We’ll store the information in two tables:

### BankCustomer

<table>
<thead>
<tr>
<th>Customer_Number</th>
<th>PIN</th>
<th>Checking_Account_Number</th>
<th>Savings_Account_Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

### Account

<table>
<thead>
<tr>
<th>Account_Number</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>DECIMAL(10, 2)</td>
</tr>
</tbody>
</table>

The `Bank` class now needs to connect to the database whenever it is asked to find a customer. The method that finds a customer makes a query

```
SELECT * FROM BankCustomer WHERE Customer_Number = . . .
```

It then checks that the PIN matches, and it constructs a `Customer` object. This method also turns the row-and-column information of the database into object-oriented data:

```java
public Customer findCustomer(int customerNumber, int pin)
   throws SQLException
{
    Connection conn = SimpleDataSource.getConnection();
    try
    {
      Customer c = null;
      PreparedStatement stat = conn.prepareStatement(
         "SELECT * FROM BankCustomer WHERE Customer_Number = ?");
      stat.setInt(1, customerNumber);

      ResultSet result = stat.executeQuery();
      if (result.next() && pin == result.getInt("PIN"))
      {
        c = new Customer(customerNumber,
            result.getInt("Checking_Account_Number"),
            result.getInt("Savings_Account_Number"));
      }
    return c;
    }
    finally
    {
      conn.close();
    }
}
```

Note that the method throws a `SQLException`. Why don’t we catch that exception and return `null` if an exception occurs? There are many potential reasons for a SQL exception, and the
Bank class doesn’t want to hide the exception details. But the Bank class also doesn’t know anything about the user interface of the application, so it can’t display information about the exception to the user. By throwing the exception to the caller, the information can reach the part of the program that interacts with the user.

The BankAccount class in this program is quite different from the implementation you have seen throughout the book. Now we do not store the balance of the bank account in the object; instead, we look it up from the database:

```java
public double getBalance() throws SQLException
{
    Connection conn = SimpleDataSource.getConnection();
    try
    {
        double balance = 0;
        PreparedStatement stat = conn.prepareStatement(
            "SELECT Balance FROM Account WHERE Account_Number = ?";
        stat.setInt(1, accountNumber);
        ResultSet result = stat.executeQuery();
        if (result.next())
        {
            balance = result.getDouble(1);
        }
        return balance;
    }
    finally
    {
        conn.close();
    }
}
```

The deposit and withdraw operations immediately update the database as well:

```java
public void deposit(double amount)
    throws SQLException
{
    Connection conn = SimpleDataSource.getConnection();
    try
    {
        PreparedStatement stat = conn.prepareStatement(
            "UPDATE Account"
            + " SET Balance = Balance + ?"
            + " WHERE Account_Number = ?";
        stat.setDouble(1, amount);
        stat.setInt(2, accountNumber);
        stat.executeUpdate();
    }
    finally
    {
        conn.close();
    }
}
```

It seems somewhat inefficient to connect to the database whenever the bank balance is accessed, but it is much safer than storing it in an object. Suppose you have two instances of the ATM program running at the same time. Then it is possible that both programs modify the same bank account. If each of them copied the bank balances from the database into objects, then the modifications made by one user would not be seen by the other.

You can try out this simultaneous access yourself, simply by running two instances of the ATM simulation. Alternatively, you can modify the main method of the ATMViewer class to pop up two ATM frames.
The source code for the modified ATM application follows. The source code for the ATM and ATMSimulator/ATMFrame classes is only changed minimally, by adding code to deal with the SQLException. The Customer class is unchanged. We do not list those classes, but you will find them in the worked_example_1 folder of this chapter’s companion code.

**worked_example_5/Bank.java**

```java
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.PreparedStatement;
import java.sql.SQLException;

/**
 * A bank consisting of multiple bank accounts.
 */
public class Bank {
    /**
     * Finds a customer with a given number and PIN.
     * @param customerNumber the customer number
     * @param pin the personal identification number
     * @return the matching customer, or null if none found
     */
    public Customer findCustomer(int customerNumber, int pin) throws SQLException {
        Connection conn = SimpleDataSource.getConnection();
        try {
            Customer c = null;
            PreparedStatement stat = conn.prepareStatement(
                "SELECT * FROM BankCustomer WHERE Customer_Number = ?\n            ");
            stat.setInt(1, customerNumber);
            ResultSet result = stat.executeQuery();
            if (result.next() && pin == result.getInt("PIN")) {
                c = new Customer(customerNumber,
                    result.getInt("Checking_Account_Number"),
                    result.getInt("Savings_Account_Number"));
            }
            return c;
        } finally {
            conn.close();
        }
    }
}
```

**worked_example_5/BankAccount.java**

```java
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.PreparedStatement;
import java.sql.SQLException;

/**
 * 
 */
```
A bank account has a balance that can be changed by deposits and withdrawals.

```java
public class BankAccount {
    private int accountNumber;

    // Constructs a bank account with a given balance.
    // @param anAccountNumber the account number
    public BankAccount(int anAccountNumber) {
        accountNumber = anAccountNumber;
    }

    // Deposits money into a bank account.
    // @param amount the amount to deposit
    public void deposit(double amount) throws SQLException {
        Connection conn = SimpleDataSource.getConnection();
        try {
            PreparedStatement stat = conn.prepareStatement(
                "UPDATE Account
                SET Balance = Balance + ?
                WHERE Account_Number = ?");
            stat.setDouble(1, amount);
            stat.setInt(2, accountNumber);
            stat.executeUpdate();
        } finally {
            conn.close();
        }
    }

    // Withdraws money from a bank account.
    // @param amount the amount to withdraw
    public void withdraw(double amount) throws SQLException {
        Connection conn = SimpleDataSource.getConnection();
        try {
            PreparedStatement stat = conn.prepareStatement(
                "UPDATE Account
                SET Balance = Balance - ?
                WHERE Account_Number = ?");
            stat.setDouble(1, amount);
            stat.setInt(2, accountNumber);
            stat.executeUpdate();
        } finally {
            conn.close();
        }
    }
}
```
```java
    conn.close();
    }
}

/**
   * Gets the balance of a bank account.
   * @return the account balance
   */
    public double getBalance()
            throws SQLException
    {
        Connection conn = SimpleDataSource.getConnection();
        try
        {
            double balance = 0
            PreparedStatement stat = conn.prepareStatement(
                    "SELECT Balance FROM Account WHERE Account_Number = ?");
            stat.setInt(1, accountNumber);
            ResultSet result = stat.executeQuery();
            if (result.next())
            {
                balance = result.getDouble(1);
            }
            return balance;
        }
        finally
        {
            conn.close();
        }
    }
```