

Creating a Custom Java Desktop Database Application

This tutorial guides you through the creation of a complete desktop database application that enables its user to browse and edit customer records and purchase history. The resulting application includes the following main features:

- A main view that enables users to browse customer records and customer purchases.
- A search field for the customer records.
- Separate dialog boxes for entering new records or modifying existing records.
- Code to interact with multiple related database tables.

Contents

- [Introduction](#)
- [Setting Up the Database](#)
- [Creating the Application Skeleton](#)
- [Customizing the Master/Detail View](#)
- [Adding Dialog Boxes](#)
- [Activating the Save and Cancel Buttons in the Dialog Boxes](#)
- [Currency Rendering, Date Verifying, and Search](#)
- [See Also](#)



To complete this tutorial, you need the following software and resources.

Software or Resource	Version Required
NetBeans IDE	version 6.5
Java Development Kit (JDK)	version 6
SQL script to create the database tables	
zip file of utility classes	

Note: You can download the [final working project](#) created in this tutorial at any time and open it in the IDE to view the classes. If you want to run the downloaded project, be sure to clean and build it before running.

Introduction

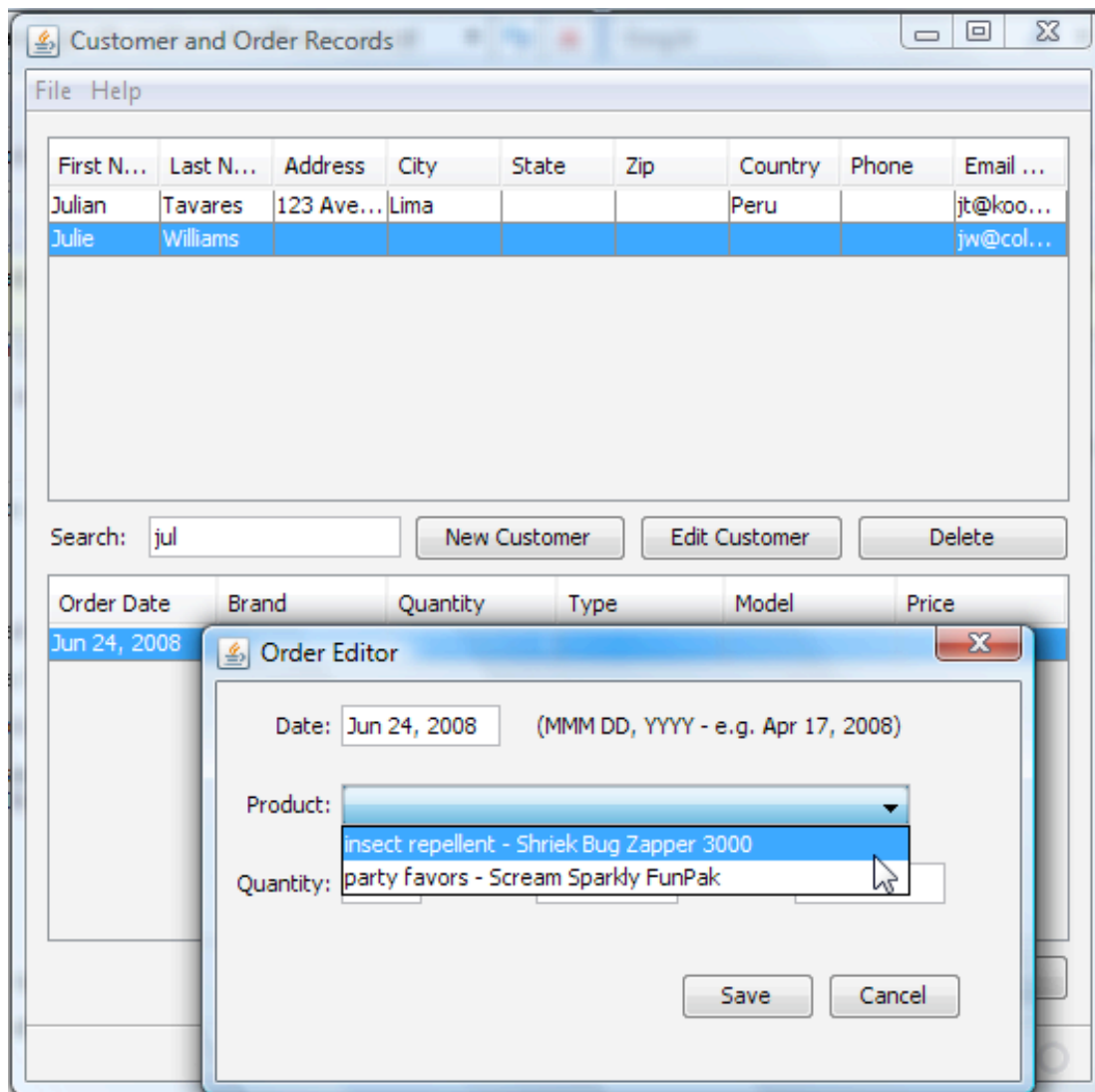
This application takes advantage of the following technologies:

- The Java Persistence API (JPA), which helps you interact with a database using Java code.
- Beans Binding, which enables you to keep Swing component properties synchronized.
- The Swing Application Framework, which simplifies basic application functions such as persisting session information, handling actions, and managing resources.

The tutorial makes use of IDE wizards and other code generation features to provide much of the boilerplate code. It also shows you how to customize the generated code and hand code other parts of the application.

This tutorial takes approximately 2 hours to complete. For a shorter tutorial that shows the creation of a less customized user interface, see [Building a Java Desktop Database Application](#).

Below is a screenshot of the working application that you will have when you complete the tutorial.



Setting Up the Database

Before creating any application code, you need to set up the database. You can then take advantage of wizards that generate much of the application code based on the structure of the database.

The instructions in this tutorial are based using a MySQL database that you create with [this SQL script](#).


Note: You can use other database management software, but doing so might require making some adjustments to the SQL script. In addition, you will need to create the database from outside of the IDE.


To set up the IDE to work with MySQL databases, see the [Connecting to a MySQL Database](#) page.

To create the database:

1. In the Services window, right-click the MySQL Server node and choose Start.
2. Right-click the MySQL Server node and choose Create Database.
If the Create Database item is not enabled, choose Connect. You might then need to enter a password. The Create Database item should then be enabled.
3. For Database Name, type `MyBusinessRecords` and click OK.
A node called MyBusinessRecords should appear in the list of database connections.
4. Right-click the MyBusinessRecords node and choose Connect.
5. If the Connect dialog box appears, type the password that you have set for the database server.
6. If the Advanced tab of the dialog box opens, click OK to close the dialog box.

-->

7. Scroll down to the node for connection that you have just created. The node should have the  icon.

8. Right-click the connection node and choose Execute Command.
9. Copy the contents of the [MyBusinessRecords SQL script](#) and paste them into the SQL Command 1 tab of the Source Editor.
10. Click the Run SQL button () in the toolbar of the Source Editor to run the script. Output of the script should appear in the Output window.
11. Right-click the connection node again and choose refresh.
12. Expand the node, and expand its Tables subnode. You should see four database tables listed.

The database structure was designed with normalization and referential integrity in mind. Here are some notes on the structure:

- The SQL script specifies the [InnoDB storage engine](#) in order to handle the foreign keys in this database. MySQL's default storage engine, MyISAM, will not work with this tutorial.
- The data is split among several tables to reduce duplication and the possibility for inconsistencies. Some tables are connected to each other through foreign keys.
- All of the tables use MySQL's AUTO_INCREMENT attribute so that there is a unique identifier for each row in those tables. This identifier is created by the database management software, so your application and/or your application's user do not have to create this identifier. (So that the AUTO_INCREMENT is used correctly within the application, the IDE adds the `@GeneratedValue(strategy= GenerationType.IDENTITY)` annotation for that column in the table's entity class. This ensures that the application does not try to submit a value for that column when you create a new record.)
- The foreign key in the ORDERS table is there to link each order record with a customer. In the application's user interface, ORDER records are only displayed for the selected CUSTOMER.
- The ON CASCADE DELETE attribute for the foreign key to the CUSTOMERS class ensures that a customer's orders are also deleted when a customer is deleted.
- The foreign key in the CUSTOMERS table points to a COUNTRIES table. You will use this relationship in the application to enable the user to select a customer's country from a combo box.
- The ORDERS table has a foreign key to the PRODUCTS table. When adding a new order record, the user will be able to choose a product from a combo box.
- The COUNTRIES and PRODUCTS tables are pre-populated with data so that you can choose from those tables when the user of the application is adding customer and order records.
- Though this tutorial does not cover it, you might find it useful to create separate applications to populate the COUNTRIES and PRODUCTS tables. Such applications could be created with the Java Desktop Application project template and would not require additional hand-coding.

Creating the Application Skeleton

You will use the IDE's Java Desktop Application project template to create much of the base code for the application. This template generates code for the following features:

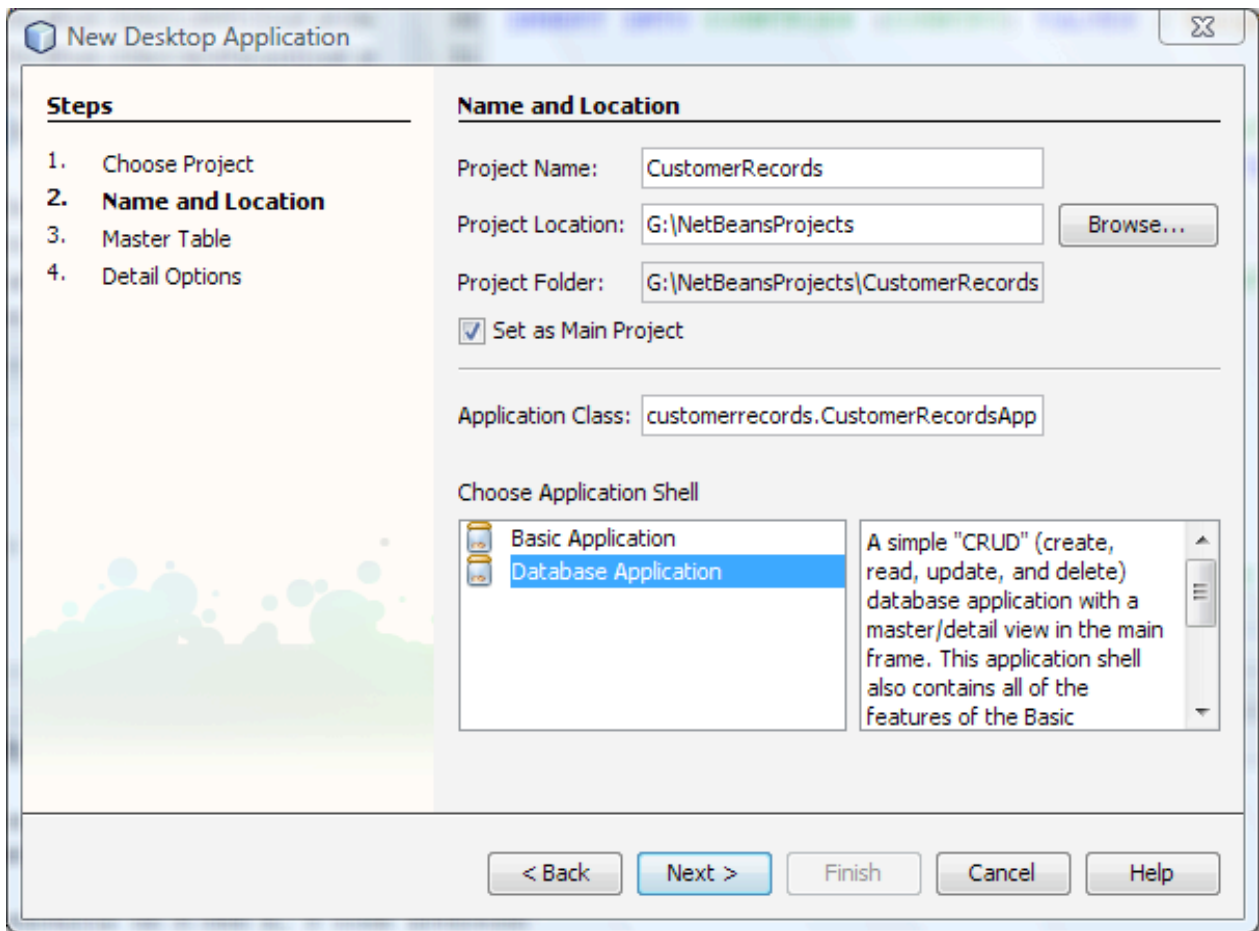
- Connection to the database.
- A main application frame that contains tables for customer details and customer orders.
- A main application class that handles basic application life-cycle functions, including persisting of window state between sessions and resource injection.
- Actions (and corresponding buttons) for standard database application commands.

For some database structures, you can get a working application as soon as you exit the wizard. However, in this tutorial you will use some constructs for which you need to customize the generated code, such as AUTO-INCREMENT and one-to-many relationships.

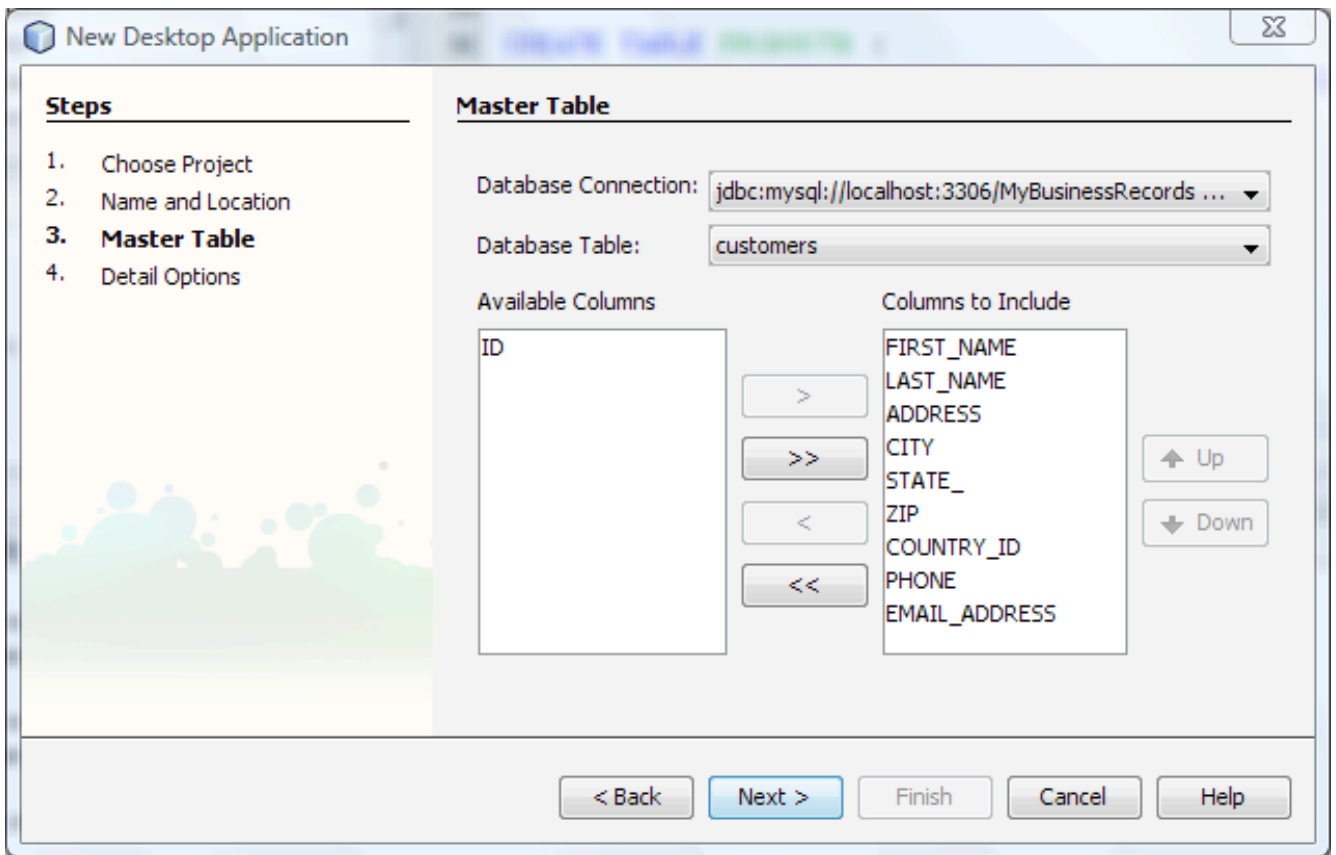
To create the application skeleton:

1. Choose File > New Project.
2. Select the Java category and the Java Desktop Application template. Then click Next.
3. In the Name and Location page of the wizard, follow these steps:
 1. Type `customerRecords` in the Project Name field.

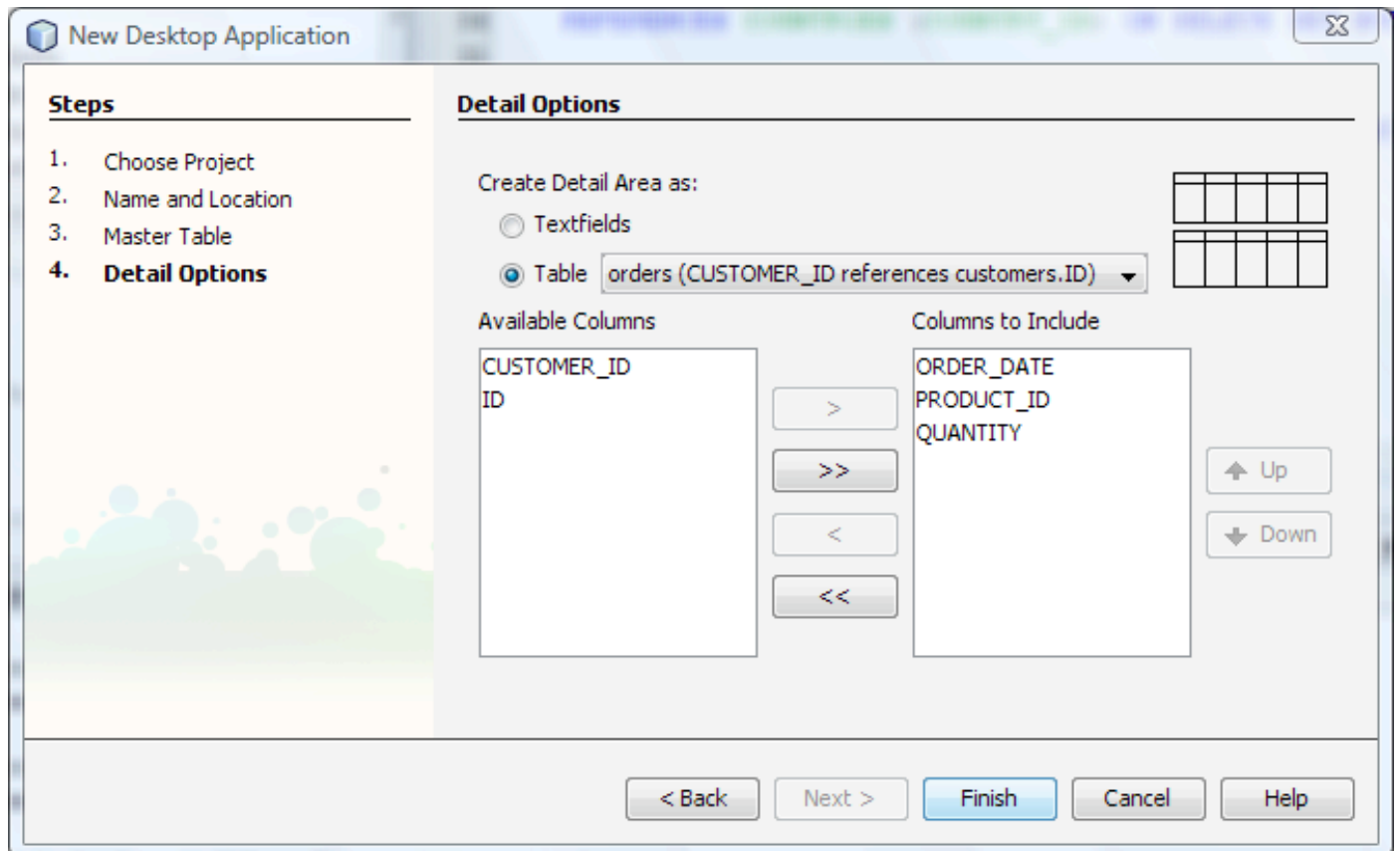
2. Select the Database Application shell.
3. Click Next.



4. In the Master Table page of the wizard, follow these steps:
 - Select the connection to the just-created MyBusinessRecords database.
 - Select the CUSTOMERS table.
 - Move the ID entry from the Columns to Include column to Available Columns.
 - Click Next.



5. In the Detail Options page, follow these steps:
 - o Click the Table radio button to create a JTable for the ORDERS table.
 - o Move the ID entry from the Columns to Include column to Available Columns.



6. Click Finish to exit the wizard.

Several classes are generated, including the following:

- The Customers and Orders entity classes, which represent the data from the CUSTOMERS AND ORDERS database tables.
- The main form with two JTable components that provide a master/detail view of the CUSTOMERS and ORDERS database tables. The view also contains buttons which are connected to actions for creating, deleting, and saving records.

The master table is bound to a list of Customers objects. That list of objects represents all of the rows of the CUSTOMERS database table.

The detail table is bound to a collection of Orders objects. That collection represents all of the rows of the ORDERS database table that are linked with the currently selected customer in the master table.

At this point you can choose Run > Run Main Project to see the main application window. However, the application does not yet function properly, because the database has some attributes for which the wizard did not generate necessary code. You will add this code in the next section of the tutorial.

Changing the Title of the Main Application Frame

When you run the application, the application's frame has the title "Database Application Example". For a simple JFrame, you would normally change the title by modifying the `title` property in the property sheet for the component. However, the frame of this application uses the Swing Application Framework's `FrameView` class, and the title is a general application property. You can modify such application properties in the Project Properties window.

To change the title of the main frame of the application:

1. In the Projects window, select the project's node and choose Properties.
2. In the Project Properties dialog box, select the Application node.
3. Change the Title property to `Customer and Order Records`.
4. If desired, modify the other properties as well, such as Description, and Splash Screen.

Customizing the Master/Detail View

The code that has been generated so far is for an application that is much simpler than the application created in this tutorial. The database has four tables, but the wizard only generated entity classes for the two tables that are displayed in the master/detail view. Also, the wizard did not take into account the foreign key from the Customers table to the Countries table or the foreign key from the Orders table to the Products table. The only relationship between tables that was acknowledged was the one pertinent to the master/detail relationship between the Customers and the Orders tables. To fix these problems, you need to do the following:

- Create entity classes for the COUNTRIES and PRODUCTS tables.
- Customize the Customers entity class to refer to the Countries entity class.
- Customize the Orders entity class to refer to the Products entity class.
- Update the binding code for the master and detail tables in the main form so that the Countries and Products entity classes are used.

Generating Missing Entity Classes

To create the missing entity classes:

1. Create entity classes for the Countries and Products tables by right-clicking the `customerrecords` package and choosing New > Entity Classes from Database.
2. Select the countries table and click the Add button.
3. Select the products table and click the Add button.
4. Click Next.
5. On the Entity Classes page of the wizard, click Next.
6. On the Mapping Options page, change the Collection Type to `java.util.List`.
7. Click Finish to exit the wizard.
New classes called Countries and Products should appear in your project.

Establishing Relations Between Entity Classes

Now you need to modify the Customers and Orders entity classes so that they properly handle the database foreign keys to the Countries and Products tables. `countryId` property is of type `Countries` instead of `Integer` and that it is joined with the `COUNTRIES` database table.

To establish the relation between the Customers and Countries entity classes:

1. In the Customers class, replace this field declaration and annotation

```
@Column(name = "COUNTRY_ID")
private Integer countryId;
```

with this code:

```
@JoinColumn(name = "COUNTRY_ID", referencedColumnName = "COUNTRY_ID")
@ManyToOne
private Countries countryId;
```

2. Press `Ctrl-Shift-I` to add the imports for the pasted code.
3. Change the type of the `getCountryId()` method from `Integer` to `Countries`.
4. In the `setCountryId()` method, change the types of `countryId` and `oldCountryId` from `Integer` to `Countries`.

To establish the relation between the Orders and Products entity classes:

1. In the Orders class, replace this field declaration and annotation

```
@Basic(optional = false)
@Column(name = "PRODUCT_ID")
private int productId;
```

with this code:

```
@JoinColumn(name = "PRODUCT_ID", referencedColumnName = "PRODUCT_ID")
@ManyToOne
private Products productId;
```

2. Press `Ctrl-Shift-I` to add the imports for the pasted code.
3. In the `public Orders(Integer id, Products productId, int quantity)` constructor, change the type of the `productId` argument from `int` to `Products`.
4. Change the type of the `getProductId()` method from `int` to `Products`.
5. In the `setProductId()` method, change the types of the `productId` parameter and the `oldProductId` variable from `int` to `Products`.

Customizing Column Binding Code

You also need to update the column binding for the country field so that it refers to the `country` property of the `Countries` object. (Code to use a country ID `int` was generated by the project since the skeleton was generated without having an entity class for the `COUNTRIES` table. If you do not make a change here, a `ClassCastException` will be thrown when you run the application.)

To fix the column binding in the master table:

1. Open the `CustomerRecordsView` class.
2. In the Design view of the class, right-click the top table and choose `Table Contents`.
3. Select the `Columns` tab.
4. In the customizer, select the `Country Id` row.

5. In the Expression combo box, type `${countryId.country}` or generate that expression by selecting `countryId > country` from the drop-down list. The `country` property is the part of `Countries` objects that contains the actual country name. Using this expression ensures that the column displays country names instead of a country identification number or the value of `Countries` object's `toString()` method.
6. Change the Title from `Country Id` to `Country`. This affects the column heading in the running application.
7. Click `Close` to save the changes.

Customizing the Detail Table

Similarly, you need to make changes to the `Product Id` column of the detail table to avoid `ClassCastException`s. You also need to take some extra steps when customizing the detail table. You will have multiple columns to show product information, but only one column has been generated so far.

In the `Products` table, there is a `productId` column, which is based on the `PRODUCT_ID` column from the `ORDERS` table. You could just modify the binding to show the product model and avoid a `ClassCastException`. However, it would be useful to show the data corresponding with the product in additional columns.

To complete the columns in the detail table:

1. In the Design view of the `CustomerRecordsView` class, right-click the bottom table and choose `Table Contents`.
2. Select the `Columns` tab.
3. In the customizer, select the `Product Id` row.
4. Change the Expression to `${productId.brand}`. After you do so, the type should also change to `String`.
5. Change the Title from `productId` to `Brand`.
6. Click the `Insert` button.
7. Select the row that has just been added to the table.
8. For Title, type `Type`.
9. For Expression, select `productId > prodType` from the drop-down list.
10. Click the `Insert` button again and select the newly added row.
11. For Title, type `Model`.
12. For Expression, select `productId > model` from the drop-down list.
13. Click the `Insert` button again and select the newly added row.
14. For Title, type `Price`.
15. For Expression, select `productId > price` from the drop-down list.
16. Click `Close` to apply the changes.

At this point, the application is partially functional. You can run the application and add, edit, delete, and save records. However, you can not yet properly modify the fields that are based on the `Countries` and `Products` entity classes. In addition, you have some work to make the `Order Date` field behave in a more user-friendly way.

You could make some adjustments to the `Country` and `Model` columns to use combo boxes that are bound to their respective tables. That would enable the user to select those fields without having to hand enter them. Instead, you will use dialog boxes as data entry mechanisms for these tables to make it harder for the user to accidentally delete data while browsing it.

Adding Dialog Boxes

In this section of the tutorial, you will:

- Add necessary utility classes.
- Adjust the actions in the main form.
- Create dialog boxes to edit data for each of the tables on the main form.
- Create intermediary beans to carry the data between the dialogs and form.
- Bind the fields in the dialog boxes to the intermediary beans.
- Create event handlers for the buttons in the dialogs.

Adding Necessary Utility Classes to the Project

This tutorial relies on several utility classes that specify things such as how to render and validate certain values.

To add all of the necessary utility classes to the project:

1. Unpack the [zip of file of utility classes](#) and unzip its contents on your system.
2. On your system, copy all of the files from the zip file and paste them into the folder that contains the project's `customerrecords` folder.
3. If your classes are in a different package than `customerreccords`, adjust the package statement in each of the files you have just added to the project.

Adjusting Action Details

Before you add the dialogs to the application, change the names of the New buttons to make them clearer. You will make the changes in the Actions that have been assigned for the buttons.

To change the button text:

1. Open the `CustomerRecordsView` class and select the Design view.
2. Right-click the first New button and select Set Action.
3. In the Set Action Dialog box, change the Text property to `New Customer`.
4. Change the Tool Tip field to `Create a new customer record`.
5. Click OK.
6. Right-click the second New button and select Set Action.
7. In the Set Action Dialog box, change the Text property to `Enter Order`.
8. Change the Tool Tip field to `Create a new customer order record`.
9. Click OK.

Creating the Customer Dialog

To create and populate the dialog box for the customer table, follow these steps:

1. Right-click the package that contains your classes and choose New > Other. Select Swing GUI Forms > JDialog Form template and name it `CustomerEditor`.
2. From the Palette window drag, drop, and arrange components for the customer's personal details.
Add labels for each of the following fields: first name, last name, address, city, state, zip code, country, and phone number.
Add text fields for all of the above fields, except for Country.
For Country, add a combo box.
3. Edit the display text for JLabels.
4. Add two buttons and name them Save and Cancel.
5. (Optional) Rename all of the components you have added to more memorable names, such as `firstNameLabel`. You can do this inline in the Inspector window.
6. Select the whole JDialog form.
7. In the Properties window, change the Title property to `Customer Editor`.

The resulting layout should look something like what you see below.

Note: For a detailed guide to using the GUI Editor's layout features, see [Designing a Swing GUI in NetBeans IDE](#).

Binding the Customer Dialog Box Fields

Now you need to bind the various fields to the corresponding columns in the table. You cannot bind directly to components from other forms in the Bind dialog box, so you will have to create an intermediary bean property of type `Customers` to hold the record. When the user clicks New Customer, the bean property will be given the value of an empty record that the user modifies in the dialog box. When the user clicks Save in the dialog box, the values from the bean property are passed back to the main form and the database.

To generate the intermediary bean property:

1. At the top of the design area of the `CustomerEditor` form, click the Source tab. Click somewhere within the class, such as in the line below the constructor.
2. Press Alt-Insert (or right-click and choose Insert Code) and choose Add Property.
3. In the Add Property dialog, name the property `currentRecord`, give it the type `Customers`, select Generate Getter and Setter, and select Generate Property Change Support.
4. Click OK to generate the property.

You now need to customize the generated `setCurrentRecord` method to fire a property change notification.

To fire the property change notification:

- Replace the `this.currentRecord = currentRecord;` line with these three lines:

```
Customers oldRecord = this.currentRecord;
this.currentRecord = currentRecord;
propertyChangeSupport.firePropertyChange("currentRecord", oldRecord, currentRecord);
```

Now you need to add code to open the Customer Editor dialog box when the user clicks the New Customer button. In addition, you need code to clear the `currentRecord` property.

To add code to open the dialog from the main view:

1. Open the `CustomerRecordsView` class and select the Source view.
2. Navigate to the `newRecord()` method.
3. Paste the following code at the bottom of the method.

```
JFrame mainFrame = CustomerRecordsApp.getApplication().getMainFrame();
CustomerEditor ce = new CustomerEditor(mainFrame, false);
ce.setCurrentRecord(c);
```

```
ce.setVisible(true);
```

You can now proceed with the binding of the dialog box's fields. You will bind the `text` property of each text field to the corresponding property of the `Customers` object represented by `currentRecord`. Similarly, you will bind the combo box's `selectedItem` property to the `countryId` property of `currentRecord`. You will bind the combo box's `elements` property to a list of Countries entities.

To bind the text fields to properties of the `currentRecord` bean:

1. Switch back to the Design view of the CustomerEditor class.
2. Select the First Name text field on your form.
3. In the Properties window, select the Binding category.
4. Next to the `text` property, click the ellipsis (...) button.
5. In the Bind dialog box, select Form as the Binding Source. (Note that Form is at the very bottom of the drop-down list.)
6. In the Binding Expression drop-down list, expand the `currentRecord` node and select the property corresponding to the text field that you are binding.
7. Click OK to close the Bind dialog box.
8. Repeat steps 1 through 6 for each of the other text fields.

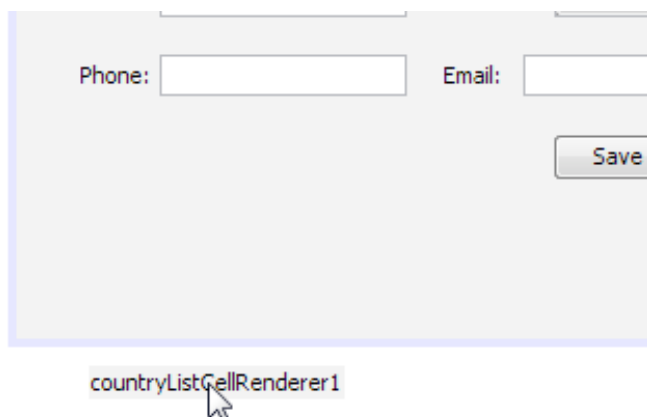
To bind the Country combo box:

1. Right-click the combo box and choose Bind > elements.
2. Click Import Data to Form, select the database connection, and select the Countries table. `countriesList` should appear as the binding source. Click OK.
3. Right-click the combo box again and choose Bind > selectedItem.
4. Select **Form** as the binding source and **currentRecord > countryId** as the expression. Click OK.

The combo box is almost ready to work properly in the dialog. It is set up to draw its values from the COUNTRIES database table, and the item that the user selects is then applied to the country field in the current record. However, you still need to customize the rendering of the combo box, since the values bound to the combo box are Countries objects, not simple names. You will do that by specifying a custom cell renderer. (For JTables and JLists, the beans binding library enables you to specify display expressions, thus avoiding the need to create a custom renderer. However, that feature does not exist yet for combo boxes.)

To get the combo boxes to render country names, do the following:

1. In the Project's window, right-click CountryListCellRenderer and choose Compile File. Compiling the file enables you to treat it as a bean that you can add to the form by dragging and dropping from within the IDE's GUI builder.
2. Select the CustomerEditor form in the Source Editor and select the Design view.
3. Drag the class from the Projects window to the white space surrounding the form, as shown in the screenshot below.



Doing so adds the renderer to your form as a bean, much like dragging a component from the Palette adds that component to your form.

4. In the form, select the combo box.
5. In the Properties window, select the Properties category.
6. Scroll to the `renderer` property and choose `countryListCellRenderer1` from the drop-down list for that property.

Now you should be able to run the application, press the first New button, and enter data in the dialog. You should also be able to select a country from the Country combo box. The Save and Cancel buttons on the dialog do not do anything yet, but you can save the records from the main frame. You will code those buttons later.

Creating the Order Entry Dialog Box

Now you will create a dialog box for the detail part of the form.

To create the Order Editor dialog box:

1. Right-click the `customerrecords` package and choose New > JDialog Form.
2. Name the class `OrderEditor` and click Finish.
3. Drag and drop the following components to the form:
 - Label components for Date, Product, Quantity, Price, and Total.
 - Formatted Field components for the date, price, and total.
 - A label to display the date format that needs to be entered. Give this label the text `(MMM DD, YYYY - e.g. Apr 17, 2008)`. If you plan to use a different date format, add text that corresponds to the format that you plan to use.
 - A combo box for the product field.
 - A text field for the Quantity field.
 - Two buttons for the Save and Cancel commands.
4. Select the whole JDialog form.
5. In the Properties window, change the Title property to `Order Editor`.

Now that the Order Editor has its visual design, you need to do the following things:

- Create an intermediary bean to carry the record values back to the main form.
- Specify the formatting for the formatted text fields.
- Bind the various fields.
- Set the behavior of the formatted field for the date.
- Handle the currency formatting of the Price and Total fields.


Connecting the Order Editor Dialog Box With the Main Form

As you did for the CustomerEditor dialog box, you will have to create an intermediary bean property of type `Orders` to hold the record. When the user presses Enter Order, the property will be given the value of the currently selected order record.

To create the bean property:

1. At the top of the design area of the `OrderEditor` form, click the Source tab.
2. Click somewhere within the class, such as in the line below the constructor.
3. Press Alt-Insert (or right-click and choose Insert Code) and choose Add Property.
4. In the Add Property dialog, name the property `currentOrderRecord`, give it the type `Orders`, select Generate Getter and Setter, and select Generate Property Change Support.
5. Click OK to generate the property.
6. In the `setCurrentOrderRecord()` method, replace the `this.currentOrderRecord = currentOrderRecord;` line with these three lines:

```
Orders oldRecord = this.currentOrderRecord;
this.currentOrderRecord = currentOrderRecord;
propertyChangeSupport.firePropertyChange("currentOrderRecord", oldRecord, currentOrderRecord);
```



Now you need to add code to open the Order Editor dialog box when the user clicks the Enter Order button. In addition, you need code to clear the `currentOrderRecord` property.

To connect the dialog with the Enter Order button:

1. Open the `CustomerRecordsView` class and select the Source view.
2. Navigate to the `newDetailRecord()` method.
3. Paste the following code at the bottom of the method.

```
JFrame mainFrame = CustomerRecordsApp.getApplication().getMainFrame();
OrderEditor oe = new OrderEditor(mainFrame, false);
oe.setCurrentOrderRecord(o);
oe.setVisible(true);
```

Binding the Order Editor's Fields

Note: For the formatted text fields, you bind to the `value` property, not the `text` property. See the [Java Tutorial](#) for details.

To bind the Order dialog box's fields:

1. Open the `OrderEditor` class in Design view.
2. Select the text field for the date.
3. In the Properties window, select the Binding category.
4. Select the `value` property and click the ellipsis (...) button that is next to the property.
5. In the Bind dialog box, select Form as the binding source, and select `currentOrderRecord > orderDate` as the binding expression.
6. Select the combo box for the product and click the ellipsis (...) button for the `elements` property.
7. In the Bind dialog box, click Import Data to Form.
8. Select the `MyBusinessRecords` database connection and select the `products` table.
When you import this data to the form, code for a `List` object of `Products` is generated. This object (`listProducts`) is then set as the binding source.
9. Leave the binding expression as `null` and click OK.
10. Click the ellipsis (...) button for the combo box's `selectedItem` property.
11. Select Form as the binding source, and select `currentOrderRecord > productId` as the binding expression.
12. Select the Quantity text field.
13. Click the ellipsis (...) button for the `text` property.
14. Select Form as the binding source, and select `currentOrderRecord > quantity` as the binding expression.
15. Select the price's formatted field.
16. Click the ellipsis (...) button for the `value` property.
17. Select Form as the binding source, and select `${currentOrderRecord.productId.price}` as the binding expression.
18. Select the total's formatted field.
19. Click the ellipsis (...) button for the `value` property.
20. Select Form as the binding source, and type `${currentOrderRecord.productId.price*currentOrderRecord.quantity}` as the binding expression.
This custom binding expression enables you to generate the total price for the order by multiplying the price of the selected item times the quantity selected by the user.

Setting Date and Currency Formatting

For the order date, price, and total, you have added formatted text fields, which make it easy to provide formatting.

To set the date and currency formatting for those fields:

1. Select the formatted field for the date.
2. In the Properties window, select the Properties category.
3. Click the ellipsis (...) button next to the `formatterFactory` property.
4. In the `formatterFactory` property editor, select `date` in the Category column. Then select `Default` in the Format column.
5. Click OK to exit the dialog box.
6. Select the price formatted field.
7. In the Properties window, clear the `editable` property. (You do not want users to edit this field. The value of this field will be derived from the price property of the selected item in the product combo box.)
8. In the `formatterFactory` property editor, select `currency` in the Category column. Then select `Default` in the Format column.
9. Click OK to exit the dialog box.
10. Follow steps 6-9 for the total formatted field.

Now when you run the application and select an item, the price and currency fields should display as currencies.

Rendering the Product combo box

Now you need to modify the rendering of the Product combo box. As you did for the Country combo box in the New Customer dialog box, you need to use a custom cell renderer. In this cell renderer, you will combine information from several properties so that the user can see the type, brand, and model listed for each item.

To apply the cell renderer for the product.

1. Right-click the `ProductListCellRenderer` class and choose Compile File.
2. Open the `OrderEditor` form in Design view.
3. Drag the class from the Projects window to the white space surrounding the form in the same way that you did for the `CountryListCellRenderer`.
4. In the form, select the combo box.
5. In the Properties window, scroll to the `renderer` property and choose `productListCellRenderer1` from the drop-down list for that property.

Pre-populating the Order Dialog With Data

Since the point of this dialog is to enter new orders, it is likely that the user of the application might be entering an order for the current date. To save the user the trouble of entering that date, you can pre-populate that field with the current date.

Likewise, you should change the default value for quantity, since 0 is an invalid value. You could create a validator here, but for now it is simpler and more practical to simply set a reasonable default value.

To pre-populate the date and quantity fields in the Order Editor dialog box:

1. Open the `CustomerRecordsView` class and select the Source view.
2. Navigate to the `newDetailRecord()` method.
3. Below the `customerrecords.Orders o = new customerrecords.Orders();` line paste these lines:

```
o.setOrderDate(new java.util.Date());
o.setQuantity(1);
```

Now when you run the application, the current date should appear in the Date field when you open the dialog box and the default quantity should be 1.

Activating the Save and Cancel Buttons in the Dialog Boxes

Now you need to finish coding the connection between the dialogs and the main form.

Activating the Customer Editor's Save and Cancel Buttons

First hook up the buttons in the CustomerEditor dialog with appropriate event-handling code. You already have `save()` and `refresh()` actions that are provided with the skeleton application. You will code the dialog so that the buttons reuse these actions. You can accomplish this by setting up a boolean property in the dialog that returns true when the Save Record button is pushed and returns false when Cancel is selected. Based on the value that is returned when the dialog is closed, the `save()` or the `refresh()` action will be run from the CustomerRecordsView class.

To set up the property, do the following:

1. Open up the CustomerEditor file and select the Source view.
2. Place the cursor above the `propertyChangeSupport` field.
3. Press Alt-Insert and choose Add Property.
4. In the Add Property dialog, type `customerConfirmed` as the property name.
5. Set the type to boolean.
6. Make sure the Generate Getters and Setters checkbox is selected.
7. Click OK to close the dialog box and generate the code.

You will set this property's value in event handling code for the buttons.

To create the event listeners and handlers:

1. Switch to the Design view for the CustomerEditor class.
2. Select the Save button in the CustomerEditor form.
3. In the Properties window, click the Events button.
4. Click the ellipsis (...) button next to the `actionPerformed` property.
5. In the Handlers for actionPerformed dialog box, add a handler called `saveCustomer`.
6. Within the `saveCustomer` method in the Source Editor (where the cursor jumps after you create the new handler), type the following code:

```
setCustomerConfirmed(true);  
setVisible(false);
```

7. Repeat steps 2-5 for the Cancel button and call its handler `cancelCustomer`.
8. In the `cancelCustomer` method, type the following:

```
setCustomerConfirmed(false);  
setVisible(false);
```

In the CustomerRecordsView class, navigate to the `newRecord()` method and add the following code to the bottom of the method:

```
if (ce.isCustomerConfirmed()) {  
    save().run();  
} else {  
    refresh().run();  
}
```

Since the `save()` and `refresh()` actions act on any changes made during the application's session, you should make the dialog modal and make the tables in the main form uneditable. Another reason to make the dialog modal is so that when the user presses either the Save or Cancel button, the `setVisible()` method does not return until the event handler (which includes the `setCustomerConfirmed` method) has run.

To make the dialog modal:

1. Open the Design view of the CustomerEditor class.

2. Select the dialog.
3. In the Properties window, click Properties and select the checkbox for the `modal` property.

To make the main form's Customers table uneditable:

1. Open the `CustomerRecordsView` class and select the Design view.
2. Right-click the top table and choose Table Contents.
3. In the Customizer dialog, select the Columns tab.
4. For each column, clear the Editable checkbox.
5. Click Close.

You should now be able to create new records and save them from the Customer Editor. You should also be able to create a new record and cancel from the Customer Editor.

In the `RefreshTask` inner class, `Thread.sleep` is called four times to slow down the rollback code to better demonstrate how Swing Application Framework tasks work. You do not need this code for this application, so delete those four statements. Similarly, you do not need a try/catch block here, so delete the `try` and `catch` statements as well (but leave the rest of the body of the `try` block).

Activating the Order Editor's Save and Cancel Buttons

As you did for the Customer Editor, you need to hook up the buttons in the `OrderEditor` dialog with the `save()` and `refresh()` actions that are provided with the skeleton application. As you have done before, you will use a boolean property in the dialog that returns true when the Save Record button is pushed and returns false when Cancel is selected. Based on the value that is returned when the dialog is closed, the `save()` or the `refresh()` action will be run from the `CustomerRecordsView` class.

To set up the property, do the following:

1. Open up the `OrderEditor` file and select the Source view.
2. Place the cursor above the `propertyChangeSupport` field.
3. Press Alt-Insert and choose Add Property.
4. In the Add Property dialog, type `orderConfirmed` as the property name.
5. Set the type to boolean.
6. Make sure the Generate Getters and Setters checkbox is selected.
7. Click OK to close the dialog box and generate the code.

You will set this property's value in event handling code for the buttons.

To create event handling code for the buttons:

1. Switch to the Design view for the `OrderEditor` class.
2. Select the Save button in the `OrderEditor` form.
3. In the Properties window, click the Events button.
4. Click the ellipsis (...) button next to the `actionPerformed` property.
5. In the Handlers for `actionPerformed` dialog box, add a handler called `saveOrder`.
6. Within the `saveOrder` method in the Source Editor (where the cursor jumps after you create the new handler), type the following code:

```
setOrderConfirmed(true);  
setVisible(false);
```

7. Repeat steps 2-5 for the Cancel button and call its handler `cancelOrder`.
8. In the `cancelOrder` method, type the following:

```
setOrderConfirmed(false);
```



```
setVisible(false);
```

In the `CustomerRecordsView` class, navigate to the `newDetailRecord()` method and add the following code to the bottom of the method:

```
if (oe.isOrderConfirmed()) {
    save().run();
} else {
    refresh().run();
}
```

To make the dialog modal:

1. Open the Design view of the `OrderEditor` class.
2. Select the dialog.
3. In the Properties window, click Properties and select the checkbox for the **modal** property.

To make the main form's Orders table uneditable:

1. Open the `CustomerRecordsView` class in the Source Editor and select the Design view.
2. Right-click the bottom table and choose Table Contents.
3. In the Customizer dialog, select the Columns tab.
4. For each column, clear the Editable checkbox.
5. Click Close.

Verifying Deletion of Records

You have moved access to most of the main actions from the main form to the dialog boxes. The only exception is the Delete action. Previously in the application, deletes were confirmed by pressing Save and were cancelled by pressing Refresh. Since the Save and Refresh buttons are not on the main form anymore, you need to replace this functionality. You will do so by adding a confirmation dialog within the `deleteRecord()` and `deleteDetailRecord()` methods. If the user clicks OK, the `save()` method will be called and therefore the record will be deleted permanently. If the user clicks Cancel, the `refresh()` method will be called and the record will be restored.

To add the confirmation dialogs to the Delete buttons:

1. Open the main form in the Source view.
2. Delete the `deleteRecord()` method and replace it with the following code:

```
@Action(enabledProperty = "recordSelected")
public void deleteRecord() {
    int n = JOptionPane.showConfirmDialog(null, "Delete the records permanently?", "Warning",
        JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE, null);
    if (n == JOptionPane.YES_OPTION) {
        int[] selected = masterTable.getSelectedRows();
        List<customerrecords.Customers> toRemove = new ArrayList<customerrecords.Customers>(sel
        for (int idx = 0; idx < selected.length; idx++) {
            customerrecords.Customers c = list.get(masterTable.convertRowIndexToModel(selected[
            toRemove.add(c);
            entityManager.remove(c);
        }
        list.removeAll(toRemove);
        save().run();
    } else {
        refresh().run();
    }
}
```

3. Press Ctrl-Shift-I to add the missing import statements.
4. Delete the `deleteDetailRecord()` method and replace it with the new version of the method:

```

@Action(enabledProperty = "detailRecordSelected")
public void deleteDetailRecord() {
    Object[] options = {"OK", "Cancel"};
    int n = JOptionPane.showConfirmDialog(null, "Delete the records permanently?", "Warning",
        JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE, null);
    if (n == JOptionPane.YES_OPTION) {
        int index = masterTable.getSelectedRow();
        customerrecords.Customers c = list.get(masterTable.convertRowIndexToModel(index));
        List<customerrecords.Orders> os = c.getOrdersCollection();
        int[] selected = detailTable.getSelectedRows();
        List<customerrecords.Orders> toRemove = new ArrayList<customerrecords.Orders>(selected);
        for (int idx = 0; idx < selected.length; idx++) {
            selected[idx] = detailTable.convertRowIndexToModel(selected[idx]);
            int count = 0;
            Iterator<customerrecords.Orders> iter = os.iterator();
            while (count++ < selected[idx]) {
                iter.next();
            }
            customerrecords.Orders o = iter.next();
            toRemove.add(o);
            entityManager.remove(o);
        }
        os.removeAll(toRemove);
        masterTable.clearSelection();
        masterTable.setRowSelectionInterval(index, index);
        list.removeAll(toRemove);
        save().run();
    } else {
        refresh().run();
    }
}
}

```

Adding Edit Functionality to the Dialog Boxes

You can now run the application and click New Customers to add a new record. When you press Save in the New Customers dialog, the record is saved. When you press Cancel, the new record you have changed is rolled back.

However, you can no longer edit existing records, because disabled editing of the tables in the main form. To solve this, you will add Edit buttons to the main customer form so that you can edit existing records. For event-handling, you will take advantage of the Swing Application Framework's Action facility.

Making Customer Records Editable

To add the button and its corresponding event-handling code, do the following:

1. Open the CustomerRecordsView class and select the Design view.
2. Drag the New Customers button a bit to the left.
3. Drag a button from the palette into the opening just created.
4. Right-click the button and choose Set Action.
5. In the Action field, select Create New Action.
6. For Action Method, type `editCustomer`.
7. For Text, type `Edit Customer`.
8. Click the Advanced Tab and select `recordSelected` for the Enabled Property.
This generates an annotation attribute to ensure that the button and any other trigger for the action (e.g. a menu item) are only enabled when a record is selected.
9. Click OK to close the dialog box.
The Source view of the file should appear with the cursor in the newly generated `editCustomer()` method.
10. Within the method, paste the following code:

```

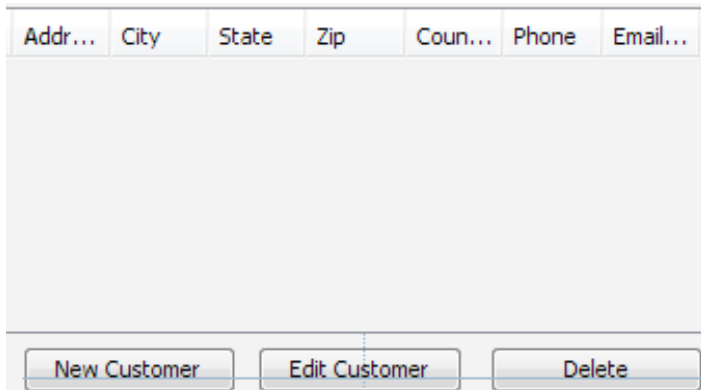
setSaveNeeded(true);
JFrame mainFrame = CustomerRecordsApp.getApplication().getMainFrame();
CustomerEditor ce = new CustomerEditor(mainFrame, false);
ce.setCurrentRecord(list.get(masterTable.convertRowIndexToModel(masterTable.getSelectedRow()));
ce.setVisible(true);
if (ce.isCustomerConfirmed()) {
    save().run();
} else {
    refresh().run();
}

```

}



Most of that code is copied straight from the `newRecord` action. The key difference is the line `ce.setCurrentRecord(list.get(masterTable.convertRowIndexToModel(masterTable.getSelectedRow())));`, which populates the current record in the dialog with the currently selected record.



The Customer part of the application is almost completely set. You should be able to freely add, edit, and delete records from your CUSTOMERS table using the specialized GUI you have created.

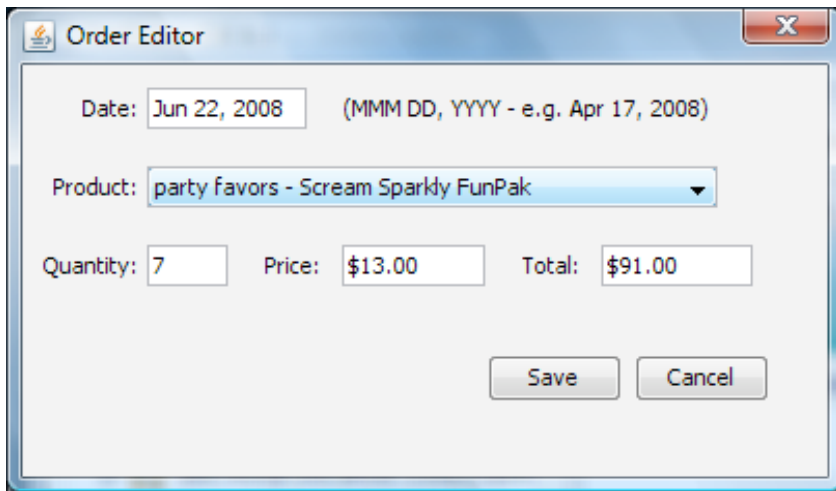
Making Order Records Editable

To add the Edit Orders button and its corresponding event-handling code, do the following:

1. In the Design view of the `CustomerRecordsView` class, delete the Refresh and Save buttons.
2. Drag the Delete button to the right.
3. Drag a button from the palette into the opening just created.
4. Right-click the button and choose Set Action.
5. In the Action field, select Create New Action.
6. For Action Method, type `editOrder`.
7. For Text, type `Edit Order`.
8. Click the Advanced Tab and select `detailRecordSelected` for the Enabled Property.
9. Click OK to close the dialog box.
The Source view of the file should appear with the cursor in the new `editOrder` method.
10. Within the `editOrder` method, paste the following code:

```
setSaveNeeded(true);
int index = masterTable.getSelectedRow();
customerrecords.Customers c = list.get(masterTable.convertRowIndexToModel(index));
List<customerrecords.Orders> os = c.getOrdersCollection();
JFrame mainFrame = CustomerRecordsApp.getApplication().getMainFrame();
OrderEditor oe = new OrderEditor(mainFrame, false);
oe.setCurrentOrderRecord(os.get(detailTable.getSelectedRow()));
oe.setVisible(true);
if (oe.isOrderConfirmed()) {
    save().run();
} else {
    refresh().run();
}
```

Now when you run the application, all of the key elements are in place. You can create, retrieve, update, and delete records for customers and orders. In the screenshot below, you can see the Order Editor dialog as it appears after having selected a record and pressed the Edit Order button.



The section below shows some other things that you can do to enhance and fine tune the application.

Currency Rendering, Date Verifying, and Search

You should now have a fully functioning application. However, there are still many ways to enhance the application. Below are examples of some ways that you can improve the application.

Rendering the Currency in the Main View

You have handled the formatting of the dates and currencies within the Orders Editor, but you have not yet done so for the CustomerRecordsView class. You do not need to do anything for the date field. The format is effectively passed between the Order Editor and the main form. However, that is not the case for the Price field. You will need to add a currency renderer class to render that field correctly.

To render the Price field with currency formatting in the main view:

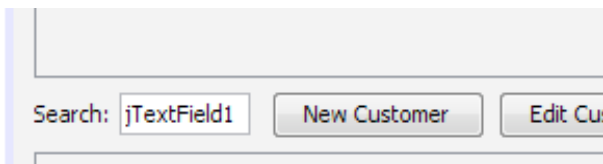
1. In the Projects window, right-click the `CurrencyCellRenderer` class and choose Compile File.
2. Open the `CustomerRecordsView` class and switch to the Design view.
3. Drag the `CurrencyCellRenderer` class from the Projects window and drop it in white area surrounding the form. A node called `currencyCellRenderer1` should appear in the Inspector window.
4. Right-click the lower table in the form and choose Table Contents.
5. Click the Column tab.
6. Select the `price` column.
7. In the Renderer combo box, select `currencyCellRenderer1`.

Now when you run the application, the price should appear with a dollar sign (\$), a decimal point, and two digits after the decimal point.

Adding a Search Function

Now you will add a search function for the customer table. You will use mechanisms that already exist in Swing and the Beans Binding library. You will create a binding between the `rowSorter` property of the master table a text field for the search string. For this binding you will need a binding converter so that the table knows how to respond to the search string.

First of all, add a label and a text field for the search field as shown below.



Now you will add a converter class to the project.

1. In the Projects window, right-click the `RowSorterToStringConverter` class and choose Compile File.

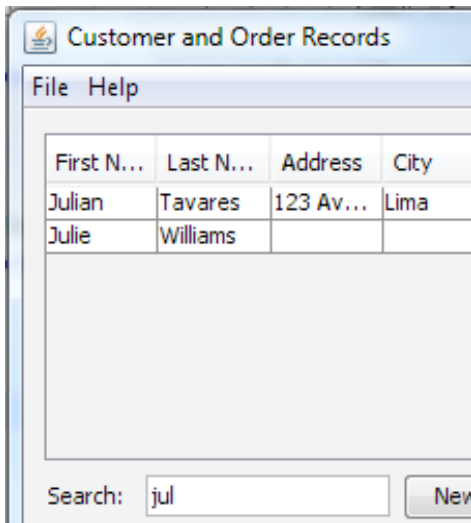
2. Drag the class from the Projects window and drop it in the white area surrounding the form.
3. In the Inspector window, select the `rowSorterToStringConverter1` node and set its `table` property to `masterTable`.

You will use this converter when you create the binding.

To create the binding:

1. In the main form, right-click the Search text field and choose Bind > text.
2. In the Bind dialog, select `masterTable` as the binding source and `rowSorter` as the expression.
3. Click the Advanced tab of the dialog box.
4. From the Converter combo box, select `rowSorterToStringConverter1`.
5. Click OK to close the dialog and generate the binding code.

Now when you run the application, you should be able to type in the Search Filter field and see that the list of rows is reduced to only rows that contain text matching what you have typed.



However, adding this search feature creates a side effect. If you use the search and then click New Customer, an exception appears because the code to select the new row is determined according to number of records in the database table, not according to the number of records currently displayed in the table.

To fix the exception:

- Replace the following line in the `newRecord`:

```
int row = list.size() - 1;
```

with:

```
int row = masterTable.getRowCount() - 1;
```

Verifying Date Formatting

Though you have a date formatter in the Order Editor dialog box, you have not yet specified what to do when the user does not correctly enter the date. The default behavior is to return to the previously entered valid date. To customize this behavior, you can specify a verifier for the field. You can use the `DateVerifier` class, which is among the utility classes for this project.

To connect the date verifier to the Order Date column:

1. In the Projects window, right-click the `DateVerifier` class and choose Compile File.
2. Open the `OrderEditor` and switch to Design view.
3. Drag the `DateVerifier` class from the Projects window to the white space that surrounds the form.
4. Select the formatted text field for the Date.

5. In the Properties window, click the Properties tab, and select the `dateVerifier1` from the combo box for the Input Verifier property. This property appears within the Other Properties section of the tab.

[Send Us Your Feedback](#)

See Also

For a more general introduction to using the IDE's GUI Builder, see [Introduction to GUI Building](#).

For a more comprehensive guide to the GUI Builder's design features, including video demonstrations of the various features, see [Designing a Swing GUI in NetBeans IDE](#).

To see how you can use the Java Desktop Application project template to build a database application with a Master/Detail view, see [Building a Java Desktop Database Application](#).

For other information on Java application development in the IDE, see the [Basic IDE and Java Programming Learning Trail](#).

For more information on Beans Binding, see the [Beans Binding project page on java.net](#).

For more information on beans binding in the IDE, see the [Binding Beans & Data in a Desktop Application](#).

For information on using Hibernate for a Swing application's persistence layer, see [Using Hibernate in a Java Swing Application](#).

For general tips and tricks on using the GUI Builder in NetBeans IDE, see the [GUI Editor FAQ](#) and [Patrick Keegan's web log](#).