CHAPTER 9

Working with Date, Time, and Timestamp in JDBC

Date, Time, and Timestamp are important data types in commercial and banking applications. For example, the following are some of their uses:

- Time of airline arrival
- Date of purchase
- Hire date of an employee
- Update date of employee's salary
- Purge date of database
- Purchase date of items
- Account creation date
- Account expiration date

The purpose of this chapter is to provide snippets and code samples that deal with Date, Time, and Timestamp. Please note that most of the solutions are provided as independent static methods (In other words, these methods do not depend on external static data structures and rely only on the passed input arguments; therefore, you can use these methods just as they appear in this chapter.)

The java.sql package provides the following classes that deal with date-related data types (portions of these descriptions were taken from the Java 2 Platform, Standard Edition 1.4.1):

java.sql.Date: Mapping for SQL DATE. The java.sql.Date class extends the java.util.Date class. This is a thin wrapper around a millisecond value that allows JDBC to identify this as a SQL DATE value. A millisecond value represents the number of milliseconds that have passed since January 1, 1970, 00:00:00.000 GMT. To conform with the definition of SQL DATE, the millisecond values wrapped by a java.sql.Date instance must be "normalized" by setting the hours, minutes, seconds, and milliseconds to zero in the particular time zone with which the instance is associated.

java.sql.Time: Mapping for SQL TIME. The java.sql.Time class extends the java.util.Date class. This is a thin wrapper around the java.util.Date class that allows the JDBC API to identify this as a SQL TIME value. The Time class adds formatting and parsing operations to support the JDBC escape syntax for time values. The date components should be set to the "zero epoch" value of January 1, 1970, and should not be accessed.

`java.sql.Timestamp`: Mapping for SQL `TIMESTAMP`. The `java.sql.Timestamp` class extends the `java.util.Date` class. This is a thin wrapper around `java.util.Date` that allows the JDBC API to identify this as a SQL `TIMESTAMP` value. It adds the ability to hold the SQL `TIMESTAMP` "nanos value" and provides formatting and parsing operations to support the JDBC escape syntax for timestamp values. This type is a composite of a `java.util.Date` and a separate nanosecond value. Only integral seconds are stored in the `java.util.Date` component. The fractional seconds—the nanos—are separate. The `getTime` method will return only integral seconds. If a time value that includes the fractional seconds is desired, you must convert nanos to milliseconds (nanos/1,000,000) and add this to the `getTime` value. The `Timestamp.equals(Object)` method never returns `true` when passed a value of type `java.util.Date` because the nanos component of a date is unknown. As a result, the `Timestamp.equals(Object)` method is not symmetric with respect to the `java.util.Date.equals(Object)` method. Also, the hash code method uses the underlying `java.util.Data` implementation and therefore does not include nanos in its computation. Because of the differences between the `Timestamp` class and the `java.util.Date` class mentioned previously, it is recommended that you check the vendor's implementation of SQL's `DATE`, `TIME`, and `TIMESTAMP` data types. The inheritance relationship between `Timestamp` and `java.util.Date` really denotes implementation inheritance, not type inheritance.

Each database vendor provides specific data types for supporting `Date`, `Time`, and `Timestamp`. To fully understand these data types and provide JDBC access to these types, you will see how to use the MySQL and Oracle databases in the examples.

9-1. What Is the Mapping of Date-Related SQL and Java Types?

Three JDBC types are related to time:

- The JDBC `DATE` type (`java.sql.Date`, which extends `java.util.Date`) represents a date consisting of the day, month, and year. The corresponding SQL `DATE` type is defined in SQL-92.
- The JDBC `TIME` type (`java.sql.Time`, which extends `java.util.Date`) represents a time consisting of hours, minutes, and seconds. The corresponding SQL `TIME` type is defined in SQL-92.
- The JDBC `TIMESTAMP` type (`java.sql.Timestamp`, which extends `java.util.Date`) represents `DATE` plus `TIME` plus a nanosecond field. The corresponding SQL `TIMESTAMP` type is defined in SQL-92.

The `DATE` type is where most mismatches occur. This is because the `java.util.Date` class represents both `Date` and `Time`, but SQL has the following three types to represent date and time information:

- A SQL `DATE` type that represents the date only (01/26/88)
- A SQL `TIME` type that specifies the time only (09:06:56)
- A SQL `TIMESTAMP` that represents the time value in nanoseconds

Table 9-1 shows the mapping of JDBC types to Java types, which can be mapped back to JDBC types, because this is a one-to-one relationship mapping.

Table 9-1. *JDBC Types Mapped to Java Types*

JDBC Type	Java Type
`DATE`	`java.sql.Date`
`TIME`	`java.sql.Time`
`TIMESTAMP`	`java.sql.Timestamp`

9-2. How Do You Retrieve Date, Time, and Timestamp from a Database?

The `ResultSet` interface provides the necessary methods for extracting `Date`, `Time`, and `Timestamp` from a database. Table 9-2 lists the `ResultSet.getXXX()` methods for retrieving data from the `Date`, `Time`, and `Timestamp` types.

Table 9-2. `ResultSet` *Methods for Retrieving* `Date`, `Time`, *and* `Timestamp` *(According to the JDK)*

Return Type	Method	Description
`java.sql.Date`	`getDate(int columnIndex)`	Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Date` object in the Java programming language.
`java.sql.Date`	`getDate(int columnIndex, Calendar cal)`	Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Date` object in the Java programming language.
`java.sql.Date`	`getDate(String columnName)`	Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Date` object in the Java programming language.
`java.sql.Date`	`getDate(String columnName, Calendar cal)`	Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Date` object in the Java programming language.
`java.sql.Time`	`getTime(int columnIndex)`	Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Time` object in the Java programming language.
`java.sql.Time`	`getTime(int columnIndex, Calendar cal)`	Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Time` object in the Java programming language.
`java.sql.Time`	`getTime(String columnName)`	Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Time` object in the Java programming language.
`java.sql.Time`	`getTime(String columnName, Calendar cal)`	Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Time` object in the Java programming language. This method uses the given calendar to construct an appropriate millisecond value for the time if the underlying database does not store time zone information.

Continued

Table 9-2. (*Continued*)

Return Type	Method	Description
java.sql.Timestamp	getTimestamp(int columnIndex)	Retrieves the value of the designated column in the current row of this ResultSet object as a java.sql.Timestamp object in the Java programming language.
java.sql.Timestamp	getTimestamp(int columnIndex, Calendar cal)	Retrieves the value of the designated column in the current row of this ResultSet object as a java.sql.Timestamp object in the Java programming language.
java.sql.Timestamp	getTimestamp(String columnName)	Retrieves the value of the designated column in the current row of this ResultSet object as a java.sql.Timestamp object.
java.sql.Timestamp	getTimestamp(String columnName, Calendar cal)	Retrieves the value of the designated column in the current row of this ResultSet object as a java.sql.Timestamp object in the Java programming language.

9-3. How Does MySQL Handle Date, Time, and Timestamp?

In MySQL, the date and time types are DATETIME, DATE, TIMESTAMP, TIME, and YEAR. According to the MySQL manual, the DATETIME, DATE, and TIMESTAMP types are related:

> DATETIME: Use the DATETIME type when you need values that contain both date and time information. MySQL retrieves and displays DATETIME values in YYYY-MM-DD HH:MM:SS format. The supported range is from 1000-01-01 00:00:00 to 9999-12-31 23:59:59. (*Supported* means that although other values might work, there is no guarantee they will.)
>
> DATE: Use the DATE type when you need only a date value without a time part. MySQL retrieves and displays DATE values in YYYY-MM-DD format. The supported range is from 1000-01-01 to 9999-12-31.
>
> TIMESTAMP: The TIMESTAMP column type provides a type that you can use to automatically mark INSERT or UPDATE operations with the current date and time. If you have multiple TIMESTAMP columns, only the first one is updated automatically. For automatic updating of the first TIMESTAMP column, please refer to the MySQL manual.

9-4. How Does Oracle Handle Date, Time, and Timestamp?

The Oracle DATE data type contains both date and time information. The Oracle DATE data type is a complex data type that encapsulates date, time, and timestamp concepts.

Table 9-3 shows the mapping between JDBC data types, Java native data types, and Oracle data types:

Table 9-3.Oracle's Date Types

Standard JDBC Data Types	Java Native Data Types	Oracle Data Types
java.sql.Types.DATE	java.sql.Date	DATE
java.sql.Types.TIME	java.sql.Time	DATE
java.sql.Types.TIMESTAMP	javal.sql.Timestamp	DATE

To understand these three classes, I will show how to define a table that has columns that directly deal with these three classes. Here is the definition of the table:

```
create table TestDates(
    id VARCHAR2(10) NOT NULL Primary Key,
    date_column DATE,
    time_column DATE,
    timestamp_column DATE
);
```

Here is how you create the `TestDates` table (the output has been formatted):

```
$ sqlplus scott/tiger
SQL*Plus: Release 9.2.0.1.0 - Production on Sun Nov 24 21:47:08 2002

SQL>    create table TestDates(
  2             id VARCHAR2(10) NOT NULL Primary Key,
  3             date_column DATE,
  4             time_column DATE,
  5             timestamp_column DATE
  6     );

Table created.
SQL> commit;
Commit complete.

SQL> describe  TestDates
 Name                  Null?    Type
 --------------------- -------- -------------
 ID                    NOT NULL VARCHAR2(10)
 DATE_COLUMN                    DATE
 TIME_COLUMN                    DATE
 TIMESTAMP_COLUMN               DATE
```

I will cover the following topics for the `TestDates` table:

- Inserting a new record
- Retrieving an existing record
- Updating an existing record
- Deleting an existing record

To simplify your work in these examples, you will get a JDBC `Connection` object from a static method. (In real database applications, you would get the `Connection` object, `java.sql.Connection`, from a connection pool manager, which manages a set of JDBC `Connection` objects.)

Oracle Date and Time Types: Inserting a New Record

The following program creates a new record in the TestDates table (with an ID of "id100"). Here is the program listing:

```
import java.sql.*;
import jcb.util.DatabaseUtil;

public class InsertDate {
   public static Connection getConnection() throws Exception {
       String driver = "oracle.jdbc.driver.OracleDriver";
       String url = "jdbc:oracle:thin:@localhost:1521:scorpian";
       Class.forName(driver);
       System.out.println("ok: loaded oracle driver.");
       return DriverManager.getConnection(url, "scott", "tiger");
   }

   public static void main(String args[]) {
      String INSERT_RECORD = "insert into TestDates(id, date_column, "+
           "time_column, timestamp_column) values(?, ?, ?, ?)";
      Connection conn = null;
      PreparedStatement pstmt = null;
      try {
         conn = getConnection();
         pstmt = conn.prepareStatement(INSERT_RECORD);
         pstmt.setString(1, "id100");

         java.util.Date date = new java.util.Date();
         long t = date.getTime();
         java.sql.Date sqlDate = new java.sql.Date(t);
         java.sql.Time sqlTime = new java.sql.Time(t);
         java.sql.Timestamp sqlTimestamp = new java.sql.Timestamp(t);
         System.out.println("sqlDate="+sqlDate);
         System.out.println("sqlTime="+sqlTime);
         System.out.println("sqlTimestamp="+sqlTimestamp);
         pstmt.setDate(2, sqlDate);
         pstmt.setTime(3, sqlTime);
         pstmt.setTimestamp(4, sqlTimestamp);
         pstmt.executeUpdate();
      }
      catch( Exception e ) {
         e.printStackTrace();
         System.out.println("Failed to insert the record.");
         System.exit(1);
      }
      finally {
          DatabaseUtil.close(pstmt);
          DatabaseUtil.close(conn);
      }
   }
}
```

Testing InsertDate

Running the InsertDate.java program will generate the following output (the values are before insertion to TestDates):

```
$ javac InsertDate.java
$ java InsertDate
ok: loaded oracle driver.
sqlDate=2002-11-24
sqlTime=23:05:52
sqlTimestamp=2002-11-24 23:05:52.717
```

Oracle Date and Time Types: Retrieving an Existing Record

The following program retrieves an existing record from the `TestDates` table (with an ID of "`id100`"). Here is the program listing:

```
import java.sql.*;
import jcb.util.DatabaseUtil;

public class GetDate {

   public static Connection getConnection() throws Exception {
      String driver = "oracle.jdbc.driver.OracleDriver";
      String url = "jdbc:oracle:thin:@localhost:1521:scorpian";
      Class.forName(driver);
      System.out.println("ok: loaded oracle driver.");
      return DriverManager.getConnection(url, "scott", "tiger");
   }

   public static void main(String args[]) {
      String GET_RECORD = "select date_column, time_column, "+
        "timestamp_column from TestDates where id = ?";
      ResultSet rs = null;
      Connection conn = null;
      PreparedStatement pstmt = null;
      try {
          conn = getConnection();
          pstmt = conn.prepareStatement(GET_RECORD);
          pstmt.setString(1, "id100");
          rs = pstmt.executeQuery();
          if (rs.next()) {
             java.sql.Date dbSqlDate = rs.getDate(1);
             java.sql.Time dbSqlTime = rs.getTime(2);
             java.sql.Timestamp dbSqlTimestamp = rs.getTimestamp(3);
             System.out.println("dbSqlDate="+dbSqlDate);
             System.out.println("dbSqlTime="+dbSqlTime);
             System.out.println("dbSqlTimestamp="+dbSqlTimestamp);
          }
      }
      catch( Exception e ) {
         e.printStackTrace();
         System.out.println("Failed to insert the record.");
         System.exit(1);
      }
      finally {
          DatabaseUtil.close(rs);
          DatabaseUtil.close(pstmt);
          DatabaseUtil.close(conn);
      }
   }
}
```

Testing GetDate

Running the `GetDate.java` program will generate the following output (the values are after retrieval from `TestDates`):

```
$ javac GetDate.java
$ java GetDate
ok: loaded oracle driver.
dbSqlDate=2002-11-24
dbSqlTime=23:05:52
dbSqlTimestamp=2002-11-24 23:05:52.0
```

As you can observe, whatever values you deposited into the database, you were able to get back correctly (with the exception of `Timestamp`, which differs in value, so this is negligible).

9-5. How Do You Get the Current Date As a java.util.Date Object?

The class `java.util.Date` represents a specific instant in time, with millisecond precision. The following method returns the current date as a `java.util.Date` object:

```
/**
 * Get current date as a java.util.Date object
 * @return current date as a java.util.Date object
 */
public static java.util.Date getJavaUtilDate() {
    java.util.Date date = new java.util.Date();
    return date;
}
```

9-6. How Do You Create a java.sql.Date Object?

The class `java.sql.Date` descends from the `java.util.Date` class but uses only the year, month, and day values. The JDK defines `java.sql.Date` as follows:

```
public class java.sql.Date
    extends java.util.Date
```

Also, the JDK says this about `java.sql.Date`:

> *A thin wrapper around a millisecond value that allows JDBC to identify this as an SQL DATE value. A milliseconds value represents the number of milliseconds that have passed since January 1, 1970 00:00:00.000 GMT. To conform with the definition of SQL DATE, the millisecond values wrapped by a java.sql.Date instance must be "normalized" by setting the hours, minutes, seconds, and milliseconds to zero in the particular time zone with which the instance is associated.*

You have several ways to create a `java.sql.Date` object.

Creating java.sql.Date Using java.util.Date

This is how you create a `java.sql.Date` object using `java.util.Date`:

```
public static java.sql.Date getJavaSqlDate() {
    java.util.Date today = new java.util.Date();
    return new java.sql.Date(today.getTime());
}
```

Creating java.sql.Date Using java.util.Calendar

The following is according to the JDK:

> *The Calendar class is an abstract class that provides methods for converting between a specific instant in time and a set of calendar fields such as YEAR, MONTH, DAY_OF_MONTH, HOUR, and so on, and for manipulating the calendar fields, such as getting the date of the next week. An instant in time can be represented by a millisecond value that is an offset from the epoch, January 1, 1970, 00:00:00.000 GMT (Gregorian).*

Using the `Calendar` object, you can set the year, month, and day portions to the desired values. But you must set the hour, minute, second, and millisecond values to zero. At that point, `Calendar.getTime().getTime()` is invoked to get the `java.util.Date` in milliseconds. That value is then passed to a `java.sql.Date` constructor.

```
// get a calendar using the default time zone and locale.
Calendar calendar = Calendar.getInstance();

// set Date portion to January 1, 1970
calendar.set(Calendar.YEAR, 1970 );
calendar.set(Calendar.MONTH, Calendar.JANUARY );
calendar.set(Calendar.DATE, 1 );

// normalize the object
calendar.set(Calendar.HOUR_OF_DAY, 0 );
calendar.set(Calendar.MINUTE, 0 );
calendar.set(Calendar.SECOND, 0 );
calendar.set(Calendar.MILLISECOND, 0 );

java.sql.Date javaSqlDate =
   new java.sql.Date( calendar.getTime().getTime() );
```

Creating java.sql.Date Using the java.sql.Date.valueOf() Method

You can use a static method (`valueOf()`) of the `java.sql.Date` class. The `java.sql.Date` object's `valueOf()` method accepts a `String`, which must be the date in JDBC time escape format: YYYY-MM-DD. For example, to create a `Date` object representing November 1, 2000, you would use this:

```
String date = "2000-11-01";
java.sql.Date javaSqlDate = java.sql.Date.valueOf(date);
```

The `java.sql.Date.valueOf()` method throws an `IllegalArgumentException` (a runtime exception) if the given date is not in the JDBC date escape format (YYYY-MM-DD).

Creating java.sql.Date Using GregorianCalendar

`GregorianCalendar` is a concrete subclass of `Calendar` and provides the standard calendar system used by most of the world. You may use the year, month, and day of month to construct a `GregorianCalendar` with the given date set in the default time zone with the default locale. For example, to create a `Date` object representing November 1, 2000, you would use this:

```
// month value is 0-based. e.g., 0 for January.
Calendar calendar = new GregorianCalendar
                          (2000, // year
                           10,   // month
                           1);   // day of month
java.sql.Date date = new java.sql.Date(calendar.getTimeInMillis());
```

9-7. How Do You Get the Current Timestamp As a java.sql.Timestamp Object?

The following methods return a current `java.sql.Timestamp` object.

Getting a Timestamp Object Using java.util.Date

This shows how to get a `Timestamp` object using `java.util.Date`:

```
/**
 * Return a Timestamp for right now
 * @return Timestamp for right now
 */
public static java.sql.Timestamp getJavaSqlTimestamp() {
    java.util.Date today = new java.util.Date();
    return new java.sql.Timestamp(today.getTime());
}
```

Getting a Timestamp Object Using System.currentTimeMillis()

This shows how to get a `Timestamp` object using `System.currentTimeMillis()`:

```
/**
 * Return a Timestamp for right now
 * @return Timestamp for right now
 */
public static java.sql.Timestamp nowTimestamp() {
    return new java.sql.Timestamp(System.currentTimeMillis());
}
```

9-8. How Do You Get the Current Timestamp As a java.sql.Time Object?

The following method returns a current `java.sql.Time` object:

```
public static java.sql.Time getJavaSqlTime() {
    java.util.Date today = new java.util.Date();
    return new java.sql.Time(today.getTime());
}
```

9-9. How Do You Convert from a java.util.Date Object to a java.sql.Date Object?

The following code snippet shows how to convert from a `java.util.Date` object to a `java.sql.Date` object:

```
java.util.Date utilDate = new java.util.Date();
java.sql.Date sqlDate =  new java.sql.Date(utilDate.getTime());
System.out.println("utilDate:" + utilDate));
System.out.println("sqlDate:" + sqlDate));
```

This shows how to do the preceding as a method:

```
/**
  * Creates a java.sql.Date from a java.util.Date
  * return null if the parameter was null
  * @param utilDate the date to turn into the java.sql.Date object
```

```
  * @return the date or null if the utilDate was null
  */
 public static java.sql.Date sqlDate(java.util.Date utilDate) {
   if (utilDate == null) {
       return null;
   }
   else {
       return new java.sql.Date(utilDate.getTime());
   }
 }
```

9-10. How Do You Convert a String Date Such As 2003/01/10 into a java.util.Date Object?

When you want to convert a date expressed as a `String` object, you can use `java.text.SimpleDateFormat` for formatting purposes, like so:

```
import java.text.SimpleDateFormat;
...
try {
     SimpleDateFormat formatter = new SimpleDateFormat("yyyy/MM/dd");
     String date = "2003/01/10";
     java.util.Date utilDate  = formatter.parse(date);
     System.out.println("date:" + date));
     System.out.println("utilDate:" + utilDate));
 }
 catch (ParseException e) {
     System.out.println(e.toString());
     e.printStackTrace();
 }
```

This shows how to do the preceding as a method:

```
import java.text.SimpleDateFormat;
...
/**
 * Make a java.util.Date from a date as a string format of MM/DD/YYYY.
 * @param dateString date as a string of MM/DD/YYYY format.
 * @return a java.util.Date; if input is null return null;
 * @exception throws ParseException if the input is not valid format.
 */
public static java.util.Date makeDate(String dateString)
    throws ParseException {
    if ((dateString == null) || (dateString.length() == 0)) {
        return null;
    }
    SimpleDateFormat formatter = new SimpleDateFormat("MM/dd/yyyy");
    return formatter.parse(dateString);
}
```

9-11. How Do You Create Yesterday's Date from a Date in the String Format of MM/DD/YYYY?

The following code listing creates yesterday's date from a `Date` in the `String` format of MM/DD/YYYY:

```
/**
 * Create yesterday's date (as java.util.Date) from a date as a string
 * format of MM/DD/YYYY. If input is 03/01/2000, then it will
```

```
 * return a date with string value of 02/29/2000; and if your input
 * is 03/01/1999, then it will return a date with string value of
 * 02/28/1999.
 * @param dateString date as a string of MM/DD/YYYY format.
 * @return a yesterday's date (as java.util.Date); if input is null
 * return null;
 * @exception throws ParseException if the input is not valid format.
 */
 public static java.util.Date makeYesterdayDate(String dateString)
    throws ParseException {

    if ((dateString == null) || (dateString.length() == 0)) {
        return null;
    }

    SimpleDateFormat sdf = new SimpleDateFormat("MM/dd/yyyy");
    GregorianCalendar gc = new GregorianCalendar();
    // the following stmt. will throw a ParseException
    // if dateString does not have a valid format.
    java.util.Date d = sdf.parse(dateString);
    gc.setTime(d);
    System.out.println("Input Date = " + sdf.format(d));
    int dayBefore = gc.get(Calendar.DAY_OF_YEAR);
    gc.roll(Calendar.DAY_OF_YEAR, -1);
    int dayAfter = gc.get(Calendar.DAY_OF_YEAR);
    if(dayAfter > dayBefore) {
        gc.roll(Calendar.YEAR, -1);
    }
    gc.get(Calendar.DATE);
    java.util.Date yesterday = gc.getTime();
    System.out.println("Yesterdays Date = " + sdf.format(yesterday));
    return yesterday;
}
```

9-12. How Do You Create a java.util.Date Object from a Year, Month, Day Format?

The following code listing creates a `java.util.Date` object from a Year, Month, and Day format:

```
import java.text.SimpleDateFormat;
import java.util.Date;
...
/**
 * Make a java.util.Date from Year, Month, Day.
 * @param year the year
 * @param month the month
 * @param day the day
 * @throws ParseException failed to make a date.
 */
public static Date getJavaUtilDate(int year, int month, int day)
    throws ParseException {

    String date = new String(year) +"/"+ new String(month) +"/"+ new String(day);
    java.util.Date  utilDate = null;
```

```
    try {
           SimpleDateFormat formatter = new SimpleDateFormat("yyyy/MM/dd");
           utilDate  = formatter.parse(date);
           System.out.println("utilDate:" + utilDate));
           return utilDate;
    }
    catch (ParseException e) {
           System.out.println(e.toString());
           e.printStackTrace();
    }
}
```

9-13. How Do You Convert a String Date Such As 2003/01/10 to a java.sql.Date Object?

The following shows how to convert a `String` date such as 2003/01/10 into a `java.sql.Date` object:

```
import java.text.SimpleDateFormat;
...
try {
   SimpleDateFormat formatter = new SimpleDateFormat("yyyy/MM/dd");
   String  date  = "2003/01/10";
   java.util.Date utilDate  = formatter.parse(date);
   java.sql.Date  sqlDate = new java.sql.Date(util.getTime());
   System.out.println("date:" + date));
   System.out.println("sqlDate:" + sqlDate));
}
catch (ParseException e) {
   System.out.println(e.toString());
   e.printStackTrace();
}
```

This shows how to do the preceding as a method using `getJavaSqlDate()`:

```
import java.text.SimpleDateFormat;
...
/**
 * Make a java.sql.Date from (year, month, day).
 * @param year the year
 * @param month the month
 * @param day the day
 * @throws ParseException failed to make a date.
 */
public static java.sql.Date getJavaSqlDate(int year, int month, int day)
    throws ParseException {
    String date = new String(year) +"/"+ new String(month) +"/"+ new String(day);
    java.sql.Date  sqlDate = null;
    try {
           SimpleDateFormat formatter = new SimpleDateFormat("yyyy/MM/dd");
           java.util.Date utilDate  = formatter.parse(date);
           sqlDate = new java.sql.Date(util.getTime());
           System.out.println("sqlDate:" + sqlDate));
           return sqlDate;
    }
```

```
    catch (ParseException e) {
           System.out.println(e.toString());
           e.printStackTrace();
    }
}
```

9-14. How Do You Get a Timestamp Object?

This shows how to get a `Timestamp` object:

```
java.sql.Date date = new java.util.Date();
java.sql.Timestamp timestamp = new java.sql.Timestamp(date.getTime());
System.out.println("date: " + date);
System.out.println("Timestamp: " + timestamp);
```

This shows how to get a `Timestamp` object as a method:

```
import java.sql.Timestamp;
import java.util.Date;
...
/**
 * Creates a java.sql.Timestamp from a java.util.Date or
 * return null if the parameter was null
 * @param utilDate the date to turn into the SQL Timestamp object
 * @return the timestamp or null if the utilDate was null
 */
public static Timestamp getJavaSqlTimestamp(Date utilDate) {
   if (utilDate == null) {
      return null;
   }
   else {
      return new Timestamp(utilDate.getTime());
   }
}

/**
 * Creates a current java.sql.Timestamp
 * @return the timestamp
 */
public static Timestamp getJavaSqlTimestamp() {
  Date date = new Date();
  return new Timestamp(date.getTime());
}
```

9-15. How Do You Create a java.sql.Time Object?

`java.sql.Time` descends from `java.util.Date` but uses only the hour, minute, and second values. The JDK defines `java.sql.Time` as follows:

```
public class java.sql.Time extends java.util.Date
```

In addition, the JDK says this about `java.sql.Time`:

A thin wrapper around the java.util.Date class that allows the JDBC API to identify this as an SQL TIME value. The Time class adds formatting and parsing operations to support the JDBC escape syntax for time values. The date components should be set to the "zero epoch" value of January 1, 1970, and should not be accessed.

You have different ways to create a `java.sql.Time` object, which are presented next.

Creating a java.sql.Time Object from a java.util.Date Object

The following shows how to create a `java.sql.Time` object from a `java.util.Date` object:

```
/**
 * Creates a java.sql.Time from a java.util.Date or return null
 * if the parameter was null.
 * @param utilDate the date to turn into the SQL Time object
 * @return the time or null if the utilDate was null
 */
public static java.sql.Time getJavaSqlTime(java.util.Date utilDate) {
  if (utilDate == null) {
      return null;
  }
  else {
      return new java.sql.Time(utilDate.getTime());
  }
}
```

Creating a java.sql.Time Object from a java.util.Calendar Object

You can use a `Calendar` object by setting the year, month, and day portions to January 1, 1970, which is Java's zero epoch. The millisecond value must also be set to zero. At that point, `Calendar.getTime().getTime()` is invoked to get the time in milliseconds. That value is then passed to a `java.sql.Time` constructor.

```
Calendar cal = Calendar.getInstance();

// set Date portion to January 1, 1970
cal.set( Calendar.YEAR, 1970 );
cal.set( Calendar.MONTH, Calendar.JANUARY );
cal.set( Calendar.DATE, 1 );

// set milliseconds portion to 0
cal.set( Calendar.MILLISECOND, 0 );

java.sql.Time javaSqlTime = new java.sql.Time( cal.getTime().getTime() );
```

Creating a java.sql.Time Object Using java.sql.Time.valueOf()

The `java.sql.Time` object's `valueOf()` method accepts a `String`, which must be the time in JDBC time escape format—HH:MM:SS. For example, to create a `java.sql.Time` object for 9:23 p.m., you can write this:

```
java.sql.Time javaSqlTime = java.sql.Time.valueOf( "21:23:00" );
```

Creating a java.sql.Time Object Using the java.lang.System Class

The following shows how to create a `java.sql.Time` object using the `java.lang.System` class:

```
java.sql.Time tm = new java.sql.Time(System.currentTimeMillis());
```

9-16. How Do You Convert the Current Time to a java.sql.Date Object?

The following shows how to convert the current time to a `java.sql.Date` object:

```
import java.util.Calendar;
import java.sql.Date;
...
Calendar currenttime = Calendar.getInstance();
Date sqldate = new Date((currenttime.getTime()).getTime());
or as a method:

import java.util.Calendar;
import java.sql.Date;
...
public static Date  getCurrentJavaSqlDate () {
    Calendar currenttime = Calendar.getInstance();
    return new Date((currenttime.getTime()).getTime());
}
```

9-17. How Do You Determine the Day of the Week from Today's Date?

The answer to this question is to use `java.util.Calendar.SUNDAY`, `java.util.Calendar.MONDAY`, and so on:

```
public static int getDayOfWeek() {
    java.util.Date today = new java.util.Date();
    java.sql.Date date = new java.sql.Date(today.getTime());
    java.util.GregorianCalendar cal = new java.util.GregorianCalendar();
    cal.setTime(date);
    return cal.get(java.util.Calendar.DAY_OF_WEEK);
}
```

9-18. How Do You Determine the Day of the Week from a Given java.util.Date Object?

The answer to this question is to use `java.util.Calendar.SUNDAY`, `java.util.Calendar.MONDAY`, and so on:

```
public static int getDayOfWeek(java.util.Date utilDate)
    throws Exception {

    if (utilDate == null) {
        throw new Exception("date can not be null.");
    }
    java.sql.Date d = new java.sql.Date(utilDate.getTime());
    java.util.GregorianCalendar cal = new java.util.GregorianCalendar();
    cal.setTime(d);
    return cal.get(java.util.Calendar.DAY_OF_WEEK);
}
```

9-19. How Do You Convert java.sql.Date to java.util.Date?

The following shows how to convert `java.sql.Date` to `java.util.Date`:

```
public java.util.Date convert(java.sql.Date source) {
    if (source == null) {
        return null;
    }
    return new java.util.Date(source.getTime());
}
```

9-20. What Is java.text.SimpleDateFormat?

According to J2SE 5.0, `SimpleDateFormat` extends the `DateFormat` class and is defined as follows:

> *SimpleDateFormat is a concrete class for formatting and parsing dates in a locale-sensitive manner. It allows for formatting (date -> text), parsing (text -> date), and normalization. SimpleDateFormat allows you to start by choosing any user-defined patterns for date-time formatting. However, you are encouraged to create a date-time formatter with either getTimeInstance, getDateInstance, or getDateTimeInstance in DateFormat. Each of these class methods can return a date/time formatter initialized with a default format pattern. You may modify the format pattern using the applyPattern methods as desired. For more information on using these methods, see DateFormat.*

Table 9-4 lists some useful format strings for `java.text.SimpleDateFormat`.

Table 9-4. *Sample Format Strings for* `SimpleDateFormat`

Standard	String Format	Purpose
ISO 8160	`yyyy-MM-dd'T'H:mm:ss'Z'`	Used in EDI/OBI data, and so on
ISO 8610	`yyyyMMdd'T'HH:mm:ss`	Used in XML-RPC
SQL `Date`	`yyyy.MM.dd`	JDBC date format

9-21. How Do You Convert java.util.Date to a Date String in the Format MM/DD/YYYY?

The following shows how to convert `java.util.Date` to a `Date` string in the format MM/DD/YYYY:

```
import java.util.Calendar;
import java.util.Date;
...
/**
 * Convert java.util.Date to a date String in the format MM/DD/YYYY
 * @param date the java.util.Date
 * @return a date String in the format MM/DD/YYYY
 * (if input is null, then return null).
 */
public static String toDateString(Date date) {
    if (date == null) {
        return null;
    }

    Calendar calendar = Calendar.getInstance();
    calendar.setTime(date);
    int month = calendar.get(Calendar.MONTH) + 1;
    int day = calendar.get(Calendar.DAY_OF_MONTH);
    int year = calendar.get(Calendar.YEAR);
```

```
    String monthString;
    if (month < 10) {
        monthString = "0" + month;
    }
    else {
        monthString = "" + month;
    }

    String dayString;
    if (day < 10) {
        dayString = "0" + day;
    }
    else {
        dayString = "" + day;
    }

    String yearString = "" + year;
    return monthString + "/" + dayString + "/" + yearString;
}
```

9-22. How Do You Create a Time String in the Format HH:MM:SS from an Hour, Minute, Second Format?

The following shows how to make a `Time` string in the format HH:MM:SS from an Hour, Minute, Second format:

```
/**
 * Makes a time String in the format HH:MM:SS from separate ints
 * for hour, minute, and second. If the seconds are 0, then the output
 * is in HH:MM. It is assumed that all of the input arguments will have
 * proper values.
 * @param hour the hours as integer
 * @param minute the minutes as integer
 * @param second the seconds integer
 * @return a time String in the format HH:MM:SS or HH:MM
 */
public static String toTimeString(int hour, int minute, int second) {

    String hourString;
    if (hour < 10) {
        hourString = "0" + hour;
    }
    else {
        hourString = "" + hour;
    }

    String minuteString;
    if (minute < 10) {
        minuteString = "0" + minute;
    }
    else {
        minuteString = "" + minute;
    }
    String secondString;
    if (second < 10) {
        secondString = "0" + second;
    }
```

```
    else {
        secondString = "" + second;
    }

    if (second == 0) {
        return hourString + ":" + minuteString;
    }
    else {
        return hourString + ":" + minuteString + ":" + secondString;
    }
}
```

9-23. How Do You Convert a java.util.Date Object to a Time String in the Format HH:MM:SS?

This method uses the `toTimeString(int hour, int minute, int second)` method defined previously:

```
import java.util.Date;
import java.util.Calendar;
...
/**
 * Convert java.util.Date to a time String in the format HH:MM:SS.
 *  If the seconds are 0, then the output is in HH:MM format.
 * @param date The java.util.Date
 * @return a time String in the format HH:MM:SS or HH:MM
 */
public static String toTimeString(Date date) {
    if (date == null)
        return null;
    }

    Calendar calendar = Calendar.getInstance();
    calendar.setTime(date);
    return (toTimeString(calendar.get(Calendar.HOUR_OF_DAY),
                         calendar.get(Calendar.MINUTE),
                         calendar.get(Calendar.SECOND)));
}
```

9-24. How Do You Check for a Leap Year?

The following methods check whether a given year is a leap year. A year is a leap year if it is an even multiple of 4; however, years divisible by 100 but not by 400 aren't leap years. For example, 1900 isn't a leap year, but 1600 and 2000 both are. (An alternate definition of a leap year is that a specific year is a leap year if it is either evenly divisible by 400 or evenly divisible by 4 and not evenly divisible by 100.)

You can check whether a given year is a leap year in one of two ways.

Solution 1: Checking for a Leap Year

This checks for a leap year:

```
/**
 * Checks if a year is a leap year. If input is
 * a negative integer, then it returns false.
```

```
 * @param year The year to check.
 * @return true: the year is a leap year;
 * false: the year is a normal year.
 */
public static boolean isLeapYear(int year) {

    if (year < 0) {
        return false;
    }

    if (year % 400 == 0) {
        return true;
    }
    else if (year % 100 == 0) {
        return false;
    }
    else if (year % 4 == 0) {
        return true;
    }
    else {
        return false;
    }
}
```

Solution 2: Checking for a Leap Year

This also checks for a leap year:

```
import java.util.GregorianCalendar;
...
/**
 * Determining If a Year Is a Leap Year. If input is
 * a negative integer, then it returns false.
 * @param year The year to check.
 * @return true: the year is a leap year;
 * false: the year is a normal year.
 */
public static boolean isLeapYear(int year) {

    if (year < 0) {
        return false;
    }

    GregorianCalendar gcal = new GregorianCalendar();
    return gcal.isLeapYear(year);
}
```

9-25. How Do You Convert Between Different Java Date Classes?

Table 9-5 shows the conversion of Java `Date` classes, and Table 9-6 shows the conversion of the Java `Date` class to `Calendar`.

Table 9-5. *Conversion of Java* Date *Classes*

From/To...	java.util.Date	java.sql.Date
java.util.Date		to.setTime(from.getTime())
java.sql.Date	to.setTime(from.getTime())	
java.util.Calendar	to = from.getTime()	to.setTime(from.getTime(). getTime())
long (milliseconds)	to.setTime(from)	to.setTime(from)

Table 9-6. *Conversion of Java* Date *Classes to* Calendar

From/To	java.util.Calendar	long (Milliseconds)
java.util.Date	to.setTime(from)	to = from.getTime()
java.sql.Date	to.setTimeInMillis(from.getTime())	to = from.getTime()
java.util.Calendar		to = from.getTime(). getTime()
long (milliseconds)	to.setTimeInMillis(from)	

9-26. How Do You Add/Subtract Days for a Given Date (Given As a String)?

The following shows how to add/subtract days for a given date (given as a String):

```
import java.util.Date;
import java.util.Calendar;
import java.text.SimpleDateFormat;
import java.text.ParseException;
...
/**
 * Add/Subtract days for a given date given as a string
 * format of MM/DD/YYYY. If input is 03/01/2000, and delta is -1,
 * then it will return a date with string value of 02/29/2000;
 * and if your input is 03/01/1999, and delta is -1, then it will
 * return a date with string value of 02/28/1999.
 * @param dateString date as a string of MM/DD/YYYY format.
 * @param delta the number of days to add/subtract
 * @return a new date (as java.util.Date) based on delta days; if
 * input is null return null;
 * @exception throws ParseException is the input is not valid format.
 */
 public static Date makeDate(String dateString, int delta)
    throws ParseException {

    if ((dateString == null) || (dateString.length() == 0)) {
        return null;
    }

    SimpleDateFormat formatter = new SimpleDateFormat("MM/dd/yyyy");

    // the following stmt. will throw a ParseException
    // if dateString does not have a valid format.
    Date d = formatter.parse(dateString);
    System.out.println("originalDate="+formatter.format(d));
```

```
    if (delta == 0) {
        return d;
    }

    Calendar cal = Calendar.getInstance();
    cal.setTime(d);
    cal.add(Calendar.DATE ,delta);
    Date revisedDate = cal.getTime();
    System.out.println("revisedDate="+formatter.format(revisedDate));
    return revisedDate;
}
```

Testing makeDate()

The following shows how to test `makeDate()`:

```
java.util.Date revisedDate2 = makeDate("03/01/2000", -1);
System.out.println("revisedDate2="+revisedDate2);
System.out.println("---");
java.util.Date revisedDate3 = makeDate("03/01/2000", 10);
System.out.println("revisedDate3="+revisedDate3);
```

Viewing the Output of the Test

The following is the output of the test:

```
originalDate=03/01/2000
revisedDate=02/29/2000
revisedDate2=Tue Feb 29 00:00:00 PST 2000
---
originalDate=03/01/2000
revisedDate=03/11/2000
revisedDate3=Sat Mar 11 00:00:00 PST 2000
```

9-27. How Do You Find the Difference Between Two Given Dates?

Let's say that `d1` represents 1/10/2000 (as a `java.util.Date` object) and `d2` represents 2/31/2000 (as a `java.util.Date` object). How do you find the difference of `d1` and `d2`? In other words, how many days apart are these dates from each other? The answer is 52 days. I will provide the solution for finding the difference between two `java.util.Date` objects as well as for between two `java.sql.Date` objects. The following solution is a simple class called `DateDiff`.

Finding the Difference Between Two java.util.Date Objects

The following shows how to find the difference between two `java.util.Date` objects:

```
java.util.Date  d1 = DateDiff.makeDate("1/10/2000");
java.util.Date  d2 = DateDiff.makeDate("2/31/2000");
int diff = DateDiff.diff( d1, d2 );
System.out.println("d1="+d1);
System.out.println("d2="+d2);
System.out.println("diff="+diff);
output will be:
```

```
d1=Mon Jan 10 00:00:00 PST 2000
d2=Thu Mar 02 00:00:00 PST 2000
diff=52
```

Finding the Difference Between Two java.sql.Date Objects

The following shows how to find the difference between two `java.sql.Date` objects:

```
java.util.Date utilDate1 = DateDiff.makeDate("12/01/1990");
java.sql.Date sqlDate1 =  new java.sql.Date(utilDate1.getTime());
System.out.println("utilDate1:" + utilDate1);
System.out.println("sqlDate1:" + sqlDate1);

java.util.Date utilDate2 = DateDiff.makeDate("1/24/1991");
java.sql.Date sqlDate2 =  new java.sql.Date(utilDate2.getTime());
System.out.println("utilDate2:" + utilDate2);
System.out.println("sqlDate2:" + sqlDate2);

int diffSqlDates = diff( sqlDate1, sqlDate2 );
System.out.println("diffSqlDates="+diffSqlDates);
output will be:

utilDate1:Sat Dec 01 00:00:00 PST 1990
sqlDate1:1990-12-01
utilDate2:Thu Jan 24 00:00:00 PST 1991
sqlDate2:1991-01-24
Diffirent Day : 54
diffSqlDates=54
```

Viewing the DateDiff Class

This is the `DateDiff` class:

```
import java.util.*;
import java.text.*;

public class DateDiff {

    /**
     * Calculate the difference of two dates
     * (in terms of number of days).
     * @param date1 the java.util.Date object
     * @param date2 the java.util.Date object
     */
     public static int diff( Date date1, Date date2 ) {
         Calendar c1 = Calendar.getInstance();
         Calendar c2 = Calendar.getInstance();

         c1.setTime( date1 );
         c2.setTime( date2 );
         int diffDay = 0 ;

         if ( c1.before( c2 ) ) {
              diffDay = countDiffDay ( c1, c2 );
         }
         else {
              diffDay = countDiffDay ( c2, c1 );
         }
```

```
        return diffDay;
    }

    public DateDiff( Date date1, Date date2 ) {
        int diffDay = diff(date1, date2);
        System.out.println("Different Day : " + diffDay );
    }

    public static int countDiffDay( Calendar c1, Calendar c2 ) {
        int returnInt = 0;
        while ( !c1.after(c2) ) {
            c1.add( Calendar.DAY_OF_MONTH, 1 );
            returnInt ++;
        }

        if ( returnInt > 0 ) {
            returnInt = returnInt - 1;
        }

        return ( returnInt );
    }

    public static Date makeDate(String dateString)
        throws Exception {
        SimpleDateFormat formatter = new SimpleDateFormat("MM/dd/yyyy");
        return formatter.parse(dateString);
    }

    public static void main ( String argv[] ) throws Exception {
        Calendar cc1 = Calendar.getInstance();
        Calendar cc2 = Calendar.getInstance();
        cc1.add( Calendar.DAY_OF_MONTH, 10 );

        DateDiff myDate = new DateDiff( cc1.getTime(), cc2.getTime() );

        java.util.Date  d1 = makeDate("10/10/2000");
        java.util.Date  d2 = makeDate("10/18/2000");
        DateDiff diff12 = new DateDiff( d1, d2 );

        java.util.Date  d3 = makeDate("1/1/2000");
        java.util.Date  d4 = makeDate("12/31/2000");
        int diff34 = diff( d3, d4 );
        System.out.println("diff34="+diff34);

        java.util.Date  d5 = makeDate("1/10/2000");
        java.util.Date  d6 = makeDate("2/31/2000");
        int diff56 = diff( d5, d6 );
        System.out.println("d5="+d5);
        System.out.println("d6="+d6);
        System.out.println("diff56="+diff56);

        java.util.Date utilDate1 = DateDiff.makeDate("12/01/1990");
        java.sql.Date sqlDate1 =  new java.sql.Date(utilDate1.getTime());
        System.out.println("utilDate1:" + utilDate1);
        System.out.println("sqlDate1:" + sqlDate1);
```

```
        java.util.Date utilDate2 = DateDiff.makeDate("1/24/1991");
        java.sql.Date sqlDate2 =  new java.sql.Date(utilDate2.getTime());
        System.out.println("utilDate2:" + utilDate2);
        System.out.println("sqlDate2:" + sqlDate2);

        int diffSqlDates = diff( sqlDate1, sqlDate2 );
        System.out.println("diffSqlDates="+diffSqlDates);
    }
}
```

9-28. How Do You Convert a Timestamp to Month-Day-Year?

In Web/GUI applications, you do not need to display a complete `Timestamp` object (it is too long and may be unreadable). Some applications may want to display a `Timestamp` object as a Month-Day-Year string. Here is a class with three methods to accomplish this task:

```
import java.sql.Timestamp;
import java.text.SimpleDateFormat;

/**
 * DateUtil provides some basic methods
 * for formatting Timestamp objects.
 */
public class DateUtil {

    /**
     * SimpleDateFormat object to format Timestamp into Month-Day-Year String.
     */
     private static final SimpleDateFormat monthDayYearformatter =
            new SimpleDateFormat("MMMMM dd, yyyy");

    /**
     * SimpleDateFormat object to format Timestamp into "Month-Day" String.
     */
    private static final SimpleDateFormat monthDayformatter =
            new SimpleDateFormat("MMMMM dd");

    /**
     * Return Timestamp object as MMMMM DD, YYYY.
     * @param timestamp a Timestamp object
     * @return Timestamp object as MMMMM DD, YYYY.
     */
    public static String timestampToMonthDayYear(Timestamp timestamp){
        if (timestamp == null) {
            return null;
        }
        else {
            return monthDayYearformatter.format((java.util.Date) timestamp);
        }
    }

    /**
     * Return Timestamp object as MMMMM DD.
     * @param timestamp a Timestamp object
     * @return Timestamp object as MMMMM DD.
     */
```

```
    public static String timestampToMonthDay(Timestamp timestamp){
        if (timestamp == null) {
            return null;
        }
        else {
            return monthDayformatter.format((java.util.Date) timestamp);
        }
    }

    /**
     * Get the current timestamp.
     * @return the current timestamp.
     */
    public static java.sql.Timestamp getTimestamp() {
        java.util.Date today = new java.util.Date();
        return new java.sql.Timestamp(today.getTime());
    }
}
```

9-29. How Do You Determine the Validity of a Format Pattern for SimpleDateFormat?

A format pattern such as MMMMM DD, YYYY is a valid one, but MMMMM DD, YYYYZZ is not valid. The following method accepts a format pattern and then returns `true` if it is a valid format pattern; otherwise, it returns `false`:

```
/**
 * Tests if the date format pattern specified is valid.
 *
 * @param format The format string to test.
 *
 * @return true if the format parameter contains
 * a valid formatting string; false otherwise.
 */
public static boolean isValidDateFormat(String format) {
    if ((format == null) || ((format.length() == 0)) {
        // not a valid format pattern
        return false;
    }

    java.text.SimpleDateFormat formatter = null;
    try {
        formatter = new java.text.SimpleDateFormat( format );
        formatter.format( new java.util.Date() );
        return true;
    }
    catch(Exception e) {
        // not a valid format pattern
        return false;
    }
}
```

9-30. How Do You Get a Date Label from a java.sql.Timestamp Object?

In GUI database applications, you may need to convert a `Timestamp` object into a date label such as Today, Yesterday, This Month, and Older Than a Month. By getting a date label, the user can categorize data.

The following method gets a date label for a given `Timestamp` object:

```
import java.sql.Timestamp;

/**
 * DateLabel provides some basic methods
 * for formatting Date and Timestamp objects.
 */
public class DateLabel {

    private static final long One_Day_In_Milliseconds = 86400000;

    /**
     * This date label represents "Today".
     */
    public static final String DATE_LABEL_TODAY = "Today";

    /**
     * This date label represents "Yesterday".
     */
    public static final String DATE_LABEL_YESTERDAY = "Yesterday";

    /**
     * This date label represents "This Month".
     */
    public static final String DATE_LABEL_THIS_MONTH = "This Month";

    /**
     * This date label represents "Older" (older than a month).
     */
    public static final String DATE_LABEL_OLDER = "Older";

    /**
     * This date label represents "none" (when
     * timestamp is null/undefined).
     */
    public static final String DATE_LABEL_NONE = "";


    /**
     * Get the current timestamp.
     * @return the current timestamp.
     */
    public static java.sql.Timestamp getTimestamp() {
        java.util.Date today = new java.util.Date();
        return new java.sql.Timestamp(today.getTime());
    }

    /**
     * Get the Date Label.
     * @param ts the timestamp you want to get a data label
```

```
     * @param now the timestamp you want to compare to
     * @return the date label for a given timestamp.
     */
    public static String getDateLabel(java.sql.Timestamp ts,
                                      java.sql.Timestamp now) {
        if (ts == null) {
            return DATE_LABEL_NONE;
        }

        if (now == null) {
            return DATE_LABEL_NONE;
        }

        long tsTime = ts.getTime();
        long nowTime = now.getTime();
        long quotient = (nowTime - tsTime)/One_Day_In_Milliseconds;

        if (quotient < 1) {
            return DATE_LABEL_TODAY;
        }
        else if (quotient < 2) {
            return DATE_LABEL_YESTERDAY;
        }
        else if (quotient < 30) {
            return DATE_LABEL_THIS_MONTH;
        }
        else {
            return DATE_LABEL_OLDER;
        }
    }

    public static void main(String[] args) {
        java.sql.Timestamp now = getTimestamp();

        java.sql.Timestamp ts1 = getTimestamp();
        System.out.println(getDateLabel(ts1, now));
        System.out.println(ts1.toString());
        System.out.println("-------------");

        // timestamp in format yyyy-mm-dd hh:mm:ss.fffffffff
        java.sql.Timestamp ts22 =
            java.sql.Timestamp.valueOf("2005-04-06 09:01:10");
        System.out.println(getDateLabel(ts22, now));
        System.out.println(ts22.toString());
        System.out.println("-------------");

        java.sql.Timestamp ts2 =
            java.sql.Timestamp.valueOf("2005-03-26 10:10:10");
        System.out.println(getDateLabel(ts2, now));
        System.out.println(ts2.toString());
        System.out.println("-------------");

        java.sql.Timestamp ts3 =
            java.sql.Timestamp.valueOf("2004-07-18 10:10:10");
        System.out.println(getDateLabel(ts3, now));
        System.out.println(ts3.toString());
        System.out.println("-------------");
```

```
        java.sql.Timestamp ts4 =
            java.sql.Timestamp.valueOf("2004-06-20 10:10:10");
        System.out.println(getDateLabel(ts4, now));
        System.out.println(ts4.toString());
        System.out.println("-------------");
    }
}
```

To run the test program, use this:

```
$ javac DateLabel.java
$ java DateLabel
Today
2005-04-07 09:09:47.605
-------------
Yesterday
2005-04-06 09:01:10.0
-------------
This Month
2005-03-26 10:10:10.0
-------------
Older
2004-07-18 10:10:10.0
-------------
Older
2004-06-20 10:10:10.0
-------------
```

9-31. How Do You Convert a java.sql.Timestamp Object to a java.util.Date Object?

`java.sql.Timestamp` is a wrapper around `java.util.Date` that allows the JDBC API to identify it as a SQL `TIMESTAMP` value. This adds the ability to hold the SQL `TIMESTAMP` nanos value and provides formatting and parsing operations to support the JDBC escape syntax for timestamp values. The `java.sql.Timestamp` object stores the fractional part of the time within itself instead of within the `Date` superclass.

You can use the following to convert a `java.sql.Timestamp` object to a `java.util.Date` object:

```
public static java.util.Date toDate(java.sql.Timestamp timestamp) {
    if (timestamp == null) {
        return null;
    }

    long milliseconds  = timestamp.getTime() + (timestamp.getNanos() / 1000000 );
    return new java.util.Date(milliseconds);
}
```

9-32. What Does Normalization Mean for java.sql.Date and java.sql.Time?

To understand normalization, you need to look at the `java.sql.Date` and `java.sql.Time` classes. The class `java.sql.Date` descends from the `java.util.Date` class but uses only the year, month, and day values. JDK defines `java.sql.Date` and `java.sql.Time` as follows:

```
public class java.sql.Date extends java.util.Date
```

The JDK also says this:

A thin wrapper around a millisecond value that allows JDBC to identify this as an SQL DATE value. A milliseconds value represents the number of milliseconds that have passed since January 1, 1970 00:00:00.000 GMT. To conform with the definition of SQL DATE, the millisecond values wrapped by a java.sql.Date instance must be "normalized" by setting the hours, minutes, seconds, and milliseconds to zero in the particular time zone with which the instance is associated.

`public class java.sql.Time extends java.util.Date`

A thin wrapper around the java.util.Date class that allows the JDBC API to identify this as an SQL TIME value. The Time class adds formatting and parsing operations to support the JDBC escape syntax for time values. The date components should be set to the "zero epoch" value of January 1, 1970, and should not be accessed.

These classes (`java.sql.Date` and `java.sql.Time`) are thin wrappers that extend the `java.util.Date` class, which has both date and time components. `java.sql.Date` should carry only date information, and a normalized instance has the time information set to zeros. `java.sql.Time` should carry only time information; a normalized instance has the date set to the Java zero epoch (January 1, 1970) and the milliseconds portion set to zero.

The following sections give a complete example for determining the normalization of `java.sql.Time` and `java.sql.Date` objects.

9-33. Does MySQL/Oracle JDBC Driver Normalize java.sql.Date and java.sql.Time Objects?

Normalization depends on the JDBC driver's implementation of the `ResultSet.getDate()` and `ResultSet.getTime()` methods. To determine normalization, you must convert the `java.sql.Date` and `java.sql.Time` objects to an associated `java.util.Date` object.

How It Works

For example, if a `java.sql.Date` object displays 2005-07-01, it's normalized only if its associated `java.util.Date` value is as follows:

```
Fri Jul 01 00:00:00 EDT 2005
```

Further, if a `java.sql.Time` object displays 16:10:12, it's normalized only if its associated `java.util.Date` value is as follows:

```
Thu Jan 01 16:10:12 EST 1970
```

The following solution (the `TestNormalization` class) checks whether the MySQL/Oracle JDBC driver supports normalization for the `java.sql.Date` and `java.sql.Time` objects. Before reviewing this solution, you will see how to set up some database objects.

Setting Up the MySQL Database

This shows how to set up the MySQL database:

```
mysql> create table date_time_table (
    ->  time_col time,
    ->  date_col date,
```

```
    ->  date_time_col datetime
    -> );
Query OK, 0 rows affected (0.07 sec)

mysql> desc date_time_table;
+---------------+----------+------+-----+---------+-------+
| Field         | Type     | Null | Key | Default | Extra |
+---------------+----------+------+-----+---------+-------+
| time_col      | time     | YES  |     | NULL    |       |
| date_col      | date     | YES  |     | NULL    |       |
| date_time_col | datetime | YES  |     | NULL    |       |
+---------------+----------+------+-----+---------+-------+
3 rows in set (0.00 sec)

mysql> insert into date_time_table(time_col, date_col, date_time_col)
values ('10:34:55', '2004-10-23', '2004-10-23 10:34:55');
Query OK, 1 row affected (0.00 sec)

mysql> insert into date_time_table(time_col, date_col, date_time_col)
values ('16:12:50', '2005-07-01', '2005-07-01 16:12:50');
Query OK, 1 row affected (0.00 sec)

mysql> select * from date_time_table;
+----------+------------+---------------------+
| time_col | date_col   | date_time_col       |
+----------+------------+---------------------+
| 10:34:55 | 2004-10-23 | 2004-10-23 10:34:55 |
| 16:12:50 | 2005-07-01 | 2005-07-01 16:12:50 |
+----------+------------+---------------------+
2 rows in set (0.00 sec)
```

Setting Up the Oracle Database

This shows how to set up the Oracle database:

```
SQL> create table date_time_table (
  2    time_col date,
  3    date_col date,
  4    date_time_col date
  5  );

Table created.

SQL> desc date_time_table;
 Name                  Null?    Type
 --------------------- -------- -------
 TIME_COL                       DATE
 DATE_COL                       DATE
 DATE_TIME_COL                  DATE

SQL> insert into date_time_table(time_col, date_col, date_time_col)
  2  values (sysdate, sysdate, sysdate);

1 row created.

SQL> select * from date_time_table;
```

```
TIME_COL  DATE_COL  DATE_TIME
--------- --------- ---------
01-JUL-05 01-JUL-05 01-JUL-05

SQL> commit;
Commit complete.
```

Solution

This shows the solution:

```
import java.sql.*;
import jcb.util.DatabaseUtil;
import jcb.db.VeryBasicConnectionManager;

public class TestNormalization {

   public static void main(String args[]) {
      String GET_RECORDS =
        "select time_col, date_col, date_time_col from date_time_table";
      ResultSet rs = null;
      Connection conn = null;
      Statement stmt = null;
      try {
          String dbVendor = args[0];  // {"mysql", "oracle" }
          conn = VeryBasicConnectionManager.getConnection(dbVendor);
          System.out.println("conn="+conn);
          stmt = conn.createStatement();
          rs = stmt.executeQuery(GET_RECORDS);
          while (rs.next()) {
             java.sql.Time dbSqlTime = rs.getTime(1);
             java.sql.Date dbSqlDate = rs.getDate(2);
             java.sql.Timestamp dbSqlTimestamp = rs.getTimestamp(3);
             System.out.println("dbSqlTime="+dbSqlTime);
             System.out.println("dbSqlDate="+dbSqlDate);
             System.out.println("dbSqlTimestamp="+dbSqlTimestamp);
             System.out.println("-- check for Normalization --");
             java.util.Date dbSqlTimeConverted =
                 new java.util.Date(dbSqlTime.getTime());
             java.util.Date dbSqlDateConverted =
                 new java.util.Date(dbSqlDate.getTime());
             System.out.println("dbSqlTimeConverted="+dbSqlTimeConverted);
             System.out.println("dbSqlDateConverted="+dbSqlDateConverted);
          }
      }
      catch( Exception e ) {
         e.printStackTrace();
         System.out.println("Failed to get the records.");
         System.exit(1);
      }
      finally {
          DatabaseUtil.close(rs);
          DatabaseUtil.close(stmt);
          DatabaseUtil.close(conn);
      }
   }
}
```

Running the Solution for the MySQL Database

As you can see from the following results, the MySQL driver does support normalization for the `java.sql.Date` and `java.sql.Time` objects:

```
$ javac TestNormalization.java
$ java TestNormalization mysql
ok: loaded mysql driver.
conn=com.mysql.jdbc.Connection@1e4cbc4
dbSqlTime=10:34:55
dbSqlDate=2004-10-23
dbSqlTimestamp=2004-10-23 10:34:55.0
-- check for Normalization --
dbSqlTimeConverted=Thu Jan 01 10:34:55 PST 1970
dbSqlDateConverted=Sat Oct 23 00:00:00 PDT 2004
dbSqlTime=16:12:50
dbSqlDate=2005-07-01
dbSqlTimestamp=2005-07-01 16:12:50.0
-- check for Normalization --
dbSqlTimeConverted=Thu Jan 01 16:12:50 PST 1970
dbSqlDateConverted=Fri Jul 01 00:00:00 PDT 2005
```

Running Solution for the Oracle Database

As you can see from the following results, the Oracle 10*g* driver does support normalization for the `java.sql.Date` and `java.sql.Time` objects:

```
$ java TestNormalization oracle
ok: loaded oracle driver.
conn=oracle.jdbc.driver.OracleConnection@6e70c7
dbSqlTime=16:15:22
dbSqlDate=2005-07-01
dbSqlTimestamp=2005-07-01 16:15:22.0
-- check for Normalization --
dbSqlTimeConverted=Thu Jan 01 16:15:22 PST 1970
dbSqlDateConverted=Fri Jul 01 00:00:00 PDT 2005
```

9-34. How Do You Make a java.sql.Timestamp Object for a Given Year, Month, Day, Hour, and So On?

Given the year, month, day, hour, minutes, seconds, and milliseconds, the objective of the following code is to create a `java.sql.Timestamp` object:

```
import java.sql.Timestamp;
import java.util.Calendar;
import java.util.GregorianCalendar;
...
/**
 * Given year, month, day, hour, minutes, seconds, and
 * milliseconds, the objective is to create a Timestamp object.
 * @param year the year
 * @param month the month
 * @param day the day
 * @param hour the hour
```

```
 * @param minute the minute
 * @param second the second
 * @param millisecond the millisecond
 * @return a java.sql.Timestamp object
 */
public static Timestamp makeTimestamp(int year,
                                      int month,
                                      int day,
                                      int hour,
                                      int minute,
                                      int second,
                                      int millisecond) {
   Calendar cal = new GregorianCalendar();
   cal.set(Calendar.YEAR, year);
   cal.set(Calendar.MONTH, month - 1);
   cal.set(Calendar.DATE, day);
   cal.set(Calendar.HOUR_OF_DAY, hour);
   cal.set(Calendar.MINUTE, minute);
   cal.set(Calendar.SECOND, second);
   cal.set(Calendar.MILLISECOND, millisecond);

   // now convert GregorianCalendar object to Timestamp object
   return new Timestamp(cal.getTimeInMillis());
}
```

9-35. How Do You Get a Date for a Specific Time Zone?

The following Java code fragment illustrates how to use a `Calendar` object to retrieve a date for Los Angeles, California:

```
import java.sql.Date;
import java.sql.ResultSet;
import java.util.Calendar;
import java.util.TimeZone;
...
ResultSet rs = stmt.executeQuery(
   "SELECT date_created FROM products WHERE product_id = 'PRD-123456'");

//creating an instance of Calendar
Calendar cal = Calendar.getInstance();

// get the TimeZone for "America/Los_Angeles"
TimeZone tz = TimeZone.getTimeZone("America/Los_Angeles");
cal.setTimeZone(tz);
if (rs.next()) {
   // the JDBC driver will use the time zone information in
   // Calendar to calculate the date, with the result that
   // the variable dateCreated contains a java.sql.Date object
   // that is accurate for "America/Los_Angeles".
   Date dateCreated = rs.getDate(1, cal);
}
```