Exercise 1:

Creating Simple Classes and JavaBean™ Data Models

Case Study: Creating Simple Classes & JavaBean Data Models

Estimated completion time: 45 minutes.

In this exercise, you create a simple `HelloWorld` class to familiarize yourself with the course integrated development environment (IDE) and then build two data model beans for use in the banking application developed during this course. A data model bean is a read-only, serializable object with fields that reflect the state of business data in an application's data store. Components requiring read-only access to an application's business data can use local data model beans in lieu of a resource intensive database connection or a remote call to an Enterprise JavaBean™. Data model beans are analogous to the value objects discussed in the J2EE Blueprints.As shown in Figure 1, the model beans created in this exercise reflect state information for the account and customer database tables. Both data models use only a subset of the information contained in a row of the corresponding database table.

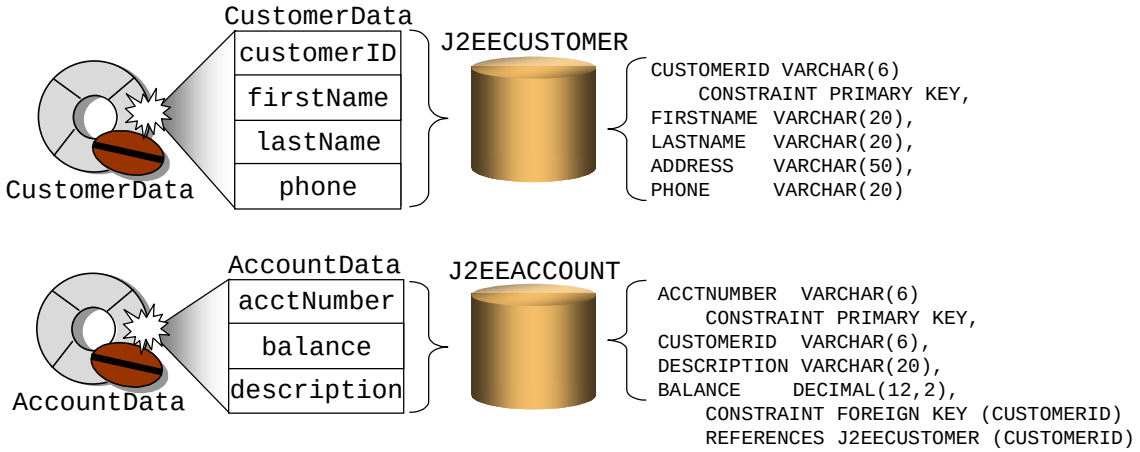

Figure 1 - The `CustomerData` and `AccountData` Model Beans

There are two sections to this lab exercise. In the first section, you code and test the `HelloWorld` class. In Section 2, you code data model beans for the bank application `Customer` and `Account` business classes. You then test the data model beans using a simple test class.

After completing this lab you will be able to:

- Use the course IDE to create Java packages, classes, and beans
- Use the course IDE to edit and compile Java components
- Execute a small application within the IDE

Discussion Topics:

- What is a Java™ package?
- What is the function of a data model bean?

Important Terms and Concepts

Please refer to the JDK™ Javadoc or the lab Appendix for information on the important classes and methods used in this exercise, including parameters and return values.

`java.util`

```
public class HashMap extends AbstractMap implements Map, Cloneable,
                                    Serializable
   Object clone()
   Object put(Object key, Object value)
   Object get(Object key)
   Set keySet()
   Object remove(Object key)
public interface Set extends Collection
   Iterator iterator()
```

Setup Instructions

Before attempting this exercise, you should be familiar with the IDE used for this course. Refer to the IDE documentation for additional information required to complete this exercise. Use the "Guidelines for Configuring the Course IDE" located in the lab Appendix or the accompanying course material as a resource when completing this exercise. You can view an example of the completed exercise in the "Solution Code" section of this lab module.

Section 1

Coding a `Hello World` Class

In this exercise, you create a Java package and build a small application that prints the message "Hello, World" to the system output screen.

1. In the IDE workspace, create a Java package named `javahello`.
2. Create a class inside the `javahello` package using the criteria listed below:
   - Type: `Main – Main` is a template provided with Forte for Java, Community Edition for ceating a simple Java class containing a `main` method.
   - Name: `HelloWorld`
3. Add a method to the `HelloWorld` class using the following criteria:
   - Name: `sayHello`
   - Returns: `void`
   - Visibility: `public`
   - Parameters: None
   - Functionality: Code the method to output "Hello, World". To print to standard output, use the `System.out.println` method.
4. Modify the main method to instantiate an instance of the `HelloWorld` class and call the class's `sayHello` method.

Compile

5. Compile the `HelloWorld` class, and correct any errors before continuing.

Test

6. Execute the `HelloWorld` class, and view the output in the runtime console.

Section 2

Creating Data Model Beans for Business Classes

In this exercise, you create a Java package named `J2EEApp` and begin constructing the components of a simple banking application used within the course labs. The following steps guide you through creating two data model beans, `CustomerData` and `AccountData`, to represent elements from the associated `Customer` and `Account` business classes. These model beans are used in later exercises to hold information for a particular customer or account. Model beans are serializable components with fields that represent the state of an application's business data. The fields on a model bean are usually intended for read-only access. Using model beans to store the state of application data allows other application components to read application data without incurring the overhead of connecting to the data store. In this instance, both the `AccountData` and `CustomerData` model beans require a constructor and a `toString` method to display the class data using the `println` method. After creating the data model beans, you test the beans using a simple Java client.

1. Create a Java package named `J2EEApp` in the IDE workspace.
2. Create a data model bean inside the `J2EEApp` package using the following criteria. The bean serves as a model to hold customer information retrieved from a persistent store.
   - Name: `CustomerData`
   - Visibility: `public`
   - Superclass: `Object`
   - Implements: `java.io.Serializable`
3. Add the following properties to the `CustomerData` class. Select the check boxes in the property creation wizard to automatically create class fields and return methods.
   - Name: `customerID`; Type: `String`; Mode: Read-only; Accessor methods: `get`
   - Name: `firstName`; Type: `String`; Mode: Read-only; Accessor methods: `get`
   - Name: `lastName`; Type: `String`; Mode: Read-only; Accessor methods: `get`
   - Name`: phone`; Type: `String`; Mode: Read-only; Accessor methods: `get`
4. The `CustomerData` class requires two constructors as shown below. Initialize the class's private fields with the values in the parameter list when implementing the parameterized constructor.
   - `CustomerData()`
   - `CustomerData(String pCustomerID, String pFirstName, String pLastName, String pPhone)`
5. The Java object class has a `toString` method that you can override on subclasses. This is useful for converting an object's field values into a `String`, so that you can print them out for testing. Add a method to the `CustomerData` bean as described below:
   - Name: `toString`
   - Visibility: `public`
   - Returns: `String`
   - Parameters: None

- Functionality: Code the method to return a `String` containing labels and values for the `CustomerData` fields. The simplest way to do this (although not the most efficient) is to use the "+" operator to concatenate the string values. For example:

```
String theString =
            "\ncustomerID: " + this.customerID +
            "\nfirstName: " + this.firstName +
            "\nlastName: " + this.lastName +
            "\nphone: " + this.phone;
return theString;
```

6. Create a data model bean inside the `J2EEApp` package using the following criteria. The bean serves as a model to hold account information retrieved from a persistent store.

   - Name: `AccountData`
   - Visibility: `public`
   - Superclass: `Object`
   - Implements: `java.io.Serializable`

7. Add the following properties to the `AccountData` class. Select the check boxes in the property creation wizard to automatically create class fields and return methods.

   - Name: `acctNumber`; Type: `String`; Mode: Read-only; Accessor methods: `get`
   - Name: `balance`; Type: `double`; Mode: Read-only; Accessor methods: `get`
   - Name: `description`; Type: `String`; Mode: Read-only; Accessor methods: `get`

8. The `AccountData` class requires two constructors as shown below. Initialize the class's Private fields with the values in the parameter list when implementing the parameterized constructor.

   - `AccountData()`
   - `AccountData(String pAcctNumber, double pBalance, String pDescription)`

9. Add a `toString` method to the `AccountData` bean as described below:

   - Name: `toString`
   - Visibility: `public`
   - Returns: `String`
   - Parameters: None
   - Functionality: Code the method to return a `String` containing labels and values for the `AccountData` fields. For example:

```
String theString =
            "\nacctNumber: " + this.acctNumber +
            "\ndescription: " + this.description +
            "\nbalance: " + this.balance;
return theString;
```

Compile

10. Compile the two data model beans, and correct any errors before continuing.

Create a Test Client

11. Create a class in the `J2EEApp` package as described below:
    - Name: `BankAccess`
    - Type: `Main`
12. Code the main method of the `BankAccess` class to instantiate a `BankAccess` object, construct a new `CustomerData` object and a new `AccountData` object using the parameterized constructors, and then print the data model objects using the `System.out.println` method (it automatically calls the object's `toString` method that you wrote). As a reminder, here is the Java syntax for invoking the `AccountData` class constructor:

```
AccountData myAccountData = new AccountData(<params>);
```

Compile

13. Build the application in the `J2EEApp` package. Correct any errors before continuing with this exercise.

Run the Application

14. Execute the `BankAccess` class, and view the output in the IDE run console.

Discussion Topics:

- What are some of the issues that must be considered when using data model beans to represent the state of data elements in an application's persistent data store?
- In what way does a data model bean differ from a typical JavaBean™ component?
- Why must a data model bean implement the `java.io.Serializable` interface?

Solution Code: Section 1

```
// HelloWorld.java

package javahello;

public class HelloWorld extends Object {

   /** Creates new HelloWorld */
   public HelloWorld() {
   }

   public static void main (String args[]) {
      System.out.println("Entering HelloWorld.main()");
      HelloWorld hw = new HelloWorld();
      hw.sayHello();
      System.out.println("Leaving HelloWorld.main()");
   }

   public void sayHello() {
      System.out.println("Entering HelloWorld.sayHello()");
      System.out.println("Hello, World");
      System.out.println("Leaving HelloWorld.sayHello()");
   }
}
```

```
Solution Code: Section 2
// AccountData.java

package J2EEApp;

import java.beans.*;

public class AccountData extends Object
   implements java.io.Serializable {

   private String acctNumber;
   private double balance;
   private String description;

   public AccountData() {}

   public AccountData(String pAcctNumber,double pBalance,
                      String pDescription) {
      System.out.println("Creating new AccountData");
      this.acctNumber = pAcctNumber;
      this.balance = pBalance;
      this.description = pDescription;
   }

   public String getAcctNumber() { return acctNumber;}

   public double getBalance() { return balance; }

   public String getDescription() { return description; }

   public String toString() {
      String theString = "\nacctNumber: " + this.acctNumber +
                         "\ndescription: " + this.description +
                         "\nbalance: " + this.balance;
      return theString;
   }
}

************************************************************

// CustomerData.java

package J2EEApp;

import java.beans.*;

public class CustomerData extends Object
      implements java.io.Serializable {

   private String customerID;
   private String firstName;
   private String lastName;
   private String phone;

   public CustomerData() {}

   public CustomerData(String pCustomerID,String pFirstName,
                       String pLastName,String pPhone) {
```

```
      System.out.println("Creating new CustomerData");
      this.customerID = pCustomerID;
      this.firstName = pFirstName;
      this.lastName = pLastName;
      this.phone = pPhone;
   }

   public String toString() {
      String theString = "\ncustomerID: " + this.customerID +
                         "\nfirstName: " + this.firstName +
                         "\nlastName: " + this.lastName +
                         "\nphone: " + this.phone;
      return theString;
   }

   public String getCustomerID() { return customerID; }

   public String getFirstName() { return firstName; }

   public String getLastName() { return lastName; }

   public String getPhone() { return phone; }
}

*************************************************************

// BankAccess.java

package J2EEApp;

public class BankAccess extends Object {

   public BankAccess() {}

   public static void main (String args[]) {
      System.out.println("Entering BankAccess.main()");
      AccountData myAccountData =
         new AccountData("1",1000.00,"Savings");
      CustomerData myCustomerData =
         new CustomerData("11", "Tom", "Thumb", "555-123-4567");
      System.out.println("myAccountData is: " + myAccountData);
      System.out.println("myCustomerData is: " + myCustomerData);
      System.out.println("Leaving BankAccess.main()");
   }
}
```

Answers for Discussion Topics:

- What is a Java package?

  Answer: A Java package is a mechanism, analogous to a folder, for organizing application files.

- What is the function of a data model bean?

  Answer: Data model beans hold the state representation of data elements from an entity data store. Data model beans are used by client components requiring read-only access to data values stored in an entity data store normally accessed using entity EJBs. Data model beans reduce the necessity for remote calls to an entity EJB to retrieve entity state information.

- What are some of the issues that must be considered when using data model beans to represent the state of data elements in an application's persistent data store?

  Answer: Data model beans must be serializable objects. Maintaining concurrency between data model beans and the entity data store can also be an issue. When using data model beans, you must use a mechanism for synchronizing data model state with the associated entity data store elements to propagate changes in the entity state to the remote data model representations when necessary.

- In what way does a data model bean differ from a typical JavaBean component?

  Answer: Because the bean fields are read-only, they do not require mutator methods on the bean.

- Why must a data model bean implement the `java.io.Serializable` interface?

  Answer: From the Java documentation, "Object serialization supports the encoding of objects, and the objects reachable from them, into a stream of bytes, and it supports the complementary reconstruction of the object graph from the stream. Serialization is used for lightweight persistence and for communication using sockets or Remote Method Invocation (RMI)."

Filename:	lab01.doc
Directory:	H:\FJ310_revC_0301\BOOK\EXERCISE\Exercise_Originals
Template:	C:\Program Files\Microsoft Office\Templates\Normal.dot
Title:	Case Study: The New User Interface
Subject:	
Author:	jdibble
Keywords:	
Comments:	
Creation Date:	04/24/01 10:05 AM
Change Number:	4
Last Saved On:	04/24/01 12:35 PM
Last Saved By:	Nancy Clarke
Total Editing Time:	2 Minutes
Last Printed On:	04/24/01 12:35 PM

As of Last Complete Printing

Number of Pages:	11
Number of Words:	2,006 (approx.)
Number of Characters:	11,436 (approx.)