12

# Accessing and Manipulating Oracle Data

This chapter describes Oracle extensions (`oracle.sql.*` formats) and compares them to standard Java formats (`java.sql.*`). Using Oracle extensions involves casting your result sets and statements to `OracleResultSet`, `OracleStatement`, `OraclePreparedStatement`, and `OracleCallableStatement`, as appropriate, and using the `getOracleObject`, `setOracleObject`, `get`*`XXX`*, and `set`*`XXX`* methods of these classes, where *`XXX`* corresponds to the types in the `oracle.sql` package.

This chapter covers the following topics:

- Data Type Mappings
- Data Conversion Considerations
- Result Set and Statement Extensions
- Comparison of Oracle get and set Methods to Standard JDBC
- Using Result Set Metadata Extensions
- About Using SQL CALL and CALL INTO Statements

## 12.1 Data Type Mappings

The Oracle JDBC drivers support standard JDBC types as well as Oracle-specific data types. This section documents standard and Oracle-specific SQL-Java default type mappings. This section contains the following topics:

- Table of Mappings
- Notes Regarding Mappings

### 12.1.1 Table of Mappings

This section describes the default mappings between SQL data types, JDBC type codes, standard Java types, and Oracle extended types.

The SQL Data Types column lists the SQL types that exist in Oracle Database Release 23ai. The JDBC Type Codes column lists data type codes supported by the JDBC standard and defined in the `java.sql.Types` class or by Oracle in the `oracle.jdbc.OracleTypes` class. For standard type codes, the codes are identical in these two classes.

The Standard Java Types column lists standard types defined in the Java language. The Oracle Extension Java Types column lists the `oracle.sql.*` Java types that correspond to each SQL data type in the database. These are Oracle extensions that let you retrieve all SQL data in the form of an `oracle.sql.*` Java type.

Starting with Oracle Database Release 23ai, the JDBC driver supports the `BOOLEAN` SQL type. So, while connecting to a 23ai Database, when the `setBoolean(x)` or `setObject(x)` method is called, where `x` is a Java `boolean`, the value of `x` is sent using this new `Boolean` type representation. This may break existing applications that store `boolean` values in `VARCHAR` or `TINYINT/NUMBER` types because the server may not always convert a `BOOLEAN` native value to

these types. To workaround such situations, use the new connection property, `oracle.jdbc.sendBooleanAsNativeBoolean` that controls how the driver sends a `Boolean` parameter to the Database. If you set the value of this property to `false`, then the driver sends a `boolean` bind as a `NUMBER`, like it did in the earlier versions.

> **Note:**
>
> In general, the Oracle JDBC drivers are optimized to manipulate SQL data using the standard JDBC types. In a few specialized cases, it may be advantageous to use the Oracle extension classes that are available in the `oracle.sql` package. But, Oracle strongly recommends to use the standard JDBC types instead of Oracle extensions, whenever possible.

**Table 12-1 Default Mappings Between SQL Types and Java Types**

| SQL Data Types | JDBC Type Codes | Standard Java Types | Oracle Extension Java Types |
|---|---|---|---|
| BOOLEAN | java.sql.Types.BOOLEAN | java.lang.Boolean or java.lang.boolean | oracle.sql.BOOLEAN |
| CHAR | java.sql.Types.CHAR | java.lang.String | oracle.sql.CHAR |
| VARCHAR2 | java.sql.Types.VARCHAR | java.lang.String | oracle.sql.CHAR |
| LONG | java.sql.Types.LONGVARCHAR | java.lang.String | oracle.sql.CHAR |
| NUMBER | java.sql.Types.NUMERIC | java.math.BigDecimal | oracle.sql.NUMBER |
| NUMBER | java.sql.Types.DECIMAL | java.math.BigDecimal | oracle.sql.NUMBER |
| NUMBER | java.sql.Types.BIT | boolean | oracle.sql.NUMBER |
| NUMBER | java.sql.Types.TINYINT | byte | oracle.sql.NUMBER |
| NUMBER | java.sql.Types.SMALLINT | short | oracle.sql.NUMBER |
| NUMBER | java.sql.Types.INTEGER | int | oracle.sql.NUMBER |
| NUMBER | java.sql.Types.BIGINT | long | oracle.sql.NUMBER |
| NUMBER | java.sql.Types.REAL | float | oracle.sql.NUMBER |
| NUMBER | java.sql.Types.FLOAT | double | oracle.sql.NUMBER |
| NUMBER | java.sql.Types.DOUBLE | double | oracle.sql.NUMBER |
| RAW | java.sql.Types.BINARY | byte[] | oracle.sql.RAW |
| RAW | java.sql.Types.VARBINARY | byte[] | oracle.sql.RAW |
| LONGRAW | java.sql.Types.LONGVARBINARY | byte[] | oracle.sql.RAW |
| DATE | java.sql.Types.DATE | java.sql.Date | oracle.sql.DATE |
| DATE | java.sql.Types.TIME | java.sql.Time | oracle.sql.DATE |
| TIMESTAMP | java.sql.Types.TIMESTAMP | javal.sql.Timestamp | oracle.sql.TIMESTAMP |
| BLOB | java.sql.Types.BLOB | java.sql.Blob | oracle.jdbc.OracleBlob |
| CLOB | java.sql.Types.CLOB | java.sql.Clob | oracle.jdbc.OracleClob |
| user-defined object | java.sql.Types.STRUCT | java.sql.Struct | oracle.jdbc.OracleStruct[1] |
| user-defined reference | java.sql.Types.REF | java.sql.Ref | oracle.jdbc.OracleRef |

**Table 12-1 (Cont.) Default Mappings Between SQL Types and Java Types**

| SQL Data Types | JDBC Type Codes | Standard Java Types | Oracle Extension Java Types |
|---|---|---|---|
| user-defined collection | java.sql.Types.ARRAY | java.sql.Array | oracle.jdbc.OracleArray [2] |
| ROWID | java.sql.Types.ROWID | java.sql.RowId | oracle.sql.ROWID |
| NCLOB | java.sql.Types.NCLOB | java.sql.NClob | oracle.sql.NCLOB |
| NCHAR | java.sql.Types.NCHAR | java.lang.String | oracle.sql.CHAR |
| BFILE | oracle.jdbc.OracleTypes.BFILE (ORACLE EXTENSION) | NA | oracle.sql.BFILE |
| REF CURSOR | oracle.jdbc.OracleTypes.CURSOR (ORACLE EXTENSION) | java.sql.ResultSet | oracle.jdbc.OracleResultSet |
| TIMESTAMP | oracle.jdbc.OracleTypes.TIMESTAMP (ORACLE EXTENSION) | java.sql.Timestamp | oracle.sql.TIMESTAMP |
| TIMESTAMP WITH TIME ZONE | oracle.jdbc.OracleTypes.TIMESTAMPTZ (ORACLE EXTENSION) | java.sql.Timestamp | oracle.sql.TIMESTAMPTZ |
| TIMESTAMP WITH LOCAL TIME ZONE | oracle.jdbc.OracleTypes.TIMESTAMPLTZ (ORACLE EXTENSION) | java.sql.Timestamp | oracle.sql.TIMESTAMPLTZ |

1

2

**Related Topics**

- Standard Types Versus Oracle Types
- Supported SQL-JDBC Data Type Mappings
  This section lists all the possible Java types to which a given SQL data type can have a valid mapping.

## 12.1.2 Notes Regarding Mappings

This section provides further details regarding mappings for `NUMBER` and user-defined types.

### NUMBER Types

For the different type codes that an Oracle `NUMBER` value can correspond to, call the getter routine that is appropriate for the size of the data for mapping to work properly. For example, call `getByte` to get a Java `tinyint` value for an item *x*, where -128 < *x* < 128.

### User-Defined Types

User-defined types, such as objects, object references, and collections, map by default to weak Java types, such as `java.sql.Struct`, but alternatively can map to strongly typed custom Java classes. Custom Java classes can implement one of two interfaces:

- The standard `java.sql.SQLData`

- The Oracle-specific `oracle.jdbc.OracleData`

**Related Topics**

- About Mapping Oracle Objects
- About Creating and Using Custom Object Classes for Oracle Objects

# 12.2 Data Conversion Considerations

When JDBC programs retrieve SQL data into Java, you can use standard Java types, or you can use types of the `oracle.sql` package. This section covers the following topics:

- Standard Types Versus Oracle Types
- About Converting SQL NULL Data
- About Testing for NULLs

## 12.2.1 Standard Types Versus Oracle Types

The Oracle data types in `oracle.sql` store data in the same bit format as used by the database. In versions of the Oracle JDBC drivers prior to Oracle Database 10*g*, the Oracle data types were generally more efficient. Starting from Oracle Database 10*g*, the JDBC drivers were substantially updated. As a result, in most cases the standard Java types are preferred to the data types in `oracle.sql.*`. In particular, `java.lang.String` is much more efficient than `oracle.sql.CHAR`.

In general, Oracle recommends that you use the Java standard types. The exceptions to this are:

- Use the `oracle.jdbc.OracleData` rather than the `java.sql.SqlData` if the `OracleData` functionality better suits your needs.
- Use `oracle.sql.NUMBER` rather than `java.lang.Double` if you need to retain the exact values of floating point numbers. Oracle `NUMBER` is a decimal representation and Java `Double` and `Float` are binary representations. Conversion from one format to the other can result in slight variations in the actual value represented. Additionally, the range of values that can be represented using the two formats is different.

  Use `oracle.sql.NUMBER` rather than `java.math.BigDecimal` when performance is critical and you are not manipulating the values, just reading and writing them.
- Use `oracle.sql.DATE` or `oracle.sql.TIMESTAMP` if you are using a JDK version earlier than JDK 6. Use `java.sql.Date` or `java.sql.Timestamp` if you are using JDK 6 or a later version.

  > **Note:**
  >
  > Due to a bug in all versions of Java prior to JDK 6, construction of `java.lang.Date` and `java.lang.Timestamp` objects is slow, especially in multithreaded environments. This bug is fixed in JDK 6.

- Use `oracle.sql.CHAR` only when you have data from some external source, which has been represented in an Oracle character set encoding. In all other cases, you should use `java.lang.String`.

- `STRUCT`, `ARRAY`, `BLOB`, `CLOB`, `REF`, and `ROWID` are all the implementation classes of the corresponding JDBC standard interface types. So, there is no benefit of using the Oracle extension types as they are identical to the JDBC standard types.
- `BFILE`, `TIMESTAMPTZ`, and `TIMESTAMPLTZ` have no representation in the JDBC standard. You must use these Oracle extensions.
- In all other cases, you should use the standard JDBC type rather than the Oracle extensions.

> **Note:**
>
> If you convert an `oracle.sql` data type to a Java standard data type, then the benefits of using the `oracle.sql` data type are lost.

## 12.2.2 About Converting SQL NULL Data

Java represents a SQL `NULL` datum by the Java value `null`. Java data types fall into two categories: primitive types, such as `byte`, `int`, and `float`, and object types, such as class instances. The primitive types cannot represent `null`. Instead, they store `null` as the value zero, as defined by the JDBC specification. This can lead to ambiguity when you try to interpret your results.

In contrast, Java object types can represent `null`. The Java language defines an object container type corresponding to every primitive type that can represent `null`. The object container types must be used as the targets for SQL data to detect SQL `NULL` without ambiguity.

## 12.2.3 About Testing for NULLs

You cannot use a relational operator to compare `NULL` values with each other or with other values. For example, the following `SELECT` statement does not return any row even if the `COMMISSION_PCT` column contains one or more `NULL` values.

```
PreparedStatement pstmt = conn.prepareStatement(
  "SELECT * FROM EMPLOYEES WHERE COMMISSION_PCT = ?");
pstmt.setNull(1, java.sql.Types.VARCHAR);
```

The next example shows how to compare values for equality when some return values might be `NULL`. The following code returns all the `FIRST_NAME` from the `EMPLOYEES` table that are `NULL`, if there is no value of 205 for `COMM`.

```
PreparedStatement pstmt = conn.prepareStatement("SELECT FIRST_NAME FROM EMPLOYEES
  WHERE COMMISSION_PCT =? OR  ((COMM IS NULL) AND (? IS NULL))");
pstmt.setBigDecimal(1, new BigDecimal(205));
pstmt.setNull(2, java.sql.Types.VARCHAR);
```

# 12.3 Result Set and Statement Extensions

The `Statement` object returns a `java.sql.ResultSet`. If you want to apply only standard JDBC methods to the object, then keep it as a `ResultSet` type. However, if you want to use the Oracle extensions on the object, then you must cast it to `OracleResultSet`. All of the Oracle

Result Set extensions are in the `oracle.jdbc.OracleResultSet` interface and all the `Statement` extensions are in the `oracle.jdbc.OracleStatement` interface.

For example, assuming you have a standard `Statement` object `stmt`, do the following if you want to use only standard JDBC `ResultSet` methods:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM employees");
```

If you need the extended functionality provided by the Oracle extensions to JDBC, you can select the results into a standard `ResultSet` variable and then cast that variable to `OracleResultSet` later.

Key extensions to the result set and statement classes include the `getOracleObject` and `setOracleObject` methods, used to access and manipulate data in `oracle.sql.*` formats.

# 12.4 Comparison of Oracle get and set Methods to Standard JDBC

This section describes `get` and `set` methods, particularly the JDBC standard `getObject` and `setObject` methods and the Oracle-specific `getOracleObject` and `setOracleObject` methods, and how to access data in `oracle.sql.*` format compared with Java format.

You can use the standard `get`*`XXX`* methods for all Oracle SQL types.

This section covers the following topics:



> **Note:**
>
> You cannot qualify a column name with a table name and pass it as a parameter to the `get`*`XXX`* method. For example:
>
> ```
> ResultSet rset = stmt.executeQuery("SELECT employees.department_id,
> department.department_id FROM employees, department");
> rset.getInt("employees.department_id");
> ```
>
> The `getInt` method in the preceding code will throw an exception. To uniquely identify the columns in the `get`*`XXX`* method, you can either use column index or specify column aliases in the query and use these aliases in the `get`*`XXX`* method.

## 12.4.1 Standard getObject Method

The standard `getObject` method of a result set or callable statement has a return type of `java.lang.Object`. The class of the object returned is based on its SQL type, as follows:

- For SQL data types that are not Oracle-specific, the `getObject` method returns the default Java type corresponding to the SQL type of the column, following the mapping in the JDBC specification.
- For Oracle-specific data types, `getObject` returns an object of the appropriate `oracle.sql.*` class, such as `oracle.sql.ROWID`.
- For Oracle database objects, `getObject` returns a Java object of the class specified in your type map. Type maps specify a mapping from database named types to Java classes. The `getObject(parameter_index)` method uses the default type map of the connection. The `getObject(parameter_index, map)` enables you to pass in a type map. If the type map does not provide a mapping for a particular Oracle object, then `getObject` returns an `oracle.sql.OracleStruct` object.

## 12.4.2 Oracle getOracleObject Method

If you want to retrieve data from a result set or callable statement as an `oracle.sql.*` object, then you must follow a special process. For an `OracleResultSet` object, you must cast the Result Set to `oracle.jdbc.OracleResultSet` and then call `getOracleObject` instead of `getObject`. The same applies to `CallableStatement` and `oracle.jdbc.OracleCallableStatement`.

The return type of `getOracleObject` is `oracle.sql.Datum`. The actual returned object is an instance of the appropriate `oracle.sql.*` class. The method signature is:

```
public oracle.sql.Datum getOracleObject(int parameter_index)
```

When you retrieve data into a `Datum` variable, you can use the standard Java `instanceof` operator to determine which `oracle.sql.*` type it really is.

**Example: Using getOracleObject with a Result Set**

The following example creates a table that contains a column of `CHAR` data and a column containing a `BFILE` locator. A `SELECT` statement retrieves the contents of the table as a result set. The `getOracleObject` then retrieves the `CHAR` data into the `char_datum` variable and the `BFILE` locator into the `bfile_datum` variable. Note that because `getOracleObject` returns a `Datum` object, the return values must be cast to `CHAR` and `BFILE`, respectively.

```
stmt.execute ("CREATE TABLE bfile_table (x VARCHAR2 (30), b BFILE)");
stmt.execute
    ("INSERT INTO bfile_table VALUES ('one', BFILENAME ('TEST_DIR', 'file1'))");

ResultSet rset = stmt.executeQuery ("SELECT * FROM bfile_table");
while (rset.next ())
{
   CHAR char_datum = (CHAR) ((OracleResultSet)rset).getOracleObject (1);
   BFILE bfile_datum = (BFILE) ((OracleResultSet)rset).getOracleObject (2);
   ...
}
```

**Example: Using getOracleObject in a Callable Statement**

The following example prepares a call to the procedure `myGetDate`, which associates a character string with a date. The program passes `"HR"` to the prepared call and registers the `DATE` type as an output parameter. After the call is run, `getOracleObject` retrieves the date associated with `"HR"`. Note that because `getOracleObject` returns a `Datum` object, the results are cast to `DATE`.

```
OracleCallableStatement cstmt = (OracleCallableStatement)conn.prepareCall
                                   ("begin myGetDate (?, ?); end;");

cstmt.setString (1, "HR");
cstmt.registerOutParameter (2, Types.DATE);
cstmt.execute ();

DATE date = (DATE) ((OracleCallableStatement)cstmt).getOracleObject (2);
...
```

## 12.4.3 Summary of getObject and getOracleObject Return Types

This section lists the underlying return types for the `getObject` and `getOracleObject` methods for each Oracle SQL type.

Keep in mind the following when you use these methods:

- `getObject` always returns data into a `java.lang.Object` instance
- `getOracleObject` always returns data into an `oracle.sql.Datum` instance

You must cast the returned object to use any special functionality.

**Table 12-2 getObject and getOracleObject Return Types**

| Oracle SQL Type | getObject Underlying Return Type | getOracleObject Underlying Return Type |
|---|---|---|
| CHAR | String | oracle.sql.CHAR |
| VARCHAR2 | String | oracle.sql.CHAR |
| NCHAR | String | oracle.sql.CHAR |
| LONG | String | oracle.sql.CHAR |
| NUMBER | java.math.BigDecimal | oracle.sql.NUMBER |
| RAW | byte[] | oracle.sql.RAW |
| LONGRAW | byte[] | oracle.sql.RAW |
| DATE | java.sql.Date | oracle.sql.DATE |
| TIMESTAMP | java.sql.Timestamp[1] | oracle.sql.TIMESTAMP |
| TIMESTAMP WITH TIME ZONE | oracle.sql.TIMESTAMPTZ | oracle.sql.TIMESTAMPTZ |
| TIMESTAMP WITH LOCAL TIME ZONE | oracle.sql.TIMESTAMPLTZ | oracle.sql.TIMESTAMPLTZ |
| BINARY_FLOAT | java.lang.Float | oracle.sql.BINARY_FLOAT |
| BINARY_DOUBLE | java.lang.Double | oracle.sql.BINARY_DOUBLE |

**Table 12-2 (Cont.) getObject and getOracleObject Return Types**

| Oracle SQL Type | getObject Underlying Return Type | getOracleObject Underlying Return Type |
|---|---|---|
| INTERVAL DAY TO SECOND | oracle.sql.INTERVALDS | oracle.sql.INTERVALDS |
| INTERVAL YEAR TO MONTH | oracle.sql.INTERVALYM | oracle.sql.INTERVALYM |
| ROWID | oracle.sql.ROWID | oracle.sql.ROWID |
| REF CURSOR | java.sql.ResultSet | (not supported) |
| BLOB | oracle.jdbc.OracleBlob[2] | oracle.jdbc.OracleBlob |
| CLOB | oracle.jdbc.OracleClob[3] | oracle.jdbc.OracleClob |
| NCLOB | java.sql.NClob | oracle.sql.NCLOB |
| BFILE | oracle.sql.BFILE | oracle.sql.BFILE |
| Oracle object | class specified in type map or oracle.sql.OracleStruct[4] (if no type map entry) | oracle.jdbc.OracleStruct |
| Oracle object reference | oracle.jdbc.OracleRef[5] | oracle.jdbc.OracleRef |
| collection (varray or nested table) | oracle.jdbc.OracleArray[6] | oracle.sql.ARRAY |

1 ResultSet.getObject returns java.sql.Timestamp only if the oracle.jdbc.J2EE13Compliant connection property is set to TRUE, else the method returns oracle.sql.TIMESTAMP.

2 Starting from Oracle Database 12*c* Release 1 (12.1), the oracle.sql.BLOB class is deprecated and replaced with the oracle.jdbc.OracleBlob interface.

3 Starting from Oracle Database 12*c* Release 1 (12.1), the oracle.sql.CLOB class is deprecated and replaced with the oracle.jdbc.OracleClob interface.

4 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.

5 Starting from Oracle Database 12*c* Release 1, the oracle.sql.REF class is deprecated and is replaced with the oracle.jdbc.OracleRef interface.

6 Starting from Oracle Database 12*c* Release 1, the oracle.sql.ARRAY class is deprecated and replaced with the oracle.jdbc.OracleArray interface.

> **Note:**
>
> The `ResultSet.getObject` method returns `java.sql.Timestamp` for the `TIMESTAMP` SQL type, only when the connection property `oracle.jdbc.J2EE13Compliant` is set to `TRUE`. This property has to be set when the connection is obtained. If this connection property is not set or if it is set after the connection is obtained, then the `ResultSet.getObject` method returns `oracle.sql.TIMESTAMP` for the `TIMESTAMP` SQL type.
>
> The `oracle.jdbc.J2EE13Compliant` connection property can also be set without changing the code. For this, set the system property by calling the `java` command with the flag `-Doracle.jdbc.J2EE13Compliant=true`. For example:
>
> ```
> java -Doracle.jdbc.J2EE13Compliant=true ...
> ```
>
> When the `J2EE13Compliant` property is set to `TRUE`, then the action is similar to Table B-3 of the JDBC specification.

**Related Topics**

- Supported SQL-JDBC Data Type Mappings
  This section lists all the possible Java types to which a given SQL data type can have a valid mapping.

## 12.4.4 Other getXXX Methods

Standard JDBC provides a `get`*`XXX`* for each standard Java type, such as `getByte`, `getInt`, `getFloat`, and so on. Each of these returns exactly what the method name implies.

In addition, the `OracleResultSet` and `OracleCallableStatement` interfaces provide a full complement of `get`*`XXX`* methods corresponding to all the `oracle.sql.*` types. Each `get`*`XXX`* method returns an `oracle.sql.`*`XXX`* object. For example, `getROWID` returns an `oracle.sql.ROWID` object.

There is no performance advantage in using the specific `get`*`XXX`* methods. However, they do save you the trouble of casting, because the return type is specific to the object being returned.

This section covers the following topics:

- Return Types of getXXX Methods
- Special Notes about getXXX Methods

### 12.4.4.1 Return Types of getXXX Methods

Refer to the JDBC Javadoc to know the return types for each `get`*`XXX`* method and also which are Oracle extensions under Java Development Kit (JDK) 6. You must cast the returned object to `OracleResultSet` or `OracleCallableStatement` to use methods that are Oracle extensions.

### 12.4.4.2 Special Notes about getXXX Methods

This section provides additional details about some `get`*`XXX`* methods.

#### getBigDecimal

JDBC 2.0 simplified method signatures for the `getBigDecimal` method. The previous input signatures were:

```
(int columnIndex, int scale) or (String columnName, int scale)
```

The simplified input signature is:

```
(int columnIndex) or (String columnName)
```

The `scale` parameter, used to specify the number of digits to the right of the decimal, is no longer necessary. The Oracle JDBC drivers retrieve numeric values with full precision.

#### getBoolean

If the column from which you are trying to fetch the data is not of `BOOLEAN` type, which is introduced in Oracle Database 23ai Release, then the use of the `getBoolean` method results in a data type conversion. Apart from the `BOOLEAN` type, the `getBoolean` method also supports the following columns:

- `INT`
- `CHAR`
- `NCHAR`
- `NUMBER`
- `VARCHAR`
- `NVARCHAR`

When you apply this method to the supported data types, then the `getBoolean` method interprets any zero value as `false` and any other value as `true`. When applied to any other column, apart from the supported columns, the `getBoolean` method raises the exception `java.lang.NumberFormatException`.

> **See Also:**
>
> Class oracle.jdbc.OracleTypes for more information about the `BOOLEAN` data type

## 12.4.5 Data Types For Returned Objects from getObject and getXXX

The return type of `getObject` is `java.lang.Object`. The returned value is an instance of a subclass of `java.lang.Object`. Similarly, the return type of `getOracleObject` is `oracle.sql.Datum`, and the class of the returned value is a subclass of `oracle.sql.Datum`. You typically cast the returned object to the appropriate class to use particular methods and functionality of that class.

In addition, you have the option of using a specific `get`*`XXX`* method instead of the generic `getObject` or `getOracleObject` methods. The `get`*`XXX`* methods enable you to avoid casting, because the return type of `get`*`XXX`* corresponds to the type of object returned. For example, the return type of `getCLOB` is `oracle.sql.CLOB`, as opposed to `java.lang.Object`.

#### Example of Casting Return Values

This example assumes that you have fetched data of the `NUMBER` type as the first column of a result set. Because you want to manipulate the `NUMBER` data without losing precision, cast your result set to `OracleResultSet` and use `getOracleObject` to return the `NUMBER` data in `oracle.sql.*` format. If you do not cast your result set, then you have to use `getObject`, which returns your numeric data into a Java `Float` and loses some of the precision of your SQL data.

The `getOracleObject` method returns an `oracle.sql.NUMBER` object into an `oracle.sql.Datum` return variable unless you cast the output. Cast the `getOracleObject` output to `oracle.sql.NUMBER` if you want to use a `NUMBER` return variable and any of the special functionality of that class.

```
NUMBER x = (NUMBER)ors.getOracleObject(1);
```

## 12.4.6 The setObject and setOracleObject Methods

Just as there is a standard `getObject` and Oracle-specific `getOracleObject` in result sets and callable statements, there are also standard `setObject` and Oracle-specific `setOracleObject` methods in `OraclePreparedStatement` and `OracleCallableStatement`. The `setOracleObject` methods take `oracle.sql.*` input parameters.

To bind standard Java types to a prepared statement or callable statement, use the `setObject` method, which takes a `java.lang.Object` as input. The `setObject` method does support a few of the `oracle.sql.*` types. However, the method has been implemented so that you can enter instances of the `oracle.sql.*` classes that correspond to the following JDBC standard types: `Blob`, `Clob`, `Struct`, `Ref`, and `Array`.

To bind `oracle.sql.*` types to a prepared statement or callable statement, use the `setOracleObject` method, which takes a subclass of `oracle.sql.Datum` as input. To use `setOracleObject`, you must cast your prepared statement or callable statement to `OraclePreparedStatement` or `OracleCallableStatement`.

#### Example of Using setObject and setOracleObject

For a prepared statement, the `setOracleObject` method binds the `oracle.sql.CHAR` data represented by the `charVal` variable to the prepared statement. To bind the `oracle.sql.*` data, the prepared statement must be cast to `OraclePreparedStatement`. Similarly, the `setObject` method binds the Java `String` data represented by the variable `strVal`.

```
PreparedStatement ps= conn.prepareStatement("text_of_prepared_statement");
((OraclePreparedStatement)ps).setOracleObject(1,charVal);
ps.setObject(2,strVal);
```

## 12.4.7 Other setXXX Methods

As with the `get`*`XXX`* methods, there are several specific `set`*`XXX`* methods. Standard `set`*`XXX`* methods are provided for binding standard Java types, and Oracle-specific `set`*`XXX`* methods are provided for binding Oracle-specific types.

Similarly, there are two forms of the `setNull` method:

- `void setNull(int` *`parameterIndex`*`, int` *`sqlType`*`)`

  This is specified in the standard `java.sql.PreparedStatement` interface. This signature takes a parameter index and a SQL type code defined by the `java.sql.Types` or

`oracle.jdbc.OracleTypes` class. Use this signature to set an object other than a `REF`, `ARRAY`, or `STRUCT` to `NULL`.

- `void setNull(int parameterIndex, int sqlType, String sql_type_name)`

  With JDBC 2.0, this signature is also specified in the standard `java.sql.PreparedStatement` interface. This method takes a SQL type name in addition to a parameter index and a SQL type code. Use this method when the SQL type code is `java.sql.Types.REF`, `ARRAY`, or `STRUCT`. If the type code is other than `REF`, `ARRAY`, or `STRUCT`, then the given SQL type name is ignored.

Similarly, the `registerOutParameter` method has a signature for use with `REF`, `ARRAY`, or `STRUCT` data:

```
void registerOutParameter
            (int parameterIndex, int sqlType, String sql_type_name)
```

Binding Oracle-specific types using the appropriate `setXXX` methods, instead of the methods used for binding standard Java types, may offer some performance advantage.

This section covers the following topics:

- Input Data Binding
- Method setFixedCHAR for Binding CHAR Data into WHERE Clauses

## 12.4.7.1 Input Data Binding

There are three way to bind data for input:

- Direct binding where the data itself is placed in a bind buffer
- Stream binding where the data is streamed
- LOB binding where a temporary lob is created, the data placed in the LOB using the LOB APIs, and the bytes of the LOB locator are placed in the bind buffer

The three kinds of binding have some differences in performance and have an impact on batching. Direct binding is fast and batching is fine. Stream binding is slower, may require multiple round trips, and turns batching off. LOB binding is very slow and requires many round trips. Batching works, but might be a bad idea. They also have different size limits, depending on the type of the SQL statement.

For SQL parameters, the length of standard parameter types, such as `RAW` and `VARCHAR2`, is fixed by the size of the target column. For PL/SQL parameters, the size is limited to a fixed number of bytes, which is 32766.

In Oracle Database 10*g* release 2, certain changes were made to the `setString`, `setCharacterStream`, `setAsciiStream`, `setBytes`, and `setBinaryStream` methods of `PreparedStatement`. The original behavior of these APIs were:

- `setString`: Direct bind of characters
- `setCharacterStream`: Stream bind of characters
- `setAsciiStream`: Stream bind of bytes
- `setBytes`: Direct bind of bytes
- `setBinaryStream`: Stream bind of bytes

Starting from Oracle Database 10*g* Release 2, automatic switching between binding modes, based on the data size and on the type of the SQL statement is provided.

#### setBytes and setBinaryStream

For SQL, direct bind is used for size up to 2000 and stream bind for larger.

For PL/SQL direct bind is used for size up to 32766 and LOB bind is used for larger.

#### setString, setCharacterStream, and setAsciiStream

For SQL, direct bind is used up to 32766 Java characters and stream bind is used for larger. This is independent of character set.

For PL/SQL, you must be careful about the byte size of the character data in the database character set or the national character set depending on the setting of the form of use parameter. Direct bind is used for data where the byte length is less than 32766 and LOB bind is used for larger.

For fixed length character sets, multiply the length of the Java character data by the fixed character size in bytes and compare that to the restrictive values. For variable length character sets, there are three cases based on the Java character length, as follows:

- If character length is less than 32766 divided by the maximum character size, then direct bind is used.
- If character length is greater than 32766 divided by the minimum character size, then LOB bind is used.
- If character length is in between and if the actual length of the converted bytes is less than 32766, then direct bind is used, else LOB bind is used.

> **Note:**
>
> When a PL/SQL procedure is embedded in a SQL statement, the binding action is different.

The server-side internal driver has the following additional limitations:

- `setString`, `setCharacterStream`, and `setASCIIStream` APIs are not supported for SQL CLOB columns when the data size in characters is over 32767 bytes
- `setBytes` and `setBinaryStream` APIs are not supported for SQL BLOB columns when the data size is over 32767 bytes

> **Note:**
>
> Do not use these APIs with the server-side internal driver, without careful checking of the data size in client code.

#### Related Topics

- Data Interface for LOBs
  The data interface for LOBs includes a set of Java and OCI APIs that are extended to work with LOB data types.

> **See Also:**
>
> JDBC Release Notes for further discussion and possible workarounds

## 12.4.7.2 Method setFixedCHAR for Binding CHAR Data into WHERE Clauses

`CHAR` data in the database is padded to the column width. This leads to a limitation in using the `setCHAR` method to bind character data into the `WHERE` clause of a `SELECT` statement. The character data in the `WHERE` clause must also be padded to the column width to produce a match in the `SELECT` statement. This is especially troublesome if you do not know the column width.

To remedy this, Oracle has added the `setFixedCHAR` method to the `OraclePreparedStatement` class. This method runs a non-padded comparison.

> **Note:**
>
> - Remember to cast your prepared statement object to `OraclePreparedStatement` to use the `setFixedCHAR` method.
> - There is no need to use `setFixedCHAR` for an `INSERT` statement. The database always automatically pads the data to the column width as it inserts it.

**Example**

The following example demonstrates the difference between the `setCHAR` and `setFixedCHAR` methods.

```
/* Schema is :
 create table my_table (col1 char(10));
 insert into my_table values ('JDBC');
*/
 PreparedStatement pstmt = conn.prepareStatement
                   ("select count(*) from my_table where col1 = ?");

 pstmt.setString (1, "JDBC");  // Set the Bind Value
 runQuery (pstmt);             // This will print " No of rows are 0"

 CHAR ch = new CHAR("JDBC      ", null);
 ((OraclePreparedStatement)pstmt).setCHAR(1, ch); // Pad it to 10 bytes
 runQuery (pstmt);             // This will print "No of rows are 1"

 ((OraclePreparedStatement)pstmt).setFixedCHAR(1, "JDBC");
  runQuery (pstmt);            // This will print "No of rows are 1"

 void runQuery (PreparedStatement ps)
 {
   // Run the Query
   ResultSet rs = pstmt.executeQuery ();

   while (rs.next())
     System.out.println("No of rows are " + rs.getInt(1));

   rs.close();
```

```
    rs = null;
  }
```

# 12.5 Using Result Set Metadata Extensions

The `oracle.jdbc.OracleResultSetMetaData` interface is JDBC 2.0-compliant but does not implement the `getSchemaName` and `getTableName` methods because Oracle Database does not make this feasible.

The following code snippet uses several of the methods in the `OracleResultSetMetadata` interface to retrieve the number of columns from the `EMPLOYEES` table and the numerical type and SQL type name of each column:

```
DatabaseMetaData dbmd = conn.getMetaData();
ResultSet rset = dbmd.getTables("", "HR", "EMPLOYEES", null);

 while (rset.next())
 {
   OracleResultSetMetaData orsmd = ((OracleResultSet)rset).getMetaData();
   int numColumns = orsmd.getColumnCount();
   System.out.println("Num of columns = " + numColumns);

   for (int i=0; i<numColumns; i++)
   {
     System.out.print ("Column Name=" + orsmd.getColumnName (i+1));
     System.out.print (" Type=" + orsmd.getColumnType (i + 1) );
     System.out.println (" Type Name=" + orsmd.getColumnTypeName (i + 1));
  }
}
```

The program returns the following output:

```
Num of columns = 5
Column Name=TABLE_CAT Type=12 Type Name=VARCHAR2
Column Name=TABLE_SCHEM Type=12 Type Name=VARCHAR2
Column Name=TABLE_NAME Type=12 Type Name=VARCHAR2
Column Name=TABLE_TYPE Type=12 Type Name=VARCHAR2
Column Name=TABLE_REMARKS Type=12 Type Name=VARCHAR2
```

# 12.6 About Using SQL CALL and CALL INTO Statements

You can use the CALL statement to execute a routine from within SQL in the following two ways:

> **Note:**
>
> A routine is a procedure or a function that is standalone or is defined within a type or package. You must have `EXECUTE` privilege on the standalone routine or on the type or package in which the routine is defined.

- By issuing a call to the routine itself by name or by using the `routine_clause`
- By using an `object_access_expression` inside the type of an expression

You can specify one or more arguments to the routine, if the routine takes arguments. You can use positional, named, or mixed notation for argument.

### CALL INTO Statement

The `INTO` clause applies only to calls to functions. You can use the following types of variables with this clause:

- Host variable
- Indicator variable

> **See Also:**
>
> *Oracle Database SQL Language Reference* for more information

### PL/SQL Blocks

The basic unit in PL/SQL is a block. All PL/SQL programs are made up of blocks, which can be nested within each other. A PL/SQL block has three parts: a declarative part, an executable part, and an exception-handling part. You get the following advantages by using PL/SQL blocks in your application:

- Better performance
- Higher productivity
- Full portability
- Tight integration with Oracle
- Tight security