4
Oracle Extensions

Oracle provides Java classes and interfaces that extend the Java Database Connectivity (JDBC) standard implementation, enabling you to access and manipulate Oracle data types and use Oracle performance extensions. This chapter provides an overview of the classes and interfaces provided by Oracle that extend the JDBC standard implementation. It also describes some of the key support features of the extensions.

This chapter contains the following sections:

- Overview of Oracle Extensions
- Features of the Oracle Extensions
- Oracle JDBC Packages
- Oracle Character Data Types Support
- Additional Oracle Type Extensions
- DML Returning
- Accessing PL/SQL Associative Arrays

Related Topics

- Performance Extensions

4.1 Overview of Oracle Extensions

Beyond standard features, Oracle JDBC drivers provide Oracle-specific type extensions and performance extensions. These extensions are provided through the following Java packages:

- `oracle.sql`

  Provides classes that represent SQL data in Oracle format

- `oracle.jdbc`

  Provides interfaces to support database access and updates in Oracle type formats

Related Topics

- Oracle JDBC Packages

4.2 Features of the Oracle Extensions

The Oracle extensions to JDBC include a number of features that enhance your ability to work with Oracle Databases. These include the following:

- Support for Pipelined Database Operations
- Database Management Using JDBC
- Support for Oracle Data Types
- Support for Oracle Objects

- Support for Schema Naming
- DML Returning
- About Accessing PL/SQL Associative Arrays

4.2.1 Database Management Using JDBC

Starting from Oracle Database 11*g* Release 1, the `oracle.jdbc.OracleConnection` interface has two JDBC methods, `startup` and `shutdown`, which enable you to start up and shut down an Oracle Database instance.

> **Note:**
>
> My Oracle Support Note 335754.1 announces the desupport of the `oracle.jdbc.driver.*` package in Oracle Database 11*g* JDBC drivers. In other words, Oracle Database 10*g* Release 2 was the last database to support this package and any API depending on the `oracle.jdbc.driver.*` package will fail to compile in the current release of the Database. You must remove such APIs and migrate to the standard APIs. For example, if your code uses the `oracle.jdbc.CustomDatum` and `oracle.jdbc.CustomDatumFactory` interfaces, then you must replace them with the `java.sql.Struct` or `java.sql.SQLData` interfaces.

Related Topics

- Database Administration

4.2.2 Support for Oracle Data Types

One of the features of the Oracle JDBC extensions is the type support in the `oracle.sql` package. This package includes classes that are an exact representation of the data in Oracle format. Keep the following important points in mind, when you use `oracle.sql` types in your program:

- For numeric type of data, the conversion to standard Java types does not guarantee to retain full precision due to limitations of the data conversion process. Use the `BigDecimal` type to minimize any data loss issues.
- For certain data types, the conversion to standard Java types can be dependent on the system settings and your program may not run as expected. This is a known limitation while converting data from `oracle.sql` types to standard Java types.
- If the functionalities of your program is limited to reading data from one table and writing the same to another table, then for numeric and date data, `oracle.sql` types are slightly faster as compared to standard Java types. But, if your program involves even a simple data manipulation operation like compare or print, then standard Java types are faster.
- `oracle.sql.CHAR` is not an exact representation of the data in Oracle format. `oracle.sql.CHAR` is constructed from `java.lang.String`. There is no advantage of using `oracle.sql.CHAR` because `java.lang.String` is always faster and represents the same character sets, excluding a couple of desupported character sets.

> **Note:**
>
> Oracle strongly recommends you to use standard Java types and convert any existing `oracle.sql` type of data to standard Java types. Internally, the Oracle JDBC drivers strive to maximize the performance of Java standard types. `oracle.sql` types are supported *only* for backward compatibility and their use is discouraged.

Related Topics

- Package oracle.sql
  The `oracle.sql` package supports direct access to data in SQL format. This package consists primarily of classes that provide Java mappings to SQL data types and their support classes. Essentially, the classes act as Java containers for SQL data.
- Oracle Character Data Types Support
- Additional Oracle Type Extensions

4.2.3 Support for Oracle Objects

Oracle JDBC supports the use of structured objects in the database, where an object data type is a user-defined type with nested attributes. For example, a user application could define an `Employee` object type, where each `Employee` object has a `firstname` attribute (character string), a `lastname` attribute (character string), and an `employeenumber` attribute (integer).

Oracle JDBC supports Oracle object data types. When you work with Oracle object data types in a Java application, you must consider the following:

- How to map between Oracle object data types and Java classes
- How to store Oracle object attributes in corresponding Java objects
- How to convert attribute data between SQL and Java formats
- How to access data

Oracle objects can be mapped either to the weak `java.sql.Struct` type or to strongly typed customized classes. These strong types are referred to as custom Java classes, which must implement either the standard `java.sql.SQLData` interface or the Oracle extension `oracle.jdbc.OracleData` interface. Each interface specifies methods to convert data between SQL and Java.

> **Note:**
>
> Starting from Oracle Database 12*c* Release 1 (12.1), the `OracleData` interface has replaced the `ORAData` interface.

Oracle recommends the use of the Oracle JVM Web Service Call-Out Utility to create custom Java classes to correspond to your Oracle objects.

Related Topics

- Working with Oracle Object Types
- *Oracle Database Java Developer's Guide*

4.2.4 Support for Schema Naming

Oracle object data type classes have the ability to accept and return fully qualified schema names. A fully qualified schema name has this syntax:

```
{[schema_name].}[sql_type_name]
```

Where, *schema_name* is the name of the schema and *sql_type_name* is the SQL type name of the object. *schema_name* and *sql_type_name* are separated by a period (.).

To specify an object type in JDBC, use its fully qualified name. It is not necessary to enter a schema name if the type name is in the current naming space, that is, the current schema. Schema naming follows these rules:

- Both the schema name and the type name may or may not be within quotation marks. However, if the SQL type name has a period in it, such as `CORPORATE.EMPLOYEE`, the type name must be quoted.
- The JDBC driver looks for the first period in the object name that is not within quotation marks and uses the string before the period as the schema name and the string following the period as the type name. If no period is found, then the JDBC driver takes the current schema as default. That is, you can specify only the type name, without indicating a schema, instead of specifying the fully qualified name if the object type name belongs to the current schema. This also explains why you must put the type name within quotation marks if the type name has a dot in it.

  For example, assume that user `HR` creates a type called `person.address` and then wants to use it in their session. `HR` may want to skip the schema name and pass in `person.address` to the JDBC driver. In this case, if `person.address` is not within quotation marks, then the period is detected and the JDBC driver mistakenly interprets `person` as the schema name and `address` as the type name.
- JDBC passes the object type name string to the database unchanged. That is, the JDBC driver does not change the character case even if the object type name is within quotation marks.

  For example, if `HR.PersonType` is passed to the JDBC driver as an object type name, then the JDBC driver passes the string to the database unchanged. As another example, if there is white space between characters in the type name string, then the JDBC driver will not remove the white space.

4.2.5 DML Returning

Oracle Database supports the use of the `RETURNING` clause with data manipulation language (DML) statements. This enables you to combine two SQL statements into one. Both the Oracle JDBC Oracle Call Interface (OCI) driver and the Oracle JDBC Thin driver support DML returning.

> **See Also:**
>
> "DML Returning"

4.2.6 PL/SQL Associative Arrays

Oracle JDBC drivers enable JDBC applications to make PL/SQL calls with Associative Array parameters. Oracle JDBC drivers support PL/SQL Associative Arrays of scalar data types

> **See Also:**
>
> "Accessing PL/SQL Associative Arrays"

4.3 Oracle JDBC Packages

This section describes the following Java packages, which support the Oracle JDBC extensions:

- Package oracle.sql
- Package oracle.sql.json
- Package oracle.jdbc

4.3.1 Package oracle.sql

The `oracle.sql` package supports direct access to data in SQL format. This package consists primarily of classes that provide Java mappings to SQL data types and their support classes. Essentially, the classes act as Java containers for SQL data.

Each of the `oracle.sql.*` data type classes extends `oracle.sql.Datum`, a superclass that encapsulates functionality common to all the data types. Some of the classes are for JDBC 2.0-compliant data types. These classes, implement standard JDBC 2.0 interfaces in the `java.sql` package, as well as extending the `oracle.sql.Datum` class.

The `LONG` and `LONG RAW` SQL types and `REF CURSOR` type category have no `oracle.sql.*` classes. Use standard JDBC functionality for these types. For example, retrieve `LONG` or `LONG RAW` data as input streams using the standard JDBC result set and callable statement methods `getBinaryStream` and `getCharacterStream`. Use the `getCursor` method for `REF CURSOR` types.

> **Note:**
>
> Oracle recommends the use of standard JDBC types or Java types whenever possible. The types in the package `oracle.sql.*` are provided primarily for backward compatibility or for support of a few Oracle specific features such as `OPAQUE`, `OracleData`, `TIMESTAMPTZ`, and so on.

General oracle.sql.* Data Type Support

Each of the Oracle data type classes provides, among other things, the following:

- Data storage as Java byte arrays for SQL data
- A `getBytes()` method, which returns the SQL data as a byte array

- A `toJdbc()` method that converts the data into an object of a corresponding Java class as defined in the JDBC specification

  The JDBC driver does not convert Oracle-specific data types that are not part of the JDBC specification, such as `BFILE`. The driver returns the object in the corresponding `oracle.sql.*` format.
- Appropriate *`xxx`*`Value` methods to convert SQL data to Java type. For example, `stringValue`, `intValue`, `booleanValue`, `dateValue`, and `bigDecimalValue`
- Additional conversion methods, `get`*`XXX`* and `set`*`XXX`*, as appropriate, for the functionality of the data type, such as methods in the large object (LOB) classes that get the data as a stream and methods in the `REF` class that get and set object data through the object reference.

Overview of Class oracle.sql.STRUCT

`oracle.sql.STRUCT` class is the Oracle implementation of `java.sql.Struct` interface. This class is a value class and you should not change the contents of the class after construction. This class, as with all `oracle.sql.*` data type classes, is a subclass of the `oracle.sql.Datum` class.

> **Note:**
>
> Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.STRUCT` class is deprecated and replaced with the `oracle.jdbc.OracleStruct` interface, which is a part of the `oracle.jdbc` package. Oracle strongly recommends you to use the methods available in the `java.sql` package, where possible, for standard compatibility and methods available in the `oracle.jdbc` package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about the `oracle.jdbc.OracleStruct` interface.

Overview of Class oracle.sql.REF

The `oracle.sql.REF` class is the generic class that supports Oracle object references. This class, as with all `oracle.sql.*` data type classes, is a subclass of the `oracle.sql.Datum` class.

> **Note:**
>
> Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.REF` class is deprecated and replaced with the `oracle.jdbc.OracleRef` interface, which is a part of the `oracle.jdbc` package. Oracle strongly recommends you to use the methods available in the `java.sql` package, where possible, for standard compatibility and methods available in the `oracle.jdbc` package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about the `oracle.jdbc.OracleRef` interface.

The `REF` class has methods to retrieve and pass object references. However, selecting an object reference retrieves only a pointer to an object. This does not materialize the object itself. But the `REF` class also includes methods to retrieve and pass the object data. You cannot create `REF` objects in your JDBC application. You can only retrieve existing `REF` objects from the database.

You should use the JDBC standard type, `java.sql.Ref`, and the JDBC standard methods in preference to using `oracle.sql.REF`. If you want your code to be more portable, then you must use the standard type because only the Oracle JDBC drivers will use instances of `oracle.sql.REF` type.

Overview of Classes oracle.sql.BLOB, oracle.sql.CLOB, oracle.sql.BFILE

Binary large objects (BLOBs), character large objects (CLOBs), and binary files (BFILEs) are for data items that are too large to store directly in a database table. Instead, the database table stores a locator that points to the location of the actual data.

> **Note:**
>
> - Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.BLOB` and `Oracle.sql.CLOB` classes are deprecated and replaced with the `oracle.jdbc.OracleBlob` and `oracle.jdbc.OracleClob` interfaces respectively, which are a part of the `oracle.jdbc` package. Oracle strongly recommends you to use the methods available in the `java.sql` package, where possible, for standard compatibility and methods available in the `oracle.jdbc` package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about the `oracle.jdbc.OracleBlob` and `oracle.jdbc.OracleClob` interfaces.
> - `oracle.sql.BFILE` is an Oracle proprietary extension and there is no JDBC standard equivalent.

The `oracle.sql` package supports these data types in several ways:

- BLOBs point to large unstructured binary data items and are supported by the `oracle.sql.BLOB` class.
- CLOBs point to large character data items and are supported by the `oracle.sql.CLOB` class.
- BFILEs point to the content of external files (operating system files) and are supported by the `oracle.sql.BFILE` class. BFiles are read-only.

You can select a BLOB, CLOB, or BFILE locator from the database using a standard `SELECT` statement. However, you receive only the locator, and not the data. Additional steps are necessary to retrieve the data.

Overview of Classes oracle.sql.DATE, oracle.sql.NUMBER, and oracle.sql.RAW

These classes hold primitive SQL data types in Oracle native representation. In most cases, these types are not used internally by the drivers and you should use the standard JDBC types instead.

Java `Double` and `Float NaN` values do not have an equivalent Oracle `NUMBER` representation. For example, for Oracle `BINARY_FLOAT` and `BINARY_DOUBLE` data types, negative zero is coerced to positive zero and all NaNs are coerced to the canonical one. So, a `NullPointerException` is thrown whenever a `Double.NaN` value or a `Float.NaN` value is converted into an Oracle `NUMBER` using the `oracle.sql.NUMBER` class. For instance, the following code throws a `NullPointerException`:

```
oracle.sql.NUMBER n = new oracle.sql.NUMBER(Double.NaN);
System.out.println(n.doubleValue());  // throws NullPointerException
```

Overview of Classes oracle.sql.TIMESTAMP, oracle.sql.TIMESTAMPTZ, and oracle.sql.TIMESTAMPLTZ

The JDBC drivers support the following date/time data types:

- `TIMESTAMP` (`TIMESTAMP`)
- `TIMESTAMP WITH TIME ZONE` (`TIMESTAMPTZ`)
- `TIMESTAMP WITH LOCAL TIME ZONE` (`TIMESTAMPLTZ`)

The JDBC drivers allow conversions between `DATE` and date/time data types. For example, you can access a `TIMESTAMP WITH TIME ZONE` column as a `DATE` value.

The JDBC drivers support the most popular time zone names used in the industry as well as most of the time zone names defined in the JDK. Time zones are specified by using the `java.util.TimeZone` class.

> **Note:**
>
> - Do not use `TimeZone.getTimeZone` to create time zone objects. The Oracle time zone data types support more time zone names than JDK.
> - If a result set contains a `TIMESTAMPLTZ` column followed by a `LONG` column, then reading the `LONG` column results in an error.

The following code shows how the `TimeZone` and `Calendar` objects are created for `US_PACIFIC`, which is a time zone name not defined in JDK:

```
TimeZone tz = TimeZone.getDefault();
tz.setID("US_PACIFIC");
GregorianCalendar gcal = new GregorianCalendar(tz);
```

The following Java classes represent the SQL date/time types:

- `oracle.sql.TIMESTAMP`
- `oracle.sql.TIMESTAMPTZ`
- `oracle.sql.TIMESTAMPLTZ`

Before accessing `TIMESTAMP WITH LOCAL TIME ZONE` data, call the `OracleConnection.setSessionTimeZone(String regionName)` method to set the session time zone. When this method is called, the JDBC driver sets the session time zone of the connection and saves the session time zone so that any `TIMESTAMP WITH LOCAL TIME ZONE` data accessed through JDBC can be adjusted using the session time zone.

> **Note:**
>
> `TIMESTAMP WITH TIME ZONE` and `TIMESTAMP WITH LOCAL TIME ZONE` types can be represented as standard `java.sql.Timestamp` type. The byte representation of `TIMESTAMP WITH TIME ZONE` and `TIMESTAMP WITH LOCAL TIME ZONE` types to `java.sql.Timestamp` is straight forward. This is because the internal format of `TIMESTAMP WITH TIME ZONE` and `TIMESTAMP WITH LOCAL TIME ZONE` data types is GMT, and `java.sql.Timestamp` type objects internally use a milliseconds time value that is the number of milliseconds since EPOCH. However, the `String` representation of these data types requires time zone information that is obtained dynamically from the server and cached on the client side.
>
> In earlier versions of JDBC drivers, the cache of time zone was shared across different connections. This used to cause problems sometimes due to incompatibility in various time zones. Starting from Oracle Database 11 Release 2 version of JDBC drivers, the time zone cache is based on the time zone version supplied by the database. This newly designed cache avoids any issues related to version incompatibility of time zones.

Overview of Class oracle.sql.OPAQUE

The `oracle.sql.OPAQUE` class provides the name and characteristics of the `OPAQUE` type and any attributes. The `OPAQUE` type provides access only to the uninterrupted bytes of the instance.

> **Note:**
>
> Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.OPAQUE` class is deprecated and replaced with the `oracle.jdbc.OracleOpaque` interface, which is a part of the `oracle.jdbc` package. Oracle recommends you to use the methods available in the `java.sql` package, where possible, for standard compatibility and methods available in the `oracle.jdbc` package for Oracle specific extensions. Refer to MoS Note 1364193.1 for more information about the `oracle.jdbc.OracleOpaque` interface.

Related Topics

- *Oracle Database SQL Language Reference*
- JDBC Java API Reference
- Working with Large Objects and SecureFiles
  Large Objects (LOBs) are a set of data types that are designed to hold large amounts of data. This chapter describes how to use Java Database Connectivity (JDBC) to access and manipulate LOBs and SecureFiles using either the data interface or the locator interface.

4.3.2 Package oracle.sql.json

Starting with release 21c, Oracle Database provides a native JSON SQL type in the database. The `oracle.sql.json` package provides functionality to work with the JSON type values.

Specifically, you can use the `oracle.sql.json` package to perform the following tasks:

- Store and retrieve JSON type values in the database
- Read, create, and modify JSON type values
- Encode or decode JSON type values in the same binary JSON storage format as used by the database
- Convert JSON type values to and from JSON text
- Bind and access JSON type values using the JSON-P interfaces like `javax.json.*`

> **See Also:**
>
> JDBC Java API Reference

The following example shows how to insert, get, and modify JSON type values:

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

import oracle.jdbc.OracleConnection;
import oracle.jdbc.pool.OracleDataSource;
import oracle.sql.json.OracleJsonFactory;
import oracle.sql.json.OracleJsonObject;

public class JsonExample {

  public static void main(String[] args) throws SQLException {
    OracleDataSource ds = new OracleDataSource();
    ds.setURL("jdbc:oracle:thin:@myhost:1521:orcl");
    ds.setUser(<user>);
    ds.setPassword(<password>);
    OracleConnection con = (OracleConnection) ds.getConnection();

    // create a table with a JSON column and insert one value
    Statement stmt = con.createStatement();
    stmt.executeUpdate("CREATE TABLE fruit (data JSON)");
    stmt.executeUpdate("INSERT INTO fruit VALUES
('{"name":"pear","count":10}')");

    // create another JSON object
    OracleJsonFactory factory = new OracleJsonFactory();
    OracleJsonObject orange = factory.createObject();
    orange.put("name", "orange");
    orange.put("count", 12);

    // insert the orange object
    PreparedStatement pstmt = con.prepareStatement("INSERT INTO fruit VALUES
(:1)");
    pstmt.setObject(1, orange, OracleType.JSON);
    pstmt.executeUpdate();
    pstmt.close();

    // retrieve the pear object
```

```
    ResultSet rs = stmt.executeQuery("SELECT data FROM fruit f WHERE
f.data.name = 'pear'");
    rs.next();
    OracleJsonObject pear = rs.getObject(1, OracleJsonObject.class);
    int count = pear.getInt("count");

    // create a modifiable copy of the pear object
    pear = factory.createObject(pear);
    pear.put("count", count + 1);
    pear.put("color", "green");

    // update the pear object
    pstmt = con.prepareStatement("UPDATE fruit f SET data = :1 WHERE
f.data.name = 'pear');
    pstmt.setObject(1, pear, OracleType.JSON);
    pstmt.executeUpdate();
    pstmt.close();

    rs.close();
    stmt.close();
    con.close();
  }
}
```

Compatibility with Client Libraries Prior to Release 21c

If you have a client library earlier to release 21c, then your application treats the JSON type column in the database as a String, BLOB, or CLOB column. So, to query the JSON type column, you must use the query methods for those data types. However, when you use the BLOB APIs like the `getBlob` method, they return an error, even after you upgrade the client library. So, Oracle recommends that for BLOB data type, you use the `getBytes` method or the `getBinaryStream` method, which will return text data with UTF-8 JSON encoding. The following code snippets show how to query a JSON type column for BLOB data type:

Example: Using the `getBinaryStream` Method

```
public static void fetchStream(ResultSet rs) throws SQLException, IOException
{
  InputStream is = rs.getBinaryStream("JCOL");
  ByteArrayOutputStream baos = new ByteArrayOutputStream();
  int n = -1;
  byte[] buffer = new byte[1024];
  while ((n = is.read(buffer)) != -1) {
    baos.write(buffer, 0, n);
  }
  is.close();
  byte[] bytes = baos.toByteArray();
  System.out.println(new String(bytes, StandardCharsets.UTF_8));
}
public static void example1(Connection con) throws SQLException, IOException
{
  Statement stmt = con.createStatement();

  ResultSet rs = stmt.executeQuery("select jcol from jtab where rownum < 2");
```

```
  while (rs.next()){
    fetchStream(rs);
  }
}
```

Example: Using the `getBytes` Method

```
public static void fetchString(ResultSet rs) throws SQLException
{
  byte[] utf8 = rs.getBytes("JCOL");

  System.out.println(new String(utf8, StandardCharsets.UTF_8));
}
public static void example2(Connection con) throws SQLException
{
  Statement stmt = con.createStatement();

  ResultSet rs = stmt.executeQuery("select jcol from jtab where rownum < 2");

  while (rs.next()){
    fetchString(rs);
  }
}
```

Support for Array Enqueue and Dequeue of JSON Payload Type

Oracle Database 21c release introduced support for JSON payload type in Oracle Advance Queuing (AQ). For example, when you create a queue in the database, you can mention the queue payload type in the following way:

```
BEGIN

    DBMS_AQADM.CREATE_QUEUE_TABLE(

            QUEUE_TABLE =>'HR.JSON_SINGLE_QUEUE_TABLE',

            QUEUE_PAYLOAD_TYPE =>'JSON',

            COMPATIBLE => '10.0');

END;
```

Once you create a JSON payload type queue in the database, you can enqueue and dequeue JSON messages to/from that queue.

Starting from Oracle Database 23ai release, the driver supports the array enqueue and dequeue of JSON messages, which means that the client applications can enqueue and dequeue more than one JSON type messages to/from the database.

> **See Also:**
>
> Oracle Advanced Queuing for more information about AQ

4.3.3 Package oracle.jdbc

The interfaces of the `oracle.jdbc` package define the Oracle extensions to the interfaces in `java.sql`. These extensions provide access to Oracle SQL-format data and other Oracle-specific functionality, including Oracle performance enhancements.

> **See Also:**
>
> "The oracle.jdbc Package"

4.4 Oracle Character Data Types Support

Oracle character data types include the SQL CHAR and NCHAR data types. The following sections describe how these data types can be accessed using the `oracle.sql.*` classes:

- SQL CHAR Data Types
- SQL NCHAR Data Types
- Class oracle.sql.CHAR

4.4.1 SQL CHAR Data Types

The SQL CHAR data types include `CHAR`, `VARCHAR2`, and `CLOB`. These data types let you store character data in the database character set encoding scheme. The character set of the database is established when you create the database.

4.4.2 SQL NCHAR Data Types

The SQL `NCHAR` data types were created for Globalization Support. The SQL `NCHAR` data types include `NCHAR`, `NVARCHAR2`, and `NCLOB`. These data types enable you to store Unicode data in the database `NCHAR` character set encoding. The `NCHAR` character set, which never changes, is established when you create the database.

> **Note:**
>
> Because the `UnicodeStream` class is deprecated in favor of the `CharacterStream` class, the `setUnicodeStream` and `getUnicodeStream` methods are not supported for `NCHAR` data type access. Use the `setCharacterStream` method and the `getCharacterStream` method if you want to use stream access.

The usage of SQL `NCHAR` data types is similar to that of the SQL `CHAR` data types. JDBC uses the same classes and methods to access SQL `NCHAR` data types that are used for the corresponding SQL `CHAR` data types. Therefore, there are no separate, corresponding classes defined in the `oracle.sql` package for SQL `NCHAR` data types. Similarly, there is no separate, corresponding constant defined in the `oracle.jdbc.OracleTypes` class for SQL `NCHAR` data types.

> **See Also:**
>
> "NCHAR_ NVARCHAR2_ NCLOB and the defaultNChar Property"

> **Note:**
>
> The `setFormOfUse` method must be called before the `registerOutParameter` method is called in order to avoid unpredictable results.

The following code shows how to access SQL `NCHAR` data:

```
//
// Table TEST has the following columns:
// - NUMBER
// - NVARCHAR2
// - NCHAR
//
oracle.jdbc.OraclePreparedStatement pstmt =
  (oracle.jdbc.OraclePreparedStatement)
conn.prepareStatement("insert into TEST values(?, ?, ?)");

//
// oracle.jdbc.OraclePreparedStatement.FORM_NCHAR should be used for all NCHAR,
// NVARCHAR2 and NCLOB data types.
//

pstmt.setInt(1, 1);                    // NUMBER column
pstmt.setNString(2, myUnicodeString1);  // NVARCHAR2 column
pstmt.setNString(3, myUnicodeString2);  // NCHAR column
pstmt.execute();
```

4.4.3 Class oracle.sql.CHAR

The `oracle.sql.CHAR` class is used by Oracle JDBC in handling and converting character data. This class provides the Globalization Support functionality to convert character data. This class has two key attributes: Globalization Support character set and the character data. The Globalization Support character set defines the encoding of the character data. It is a parameter that is always passed when a `CHAR` object is constructed. Without the Globalization Support character set information, the data bytes in the `CHAR` object are meaningless. The `oracle.sql.CHAR` class is used for both SQL `CHAR` and SQL `NCHAR` data types.

> **Note:**
>
> In versions of Oracle JDBC drivers prior to 10*g* Release 1, there were performance advantages to using the `oracle.SQL.CHAR`. Starting from Oracle Database 10*g*, there are no longer any such advantages. In fact, optimum performance is achieved using the `java.lang.String`. All Oracle JDBC drivers handle all character data in the Java UCS2 character set. Using the `oracle.sql.CHAR` does not prevent conversions between the database character set and UCS2 character set.

The only remaining use of the `oracle.sql.CHAR` class is to handle character data in the form of raw bytes encoded in an Oracle Globalization Support character set. All character data retrieved from Oracle Database should be accessed using the `java.lang.String` class. When processing byte data from another source, you can use an `oracle.sql.CHAR` to convert the bytes to `java.lang.String`.

To convert an `oracle.sql.CHAR`, you must provide the data bytes and an `oracle.sql.CharacterSet` instance that represents the Globalization Support character set used to encode the data bytes.

The `CHAR` objects that are Oracle object attributes are returned in the database character set.

JDBC application code rarely needs to construct `CHAR` objects directly, because the JDBC driver automatically creates `CHAR` objects, when it is needed to create them on those rare occasions.

To construct a `CHAR` object, you must provide character set information to the `CHAR` object by way of an instance of the `CharacterSet` class. Each instance of this class represents one of the Globalization Support character sets that Oracle supports. A `CharacterSet` instance encapsulates methods and attributes of the character set, mainly involving functionality to convert to or from other character sets.

Constructing an oracle.sql.CHAR Object

Follow these general steps to construct a `CHAR` object:

1. Create a `CharacterSet` object by calling the `static CharacterSet.make` method.

   This method is a factory for the character set instance. The `make` method takes an integer as input, which corresponds to a character set ID that Oracle supports. For example:

   ```
   int oracleId = CharacterSet.JA16SJIS_CHARSET; // this is character set ID,
                                                 // 832
   ...
   CharacterSet mycharset = CharacterSet.make(oracleId);
   ```

   Each character set that Oracle supports has a unique, predefined Oracle ID.

2. Construct a `CHAR` object.

   Pass a string, or the bytes that represent the string, to the constructor along with the `CharacterSet` object that indicates how to interpret the bytes based on the character set. For example:

   ```
   String mystring = "teststring";
   ...
   CHAR mychar = new CHAR(teststring, mycharset);
   ```

   There are multiple constructors for `CHAR`, which can take a `String`, a `byte` array, or an object as input along with the `CharacterSet` object. In the case of a `String`, the string is converted to the character set indicated by the `CharacterSet` object before being placed into the `CHAR` object.

> **Note:**
>
> - The `CharacterSet` object cannot be a null value.
> - The `CharacterSet` class is an abstract class, therefore it has no constructor. The only way to create instances is to use the `make` method.
> - The server recognizes the special value `CharacterSet.DEFAULT_CHARSET` as the database character set. For the client, this value is not meaningful.
> - Oracle does not intend or recommend that users extend the `CharacterSet` class.

oracle.sql.CHAR Conversion Methods

The `CHAR` class provides the following methods for translating character data to strings:

- `getString`

  This method converts the sequence of characters represented by the `CHAR` object to a string, returning a Java `String` object. If you enter an invalid `OracleID`, then the character set will not be recognized and the `getString` method will throw a `SQLException` exception.

- `toString`

  This method is identical to the `getString` method. But if you enter an invalid `OracleID`, then the character set will not be recognized and the `toString` method will return a hexadecimal representation of the `CHAR` data and will *not* throw a `SQLException` exception.

- `getStringWithReplacement`

  This method is identical to the `getString` method, except a default replacement character replaces characters that have no unicode representation in the `CHAR` object character set. This default character varies from character set to character set, but is often a question mark (`?`).

The database server and the client, or application running on the client, can use different character sets. When you use the methods of the `CHAR` class to transfer data between the server and the client, the JDBC drivers must convert the data from the server character set to the client character set or vice versa. To convert the data, the drivers use Globalization Support.

> **See Also:**
>
> Globalization Support

4.5 Additional Oracle Type Extensions

Oracle JDBC drivers support the Oracle-specific `BFILE` and `ROWID` data types and `REF CURSOR` types, which are not part of the standard JDBC specification. This section describes the `ROWID` and `REF CURSOR` type extensions. The `ROWID` is supported as a Java string, and `REF CURSOR` types are supported as JDBC result sets.

This section covers the following topics:

- Oracle ROWID Type
- Oracle REF CURSOR Type Category
- Oracle BINARY_FLOAT and BINARY_DOUBLE Types
- Oracle SYS.ANYTYPE and SYS.ANYDATA Types
- The oracle.jdbc Package

4.5.1 Oracle ROWID Type

A ROWID is an identification tag unique for each row of an Oracle Database table. The ROWID can be thought of as a virtual column, containing the ID for each row.

The `oracle.sql.ROWID` class is supplied as a container for `ROWID` SQL data type.

ROWIDs provide functionality similar to the `getCursorName` method specified in the `java.sql.ResultSet` interface and the `setCursorName` method specified in the `java.sql.Statement` interface.

If you include the ROWID pseudo-column in a query, then you can retrieve the ROWIDs with the result set `getString` method. You can also bind a ROWID to a `PreparedStatement` parameter with the `setString` method. This enables in-place updating, as in the example that follows.

> **Note:**
>
> Use the `oracle.sql.ROWID` class, only when you are using J2SE 5.0. For JSE 6, you should use the standard `java.sql.RowId` interface instead.

Example

The following example shows how to access and manipulate ROWID data:

> **Note:**
>
> The following example works only with JSE 6.

```
Statement stmt = conn.createStatement();

// Query the employee names with "FOR UPDATE" to lock the rows.
// Select the ROWID to identify the rows to be updated.

ResultSet rset =
   stmt.executeQuery ("SELECT first_name, rowid FROM employees FOR UPDATE");

// Prepare a statement to update the first_name column at a given ROWID

PreparedStatement pstmt =
   conn.prepareStatement ("UPDATE employees SET first_name = ? WHERE rowid = ?");

// Loop through the results of the query
while (rset.next ())
{
```

```
    String ename = rset.getString (1);
 RowId rowid = rset.getROWID(2); // Get the ROWID as a String
    pstmt.setString (1, ename.toLowerCase ());
    pstmt.setROWID (2, rowid); // Pass ROWID to the update statement
    pstmt.executeUpdate ();     // Do the update
}
```

4.5.2 Oracle REF CURSOR Type Category

A cursor variable holds the memory location of a query work area, rather than the contents of the area. Declaring a cursor variable creates a pointer. In SQL, a pointer has the data type `REF x`, where `REF` is short for `REFERENCE` and *x* represents the entity being referenced. A `REF CURSOR`, then, identifies a reference to a cursor variable. Because many cursor variables might exist to point to many work areas, `REF CURSOR` can be thought of as a category or data type specifier that identifies many different types of cursor variables. Starting from Oracle Database Release 18 *c*, JDBC drivers support `REF CURSOR` as IN bind variables.

> **Note:**
>
> `REF CURSOR` instances are not scrollable.

Perform the following steps to create a cursor variable:

1. Identify a type that belongs to the `REF CURSOR` category. For example:

   ```
   DECLARE TYPE DeptCursorTyp IS REF CURSOR
   ```

2. Then, create the cursor variable by declaring it to be of the type `DeptCursorTyp`:

   ```
   dept_cv DeptCursorTyp  - - declare cursor variable
   ...
   ```

   `REF CURSOR`, then, is a category of data types, rather than a particular data type.

Stored procedures can accept or return cursor variables of the `REF CURSOR` category. This output is equivalent to a database cursor or a JDBC result set. A `REF CURSOR` essentially encapsulates the results of a query.

In JDBC, a `REF CURSOR` can be accessed as follows:

1. Use a JDBC callable statement or a prepared statement to call a stored procedure.
2. The stored procedure accepts or returns a `REF CURSOR`.
3. The Java application casts the callable statement or prepared statement to an Oracle callable statement or Oracle prepared statement.
4. The Java application uses the `setCursor` method of the `OraclePreparedStatement` interface or the `getCursor` method of the `OracleCallableStatement` interface to materialize the `REF CURSOR` as a JDBC `ResultSet` object.
5. The result set is processed as requested.

> **Note:**
>
> - The cursor associated with a `REF CURSOR` is closed whenever the statement object that produced the `REF CURSOR` is closed.
> - Unlike in past releases, the cursor associated with a `REF CURSOR` is *not* closed when the result set object in which the `REF CURSOR` was materialized is closed.

Example

This example shows how to access `REF CURSOR` data.

```
    ...
    // Prepare a PL/SQL call
    CallableStatement cstmt =
      conn.prepareCall ("DECLARE  rc sys_refcursor; curid NUMBER;BEGIN open
rc FOR SELECT empno FROM emp order by empno; ? := rc;  END;");

    cstmt.registerOutParameter (1, OracleTypes.CURSOR);
    cstmt.execute ();
    ResultSet rset = (ResultSet)cstmt.getObject (1);

    if (rset.next ()) {
      show (rset.getString ("empno"));
    }

    CallableStatement cstmt2 =
      conn.prepareCall ("DECLARE  rc sys_refcursor; v1 NUMBER; BEGIN rc := ?;
fetch rc INTO v1; ? := v1; END;");
    ((OracleCallableStatement)call2).setCursor(1, rset);
    cstmt2.registerOutParameter (2, OracleTypes.INTEGER);

    cstmt2.execute();

    int empno = cstmt2.getInt(2);

    show("Fetch in PL/SQL empno=" + empno);

    // Dump the cursor
    while (rset.next ())
      show (rset.getString ("empno"));

    // Close all the resources
    rset.close();
    cstmt.close();
    cstmt2.close();
  ...
```

In the preceding example:

- Two `CallableStatement` objects `cstmt1` and `cstmt2` are created using the `prepareCall` method of the `Connection` class.

- The `cstmt2` callable statement uses `REF CURSOR` as input parameter.
- The callable statements implement PL/SQL procedure that returns a `REF CURSOR`.
- As always, the output parameter of the callable statement must be registered to define its type. Use the type code `OracleTypes.CURSOR` for a `REF CURSOR`.
- The callable statements are run, returning the `REF CURSOR` or sending the `REF CURSOR` as input bind.

4.5.3 Oracle BINARY_FLOAT and BINARY_DOUBLE Types

The Oracle `BINARY_FLOAT` and `BINARY_DOUBLE` types are used to store IEEE 574 float and double data. These correspond to the Java `float` and `double` scalar types with the exception of negative zero and `NaN`.

> **See Also:**
>
> *Oracle Database SQL Language Reference*

If you include a `BINARY_DOUBLE` column in a query, then the data is retrieved from the database in the binary format. Also, the `getDouble` method will return the data in the binary format. In contrast, for a `NUMBER` data type column, the number bits are returned and converted to the Java `double` data type.

> **Note:**
>
> The Oracle representation for the SQL `FLOAT`, `DOUBLE PRECISION`, and `REAL` data types use the Oracle `NUMBER` representation. The `BINARY_FLOAT` and `BINARY_DOUBLE` data types can be regarded as proprietary types.

A call to the JDBC standard `setDouble(int, double)` method of the `PreparedStatement` interface converts the Java `double` argument to Oracle `NUMBER` style bits and send them to the database. In contrast, the `setBinaryDouble(int, double)` method of the `oracle.jdbc.OraclePreparedStatement` interface converts the data to the internal binary bits and sends them to the database.

You must ensure that the data format used matches the type of the target parameter of the `PreparedStatement` interface. This will result in correct data and least use of CPU. If you use `setBinaryDouble` for a `NUMBER` parameter, then the binary bits are sent to the server and converted to `NUMBER` format. The data will be correct, but server CPU load will be increased. If you use `setDouble` for a `BINARY_DOUBLE` parameter, then the data will first be converted to `NUMBER` bits on the client and sent to the server, where it will be converted back to binary format. This will increase the CPU load on both client and server and can result in data corruption as well.

The `SetFloatAndDoubleUseBinary` connection property when set to `true` causes the JDBC standard APIs, `setFloat(int, float)`, `setDouble(int, double)`, and all the variations, to send internal binary bits instead of `NUBMER` bits.

> **Note:**
>
> Although this section largely discusses `BINARY_DOUBLE`, the same is true for `BINARY_FLOAT` as well.

4.5.4 Oracle SYS.ANYTYPE and SYS.ANYDATA Types

Oracle Database 12*c* Release 1 (12.1) provides a Java interface to access the `SYS.ANYTYPE` and `SYS.ANYDATA` Oracle types.

> **See Also:**
>
> For information about these Oracle types, refer *Oracle Database PL/SQL Packages and Types Reference*

An instance of the `SYS.ANYTYPE` type contains a type description of any SQL type, persistent or transient, named or unnamed, including object types and collection types. You can use the `oracle.sql.TypeDescriptor` class to access the `SYS.ANYTYPE` type. An `ANYTYPE` instance can be retrieved from a PL/SQL procedure or a SQL `SELECT` statement where `SYS.ANYTYPE` is used as a column type. To retrieve an `ANYTYPE` instance from the database, use the `getObject` method. This method returns an instance of the `TypeDescriptor`.

The retrieved `ANYTYPE` instance could be any of the following:

- Transient object type
- Transient predefined type
- Persistent object type
- Persistent predefined type

Example 4-1 Accessing SYS.ANYTYPE Type

The following code snippet illustrates how to retrieve an instance of `ANYTYPE` from the database:

```
...
ResultSet rs = stmt.executeQuery("select anytype_column from my_table");
TypeDescriptor td = (TypeDescriptor)rs.getObject(1);
short typeCode = td.getInternalTypeCode();
if(typeCode == TypeDescriptor.TYPECODE_OBJECT)
{
  // check if it's a transient type
  if(td.isTransientType())
  {
    AttributeDescriptor[] attributes = ((StructDescriptor)td).getAttributesDescriptor();
    for(int i=0; i<attributes.length; i++)
      System.out.println(attributes[i].getAttributeName());
  }
  else
  {    System.out.println(td.getTypeName());  }}
...
```

Example 4-2 Creating a Transient Object Type Through PL/SQL and Retrieving Through JDBC

This example provides a code snippet illustrating how to retrieve a transient object type through JDBC.

```
...
OracleCallableStatement cstmt = (OracleCallableStatement)conn.prepareCall
  ("BEGIN ? := transient_obj_type (); END;");
cstmt.registerOutParameter(1,OracleTypes.OPAQUE,"SYS.ANYTYPE");
cstmt.execute();
TypeDescriptor obj = (TypeDescriptor)cstmt.getObject(1);
if(!obj.isTransient())
  System.out.println("This must be a JDBC bug");
cstmt.close();
return obj;
...
```

Example 4-3 Calling a PL/SQL Stored Procedure That Takes an ANYTPE as IN Parameter

The following code snippet illustrates how to call a PL/SQL stored procedure that takes an `ANYTYPE` as `IN` parameter:

```
...
CallableStatement cstmt = conn.prepareCall("BEGIN ? := dumpanytype(?); END;");
cstmt.registerOutParameter(1,OracleTypes.VARCHAR);
// obj is the instance of TypeDescriptor that you have retrieved
cstmt.setObject(2,obj);
cstmt.execute();
String str = (String)cstmt.getObject(1);
...
```

The `oracle.sql.ANYDATA` class enables you to access `SYS.ANYDATA` instances from the database. An instance of this class can be obtained from any valid instance of `oracle.sql.Datum` class. The `convertDatum` factory method takes an instance of `Datum` and returns an instance of `ANYDATA`. The syntax for this factory method is as follows:

```
public static ANYDATA convertDatum(Datum datum) throws SQLException
```

The following is sample code for creating an instance of `oracle.sql.ANYDATA`:

```
// struct is a valid instance of oracle.sql.STRUCT that either comes from the
// database or has been constructed in Java.
ANYDATA myAnyData = ANYDATA.convertDatum(struct);
```

Example 4-4 Accessing an Instance of ANYDATA from the Database

```
...
// anydata_table has been created as:
// CREATE TABLE anydata_tab (data SYS.ANYDATA)
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select data from my_anydata_tab");
while(rs.next())
{
  ANYDATA anydata = (ANYDATA)rs.getObject(1);
  if(!anydata.isNull())
  {
    TypeDescriptor td = anydata.getTypeDescriptor();
    if(td.getTypeCode() == OracleType.TYPECODE_OBJECT)
      STRUCT struct = (STRUCT)anydata.accessDatum();
  }
```

```
}
...
```

Example 4-5 Inserting an Object as ANYDATA in a Database Table

Consider the following table and object type definition:

```
CREATE TABLE anydata_tab ( id NUMBER, data SYS.ANYDATA)

CREATE OR REPLACE TYPE employee AS OBJECT ( employee_id NUMBER, first_name VARCHAR2(10) )
```

You can create an instance of the `EMPLOYEE` SQL object type and to insert it into `anydata_tab`le in the following way:

```
...
PreparedStatement pstmt = conn.prepareStatement("insert into anydata_table values
(?,?)");
Struct myEmployeeStr = conn.createStruct("EMPLOYEE", new Object[]{1120, "Papageno"});
ANYDATA anyda = ANYDATA.convertDatum(myEmployeeStr);
pstmt.setInt(1,123);
pstmt.setObject(2,anyda);
pstmt.executeUpdate();
...
```

Example 4-6 Selecting an ANYDATA Column from a Database Table

```
...
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select data from anydata_table");
while(rs.next())
{
  ANYDATA obj = (ANYDATA)rs.getObject(1);
  TypeDescriptor td = obj.getTypeDescriptor();
}
rs.close();
stmt.close();
...
```

4.5.5 The oracle.jdbc Package

The interfaces of the `oracle.jdbc` package define the Oracle extensions to the interfaces in `java.sql`. These extensions provide access to SQL-format data as described in this chapter. They also provide access to other Oracle-specific functionality, including Oracle performance enhancements.

For the `oracle.jdbc` package, Table 4-1 lists key interfaces and classes used for connections, statements, and result sets.

Table 4-1 Key Interfaces and Classes of the oracle.jdbc Package

Name	Interface or Class	Key Functionality
`OracleDriver`	Class	Implements `java.sql.Driver`

Table 4-1 (Cont.) Key Interfaces and Classes of the oracle.jdbc Package

Name	Interface or Class	Key Functionality
`OracleConnection`	Interface	Provides methods to start and stop an Oracle Database instance and to return Oracle statement objects and methods to set Oracle performance extensions for any statement run in the current connection. Implements `java.sql.Connection`.
`OracleStatement`	Interface	Provides methods to set Oracle performance extensions for individual statement. Is a supertype of `OraclePreparedStatement` and `OracleCallableStatement`. Implements `java.sql.Statement`.
`OraclePreparedStatement`	Interface	Provides `set`*`XXX`* methods to bind `oracle.sql.*` types into a prepared statement. Provides `getMetaData` method to get the metadata from the prepared statements without executing the SELECT statements. Implements `java.sql.PreparedStatement`. Extends `OracleStatement`. Is a supertype of `OracleCallableStatement`.
`OracleCallableStatement`	Interface	Provides `get`*`XXX`* methods to retrieve data in `oracle.sql` format and `set`*`XXX`* methods to bind `oracle.sql.*` types into a callable statement. Implements `java.sql.CallableStatement`. Extends `OraclePreparedStatement`.
`OracleResultSet`	Interface	Provides `get`*`XXX`* methods to retrieve data in `oracle.sql` format. Implements `java.sql.ResultSet`.
`OracleResultSetMetaData`	Interface	Provides methods to get metadata information about Oracle result sets, such as column names and data types. Implements `java.sql.ResultSetMetaData`.
`OracleDatabaseMetaData`	Class	Provides methods to get metadata information about the database, such as database product name and version, table information, and default transaction isolation level. Implements `java.sql.DatabaseMetaData`).
`OracleTypes`	Class	Defines integer constants used to identify SQL types. For standard types, it uses the same values as the standard `java.sql.Types` class. In addition, it adds constants for Oracle extended types.

Table 4-1 (Cont.) Key Interfaces and Classes of the oracle.jdbc Package

Name	Interface or Class	Key Functionality
`OracleArray`	Interface	Includes functionality to retrieve the array as a whole, retrieve a subset of the array elements, and retrieve the SQL base type name of the array elements.
`OracleStruct`	Interface	
`OracleClob`	Interface	
`OracleBlob`	Interface	
`OracleRef`	Interface	
`OracleOpaque`	Interface	

This section covers the following topics:

- Interface oracle.jdbc.OracleConnection
- Interface oracle.jdbc.OracleStatement
- Interface oracle.jdbc.OraclePreparedStatement
- Interface oracle.jdbc.OracleCallableStatement
- Interface oracle.jdbc.OracleResultSet
- Interface oracle.jdbc.OracleResultSetMetaData
- Class oracle.jdbc.OracleTypes

4.5.5.1 Interface oracle.jdbc.OracleConnection

This interface extends standard JDBC connection functionality to create and return Oracle statement objects, set flags and options for Oracle performance extensions, support type maps for Oracle objects, and support client identifiers.

In Oracle Database 11*g* Release 1, new methods were added to this interface that enable the starting up and shutting down of an Oracle Database instance. Also, for better visibility and clarity, all connection properties are defined as constants in the `OracleConnection` interface.

This interface also defines factory methods for constructing `oracle.sql` data values like `DATE` and `NUMBER`. Remember the following points while using factory methods:

- All code that constructs instances of the `oracle.sql` types should use the Oracle extension factory methods. For example, `ARRAY`, `BFILE`, `DATE`, `INTERVALDS`, `NUMBER`, `STRUCT`, `TIME`, `TIMESTAMP`, and so on.
- All code that constructs instances of the standard types should use the JDBC 4.0 standard factory methods. For example, `CLOB`, `BLOB`, `NCLOB`, and so on.
- There are no factory methods for `CHAR`, `JAVA_STRUCT`, `ArrayDescriptor`, and `StructDescriptor`. These types are for internal driver use only.

> **Note:**
>
> Prior to Oracle Database 11*g* Release 1, you had to construct `ArrayDescriptors` and `StructDescriptors` for passing as arguments to the `ARRAY` and `STRUCT` class constructors. The new `ARRAY` and `Struct` factory methods do not have any descriptor arguments. The driver still uses descriptors internally, but you do not need to create them.

Client Identifiers

In a connection pooling environment, the client identifier can be used to identify the lightweight user using the database session currently. A client identifier can also be used to share the Globally Accessed Application Context between different database sessions. The client identifier set in a database session is audited when database auditing is turned on.

> **See Also:**
>
> *Oracle Database JDBC Java API Reference* for more information

4.5.5.2 Interface oracle.jdbc.OracleStatement

This interface extends standard JDBC statement functionality and is the superinterface of the `OraclePreparedStatement` and `OracleCallableStatement` classes. Extended functionality includes support for setting flags and options for Oracle performance extensions on a statement-by-statement basis, as opposed to the `OracleConnection` interface that sets these on a connectionwide basis.

4.5.5.3 Interface oracle.jdbc.OraclePreparedStatement

This interface extends the `OracleStatement` interface and extends standard JDBC prepared statement functionality. Also, the `oracle.jdbc.OraclePreparedStatement` interface is extended by the `OracleCallableStatement` interface. Extended functionality consists of the following:

- `set`*`XXX`* methods for binding `oracle.sql.*` types and objects to prepared statements
- `getMetaData` method to get the metadata from the prepared statements without executing the SELECT statements
- Methods to support Oracle performance extensions on a statement-by-statement basis

> **Note:**
>
> Do not use the `PreparedStatement` interface to create a trigger that refers to a `:NEW` or `:OLD` column. Use `Statement` instead. Using `PreparedStatement` will cause execution to fail with the message `java.sql.SQLException: Missing IN or OUT parameter at index:: 1`.

4.5.5.4 Interface oracle.jdbc.OracleCallableStatement

This interface extends the `OraclePreparedStatement` interface, which extends the `OracleStatement` interface and incorporates standard JDBC callable statement functionality.

> **Note:**
>
> Do not use the `CallableStatement` interface to create a trigger that refers to a `:NEW` or `:OLD` column. Use `Statement` instead; using `CallableStatement` will cause execution to fail with the message `java.sql.SQLException: Missing IN or OUT parameter at index::1`

> **Note:**
>
> - The `set`*`XXX`*`(String,...)` and `registerOutParameter(String,...)` methods can be used only if all binds are procedure or function parameters only. The statement can contain no other binds and the parameter binds must be indicated with a question mark (`?`) and not `:`*`XX`*.
> - If you are using `set`*`XXX`*`(int,...)` or `set`*`XXX`*`AtName(String,...)` method, then any output parameter is bound with `registerOutParameter(int,...)` and not `registerOutParameter(String,...)`, which is for named parameter notation.

4.5.5.5 Interface oracle.jdbc.OracleResultSet

This interface extends standard JDBC result set functionality, implementing `get`*`XXX`* methods for retrieving data into `oracle.sql.*` objects.

4.5.5.6 Interface oracle.jdbc.OracleResultSetMetaData

This interface extends standard JDBC result set metadata functionality to retrieve information about Oracle result set objects.

> **See Also:**
>
> "Using Result Set Metadata Extensions"

4.5.5.7 Class oracle.jdbc.OracleTypes

The `OracleTypes` class defines constants that JDBC uses to identify SQL types. Each variable in this class has a constant integer value. The `oracle.jdbc.OracleTypes` class duplicates the type code definitions of the standard Java `java.sql.Types` class and contains these additional type codes for Oracle extensions:

- `OracleTypes.BFILE`

- OracleTypes.ROWID
- OracleTypes.CURSOR (for REF CURSOR types)
- OracleTypes.CHAR_BYTES (for calling setNull and setCHAR methods on the same column)

As in java.sql.Types, all the variable names in oracle.jdbc.OracleTypes are also in uppercase text. JDBC uses the SQL types identified by the elements of the OracleTypes class in the following three main areas:

- Registering output parameters
- Using the setNull method
- Supporting the new BOOLEAN data type

Registering Output Parameters

The type codes in java.sql.Types or oracle.jdbc.OracleTypes identify the SQL types of the output parameters in the registerOutParameter method of the java.sql.CallableStatement and oracle.jdbc.OracleCallableStatement interfaces.

These are the forms that the registerOutputParameter method can take for the CallableStatement and OracleCallableStatement interfaces

```
cs.registerOutParameter(int index, int sqlType);

cs.registerOutParameter(int index, int sqlType, String sql_name);

cs.registerOutParameter(int index, int sqlType, int scale);
```

In these signatures, index represents the parameter index, sqlType is the type code for the SQL data type, sql_name is the name given to the data type, for user-defined types, when sqlType is a STRUCT, REF, or ARRAY type code, and scale represents the number of digits to the right of the decimal point, when sqlType is a NUMERIC or DECIMAL type code.

The following example uses a CallableStatement interface to call a procedure named charout, which returns a CHAR data type. Note the use of the OracleTypes.CHAR type code in the registerOutParameter method.

```
CallableStatement cs = conn.prepareCall ("BEGIN charout (?); END;");
cs.registerOutParameter (1, OracleTypes.CHAR);
cs.execute ();
System.out.println ("Out argument is: " + cs.getString (1));
```

The next example uses a CallableStatement interface to call structout, which returns a STRUCT data type. The form of registerOutParameter requires you to specify the type code, Types.STRUCT or OracleTypes.STRUCT, as well as the SQL name, EMPLOYEE.

The example assumes that no type mapping has been declared for the EMPLOYEE type, so it is retrieved into a STRUCT data type. To retrieve the value of EMPLOYEE as an oracle.sql.STRUCT object, the statement object cs is cast to OracleCallableStatement and the Oracle extension getSTRUCT method is invoked.

```
CallableStatement cs = conn.prepareCall ("BEGIN structout (?); END;");
cs.registerOutParameter (1, OracleTypes.STRUCT, "EMPLOYEE");
cs.execute ();

// get the value into a STRUCT because it
// is assumed that no type map has been defined
STRUCT emp = ((OracleCallableStatement)cs).getSTRUCT (1);
```

Using the setNull Method

The type codes in `Types` and `OracleTypes` identify the SQL type of the data item, which the `setNull` method sets to `NULL`. The `setNull` method can be found in the `java.sql.PreparedStatement` and `oracle.jdbc.OraclePreparedStatement` interfaces.

These are the forms that the `setNull` method can take for the `PreparedStatement` and `OraclePreparedStatement` objects:

```
ps.setNull(int index, int sqlType);

ps.setNull(int index, int sqlType, String sql_name);
```

In these signatures, `index` represents the parameter index, `sqlType` is the type code for the SQL data type, and `sql_name` is the name given to the data type, for user-defined types, when `sqlType` is a `STRUCT`, `REF`, or `ARRAY` type code. If you enter an invalid `sqlType`, a `ParameterTypeConflict` exception is thrown.

The following example uses a prepared statement to insert a null value into the database. Note the use of `OracleTypes.NUMERIC` to identify the numeric object set to `NULL`. Alternatively, `Types.NUMERIC` can be used.

```
PreparedStatement pstmt =
    conn.prepareStatement ("INSERT INTO num_table VALUES (?)");

pstmt.setNull (1, OracleTypes.NUMERIC);
pstmt.execute ();
```

In this example, the prepared statement inserts a `NULL STRUCT` object of type `EMPLOYEE` into the database.

```
PreparedStatement pstmt = conn.prepareStatement
                              ("INSERT INTO employees VALUES (?)");

pstmt.setNull (1, OracleTypes.STRUCT, "EMPLOYEE");
pstmt.execute ();
```

You can also use the `OracleTypes.CHAR_BYTES` type with the `setNull` method, if you also want to call the `setCHAR` method on the same column. For example:

```
 ps.setCHAR(n, aCHAR);
  ps.addBatch();
  ps.setNull(n, OracleTypes.CHAR_BYTES);
  ps.addBatch();
```

In this preceding example, any other type, apart from the `OracleTypes.CHAR_BYTES` type, will cause extra round trips to the Database. Alternatively, you can also write your code without using the `setNull` method. For example, you can also write your code as shown in the following example:

```
ps.setCHAR(n, null);
```

Supporting the BOOLEAN Data Type

Starting from Release 23ai, Oracle Database supports the `BOOLEAN` data type in compliance with the ISO SQL standard. The current release of JDBC Thin driver provides support for the newly introduced `BOOLEAN` data type, with the introduction of a new type `BOOLEAN` in `oracle.jdbc.OracleType`.

The following table lists the APIs to work with this new type:

Operation	API Used	Code Example

Insert	`java.sql.PreparedStatement.setBoolean`or`java.sql.PreparedStatement.setObject`	**Example**:
		
		`...`
		`String query = "INSERT INTO BoolTable values (?) ";`
		`PreparedStatement pstmt = con.prepareStatement(query);`
		`pstmt.setBoolean(1, true);`
		`pstmt.execute();`
		`...`

Fetch	`java.sql.ResultSet.getBoolean` or any equivalent method	…
		`Statement stmt = con.createStatement();`
		`ResultSet rs = stmt.executeQuery("select * from BoolTable");`
		`while(rs.next()) {`
		`  System.out.print("Value: " + rs.getBoolean("booleanColumn"));`
		…
		`}`

Fetch column metadata	`java.sql.ResultSetMetaData`	`...`
		`try (OraclePreparedStatement stmt = (OraclePreparedStatement)`
		`  conn.prepareStatement("SELECT ? < ?")) {`
		`  ResultSetMetaData rsmd = stmt.getMetaData();`
		
		`System.out.print(rsmd.getColumnType(1) + " ");`
		
		`System.out.println(rsmd.getColumnTypeName(1) + " ");`
		`}`
		`...`
		
		The expected output of the preceding code snippet is: `16 BOOLEAN`

> **Note:**
>
> If the columns with which the values are being compared to or the column into which values are inserted, are not of the appropriate types, then an implicit conversion may take place. If the value cannot be converted, an exception is thrown.

Supported and Unsupported Conversions for the BOOLEAN Type

Oracle Database allows implicit conversion from character and number types of data to `BOOLEAN` type, and `BOOLEAN` type to character and number types. So, you can also use external types for these types, while the Database takes care of implicit conversions for the binds and the client takes care of the implicit conversions for the defines.

The following table lists the supported conversions:

Java Type	Value	BOOLOEAN Value
boolean	`TRUE` or `FALSE`	`TRUE` or `FALSE`
String	"`TRUE`" or "`FALSE`" (case-insensitive)	`TRUE` or `FALSE`
`char`, `int`, `long`	Non-zero value or zero-value	`TRUE` or `FALSE`

The following table lists the String literals to represent "`True`" and "`False`":

True (case-insensitive)	False (case-insensitive)
"`true`"	"`false`"
"`yes`"	"`no`"
"`on`"	"`off`"
"`1`"	"`0`"
"`t`"	"`f`"
"`y`"	"`n`"

Conversion between the `NUMBER` and `BOOLEAN` types with non-zero scale returns an error. Also, when converting from string to `BOOLEAN`, if the string literal value is not any of the values listed in the preceding table, then an error is returned.

4.6 DML Returning

The DML returning feature provides more functionality compared to retrieval of auto-generated keys. It can be used to retrieve not only auto-generated keys, but also other columns or values that the application may use.

> **Note:**
>
> - The server-side internal driver does not support DML returning and retrieval of auto-generated keys.
> - You cannot use both DML returning and retrieval of auto-generated keys in the same statement.

The following sections explain the support for DML returning:

- Oracle-Specific APIs
- About Running DML Returning Statements
- Example of DML Returning
- Limitations of DML Returning

4.6.1 Oracle-Specific APIs

The `OraclePreparedStatement` interface is enhanced with Oracle-specific application programming interfaces (APIs) to support DML returning. The `registerReturnParameter` and `getReturnResultSet` methods have been added to the `oracle.jdbc.OraclePreparedStatement` interface, to register parameters that are returned and data retrieved by DML returning.

The `registerReturnParameter` method is used to register the return parameter for DML returning. The method throws a `SQLException` instance if an error occurs. You must pass a positive integer specifying the index of the return parameter. You also must specify the type of the return parameter. You can also specify the maximum bytes or characters of the return parameter. This method can be used only with `char` or `RAW` types. You can also specify the fully qualified name of a SQL structure type.

> **Note:**
>
> If you do not know the maximum size of the return parameters, then you should use `registerReturnParameter(int paramIndex, int externalType)`, which picks the default maximum size. If you know the maximum size of return parameters, using `registerReturnParameter(int paramIndex, int externalType, int maxSize)` can reduce memory consumption.

The `getReturnResultSet` method fetches the data returned from DML returning and returns it as a `ResultSet` object. The method throws a `SQLException` exception if an error occurs.

4.6.2 About Running DML Returning Statements

Before running a DML returning statement, the JDBC application must call one or more of the `registerReturnParameter` methods. The method provides the JDBC drivers with information, such as type and size, of the return parameters. The DML returning statement is then processed using one of the standard JDBC APIs, `executeUpdate` or `execute`. You can then fetch the returned parameters as a `ResultSet` object using the `getReturnResultSet` method of the `oracle.jdbc.OraclePreparedStatement` interface.

In order to read the values in the `ResultSet` object, the underlying `Statement` object must be open. When the underlying `Statement` object is closed, the returned `ResultSet` object is also closed. This is consistent with `ResultSet` objects that are retrieved by processing SQL query statements.

When a DML returning statement is run, the concurrency of the `ResultSet` object returned by the `getReturnResultSet` method must be `CONCUR_READ_ONLY` and the type of the `ResultSet` object must be `TYPE_FORWARD_ONLY` or `TYPE_SCROLL_INSENSITIVE`.

4.6.3 Example of DML Returning

This section provides two code examples of DML returning.

The following code example illustrates the use of DML returning. In this example, assume that the maximum size of the `name` column is 100 characters. Because the maximum size of the `name` column is known, the `registerReturnParameter(int paramIndex, int externalType, int maxSize)` method is used.

```
...
OraclePreparedStatement pstmt = (OraclePreparedStatement)conn.prepareStatement(
       "delete from tab1 where age < ? returning name into ?");
pstmt.setInt(1,18);

/** register returned parameter
  * in this case the maximum size of name is 100 chars
  */
pstmt.registerReturnParameter(2, OracleTypes.VARCHAR, 100);

// process the DML returning statement
count = pstmt.executeUpdate();
if (count>0)
{
  ResultSet rset = pstmt.getReturnResultSet(); //rest is not null and not empty
  while(rset.next())
  {
    String name = rset.getString(1);
    ...
  }
}
...
```

The following code example also illustrates the use of DML returning. However, in this case, the maximum size of the return parameters is not known. Therefore, the `registerReturnParameter(int paramIndex, int externalType)` method is used.

```
...
OraclePreparedStatement pstmt = (OraclePreparedStatement)conn.prepareStatement(
  "insert into lobtab values (100, empty_clob()) returning col1, col2 into ?, ?");

// register return parameters
pstmt.registerReturnParameter(1, OracleTypes.INTEGER);
pstmt.registerReturnParameter(2, OracleTypes.CLOB);

// process the DML returning SQL statement
pstmt.executeUpdate();
ResultSet rset = pstmt.getReturnResultSet();
int r;
CLOB clob;
if (rset.next())
{
  r = rset.getInt(1);
  System.out.println(r);
  clob = (CLOB)rset.getClob(2);
  ...
}
...
```

4.6.4 Limitations of DML Returning

When using DML returning, be aware of the following:

- It is unspecified what the `getReturnResultSet` method returns when it is invoked more than once. You should not rely on any specific action in this regard.
- The `ResultSet` objects returned from the execution of DML returning statements do not support the `ResultSetMetaData` type. Therefore, the applications must know the information of return parameters before running DML returning statements.
- Streams are not supported with DML returning.
- DML returning cannot be combined with batch update.
- You cannot use both the auto-generated key feature and the DML returning feature in a single SQL DML statement. For example, the following is not allowed:

```
...
PreparedStatement pstmt = conn.prepareStatement('insert into orders (?, ?, ?)
returning order_id into ?");
pstmt.setInt(1, seq01.NEXTVAL);
pstmt.setInt(2, 100);
pstmt.setInt(3, 966431502);
pstmt.registerReturnParam(4, OracleTypes.INTEGER);
pstmt.executeUpdate;
ResultSet rset = pstmt.getGeneratedKeys;
...
```

4.7 Accessing PL/SQL Associative Arrays

Oracle JDBC drivers enable JDBC applications to make PL/SQL calls with Associative Arrays parameters. This section describes the methods to access Associative Arrays.

In PL/SQL, an Associative Array is a set of key-value pairs, where the keys may be `PLS_INTEGER`s or Strings. The keys may have any value and need not be dense. From a client application, you can work only with `PLS_INTEGER` or `BINARY_INTEGER` keys.

> **Note:**
>
> - Associative Arrays were previously known as index-by tables.
> - The `PLS_INTEGER` and `BINARY_INTEGER` are identical data types.
> - When you use String data types, the size is limited to the size in PL/SQL that is 32767 characters. For the server-side internal driver, the limits are lower.

The previous release of Oracle JDBC drivers provided support only for PL/SQL Associative Arrays of Scalar data types. Also, the support was restricted only to the values of the key-value pairs of the Arrays. Starting from Release 18c, Oracle Database supports accessing both the keys (indexes) and values of Associative Arrays, and also provides support for Associative Arrays of object types. Use the following methods to achieve the new functionalities:

- `Array createOracleArray(String arrayTypeName,`

```
                                Object elements)

                                throws SQLException
```

- `ARRAY createARRAY(String typeName,`

```
                        Object elements)

                        throws SQLException
```

> **Note:**
>
> You can use the `createOracleArray` method and the `createARRAY` method for Associative Arrays on Oracle Database release 12*c* and later releases. If you are using an earlier release of Oracle Database, then you should continue using the following deprecated APIs:
>
> - `oracle.jdbc.OraclePreparedStatement.setPlsqlIndexTable`
> - `oracle.jdbc.OracleCallableStatement.getPlsqlIndexTable`
> - `oracle.jdbc.OracleCallableStatement.getOraclePlsqlIndexTable`
> - `oracle.jdbc.OracleCallableStatement.registerIndexTableOutParameter`

In both the preceding methods, the second parameter can either be a `java.util.Map<Integer, ?>` that holds the key-value pairs of the Associative Arrays, or it can only be an array of values. If it is an array of values, then the JDBC driver defaults the indexes to 0,1,2 and so on. If it is `java.util.Map<Integer, ?>`, then the JDBC driver does not default the keys. They remain as specified in the Map, and can be sparse and negative.

- `Map<?,?> oracle.jdbc.OracleArray.getJavaMap();`

This method returns a `Map<?,?>` for the data types in the Associative Array and `null` for Nested Tables and VARRAYs.

> **See Also:**
>
> - *Oracle Database JDBC Java API Reference*
> - *Oracle Database PL/SQL Language Reference* for more information about Associative Arrays