

# Build a dynamic Derby application

Skill Level: Intermediate

[Tyler Anderson \(tyleranderson5@yahoo.com\)](mailto:tyleranderson5@yahoo.com)  
Freelance Writer

[Christopher Judd \(cjudd@juddsolutions.com\)](mailto:cjudd@juddsolutions.com)  
President  
Judd Solutions, LLC

04 Oct 2005

This tutorial shows you how to build a dynamic Java™ analysis application that connects to Apache Derby. Explore the dynamic way the database stores new application logic, changing the logic of the database without touching the core client program. And learn about Java archive (JAR) signing and how to provide security for the application that uses hot-swappable .jar files.

## Section 1. Before you start

If you're a developer interested in creating a Java analysis application that uses Derby to modify functions without modifying the core application code, this tutorial is for you. It's also for those interested in securing the Derby database in a signed .jar file.

### About this tutorial

In this tutorial, you'll build a dynamic Derby application and learn more about the dynamic abilities of the Java language through a Java analysis program that determines insurability of automobile drivers. The key concepts of this tutorial are creating dynamic functions in Derby and then calling these functions in your Java application.

As soon as you're able to store application logic (using functions) in your database,

you'll discover how to modify the application logic, or the functions that your application uses, by modifying the .jar file. Modifying the functions allows your program to produce different results, without you having to modify the core application code. The ability to do this is a huge advantage for code maintenance.

Because this process requires working with a database in a .jar file, security is crucial. Toward the end of this tutorial, you'll create a signed .jar file and verify that it's genuine and hasn't been tampered with, assuring your content is safe.

The following topics are covered in this tutorial:

- Application overview, Derby refresher, and setup
- Analysis application
- Storing logic in Derby
- Changing the application logic
- Looking out for tampering

## Prerequisites

This tutorial assumes you're familiar with basic programming and Java concepts, including classes, methods, and variables. Knowledge of Java GUI programming is helpful, but not required. For more information, see the [Resources](#) section at the end of this tutorial.

## System requirements

To follow along with this tutorial, you'll need the following tools (see the [Resources](#) section for links to these downloads):

- **Database.** Obviously, you need Derby. Download Derby from Apache and the IBM DB2® JDBC Universal Driver, making sure you've set your class path appropriately by following the installation instructions.
- **Java technology.** Derby requires the use of Java code. Working on a Linux® box running Red Hat Fedora Core, the gcj compiler provided in the distribution was insufficient. Download the Java 2 Platform, Standard Edition (J2SE) Development Kit.
- **GUI front end.** The GUI front end isn't required. It is, however, a recommended download for your experience throughout this tutorial. Download the [GUI and other miscellaneous application code](#).

---

## Section 2. Overview and setup

This section introduces you to the analysis application and gives you instructions about how to set up the database and the directory structure.

### Application overview

The Java analysis program that you are creating in this tutorial determines the insurability of automobile drivers. The application takes the following elements as input: product type, age, gender, number of traffic violations, and credit score. These parameters are then used to calculate a score that determines whether this data qualifies the individual for auto insurance.

Each product uses a different method and class for calculating this score, and that's where you get into the dynamic abilities of Derby. After you've completed the application without Derby, you'll use Derby to store the application logic that your application is currently using. The application logic is then removed from your code, allowing your application to communicate with Derby to determine whether or not an individual qualifies for auto insurance.

After you can communicate with Derby to determine who qualifies for insurance, you learn how to swap out or point Derby to different application logic without modifying the core application code. This capability makes the maintenance of such an application simpler and controlled.

### Review Derby concepts

Derby was created in Java, giving it all the advantages of Java technology. This tutorial focuses on its dynamic class-loading abilities and its ability to create .jar files. The dynamic class-loading abilities enable you to modify application logic without modifying application code. Creating a JAR allows you to easily transport your database from one location to another, and signed .jar files allow you to transport your database securely.

### Create the Derby database

Before you get started, create the database. Enter the following in a new console window:

```
java org.apache.derby.tools.ij
```

This command takes you to the `ij` prompt. Next, communicate to Derby that you want to create a new database by entering:

```
connect 'jdbc:derby:DYNDERBY;create=true';
```

This step creates and connects to the database. You also need to create the table that will contain the products used in the database. Do this by entering the following next:

```
create table product (  
  id varchar(100),  
  name varchar(100) );
```

Now that you've created the table, insert the two auto insurance products you'll query throughout the tutorial by entering:

```
insert into product (id, name) values ('BAD', 'Best American Driver');  
insert into product (id, name) values ('NBP', 'Next Best Product');
```

Your database is set up, complete with products. Now you can begin to set up the directory structure.

## Set up the directory structure

The directory structure used by the GUI code in this tutorial is close to the directory structure you'll use for the remainder of the application. In the JAR containing the GUI Java files, you should see two files in the `/com/ibm/insurance/client` directory: `Application.java` (the main file) and `Frame.java` (the actual GUI front-end file). These two files are packaged at `com.ibm.insurance.client`. Another file, `DriverServiceDeligate.java`, is also contained in the `/com/ibm/insurance/` directory. This file, and the remaining files in this tutorial, are packaged in the `com.ibm.insurance` directory.

You'll have two identical directory trees -- one for the client Java files and another for the database application logic files. Create two directories: `application` and `client`. Place the two GUI files in the `/client/com/ibm/insurance/client` directory and the `DriverServiceDeligate.java` file in `/client/com/ibm/insurance/` directory. Create a `/application/com/ibm/insurance` directory structure in the `application` directory. Most of the other files that you create for this tutorial will go in this `application` directory.

## Section 3. Build the basic application

This section explains the basic application and shows you how to build it.

### Overview

As the directory structure implies, the core application files are packaged at `com.ibm.insurance`. This application deals with automobile drivers, and drivers are processed by any child of the abstract `InsurableDriverAlgorithm` class. The GUI of your program sends the parameters that make up a driver to the `DriverServiceDelegate` class, which, in turn, passes the information to the `BasicInsurableDriverAlgorithm` or the `ComplexInsurableDriverAlgorithm` classes for processing. As soon as the parameters are processed, the result is returned to the GUI and displayed for the user to see. Possible outcomes range from below 0 to 10.

### Using the Driver class

The `Driver` class houses each driver's information. Place the code, as shown in [Listing 1](#), in the `Driver.java` file. Save it in the `/application/com/ibm/insurance` directory.

#### Listing 1. The Driver class

```
package com.ibm.insurance;

public class Driver {

    public static final String MALE = "M";
    public static final String FEMALE = "F";

    int age;

    String gender;

    int violations;

    int creditScore; // range from 300 to 900

    public Driver(int age, String gender,
                 int violations, int creditScore) {
        super();
        this.age = age;
        this.gender = gender;
        this.violations = violations;
        this.creditScore = creditScore;
    }

    public int getAge() {
```

```
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public int getCreditScore() {
        return creditScore;
    }

    public void setCreditScore(int creditScore) {
        this.creditScore = creditScore;
    }

    public String getGender() {
        return gender;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public int getViolations() {
        return violations;
    }

    public void setViolations(int violations) {
        this.violations = violations;
    }
}
```

A driver has four parameters entered into the GUI:

- Driver's gender: M for male or F for female. This parameter has no effect on the result from the algorithms.
- Driver's age: Notice that age affects the algorithms, because no company wants to insure those people who aren't eligible for a driver's license.
- Number of traffic violations: If a driver has too many traffic violations, this will affect their insurability and raise their premiums.
- Credit score: Drivers with low credit scores may not always be able to pay their insurance on time, increasing risk to an insurance company.

## Using InsurableDriverAlgorithm interface

The `InsurableDriverAlgorithm` interface is used by `InsurableDriverAlgorithm` classes. This interface contains a single method, as shown in [Listing 2](#). Place the code in the `InsurableDriverAlgorithm.java` file, and save it in the `/application/com/ibm/insurance` directory.

### Listing 2. The `InsurableDriverAlgorithm` interface.

```
package com.ibm.insurance;
```

```
public interface InsurableDriverAlgorithm {
    public int insurable(Driver driver);
}
```

When implemented by a `InsurableDriverAlgorithm` class, the code in [Listing 2](#) forces each analyzer class to implement this method. This allows you to create multiple analyzer classes that can be called from a single point in the program.

## Using BasicInsurableDriverAlgorithm class

This class contains an algorithm for computing the insurability of a driver. Create a `BasicInsurableDriverAlgorithm.java` file, and place the following code in it, as shown in [Listing 3](#). Save this file in the `/application/com/ibm/insurance` directory, along with the `InsurableDriverAlgorithm.java` file.

### Listing 3. The BasicInsurableDriverAlgorithm class

```
package com.ibm.insurance;

public class BasicInsurableDriverAlgorithm
    implements InsurableDriverAlgorithm {

    public int insurable(Driver driver) {
        if(driver.getAge() > 16 &&& driver.getViolations() < 10 &&&
            driver.getCreditScore() > 700) {
            return 0;
        }
        return 10;
    }
}
```

If a driver is older than 16 years of age, has fewer than 10 traffic violations, and has a credit score greater than 700, then their insurability is determined to be 8. Otherwise, it is 0. This is a good algorithm for certain insurance products that require superior drivers. So, to insure only excellent drivers, the constraints are made more stringent.

## Using ComplexInsurableDriverAlgorithm class

Here is a more complex class for computing how insurable a driver is. Create another file, `ComplexInsurableDriverAlgorithm.java`, and add the following code, as shown in [Listing 4](#). Place this file in the same directory as the `InsurableDriverAlgorithm.java` file.

### Listing 4. The ComplexInsurableDriverAlgorithm class

```
package com.ibm.insurance;
```

```
/**
 * Simple implementation of an insurable driver.
 */
public class ComplexInsurableDriverAlgorithm implements InsurableDriverAlgorithm {
    /**
     * @see com.ibm.insurance.InsurableDriverAlgorithm
     */
    public int insurable(Driver driver) {
        int result = 10;
        if(driver.getAge() >= 25){
            result -= 4;
        }
        else if(driver.getAge() >= 21){
            result -= 3;
        }
        else if(driver.getAge() >= 18){
            result -= 2;
        }
        else if(driver.getAge() >= 16){
            result -= 1;
        }
        else if(driver.getAge() < 16){
            result += 10;
        }

        if(driver.getViolations() > 10){
            result += 2;
        }
        else if(driver.getViolations() >= 5){
            result -= 1;
        }
        else if(driver.getViolations() >= 0){
            result -= 2;
        }

        if(driver.getCreditScore() >= 700){
            result -= 4;
        }
        else if(driver.getCreditScore() >= 600){
            result -= 2;
        }
        else if(driver.getCreditScore() >= 500){
            result += 2;
        }
        else if(driver.getCreditScore() < 500){
            result += 10;
        }
        return result;
    }
}
```

This class has more intervals, or buckets, that characteristics of a driver can fall under, widening the possible values of insurability. This means that more attributes must be strong to get a good score. If a driver is poor in one area, it can destroy his or her chances of obtaining auto insurance.

## Using DriverServiceDelegate class

This class has a method that is called by the GUI, passing in the parameters of a driver. Modify the DriverServiceDelegate.java file, and place it in the /client/com/ibm/insurance/ directory. Notice that this is the only file (other than the

two GUI files) that go in the /client directory structure. (See [Listing 5](#).)

### Listing 5. The DriverServiceDelegate class

```
package com.ibm.insurance;

public class DriverServiceDelegate {
    public static String isInsurable(String product, int age,
                                    String gender, int violations,
                                    int creditScore) throws Exception{
        int result;
        InsurableDriverAlgorithm algorithm;

        if(product == "Next Best Product")
            algorithm = new ComplexInsurableDriverAlgorithm();
        else
            algorithm = new BasicInsurableDriverAlgorithm();
        result = algorithm.insurable(new Driver (age, gender,
                                                violations,
                                                creditScore));

        if(result > 10) result = 10;
        else if(result < 0) result = 0;

        return Integer.toString(result);
    }
    ...
    public static Object[] getProducts() throws Exception {
    ...
    }
}
```

When you click **Check**, the `isInsurable` function in [Listing 5](#) is called by the GUI. Depending on the selected product, an `InsurableDriverAlgorithm` class is chosen. The appropriate `insurable` function is then called, passing in the driver's attributes. The result is returned and compared against the maximum and minimum values and finally returned as a `String`.

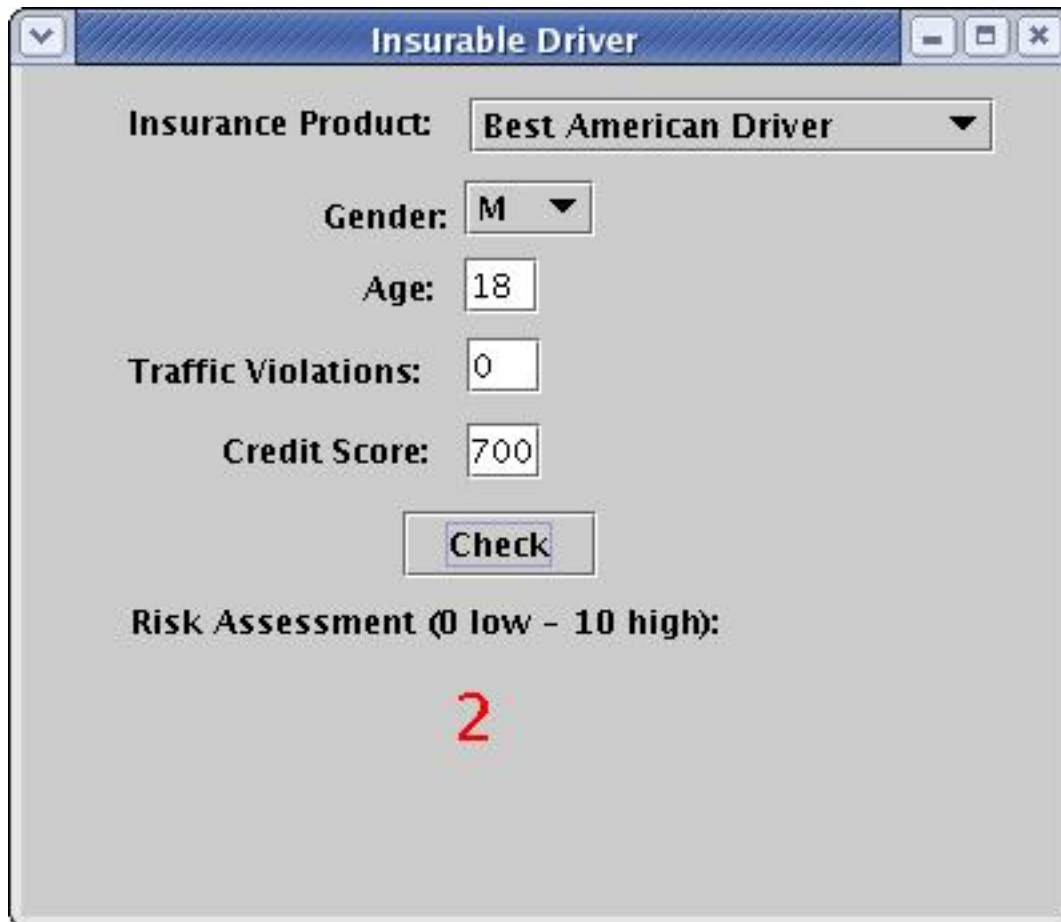
## Run the application

Before running your application, make sure the /application/ and /client/ directories are in your class path. Next, test your application by entering:

```
java com.ibm.insurance.client.Application
```

The GUI should display as shown in [Figure 1](#).

### Figure 1. The GUI of the application



Insurance Product: Best American Driver

Gender: M

Age: 18

Traffic Violations: 0

Credit Score: 700

Check

Risk Assessment (0 low - 10 high):

2

Familiarize yourself with the application and the values that are supposed to emerge by following the values you entered into what's being sent in to the `InsurableDriverAlgorithm.insurable` functions.

---

## Section 4. Store logic in Derby

This section demonstrates how to store application logic within Derby and how to test the logic within Derby. You'll verify that a needed class path exists for your application and find out how to connect to Derby and execute the application logic stored there.

### Using the `DriverService` class

This class is the main class called from the database. Create a file,

DriverService.java, and save it in the /application/com/ibm/insurance/ directory. (See [Listing 6.](#))

This class contains the function that is to be called from the database, the `isInsurable` function. This function first packages the driver's attributes into the `Driver` class and determines which product is chosen. It then calls the corresponding `InsurableDriverAlgorithm.insurable` function.

### Listing 6. The DriverClass class

```
package com.ibm.insurance;

public class DriverService {

    public static int isInsurable(String product, int age,
        String gender, int violations, int creditScore)
    throws Exception {

        Driver driver = new Driver(age, gender,
            violations,
        creditScore);
        InsurableDriverAlgorithm algorithm;

        if(product.indexOf("Next Best Product") != -1)
            algorithm = new BasicInsurableDriverAlgorithm();
        else
            algorithm = new ComplexInsurableDriverAlgorithm();
        return algorithm.insurable(driver);
    }
}
```

## Create a JAR of the application logic

At this point, you need to place some of your classes in the database. Here are the files you must add to a .jar file and store in your database:

- Driver
- DriverService
- InsurableDriverAlgorithm
- BasicInsurableDriverAlgorithm
- ComplexInsurableDriverAlgorithm

After completing this, only the preceding list of classes should be in the /application/com/ibm/insurance/ directory.

Navigate to the /application/ directory and enter:

```
jar -cvf algorithm.jar com
```

This command creates a JAR containing your application logic. Next, you need to add this logic into the database.

## Add application logic JAR to the database

Return to the `ij` prompt. This is where you add the application logic JAR to your database. Move the `algorithm.jar` file to the same directory as your database, if it isn't there already. Enter the following:

```
CALL sqlj.install_jar('algorithm.jar', 'APP.algorithm', 0);
```

This command adds the `algorithm.jar` file into your database, allowing you to use the application logic. Next you need to set the `derby.database.classpath` property:

```
CALL SYCS_UTIL.SYCS_SET_DATABASE_PROPERTY(  
    'derby.database.classpath',  
    'APP.algorithm');
```

When your database loads, this step also causes it to load your application logic.

## Create a function in the database

Creating a function in Derby is similar to creating a table. Using the `ij` prompt, enter the following to create the function (as shown in [Listing 7](#)).

### Listing 7. Creating a function in a Derby database

```
create function isInsurable(product varchar(256),  
    age integer, gender varchar(1),  
    violations integer,  
    creditScore integer) returns integer  
  
parameter style java  
no sql  
language java  
external name  
    'com.ibm.insurance.DriverService.isInsurable';
```

You have now created a function, `isInsurable`, that will take the driver parameters in SQL and call the `isInsurable` function located in your `DriverService` class.

## Test the functions

The functions now exist in the database, and you should be able to use them in your Java application. Before you do that, however, it's important to test them so that you know they exist and are functioning properly. Again, at the `ij` prompt, enter this

Derby SQL statement:

```
values isInsurable ('Best American Driver', 25, 'M', 13, 600);
```

With this step, you're calling your Java function, `isInsurable`, in the `DriverService` class that will in turn call the `insurable` function in the `ComplexInsurableDriverAlgorithm` class, because that is the function that lines up with the Best American Driver product. The result should display as shown in [Listing 8](#).

### Listing 8. Results of testing `isInsurableBasic`

```
----- 1
6
1 row selected
```

The resulting value is 6, and your function now works. Next, prepare the database for use by your Java application.

## Creating a .jar file of the database

For your application to load the database, you must load it into a .jar file. First, close the `ij` prompt by entering this command:

```
exit;
```

This command exits you out of the `ij` prompt, causing the database to be saved in its directory.

Create another window, and go to the directory containing your new database, which is usually the same directory where you started the `ij` prompt). Enter the following command:

```
jar -cvf insurance-db.jar DYNDERBY
```

The output shows all the files of the database being added into the .jar file. When completed, your database is contained in the `insurance-db.jar` file.

## Verify that Derby is in the class path in Java code

Your application requires that the Derby database JARs be in your class path. The following code, as shown in [Listing 9](#), verifies that the necessary Derby JAR is in

your class path. The following code will be added to your `DriverServiceDelegate.java` file:

### Listing 9. Verifying the necessary Derby JAR is in the Classpath

```
public class DriverServiceDelegate {  
    ...  
    static {  
        try {  
            Class.forName("org.apache.derby.jdbc.EmbeddedDriver");  
        } catch (ClassNotFoundException e) {  
            e.printStackTrace();  
            throw new RuntimeException(e);  
        }  
    }  
    public static String isInsurable(String product, int age,  
    ...
```

The statements within the `static{ }` statement are only executed once after loading the class. The `Class.forName` function loads the class, and if it isn't found in the class path, then an exception is thrown, ending code execution with the `throw new RuntimeException` statement. Your application is now ready to connect to the database.

## Connect to the database inside your application

Connecting to the database allows you to access the application logic. First set up the connection URL, as shown in [Listing 10](#).

### Listing 10. The connection URL

```
public class DriverServiceDelegate {  
    ...  
    private static final String url =  
        "jdbc:derby:jar:(//home/tfunk/another-db.jar)DYNDERBY";  
    static {  
        try {  
            ...  
        }  
    }  
    ...
```

This URL allows you to connect to the database. The code in [Listing 11](#) begins to modify the entire `isInsurable` method. You need to do this because you will no longer call the `InsurableDriverAlgorithm` classes directly, but through the database instead.

### Listing 11. Connecting to the database

```
package com.ibm.insurance;  
  
import java.sql.Connection;
```

```
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
...
    public static String isInsurable(String product, int age,
                                    String gender, int violations,
                                    int creditScore)
                                    throws Exception{

        int result = 10;
        Connection connection = null;
        PreparedStatement statement = null;
        ResultSet resultSet = null;

        try{
            connection = DriverManager.getConnection(url);
...

```

To use the `java.sql` extensions, they first need to be imported using the import statements in [Listing 11](#). Java automatically connects to your database on the last line of the code. You have to first set up the connection, and then the database is ready to accept SQL statements.

## Prepare the SQL statement

You can write SQL statements using the `java.sql` extensions. First you need to know which function you want to call within the Derby database of the functions you created earlier in this section. This time you'll have to modify the code you had before, because you're now connecting to the database and using the database to call the functions (rather than your application code calling the functions directly). (See [Listing 12](#).)

### Listing 12. Preparing the SQL statement

```
...
try{
    connection = DriverManager.getConnection(url);

    statement = connection
        .prepareStatement
        ("values(isInsurable(?, ?, ?, ?, ?))");
    statement.setString(1, product);          statement.setInt(2, age);
    statement.setString(3, gender);
    statement.setInt(4, violations);
    statement.setInt(5, creditScore);
...

```

The algorithm variable is now a `String`, allowing you to reconfigure which function you call in the database. You also need to prepare the statement through the connection with the `connection.prepareStatement` function. Next, each variable in the statement variable is set using `setInt` and `setString` functions. The query is ready to be executed.

## Executing the query and results

With the variables you set up earlier, the `PreparedStatement` statement variable is ready to be executed. Execute the query, as shown in [Listing 13](#).

### Listing 13. Executing the query and obtaining the results

```
...
    statement.setInt(currVar++, creditScore);

    resultSet = statement.executeQuery();
    if(resultSet.next())
        result = resultSet.getInt(1);
} finally {}
...

```

After executing the query and retrieving the `resultSet`, check to make sure that the `resultSet` variable contains rows. If so, the result will be saved in the `result` variable.

## Close the database and return the result

Close the database and return the result to the GUI front end. Do this by adding the following code, as shown in [Listing 14](#). This completes the modifications to the `DriverServiceDelegate.java` file.

### Listing 14. Closing the database and returning the result

```
    } finally {} ...

    if(resultSet != null)
        resultSet.close();
    if(statement != null)
        statement.close();
    if(connection != null)
        connection.close();

    if(result > 10) result = 10;
    else if(result < 0) result = 0;

    return Integer.toString(result);
}
}

```

You have just cleaned up the database by closing it. The result will now be clipped, as mentioned earlier. If the insurability of the driver is less than 0 (low risk), clip the value to 0. If it's greater than 10 (high risk), clip the value to 10.


## Run the application and compare results

This time when you run the application, make sure the /client/ directory is in your class path and that the /application/ directory is not in your class path, which will force Derby to load the class files in the database. Test your application by entering the following:

```
java com.ibm.insurance.client.Application
```

The GUI should display as shown in [Figure 2](#).

**Figure 2. Displays the GUI of the application using logic stored in the database**



The screenshot shows a Java Swing window titled "Insurable Driver". The window contains the following elements:

- Insurance Product:** A dropdown menu with "Next Best Product" selected.
- Gender:** A dropdown menu with "M" selected.
- Age:** A text input field containing "25".
- Traffic Violations:** A text input field containing "0".
- Credit Score:** A text input field containing "0".
- Check:** A button with the text "Check".
- Risk Assessment (0 low - 10 high):** A label followed by the number "10" in a large, bold, red font.

The results should be the same, but how you arrived at them is different. In the next section, you'll change the logic you just added to your database.

## Section 5. Change the data file

This section covers creating a new `InsurableDriverAlgorithm` class that you'll add to the database using a `.jar` file. In this section, you replace the current `.jar` file in the database with a new one (similar to the initial installation you did in the previous section), effectively changing the application logic of the program without touching the core client files.

### Using `Over25InsurableDriverAlgorithm` class

This class will only qualify drivers 25 years of age and older. Create a `Over25InsurableDriverAlgorithm.java` file, and add the following code, as shown in [Listing 15](#). Place this file in the same directory as the other `InsurableDriverAlgorithm` Java files.

This class is straightforward. If a driver is 25 years of age or older, they are qualified for insurance. Now you can modify the database.

#### Listing 15. The `Over25InsurableDriverAlgorithm` class

```
package com.ibm.insurance;

public class Over25InsurableDriverAlgorithm implements
InsurableDriverAlgorithm {
    public int insurable(Driver driver) {
        if(driver.getAge() >= 25){
            return 0;
        }
        return 10;
    }
}
```

### Modify the `DriverService` class

Modify the `DriverService` class to use your new function. Change the function, as shown in [Listing 16](#).

#### Listing 16. The `Over25InsurableDriverAlgorithm` class

```
...
if(product == "Next Best Product")
    algorithm = new Over25InsurableDriverAlgorithm();
else
    algorithm = new ComplexInsurableDriverAlgorithm();
...
```

Now when the Next Best Product is chosen in your application, only those drivers 25 years of age and older will qualify.

## Create another JAR of the application logic

In the last section you created a JAR of the application logic. Do that again here. Go to `/application/`, and enter:

```
jar -cvf algorithm.jar com
```

This step picks up the new `Over25InsurableDriverAlgorithm` class and the modified `DriverService` class. Move the `algorithm.jar` file to the directory of your database, if you haven't already.

## Replace the application logic of the database

Now that the new JAR is ready, it's time to replace the old JAR in the database with the new one that you just created. Go to the `ij` window, reopen the `ij` prompt, and connect to the database again using the same statements as before. Enter the following:

```
CALL sqlj.replace_jar('algorithm.jar', 'APP.algorithm');
```

This replaces the application logic within the database with the new one containing a new class and a modified `DriverService` class that calls the new class, effectively changing the application logic without touching the core database.

## Create another read-only JAR of the database

Next, you need to create another `.jar` file containing your database, as before, that has the new updates. This swaps out the other database you were using; or you can rename the old one before creating this one. Go to the directory containing your database, and after exiting out of the `ij` prompt, enter the following :

```
jar -cvf insurance-db.jar DYN Derby
```

Load your application, and see the new output of your application for the Next Best Product selection.


## Rerun the application and compare results

Again, make sure the /client/ directory is in your class path and that the /application/ directory is not in your class path, which forces Derby to load the class files in the database. Run your application:

```
java com.ibm.insurance.client.Application
```

The GUI should display as shown in [Figure 3](#).

**Figure 3. Displays the GUI of the application using a hot-swapped .jar file**



The screenshot shows a Java Swing window titled "Insurable Driver". The window contains the following elements:

- Insurance Product:** A dropdown menu currently showing "Next Best Product".
- Gender:** A dropdown menu currently showing "M".
- Age:** A text input field containing the number "25".
- Traffic Violations:** A text input field containing the number "0".
- Credit Score:** A text input field containing the number "0".
- Check:** A button labeled "Check".
- Risk Assessment (0 low - 10 high):** A label followed by a large red "0" indicating the result of the check.

The results should be the same for the Best American Driver product. For the Next Best Product, you should only get approvals for drivers 25 and over, no matter how good the other data points are.

---

## Section 6. Check for tampering

This section covers the security of .jar files, an important issue because your application involves hot-swapping them with other .jar files as you change your application logic in the database. Encrypting the database in Derby is another route you can take, but this tutorial focuses on securing the .jar files.

## Create a KeyStore file

This is the first step in signing your JAR. To create a KeyStore file named tylerKeyStore, enter the following command:

```
keytool -genkey -alias tyler -keypass tylerpass
        -validity 80 -keystore tylerKeyStore
        -storepass tylerKeyStorePass
```

The keytool asks you a series of questions and then stores the KeyStore file in tylerKeyStore.

Use the generated file to sign the .jar files of your database.

## Sign the database .jar file

You can secure your .jar file now with the jarsigner tool. Go to the directory containing your insurance-db.jar file, and move your KeyStore file to that directory as well. Enter the following to sign your .jar file:

```
jarsigner -keystore tylerKeyStore -storepass tylerKeyStorePass
        -keypass tylerpass insurance-db.jar tyler
```

The .jar file is now signed, allowing your application to pick up on unauthorized occurrences that might happen to your .jar file.

## Use the .jar utility file to verify signed .jar files

The JarUtil class located in the /com/ibm/insurance/util/ directory is used to verify the origin of your database. This section covers some of its key features in determining the authenticity of a signed .jar file.

The code in [Listing 17](#) verifies that the JAR has been signed and that the KeyStore is located in the .jar file.

### Listing 17. Verifying the KeyStore exists and that the .jar file is signed

```
if (manifest != null && inManifest == true) {
```

```
Certificate[] certs = jarEntry.getCertificates();
boolean isSigned = ((certs != null) && (certs.length > 0));
boolean inKeyStore = findInKeyStore(certs, keyStore);

if ((verifySigned && !isSigned)
    || (verifyInKeyStore && !inKeyStore)) {
    isVerified = false;
    break;
} else {
    isVerified = true;
}
}
```

These two entries in the manifest file must be there for your .jar file to be uncompromised. The code simply checks to see whether the two entries exist and that they have valid values.

Opening the .jar file is the key way to determine if a signed JAR has been compromised:

```
jarFile = new JarFile(file, true);
```

If the JAR is signed and unauthentic (that is, the file in the JAR is modified prior to the signing), this method will throw an exception and halt code execution.

This next function is called on each entry in the JAR's manifest file and includes every file in the JAR. [Listing 18](#) shows you how the file for the current entry is verified.

### Listing 18. Verifying the file has not been tampered with

```
private static void verifyDigestsImplicitly(JarFile jar,
                                           JarEntry entry)
    throws SecurityException, IOException {
    InputStream is = jar.getInputStream(entry);
    byte[] buffer = new byte[8192];

    while ((is.read(buffer, 0, buffer.length)) != -1) { }
    is.close();
}
```

This method ensures that the entry it is checking has not been tampered with simply by reading it. If something is wrong, an exception is thrown. You'll see how these work next when you experiment with using compromised or unsigned .jar files.

## Verify the origin of a database JAR in the application

You should modify the `DriverServiceDelegate.java` file to validate the authenticity of the .jar files that it uses. (This code for verifying signed JARs was adapted from a post by Robert Paris; see [Resources](#) for a link to the post.) [Listing 19](#) shows you how to set up the application for verification.

## Listing 19. Setting up the application to verify the authenticity of .jar files

```
import com.ibm.insurance.util.JarUtil;

public class DriverServiceDelegate {

    private static final String keyStoreFileName =
        "/home/tyfunk/tylerKeyStore";
    private static final String keyStorePass =
        "tylerKeyStorePass";
    private static final String pathToJar =
        "/home/tyfunk/insurance-db.jar";
    private static final String url =
        "jdbc:derby:jar:(/" + pathToJar + ")DYNDERBY";

    static {
        boolean result;
        try {
            result =
                JarUtil.verifyJar(pathToJar, keyStoreFileName,
                                keyStorePass);

            if(!result)
                throw new
                    RuntimeException("Jar file failed verification");
            Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
        } catch (ClassNotFoundException e) {
            ...
        }
    }
}
```

Start by declaring where your KeyStore, database JAR, and your KeyStore password reside. Next, modify the connection URL to contain the `pathToJar` variable. Then save the result of verifying the JAR. If the result is `false`, an exception is thrown, halting execution.

This is all you need to do to have your application verify the origins of .jar files. Run your application, and notice how it opens without any problems. If there's a problem, not even the GUI will appear, and you'll see the stack trace as well as the type of reason for the thrown exception.

## Unsigned JAR fails verification

Pass one of the earlier unsigned JARs of your database, created earlier, or create another one without signing it, and run your application with it. The following is the example output of the failure:

```
Java.lang.RuntimeException: Jar file failed verification
    At com.ibm.insurance.util.JarUtil.verifyJar(JarUtil.java:23)
    ...
```

No unsigned JARs can be used with your application. Now, modify your current signed .jar file.

## Modify a signed .jar file to fail verification

Modify the file by first copying the signed .jar file and then opening one of the files inside the .jar file in a text editor. You can do this by double-clicking on the .jar file using a graphical browser in Linux. You'll see that the JAR verification tool can pick up miniscule changes. Here's the sample output:

### Listing 20. Sample output of JAR verification tool

```
Java.lang.SecurityException: SHA1 digest error for
    DYNDERBY/service.properties
At sun.security.util.ManifestEntryVerifier.
    verify(ManifestEntryVerifier.java:195)
```

The only thing you did to the file was type a few key strokes. If a file exists at signing and it changes, the tool picks up those changes, and the application execution will halt.

---

## Section 7. Summary

In this tutorial, you successfully created a dynamic Derby application that allows you to change the data file, modifying the application logic without having to recompile or touch the core application code. You can now create functions in Derby that are accessible by your Java application. And you can create multiple .jar files with different configurations of the same database. You also learned about an important security element added to the auto insurance application by signing the hot-swappable .jar files.

## Downloads

Description	Name	Size	Download method
Source code for the dynamic auto insurance example	dynamicsource.zip	17 KB	<a href="#">HTTP</a>

[Information about download methods](#)

# Resources

## Learn

- Learn about [securing your database through encryption](#).
- Learn how to [create .jar files](#).
- Get more information about [JAR signing](#).
- Find out about [installing and/or replacing .jar files in the database](#).
- Find information on [creating functions in the Derby database](#).
- Read Sun's [Java manual for the java.sql package](#).
- Learn more about the possibilities of [DB2 and Java technology](#).
- Start at the developerWorks [Apache Derby resource area](#) for all your Derby resources and articles.
- See "[IBM Cloudscape Version 10.1, New release](#)" (developerWorks, August 2005).
- Visit the [developerWorks Open source zone](#) for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.
- Browse all the [Apache articles](#) and free [Apache tutorials](#) available in the developerWorks Open source area.

## Get products and technologies

- Download the following tools to follow along with this tutorial:
  - [Apache Derby](#) and the [IBM DB2 JDBC Universal Driver](#)
  - [J2SE Development Kit \(JDK\)](#)
  - [GUI and other miscellaneous application code](#)
- Visit the [official Apache Derby Web site](#).
- Innovate your next open source development project with [IBM trial software](#), available for download or on DVD.

## Discuss

- [Participate in the discussion forum for this content](#).
- Get involved in the developerWorks community by participating in [developerWorks blogs](#).

## About the authors

### Tyler Anderson

Tyler Anderson graduated with a degree in computer science from Brigham Young University in 2004 and is currently in his final semester as a master-of-science student in computer engineering. In the past, he worked as a database programmer for DPMG.COM, and he's currently an engineer for Stexar Corp. based in Beaverton, Oregon.

---

### Christopher Judd

Christopher Judd is the President and Primary Consultant for [Judd Solutions, LLC](#), international speaker, open source evangelist, [Central Ohio Java Users Group](#) coordinator, and coauthor of *Enterprise Java Development on a Budget* and *Pro Eclipse JST*. He has spent eight years developing software in the insurance, retail, government, manufacturing, service, and transportation industries. His current focus is consulting, mentoring, and training with Java, Java 2 Platform, Enterprise Edition (J2EE), Java 2 Platform, Micro Edition (J2ME), Web services, and related technologies.