JDBC - DATA TYPES

The JDBC driver converts the Java data type to the appropriate JDBC type, before sending it to the database. It uses a default mapping for most data types. For example, a Java int is converted to an SQL INTEGER. Default mappings were created to provide consistency between drivers.

The following table summarizes the default JDBC data type that the Java data type is converted to, when you call the setXXX method of the PreparedStatement or CallableStatement object or the ResultSet.updateXXX method.

SQL	**JDBC/Java**	**setXXX**	**updateXXX**
VARCHAR	java.lang.String	setString	updateString
CHAR	java.lang.String	setString	updateString
LONGVARCHAR	java.lang.String	setString	updateString
BIT	boolean	setBoolean	updateBoolean
NUMERIC	java.math.BigDecimal	setBigDecimal	updateBigDecimal
TINYINT	byte	setByte	updateByte
SMALLINT	short	setShort	updateShort
INTEGER	int	setInt	updateInt
BIGINT	long	setLong	updateLong
REAL	float	setFloat	updateFloat
FLOAT	float	setFloat	updateFloat
DOUBLE	double	setDouble	updateDouble
VARBINARY	byte[ ]	setBytes	updateBytes
BINARY	byte[ ]	setBytes	updateBytes
DATE	java.sql.Date	setDate	updateDate
TIME	java.sql.Time	setTime	updateTime
TIMESTAMP	java.sql.Timestamp	setTimestamp	updateTimestamp
CLOB	java.sql.Clob	setClob	updateClob
BLOB	java.sql.Blob	setBlob	updateBlob
ARRAY	java.sql.Array	setARRAY	updateARRAY
REF	java.sql.Ref	SetRef	updateRef
STRUCT	java.sql.Struct	SetStruct	updateStruct

JDBC 3.0 has enhanced support for BLOB, CLOB, ARRAY, and REF data types. The ResultSet object now has updateBLOB, updateCLOB, updateArray, and updateRef methods that enable you to directly manipulate the respective data on the server.

The setXXX and updateXXX methods enable you to convert specific Java types to specific JDBC data types. The methods, setObject and updateObject, enable you to map almost any Java type to a JDBC data type.

ResultSet object provides corresponding getXXX method for each data type to retrieve column value. Each method can be used with column name or by its ordinal position.

SQL	**JDBC/Java**	**setXXX**	**getXXX**
VARCHAR	java.lang.String	setString	getString
CHAR	java.lang.String	setString	getString
LONGVARCHAR	java.lang.String	setString	getString
BIT	boolean	setBoolean	getBoolean
NUMERIC	java.math.BigDecimal	setBigDecimal	getBigDecimal
TINYINT	byte	setByte	getByte
SMALLINT	short	setShort	getShort
INTEGER	int	setInt	getInt
BIGINT	long	setLong	getLong
REAL	float	setFloat	getFloat
FLOAT	float	setFloat	getFloat
DOUBLE	double	setDouble	getDouble
VARBINARY	byte[ ]	setBytes	getBytes
BINARY	byte[ ]	setBytes	getBytes
DATE	java.sql.Date	setDate	getDate
TIME	java.sql.Time	setTime	getTime
TIMESTAMP	java.sql.Timestamp	setTimestamp	getTimestamp
CLOB	java.sql.Clob	setClob	getClob
BLOB	java.sql.Blob	setBlob	getBlob
ARRAY	java.sql.Array	setARRAY	getARRAY
REF	java.sql.Ref	SetRef	getRef
STRUCT	java.sql.Struct	SetStruct	getStruct

Date & Time Data Types

The java.sql.Date class maps to the SQL DATE type, and the java.sql.Time and java.sql.Timestamp classes map to the SQL TIME and SQL TIMESTAMP data types, respectively.

Following example shows how the Date and Time classes format the standard Java date and time values to match the SQL data type requirements.

```
import java.sql.Date;
import java.sql.Time;
import java.sql.Timestamp;
import java.util.*;

public class SqlDateTime {
   public static void main(String[] args) {
      //Get standard date and time
      java.util.Date javaDate = new java.util.Date();
      long javaTime = javaDate.getTime();
      System.out.println("The Java Date is:" +
             javaDate.toString());

      //Get and display SQL DATE
      java.sql.Date sqlDate = new java.sql.Date(javaTime);
      System.out.println("The SQL DATE is: " +
             sqlDate.toString());

      //Get and display SQL TIME
      java.sql.Time sqlTime = new java.sql.Time(javaTime);
      System.out.println("The SQL TIME is: " +
             sqlTime.toString());
      //Get and display SQL TIMESTAMP
      java.sql.Timestamp sqlTimestamp =
      new java.sql.Timestamp(javaTime);
      System.out.println("The SQL TIMESTAMP is: " +
             sqlTimestamp.toString());
     }//end main
}//end SqlDateTime
```

Now let us compile the above example as follows −

```
C:\>javac SqlDateTime.java
C:\>
```

When you run **JDBCExample**, it produces the following result −

```
C:\>java SqlDateTime
The Java Date is:Tue Aug 18 13:46:02 GMT+04:00 2009
The SQL DATE is: 2009-08-18
The SQL TIME is: 13:46:02
The SQL TIMESTAMP is: 2009-08-18 13:46:02.828
C:\>
```

Handling NULL Values

SQL's use of NULL values and Java's use of null are different concepts. So, to handle SQL NULL values in Java, there are three tactics you can use −

- Avoid using getXXX methods that return primitive data types.
- Use wrapper classes for primitive data types, and use the ResultSet object's wasNull method to test whether the wrapper class variable that received the value returned by the getXXX method should be set to null.
- Use primitive data types and the ResultSet object's wasNull method to test whether the primitive variable that received the value returned by the getXXX method should be set to an acceptable value that you've chosen to represent a NULL.

Here is one example to handle a NULL value −

```
Statement stmt = conn.createStatement( );
String sql = "SELECT id, first, last, age FROM Employees";
ResultSet rs = stmt.executeQuery(sql);

int id = rs.getInt(1);
if( rs.wasNull( ) ) {
   id = 0;
}
```

Loading [MathJax]/jax/output/HTML-CSS/jax.js