# Build a Derby calendar, Part 3: Using transactions and locking

Skill Level: Intermediate

Nicholas Chase (ibmquestions@nicholaschase.com)
Freelance Writer

27 Sep 2005

In this final tutorial of the series, you'll finish the calendar and reminder application using the Java™ language and the Apache Derby database. In Part 1 and Part 2 of this three-part series, you created a basic calendar and reminder application using a Derby database back end and a GUI and a Web-based front end. Now that the proof of concept is complete, you can add a more friendly interface and use transactions and locking to create a truly multiuser system.

## Section 1. Before you start

This tutorial is for developers who need to handle multiple simultaneous users on a single Derby (or IBM Cloudscape) database. Developers who want to learn more about building Java GUI applications will also find it helpful.

### About this tutorial

In Part 1, you connected to the database using Java Database Connectivity (JDBC), thus creating basic entity classes. Part 2 used those classes to create the application, using first a GUI and then a Web-based interface, and examined the different ways of embedding, or incorporating Derby into the application. The single-user embedded application was turned into a multiuser networked application in which concurrency issues were solved by giving each user a unique schema.

While the calendar application in Part 2 allowed for multiple users, it wasn't possible for users to view one another's data. In this tutorial, you'll create this common


© Copyright IBM Corporation 2005. All rights reserved.

calendar by moving everything back into the default schema and controlling concurrency issues using locking and transactions in Derby.

Over the course of this tutorial, you'll learn about:

- How to create a date picker.
- How to easily display a table of information using Java code.
- How to make information in that table editable.
- How to create and use a database transaction.
- Different types of selection anomalies.
- Database isolation levels and how to control them.
- Different types of database locks.
- The effects of scope on database locks.
- Out-of-control lock types.

## Prerequisites

This tutorial covers the building and alteration of a Java GUI application, so you should have at least a basic grasp of the Java language to follow along with that part of the tutorial. You should also have at least a fundamental grasp of how Derby works. You can get the basics in Part 1, "Understanding JDBC" and Part 2, "Embedding options" (developerWorks, September 2005).

## System requirements

Apache Derby is the open source center of the IBM Cloudscape database, and as such it shares many of its features, such as a small footprint and the ability to easily embed it in an application (see Resources for a link to more information about Cloudscape). Derby also provides an easy upgrade path to other databases, such as IBM DB2®.

To follow along with this tutorial, you need to have the following tools:

- **Java 2 Platform, Standard Edition 1.4.2 or higher**. Note that you must have the Java Developer Kit, not just the Java Runtime Environment. Download the Java code from the Sun Web site.
- **The existing application**. This tutorial demonstrates changes to the application created in Part 2. Download those classes.


© Copyright IBM Corporation 2005. All rights reserved.

- **Apache Derby**, which you can download from the Apache Software Foundation.
- **JavaMail**. To send e-mail reminders, you need mail.jar, which is now part of the Java package. You also need activation.jar, downloadable as part of the JavaBeans activation framework. Make sure both files are in your class path.
- **A Web application server**, such as Apache's Jakarta Tomcat. You can use any servlet-compatible application server, but this tutorial was written using Tomcat Version 5.5.
- **JCalendar**. To make life easier, you'll add a calendar picker to the application. Download the required library.
- **A text editor or Integrated Development Environment (IDE)**. You can easily create Derby applications using an IDE, such as Eclipse, but I'm going to assume you're using a simple text editor.

---

# Section 2. Overview

The calendar application has been through several iterations so far. Let's look at where it will ultimately wind up.

## The application so far

At the end of Part 2, the application consisted of three parts:

1. Three base classes: `EventClass`, which handles creating and editing the actual events; `Calendar`, required solely to retrieve an array of events for a particular day; and `Reminder`, which sent e-mail reminders for each event.
2. The GUI application, consisting of the `CalendarFrame` class, which creates an entry form for events and a display form for existing events.
3. The Web-based application, which consists of a single servlet that also contains an entry form, display area, and the ability to edit existing events.

You can download these classes as they were left in Part 2.


© Copyright IBM Corporation 2005. All rights reserved.

## Potential problems for a common event calendar

In Part 2, you took great pains to separate data being entered to prevent concurrency issues. But what does that mean?

Concurrency issues arise when more than one calendar user is dealing with the same day at the same time. In some cases, this doesn't matter. For example, it doesn't matter if one, two, or 137 users look at a single date at the same time as long as none of them tries to change it.

Suppose two users are looking at the same event scheduled for October 28, 2005: an award ceremony for a retiring programmer. Both have the event on their screens at the same time. Bob decides to alter the description to indicate a location change to a quiet restaurant, and Troy decides to change it to an amusement park. When they both try to save the changes, which one wins out? If Bob saves the changes first, Troy's changes will wipe his out, and neither of them will know.

To prevent this from happening, you must use transactions, locking, and isolation levels.

## Transactions, locking, and isolation levels

A transaction is a group of database operations that must all succeed or fail as a group. For example, suppose you have events that take place on two consecutive days, such as a two-day conference. If you move one, you need to move the other as well.

So if you issue an SQL `update` statement to change the first one and it succeeded, and then a second `update` statement to change the second and it failed, you should undo, or roll back, the first one.

A transaction solves this issue by creating a group of actions that must all succeed together.

So when you create a transaction and perform an action, such as inserting data, the action doesn't take effect until you finalize, or commit, the transaction. That's not to say that you won't see the changes. You can always see your changes whether or not you've committed them.

The question is, "Who else can see them?" That's the question that this tutorial explores by delving into locks and isolation levels.

## Changes to the application


© Copyright IBM Corporation 2005. All rights reserved.

In this tutorial, you'll adapt the application in two ways.

The first change addresses a usability issue. The proof of concept for the application required only that events could be added and displayed by multiple users. For added user convenience, easier date management and event editing needs to be added.

The second change involves transaction management. The proof of concept executed all database changes immediately; however, in this case the application sends each change to the database as it's requested, but won't commit those changes until the user specifically requests it. Until then, the user can also choose to roll back the transaction.

The application will also have the ability to simulate multiple user sessions and to specifically choose different locking mechanisms and isolation levels.

## Key differences between GUI and Web

While on the subject of application changes, it's important to understand the differences between the GUI and the Web calendar applications.

In the GUI application, the user initiates a database connection upon starting the application and holds it until the application terminates, which might take a great deal of time. During that time, the user may initiate multiple transactions, which can also take a significant amount of time.

That's not the case in the Web-based application. Web interaction with the database is inherently stateless. The user submits a form to add or edit an event, and the servlet connects, makes the changes, and disconnects, committing the transaction.

Because of this, transaction management becomes less of an issue unless you want to get into complex session management, which is beyond the scope of this tutorial. Instead, the Web application will be left pretty much as is, affected by, but not affecting, other sessions in their locks.

---

# Section 3. Introducing transactions

At the heart of this process is the application's ability to group actions into transactions, so it helps to understand precisely what they are and how they work.

## Inserting data


© Copyright IBM Corporation 2005. All rights reserved.

To begin looking at transactions, open three command windows. In the first, start the database in network mode, as shown in Listing 1.

### Listing 1. Starting the database

```
                    >set DERBY_INSTALL=c:\derby
>set
CLASSPATH=%DERBY_INSTALL%\lib\derby.jar;%DERBY_INSTALL%\lib\derbytools.jar;
>set CLASSPATH=%DERBY_INSTALL%\lib\derbynet.jar;%CLASSPATH%
>java org.apache.derby.drda.NetworkServerControl start -h
0.0.0.0
```

Make sure, of course, to use your own values for `DERBY_INSTALL`. In the other windows, open the `ij` tool, shown in Listing 2.

### Listing 2. Starting the clients

```
                    >set DERBY_INSTALL=c:\derby
>set CLASSPATH=%DERBY_INSTALL%\lib\derby.jar;%DERBY_INSTALL%\lib\derbytools.jar;
>set CLASSPATH=%DERBY_INSTALL%\lib\derbyclient.jar;%CLASSPATH%
>set PATH=C:\derby\frameworks\NetworkServer\bin;%PATH%
>ij
>java -Dij.driver=org.apache.derby.jdbc.ClientDriver
-Dij.protocol=jdbc:derby://localhost:1527/ -Dij.user=APP
-Dij.password=APP  org.apache.derby.tools.ij
ij version 10.1
ij>
```

By default, the tool operates in autocommit mode. In other words, it never creates a transaction because it commits after every operation. For example, in one of the client windows, create a table and insert some data (see Listing 3).

### Listing 3. Inserting data

```
                    ij> connect 'c:\derby\calendar';
ij> create table demo (theText varchar(50), minValue int, maxValue int);
0 rows inserted/updated/deleted
ij> insert into demo (theText, minValue, maxValue) values ('first value',
10, 20);
1 row inserted/updated/deleted
ij> insert into demo (theText, minValue, maxValue) values ('second value',
15, 20);
1 row inserted/updated/deleted
```

Now go to the other window and select the data, as shown in Listing 4.

### Listing 4. Seeing the data

```
                    ij> connect 'c:\derby\calendar';
ij> select * from demo;
THETEXT                                           |MINVALUE   |MAXVALUE
--------------------------------------------------------------------------
first value                                       |10         |20
second value                                      |15         |20
```


© Copyright IBM Corporation 2005. All rights reserved.

That was with autocommit on, so the database performs a commit after each statement.

Now let's see what happens when autocommit is off and the database doesn't commit after each statement.

## Creating a transaction

Derby enables you to turn off the autocommit mode, so you can see what happens when you create a transaction. In the first window, turn off autocommit, and insert a new record, as in Listing 5.

### Listing 5. Turning off autocommit

```
                    ij> autocommit off;
ij> insert into demo (theText, minValue, maxValue) values ('third value',
100, 200);
1 row inserted/updated/deleted
```

Now try selecting the data again in the other window, as in Listing 6.

### Listing 6. Selecting uncommitted data

```
                    ij> select * from demo;
THETEXT                                           |MINVALUE    |MAXVALUE
--------------------------------------------------------------------------
ERROR 40XL1: A lock could not be obtained within the time requested
```

It takes about a minute for you to see a response, as that's the default amount of time the database waits. So why did you get an error? It happened for two reasons. First, by default, Derby doesn't let you see a row that hasn't been committed. Second, although Derby normally locks individual rows by default, it does automatically escalate the locks to table locks if it perceives this to be an advantage. Because there is a tiny table in a tiny database, this one row represents a huge percentage of the database, so it has locked the whole table. (The different types of locks will be covered later in this tutorial.)

## Committing a transaction

The lock remains in place as long as the transaction is active. You can end the transaction by committing it. In the original window, `commit` the transaction:

```
ij> commit;
```

Now try selecting the data again in the second window (see Listing 7).


© Copyright IBM Corporation 2005. All rights reserved.

**Listing 7. Seeing the committed data**

```
                    ij> select * from demo;
THETEXT                                             |MINVALUE    |MAXVALUE
--------------------------------------------------------------------------
first value                                         |10          |20
second value                                        |15          |20
third value                                         |100         |200
```

## Rolling back a transaction

The whole point of holding the data in a sort of limbo is to acknowledge the possibility that it might never be committed. It might instead be rolled back or undone. When you roll back a transaction, you are essentially canceling all the changes you've made since starting the transaction. For example, in the first window, insert two new records (see Listing 8).

**Listing 8. Inserting more records**

```
ij> insert into demo (theText, minValue, maxValue) values ('fourth  value',\
100, 300);
1 row inserted/updated/deleted
ij> insert into demo (theText, minValue, maxValue) values ('fifth  value',
200, 300);
1 row inserted/updated/deleted
```

Now, in that same window, select the data, as shown in Listing 9.

**Listing 9. Seeing the pending data**

```
                    ij> select * from demo;
THETEXT                                             |MINVALUE    |MAXVALUE
--------------------------------------------------------------------------
first value                                         |10          |20
second value                                        |15          |20
third value                                         |100         |200
fourth value                                        |100         |300
fifth value                                         |200         |300
```

In the second window, select the data (see Listing 10).

**Listing 10. Not seeing the pending data**

```
                    ij> select * from demo;
THETEXT                                             |MINVALUE    |MAXVALUE
--------------------------------------------------------------------------
first value                                         |10          |20
second value                                        |15          |20
third value                                         |100         |200
```

Now in the first window, issue a rollback:


© Copyright IBM Corporation 2005. All rights reserved.

```
ij> rollback;
```

Now repeat the select statement in both windows, as shown in Listing 11.

**Listing 11. Seeing the permanent data**

```
                    ij> select * from demo;
THETEXT                                                   |MINVALUE    |MAXVALUE
---------------------------------------------------------------------------
first value                                               |10          |20
second value                                              |15          |20
third value                                               |100         |200
```

Commits and rollbacks are an essential part of transaction management, and will be integrated into the updated application. It also helps to understand what's really going on behind the scenes.

---

# Section 4. Transactions, locks, and isolation levels

What you've just seen is the typical level of isolation in a database. Derby provides other levels as well.

## Types of anomalies

Transactions are isolated from each other to prevent problems that can occur when more than one person is using the same database. These problems are called *anomalies*, and they fall into the following three categories.

### Dirty reads

When one user makes a change to data already in the database, that data is considered *dirty* until the change gets committed. Because the change can be rolled back at any time, it's not considered permanent; so it's considered an anomaly for another transaction to see the dirty data.

Cause: One transaction performs a select on data that's been changed, but not committed, by another transaction.

### Nonrepeatable reads

When you select data from the database, you expect to get back the data in the database. For example, if you pull up a list of events for the day, you have a


© Copyright IBM Corporation 2005. All rights reserved.

reasonable expectation that you can make decisions based on that data. A nonrepeatable read happens when one transaction is viewing data that no longer represents what's in the database.

Cause: One transaction selects a set of data, but while that transaction is active — in other words, before it's committed — another transaction makes changes to that data.

### Phantom reads

This anomaly is similar to *nonrepeatable reads* in that it involves your expectation that the data you're looking at represents what's in the database. In this case, however, the issue is *inserted data* rather than updated data. For example, if you've selected all the events for October 28, 2005, and — while you're looking at that set — I add an additional event that should be part of that SQL result, that additional record is hanging out there as a phantom read.

Cause: One transaction selects a set of data. But while that transaction is active, another transaction inserts data that should be part of that result.

## Database locks

To prevent some or all of these anomalies, the database institutes a locking system. When a transaction attempts any operation on the database, including a select statement, it generally must first request the appropriate lock. What type of locks apply to what situations depends on several factors, including the isolation level. But it's important to first understand the types of locks that are available.

### Shared locks

When a transaction selects data, it typically requests a *shared* lock first. If a second transaction wants to select the same data, it can also get a shared lock. If a transaction that holds a shared lock wants to make changes to the data, it must first convert that lock to an exclusive lock.

### Exclusive locks

When a transaction makes a change to data, it must first request an *exclusive* lock on that data. Once that lock is in place, no other transaction can get a lock on the data, so no other transaction can change that data while the lock is in place. After an exclusive lock is in place, not even a shared lock can be added, so another transaction can no longer select that data.

(A third type of lock, the *update* lock, is a combination of these two types.)

Committing or rolling back the transaction releases all of that transaction's locks.


© Copyright IBM Corporation 2005. All rights reserved.

## Lock scope

Scope is also important in locking. Derby provides two different major locking scopes: row locking and table locking.

For example, suppose you update a single row in the `Event` table. If the database is using row locking, that transaction holds an exclusive lock on that single row. Another transaction can't select that row (because it can't get a shared lock on it), but it can select other rows in that table, or insert or update other data.

On the other hand, if the database is using table locking, after you update that single record, you hold an exclusive lock on the entire table, so nobody can perform a select statement on that table, let alone make a change to the data.

Obviously, if you have a system used by multiple sessions concurrently, row locking will offer better performance, and, by default, that's the mode Derby uses. If, however, you start to accumulate so many locks that it is more efficient to use a table lock, Derby makes the change automatically. You can control the threshold for this behavior with the `derby.storage.rowLocking` property. (See *Tuning Derby* in the Resources section for details.) For example, in initial development of this application, as you saw earlier, the nearly empty tables were so small that Derby always chose table locks. It wasn't until a significant number of records were added that row locking could be observed.

## Isolation levels

Now you see the different types of anomalies that are possible and the locks Derby can use to prevent them. The way they're used determines the *isolation level* of the transaction.

Derby transactions can exist in one of the following four isolation levels:

### TRANSACTION_READ_UNCOMMITTED

A transaction using this isolation level requests no locks. In other words, if you perform an update, another transaction can see your uncommitted data. In this mode, dirty reads, nonrepeatable reads, and phantom reads are all possible. Whether you're using table or row locks doesn't matter, because the transaction doesn't request any.

### TRANSACTION_READ_COMMITTED

In this mode, the database requests an exclusive lock when you update (or delete or insert) data, which prevents dirty reads, because once that lock is in place, another


© Copyright IBM Corporation 2005. All rights reserved.

transaction can't get a shared lock on the data. On the other hand, the database requests only shared locks for selects. While the data is being viewed, it can be changed by other transactions, so nonrepeatable reads (in which the other transaction updates a row being viewed) and phantom reads (in which the other transaction adds data that should be in the set being viewed) are still possible.

These two isolation levels are the most commonly used in a typical application, with TRANSACTION_READ_COMMITTED being the default for most RDBMS systems. There are, however, two additional levels for more specialized purposes.

### TRANSACTION_REPEATABLE_READ

This is the isolation level to use when users need to be certain the data they're looking at won't change until they're finished with it. For example, suppose you perform a select of:

```
select * from Event where eventMonth = 12 and
        eventDay = 28 and eventYear = 2005
```

In the case of row locking, the database prevents updates to that data from another transaction until you commit the transaction that issued the select. So over the course of your transaction, all reads are completely repeatable. (And, of course, that means you can't read uncommitted data from someone else's transaction either.) In row locking mode, however, this level doesn't prevent inserts to the table, so while dirty reads and nonrepeatable reads are out, phantom reads are still possible.

If you use table locking, the system acts just as it would in the remaining mode, TRANSACTION_SERIALIZABLE.

### TRANSACTION_SERIALIZABLE

In this mode you can be certain of the data you're looking at, because the database treats transactions as though they're being run one after the other rather than concurrently. In the previous example, the select statement results — in table locking mode — in a lock to the entire table, preventing updates, deletes, and inserts until the transaction commits.

Even in row locking mode the database prevents all three types of anomalies, including phantom reads, by issuing range locks. So in this case, the database locks all records satisfying this where clause, even if they don't exist yet. So you would not be able to add a new event to this date in this situation.

---

# Section 5. Updating the application: EventClass


© Copyright IBM Corporation 2005. All rights reserved.

You can get the completed `EventClass` file in the Download section, but let's look at some of the changes that have been made since Part 2 of the series.

## Reorganizing

The first change since Part 2 is a real organization of sorts. Previously, if an `EventClass` object instantiated with the event data, the object would go ahead and create the record in the database. Now the constructor merely populates the class attributes, as shown in Listing 12.

### Listing 12. Reorganizing EventClass

```
                    import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Date;

import javax.swing.JOptionPane;

public class EventClass {

     public static void add(Connection conn, String newTitle,
          String newDescription, String newRemindersTo, Date
date) {
          java.util.Calendar calendar =
java.util.Calendar.getInstance();
          calendar.setTime(date);
          int month =
calendar.get(java.util.Calendar.DAY_OF_MONTH);
          int day =
calendar.get(java.util.Calendar.DAY_OF_MONTH);
          int year = calendar.get(java.util.Calendar.YEAR);
          add(conn, newTitle, newDescription, newRemindersTo,
month, day, year);
     }

     public static void add(Connection conn, String newTitle,
          String newDescription, String newRemindersTo, int
eventMonth,
          int eventDay, int eventYear) {
          System.out.println("Creating event for " + eventMonth
+ "/" + eventDay
                    + "/" + eventYear);
          System.out.println(newTitle);
          System.out.println(newDescription);
          System.out.println("Reminders to: " +
newRemindersTo);

          try {
               PreparedStatement ps = conn
                         .prepareStatement("insert into Event
(title, description,"
                                   + "remindersTo, eventMonth,
"
                                   + "eventDay, eventYear)" +
"values (?,?,?,?,?,?)");
               ps.setString(1, newTitle);
               ps.setString(2, newDescription);
               ps.setString(3, newRemindersTo);
               ps.setInt(4, eventMonth);
```


© Copyright IBM Corporation 2005. All rights reserved.

```
                ps.setInt(5, eventDay);
                ps.setInt(6, eventYear);
                ps.executeUpdate();
                System.out.println("Record updated");
                ps.close();
           } catch (Exception e) {
                ...
                e.printStackTrace();
           }
     }
...
     public EventClass(int eventId) {
          this.id = eventId;
     }

     public EventClass(String newTitle, String newDescription,
          String newRemindersTo, int eventMonth, int eventDay,
int eventYear) {
          this.title = newTitle;
          this.description = newDescription;
          this.remindersTo = newRemindersTo;
          this.eventDay = eventDay;
          this.eventMonth = eventMonth;
          this.eventYear = eventYear;
     }
...
}
```

The functionality for saving the object to the database has now been moved to the `add()` method.

## Detecting problems

Each method that makes a change to the database risks encountering a situation where the table it wants to change is locked. The class now has the ability to let the user know what's going on (see Listing 13).

### Listing 13. Detecting locks

```
                    ...
     private boolean update() {

          try {
                Statement s = this.conn.createStatement();
                PreparedStatement ps = conn
                          .prepareStatement("update event set title=?, "
                                    + "description=?, " + "remindersTo=?,"
                                    + "eventMonth=?, eventDay=?, "
                                    + "eventYear=? where id = ?");
                ps.setString(1, this.title);
                ps.setString(2, this.description);
                ps.setString(3, this.remindersTo);
                ps.setInt(4, this.eventMonth);
                ps.setInt(5, this.eventDay);
                ps.setInt(6, this.eventYear);
                ps.setInt(7, this.id);
                ps.executeUpdate();
                System.out.println("Record updated");
                ps.close();
                s.close();
          } catch (Exception e) {
```


© Copyright IBM Corporation 2005. All rights reserved.

```
                if (e.getMessage().indexOf("lock") >= 0) {
                    JOptionPane.showMessageDialog(null,
                         "Unable to update this event as the row "
                                 + "and/or table is locked.", "Lock",
                         JOptionPane.WARNING_MESSAGE);
                }
                e.printStackTrace();
           }
           return true;
      }
}
```

The last change involves the frequency with which this might come into play.

## Incremental updates to the database

Finally, the `EventClass` has also been changed so that it calls the `update()` method each time any change is made, as shown in Listing 14.

### Listing 14. Incremental changes

```
                    import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Date;

import javax.swing.JOptionPane;

public class EventClass {

     public static void add(Connection conn, String newTitle,
          String newDescription, String newRemindersTo, Date date) {
          java.util.Calendar calendar = java.util.Calendar.getInstance();
          calendar.setTime(date);
          int month = calendar.get(java.util.Calendar.DAY_OF_MONTH);
          int day = calendar.get(java.util.Calendar.DAY_OF_MONTH);
          int year = calendar.get(java.util.Calendar.YEAR);
          add(conn, newTitle, newDescription, newRemindersTo, month, day, year);
     }

     public static void add(Connection conn, String newTitle,
          String newDescription, String newRemindersTo, int eventMonth,
          int eventDay, int eventYear) {
          System.out.println("Creating event for " + eventMonth + "/" + eventDay
                    + "/" + eventYear);
          System.out.println(newTitle);
          System.out.println(newDescription);
          System.out.println("Reminders to: " + newRemindersTo);

          try {
               PreparedStatement ps = conn
                         .prepareStatement("insert into Event (title, description,"
                                   + "remindersTo, eventMonth, "
                                   + "eventDay, eventYear)" + "values (?,?,?,?,?,?)");
               ps.setString(1, newTitle);
               ps.setString(2, newDescription);
               ps.setString(3, newRemindersTo);
               ps.setInt(4, eventMonth);
               ps.setInt(5, eventDay);
               ps.setInt(6, eventYear);
               ps.executeUpdate();
```


© Copyright IBM Corporation 2005. All rights reserved.

```
                System.out.println("Record updated");
                ps.close();
           } catch (Exception e) {
             if (e.getMessage().indexOf("lock") >= 0) {
                 JOptionPane.showMessageDialog(null,
                         "Unable to add this event as the"
                                 + "table is locked.", "Lock",
                         JOptionPane.WARNING_MESSAGE);
             }
                e.printStackTrace();
           }
      }

      private Connection conn;

      private String description;

      private int eventDay;

      private int eventMonth;

      private int eventYear;

      private int id;

      private String remindersTo;

      private String title;

      public EventClass(int eventId) {
           this.id = eventId;
      }

      public EventClass(String newTitle, String newDescription,
           String newRemindersTo, int eventMonth, int eventDay, int eventYear) {
           this.title = newTitle;
           this.description = newDescription;
           this.remindersTo = newRemindersTo;
           this.eventDay = eventDay;
           this.eventMonth = eventMonth;
           this.eventYear = eventYear;
      }

      public boolean delete(Connection conn) {
           try {
                Statement s = conn.createStatement();
                String sql = "delete from Event where id= " + this.getId();
                System.out.println(sql);

                int rowsDeleted = s.executeUpdate(sql);

                System.out.println(rowsDeleted + " record(s) deleted");
                s.close();
           } catch (Exception e) {
             if (e.getMessage().indexOf("lock") >= 0) {
                 JOptionPane.showMessageDialog(null,
                         "Unable to delete this event as the row"
                                 + "and/or table is locked.", "Lock",
                         JOptionPane.WARNING_MESSAGE);
             }
                e.printStackTrace();
           }
           return true;
      }

      public String getDescription() {
           return this.description;
      }
```


© Copyright IBM Corporation 2005. All rights reserved.

```
    public int getEventDay() {
         return this.eventDay;
    }

    public int getEventMonth() {
         return this.eventMonth;
    }

    public int getEventYear() {
         return this.eventYear;
    }

    public int getId() {
         return this.id;
    }

    public String getRemindersTo() {
         return this.remindersTo;
    }

    public String getTitle() {
         return this.title;
    }

    public void load() {
         try {
              Statement s = conn.createStatement(
                        ResultSet.TYPE_SCROLL_INSENSITIVE,
                        ResultSet.CONCUR_READ_ONLY);

              ResultSet rs = s.executeQuery("SELECT * FROM Event "
                        + "where id = " + this.getId());
              if (rs.next()) {
                   this.title = rs.getString(2);
                   this.description = rs.getString(3);
                   this.remindersTo = rs.getString(4);
                   this.eventMonth = rs.getInt(5);
                   this.eventDay = rs.getInt(6);
                   this.eventYear = rs.getInt(7);
              }
         } catch (Exception e) {
              e.printStackTrace();
         }
    }

    public void setConnection(Connection conn) {
         this.conn = conn;
    }

    public void setDescription(String value) {
         this.description = value;
         update();
    }

    public void setEventDay(int value) {
         this.eventDay = value;
         update();
    }

    public void setEventMonth(int value) {
         this.eventMonth = value;
         update();
    }

    public void setEventYear(int value) {
         this.eventYear = value;
         update();
    }
```


© Copyright IBM Corporation 2005. All rights reserved.

```
    public void setId(int value) {
         this.id = value;
         update();
    }

    public void setRemindersTo(String value) {
         this.remindersTo = value;
         update();
    }

    public void setTitle(String value) {
         this.title = value;
         update();
    }

    private boolean update() {
...
```

Obviously, in your own application you should decide whether this is a viable approach to take, trading the immediacy of data going into the database with the performance issues involved in a large number of updates.

---

# Section 6. Using table models

That takes care of the changes to the back end. Now let's look at the front end. The GUI application displays the events in a table, a task that could be tedious if it weren't for models. To prevent getting sidetracked when discussing the actual interface, let's look at how these models work so that when it comes up you'll know what it's doing.

## What is a table model?

If you were a programmer, and I told you I wanted you to display information in a table, you'd know that there was a certain sequence of steps you'd need to take. You'd need to lay out the various rows and columns, keep track of which row represented which object, and so on. And if I wanted the table to be editable, there would be an even longer list of tasks you would need to perform.

The thing is, this is a pretty well-defined sequence of events. If you knew that you were going to have to do this in several places in the application, you'd probably decide to create some sort of automated process to make this easier. Well, it should come as no surprise that someone has already done that. In fact, this concept of models is part of Swing, the Java API used to put together the GUI.

The application uses an `AbstractTableModel` object to instruct the GUI on how to perform actions, such as populating the table and reacting to changes in values.


© Copyright IBM Corporation 2005. All rights reserved.

You set the model when you create the table (see Listing 15).

### Listing 15. Utilizing the table model

```
                        ...
    this.eventModel = new EventTableModel();
    reloadTableModel();
    final JTable eventTable = new JTable(this.eventModel);
    tablePanel.add(new JScrollPane(eventTable), BorderLayout.CENTER);
...
```

The `EventTableModel` is the class in which these behaviors are defined, so the `eventTable` object immediately knows what to do. Let's look at that class.

## Defining the table

The first steps are to define the class as a child of `AbstractTableModel` and to define the structure of the table itself, shown in Listing 16.

### Listing 16. Defining the table itself

```
                    import java.sql.Connection;
import java.util.Date;

import javax.swing.table.AbstractTableModel;

public class EventTableModel extends AbstractTableModel {

        String[] columns = { "Title", "Description", "Reminders To" };

        EventClass[] events;

        public int getRowCount() {
                if (events != null) {
                        return events.length;
                }
                return 0;
        }

        public int getColumnCount() {
                return columns.length;
        }

        public String getColumnName(int col) {
                return columns[col];
        }

}
```

The `EventClass[]` array represents the data that the table will display. The row count is based on the size of this array. The columns are defined by the `columns` array.

## Getting and setting values


© Copyright IBM Corporation 2005. All rights reserved.

After the Java code starts to display the table, it needs to know what data goes into each table cell, as shown in Listing 17.

**Listing 17. Providing for individual cells**

```
                        ...
        public String getColumnName(int col) {
                return columns[col];
        }

        public Object getValueAt(int rowIndex, int columnIndex) {
                if ((events != null) && (rowIndex < events.length)
                                && (columnIndex < columns.length)) {
                        switch (columnIndex) {
                        case 0:
                                return events[rowIndex].getTitle();
                        case 1:
                                return events[rowIndex].getDescription();
                        case 2:
                                return events[rowIndex].getRemindersTo();
                        default:
                                return null;
                        }
                } else {
                        return null;
                }
        }

        public boolean isCellEditable(int rowIndex, int columnIndex) {
                return true;
        }

        public void setValueAt(Object aValue, int rowIndex, int columnIndex) {
                if ((events != null) && (rowIndex < events.length)
                                && (columnIndex < columns.length)) {
                        switch (columnIndex) {
                        case 0:
                                events[rowIndex].setTitle(aValue.toString());
                                break;
                        case 1:
                                events[rowIndex].setDescription(aValue.toString());
                                break;
                        case 2:
                                events[rowIndex].setRemindersTo(aValue.toString());
                                break;
                        }
                }
        }
}
```

If the cell is editable — using the row and column information, you can set that attribute for each cell individually — the GUI needs to know what to do if the user actually edits it. In this case, the GUI sets the appropriate value on the appropriate `EventClass` object.

## Mapping a row to an object

Of course, the GUI needs to know how to relate a row in the table to a particular object (see Listing 18).


© Copyright IBM Corporation 2005. All rights reserved.

### Listing 18. Mapping a row to an object

```
                    ...
                                events[rowIndex].setRemindersTo(aValue.toString());
                                break;
                        }
                }
        }

        public EventClass getEventAtRow(int rowIndex) {
                if ((this.events != null) && (rowIndex < this.events.length)) {
                        return this.events[rowIndex];
                }
                return null;
        }
}
```

In this case, because an array is used, it's pretty straightforward.

## Refreshing the table

Finally, the GUI needs to know how to refresh itself, as shown in Listing 19.

### Listing 19. Refreshing the model

```
                    ...
public class EventTableModel extends AbstractTableModel {

        String[] columns = { "Title", "Description", "Reminders To" };

        EventClass[] events;

        public void reloadEventsForDate(Connection conn, Date date) {
                events = Calendar.getEvents(conn, date);
                fireTableDataChanged();
        }

        public int getRowCount() {
                if (events != null) {
...
```

The `reloadEventsForDate()` method takes a connection and a date and uses the `Calendar` class to get the array of appropriate events. After it has it, it calls the `fireTableDataChanged()` method, which is inherited from the `AbstractTableModel` class. That method knows how to use the ones you've already defined, such as `getColumnCount()` and `getValueAt()` to display the table.

---

# Section 7. Updating the application: CalendarFrame


© Copyright IBM Corporation 2005. All rights reserved.

Now it's time to put everything together. The main body of the application, including each of the GUI elements, is in the `CalendarFrame` class.

## The overall application

Since Part 2, the calendar application has gone through a number of changes to make it easier for people to use and maintain. Some of these changes are visible to the end user, and some are not. The major changes include a new look and feel, including a calendar picker that makes it easier to choose a date. And, as you've seen, the application now sports an editable table through which users can change information for existing events.

Other changes are not so obvious to end users. For example, earlier versions of the application connected to the database each time a change, such as an additional record, was added to the database. Now it connects when the user opens the application and maintains that connection throughout the user's session.

For easy maintenance, connections are now handled by a separate class.

## The Database class

The `Database` class is strictly a utility class, providing constants and necessary functions, such as returning a `Connection` to be used by the rest of the application (see Listing 20).

**Listing 20. The database class**

```
                    import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

class Database {

   private static final String EMBEDDED_DRIVER =
                    "org.apache.derby.jdbc.EmbeddedDriver";
   private static final String NETWORKED_DRIVER =
                      "org.apache.derby.jdbc.ClientDriver";
   private static final String PROTOCOL = "jdbc:derby";
   private static final String EMBEDDED_DB = "c:/derby/calendar";
   private static final String NETWORK_DB =
                          "//localhost:1527/c:/derby/calendar";
   public static final int EMBEDDED = 1;
   public static final int NETWORKED = 2;

   static Connection getConnection(int DB_TYPE) throws Exception {
      return getConnection(null, DB_TYPE);
   }

   static Connection getConnection(String options, int DB_TYPE)
         throws Exception {
```


© Copyright IBM Corporation 2005. All rights reserved.

```
      String url = PROTOCOL + ':';

      if (DB_TYPE == EMBEDDED) {
         url += EMBEDDED_DB;
         Class.forName(EMBEDDED_DRIVER).newInstance();
      } else if (DB_TYPE == NETWORKED) {
         url += NETWORK_DB;
         Class.forName(NETWORKED_DRIVER).newInstance();
      }

      if (options != null) {
         url += ';' + options;
      }
      return DriverManager.getConnection(url);
   }

   public static void closeConnection(Connection conn) {
      if (conn == null) {
         return;
      }

      try {
         conn.close();
         DriverManager.getConnection("jdbc:derby:;shutdown=true");
      } catch (SQLException se) {
         //se.printStackTrace();
      }
   }

   public static void createTableIfNotExists() {
      Connection conn = null;
      try {
         conn = Database.getConnection("create=true", EMBEDDED);
         Statement st = conn.createStatement();
         st.executeUpdate("create table Event (id INT GENERATED "
                     + "ALWAYS AS IDENTITY, title VARCHAR(50), "
                     + "description VARCHAR(255), remindersTo "
                     + "VARCHAR(255), eventMonth INT, eventDay INT, "
                     + "eventYear INT)");
      } catch (Exception e) {
         // e.printStackTrace();
      } finally {
         Database.closeConnection(conn);
      }
   }
}
```

The constants enable you to easily choose between the embedded and network forms of the Derby database, while the `getConnection()` and `closeConnection()` methods do just what you'd think they do. The `createTableIfNotExists()` method attempts to create the table, failing silently if it already exists.

## Creating the frame and the connection

Let's create the various areas of the GUI. The overall goal is a frame (see Listing 21) with three sections: two on the left and one on the right.

### Listing 21. Creating the frame


© Copyright IBM Corporation 2005. All rights reserved.

```
                    ...
public class CalendarFrame extends JFrame implements PropertyChangeListener {

    private Connection conn;

    private JCalendar calendarPicker;

    private EventTableModel eventModel;

    public CalendarFrame() {
        super();
        this.setTitle("Derby Calendar");
        this.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
        try {
            this.conn = Database.getConnection(Database.NETWORKED);
            this.conn.setAutoCommit(false);
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(-1);
        }

        this.getContentPane().setLayout(new BorderLayout());
        this.getContentPane().add(layoutMenuBar(), BorderLayout.NORTH);

        JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT);
        splitPane.setDividerLocation(0.6);
        splitPane.setLeftComponent(layoutLeftPanel());
        splitPane.setRightComponent(layoutRightPanel());
        this.getContentPane().add(splitPane, BorderLayout.CENTER);
    }

    private JComponent layoutLeftPanel() {
        JSplitPane viewSplitPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT);
        return viewSplitPane;
    }

    private JPanel layoutRightPanel() {
        JPanel tablePanel = new JPanel(new BorderLayout());
        return tablePanel;
    }

    private JComponent layoutMenuBar() {
        JMenuBar menuBar = new JMenuBar();
        return menuBar;
    }

    public static void main(String args[]) {
        Database.createTableIfNotExists();
        // Set the SQL lock time out to just 1 second (default is 60)
        System.setProperty("derby.locks.waitTimeout", "1");
        CalendarFrame w = new CalendarFrame();
        w.setSize(750, 550);
        w.setVisible(true);
    }

    public void propertyChange(PropertyChangeEvent evt) {
    }

}
```

Start with the `main` method. First, be sure that the database has been set up by calling `createTableIfNotExists`. If everything has already been set up, it simply fails silently. Next, set the lock timeout to one second to see what is happening when sessions conflict. With a production application, you would likely set this much higher.

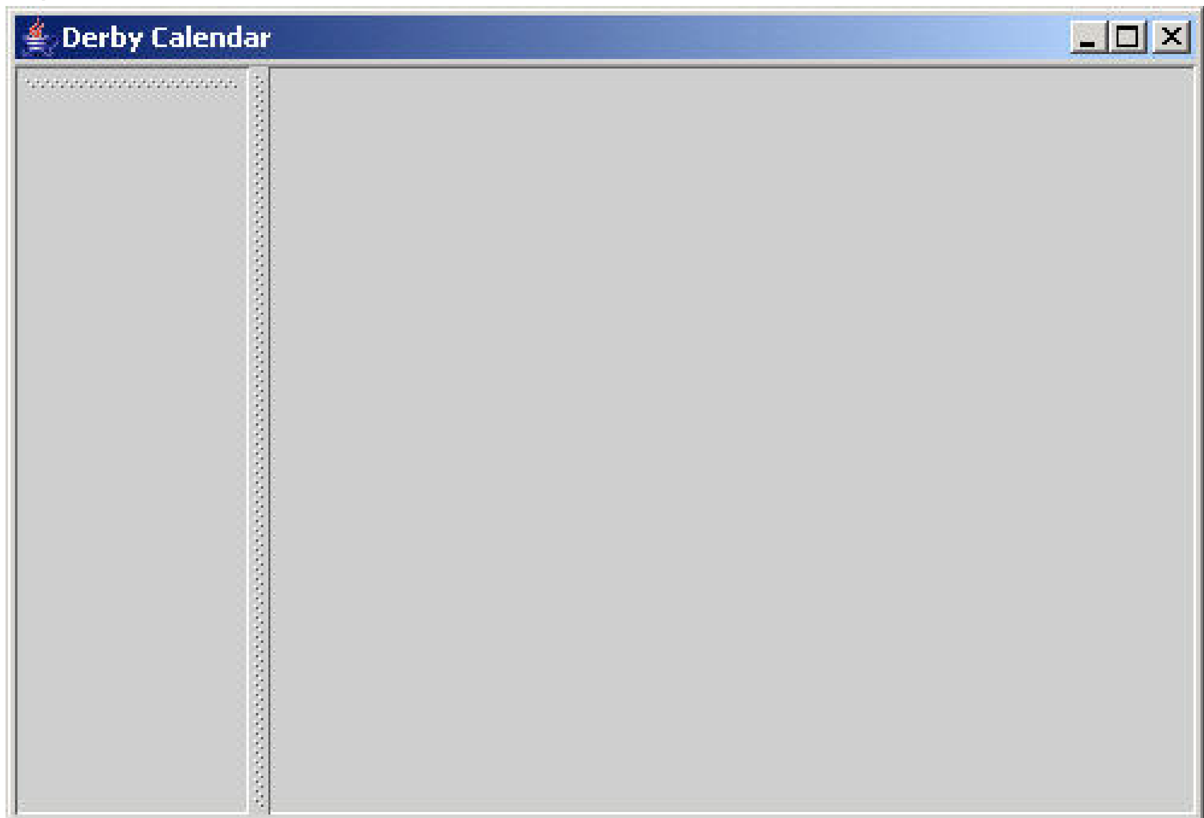

© Copyright IBM Corporation 2005. All rights reserved.

Next, create the actual frame and display it.

Creating the class involves first obtaining a connection to the database. You must then turn off the autocommit attribute if you want to be able to roll back changes.

Finally, lay out the various pieces of the GUI puzzle (see Figure 1). The application now uses `JSplitPane` objects to make it easy for the user to decide which areas of the interface need more space.

**Figure 1. The basic frame**




## Adding the basic entry form

Next, create the basic interface elements, as shown in Listing 22.

**Listing 22. The basic entry form**

```
                    ...
    private JComponent layoutLeftPanel() {

        JSplitPane viewSplitPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT);

        // Edit Panel
        JPanel editPanel = new JPanel(new GridBagLayout());
        GridBagConstraints cons = new GridBagConstraints();
        cons.fill = GridBagConstraints.HORIZONTAL;
```

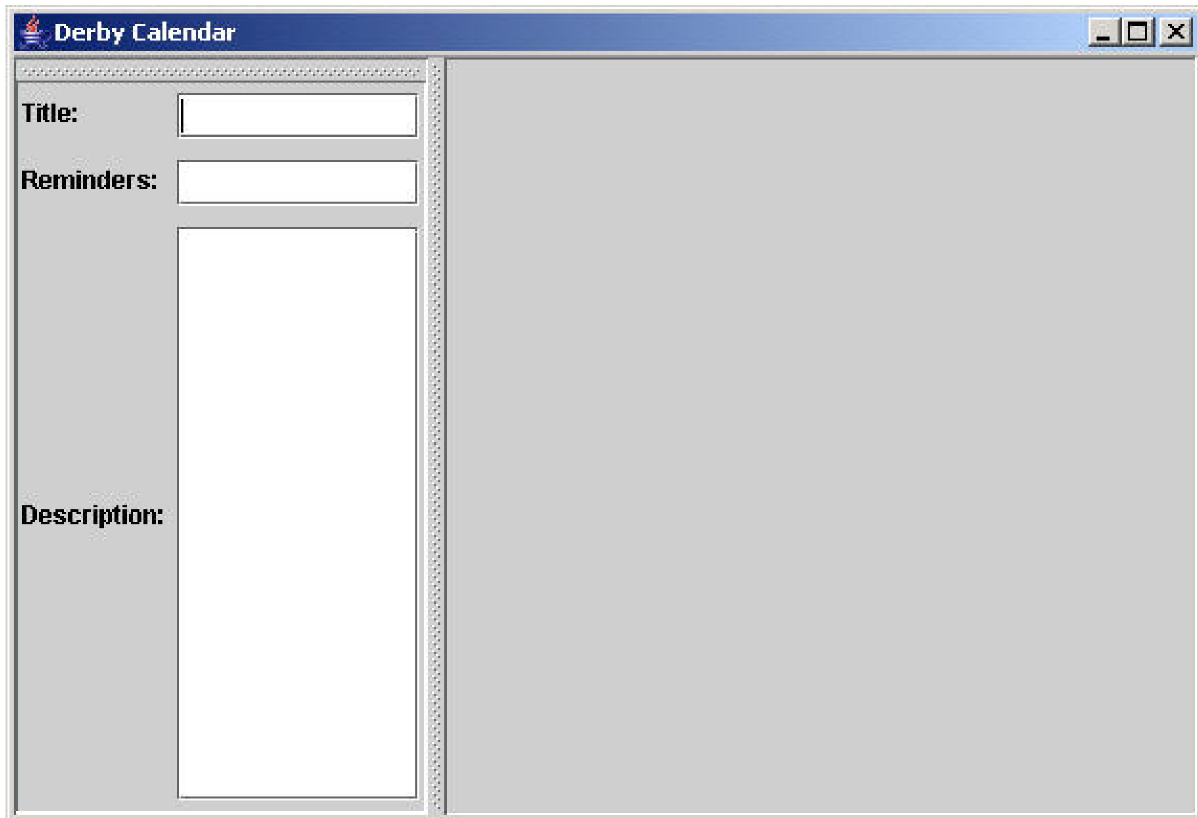

© Copyright IBM Corporation 2005. All rights reserved.

```
        cons.anchor = GridBagConstraints.PAGE_START;
        cons.insets = new Insets(5, 2, 5, 2);
        cons.ipadx = 2;
        cons.ipady = 2;

        int rows = 1;
        cons.gridy = rows++;
        cons.gridx = 0;
        editPanel.add(new JLabel("Title:"), cons);
        final JTextField titleBox = new JTextField(10);
        cons.gridx = 1;
        editPanel.add(titleBox, cons);

        cons.gridy = rows++;
        cons.gridx = 0;
        cons.gridheight = 1;
        editPanel.add(new JLabel("Reminders:"), cons);
        final JTextField reminderBox = new JTextField(10);
        cons.gridx = 1;
        editPanel.add(reminderBox, cons);

        cons.gridy = rows;
        cons.gridx = 0;
        rows += 7;
        cons.gridheight = 7;
        cons.weighty = 1;
        cons.fill = GridBagConstraints.BOTH;
        editPanel.add(new JLabel("Description:"), cons);
        final JTextArea descriptionBox = new JTextArea(8, 10);
        cons.gridx = 1;
        editPanel.add(new JScrollPane(descriptionBox), cons);

        viewSplitPane.setBottomComponent(editPanel);
        return viewSplitPane;
    }

...
```

These are essentially the same elements you added in Part 2; but these are laid out more specifically (see Figure 2).

**Figure 2. The basic entry form**


© Copyright IBM Corporation 2005. All rights reserved.




## The calendar picker

This is where it gets interesting — at least from a GUI standpoint. Rather than having the user manipulate multiple pull-down menus and hopefully come up with a valid date, you can add the calendar picker to the GUI, as shown in Listing 23.

### Listing 23. Adding the calendar picker

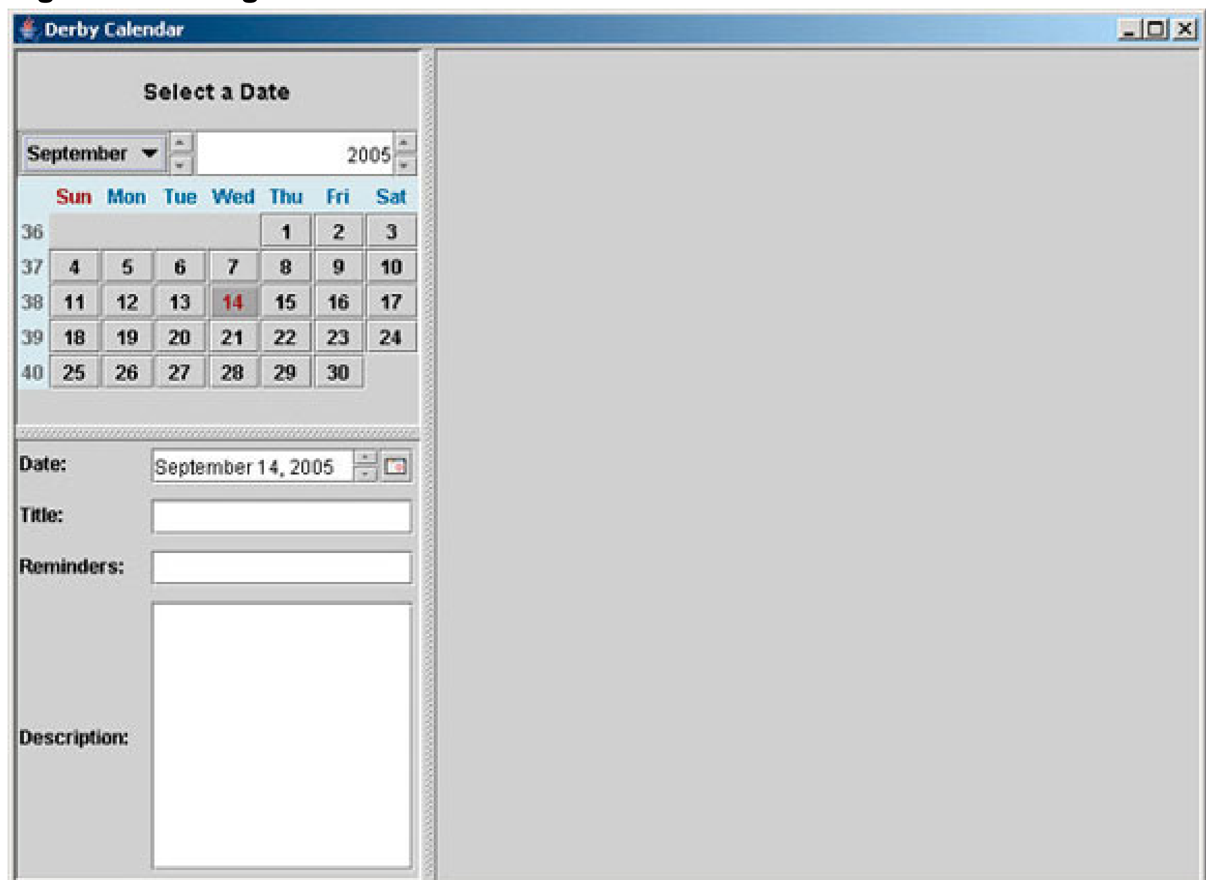
```
                ...
    private JComponent layoutLeftPanel() {

        // Calendar picker
        JPanel pickerPanel = new JPanel(new BorderLayout());
        JPanel titlePanel = new JPanel();
        titlePanel.add(new JLabel("<html><h3>Select a Date</h3></html>"));
        pickerPanel.add(titlePanel, BorderLayout.NORTH);
        JSplitPane viewSplitPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT);
        this.calendarPicker = new JCalendar();
        pickerPanel.add(this.calendarPicker, BorderLayout.CENTER);
        viewSplitPane.setTopComponent(pickerPanel);
        this.calendarPicker.addPropertyChangeListener(this);

        // Edit Panel
        JPanel editPanel = new JPanel(new GridBagLayout());
        GridBagConstraints cons = new GridBagConstraints();
        cons.fill = GridBagConstraints.HORIZONTAL;
        cons.anchor = GridBagConstraints.PAGE_START;
        cons.insets = new Insets(5, 2, 5, 2);
        cons.ipadx = 2;
```


© Copyright IBM Corporation 2005. All rights reserved.

```
            cons.ipady = 2;

            int rows = 1;
            cons.weightx = 0.5;
            cons.gridwidth = 1;
            cons.gridy = rows++;
            editPanel.add(new JLabel("Date: "), cons);
            cons.gridx = 1;
            final JDateChooser dateChooser = new JDateChooser();
            editPanel.add(dateChooser, cons);

            cons.gridy = rows++;
            cons.gridx = 0;
            editPanel.add(new JLabel("Title:"), cons);
...
```

Here you're adding the calendar in two places (see Figure 3). The upper panel provides a way for users to choose a date to display. The lower panel enables users to activate the picker to choose a date for a new event.

**Figure 3. Adding the calendar**




## Adding an event

Now that the forms are in place, what about adding an event to the database? The database action takes place in `EventClass`, but you still need to call that object


© Copyright IBM Corporation 2005. All rights reserved.

from the GUI (see Listing 24).

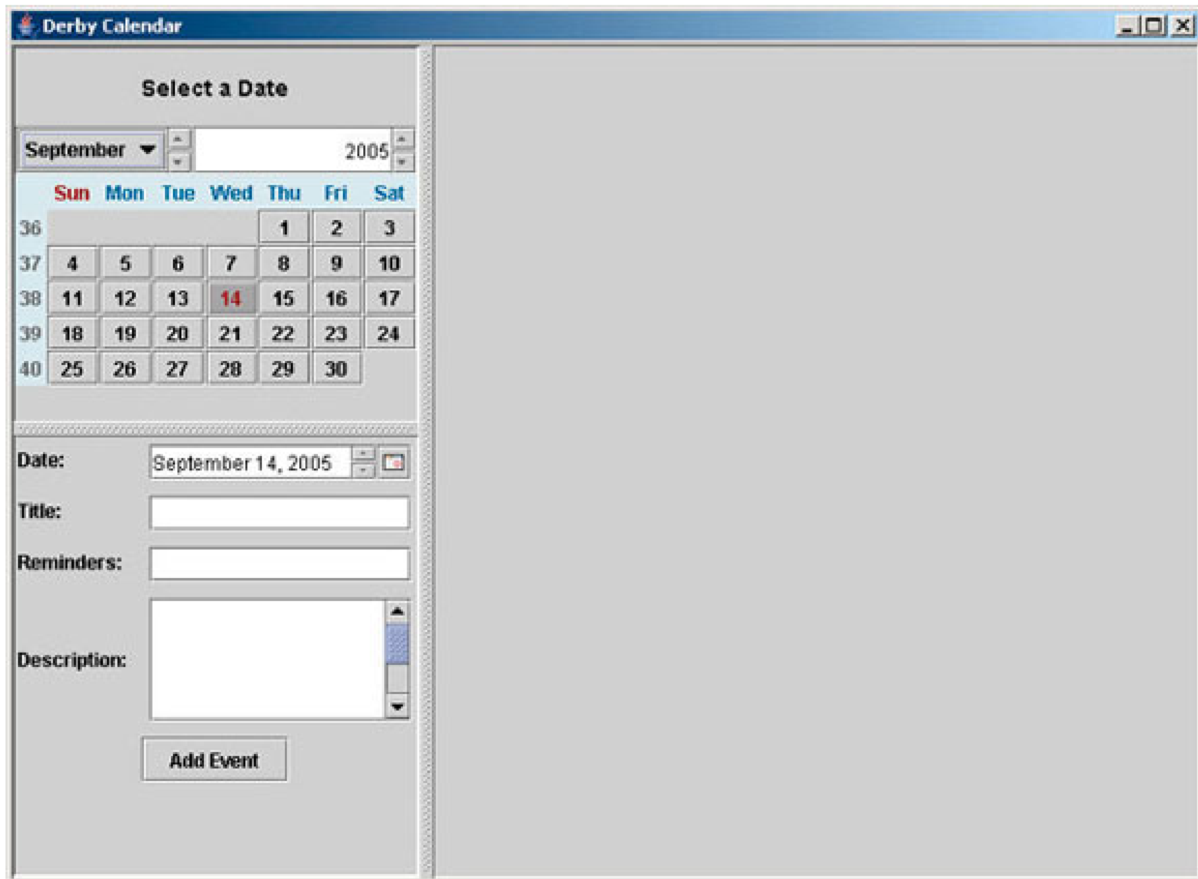
### Listing 24. Adding an event

```
                ...
    private JComponent layoutLeftPanel() {
...
        final JTextArea descriptionBox = new JTextArea(8, 10);
        cons.gridx = 1;
        editPanel.add(new JScrollPane(descriptionBox), cons);

        cons.gridy = rows++;
        cons.gridx = 0;
        cons.gridwidth = 2;
        cons.weightx = 1;
        cons.weightx = 0;
        cons.fill = GridBagConstraints.NONE;
        JButton addButton = new JButton("Add Event");
        addButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                EventClass.add(CalendarFrame.this.conn, titleBox.getText(),
                        descriptionBox.getText(), reminderBox.getText(),
                        dateChooser.getDate());
                CalendarFrame.this.calendarPicker
                        .setDate(dateChooser.getDate());
            }
        });
        editPanel.add(addButton, cons);
        viewSplitPane.setBottomComponent(editPanel);
        return viewSplitPane;
    }
...
```

Here you're adding the Add Event button, as shown in Figure 4, along with its listener. The listener calls `EventClass`, adding the event by passing the `Connection` the application created when the user started it, the information the user entered, and the date currently displayed in the editing calendar picker. It then sets the date for the upper calendar picker, which figures in later, when you add the table of data to the right side of the frame.

### Figure 4. The Add Event button


© Copyright IBM Corporation 2005. All rights reserved.




## Setting up the table

You can add all the events you like using the interface, but if users can't display them there's not much point to it. As you learned before, the application uses a table, associated with a table model, to display the data for a particular date (see Listing 25).

### Listing 25. Adding the display table

```
                    ...
    private JPanel layoutRightPanel() {
        JPanel tablePanel = new JPanel(new BorderLayout());
        JPanel topPanel = new JPanel();
        topPanel.setLayout(new BoxLayout(topPanel, BoxLayout.PAGE_AXIS));
        topPanel.add(new JLabel("<html><h3>Daily Event View</h3>"));
        tablePanel.add(topPanel, BorderLayout.NORTH);

        this.eventModel = new EventTableModel();
        reloadTableModel();
        final JTable eventTable = new JTable(this.eventModel);
        tablePanel.add(new JScrollPane(eventTable), BorderLayout.CENTER);

        return tablePanel;
    }

    private JComponent layoutMenuBar() {
```


© Copyright IBM Corporation 2005. All rights reserved.

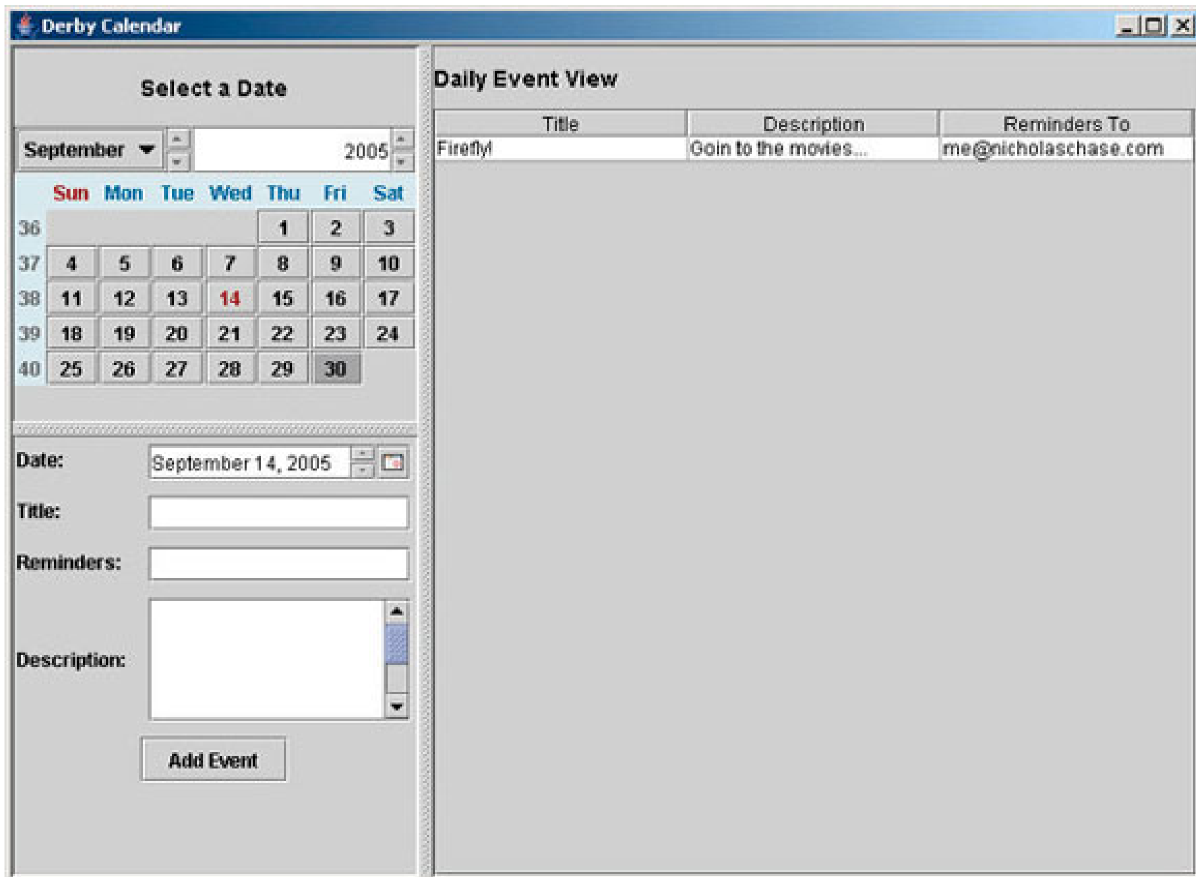
```
        JMenuBar menuBar = new JMenuBar();
        return menuBar;
    }
...
    public void propertyChange(PropertyChangeEvent evt) {
    }

    private void reloadTableModel() {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                try {
                    CalendarFrame.this.getContentPane().setCursor(
                         Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
                    CalendarFrame.this.eventModel.reloadEventsForDate(
                         CalendarFrame.this.conn,
                         CalendarFrame.this.calendarPicker.getDate());
                } finally {
                    CalendarFrame.this.getContentPane().setCursor(
                         Cursor.getPredefinedCursor(Cursor.DEFAULT_CURSOR));
                }
            }
        });
    }
}
```

Notice that first you create the `EventTableModel`, as opposed to the table itself, and populate it by calling the `reloadTableMode()` method. That method sets the cursor so the user knows something is happening, but more importantly it calls the model's `reloadEventsForDate()` method, which sets the data for specific cells.

After the `eventModel` is taken care of, associate it with a JTable and display the table (see Figure 5).

**Figure 5. Adding the table**


© Copyright IBM Corporation 2005. All rights reserved.




You still, however, have to deal with updating the table when the user clicks on a date.

## Refreshing the list of events

After you have the table in place, you need to make sure that it always displays the appropriate data. In other words, if the user chooses a new day in the calendar picker, you need to request the appropriate information from the database and display it in the table (see Listing 26). (See how handy that `EventTableMode` is?)

### Listing 26. Refreshing the table

```
                ...
    private JComponent layoutLeftPanel() {
...
        JButton addButton = new JButton("Add Event");
        addButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                EventClass.add(CalendarFrame.this.conn, titleBox.getText(),
                        descriptionBox.getText(), reminderBox.getText(),
                        dateChooser.getDate());
                CalendarFrame.this.calendarPicker
                        .setDate(dateChooser.getDate());
                reloadTableModel();
```


© Copyright IBM Corporation 2005. All rights reserved.

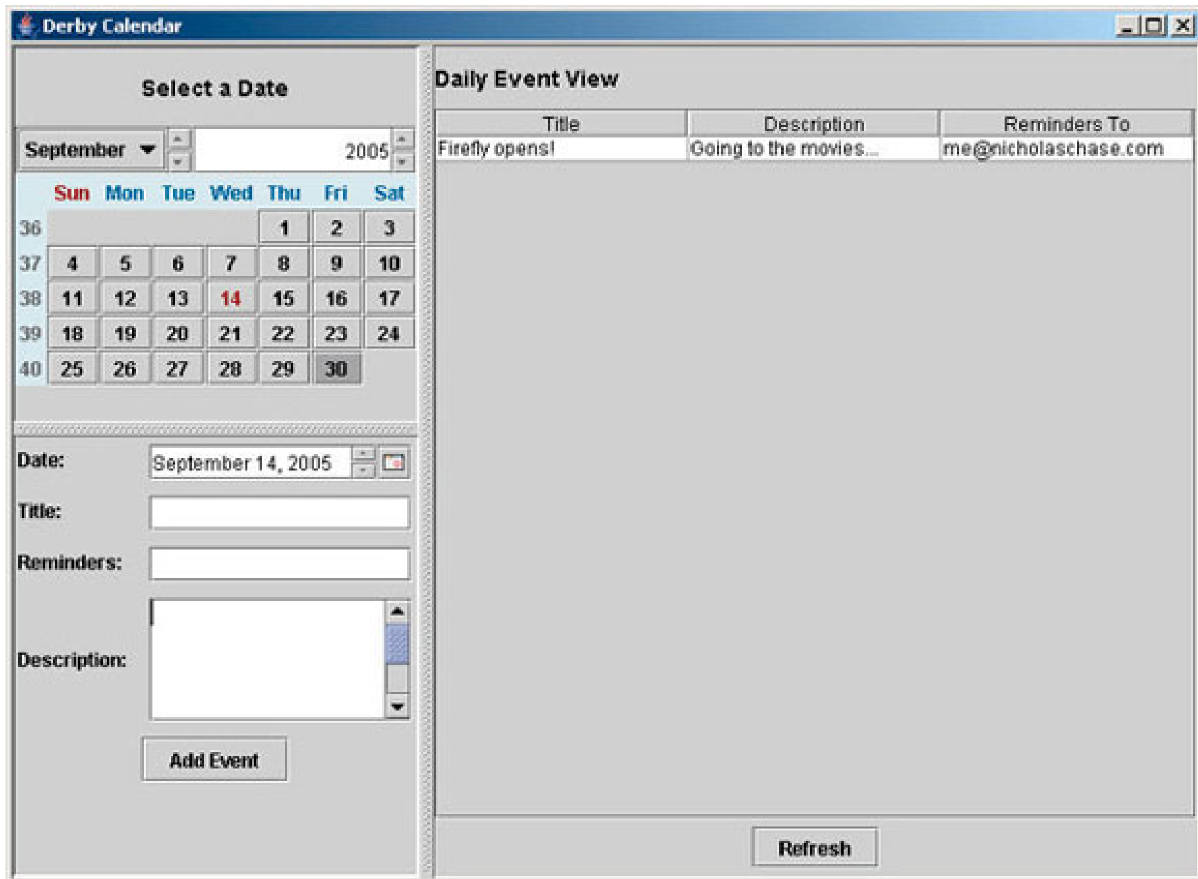
```
            }
        });
        editPanel.add(addButton, cons);
        viewSplitPane.setBottomComponent(editPanel);
        return viewSplitPane;
    }

    private JPanel layoutRightPanel() {
...
        this.eventModel = new EventTableModel();
        reloadTableModel();
        final JTable eventTable = new JTable(this.eventModel);
        tablePanel.add(new JScrollPane(eventTable), BorderLayout.CENTER);

        JPanel buttonPanel = new JPanel();

        JButton refreshButton = new JButton("Refresh");
        refreshButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                reloadTableModel();
            }
        });
        buttonPanel.add(refreshButton);
        tablePanel.add(buttonPanel, BorderLayout.SOUTH);
        return tablePanel;
    }

    private JComponent layoutMenuBar() {
        JMenuBar menuBar = new JMenuBar();
        return menuBar;
    }

    public static void main(String args[]) {
...
    }

    public void propertyChange(PropertyChangeEvent evt) {
        reloadTableModel();
    }

    private void reloadTableModel() {
        SwingUtilities.invokeLater(new Runnable() {
...
```

By adding the `reloadTableModel()` method to `propertyChange`, you tell the application to request fresh data every time the user picks a new date. Also, the user can now request the latest data by simply clicking the Refresh button.

The application also changes to the appropriate date and shows new data when the user adds a new event. This way, the user gets to see the new event in place (see Figure 6).

**Figure 6. Refreshing the table**


© Copyright IBM Corporation 2005. All rights reserved.




## Sending reminders

Now it's time to start adding the other various buttons and functions. Start with the reminders (see Listing 27).

### Listing 27. Sending reminders

```
                ...
    private JPanel layoutRightPanel() {
...
        JPanel buttonPanel = new JPanel();

        JButton remindButton = new JButton("Remind");
        remindButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                try {
                    int row = eventTable.getSelectedRow();
                    if (row >= 0) {
                        EventClass event = CalendarFrame.this.eventModel
                                .getEventAtRow(row);
                        Reminder rem = new Reminder();
                        rem.sendReminder(CalendarFrame.this.conn,
                                             event.getId());
                    }
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
```


© Copyright IBM Corporation 2005. All rights reserved.

```
            }
        });
        buttonPanel.add(remindButton);
...
        tablePanel.add(buttonPanel, BorderLayout.SOUTH);
        return tablePanel;
    }
...
```

In this case, the Remind button sends not all reminders, but only the specific reminder for the event the user has chosen. (That's where it's handy to be able to map rows to events.)

The Delete button works the same way.

## Deleting a record

The Delete button also counts on being able to map a row of data to a specific `EventClass` object (see Listing 28).

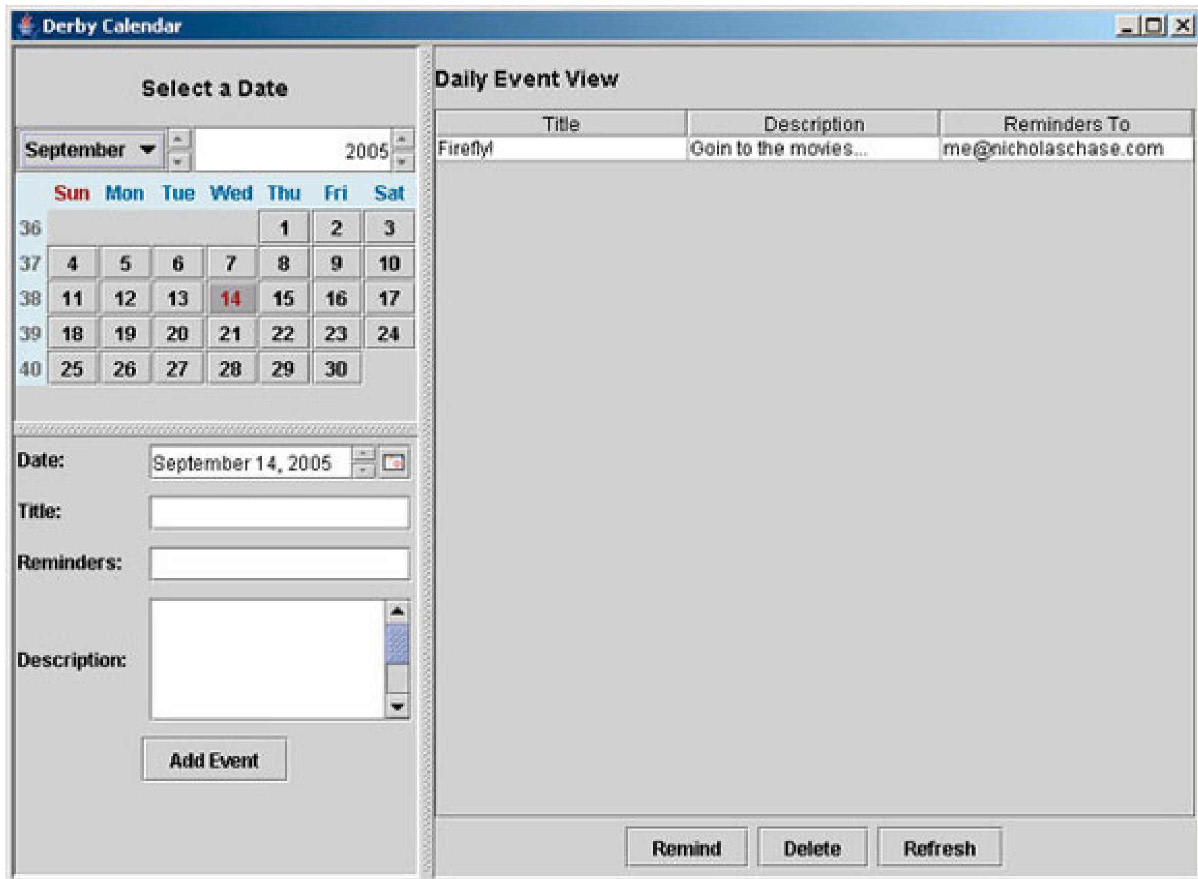
### Listing 28. Deleting an event

```
                ...
    private JPanel layoutRightPanel() {
...
        buttonPanel.add(remindButton);

        JButton deleteButton = new JButton("Delete");
        deleteButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                try {
                    int row = eventTable.getSelectedRow();
                    if (row >= 0) {
                        EventClass event = CalendarFrame.this.eventModel
                                .getEventAtRow(row);
                        event.delete(CalendarFrame.this.conn);
                        reloadTableModel();
                    }
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
        });
        buttonPanel.add(deleteButton);

        JButton refreshButton = new JButton("Refresh");
        refreshButton.addActionListener(new ActionListener() {
...
        tablePanel.add(buttonPanel, BorderLayout.SOUTH);
        return tablePanel;
    }

...
```

In this case, the application gets a reference to the specific `EventClass` object represented by the row in the table and uses that object's own `delete()` method to delete it (see Figure 7) before reloading the table.


© Copyright IBM Corporation 2005. All rights reserved.

**Figure 7. Deleting a record**




# Section 8. Managing the transaction

If the application were set to use autocommit, you wouldn't have to add any more code. Of course, you wouldn't have the ability to undo changes either. In this section, you'll add the transaction management functionality.

## Committing changes

After the user has made changes to the database, he or she needs to make a decision: Save the changes, or revert back to the database as it was before the changes? Start by giving the user an opportunity to save changes (see Listing 29).

**Listing 29. Committing changes**

```
...
```


© Copyright IBM Corporation 2005. All rights reserved.

```
    private JPanel layoutRightPanel() {
...
        buttonPanel.add(refreshButton);

        JButton saveButton = new JButton("Save");
        saveButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                try {
                    CalendarFrame.this.conn.commit();
                } catch (SQLException ex) {
                    ex.printStackTrace();
                }
            }
        });
        buttonPanel.add(saveButton);
        tablePanel.add(buttonPanel, BorderLayout.SOUTH);
        return tablePanel;
    }
...
```

The action for this button is extremely simple: Get the connection in use by this instance of the application, and use it to commit the current transaction.

The Revert button works in much the same way.

## Rolling back changes

The Revert button also uses the connection to the database (see Listing 30).

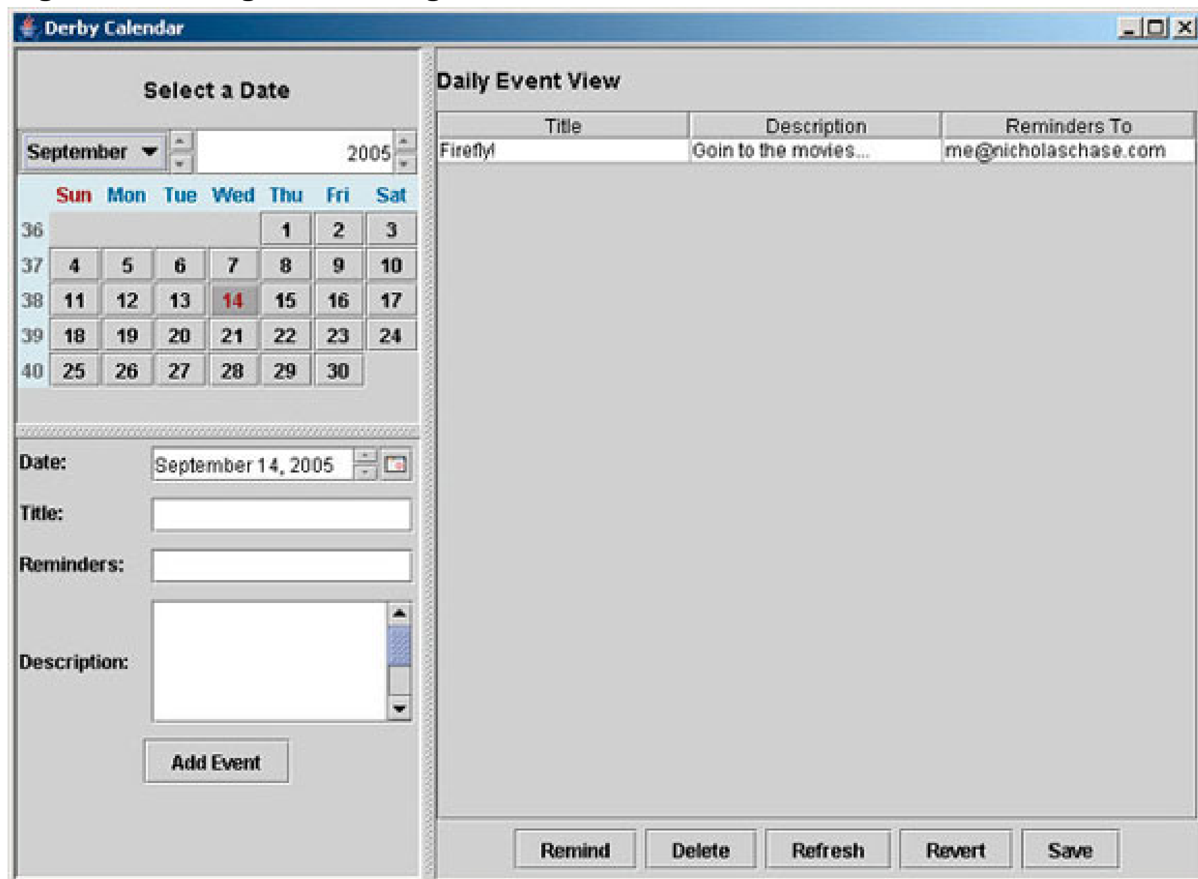
### Listing 30. Rolling back changes

```
                ...
    private JPanel layoutRightPanel() {
...
        buttonPanel.add(refreshButton);

        JButton revertButton = new JButton("Revert");
        revertButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                try {
                    CalendarFrame.this.conn.rollback();
                    reloadTableModel();
                } catch (SQLException ex) {
                    ex.printStackTrace();
                }
            }
        });
        buttonPanel.add(revertButton);

        JButton saveButton = new JButton("Save");
...
        tablePanel.add(buttonPanel, BorderLayout.SOUTH);
        return tablePanel;
    }
...
```

After rolling back the current transaction (see Figure 8), you should refresh the table to show the old data.


© Copyright IBM Corporation 2005. All rights reserved.

**Figure 8. Rolling back changes**




## Setting the isolation level

Part of the project includes enabling users to decide whether they want to see data that hasn't been saved (committed) yet, so you need to give them that choice (see Listing 31).

**Listing 31. Choosing an isolation level**

```
...
private JComponent layoutMenuBar() {
    JMenuBar menuBar = new JMenuBar();

    ActionListener transListener = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            JCheckBoxMenuItem item = (JCheckBoxMenuItem) e.getSource();
            String text = item.getText();
            try {
                if (text.equals("See all data")) {
                    CalendarFrame.this.conn
                         .setTransactionIsolation(
                               Connection.TRANSACTION_READ_UNCOMMITTED);
                } else if (text.equals("See only saved data")) {
                    CalendarFrame.this.conn
                         .setTransactionIsolation(
                                 Connection.TRANSACTION_READ_COMMITTED);
```


© Copyright IBM Corporation 2005. All rights reserved.

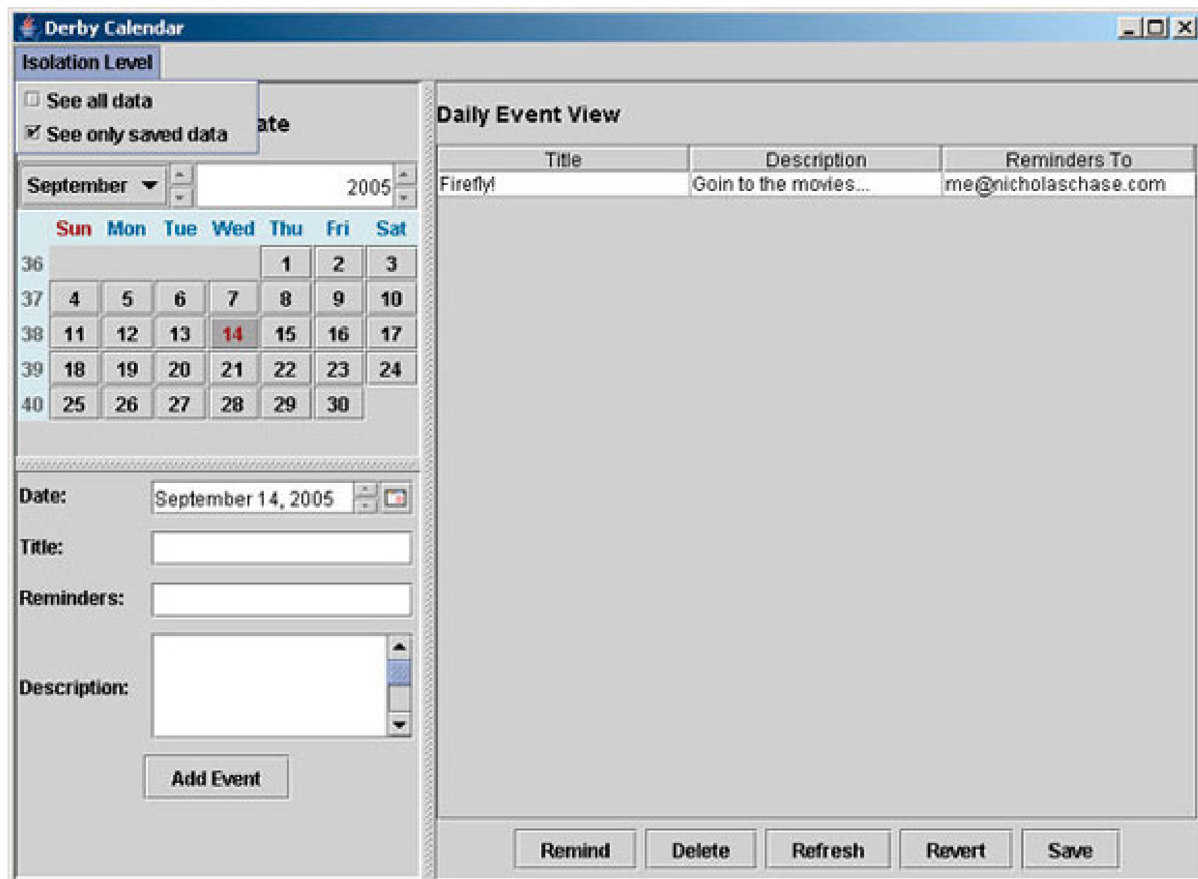
```
                    }
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
        };

        JMenu menu = new JMenu("Isolation Level");
        ButtonGroup group = new ButtonGroup();

        JMenuItem item = new JCheckBoxMenuItem("See all data");
        group.add(item);
        menu.add(item);
        item.addActionListener(transListener);

        item = new JCheckBoxMenuItem("See only saved data");
        group.add(item);
        menu.add(item);
        group.setSelected(item.getModel(), true);
        item.addActionListener(transListener);

        menuBar.add(menu);
        return menuBar;
    }
...
```

The phrase *isolation level* means nothing to most users, and the term *TRANSACTION_READ_COMMITTED* means even less, so you'll have to come up with names that represent what you're really trying to do. In this case, you're creating a new menu with two items added to it, their names explaining in more friendly terms what it is they do.

When the user chooses one of these items, the `transListener` analyzes the label on the menu item and uses it to determine which isolation level to use for the current connection (see Figure 9).

**Figure 9. Choosing the isolation level**


© Copyright IBM Corporation 2005. All rights reserved.




## Section 9. Transactions in action

It's time to stop talking about transactions and isolation levels and finally see them in action. Open two different instances of the CalendarFrame application.

### TRANSACTION_READ_COMMITTED

Make sure both application instances are set to `See only saved data`, which corresponds to the TRANSACTION_READ_COMMITTED isolation level.

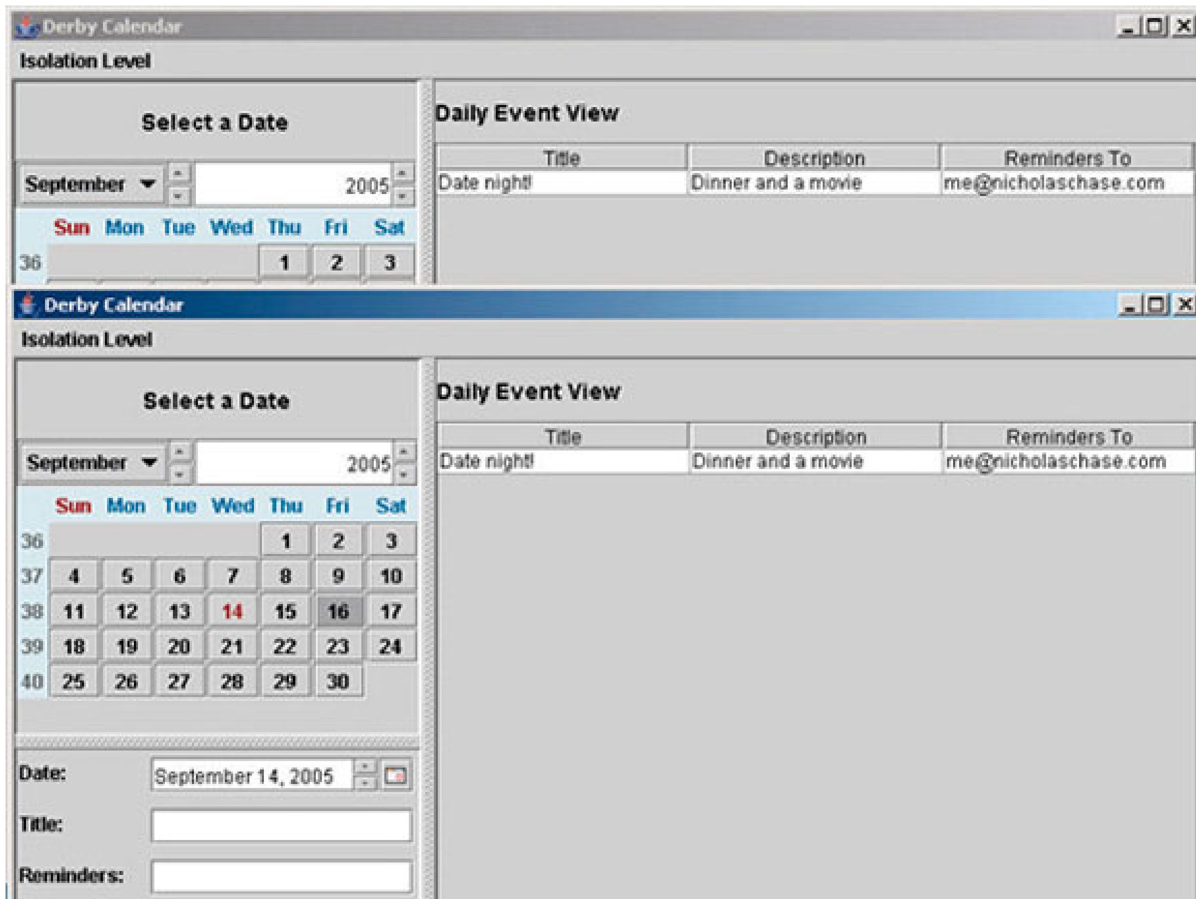
In the first window, pick a day and add an event. Then go to the second window and display the events for that day. After a pause, you should get a dialog box (see Figure 10) telling you that the application can't retrieve the data because it's locked.

**Figure 10. When you can't get a lock on the data**


© Copyright IBM Corporation 2005. All rights reserved.

Now, in the first window, click the **Save** button to commit the transaction. Go back to the second window and try to refresh the page, but only once (see Figure 11).

**Figure 11. Committing the transaction**




Notice that you can now request the same data that was previously locked.

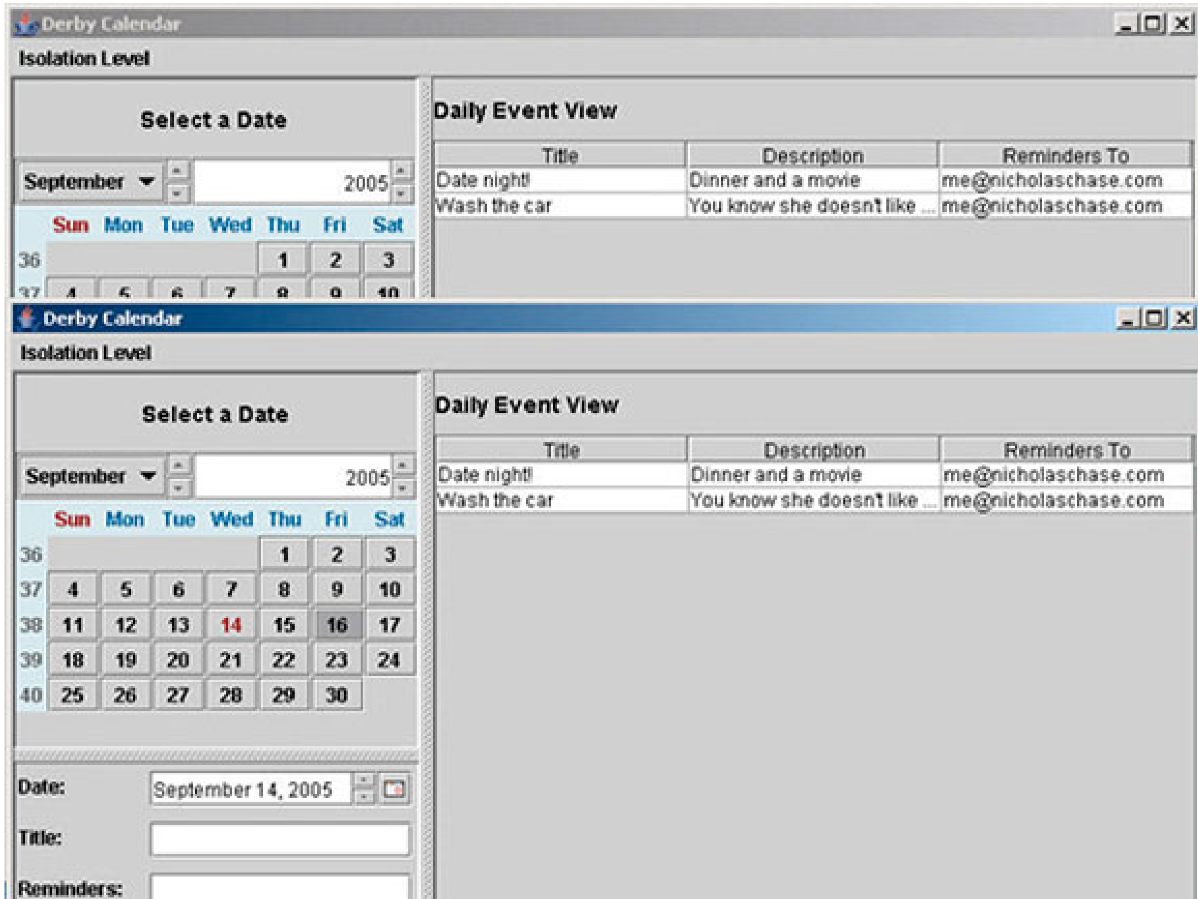
## TRANSACTION_READ_UNCOMMITTED

In the previous example, you saw how the database kept one user from seeing uncommitted data for another user. But what if you don't want that to happen? In both windows, set the isolation level to `See all data`.


© Copyright IBM Corporation 2005. All rights reserved.

Now add a new record in the first window, but don't save your changes. Go to the second window and display the events for that date. You should see all the events for that date, including the one you just entered a moment ago (see Figure 12).

**Figure 12. Seeing dirty data**




Now users can complete all their event-management tasks, including seeing what other users are up to.

---

# Section 10. Summary

This tutorial is the final part in a three-part series chronicling the building of a Java-based calendar and reminder system that uses the Derby database as its data storage system. Part 1 covered the basic database and e-mail entities. Part 2 built those entities into a simple application that explored Derby's embedded, network, and Web-based modes. It also introduced the concept of multiple users.


© Copyright IBM Corporation 2005. All rights reserved.

This tutorial took the overall system and tackled concurrency, discussing transactions, locking, and isolation levels. It also described significant improvements to the overall look and feel and usability of the application.

You now have a system that can be used by multiple individuals at the same time, all editing a common event calendar. You can also choose whether or not to see data that has not yet been committed to the database.


© Copyright IBM Corporation 2005. All rights reserved.

# Downloads

| Description | Name | Size | Download method |
| --- | --- | --- | --- |
| Source code from Part 2 | part2source.zip | 8 KB | HTTP |
| Source code for Part 3 | part3source.zip | 12 KB | HTTP |

Information about download methods


© Copyright IBM Corporation 2005. All rights reserved.

# Resources

### Learn

- Start at the developerWorks Apache Derby resource area for all your Derby resources and articles.
- Check out Lance Bader's article, "Integrate Cloudscape Version 10 or Derby with Tomcat" (developerWorks, August 2005) for more information on embedding IBM Cloudscape/Derby in an application server.
- Read this article about connecting to Apache Derby databases using Python, "Connect to Apache Derby databases using Python" (developerWorks, May 2005).
- Read the *Tuning Derby* guide.
- Read "Develop Apache Derby applications in Eclipse" (developerWorks, January 2005) if you're interested in developing Derby applications in Eclipse.
- Check out this great article on Derby development with Ant.
- Take a look at "Connecting PHP Applications to Apache Derby" (developerWorks, September 2004).
- Read "Connect to Apache Derby databases using Jython" (developerWorks, February 2005).
- 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

- 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

- Get involved in the developerWorks community by participating in developerWorks blogs.


© Copyright IBM Corporation 2005. All rights reserved.

# About the author

Nicholas Chase

Nicholas Chase has been involved in Web-site development for companies such as Lucent Technologies, Sun Microsystems, Oracle, and the Tampa Bay Buccaneers. He has been a high school physics teacher, a low-level radioactive waste facility manager, an online science fiction magazine editor, a multimedia engineer, an Oracle instructor, and the chief technology officer of an interactive communications company. He is the author of several books, including *XML Primer Plus*.


© Copyright IBM Corporation 2005. All rights reserved.