

24.9

PreparedStatement

A `PreparedStatement` enables you to create compiled SQL statements that execute more efficiently than `Statements`. `PreparedStatement`s also can specify parameters, making them more flexible than `Statements`—you can execute the same query repeatedly with different parameter values. For example, in the `books` database, you might want to locate all book titles for an author with a specific last and first name, and you might want to execute that query for several authors. With a `PreparedStatement`, that query is defined as:

```
PreparedStatement authorBooks = connection.prepareStatement(
    "SELECT LastName, FirstName, Title " +
    "FROM Authors INNER JOIN AuthorISBN " +
    "ON Authors.AuthorID=AuthorISBN.AuthorID " +
    "INNER JOIN Titles " +
    "ON AuthorISBN.ISBN=Titles.ISBN " +
    "WHERE LastName = ? AND FirstName = ?");
```

The two question marks (?) in the preceding SQL statement's last line are placeholders for values that will be passed to the database as part of the query. Before executing a `PreparedStatement`, the program must specify the values by using the `PreparedStatement` interface's *set* methods.

For the preceding query, both parameters are strings that can be set with **Prepared-Statement** method `setString` as follows:

```
authorBooks.setString(1, "Deitel");  
authorBooks.setString(2, "Paul");
```

Method `setString`'s first argument represents the parameter number being set, and the second argument is that parameter's value. Parameter numbers are *counted from 1*, starting with the first question mark (?). When the program executes the preceding **Prepared-Statement** with the parameter values set above, the SQL passed to the database is

```
SELECT LastName, FirstName, Title  
FROM Authors INNER JOIN AuthorISBN  
    ON Authors.AuthorID=AuthorISBN.AuthorID  
INNER JOIN Titles  
    ON AuthorISBN.ISBN=Titles.ISBN  
WHERE LastName = 'Deitel' AND FirstName = 'Paul'
```

Method `setString` automatically escapes **String** parameter values as necessary. For example, if the last name is O'Brien, the statement

```
authorBooks.setString(1, "O'Brien");
```

escapes the ' character in O'Brien by replacing it with two single-quote characters, so that the ' appears correctly in the

database.



Performance Tip 24.2

PreparedStatement are more efficient than *Statements* when executing SQL statements multiple times and with different parameter values.



Error-Prevention Tip 24.2

Use *PreparedStatement* with parameters for queries that receive *String* values as arguments to ensure that the *Strings* are quoted properly in the SQL statement.



Error-Prevention Tip 24.3

PreparedStatement help prevent SQL injection attacks, which typically occur in SQL statements that include user input improperly. To avoid this security issue, use *Prepared-Statements* in which user input can be supplied only via parameters—indicated with *?* when creating

a `PreparedStatement`. *Once you've created such a `PreparedStatement`, you can use its set methods to specify the user input as arguments for those parameters.*

Interface `PreparedStatement` provides *set* methods for each supported SQL type. It's important to use the *set* method that's appropriate for the parameter's SQL type in the database—`SQLExceptions` occur when a program attempts to convert a parameter value to an incorrect type.

24.9.1 AddressBook App That Uses PreparedStatement

We now present an AddressBook JavaFX app that enables you to browse existing entries, add new entries and search for entries with a last name that begins with the specified characters. Our addressbook Java DB database (created in [Section 24.5](#)) contains an `Addresses` table with the columns `AddressID`, `FirstName`, `LastName`, `Email` and `PhoneNumber`. The column `AddressID` is an auto-incremented identity column in the `Addresses` table.

24.9.2 Class Person

Our AddressBook loads data into `Person` objects ([Fig. 24.31](#)). Each represents one entry in the addressbook database. The class contains instance variables for the address ID, first name, last name, email address and phone number, as well as *set* and *get* methods for manipulating these fields and a `toString` method that returns the `Person`'s name in the format

```
last name, first name
```

Though we do not use the address ID in this example, we included it in class `Person` for use in [Exercises 24.7–24.8](#).

```
1  // Fig. 24.31: Person.java
2  // Person class that represents an entry in an a
3  public class Person {
4      private int addressID;
5      private String firstName;
6      private String lastName;
7      private String email;
8      private String phoneNumber;
9
10     // constructor
11     public Person() {}
12
13     // constructor
14     public Person(int addressID, String firstName
15         String email, String phoneNumber) {
16         setAddressID(addressID);
17         setFirstName(firstName);
18         setLastName(lastName);
19         setEmail(email);
20         setPhoneNumber(phoneNumber);
21     }
22
23     // sets the addressID
24     public void setAddressID(int addressID) {this
25
26     // returns the addressID
27     public int getAddressID() {return addressID;}
28
29     // sets the firstName
30     public void setFirstName(String firstName) {
31         this.firstName = firstName;
32     }
33
34     // returns the first name
```

```

35     public String getFirstName() {return firstName;
36
37         // sets the lastName
38     public void setLastName(String lastName) {this.lastName = lastName;
39
40         // returns the last name
41     public String getLastName() {return lastName;
42
43         // sets the email address
44     public void setEmail(String email) {this.email = email;
45
46         // returns the email address
47     public String getEmail() {return email;}
48
49         // sets the phone number
50     public void setPhoneNumber(String phoneNumber) {this.phoneNumber = phoneNumber;
51         this.phoneNumber = phoneNumber;
52     }
53
54         // returns the phone number
55     public String getPhoneNumber() {return phoneNumber;}
56
57     // returns the string representation of the Person
58     @Override
59     public String toString()
60     {return getLastName() + ", " + getFirstName() + " " + getPhoneNumber();
61     }

```

Fig. 24.31

Person class that represents an entry in an address book.

24.9.3 Class

PersonQueries

Class `PersonQueries` (Fig. 24.32) manages the **Address Book** application's database connection and creates the `PreparedStatement`s for interacting with the database. Lines 17–19 declare three `PreparedStatement` variables. The constructor (lines 22–47) connects to the database at lines 24–25.

```
1  // Fig. 24.32: PersonQueries.java
2  // PreparedStatement used by the Address Book a
3  import java.sql.Connection;
4  import java.sql.DriverManager;
5  import java.sql.PreparedStatement;
6  import java.sql.ResultSet;
7  import java.sql.SQLException;
8  import java.util.List;
9  import java.util.ArrayList;
10
11  public class PersonQueries {
12  private static final String URL = "jdbc:derby
13  private static final String USERNAME = "deite
14  private static final String PASSWORD = "deite
15
16  private Connection connection; // manages con
17  private PreparedStatement selectAllPeople;
18  private PreparedStatement selectPeopleByLastN
19  private PreparedStatement insertNewPerson;
20
21  // constructor
22  public PersonQueries() {
23      try {
24          connection =
25          DriverManager.getConnection(URL, USE
26
27  // create query that selects all entrie
```



```

28         selectAllPeople = connection.prepareStatement
29             "SELECT * FROM Addresses ORDER BY La
30
31         // create query that selects entries wi
32         // that begin with the specified charac
33         selectPeopleByLastName = connection.pre
34             "SELECT * FROM Addresses WHERE LastN
35             "ORDER BY LastName, FirstName");
36
37         // create insert that adds a new entry
38         insertNewPerson = connection.prepareStatement
39             "INSERT INTO Addresses " +
40             "(FirstName, LastName, Email, PhoneN
41             "VALUES (?, ?, ?, ?)");
42     }
43     catch (SQLException sqlException) {
44         sqlException.printStackTrace();
45         System.exit(1);
46     }
47 }
48
49 // select all of the addresses in the databas
50 public List<Person> getAllPeople() {
51     // executeQuery returns ResultSet containi
52     try (ResultSet resultSet = selectAllPeople
53         List<Person> results = new ArrayList<Pe
54
55         while (resultSet.next()) {
56             results.add(new Person(
57                 resultSet.getInt("AddressID"),
58                 resultSet.getString("FirstName"),
59                 resultSet.getString("LastName"),
60                 resultSet.getString("Email"),
61                 resultSet.getString("PhoneNumber")
62             }
63
64             return results;
65         }
66     catch (SQLException sqlException) {
67         sqlException.printStackTrace();

```

```

        68         }
        69
    70         return null;
        71     }
        72
    73     // select person by last name
    74     public List<Person> getPeopleByLastName(String lastName) {
        75         try {
    76             selectPeopleByLastName.setString(1, lastName);
        77         }
    78         catch (SQLException sqlException) {
    79             sqlException.printStackTrace();
    80             return null;
        81         }
        82
    83         // executeQuery returns ResultSet containing results
    84         try (ResultSet resultSet = selectPeopleByLastName.executeQuery()) {
    85             List<Person> results = new ArrayList<Person>();
        86
    87             while (resultSet.next()) {
    88                 results.add(new Person(
    89                     resultSet.getInt("addressID"),
    90                     resultSet.getString("FirstName"),
    91                     resultSet.getString("LastName"),
    92                     resultSet.getString("Email"),
    93                     resultSet.getString("PhoneNumber")
    94                 ));
        95
    96             return results;
        97         }
    98         catch (SQLException sqlException) {
    99             sqlException.printStackTrace();
    100             return null;
        101         }
        102     }
        103
    104     // add an entry
    105     public int addPerson(String firstName, String lastName,
    106         String email, String phoneNumber) {
        107

```

```

108         // insert the new entry; returns # of row
        109         try {
110             // set parameters
111             insertNewPerson.setString(1, firstName)
112             insertNewPerson.setString(2, lastName)
113             insertNewPerson.setString(3, email);
114             insertNewPerson.setString(4, phoneNumb
        115
116             return insertNewPerson.executeUpdate()
        117         }
118         catch (SQLException sqlException) {
119             sqlException.printStackTrace();
120             return 0;
        121         }
        122     }
        123
124     // close the database connection
    125     public void close() {
        126         try {
127             connection.close();
        128         }
129         catch (SQLException sqlException) {
130             sqlException.printStackTrace();
        131         }
        132     }
        133 }

```

Fig. 24.32

PreparedStatement used by the **Address Book** application.

Creating PreparedStatement

Lines 28–29 invoke `Connection` method `prepareStatement` to create the `PreparedStatement` `selectAllPeople` that selects all the rows in the `Addresses` table and sorts them by last name, then by first name. Lines 33–35 create the `PreparedStatement` `selectPeopleByLastName` with a parameter. This statement uses the SQL `LIKE` operator to search the `Addresses` table by last name. The `?` character specifies the last-name parameter—as you’ll see, the text we set as this parameter’s value will end with `%`, so that the database will return entries for last names that start with the characters entered by the user. Lines 38–41 create the `PreparedStatement` `insertNewPerson` with four parameters that represent the first name, last name, email address and phone number for a new entry. Again, notice the `?` characters used to represent these parameters.

PersonQueries Method `getAllPeople`

Method `getAllPeople` (lines 50–71) executes `PreparedStatement` `selectAllPeople` (line 52) by calling method `executeQuery`, which returns a `ResultSet` containing the rows that match the query (in this

case, all the rows in the `Addresses` table). Lines 55–62 place the query results in an `ArrayList<Person>`, which is returned to the caller at line 64.

PersonQueries Method `getPeopleByLastName`

Method `getPeopleByLastName` (lines 74–102) uses `PreparedStatement` method `setString` to set the parameter of `selectPeopleByLastName` (line 76). Then, line 84 executes the query and lines 87–94 place the query results in an `ArrayList<Person>`. Line 96 returns the `ArrayList` to the caller.

PersonQueries Methods `addPerson` and `close`

Method `addPerson` (lines 105–122) uses `PreparedStatement` method `setString` (lines 111–114) to set the parameters for the `insertNewPerson` `PreparedStatement`. Line 116 uses `PreparedStatement` method `executeUpdate` to update the database by inserting the new record. This method returns an integer indicating the number of rows that were updated (or inserted) in the database. Method `close` (lines 125–132) simply closes the database connection.

24.9.4 AddressBook GUI

Figure 24.33 shows the app's GUI (defined in `AddressBook.fxml`) labeled with its **fx:ids**. Here we point out only the key elements and their event-handler methods, which you'll see in class `AddressBookController` (Fig. 24.34). For the complete layout details, open `AddressBook.fxml` in Scene Builder. The GUI's primary layout is a `BorderPane`. The controller class defines three event-handling methods:

- `addEntryButtonPressed` is called when the **Add Entry** Button is pressed.
- `findButtonPressed` is called when the **Find** Button is pressed.
- `browseAllButtonPressed` is called when the **Browse All** Button is pressed.

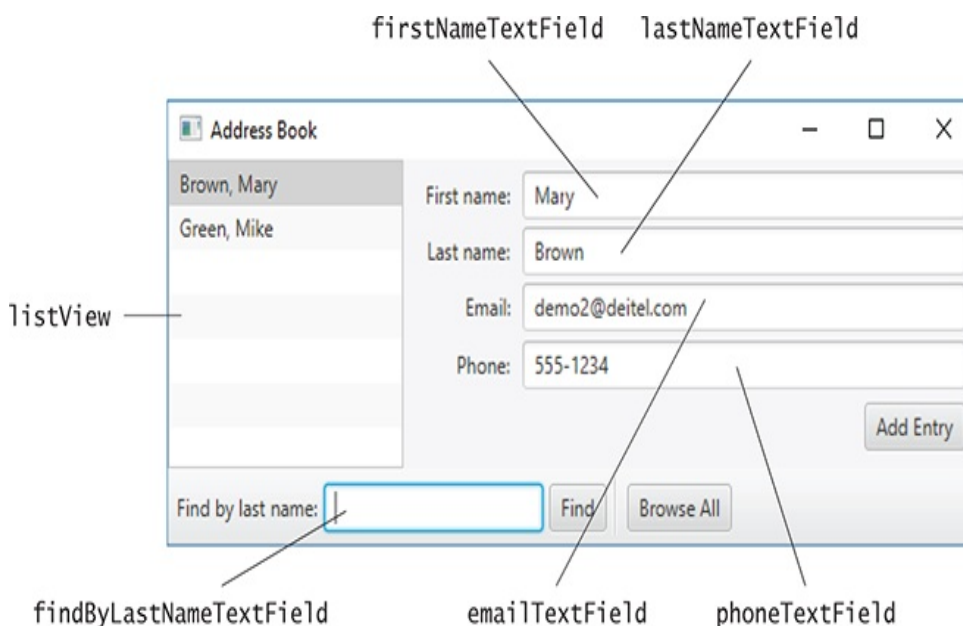


Fig. 24.33

AddressBook GUI with its **fx:ids**.

24.9.5 Class AddressBookController

The `AddressBookController` (Fig. 24.34) class uses a `PersonQueries` object to interact with the database. We do not show the `JavaFX Application` subclass here (located in `AddressBook.java`), because it performs the same tasks you've seen previously to load the app's FXML GUI and initialize the controller.

```
1  // Fig. 24.34: AddressBookController.java
2  // Controller for the AddressBook app
3      import java.util.List;
4      import javafx.application.Platform;
5      import javafx.collections.FXCollections;
6      import javafx.collections.ObservableList;
7      import javafx.event.ActionEvent;
8      import javafx.fxml.FXML;
9      import javafx.scene.control.Alert;
10     import javafx.scene.control.Alert.AlertType;
11     import javafx.scene.control.ListView;
12     import javafx.scene.control.TextField;
13
14     public class AddressBookController {
15         @FXML private ListView<Person> listView; // d
16         @FXML private TextField firstNameTextField;
17         @FXML private TextField lastNameTextField;
18         @FXML private TextField emailTextField;
```

```

19     @FXML private TextField phoneTextField;
20     @FXML private TextField findByLastNameTextFie
21
22     // interacts with the database
23     private final PersonQueries personQueries = n
24
25     // stores list of Person objects that results
26     private final ObservableList<Person> contactL
27     FXCollections.observableArrayList();
28
29     // populate listView and set up listener for
30     public void initialize() {
31         listView.setItems(contactList); // bind to
32         getAllEntries(); // populates contactList,
33
34         // when ListView selection changes, displa
35         listView.getSelectionModel().selectedItemP
36         (observableValue, oldValue, newValue) -
37         displayContact(newValue);
38     }
39
40
41
42     // get all the entries from the database to p
43     private void getAllEntries() {
44         contactList.setAll(personQueries.getAllPeo
45         selectFirstEntry();
46     }
47
48     // select first item in listView
49     private void selectFirstEntry() {
50         listView.getSelectionModel().selectFirst()
51     }
52
53     // display contact information
54     private void displayContact(Person person) {
55         if (person != null) {
56             firstNameTextField.setText(person.getFi
57             lastNameTextField.setText(person.getLas
58             emailTextField.setText(person.getEmail(

```

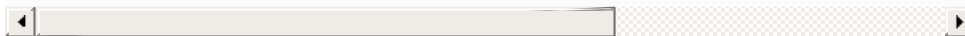


```

59         phoneTextField.setText(person.getPhoneN
60     }
61     else {
62         firstNameTextField.clear();
63         lastNameTextField.clear();
64         emailTextField.clear();
65         phoneTextField.clear();
66     }
67 }
68
69 // add a new entry
70 @FXML
71 void addEntryButtonPressed(ActionEvent event)
72     int result = personQueries.addPerson(
73         firstNameTextField.getText(), lastNameT
74         emailTextField.getText(), phoneTextFiel
75
76     if (result == 1) {
77         displayAlert(AlertType.INFORMATION, "En
78         "New entry successfully added.");
79     }
80     else {
81         displayAlert(AlertType.ERROR, "Entry No
82         "Unable to add entry.");
83     }
84
85     getAllEntries();
86 }
87
88 // find entries with the specified last name
89 @FXML
90 void findButtonPressed(ActionEvent event) {
91     List<Person> people = personQueries.getPeo
92     findByLastNameTextField.getText() + "%"
93
94     if (people.size() > 0) { // display all en
95         contactList.setAll(people);
96         selectFirstEntry();
97     }
98     else {

```

```
99         displayAlert(AlertType.INFORMATION, "La
100         "There are no entries with the speci
101         }
102     }
103
104     // browse all the entries
105     @FXML
106     void browseAllButtonPressed(ActionEvent even
107         getAllEntries();
108     }
109
110     // display an Alert dialog
111     private void displayAlert(
112         AlertType type, String title, String mess
113         Alert alert = new Alert(type);
114         alert.setTitle(title);
115         alert.setContentText(message);
116         alert.showAndWait();
117     }
118 }
```



a) Initial **Address Book** screen showing entries.

The screenshot shows a window titled "Address Book" with a list of entries on the left and a form on the right. The list contains "Brown, Mary" and "Green, Mike". The form displays the details for "Brown, Mary": First name: Mary, Last name: Brown, Email: demo2@deitel.com, and Phone: 555-1234. There is an "Add Entry" button and a search section at the bottom with a "Find by last name:" input field, a "Find" button, and a "Browse All" button.

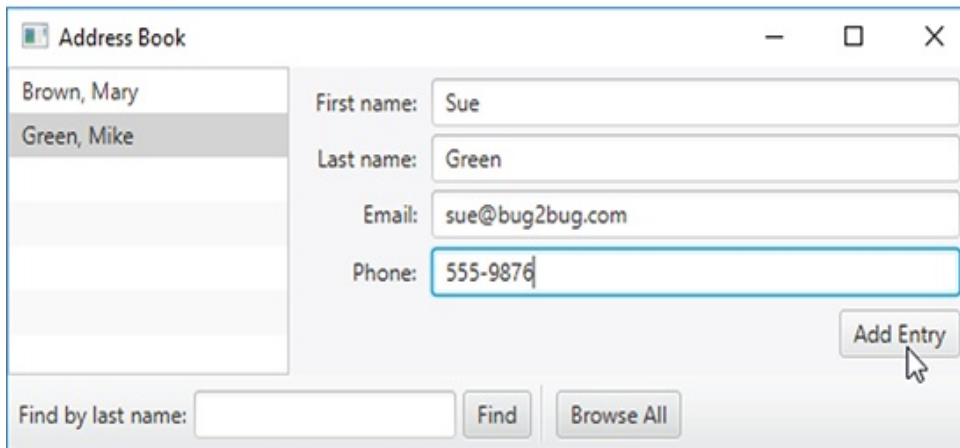
Entry	First name	Last name	Email	Phone
Brown, Mary	Mary	Brown	demo2@deitel.com	555-1234
Green, Mike				

b) Viewing the entry for **Green, Mike**.

The screenshot shows the same "Address Book" window, but now "Green, Mike" is selected in the list. The form displays the details for "Green, Mike": First name: Mike, Last name: Green, Email: demo1@deitel.com, and Phone: 555-5555. The "Add Entry" button and search section remain the same.

Entry	First name	Last name	Email	Phone
Brown, Mary				
Green, Mike	Mike	Green	demo1@deitel.com	555-5555

c) Adding a new entry for Sue Green.

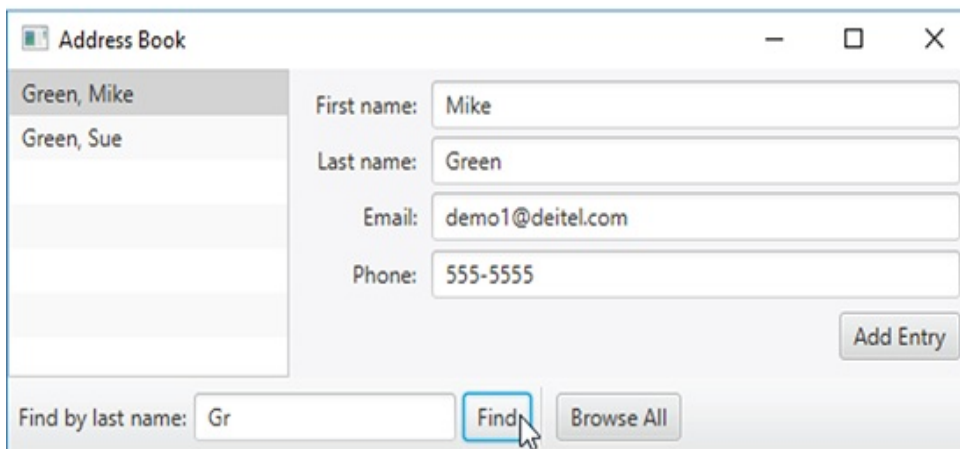


The Address Book window displays a list of contacts on the left: Brown, Mary; Green, Mike; and three empty rows. The right side contains form fields for a new entry: First name (Sue), Last name (Green), Email (sue@bug2bug.com), and Phone (555-9876). The 'Add Entry' button is highlighted with a mouse cursor.

Field	Value
First name	Sue
Last name	Green
Email	sue@bug2bug.com
Phone	555-9876

Find by last name: Find Browse All

d) Searching for last names that start with Gr.

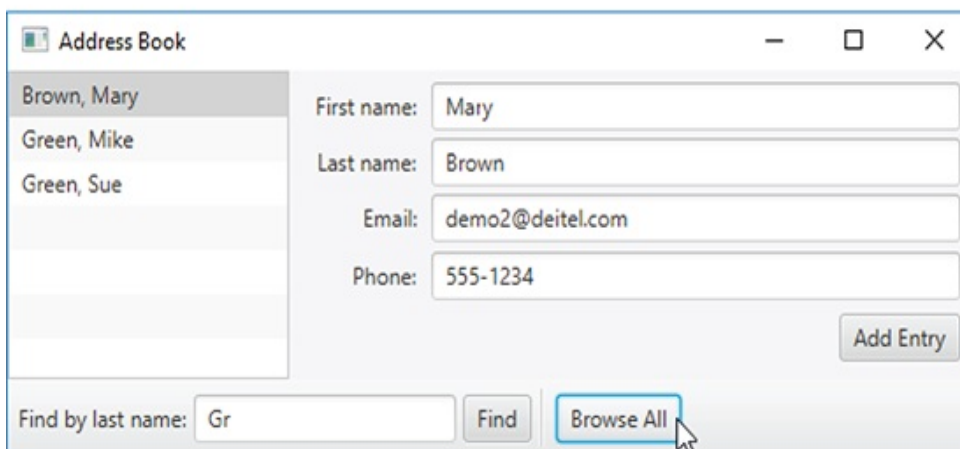


The Address Book window shows search results for last names starting with 'Gr'. The list on the left now includes Green, Mike and Green, Sue. The form fields on the right show details for Mike Green: First name (Mike), Last name (Green), Email (demo1@deitel.com), and Phone (555-5555). The 'Find' button is highlighted with a mouse cursor.

Field	Value
First name	Mike
Last name	Green
Email	demo1@deitel.com
Phone	555-5555

Find by last name: Gr Find Browse All

e) Returning to the complete list by clicking **Browse All**.



The Address Book window shows the complete list of contacts. The list on the left includes Brown, Mary; Green, Mike; and Green, Sue. The form fields on the right show details for Mary Brown: First name (Mary), Last name (Brown), Email (demo2@deitel.com), and Phone (555-1234). The 'Browse All' button is highlighted with a mouse cursor.

Field	Value
First name	Mary
Last name	Brown
Email	demo2@deitel.com
Phone	555-1234

Find by last name: Gr Find Browse All

Fig. 24.34

Controller for the AddressBook app.

Instance Variables

Line 23 creates the `PersonQueries` object. We use the same techniques to populate the `ListView` that we used in [Section 13.5](#), so lines 26–27 create an `ObservableList<Person>` named `contactList` to store the `Person` objects returned by the `PersonQueries` object.

Method `initialize`

When the `FXMLLoader` initializes the controller, method `initialize` (lines 30–40) performs the following tasks:

- Line 31 binds the `contactList` to the `ListView`, so that each time this `ObservableList<Person>` changes, the `ListView` will update its list of items.
- Line 32 calls method `getAllEntries` (declared in lines 43–46) to get all the entries from the database and place them in the `contactList`.
- Lines 35–39 register a `ChangeListener` that displays the selected contact when the user selects a new item in the `ListView`. In this case, we used a lambda expression to create the event handler ([Fig. 13.15](#) showed a similar `ChangeListener` defined as an anonymous inner class).

Methods `getAllEntries` and `selectFirstEntry`

When the app first executes, when the user clicks the **Browse All** Button and when the user adds a new entry to the database, method `getEntries` (lines 43–46) calls `PersonQueries` method `getAllPeople` (line 44) to obtain all the entries. The resulting `List<Person>` is passed to `ObservableList` method `setAll` to replace the `contactList`'s contents. At this point, the `ListView` updates its list of items based on the new contents of `contactList`.

Next line 45 selects the first item in the `ListView` by calling method `select-FirstEntry` (lines 49–51). Line 50 selects the `ListView`'s first item to display that contact's data.

Method `displayContact`

When an item is selected in the `ListView`, the `ChangeListener` registered in method `initialize` calls `displayContact` (lines 54–67) to display the selected `Person`'s data. If the argument is `null`, the method clears the `TextField`'s contents.

Method addEntryButtonPressed

To add a new entry into the database, you can enter the first name, last name, email and phone number (the `AddressID` will *autoincrement*) in the `TextFields` that display contact information, then press the **Add Entry Button**. Method `addEntryButtonPressed` (lines 70–86) calls `PersonQueries` method `addPerson` (lines 72–74) to add the new entry to the database. Line 85 calls `getAllEntries` to obtain the updated database contents and display them in the `ListView`.

Method findButtonPressed

When the user presses the **Find Button**, method `findButtonPressed` (lines 89–102) is called. Lines 91–92 call `PersonQueries` method `getPeopleByLastName` to search the database. Note that line 92 appends a `%` to the text input by the user. This enables the corresponding SQL query, which contains a `LIKE` operator, to locate last names that begin with the characters the user typed in the `findByLastNameTextField`. If there are several such entries, they're all displayed in the `ListView` when the `contactList` is updated (line 95) and the first one is selected (line 96).

Method `browseAllButtonPressed` `d`

When the user presses the **Browse All Button**, method `browseAllButtonPressed` (lines 105–108) simply calls method `getAllEntries` to get all the database entries and display them in the `ListView`.