(ex);
54     }
55 }
56
57 // Получение списка контактов
58 public List<Contact> findContacts() throws ContactBusinessException {
59     try {
60         return dao.findContacts();
61     } catch (ContactDaoException ex) {
62         throw new ContactBusinessException(ex);
63     }
64 }
65 }

```

Нам также придется исправить класс **ContactFrame** — он тоже должен обрабатывать исключения. Смотрим код. Я очень надеюсь, что разобрать его самостоятельно вы сможете.

```

1 package edu.javacourse.contact.gui;
2
3 import edu.javacourse.contact.business.ContactManager;
4 import edu.javacourse.contact.entity.Contact;
5 import edu.javacourse.contact.exception.ContactBusinessException;
6 import java.awt.BorderLayout;

```

```
7 import java.awt.GridBagConstraints;
8 import java.awt.GridBagLayout;
9 import java.awt.Insets;
10 import java.awt.event.ActionEvent;
11 import java.awt.event.ActionListener;
12 import java.util.List;
13 import javax.swing.JButton;
14 import javax.swing.JFrame;
15 import javax.swing.JOptionPane;
16 import javax.swing.JPanel;
17 import javax.swing.JScrollPane;
18 import javax.swing.JTable;
19 import javax.swing.ListSelectionModel;
20
21 public class ContactFrame extends JFrame implements ActionListener
22 {
23     private static final String FRAME = "frame";
24     private static final String C_REFRESH = "refresh";
25     private static final String C_ADD = "add";
26     private static final String C_UPDATE = "update";
27     private static final String C_DELETE = "delete";
28
29     private static final String LOAD = "LOAD";
30     private static final String ADD = "ADD";
31     private static final String EDIT = "EDIT";
32     private static final String DELETE = "DELETE";
33
34     private final ContactManager contactManager = new ContactManager();
35     private final JTable contactTable = new JTable();
36
37     // В конструкторе мы создаем нужные элементы
38     public ContactFrame() {
39         // Выставляем у таблицы свойство, которое позволяет выделить
40         // ТОЛЬКО одну строку в таблице
41         contactTable.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);
42
43         // Используем layout GridBagLayout
44         GridBagLayout gridbag = new GridBagLayout();
45         GridBagConstraints gbc = new GridBagConstraints();
46         // Каждый элемент является последним в строке
47         gbc.gridwidth = GridBagConstraints.REMAINDER;
48         // Элемент раздвигается на весь размер ячейки
49         gbc.fill = GridBagConstraints.BOTH;
50         // Но имеет границы - слева, сверху и справа по 5. Снизу - 0
51         gbc.insets = new Insets(5, 5, 0, 5);
52
53         // Создаем панель для кнопок
```

```

54 JPanel btnPanel = new JPanel();
55 // усанавливаем у него layout
56 btnPanel.setLayout(gridbag);
57 // Создаем кнопки и указываем их заголовки и ActionCommand
58 btnPanel.add(createButton(gridbag, gbc, GuiResource.getLabel(FRAME, C_REFRESH), LOAD));
59 btnPanel.add(createButton(gridbag, gbc, GuiResource.getLabel(FRAME, C_ADD), ADD));
60 btnPanel.add(createButton(gridbag, gbc, GuiResource.getLabel(FRAME, C_UPDATE), EDIT));
61 btnPanel.add(createButton(gridbag, gbc, GuiResource.getLabel(FRAME, C_DELETE), DELETE));
62
63 // Создаем панель для левой колонки с кнопками
64 JPanel left = new JPanel();
65 // Выставляем layout BorderLayout
66 left.setLayout(new BorderLayout());
67 // Кладем панель с кнопками в верхнюю часть
68 left.add(btnPanel, BorderLayout.NORTH);
69 // Кладем панель для левой колонки на форму слева - WEST
70 add(left, BorderLayout.WEST);
71
72 // Кладем панель со скроллингом, внутри которой находится наша таблица
73 // Теперь таблица может скроллиться
74 add(new JScrollPane(contactTable), BorderLayout.CENTER);
75
76 // выставляем координаты формы
77 setBounds(100, 200, 900, 400);
78 // При закрытии формы заканчиваем работу приложения
79 setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
80
81 // Загружаем контакты
82 try {
83     loadContact();
84 } catch (ContactBusinessException ex) {
85     ex.printStackTrace();
86 }
87
88
89 // Метод создает кнопку с заданными характеристиками - заголовок и действие
90 private JButton createButton(GridBagLayout gridbag, GridBagConstraints gbc, String title, String action) {
91     // Создаем кнопку с заданным заголовком
92     JButton button = new JButton(title);
93     // Действие будет проверяться в обработчике и мы будем знать, какую
94     // именно кнопку нажали
95     button.setActionCommand(action);
96     // Обработчиком события от кнопки являемся сама форма
97     button.addActionListener(this);
98     // Выставляем свойства для размещения для кнопки
99     gridbag.setConstraints(button, gbc);
100    return button;

```

```
101     }
102
103     @Override
104     // Обработка нажатий кнопок
105     public void actionPerformed(ActionEvent ae) {
106         // Получаем команду - ActionCommand
107         String action = ae.getActionCommand();
108         // В зависимости от команды выполняем действия
109         try {
110             switch (action) {
111                 // Перегрузка данных
112                 case LOAD:
113                     loadContact();
114                     break;
115                 // Добавление записи
116                 case ADD:
117                     addContact();
118                     break;
119                 // Исправление записи
120                 case EDIT:
121                     editContact();
122                     break;
123                 // Удаление записи
124                 case DELETE:
125                     deleteContact();
126                     break;
127             }
128         } catch (ContactBusinessException ex) {
129             // Очень простой способ показать сообщение
130             JOptionPane.showMessageDialog(this, ex.getMessage());
131         }
132     }
133
134     // Загрузить список контактов
135     private void loadContact() throws ContactBusinessException {
136         // Обращаемся к классу для загрузки списка контактов
137         List<Contact> contacts = contactManager.findContacts();
138         // Создаем модель, которой передаем полученный список
139         ContactModel cm = new ContactModel(contacts);
140         // Передаем нашу модель таблице - и она может ее отображать
141         contactTable.setModel(cm);
142     }
143
144     // Добавление контакта
145     private void addContact() throws ContactBusinessException {
146         // Создаем диалог для ввода данных
147         EditContactDialog ecd = new EditContactDialog();
```

```

148         // Обрабатываем закрытие диалога
149         saveContact (ecd);
150     }
151
152     // Редактирование контакта
153     private void editContact() throws ContactBusinessException {
154         // Получаем выделенную строку
155         int sr = contactTable.getSelectedRow();
156         // если строка выделена - можно ее редактировать
157         if (sr != -1) {
158             // Получаем ID контакта
159             Long id = Long.parseLong(contactTable.getModel().getValueAt(sr, 0).toString());
160             // получаем данные контакта по его ID
161             Contact cnt = contactManager.getContact(id);
162             // Создаем диалог для ввода данных и передаем туда контакт
163             EditContactDialog ecd = new EditContactDialog(contactManager.getContact(id));
164             // Обрабатываем закрытие диалога
165             saveContact (ecd);
166         } else {
167             // Если строка не выделена - сообщаем об этом
168             JOptionPane.showMessageDialog(this, "Вы должны выделить строку для редактирования");
169         }
170     }
171
172     // Удаление контакта
173     private void deleteContact() throws ContactBusinessException {
174         // Получаем выделенную строку
175         int sr = contactTable.getSelectedRow();
176         if (sr != -1) {
177             // Получаем ID контакта
178             Long id = Long.parseLong(contactTable.getModel().getValueAt(sr, 0).toString());
179             // Удаляем контакт
180             contactManager.deleteContact(id);
181             // перезагружаем список контактов
182             loadContact();
183         } else {
184             JOptionPane.showMessageDialog(this, "Вы должны выделить строку для удаления");
185         }
186     }
187
188     // Общий метод для добавления и изменения контакта
189     private void saveContact(EditContactDialog ecd) throws ContactBusinessException {
190         // Если мы нажали кнопку SAVE
191         if (ecd.isSave()) {
192             // Получаем контакт из диалогового окна
193             Contact cnt = ecd.getContact();
194             if (cnt.getContactId() != null) {

```

```

195         // Если ID у контакта есть, то мы его обновляем
196         contactManager.updateContact(cnt);
197     } else {
198         // Если у контакта нет ID - значит он новый и мы его добавляем
199         contactManager.addContact(cnt);
200     }
201     loadContact();
202 }
203 }
204 }

```

Теперь нам осталось создать код для работы с базой данных. Всю необходимую информацию мы уже изучали, так что не буду повторяться — просто приведу код нашего класса. Итак, давайте смотреть наш новый DAO. Единственный момент, который может вызвать вопросы — это блоки **try ... catch**, которые используют возможность, появившуюся в Java 7. Она позволяет автоматически освобождать ресурсы, которые вы задействовали в рамках этого блока. При такой конструкции вам не надо думать об освобождении/закрытии этого ресурса — это произойдет автоматически при любом исходе. Будет все хорошо или вы поймаете исключение — в любом случае ресурс будет закрыт. Причем обратите внимание, что можно задействовать сразу несколько ресурсов, чем я и воспользовался. Очень-очень удобная конструкция — обязательно запомните ее, но учтите, что это работает только начиная с Java 7.

```

1  package edu.javacourse.contact.dao;
2
3  import edu.javacourse.contact.entity.Contact;
4  import edu.javacourse.contact.exception.ContactDaoException;
5  import java.sql.Connection;
6  import java.sql.PreparedStatement;
7  import java.sql.ResultSet;
8  import java.sql.SQLException;
9  import java.util.LinkedList;
10 import java.util.List;
11
12 public class ContactDbDAO implements ContactDAO
13 {
14     private static final String SELECT
15         = "SELECT contact_id, first_name, last_name, phone, email FROM jc_contact ORDER BY first_name, last_name";
16     private static final String SELECT_ONE
17         = "SELECT contact_id, first_name, last_name, phone, email FROM jc_contact WHERE contact_id=?";
18     private static final String INSERT
19         = "INSERT INTO jc_contact (first_name, last_name, phone, email) VALUES (?, ?, ?, ?)";
20     private static final String UPDATE
21         = "UPDATE jc_contact SET first_name=?, last_name=?, phone=?, email=? WHERE contact_id=?";
22     private static final String DELETE
23         = "DELETE FROM jc_contact WHERE contact_id=?";
24 }

```



```

25 private ConnectionBuilder builder = new SimpleConnectionBuilder();
26
27 private Connection getConnection() throws SQLException {
28     return builder.getConnection();
29 }
30
31 @Override
32 public Long addContact(Contact contact) throws ContactDaoException {
33     try (Connection con = getConnection();
34         PreparedStatement pst = con.prepareStatement(INSERT, new String[]{"contact_id"})) {
35         Long contactId = -1L;
36         pst.setString(1, contact.getFirstName());
37         pst.setString(2, contact.getLastName());
38         pst.setString(3, contact.getPhone());
39         pst.setString(4, contact.getEmail());
40         pst.executeUpdate();
41         ResultSet gk = pst.getGeneratedKeys();
42         if (gk.next()) {
43             contactId = gk.getLong("contact_id");
44         }
45         gk.close();
46         return contactId;
47     } catch (Exception e) {
48         throw new ContactDaoException(e);
49     }
50 }
51
52 @Override
53 public void updateContact(Contact contact) throws ContactDaoException {
54     try (Connection con = getConnection();
55         PreparedStatement pst = con.prepareStatement(UPDATE)) {
56         pst.setString(1, contact.getFirstName());
57         pst.setString(2, contact.getLastName());
58         pst.setString(3, contact.getPhone());
59         pst.setString(4, contact.getEmail());
60         pst.setLong(5, contact.getContactId());
61         pst.executeUpdate();
62     } catch (Exception e) {
63         throw new ContactDaoException(e);
64     }
65 }
66
67 @Override
68 public void deleteContact(Long contactId) throws ContactDaoException {
69     try (Connection con = getConnection();
70         PreparedStatement pst = con.prepareStatement(DELETE)) {
71         pst.setLong(1, contactId);

```

```

72         pst.executeUpdate();
73     } catch (Exception e) {
74         throw new ContactDaoException(e);
75     }
76 }
77
78 @Override
79 public Contact getContact(Long contactId) throws ContactDaoException {
80     Contact contact = null;
81     try (Connection con = getConnection()) {
82         PreparedStatement pst = con.prepareStatement(SELECT_ONE);
83         pst.setLong(1, contactId);
84         ResultSet rs = pst.executeQuery();
85         if (rs.next()) {
86             contact = fillContact(rs);
87         }
88         rs.close();
89         pst.close();
90     } catch (Exception e) {
91         throw new ContactDaoException(e);
92     }
93     return contact;
94 }
95
96 @Override
97 public List<Contact> findContacts() throws ContactDaoException {
98     List<Contact> list = new LinkedList<>();
99     try (Connection con = getConnection()) {
100         PreparedStatement pst = con.prepareStatement(SELECT);
101         ResultSet rs = pst.executeQuery();
102         while (rs.next()) {
103             list.add(fillContact(rs));
104         }
105         rs.close();
106     } catch (Exception e) {
107         throw new ContactDaoException(e);
108     }
109     return list;
110 }
111
112 private Contact fillContact(ResultSet rs) throws SQLException {
113     Contact contact = new Contact();
114     contact.setContactId(rs.getLong("contact_id"));
115     contact.setFirstName(rs.getString("first_name"));
116     contact.setLastName(rs.getString("last_name"));
117     contact.setPhone(rs.getString("phone"));
118     contact.setEmail(rs.getString("email"));

```

```
119     return contact;
120 }
121 }
```

Не писал комментарии, т.к. полагаю, что они не требуются — вся необходимая информация уже была.

Reflection для возможности конфигурировать DAO<

Нам остался последний пункт из нашего плана — сделать выбор DAO конфигурируемым. Для этого мы воспользуемся технологией Reflection, о которой мы уже говорили в разделе [Reflection — основы](#).

Здесь нет ничего сложного — нам надо прописать в конфигурации (я предлагаю использовать глобальную) имя нужного нам класса. Все очень просто.

```
1 package edu.javacourse.contact.dao;
2
3 import edu.javacourse.contact.config.GlobalConfig;
4
5 /**
6  * Фабрика для создания экземпляра ContactDAO
7  */
8 public class ContactDAOFactory
9 {
10     public static ContactDAO getContactDAO() {
11         try {
12             Class dao = Class.forName(GlobalConfig.getProperty("dao.class"));
13             return (ContactDAO) dao.newInstance();
14         } catch (ClassNotFoundException | InstantiationException | IllegalAccessException ex) {
15             ex.printStackTrace();
16         }
17         return null;
18     }
19 }
```

Для загрузки конфигурации мы дополним наш запускающий класс **ContactTest**. Особо комментировать здесь нечего — код прилагается.

```
1 package edu.javacourse.contact.test;
2
3 import edu.javacourse.contact.config.GlobalConfig;
4 import edu.javacourse.contact.gui.ContactFrame;
```

```

5 import edu.javacourse.contact.gui.GuiResource;
6
7 public class ContactTest
8 {
9     public static void main(String[] args) {
10         // Загружаем конфигурацию из файла и загружаем ресурсы для разных языков
11         try {
12             GlobalConfig.initGlobalConfig();
13             GuiResource.initComponentResources();
14         } catch (Exception ex) {
15             ex.printStackTrace(System.out);
16             return;
17         }
18         ContactFrame cf = new ContactFrame();
19         cf.setVisible(true);
20     }
21 }

```

Ну и под конец посмотрим на файл **contact.properties**, который содержит нашу глобальную конфигурацию.

```

1 language=en
2
3 dao.class=edu.javacourse.contact.dao.ContactDbDAO
4
5 db.driver.class=org.postgresql.Driver
6 db.url=jdbc:postgresql://localhost:5432/contactdb
7 db.login=postgres
8 db.password=postgres

```

Как и всегда, вы можете скачать наш проект здесь: [Список контактов](#).

ВАЖНО !!! Что конечно же необходимо учесть — вам надо подключить JAR-файл с драйвером для работы с PostgreSQL. Как это сделать, мы уже разбирали в предыдущих статьях — [Что такое JAR-файлы](#). Удачи.

И теперь нас ждет следующая статья: [Переезжаем на Maven](#)

12 comments to *Список контактов — работаем с БД*



Февраль 15, 2018 at 13:28

Максим says:

В общих чертах понятно, но маленькое замечание. Я, например, обычно не скачиваю Ваш проект, а пытаюсь шаг за шагом следовать тексту статьи. И здесь возникла маленькая неувязка: Вы ни словом не обмолвились о правке класса ContactTest. Конечно оно логично вытекает из контекста, но, когда пытаешься переварить такой огромный объем информации, этот момент благополучно упускаешь. В результате программа не запускалась. Я грешил на файл contacts.properties, что с ним только не делал и только потом сообразил скачать готовый проект и посмотреть, что в нем.

[Reply](#)



o

Февраль 15, 2018 at 17:37

admin says:

Спасибо за замечание — постараюсь учитывать.

[Reply](#)



•

Февраль 15, 2018 at 13:37

Максим says:

PS. Если не ошибаюсь, в архиве ContactProject_03.zip можно удалить класс ContactSimpleDAO.java.

[Reply](#)



o

Февраль 15, 2018 at 17:36

admin says:

Он же никому не мешает — пусть остается.

[Reply](#)



Июль 6, 2018 at 15:28

Сергей says:

Добрый день. Большая Вам благодарность за такие уроки!!!!

Я вот тоже не проект скачивал готовый, а все ручками создавал по ходу изучения/прочтения.

И можно сказать, что с первой компиляции прога заработала 😊

Но вот вопрос появился — не понятно по какому принципу отображаются строки в выводимом окне, по какому столбцу сортировка идет ???
Отображается не по порядку создания (не по столбцу ID, как в pgAdmin).

[Reply.](#)



Июль 7, 2018 at 03:16

admin says:

Если посмотреть код, то можно увидеть, что выборка идет с указанием порядка — ORDER BY first_name, last_name.

[Reply.](#)



Июль 9, 2018 at 09:30

Сергей says:

Да, что-то упустил от радости из вида 😊

[Reply.](#)



Сентябрь 2, 2018 at 13:45

Владимир says:

Здравствуйте. Не работает JAR-файл, созданный в Idea. Драйвер в library files добавил. Какие могут быть причины?

[Reply](#)



o

Сентябрь 3, 2018 at 00:21

admin says:

«Не работает» — это что значит ?

[Reply](#)



■

Сентябрь 5, 2018 at 14:41

Владимир says:

Не запускался. Это от невнимательности было.

[Reply](#)



•

Сентябрь 5, 2018 at 15:10

Владимир says:

Здравствуйте. Проблема возникла со сборкой JAR в IDEA. В IDE база данных работает без проблем, но когда собираю JAR и запускаю, получаю: «No suitable driver found for jdbc:mysql».

Много раз менял настройки в project structure, ничего не помогло. В интернете нахожу только описание аналогичной проблемы, при чём и с Eclipse так же, но ни одного развёрнутого ответа((

[Reply](#)



o

Сентябрь 5, 2018 at 18:08

admin says:

Ну хоть что-то 😊 «No suitable driver found for jdbc:mysql». Скорее всего вы просто не указываете JAR файл с драйвером в момент запуска из командной строки.

[Reply](#)

Leave a reply

Comment

You may use these HTML tags and attributes: ` <abbr title=""> <acronym title=""> <blockquote cite=""> <cite> <code class="" title="" data-url=""> <del datetime=""> <i> <q cite=""> <s> <strike> <pre class="" title="" data-url=""> `

Имя *

E-mail *

Сайт

× пять = 35 

Add comment

Copyright © 2018 [Java Course](#)

Designed by [Blog templates](#), thanks to: [Free WordPress themes for photographers](#), [LizardThemes.com](#) and [Free WordPress real estate themes](#)

