18
Working with Oracle Collections

This chapter describes Oracle extensions to standard Java Database Connectivity (JDBC) that let you access and manipulate Oracle collections, which map to Java arrays, and their data. The following topics are discussed:

> **Note:**
>
> Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.ARRAY` class is deprecated and replaced with the `oracle.jdbc.OracleArray` 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.OracleArray` interface.

18.1 Oracle Extensions for Collections

This section covers the following topics:

18.1.1 Overview of Oracle Collections

An Oracle collection, either a variable array (VARRAY) or a nested table in the database, maps to an array in Java. JDBC 2.0 arrays are used to materialize Oracle collections in Java. The terms collection and array are sometimes used interchangeably. However, collection is more appropriate on the database side and array is more appropriate on the JDBC application side.

Oracle supports only named collections, where you specify a SQL type name to describe a type of collection. JDBC enables you to use arrays as any of the following:

- Columns in a `SELECT` clause
- `IN` or `OUT` bind variables

- Attributes in an Oracle object
- Elements of other arrays

18.1.2 Choices in Materializing Collections

In your application, you have the choice of materializing a collection as an instance of the `oracle.sql.ARRAY` class, which is weakly typed, or materializing it as an instance of a custom Java class that you have created in advance, which is strongly typed. Custom Java classes used for collections are referred to as custom collection classes. A custom collection class must implement the Oracle `oracle.jdbc.OracleData` interface. In addition, the custom class or a companion class must implement `oracle.jdbc.OracleDataFactory`. The standard `java.sql.SQLData` interface is for mapping SQL object types only.

The `oracle.sql.ARRAY` class implements the standard `java.sql.Array` interface.

The `ARRAY` class 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. However, you cannot write to the array, because there are no setter methods.

Custom collection classes, as with the `ARRAY` class, enable you to retrieve all or part of the array and get the SQL base type name. They also have the advantage of being strongly typed, which can help you find coding errors during compilation that might not otherwise be discovered until run time.

> **Note:**
>
> There is no difference in the code between accessing VARRAYs and accessing nested tables. `ARRAY` class methods can determine if they are being applied to a VARRAY or nested table, and respond by taking the appropriate actions.

18.1.3 Creating Collections

Because Oracle supports only named collections, you must declare a particular `VARRAY` type name or nested table type name. VARRAY and nested table are not types themselves, but categories of types.

A SQL type name is assigned to a collection when you create it using the SQL `CREATE TYPE` statement:

```
CREATE TYPE <sql_type_name> AS <datatype>;
```

A VARRAY is an array of varying size. It has an ordered set of data elements, and all the elements are of the same data type. Each element has an index, which is a number corresponding to the position of the element in the VARRAY. The number of elements in a VARRAY is the size of the VARRAY. You must specify a maximum size when you declare the `VARRAY` type. For example:

```
CREATE TYPE myNumType AS VARRAY(10) OF NUMBER;
```

This statement defines `myNumType` as a SQL type name that describes a VARRAY of `NUMBER` values that can contain no more than 10 elements.

A nested table is an unordered set of data elements, all of the same data type. The database stores a nested table in a separate table which has a single column, and the type of that

column is a built-in type or an object type. If the table is an object type, then it can also be viewed as a multi-column table, with a column for each attribute of the object type. You can create a nested table as follows:

```
CREATE TYPE myNumList AS TABLE OF integer;
```

This statement identifies `myNumList` as a SQL type name that defines the table type used for the nested tables of the type `INTEGER`.

18.1.4 Creating Multilevel Collection Types

The most common way to create a new multilevel collection type in JDBC is to pass the SQL `CREATE TYPE` statement to the `execute` method of the `java.sql.Statement` class. The following code creates a one-level nested table, `first_level`, and a two- levels nested table, `second_level`:

```
Connection conn = ....                                  // make a database
                                                        // connection
Statement stmt = conn.createStatement();                // open a database
                                                        // cursor
stmt.execute("CREATE TYPE first_level AS TABLE OF NUMBER");  // create a nested
                                                        // table of number
stmt.execute("CREATE TYPE second_level AS TABLE OF first_level"); // create a
           // two-levels nested table
...        // other operations here
stmt.close();                                           // release the
                                                        // resource
conn.close();                                           // close the
                                                        // database connection
```

Once the multilevel collection types have been created, they can be used as both columns of a base table as well as attributes of a object type.

> **Note:**
>
> Multilevel collection types are available only for Oracle9*i* and later.

18.2 Overview of Collection Functionality

You can obtain collection data in an array instance through a result set or callable statement and pass it back as a bind variable in a prepared statement or callable statement.

The `oracle.sql.ARRAY` class, which implements the standard `java.sql.Array` interface, provides the necessary functionality to access and update the data of an Oracle collection.

This section covers Array Getter and Setter Methods. Use the following result set, callable statement, and prepared statement methods to retrieve and pass collections as Java arrays.

> **Note:**
>
> Starting from Oracle Database 12*c* Release 1 (12.1), the `oracle.sql.ARRAY` class is deprecated and replaced with the `oracle.jdbc.OracleArray` 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.OracleArray` interface.

Result Set and Callable Statement Getter Methods

The `OracleResultSet` and `OracleCallableStatement` interfaces support `getARRAY` and `getArray` methods to retrieve `ARRAY` objects as output parameters, either as `oracle.sql.ARRAY` instances or `java.sql.Array` instances. You can also use the `getObject` method. These methods take as input a `String` column name or `int` column index.

> **Note:**
>
> The Oracle JDBC drivers cache array and structure descriptors. This provides enormous performance benefits; however, it means that if you change the underlying type definition of an array type in the database, the cached descriptor for that array type will become stale and your application will receive a `SQLException`.

Prepared and Callable Statement Setter Methods

The `OraclePreparedStatement` and `OracleCallableStatement` classes support `setARRAY` and `setArray` methods to take updated `ARRAY` objects as bind variables and pass them to the database. You can also use the `setObject` method. These methods take as input a `String` parameter name or `int` parameter index as well as an `oracle.sql.ARRAY` instance or a `java.sql.Array` instance.

18.3 ARRAY Performance Extension Methods

This section discusses the following topics:

18.3.1 About Accessing oracle.sql.ARRAY Elements as Arrays of Java Primitive Types

The `oracle.sql.ARRAY` class contains methods that return array elements as Java primitive types. These methods enable you to access collection elements more efficiently than accessing them as `Datum` instances and then converting each `Datum` instance to its Java primitive value.

> **Note:**
>
> These specialized methods of the `oracle.sql.ARRAY` class are restricted to numeric collections.

Each method using the first signature returns collection elements as an *`XXX`*`[]`, where *`XXX`* is a Java primitive type. Each method using the second signature returns a slice of the collection containing the number of elements specified by `count`, starting at the `index` location.

18.3.2 ARRAY Automatic Element Buffering

Oracle JDBC driver provides public methods to enable and disable buffering of `ARRAY` contents.

The following methods are included with the `oracle.sql.ARRAY` class:

- `setAutoBuffering`
- `getAutoBuffering`

It is advisable to enable auto-buffering in a JDBC application when the `ARRAY` elements will be accessed more than once by the `getAttributes` and `getArray` methods, presuming the `ARRAY` data is able to fit into the Java Virtual Machine (JVM) memory without overflow.

> **Note:**
>
> Buffering the converted elements may cause the JDBC application to consume a significant amount of memory.

When you enable auto-buffering, the `oracle.sql.ARRAY` object keeps a local copy of all the converted elements. This data is retained so that a second access of this information does not require going through the data format conversion process.

18.3.3 ARRAY Automatic Indexing

If an array is in auto-indexing mode, then the array object maintains an index table to hasten array element access.

The `oracle.sql.ARRAY` class contains the following methods to support automatic array-indexing:

- `setAutoIndexing(boolean)`
- `setAutoIndexing(boolean, int)`

By default, auto-indexing is not enabled. For a JDBC application, enable auto-indexing for `ARRAY` objects if random access of array elements may occur through the `getArray` and `getResultSet` methods.

18.4 Creating and Using Arrays

This section discusses how to create array objects and how to retrieve and pass collections as array objects, including the following topics.

18.4.1 Creating ARRAY Objects

> **Note:**
>
> Oracle JDBC does not support the JDBC 4.0 method `createArrayOf` method of `java.sql.Connection` interface. This method only allows anonymous array types, while all Oracle array types are named. Use the Oracle specific method `oracle.jdbc.OracleConnection.createARRAY` instead.

This section describes how to create `ARRAY` objects. This section covers the following topics:

Steps in Creating ARRAY Objects

Starting from Oracle Database 11*g* Release 1, you can use the `createARRAY` factory method of `oracle.jdbc.OracleConnection` interface to create an array object. The factory method for creating arrays has been defined as follows:

```
public ARRAY createARRAY(java.lang.String typeName,java.lang.Object elements)throws
SQLException
```

where, `typeName` is the name of the SQL type of the created object and `elements` is the elements of the created object.

Perform the following to create an array:

1. Create a collection with the `CREATE TYPE` statement as follows:

   ```
   CREATE TYPE elements AS varray(22) OF NUMBER(5,2);
   ```

   The two possibilities for the contents of `elements` are:

   - An array of Java primitives. For example, `int[]`.
   - An array of Java objects, such as *`xxx`*`[]`, where *`xxx`* is the name of a Java class. For example, `Integer[]`.

   > **Note:**
   >
   > The `setARRAY`, `setArray`, and `setObject` methods of the `OraclePreparedStatement` class take an object of the type `oracle.sql.ARRAY` as an argument, not an array of objects.

2. Construct the `ARRAY` object by passing the Java string specifying the user-defined SQL type name of the array and a Java object containing the individual elements you want the array to contain.

```
ARRAY array = oracle.jdbc.OracleConnection.createARRAY(sql_type_name, elements);
```

> **Note:**
>
> The name of the collection type is not the same as the type name of the elements. For example:
>
> ```
> CREATE TYPE person AS object
>             (c1 NUMBER(5), c2 VARCHAR2(30));
> CREATE TYPE array_of_persons AS varray(10)
>           OF person;
> ```
>
> In the preceding statements, the name of the collection type is `ARRAY_OF_PERSON`. The SQL type name of the collection elements is `PERSON`.

Example 18-1 Creating Multilevel Collections

As with single-level collections, the JDBC application can create an `oracle.sql.ARRAY` instance to represent a multilevel collection, and then send the instance to the database. The same `createARRAY` factory method you use to create single-level collections, can be used to create multilevel collections as well. To create a single-level collection, the elements are a one dimensional Java array, while to create a multilevel collection, the elements can be either an array of `oracle.sql.ARRAY[]` elements or a nested Java array or the combinations.

The following code shows how to create collection types with a nested Java array:

```
// prepare the multilevel collection elements as a nested Java array
int[][][] elements = { {{1}, {1, 2}}, {{2}, {2, 3}}, {{3}, {3, 4}} };

// create the ARRAY using the factory method
ARRAY array = oracle.jdbc.OracleConnection.createARRAY(sql_type_name, elements);
```

18.4.2 Retrieving an Array and Its Elements

This section first discusses how to retrieve an `ARRAY` instance as a whole from a result set, and then how to retrieve the elements from the `ARRAY` instance. This section covers the following topics:

- About Retrieving the Array
- Data Retrieval Methods
- Comparing the Data Retrieval Methods
- Retrieving Elements of a Structured Object Array According to a Type Map
- Retrieving a Subset of Array Elements
- Retrieving Array Elements into an oracle.sql.Datum Array
- About Accessing Multilevel Collection Elements

18.4.2.1 About Retrieving the Array

You can retrieve a SQL array from a result set by casting the result set to `OracleResultSet` and using the `getARRAY` method, which returns an `oracle.sql.ARRAY` object. If you want to

avoid casting the result set, then you can get the data with the standard `getObject` method specified by the `java.sql.ResultSet` interface and cast the output to `oracle.sql.ARRAY`.

18.4.2.2 Data Retrieval Methods

Once you have an `ARRAY` object, you can retrieve the data using one of these three overloaded methods of the `oracle.sql.ARRAY` class:

- `getArray`
- `getOracleArray`
- `getResultSet`

Oracle also provides methods that enable you to retrieve all the elements of an array, or a subset.

> **Note:**
>
> In case you are working with an array of structured objects, Oracle provides versions of these three methods that enable you to specify a type map so that you can choose how to map the objects to Java.

getOracleArray

The `getOracleArray` method is an Oracle-specific extension that is not specified in the standard `Array` interface. The `getOracleArray` method retrieves the element values of the array into a `Datum[]` array. The elements are of the `oracle.sql.*` data type corresponding to the SQL type of the data in the original array.

For an array of structured objects, this method will use `oracle.jdbc.OracleStruct` instances for the elements.

Oracle also provides a `getOracleArray(`*`index`*`,`*`count`*`) method` to get a subset of the array elements.

getResultSet

The `getResultSet` method returns a result set that contains elements of the array designated by the `ARRAY` object. The result set contains one row for each array element, with two columns in each row. The first column stores the index into the array for that element, and the second column stores the element value. In the case of VARRAYs, the index represents the position of the element in the array. In the case of nested tables, which are by definition unordered, the index reflects only the return order of the elements in the particular query.

Oracle recommends using `getResultSet` when getting data from nested tables. Nested tables can have an unlimited number of elements. The `ResultSet` object returned by the method initially points at the first row of data. You get the contents of the nested table by using the `next` method and the appropriate `get`*`XXX`* method. In contrast, `getArray` returns the entire contents of the nested table at one time.

The `getResultSet` method uses the default type map of the connection to determine the mapping between the SQL type of the Oracle object and its corresponding Java data type. If you do not want to use the default type map of the connection, another version of the method, `getResultSet(`*`map`*`)`, enables you to specify an alternate type map.

Oracle also provides the `getResultSet(index,count)` and `getResultSet(index,count,map)` methods to retrieve a subset of the array elements.

getArray

The `getArray` method is a standard JDBC method that returns the array elements as a `java.lang.Object`, which you can cast as appropriate. The elements are converted to the Java types corresponding to the SQL type of the data in the original array.

Oracle also provides a `getArray(index,count)` method to retrieve a subset of the array elements.

18.4.2.3 Comparing the Data Retrieval Methods

If you use `getOracleArray` to return the array elements, then the use by that method of `oracle.sql.Datum` instances avoids the expense of data conversion from SQL to Java. The non-character data inside the instance of a `Datum` class or any of its subclass remains in raw SQL format.

If you use `getResultSet` to return an array of primitive data types, then the JDBC driver returns a `ResultSet` object that contains, for each element, the index into the array for the element and the element value. For example:

```
ResultSet rset = intArray.getResultSet();
```

In this case, the result set contains one row for each array element, with two columns in each row. The first column stores the index into the array and the second column stores the element value.

If the elements of an array are of a SQL type that maps to a Java type, then `getArray` returns an array of elements of this Java type. The return type of the `getArray` method is `java.lang.Object`. Therefore, the result must be cast before it can be used.

```
BigDecimal[] values = (BigDecimal[]) intArray.getArray();
```

Here `intArray` is an `oracle.sql.ARRAY`, corresponding to a VARRAY of type `NUMBER`. The `values` array contains an array of elements of type `java.math.BigDecimal`, because the SQL `NUMBER` data type maps to Java `BigDecimal`, by default, according to Oracle JDBC drivers.

> **Note:**
>
> Using `BigDecimal` is a resource-intensive operation in Java. Because Oracle JDBC maps numeric SQL data to `BigDecimal` by default, using `getArray` may impact performance, and is not recommended for numeric collections.

18.4.2.4 Retrieving Elements of a Structured Object Array According to a Type Map

By default, if you are working with an array whose elements are structured objects, and you use `getArray` or `getResultSet`, then the Oracle objects in the array will be mapped to their corresponding Java data types according to the default mapping. This is because these methods use the default type map of the connection to determine the mapping.

However, if you do not want default behavior, then you can use the `getArray(map)` or `getResultSet(map)` method to specify a type map that contains alternate mappings. If there

are entries in the type map corresponding to the Oracle objects in the array, then each object in the array is mapped to the corresponding Java type specified in the type map. For example:

```
Object[] object = (Object[])objArray.getArray(map);
```

Where `objArray` is an `oracle.sql.ARRAY` object and `map` is a `java.util.Map` object.

If the type map does not contain an entry for a particular Oracle object, then the element is returned as an `oracle.jdbc.OracleStruct` object.

The `getResultSet(map)` method behaves similarly to the `getArray(map)` method.

Related Topics

- Using a Type Map to Map Array Elements

18.4.2.5 Retrieving a Subset of Array Elements

If you do not want to retrieve the entire contents of an array, then you can use signatures of `getArray`, `getResultSet`, and `getOracleArray` that let you retrieve a subset. To retrieve a subset of the array, pass in an index and a count to indicate where in the array you want to start and how many elements you want to retrieve. As previously described, you can specify a type map or use the default type map for your connection to convert to Java types. For example:

```
Object object = arr.getArray(index, count, map);
Object object = arr.getArray(index, count);
```

Similar examples using `getResultSet` are:

```
ResultSet rset = arr.getResultSet(index, count, map);
ResultSet rset = arr.getResultSet(index, count);
```

A similar example using `getOracleArray` is:

```
Datum[] arr = arr.getOracleArray(index, count);
```

Where *arr* is an `oracle.sql.ARRAY` object, *index* is type `long`, *count* is type `int`, and *map* is a `java.util.Map` object.

> **Note:**
>
> There is no performance advantage in retrieving a subset of an array, as opposed to the entire array.

18.4.2.6 Retrieving Array Elements into an oracle.sql.Datum Array

Use `getOracleArray` to return an `oracle.sql.Datum[]` array. The elements of the returned array is of `oracle.sql.*` type that correspond to the SQL data type of the elements of the original array. For example:

```
Datum arraydata[] = arr.getOracleArray();
```

`arr` is an `oracle.sql.ARRAY` object.

The following example assumes that a connection object `conn` and a statement object `stmt` have already been created. In the example, an array with the SQL type name `NUM_ARRAY` is

created to store a VARRAY of `NUMBER` data. The `NUM_ARRAY` is in turn stored in a table `VARRAY_TABLE`.

A query selects the contents of the `VARRAY_TABLE`. The result set is cast to `OracleResultSet`. The `getARRAY` method is applied to it to retrieve the array data into `my_array`, which is an `oracle.sql.ARRAY` object.

Because `my_array` is of type `oracle.sql.ARRAY`, you can apply the methods `getSQLTypeName` and `getBaseType` to it to return the name of the SQL type of each element in the array and its integer code.

The program then prints the contents of the array. Because the contents of `NUM_ARRAY` are of the SQL data type `NUMBER`, the elements of `my_array` are of the type, `BigDecimal`. Before you can use the elements, they must first be cast to `BigDecimal`. In the `for` loop, the individual values of the array are cast to `BigDecimal` and printed to standard output.

```
stmt.execute ("CREATE TYPE num_varray AS VARRAY(10) OF NUMBER(12, 2)");
stmt.execute ("CREATE TABLE varray_table (col1 num_varray)");
stmt.execute ("INSERT INTO varray_table VALUES (num_varray(100, 200))");

ResultSet rs = stmt.executeQuery("SELECT * FROM varray_table");
ARRAY my_array = ((OracleResultSet)rs).getARRAY(1);

// return the SQL type names, integer codes,
// and lengths of the columns
System.out.println ("Array is of type " + array.getSQLTypeName());
System.out.println ("Array element is of type code " + array.getBaseType());
System.out.println ("Array is of length " + array.length());

// get Array elements
BigDecimal[] values = (BigDecimal[]) my_array.getArray();

for (int i=0; i<values.length; i++)
{
   BigDecimal out_value = (BigDecimal) values[i];
   System.out.println(">> index " + i + " = " + out_value.intValue());
}
```

Note that if you use `getResultSet` to obtain the array, then you must would first get the result set object, and then use the `next` method to iterate through it. Notice the use of the parameter indexes in the `getInt` method to retrieve the element index and the element value.

```
ResultSet rset = my_array.getResultSet();
while (rset.next())
{
    // The first column contains the element index and the
    // second column contains the element value
    System.out.println(">> index " + rset.getInt(1)+" = " + rset.getInt(2));
}
```

18.4.2.7 About Accessing Multilevel Collection Elements

The `oracle.sql.ARRAY` class provides three methods, which are overloaded, to access collection elements. The JDBC drivers extend these methods to support multilevel collections. These methods are:

- `getArray` method
- `getOracleArray` method
- `getResultSet` method

The `getArray` method returns a Java array that holds the collection elements. The array element type is determined by the collection element type and the JDBC default conversion matrix.

For example, the `getArray` method returns a `java.math.BigDecimal` array for collection of SQL `NUMBER`. The `getOracleArray` method returns a `Datum` array that holds the collection elements in `Datum` format. For multilevel collections, the `getArray` and `getOracleArray` methods both return a Java array of `oracle.sql.ARRAY` elements.

The `getResultSet` method returns a `ResultSet` object that wraps the multilevel collection elements. For multilevel collections, the JDBC applications use the `getObject`, `getARRAY`, or `getArray` method of the `ResultSet` class to access the collection elements as instances of `oracle.sql.ARRAY`.

The following code shows how to use the `getOracleArray`, `getArray`, and `getResultSet` methods:

```
Connection conn = ...;          // make a JDBC connection
Statement stmt = conn.createStatement ();
ResultSet rset = stmt.executeQuery ("select col2 from tab2 where idx=1");

while (rset.next())
{
  ARRAY varray3 = (ARRAY) rset.getObject (1);
  Object varrayElems = varray3.getArray (1);
// access array elements of "varray3"
  Datum[] varray3Elems = (Datum[]) varrayElems;

  for (int i=0; i<varray3Elems.length; i++)
  {
    ARRAY varray2 = (ARRAY) varray3Elems[i];
    Datum[] varray2Elems = varray2.getOracleArray();
    // access array elements of  "varray2"

    for (int j=0; j<varray2Elems.length; j++)
    {
      ARRAY varray1 = (ARRAY) varray2Elems[j];
      ResultSet varray1Elems = varray1.getResultSet();
      // access array elements   of "varray1"

      while (varray1Elems.next())
        System.out.println ("idx="+varray1Elems.getInt(1)+"
          value="+varray1Elems.getInt(2));
    }
  }
}
rset.close ();
stmt.close ();
conn.close ();
```

18.4.3 Passing Arrays to Statement Objects

This section discusses how to pass arrays to prepared statement objects or callable statement objects.

Passing an Array to a Prepared Statement

Pass an array to a prepared statement as follows.

> **Note:**
>
> you can use arrays as either `IN` or `OUT` bind variables.

1. Define the array that you want to pass to the prepared statement as an `oracle.sql.ARRAY` object.

   ```
   ARRAY array = oracle.jdbc.OracleConnection.createARRAY(sql_type_name, elements);
   ```

   `sql_type_name` is a Java string specifying the user-defined SQL type name of the array and *`elements`* is a `java.lang.Object` containing a Java array of the elements.
2. Create a `java.sql.PreparedStatement` object containing the SQL statement to be run.
3. Cast your prepared statement to `OraclePreparedStatement`, and use `setARRAY` to pass the array to the prepared statement.

   ```
   (OraclePreparedStatement)stmt.setARRAY(parameterIndex, array);
   ```

   `parameterIndex` is the parameter index and *`array`* is the `oracle.sql.ARRAY` object you constructed previously.
4. Run the prepared statement.

Passing an Array to a Callable Statement

To retrieve a collection as an `OUT` parameter in PL/SQL blocks, perform the following to register the bind type for your `OUT` parameter.

1. Cast your callable statement to `OracleCallableStatement`, as follows:

   ```
   OracleCallableStatement ocs = (OracleCallableStatement)conn.prepareCall("{? = call
   func()}");
   ```
2. Register the `OUT` parameter with the following form of the `registerOutParameter` method:

   ```
   ocs.registerOutParameter
           (int param_index, int sql_type, string sql_type_name);
   ```

   `param_index` is the parameter index, *`sql_type`* is the SQL type code, and *`sql_type_name`* is the name of the array type. In this case, the *`sql_type`* is `OracleTypes.ARRAY`.
3. Run the call, as follows:

   ```
   ocs.execute();
   ```
4. Get the value, as follows:

   ```
   oracle.sql.ARRAY array = ocs.getARRAY(1);
   ```

18.5 Using a Type Map to Map Array Elements

If your array contains Oracle objects, then you can use a type map to associate the objects in the array with the corresponding Java class. If you do not specify a type map, or if the type map does not contain an entry for a particular Oracle object, then each element is returned as an `oracle.jdbc.OracleStruct` object.

If you want the type map to determine the mapping between the Oracle objects in the array and their associated Java classes, then you must add an appropriate entry to the map.

The following example illustrates how you can use a type map to map the elements of an array to a custom Java object class. In this case, the array is a nested table. The example begins by defining an EMPLOYEE object that has a name attribute and employee number attribute. EMPLOYEE_LIST is a nested table type of EMPLOYEE objects. Then an EMPLOYEE_TABLE is created to store the names of departments within a corporation and the employees associated with each department. In the EMPLOYEE_TABLE, the employees are stored in the form of EMPLOYEE_LIST tables.

```
stmt.execute("CREATE TYPE EMPLOYEE AS OBJECT
            (EmpName VARCHAR2(50),EmpNo INTEGER))");

stmt.execute("CREATE TYPE EMPLOYEE_LIST AS TABLE OF EMPLOYEE");

stmt.execute("CREATE TABLE EMPLOYEE_TABLE (DeptName VARCHAR2(20),
     Employees EMPLOYEE_LIST) NESTED TABLE Employees STORE AS ntable1");

stmt.execute("INSERT INTO EMPLOYEE_TABLE VALUES ("SALES", EMPLOYEE_LIST
            (EMPLOYEE('Susan Smith', 123), EMPLOYEE('Lee Brown', 124)))");
```

If you want to retrieve all the employees belonging to the SALES department into an array of instances of the custom object class EmployeeObj, then you must add an entry to the type map to specify mapping between the EMPLOYEE SQL type and the EmployeeObj custom object class.

To do this, first create your statement and result set objects, then select the EMPLOYEE_LIST associated with the SALES department into the result set. Cast the result set to OracleResultSet so you can use the getARRAY method to retrieve the EMPLOYEE_LIST into an ARRAY object (employeeArray in the following example).

The EmployeeObj custom object class in this example implements the SQLData interface.

```
Statement s = conn.createStatement();
OracleResultSet rs = (OracleResultSet)s.executeQuery
       ("SELECT Employees FROM employee_table WHERE DeptName = 'SALES'");

// get the array object
ARRAY employeeArray = ((OracleResultSet)rs).getARRAY(1);
```

Now that you have the EMPLOYEE_LIST object, get the existing type map and add an entry that maps the EMPLOYEE SQL type to the EmployeeObj Java type.

```
// add type map entry to map SQL type
// "EMPLOYEE" to Java type "EmployeeObj"
Map map = conn.getTypeMap();
map.put("EMPLOYEE", Class.forName("EmployeeObj"));
```

Next, retrieve the SQL EMPLOYEE objects from the EMPLOYEE_LIST. To do this, call the getArray method of the employeeArray array object. This method returns an array of objects. The getArray method returns the EMPLOYEE objects into the employees object array.

```
// Retrieve array elements
Object[] employees = (Object[]) employeeArray.getArray();
```

Finally, create a loop to assign each of the EMPLOYEE SQL objects to the EmployeeObj Java object emp.

```
// Each array element is mapped to EmployeeObj object.
for (int i=0; i<employees.length; i++)
{
   EmployeeObj emp = (EmployeeObj) employees[i];
```

```
    . . .
}
```