29

Java Persistence API (JPA)

Objectives

In this chapter you'll:

- Learn the fundamentals of JPA.
- Use classes, interfaces and annotations from the `javax.persistence` package.
- Use the NetBeans IDE's tools to create a Java DB database.
- Use the NetBeans IDE's object-relational-mapping tools to autogenerate JPA entity classes.
- Use autogenerated entity classes to query databases and access data from multiple database tables.
- Use JPA transaction processing capabilities to modify database data.
- Use Java 8 lambdas and streams to manipulate the results of JPA queries.

Outline

29.1 Introduction

Chapter 24 used JDBC to connect to relational databases and Structured Query Language (SQL) to query and manipulate relational databases. Recall that we created `String`s containing the SQL for every query, insert, update and delete operation. We also created our own classes for managing interactions with databases. If you're not already familiar with relational databases, SQL and JDBC, you should read Chapter 24 first as this chapter assumes you're already familiar with the concepts we presented there. This chapter uses the Java EE version of NetBeans 8.2. You can download the current NetBeans versions[1] from

```
https://netbeans.org/downloads/
```

In this chapter, we introduce the Java Persistence API (JPA). One of JPA's key capabilities is mapping Java classes to relational database tables and objects of those classes to rows in the tables. This is known as **object-relational mapping**. You'll use the NetBeans IDE's object-relational mapping tools to select a database and autogenerate classes that use JPA to interact with that database. Your programs can then use those classes to query the database, insert new records, update existing records and delete records. You will not have to create mappings between your Java code and database tables (as you did with JDBC), and you'll be able to perform complex database manipulations directly in Java.

Though you'll manipulate Java DB databases in this chapter, the JPA can be used with any database management system that supports JDBC. At the end of the chapter, we provide links to online JPA resources where you can learn more.

1. As NetBeans and Java EE evolve, the steps in this chapter may change. NetBeans.org provides prior NetBeans versions for download at `http://services.netbeans.org/downloads/dev.php`.

29.2 JPA Technology Overview

When using JPA in this chapter, you'll interact with an existing database via classes that the NetBeans IDE generates from the database's schema. Though we do not do so in this chapter, it's also possible for you to create such classes from scratch and use JPA annotations that enable those classes to create corresponding tables in a database.

29.2.1 Generated Entity Classes

In Section 29.3.5, you'll use the **NetBeans Entity Classes from Database...** option to add to your project classes that represent the database tables. Together, these classes and the corresponding settings are known as a **persistence unit**. The discussion in this section is based on the `books` database that we introduced in Section 24.3.

For the `books` database, the NetBeans IDE's object-relational mapping tools create two classes in the data model—`Authors` and `Titles`. Each class—known as an **entity class**—represents the corresponding table in the database and objects of these classes—known as **entities**—represent the rows in the corresponding tables. These classes contain:

- Instance variables representing the table's columns—These are named with all lowercase letters by default and have Java types that are compatible with their database types. Each instance variable is preceded by JPA annotations with information about the corresponding database column, such as whether the instance variable is the table's primary key, whether the column's value in the table is autogenerated, whether the column's value is optional and the column's name.
- Constructors for initializing objects of the class—The resulting entity objects represent rows in the corresponding table. Programs can use entity objects to manipulate the corresponding data in the database.
- *Set* and *get* methods that enable client code to access each instance variable.
- Overridden methods of class `Object`—`hashCode`, `equals` and `toString`.

29.2.2 Relationships Between Tables in the Entity Classes

We did not mention the `books` database's `AuthorISBN` table. Recall from Section 24.3 that this table links:

- each author in the `Authors` table to that author's books in the `Titles` table, and
- each book in the `Titles` table to the book's authors in the `Authors` table.

This is known as a **join table**, because it's used to join information from multiple other tables. The object-relational mapping tools do *not* create a class for the `AuthorISBN` table. Instead, relationships between tables are taken into account by the generated entity classes:

- The `Authors` class contains the `titlesList` instance variable—a `List` of `Title` objects representing books written by that author.
- The `Titles` class contains the `authorsList` instance variable—a `List` of `Author` objects representing that book's authors.

Like the other instance variables, these `List` variable declarations are preceded by JPA annotations, such as the join table's name, the `Authors` and `AuthorISBN` columns that link

authors to their books, the `Titles` and `AuthorISBN` columns that link titles to their authors, and the type of the relationship. In the book's database there is a *many-to-many relationship*, because each author can write many books and each book can have many authors. We'll show key features of these autogenerated classes later in the chapter. Section 29.4 demonstrates queries that use the relationships among the `books` database's tables to display joined data.

29.2.3 The `javax.persistence` Package

The package **`javax.persistence`** contains the JPA interfaces and classes used to interact with the databases in this chapter.

`EntityManager` Interface

An object that implements the **`EntityManager`** interface manages the interactions between the program and the database. In Sections 29.3–29.4, you'll use an `EntityManager` to create query objects for obtaining entities from the `books` database. In Section 29.5, you'll use an `EntityManager` to both query the `addressbook` database and to create transactions for inserting new entities into the database.

`EntityManagerFactory` Interface and the `Persistence` Class

To obtain an `EntityManager` for a given database, you'll use an object that implements the **`EntityManagerFactory`** interface. As you'll see, the `Persistence` class's `static` method **`createEntityManagerFactory`** returns an `EntityManagerFactory` for the persistence unit you specify as a `String` argument.

In this chapter, you'll use application-managed `EntityManagers`—that is, ones you obtain from an `EntityManagerFactory` in your app. When you use JPA in Java EE apps, you'll obtain container-managed `EntityManagers` from the Java EE server (i.e., the container) on which your app executes.

`TypedQuery` Class, Dynamic Queries and Named Queries

An object that implements the **`TypedQuery`** generic interface performs queries and returns a collection of matching entities—in this chapter, you'll specify that the queries should return `List` objects, though you can choose `Collection`, `List` or `Set` when you generate the entity classes.

To create queries, you'll use `EntityManager` methods. In Section 29.3, you'll create a query with the `EntityManager`'s **`createQuery`** method. This method's first argument is a `String` written in the **Java Persistence Query Language (JPQL)**—as you'll see, JPQL is similar to SQL (Section 24.4). JPQL queries entity objects, rather than relational database tables. When you define a query in your own code, it's known as a *dynamic query*. In Sections 29.4–29.5, you'll use autogenerated *named queries* that you can access via the `EntityManager` method **`createNamedQuery`**.

29.3 Querying a Database with JPA

In this section, we demonstrate how to create the `books` database's JPA entity classes, then use JPA and those classes to *connect* to the `books` database, *query* it and *display* the results of the query. As you'll see, NetBeans provides tools that simplify accessing data via JPA.

This section's example performs a simple query that retrieves the `books` database's `Authors` table. We then use lambdas and streams to display the table's contents. The steps you'll perform are:

- Create a Java DB database and populate it from the `books.sql` file provided with this chapter's examples.
- Create the Java project.
- Add the JPA reference implementation's libraries to the project.
- Add the Java DB library to the project so that the app can access the driver required to connect to the Java DB database over a network—though we'll use the network-capable version of Java DB here, the database will still reside on your local computer.
- Create the persistence unit containing the entity classes for querying the database.
- Create the Java app that uses JPA to obtain the `Authors` table's data.

29.3.1 Creating the Java DB Database

In this section, you'll use the SQL script (`books.sql`) provided with this chapter's examples to create the `books` database in NetBeans. Chapter 24 demonstrated several database apps that used the embedded version of Java DB. This chapter's examples use the network server version.

Creating the Database

Perform the following steps to create the `books` database:

1. In the upper-left corner of the NetBeans IDE, click the **Services** tab. (If the Services tab is not displayed, select **Services** from the **Window** menu.)
2. Expand the **Databases** node then right click **Java DB**. If **Java DB** is not already running the **Start Server** option will be enabled. In this case, Select **Start Server** to launch the Java DB server. You may need to wait a moment for the server to begin executing.[2]
3. Right click the **Java DB** node, then select **Create Database...**.
4. In the **Create Java DB Database** dialog, set **Database Name** to `books`, **User Name** to `deitel`, and **Password** and **Confirm Password** to `deitel`.[3]
5. Click **OK**.

The preceding steps create the database using Java DB's *server version* that can receive database connections over a network. A new node named

```
jdbc:derby://localhost:1527/books
```

2. If the **Start Server** option is disabled, select **Properties...** and ensure that the **Java DB Installation** option is set to the JDK's **db** folder location.
3. We used `deitel` as the user name and password for simplicity—ensure that you use secure passwords in real applications.

appears in the **Services** tab's **Database** node. This is the JDBC URL that's used to connect to the database.

29.3.2 Populating the books Database with Sample Data

You'll now populate the database with sample data using the `books.sql` script that's provided with this chapter's examples. To do so, perform the following steps:

1. Select **File > Open File...** to display the **Open** dialog.
2. Navigate to this chapter's examples folder, select `books.sql` and click **Open**.
3. In NetBeans, right click in the SQL script and select **Run File**.
4. In the **Select Database Connection** dialog, select the JDBC URL for the database you created in Section 29.3.1 and click **OK**.

The IDE will connect to the database and run the SQL script to populate the database. The SQL script attempts to remove the database's tables if they already exist. If they do not, you'll receive error messages when the three `DROP TABLE` commands in the SQL script execute, but the tables will still be created properly.

You can confirm that the database was populated properly by viewing each table's data in NetBeans. To do so:

1. In the NetBeans **Services** tab, expand the **Databases** node, then expand the node `jdbc:derby://localhost:1527/books`.
2. Expand the **DEITEL** node, then the **Tables** node.
3. Right click one of the tables and select **View Data...**.

The `books` database is now set up and ready for connections.

29.3.3 Creating the Java Project

For the examples in this section and Section 29.4, we'll create one project that contains the `books` database's JPA entity classes and two Java apps that use them. To create the project:

1. In the upper-left corner of NetBeans, select the **Projects** tab.
2. Select **File > New Project...**.
3. In the **New Project** dialog, select the **Java** category, then **Java Application** and click **Next >**.
4. For the **Project Name**, specify `BooksDatabaseExamples`, then choose where you wish to store the project on your computer.
5. Ensure that the **Create Main Class** option is checked. By default, NetBeans uses the project name as the class name and puts the class in a package named `booksdatabaseexamples` (the project name in all lowercase letters). We changed the class name for this first example to `DisplayAuthors`. Also, to indicate that the classes in this package are from this book's JPA chapter, we replaced the package name with

```
com.deitel.jhtp.jpa
```

6. Click **Finish** to create the project.

29.3.4 Adding the JPA and Java DB Libraries

For certain types of projects (such as server-side Java EE applications), NetBeans automatically includes JPA support, but not for simple **Java Application** projects. In addition, NetBeans projects do not include database drivers by default. In this section, you'll add the JPA libraries and Java DB driver library to the project so that you can use JPA's features to interact with the Java DB database you created in Sections 29.3.1–29.3.2.

EclipseLink—The JPA Reference Implementation

Each Java Enterprise Edition (Java EE) API—such as JPA—has a *reference implementation* that you can use to experiment with the API's features and implement applications. The JPA reference implementation—which is included with the NetBeans Java EE version—is **EclipseLink** (`http://www.eclipse.org/eclipselink`).

Adding Libraries

To add JPA and Java DB support to your project:

1. In the NetBeans **Projects** tab, expand the **BooksDatabaseExamples** node.
2. Right click the project's **Libraries** node and select **Add Library...**.
3. In the **Add Library** dialog, hold the *Ctrl* key—*command* (⌘) in OS X—and select **EclipseLink (JPA 2.1)**, **Java DB Driver** and **Persistence (JPA 2.1)**, then click **Add Library**.

29.3.5 Creating the Persistence Unit for the books Database

In this section, you'll create the persistence unit containing the entity classes `Authors` and `Titles` using the NetBeans object-relational mapping tools. To do so:

1. In the NetBeans **Projects** tab, right click the **BooksDatabaseExamples** node, then select **New > Entity Classes from Database...**.
2. In the **New Entity Classes from Database** dialog's **Database Tables** step, select the books database's URL from the **Database Connection** drop-down list. Then, click the **Add All >>** button and click **Next >**.
3. The **Entity Classes** step enables you to customize the entity class names and the package. Keep the default names, ensure that **Generate Named Query Annotations for Persistent Fields**, **Generate JAXB Annotations** and **Create Persistence Unit** are checked then click **Next >**.
4. In the **Mapping Options** step, change the **Collection Type** to `java.util.List` and keep the other default settings —for queries that return multiple authors or titles, the results will be placed in `List` objects.
5. Click **Finish**.

The IDE creates the persistence unit containing the `Authors` and `Titles` classes and adds their source-code files `Authors.java` and `Titles.java` to the project's package node (`com.deitel.jhtp.jpa`) in the **Source Packages** folder. As part of the persistence unit, the IDE also creates a `META-INF` package in the **Source Packages** folder. This contains the `persistence.xml` file, which specifies persistence unit settings. These include the `books` database's JDBC URL and the persistence unit's name, which you'll use to obtain an

`EntityManager` to manage the `books` database interactions. By default, the persistence unit's name is the project name followed by `PU`—`BooksDatabaseExamplesPU`. It's also possible to have multiple persistence units, but that's beyond this chapter's scope.

29.3.6 Querying the Authors Table

Figure 29.1 performs a simple `books` database query that retrieves the `Authors` table and displays its data. The program illustrates using JPA to connect to the database and query it. You'll use a dynamic query created in `main` to get the data from the database—in the next example, you'll use auto-generated queries in the persistence unit to perform the same query and others. In Section 29.5, you'll learn how to modify a database through a JPA persistence unit. *Reminder:* Before you run this example, ensure that the Java DB database server is running; otherwise, you'll get runtime exceptions indicating that the app cannot connect to the database server. For details on starting the Java DB server, see Section 29.3.1.

```
1  // Fig. 29.1: DisplayAuthors.java
2  // Displaying the contents of the authors table.
3  package com.deitel.jhtp.jpa;
4
5  import javax.persistence.EntityManager;
6  import javax.persistence.EntityManagerFactory;
7  import javax.persistence.Persistence;
8  import javax.persistence.TypedQuery;
9
10 public class DisplayAuthors
11 {
12    public static void main(String[] args)
13    {
14       // create an EntityManagerFactory for the persistence unit
15       EntityManagerFactory entityManagerFactory =
16          Persistence.createEntityManagerFactory(
17             "BooksDatabaseExamplesPU");
18
19       // create an EntityManager for interacting with the persistence unit
20       EntityManager entityManager =
21          entityManagerFactory.createEntityManager();
22
23       // create a dynamic TypedQuery<Authors> that selects all authors
24       TypedQuery<Authors> findAllAuthors = entityManager.createQuery(
25          "SELECT author FROM Authors AS author", Authors.class);
26
27       // display List of Authors
28       System.out.printf("Authors Table of Books Database:%n%n");
29       System.out.printf("%-12s%-13s%s%n",
30          "Author ID", "First Name", "Last Name");
31
32       // get all authors, create a stream and display each author
33       findAllAuthors.getResultList().stream()
34          .forEach((author) ->
35             {
```

Fig. 29.1 | Displaying contents of the `authors` table. (Part 1 of 2.)

```
36                  System.out.printf("%-12d%-13s%s%n", author.getAuthorid(),
37                     author.getFirstname(), author.getLastname());
38               }
39            );
40      }
41   }
```

```
Authors Table of Books Database:

Author ID   First Name   Last Name
1           Paul         Deitel
2           Harvey       Deitel
3           Abbey        Deitel
4           Dan          Quirk
5           Michael      Morgano
```

Fig. 29.1 | Displaying contents of the `authors` table. (Part 2 of 2.)

Importing the JPA Interfaces and Class Used in This Example

Lines 5–8 import the JPA interfaces and class from package `javax.persistence` used in this program:

- `EntityManager` interface—An object of this type manages the data flow between the program and the database.
- `EntityManagerFactory` interface—An object of this type creates the persistence unit's `EntityManager`.
- `Persistence` class—A `static` method of this class creates the specified persistence unit's `EntityManagerFactory`.
- `TypedQuery` interface—The `EntityManager` returns an object of this type when you create a query. You then execute the query to get data from the database.

Creating the `EntityManagerFactory` Object

Lines 15–17 create the persistence unit's `EntityManagerFactory` object. The `Persistence` class's `static` method `createEntityManagerFactory` receives the persistence unit's name—`BooksDatabaseExamplesPU`. In Section 29.3.5, NetBeans created this name in `persistence.xml`, based on the project's name.

Creating the `EntityManager`

Lines 20–21 use the `EntityManagerFactory`'s **`createEntityManager`** method to create an application-managed `EntityManager` that handles the interactions between the app and the database. These include querying the database, storing new entities into the database, updating existing entries in the database and removing entities from the database. You'll use the `EntityManager` in this example to create a query.

Creating a `TypedQuery` That Retrieves the `Authors` Table

Lines 24–25 use `EntityManager`'s `createQuery` method to create a `TypedQuery` that returns all of the `Authors` entities in the `Authors` table—each `Authors` entity represents one row in the table. The first argument to `createQuery` is a `String` written in the Java Per-

sistence Query Language (JPQL). The second argument specifies a `Class` object representing the type of objects the query returns—`Authors.class` is shorthand notation for a creating a `Class` object representing `Authors`. Recall that when creating the entity classes, we specified that query results should be returned as `Lists`. When this query executes, it returns a `List<Authors>` that you can then use in your code to manipulate the `Authors` table. You can learn more about JPQL in the Java EE 7 tutorial at:

```
https://docs.oracle.com/javaee/7/tutorial/persistence-
   querylanguage.htm
```

Displaying the Query Results

Lines 33–39 execute the query and use lambdas and streams to display each `Authors` object. To perform the query created in lines 24–25, line 33 calls its **`getResultsList`** method, which returns a `List<Authors>`. Next, we create a `Stream` from that `List` and invoke the `Stream`'s `forEach` method to display each `Authors` object in the `List`. The lambda expression passed to `forEach` uses the `Authors` class's autogenerated *get* methods to obtain the author ID, first name and last name from each `Authors` object.

29.3.7 JPA Features of Autogenerated Class `Authors`

In this section, we overview various JPA annotations that were inserted into the autogenerated entity class `Authors`. Class `Titles` contains similar annotations. You can see the complete list of JPA annotations and their full descriptions at:

```
http://docs.oracle.com/javaee/7/api/index.html?javax/persistence/
   package-summary.html
```

JPA Annotations for Class **`Authors`**

If you look through the source code for autogenerated class `Authors` (or class `Titles`), you'll notice that the class does not contain any code that interacts with a database. Instead, you'll see various JPA annotations that the NetBeans IDE's object-relational-mapping tools autogenerate. When you compile the entity classes, the compiler looks at the annotations and adds JPA capabilities that help manage the interactions with the database—this is known as *injecting* capabilities. For the entity classes, the annotations include:

- **`@Entity`**—Specifies that the class is an entity class.
- **`@Table`**—Specifies the entity class's corresponding database table.
- **`@NamedQueries`**/**`@NamedQuery`**—An `@NamedQueries` annotation specifies a collection of `@NamedQuery` annotations that declare various named queries. You can define your own `@NamedQuery` annotations in addition to the ones that the object-relational-mapping tools can autogenerate.

JPA Annotations for Class Authors' Instance Variables

JPA annotations also specify information about an entity class's instance variables:

- **`@Id`**—Used to indicate the instance variable that corresponds to the database table's primary key. For composite primary keys, multiple instance variables would be annotated with `@Id`.
- **`@GeneratedValue`**—Indicates that the column value in the database is autogenerated.

- `@Basic`—Specifies whether the column is optional and whether the corresponding data should load *lazily* (i.e., only when the data is accessed through the entity object) or *eagerly* (i.e., loaded immediately when the entity object is created).
- `@Column`—Specifies the database column to which the instance variable corresponds.
- `@JoinTable`/`@JoinColumn`—These specify relationships between tables. In the `Authors` class, this helps JPA determine how to populate an `Authors` entity's `titlesList`.
- `@ManyToMany`—Specifies the relationship between entities. For the `Authors` and `Titles` entity classes, there is a many-to-many relationship—each author can write many books and each book can have many authors. There are also annotations for `@ManyToOne`, `@OneToMany` and `@OneToOne` relationships.

29.4 Named Queries; Accessing Data from Multiple Tables

The next example demonstrates two named queries that were autogenerated when you created the `books` database's persistence unit in Section 29.3.5. For discussion purposes we split the program into Figs. 29.2 and 29.3, each showing the corresponding portion of the program's output. Once again, we use lambdas and streams capabilities to display the results. As you'll see, we use the relationships between the `Authors` and `Titles` entities to display information from both database tables.

29.4.1 Using a Named Query to Get the `List` of `Authors`, then Display the `Authors` with Their Titles

Figure 29.2 uses the techniques you learned in Section 29.3 to display each author followed by that author's list of titles. To add `DisplayQueryResults.java` to your project:

1. Right click the project's name in the NetBeans **Projects** tab and select **New > Java Class...**.
2. In the **New Java Class** dialog, enter `DisplayQueryResults` for the **Class Name**, select `com.deitel.jhtp.jpa` as the **Package** and click **Finish**.

The IDE opens the new file and you can now enter the code in Figs. 29.2 and 29.3. To run this file, right click its name in the project, then select **Run File**. You can also right click the project and select **Properties** then set this class as the **Main Class** in the project's **Run** settings. Then, when you run the project, this file's `main` method will execute.

```
1  // Fig. 29.2: DisplayQueryResults.java
2  // Display the results of various queries.
3
4  package com.deitel.jhtp.jpa;
5
```

Fig. 29.2 | Using a `NamedQuery` to get the list of `Authors`, then display the `Authors` with their titles. (Part 1 of 3.)

```
import java.util.Comparator;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.TypedQuery;

public class DisplayQueryResults
{
   public static void main(String[] args)
   {
      // create an EntityManagerFactory for the persistence unit
      EntityManagerFactory entityManagerFactory =
         Persistence.createEntityManagerFactory(
            "BooksDatabaseExamplesPU");

      // create an EntityManager for interacting with the persistence unit
      EntityManager entityManager =
         entityManagerFactory.createEntityManager();

      // TypedQuery that returns all authors
      TypedQuery<Authors> findAllAuthors =
         entityManager.createNamedQuery("Authors.findAll", Authors.class);

      // display titles grouped by author
      System.out.printf("Titles grouped by author:%n");

      // get the List of Authors then display the results
      findAllAuthors.getResultList().stream()
         .sorted(Comparator.comparing(Authors::getLastname)
            .thenComparing(Authors::getFirstname))
         .forEach((author) ->
            {
               System.out.printf("%n%s %s:%n",
                  author.getFirstname(), author.getLastname());

               for (Titles title : author.getTitlesList())
               {
                  System.out.printf("\t%s%n", title.getTitle());
               }
            }
         );

```

```
Titles grouped by author:

Abbey Deitel:
         Internet & World Wide Web How to Program
         Simply Visual Basic 2010
         Visual Basic 2012 How to Program
         Android How to Program
         Android for Programmers: An App-Driven Approach, 2/e, Volume 1
         Android for Programmers: An App-Driven Approach
```

Fig. 29.2 | Using a `NamedQuery` to get the list of `Authors`, then display the `Authors` with their titles. (Part 2 of 3.)

```
Harvey Deitel:
         Internet & World Wide Web How to Program
         Java How to Program
         Java How to Program, Late Objects Version
         C How to Program
         Simply Visual Basic 2010
         Visual Basic 2012 How to Program
         Visual C# 2012 How to Program
         Visual C++ How to Program
         C++ How to Program
         Android How to Program
         Android for Programmers: An App-Driven Approach, 2/e, Volume 1
         Android for Programmers: An App-Driven Approach

Paul Deitel:
         Internet & World Wide Web How to Program
         Java How to Program
         Java How to Program, Late Objects Version
         C How to Program
         Simply Visual Basic 2010
         Visual Basic 2012 How to Program
         Visual C# 2012 How to Program
         Visual C++ How to Program
         C++ How to Program
         Android How to Program
         Android for Programmers: An App-Driven Approach, 2/e, Volume 1
         Android for Programmers: An App-Driven Approach

Michael Morgano:
         Android for Programmers: An App-Driven Approach

Dan Quirk:
         Visual C++ How to Program
```

Fig. 29.2 | Using a `NamedQuery` to get the list of `Authors`, then display the `Authors` with their titles. (Part 3 of 3.)

Creating a `TypedQuery` *That Retrieves the* `Authors` *Table*

One of the default options when you created the books database's persistence unit was **Generate Named Query Annotations for Persistent Fields**—you can view these named queries before the class definitions in `Authors.java` and `Titles.java`. For class `Authors`, the object-relational mapping tool autogenerated the following queries:

- `"Authors.findAll"`—Returns the `List` of all `Authors` entities.
- `"Authors.findByAuthorid"`—Returns the `Authors` entity with the specified `authorid` value.
- `"Authors.findByFirstname"`—Returns the `List` of all `Authors` entities with the specified `firstname` value.
- `"Authors.findByLastname"`—Returns the `List` of all `Authors` entities with the specified `lastname` value.

You'll see how to provide arguments to queries in Section 29.5. Like the dynamic query you defined in Fig. 29.1, each of these queries is defined using the Java Persistence Query Language (JPQL).

Lines 17–23 get the `EntityManager` for this program, just as we did in Fig. 29.1. Lines 26–27 use `EntityManager`'s `createNamedQuery` method to create a `TypedQuery` that returns the result of the `"Authors.findAll"` query. The first argument is a `String` containing the query's name and the second is the `Class` object representing the entity type that the query returns.

Processing the Results

Lines 33–46 execute the query and use Java 8 lambdas and streams to display each `Authors` entity's name followed by the list of that author's titles. Line 33 calls the `TypedQuery`'s `getResultsList` method to perform the query. We create a `Stream` that sorts the `Authors` entities by last name then first name. Next, we invoke the `Stream`'s `forEach` method to display each `Authors` entity's name and list of titles. The lambda expression passed to `forEach` uses the `Authors` class's autogenerated *get* methods to obtain the first name and last name from each `Authors` entity. Line 41 calls the autogenerated `Authors` method `getTitlesList` to get the current author's `List<Titles>`, then lines 41–44 display the `String` returned by each `Titles` entity's autogenerated `getTitle` method.

29.4.2 Using a Named Query to Get the `List` of `Titles`, then Display Each with Its `Authors`

In Fig. 29.3, lines 49–50 use `EntityManager` method `createNamedQuery` to create a `TypedQuery` that returns the result of the `"Titles.findAll"` query. Then, lines 56–68 display each title followed by that title's list of author names. Line 56 calls the `TypedQuery`'s `getResultsList` method to perform the query. We create a `Stream` that sorts the `Titles` entities by title. Next, we invoke the `Stream`'s `forEach` method to display each `Titles` entity's title and the corresponding list of authors. Once again, the lambda expression uses the autogenerated `Titles` and `Authors` methods to access the entity data that's displayed.

```
48         // TypedQuery that returns all titles
49         TypedQuery<Titles> findAllTitles =
50            entityManager.createNamedQuery("Titles.findAll", Titles.class);
51
52         // display titles grouped by author
53         System.out.printf("%nAuthors grouped by title:%n%n");
54
55         // get the List of Titles then display the results
56         findAllTitles.getResultList().stream()
57            .sorted(Comparator.comparing(Titles::getTitle))
58            .forEach((title) ->
59               {
60                  System.out.println(title.getTitle());
61
```

Fig. 29.3 | Using a `NamedQuery` to get the list of `Titles`, then display each with its `Authors`. (Part 1 of 3.)

```
62                  for (Authors author : title.getAuthorsList())
63                  {
64                     System.out.printf("\t%s %s%n",
65                        author.getFirstname(), author.getLastname());
66                  }
67               }
68            );
69      }
70   }
```

```
Authors grouped by title:

Android How to Program
         Paul Deitel
         Harvey Deitel
         Abbey Deitel
Android for Programmers: An App-Driven Approach
         Paul Deitel
         Harvey Deitel
         Abbey Deitel
         Michael Morgano
Android for Programmers: An App-Driven Approach, 2/e, Volume 1
         Paul Deitel
         Harvey Deitel
         Abbey Deitel
C How to Program
         Paul Deitel
         Harvey Deitel
C++ How to Program
         Paul Deitel
         Harvey Deitel
Internet & World Wide Web How to Program
         Paul Deitel
         Harvey Deitel
         Abbey Deitel
Java How to Program
         Paul Deitel
         Harvey Deitel
Java How to Program, Late Objects Version
         Paul Deitel
         Harvey Deitel
Simply Visual Basic 2010
         Paul Deitel
         Harvey Deitel
         Abbey Deitel
Visual Basic 2012 How to Program
         Paul Deitel
         Harvey Deitel
         Abbey Deitel
Visual C# 2012 How to Program
         Paul Deitel
         Harvey Deitel
```

Fig. 29.3 | Using a `NamedQuery` to get the list of `Titles`, then display each with its `Authors`. (Part 2 of 3.)

```
Visual C++ How to Program
         Paul Deitel
         Harvey Deitel
         Dan Quirk
```

Fig. 29.3 | Using a `NamedQuery` to get the list of `Titles`, then display each with its `Authors`. (Part 3 of 3.)

29.5 Address Book: Using JPA and Transactions to Modify a Database

We now reimplement the address book app from Section 24.9 using JPA. As before, you can browse existing entries, add new entries and search for entries with a specific last name. Recall that the `AddressBook` Java DB database contains an `Addresses` table with the columns `addressID`, `FirstName`, `LastName`, `Email` and `PhoneNumber`. The column `addressID` is an identity column in the `Addresses` table.

29.5.1 Transaction Processing

Many database applications require guarantees that a series of database insertions, updates and deletions executes properly before the application continues processing the next database operation. For example, when you transfer money electronically between bank accounts, several factors determine whether the transaction is successful. You begin by specifying the source account and the amount you wish to transfer to a destination account. Next, you specify the destination account. The bank checks the source account to determine whether its funds are sufficient to complete the transfer. If so, the bank withdraws the specified amount and, if all goes well, deposits it into the destination account to complete the transfer. What happens if the transfer fails after the bank withdraws the money from the source account? In a proper banking system, the bank redeposits the money in the source account. How would you feel if the money was subtracted from your source account and the bank *did not* deposit the money in the destination account?

Transaction processing enables a program that interacts with a database to treat a set of operations as a *single* operation, known as an **atomic operation** or a **transaction**. At the end of a transaction, a decision can be made either to **commit the transaction** or **roll back the transaction**:

- Committing the transaction finalizes the database operation(s); all insertions, updates and deletions performed as part of the transaction cannot be reversed without performing a new database operation.
- Rolling back the transaction leaves the database in its state prior to the database operation. This is useful when a portion of a transaction fails to complete properly. In our bank-account-transfer discussion, the transaction would be rolled back if the deposit could not be made into the destination account.

JPA provides transaction processing via methods of interfaces `EntityManager` and `EntityTransaction`. `EntityManager` method `getTransaction` returns an `EntityTransaction` for managing a transaction. `EntityTransaction` method `begin` starts a transaction. Next, you perform your database's operations using the `EntityManager`. If the

operations execute successfully, you call `EntityTransaction` method **`commit`** to commit the changes to the database. If any operation fails, you call `EntityTransaction` method **`rollback`** to return the database to its state prior to the transaction. You'll use these techniques in Section 29.5.4. (In a Java EE project, the server can perform these tasks for you.)

29.5.2 Creating the `AddressBook` Database, Project and Persistence Unit

Use the techniques you learned in Sections 29.3.1–29.3.5 to perform the following steps:

Step 1: Creating the ***`addressbook`*** *Database*
Using the steps presented in Section 29.3.1, create the `addressbook` database.

Step 2: Populating the Database
Using the steps presented in Section 29.3.2, populate the `addressbook` database with the sample data in the `addressbook.sql` file that's provided with this chapter's examples.

Step 3: Creating the ***`AddressBook`*** *Project*
This app has a JavaFX GUI. For prior JavaFX apps, we created an FXML file that described the app's GUI, a subclass of `Application` that launched the app and a controller class that handled the app's GUI events and provided other app logic. NetBeans provides a **JavaFX FXML Application** project template that creates the FXML file and Java source-code files for the `Application` subclass and controller class. To use this template:

1. Select **File > New Project...** to open the **New Project** dialog.
2. Under **Categories:** select **JavaFX** and under **Projects:** select **JavaFX FXML Application**, then click **Next >**.
3. For the **Project Name** specify `AddressBook`.
4. For the FXML name, specify `AddressBook`.
5. In the **Create Application Class** textfield, replace the default package name and class name with `com.deitel.jhtp.jpa.AddressBook`.
6. Click `Finish` to create the project.

NetBeans places in the app's package the files `AddressBook.fxml`, `AddressBook.java` and `AddressBookController.java`. If you double-click the FXML file in NetBeans, it will automatically open in Scene Builder (if you have it installed) so that you can design your GUI.

For this app, rather than recreating `AddressBook` GUI, we replaced the default FXML that NetBeans generated in `AddressBook.fxml` with the contents of `AddressBook.fxml` from Section 24.9's example (right click the FXML file in NetBeans and select **Edit** to view its source code). We then changed the controller class's name from `AddressBookController` to

```
com.deitel.jhtp.jpa.AddressBookController
```

because the controller class in this example is in the package `com.deitel.jhtp.jpa`.

Also, in the autogenerated `AddressBook` subclass of `Application` (located in `AddressBook.java`), we added the following statement to set the `stage`'s title bar `String`:

```
stage.setTitle("Address Book");
```

Step 4: Adding the JPA and Java DB Libraries

Using the steps presented in Section 29.3.4, add the required JPA and JavaDB libraries to the project's **Libraries** folder.

Step 5: Creating the `AddressBook` Database's Persistence Unit

Using the steps presented in Section 29.3.5, create the `AddressBook` database's persistence unit, which will be named `AddressBookPU` by default.

29.5.3 Addresses Entity Class

When you created the `AddressBook` database's persistence unit, NetBeans autogenerated the `Addresses` entity class (in `Addresses.java`) with several named queries. In this app, you'll use the queries:

- `"Addresses.findAll"`—Returns a `List<Addresses>` containing `Addresses` entities for all the contacts.
- `"Addresses.findByLastname"`—Returns a `List<Addresses>` containing an `Addresses` entity for each contact with the specified last name.

Ordering the Named Query Results

By default, the JPQL for the autogenerated named queries does not order the query results. In Section 24.9, we used the SQL's `ORDER BY` clause to arrange query results into ascending order by last name then first name. JPQL also has an `ORDER BY` clause. To order the query results in this app, we opened `Addresses.java` and added

```
ORDER BY a.lastname, a.firstname
```

to the query strings for the `"Addresses.findAll"` and `"Addresses.findByLastname"` named queries—again these are specified in the `@NamedQuery` annotations just before the `Addresses` class's declaration.

`toString` Method of Class `Addresses`

In this app, we use the `List<Addresses>` returned by each query to populate an `ObservableList` that's bound to the app's `ListView`. Recall that, by default, a `ListView`'s cells display the `String` representation of the `ObservableList`'s elements. To ensure that each `Addresses` object in the `ListView` is displayed in the format *Last Name, First Name*, we modified the `Addresses` class's autogenerated `toString` method. To do so, open `Addresses.java` and replace its `return` statement with

```
return getLastname() + ", " + getFirstname();
```

29.5.4 `AddressBookController` Class

The `AddressBookController` class (Fig. 29.4) uses the persistence unit you created in Section 29.5.2 to interact with `addressbook` database. Much of the code in Fig. 29.4 is identical to the code in Fig. 24.34. For the discussion in this section, we focus on the highlighted JPA features.

```
1  // Fig. 29.4: AddressBookController.java
2  // Controller for a simple address book
3  package com.deitel.jhtp.jpa;
4
5  import java.util.List;
6  import javafx.collections.FXCollections;
7  import javafx.collections.ObservableList;
8  import javafx.event.ActionEvent;
9  import javafx.fxml.FXML;
10 import javafx.scene.control.Alert;
11 import javafx.scene.control.Alert.AlertType;
12 import javafx.scene.control.ListView;
13 import javafx.scene.control.TextField;
14 import javax.persistence.EntityManager;
15 import javax.persistence.EntityManagerFactory;
16 import javax.persistence.EntityTransaction;
17 import javax.persistence.Persistence;
18 import javax.persistence.TypedQuery;
19
20 public class AddressBookController {
21    @FXML private ListView<Addresses> listView;
22    @FXML private TextField firstNameTextField;
23    @FXML private TextField lastNameTextField;
24    @FXML private TextField emailTextField;
25    @FXML private TextField phoneTextField;
26    @FXML private TextField findByLastNameTextField;
27
28    // create an EntityManagerFactory for the persistence unit
29    private final EntityManagerFactory entityManagerFactory =
30       Persistence.createEntityManagerFactory("AddressBookPU");
31
32    // create an EntityManager for interacting with the persistence unit
33    private final EntityManager entityManager =
34       entityManagerFactory.createEntityManager();
35
36    // stores list of Addresses objects that results from a database query
37    private final ObservableList<Addresses> contactList =
38       FXCollections.observableArrayList();
39
40    // populate listView and set up listener for selection events
41    public void initialize() {
42       listView.setItems(contactList); // bind to contactsList
43
44       // when ListView selection changes, display selected person's data
45       listView.getSelectionModel().selectedItemProperty().addListener(
46          (observableValue, oldValue, newValue) -> {
47             displayContact(newValue);
48          }
49       );
50       getAllEntries(); // populates contactList, which updates listView
51    }
52
```

Fig. 29.4 | A simple address book. (Part 1 of 5.)

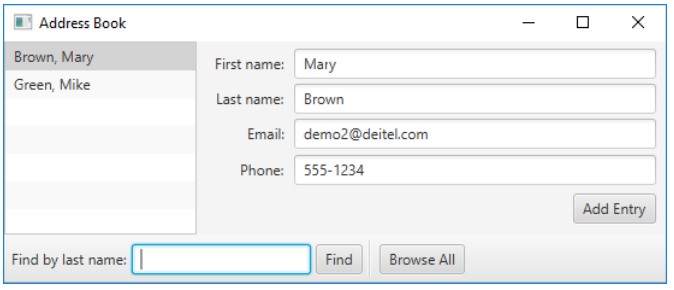
```
53      // get all the entries from the database to populate contactList
54      private void getAllEntries() {
55         // query that returns all contacts
56         TypedQuery<Addresses> findAllAddresses =
57            entityManager.createNamedQuery(
58               "Addresses.findAll", Addresses.class);
59
60         contactList.setAll(findAllAddresses.getResultList());
61         selectFirstEntry();
62      }
63
64      // select first item in listView
65      private void selectFirstEntry() {
66         listView.getSelectionModel().selectFirst();
67      }
68
69      // display contact information
70      private void displayContact(Addresses contact) {
71         if (contact != null) {
72            firstNameTextField.setText(contact.getFirstname());
73            lastNameTextField.setText(contact.getLastname());
74            emailTextField.setText(contact.getEmail());
75            phoneTextField.setText(contact.getPhonenumber());
76         }
77         else {
78            firstNameTextField.clear();
79            lastNameTextField.clear();
80            emailTextField.clear();
81            phoneTextField.clear();
82         }
83      }
84
85      // add a new entry
86      @FXML
87      void addEntryButtonPressed(ActionEvent event) {
88         Addresses address = new Addresses();
89         address.setFirstname(firstNameTextField.getText());
90         address.setLastname(lastNameTextField.getText());
91         address.setPhonenumber(phoneTextField.getText());
92         address.setEmail(emailTextField.getText());
93
94         // get an EntityTransaction to manage insert operation
95         EntityTransaction transaction = entityManager.getTransaction();
96
97         try
98         {
99            transaction.begin(); // start transaction
100           entityManager.persist(address); // store new entry
101           transaction.commit(); // commit changes to the database
102           displayAlert(AlertType.INFORMATION, "Entry Added",
103              "New entry successfully added.");
104        }
```

Fig. 29.4 | A simple address book. (Part 2 of 5.)

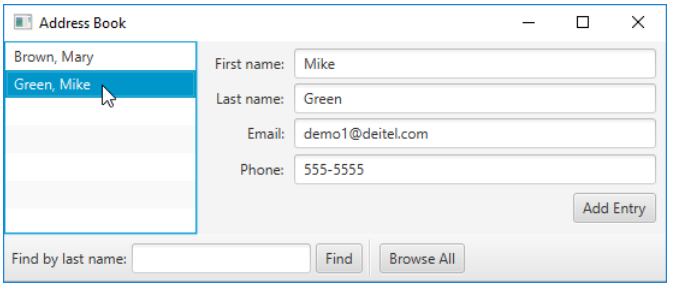
```
105         catch (Exception e) // if transaction failed
106         {
107            transaction.rollback(); // undo database operations
108            displayAlert(AlertType.ERROR, "Entry Not Added",
109               "Unable to add entry: " + e);
110         }
111
112         getAllEntries();
113      }
114
115      // find entries with the specified last name
116      @FXML
117      void findButtonPressed(ActionEvent event) {
118         // query that returns all contacts
119         TypedQuery<Addresses> findByLastname =
120            entityManager.createNamedQuery(
121               "Addresses.findByLastname", Addresses.class);
122
123         // configure parameter for query
124         findByLastname.setParameter(
125            "lastname", findByLastNameTextField.getText() + "%");
126
127         // get all addresses
128         List<Addresses> people = findByLastname.getResultList();
129
130         if (people.size() > 0) { // display all entries
131            contactList.setAll(people);
132            selectFirstEntry();
133         }
134         else {
135            displayAlert(AlertType.INFORMATION, "Lastname Not Found",
136               "There are no entries with the specified last name.");
137         }
138      }
139
140      // browse all the entries
141      @FXML
142      void browseAllButtonPressed(ActionEvent event) {
143         getAllEntries();
144      }
145
146      // display an Alert dialog
147      private void displayAlert(
148         AlertType type, String title, String message) {
149         Alert alert = new Alert(type);
150         alert.setTitle(title);
151         alert.setContentText(message);
152         alert.showAndWait();
153      }
154   }
```

Fig. 29.4 | A simple address book. (Part 3 of 5.)

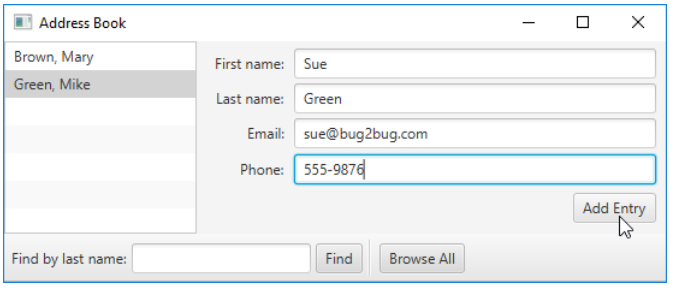
a) Initial **Address Book** screen showing entries.



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



c) Adding a new entry for `Sue Green`.



d) Searching for last names that start with `Gr`.

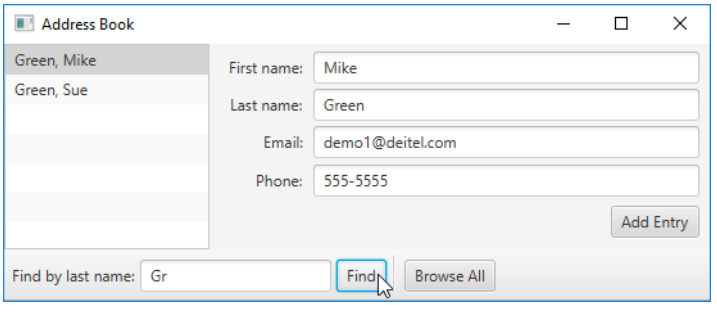


Fig. 29.4 | A simple address book. (Part 4 of 5.)

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

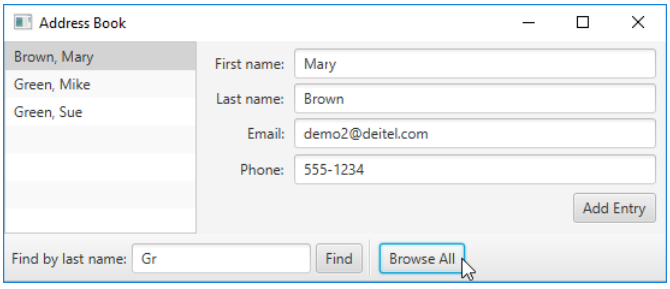


Fig. 29.4 | A simple address book. (Part 5 of 5.)

Obtaining the `EntityManager`

Lines 29–34 use the techniques you learned in Section 29.3.6 to obtain an `EntityManagerFactory` for the `AddressBook` persistence unit (`"AddressBookPU"`), then use it to get the `EntityManager` for interacting with the `addressbook` database. Lines 37–38 define an `ObservableList<Addresses>` named `contactList` that's used to bind the app's query results to the `ListView` (line 42 of method `initialize`).

Obtaining the Complete List of Contacts—Method `getAllEntries`

Lines 56–58 in method `getAllEntries` create a `TypedQuery` for the named query `"Addresses.findAll"`, which returns a `List<Addresses>` containing all the `Addresses` entities in the database. Line 60 calls the `TypedQuery`'s `getResultList` method and uses the resulting `List<Addresses>` to populate the `contactList`, which was previously bound to the `ListView`. Each time the complete contacts list is loaded, line 61 calls method `selectFirstEntry` to display the first `Addresses` entity's details. Due to the listener registered in lines 45–49, this in turn calls method `displayContact` to display the selected `Addresses` entity if there is one; otherwise, `displayContact` clears the `TextFields` that display a contact's details.

Adding an Entry to the Database—Method `addEntryButtonPressed`

When you enter new data in this app's GUI, then click the **Add Entry** button—a new row should be added to the `Addresses` table in the database. To create a new entity in the database, you must first create an instance of the entity class (line 88) and set its instance variables (lines 89–92), then use a transaction to insert the data in the database (lines 95–110). Notice that we do not specify a value for the `Addresses` entity's `addressid` instance variable—this value is autogenerated by the database when you add a new entry.

Lines 95–110 use the techniques discussed in Section 29.5.1 to perform the insert operation. Line 95 uses `EntityManager` method `getTransaction` to get the `EntityTransaction` used to manage the transaction. In the `try` block, line 99 uses `EntityTransaction` method `begin` to start the transaction. Next, line 100 calls `EntityManager` method **`persist`** to insert the new entity into the database. If this operation executes successfully, line 101 calls `EntityTransaction` method `commit` to complete the transaction and commit the changes to the database. If the `persist` operation fails, line 107 in the `catch`

block calls `EntityTransaction` method `rollback` to return the database to its state prior to the transaction.[4]

Finding by Last Name—Method `findButtonPressed`

Lines 119–121 in method `findButtonPressed` create a `TypedQuery` for the named query `"Addresses.findByLastname"`, which returns a `List<Addresses>` containing all the entities with the specified last name. If you open the autogenerated `Addresses` class in your project, you'll see that the query requires a parameter, as specified in the following JPQL that we copied from the `Addresses.java` file:

```
SELECT a FROM Addresses a WHERE a.lastname = :lastname
```

The notation `:lastname` represents a parameter named `lastname`. The autogenerated query locates only exact matches, as indicated by the JPQL equals (=) operator. For this app, we changed = to the JPQL `LIKE` operator so we can locate last names that begin with the letters typed by the user in the `findByLastNameTextField`.

Before executing the query, you set arguments for each query parameter by calling `TypedQuery` method **`setParameter`** (lines 124–125) with the JPQL parameter name as the first argument and the corresponding value as the second argument. As in SQL, line 125 appends `%` to the contents of `findByLastNameTextField` to indicate that we're searching for last names that begin with the user's input, possibly followed by more characters.

When you execute the query (line 128), it returns a `List` containing any matching entities in database. If the number of results is greater than 0, lines 131–132 display the search results in the `ListView` and select the first matching result to display its details. Otherwise, 135–136 display an `Alert` dialog indicating there were no entries with the specified last name.

29.5.5 Other JPA Operations

Though we did not do so in this example, you also can update an existing entity in the database or delete an existing entity from the database.

Updating an Existing Entity

You update an existing entity by modifying its entity object in the context of a transaction. Once you commit the transaction, the changes to the entity are saved to the database.

Deleting an Existing Entity

To remove an entity from the database, call `EntityManager` method **`remove`** in the context of a transaction, passing the entity object to delete as an argument. When you commit the transaction the entity is deleted from the database. This operation will fail if the entity is referenced elsewhere in the database.

29.6 Web Resources

Here are a few key online JPA resources.

4. For simplicity, we performed this example's database operations on the JavaFX application thread. Any potentially long-running database operations should be performed in separate threads using the techniques in Section 23.11.

`https://docs.oracle.com/javaee/7/tutorial/persistence-intro.htm`
The *Introduction to the Java Persistence API* chapter of the *Java EE 7 Tutorial*.

`https://docs.oracle.com/javaee/7/tutorial/persistence-querylanguage.htm`
The *Java Persistence Query Language* chapter of the *Java EE 7 Tutorial*.

`http://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html`
The `javax.persistence` package documentation.

`https://platform.netbeans.org/tutorials/nbm-crud.html`
A NetBeans tutorial for creating a JPA-based app.

29.7 Wrap-Up

In this chapter, we introduced the Java Persistence API (JPA). We used the NetBeans IDE to create and populate a Java DB database, using Java DB's network server version, rather than the embedded version demonstrated in Chapter 24. We created NetBeans projects and added the libraries for JPA and the Java DB driver. Next, we used the NetBeans object-relational mapping tools to autogenerate entity classes from an existing database's schema. We then used those classes to interact with the database.

We queried the databases with both dynamic queries created in code and named queries that were autogenerated by NetBeans. We used the relationships between JPA entities to access data from multiple database tables.

Next, we used JPA transactions to insert new data in a database. We also discussed other JPA operations that you can perform in the context of transations, such as updating existing entities in and deleting entities from a database. Finally, we listed several online JPA resources from which you can learn more about JPA. In the next chapter, we begin our two-chapter object-oriented design and implementation case study.