15
Working with Oracle Object Types

This chapter describes the Java Database Connectivity (JDBC) support for user-defined object types. It discusses functionality of the generic, weakly typed `oracle.sql.STRUCT` class, as well as how to map to custom Java classes that implement either the JDBC standard `SQLData` interface or the Oracle-specific `OracleData` interface.

> **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.

The following topics are covered:

- About Mapping Oracle Objects
- About Using the Default STRUCT Class for Oracle Objects
- About Creating and Using Custom Object Classes for Oracle Objects
- Object-Type Inheritance
- About Describing an Object Type

Related Topics

- About Using PL/SQL Types

15.1 About Mapping Oracle Objects

Oracle object types provide support for composite data structures in the database. For example, you can define a `Person` type that has the attributes `name` of `CHAR` type, `phoneNumber` of `CHAR` type, and `employeeNumber` of `NUMBER` type.

Oracle provides tight integration between its Oracle object features and its JDBC functionality. You can use a standard, generic JDBC type to map to Oracle objects, or you can customize the mapping by creating custom Java type definition classes.

> **Note:**
>
> In this book, Java classes that you create to map to Oracle objects will be referred to as **custom Java classes** or, more specifically, **custom object classes**. This is as opposed to **custom references classes**, which are Java classes that map to object references, and **custom collection classes**, which are Java classes that map to Oracle collections.

Custom object classes can implement either a standard JDBC interface or an Oracle extension interface to read and write data. JDBC materializes Oracle objects as instances of particular Java classes. Two main steps in using JDBC to access Oracle objects are:

1. Creating the Java classes for the Oracle objects
2. Populating these classes. You have the following options:
   - Let JDBC materialize the object as a `STRUCT` object.
   - Explicitly specify the mappings between Oracle objects and Java classes.

     This includes customizing your Java classes for object data. The driver then must be able to populate instances of the custom object classes that you specify. This imposes a set of constraints on the Java classes. To satisfy these constraints, you can define your classes to implement either the JDBC standard `java.sql.SQLData` interface or the Oracle extension `oracle.jdbc.OracleData` interface.

> **Note:**
>
> When you use the `SQLData` interface, you must use a Java type map to specify your SQL-Java mapping, unless weakly typed `java.sql.Struct` objects will suffice.

15.2 About Using the Default STRUCT Class for Oracle Objects

This section covers the following topics:

- Overview of Using the Struct Class
- Retrieving STRUCT Objects and Attributes
- About Creating STRUCT Objects
- Binding STRUCT Objects into Statements
- STRUCT Automatic Attribute Buffering

15.2.1 Overview of Using the Struct Class

If you choose not to supply a custom Java class for your SQL-Java mapping for an Oracle object, then Oracle JDBC materializes the object as an object that implements the `java.sql.Struct` interface.

You would typically want to use `STRUCT` objects, instead of custom Java objects, in situations where you do not know the actual SQL type. For example, your Java application might be a tool to manipulate arbitrary object data within the database, as opposed to being an end-user

application. You can select data from the database into `STRUCT` objects and create `STRUCT` objects for inserting data into the database. `STRUCT` objects completely preserve data, because they maintain the data in SQL format. Using `STRUCT` objects is more efficient and more precise in situations where you do not need the information in an application specific form.

15.2.2 Retrieving STRUCT Objects and Attributes

This section discusses how to retrieve and manipulate Oracle objects and their attributes, using either Oracle-specific features or JDBC 2.0 standard features.

> **Note:**
>
> The JDBC driver seamlessly handles embedded objects, that is, `STRUCT` objects that are attributes of `STRUCT` objects, in the same way that it typically handles objects. When the JDBC driver retrieves an attribute that is an object, it follows the same rules of conversion by using the type map, if it is available, or by using default mapping.

Retrieving an Oracle Object as a java.sql.Struct Object

Alternatively, in the preceding example, you can use standard JDBC functionality, such as `getObject`, to retrieve an Oracle object from the database as an instance of `java.sql.Struct`. The `getObject` method returns a `java.lang.Object`, so, you must cast the output of the method to `Struct`. For example:

```
ResultSet rs= stmt.executeQuery("SELECT * FROM struct_table");
java.sql.Struct jdbcStruct = (java.sql.Struct)rs.getObject(1);
```

Retrieving Attributes as oracle.sql Types

If you want to retrieve Oracle object attributes from a `STRUCT` or `Struct` instance as `oracle.sql` types, then use the `getOracleAttributes` method of the `oracle.sql.STRUCT` class, as follows:

```
oracle.sql.Datum[] attrs = oracleSTRUCT.getOracleAttributes();
```

or:

```
oracle.sql.Datum[] attrs = ((oracle.sql.STRUCT)jdbcStruct).getOracleAttributes();
```

Retrieving Attributes as Standard Java Types

If you want to retrieve Oracle object attributes as standard Java types from a `STRUCT` or `Struct` instance, use the standard `getAttributes` method:

```
Object[] attrs = jdbcStruct.getAttributes();
```

> **Note:**
>
> 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 a structure type in the database, the cached descriptor for that structure type will become stale and your application will receive a `SQLException` exception.

15.2.3 About Creating STRUCT Objects

For information about creating STRUCT objects, refer to "Package oracle.sql".

> **Note:**
>
> If you have already fetched from the database a `STRUCT` of the appropriate SQL object type, then the easiest way to get a `STRUCT` descriptor is to call `getDescriptor` on one of the fetched `STRUCT` objects. Only one `STRUCT` descriptor is needed for any one SQL object type.

15.2.4 Binding STRUCT Objects into Statements

To bind an `oracle.sql.STRUCT` object to a prepared statement or callable statement, you can either use the standard `setObject` method (specifying the type code), or cast the statement object to an Oracle statement type and use the Oracle extension `setOracleObject` method. For example:

```
PreparedStatement ps= conn.prepareStatement("text_of_prepared_statement");
Struct mySTRUCT = conn.createStruct (...);
ps.setObject(1, mySTRUCT, Types.STRUCT);
```

or:

```
PreparedStatement ps= conn.prepareStatement("text_of_prepared_statement");
Struct mySTRUCT = conn.createStruct (...);
((OraclePreparedStatement)ps).setOracleObject(1, mySTRUCT);
```

15.2.5 STRUCT Automatic Attribute Buffering

Oracle JDBC driver furnishes public methods to enable and disable buffering of `STRUCT` attributes.

> **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.

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

- `public void setAutoBuffering(boolean enable)`
- `public boolean getAutoBuffering()`

The `setAutoBuffering(boolean)` method enables or disables auto-buffering. The `getAutoBuffering` method returns the current auto-buffering mode. By default, auto-buffering is disabled.

It is advisable to enable auto-buffering in a JDBC application when the `STRUCT` attributes are 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 attributes may cause the JDBC application to consume a significant amount of memory.

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

Related Topics

- ARRAY Automatic Element Buffering

15.3 About Creating and Using Custom Object Classes for Oracle Objects

This section covers the following topics:

- Overview of Creating and Using Custom Object Classes
- Relative Advantages of OracleData versus SQLData
- About Type Maps for SQLData Implementations
- About Creating Type Map and Defining Mappings for a SQLData Implementation
- About Reading and Writing Data with a SQLData Implementation
- About the OracleData Interface
- About Reading and Writing Data with an OracleData Implementation
- Additional Uses of OracleData

15.3.1 Overview of Creating and Using Custom Object Classes

If you want to create custom object classes for your Oracle objects, then you must define entries in the type map that specify the custom object classes that the drivers instantiate for the corresponding Oracle objects.

You must also provide a way to create and populate instances of the custom object class from the Oracle object and its attribute data. The driver must be able to read from a custom object class and write to it. In addition, the custom object class can provide `get`*`XXX`* and `set`*`XXX`* methods corresponding to the attributes of the Oracle object, although this is not necessary. To create and populate the custom classes and provide these read/write capabilities, you can choose between the following interfaces:

- The JDBC standard `SQLData` interface

- The `OracleData` and `OracleDataFactory` interfaces provided by Oracle

The custom object class you create must implement one of these interfaces. The `OracleData` interface can also be used to implement the custom reference class corresponding to the custom object class. However, if you are using the `SQLData` interface, then you can use only weak reference types in Java, such as `java.sql.Ref` or `oracle.sql.REF`. The `SQLData` interface is for mapping SQL objects only.

As an example, assume you have an Oracle object type, `EMPLOYEE`, in the database that consists of two attributes: `Name`, which is of the `CHAR` type and `EmpNum`, which is of the `NUMBER` type. You use the type map to specify that the `EMPLOYEE` object should map to a custom object class that you call `JEmployee`. You can implement either the `SQLData` or `OracleData` interface in the `JEmployee` class.

Related Topics

- Object-Type Inheritance

15.3.2 Relative Advantages of OracleData versus SQLData

In deciding which of the two interface implementations to use, you need to consider the advantages of `OracleData` and `SQLData`.

The `SQLData` interface is for mapping SQL objects only. The `OracleData` interface is more flexible, enabling you to map SQL objects as well as any other SQL type for which you want to customize processing. You can create an `OracleData` implementation from any data type found in Oracle Database. This could be useful, for example, for serializing `RAW` data in Java.

Advantages of the OracleData Interface

The advantages of the `OracleData` interface are:

- It does not require an entry in the type map for the Oracle object.
- It has awareness of Oracle extensions.
- You can construct an `OracleData` from an `oracle.sql.STRUCT`. This is more efficient because it avoids unnecessary conversions to native Java types.
- You can obtain the corresponding JDBC object from `OracleData`, using the `toJDBCObject` method.

Advantages of SQLData

`SQLData` is a JDBC standard that makes your code portable.

15.3.3 About Type Maps for SQLData Implementations

If you use the `SQLData` interface in a custom object class, then you must create type map entries that specify the custom object class to use in mapping the Oracle object type to Java. You can either use the default type map of the connection object or a type map that you specify when you retrieve the data from the result set. The `getObject` method of the `ResultSet` interface has a signature that lets you specify a type map. You can use either of the following:

```
rs.getObject(int columnIndex);

rs.getObject(int columnIndex, Map map);
```

When using a SQLData implementation, if you do not include a type map entry, then the object maps to the `oracle.jdbc.OracleStruct` interface by default. `OracleData` implementations, by contrast, have their own mapping functionality so that a type map entry is not required. When using an `OracleData` implementation, use the Oracle `getObject(int columnindex, OracleDataFactory factory)` method.

The type map relates a Java class to the SQL type name of an Oracle object. This one-to-one mapping is stored in a hash table as a keyword-value pair. When you read data from an Oracle object, the JDBC driver considers the type map to determine which Java class to use to materialize the data from the Oracle object type. When you write data to an Oracle object, the JDBC driver gets the SQL type name from the Java class by calling the `getSQLTypeName` method of the `SQLData` interface. The actual conversion between SQL and Java is performed by the driver.

The attributes of the Java class that corresponds to an Oracle object can use either Java native types or Oracle native types to store attributes.

Related Topics

- About Creating and Using Custom Object Classes for Oracle Objects

15.3.4 About Creating Type Map and Defining Mappings for a SQLData Implementation

This section covers the following topics:

- Overview of Creating a Type Map and Defining Mappings
- Adding Entries to an Existing Type Map
- Creating a New Type Map
- About Materializing Object Types not Specified in the Type Map

15.3.4.1 Overview of Creating a Type Map and Defining Mappings

When using a `SQLData` implementation, the JDBC applications programmer is responsible for providing a type map, which must be an instance of a class that implements the standard `java.util.Map` interface.

You have the option of creating your own class to accomplish this, but the standard `java.util.Hashtable` class meets the requirement.

`Hashtable` and other classes used for type maps implement a `put` method that takes keyword-value pairs as input, where each key is a fully qualified SQL type name and the corresponding value is an instance of a specified Java class.

A type map is associated with a connection instance. The standard `java.sql.Connection` interface and the Oracle-specific `oracle.jdbc.OracleConnection` interface include a `getTypeMap` method. Both return a `Map` object.

15.3.4.2 Adding Entries to an Existing Type Map

When a connection instance is first established, the default type map is empty. You must populate it.

Perform the following general steps to add entries to an existing type map:

1. Use the `getTypeMap` method of your `OracleConnection` object to return the type map object of the connection. The `getTypeMap` method returns a `java.util.Map` object. For example, presuming an `OracleConnection` instance `oraconn`:

```
java.util.Map myMap = oraconn.getTypeMap();
```

> **Note:**
>
> If the type map in the `OracleConnection` instance has not been initialized, then the first call to `getTypeMap` returns an empty map.

2. Use the `put` method of the type map to add map entries. The `put` method takes two arguments: a SQL type name string and an instance of a specified Java class that you want to map to.

```
myMap.put(sqlTypeName, classObject);
```

The *`sqlTypeName`* is a string that represents the fully qualified name of the SQL type in the database. The *`classObject`* is the Java class object to which you want to map the SQL type. Get the class object with the `Class.forName` method, as follows:

```
myMap.put(sqlTypeName, Class.forName(className));
```

For example, if you have a `PERSON` SQL data type defined in the `CORPORATE` database schema, then map it to a `Person` Java class defined as `Person` with this statement:

```
myMap.put("CORPORATE.PERSON", Class.forName("Person"));
oraconn.setTypeMap(newMap);
```

The map has an entry that maps the `PERSON` SQL data type in the `CORPORATE` database to the `Person` Java class.

> **Note:**
>
> SQL type names in the type map must be all uppercase, because that is how Oracle Database stores SQL names.

15.3.4.3 Creating a New Type Map

Perform the following general steps to create a new type map. This example uses an instance of `java.util.Hashtable`, which extends `java.util.Dictionary` and implements `java.util.Map`.

1. Create a new type map object.

```
Hashtable newMap = new Hashtable();
```

2. Use the `put` method of the type map object to add entries to the map. For example, if you have an `EMPLOYEE` SQL type defined in the `CORPORATE` database, then you can map it to an `Employee` class object defined by `Employee.java`, as follows:

```
newMap.put("CORPORATE.EMPLOYEE", class.forName("Employee"));
```

3. When you finish adding entries to the map, you must use the `setTypeMap` method of the `OracleConnection` object to overwrite the existing type map of the connection. For example:

```
oraconn.setTypeMap(newMap);
```

In this example, the `setTypeMap` method overwrites the original map of the `oraconn` connection object with `newMap`.

> **Note:**
>
> The default type map of a connection instance is used when mapping is required but no map name is specified, such as for a result set `getObject` call that does not specify the map as input.

15.3.4.4 About Materializing Object Types not Specified in the Type Map

If you do not provide a type map with an appropriate entry when using a `getObject` call, then the JDBC driver will materialize an Oracle object as an instance of the `oracle.jdbc.OracleStruct` interface. If the Oracle object type contains embedded objects and they are not present in the type map, then the driver will materialize the embedded objects as instances of `oracle.jdbc.OracleStruct` as well. If the embedded objects are present in the type map, then a call to the `getAttributes` method will return embedded objects as instances of the specified Java classes from the type map.

15.3.5 About Reading and Writing Data with a SQLData Implementation

This section describes how to read data from an Oracle object or write data to an Oracle object if your corresponding Java class implements `SQLData`.

Reading SQLData Objects from a Result Set

The following text summarizes the steps to read data from an Oracle object into your Java application when you choose the `SQLData` implementation for your custom object class.

These steps assume you have already defined the Oracle object type, created the corresponding custom object class, updated the type map to define the mapping between the Oracle object and the Java class, and defined a statement object `stmt`.

1. Query the database to read the Oracle object into a JDBC result set.

```
ResultSet rs = stmt.executeQuery("SELECT emp_col FROM personnel");
```

The `PERSONNEL` table contains one column, `EMP_COL`, of SQL type `EMP_OBJECT`. This SQL type is defined in the type map to map to the Java class `Employee`.

2. Use the `getObject` method of Oracle result set to populate an instance of your custom object class with data from one row of the result set. The `getObject` method returns the user-defined `SQLData` object because the type map contains an entry for `Employee`.

```
if (rs.next())
   Employee emp = (Employee)rs.getObject(1);
```

Note that if the type map did not have an entry for the object, then the `getObject` method will return an `oracle.jdbc.OracleStruct` object. Cast the output to type `OracleStruct` because the `getObject` method signature returns the generic `java.lang.Object` type.

```
if (rs.next())
   OracleStruct empstruct = (OracleStruct)rs.getObject(1);
```

The `getObject` method calls `readSQL`, which, in turn, calls `read`*`XXX`* from the `SQLData` interface.

> **Note:**
>
> If you want to avoid using the defined type map, then use the `getSTRUCT` method. This method always returns a `STRUCT` object, even if there is a mapping entry in the type map.

3. If you have `get` methods in your custom object class, then use them to read data from your object attributes. For example, if `EMPLOYEE` has the attributes `EmpName` of type `CHAR` and `EmpNum` of type `NUMBER`, then provide a `getEmpName` method that returns a Java `String` and a `getEmpNum` method that returns an `int` value. Then call them in your Java application, as follows:

```
String empname = emp.getEmpName();
int empnumber = emp.getEmpNum();
```

Retrieving SQLData Objects from a Callable Statement OUT Parameter

Consider you have a `CallableStatement` instance, `cs`, that calls a PL/SQL function `GETEMPLOYEE`. The program passes an employee number to the function. The function returns the corresponding `Employee` object. To retrieve this object you do the following:

1. Prepare a `CallableStatement` to call the `GETEMPLOYEE` function, as follows:

```
CallableStatement ocs = conn.prepareCall("{ ? = call GETEMPLOYEE(?) }");
```

2. Declare the `empnumber` as the input parameter to `GETEMPLOYEE`. Register the `SQLData` object as the `OUT` parameter, with the type code `OracleTypes.STRUCT`. Then, run the statement. This can be done as follows:

```
cs.setInt(2, empnumber);
cs.registerOutParameter(1, OracleTypes.STRUCT, "EMP_OBJECT");
cs.execute();
```

3. Use the `getObject` method to retrieve the employee object.

```
Employee emp = (Employee)cs.getObject(1);
```

If there is no type map entry, then the `getObject` method will return a `java.sql.Struct` object.

```
Struct emp = cs.getObject(1);
```

Passing SQLData Objects to a Callable Statement as an IN Parameter

Suppose you have a PL/SQL function `addEmployee(?)` that takes an `Employee` object as an `IN` parameter and adds it to the `PERSONNEL` table. In this example, `emp` is a valid `Employee` object.

1. Prepare an `CallableStatement` to call the `addEmployee(?)` function.

```
CallableStatement cs =
  conn.prepareCall("{ call addEmployee(?) }");
```

2. Use `setObject` to pass the `emp` object as an `IN` parameter to the callable statement. Then, call the statement.

```
cs.setObject(1, emp);
cs.execute();
```

Writing Data to an Oracle Object Using a SQLData Implementation

The following text describes the steps in writing data to an Oracle object from your Java application when you choose the `SQLData` implementation for your custom object class.

This description assumes you have already defined the Oracle object type, created the corresponding Java class, and updated the type map to define the mapping between the Oracle object and the Java class.

1. If you have `set` methods in your custom object class, then use them to write data from Java variables in your application to attributes of your Java data type object.

```
emp.setEmpName(empname);
emp.setEmpNum(empnumber);
```

2. Prepare a statement that updates an Oracle object in a row of a database table, as appropriate, using the data provided in your Java data type object.

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

3. Use the `setObject` method of the prepared statement to bind your Java data type object to the prepared statement.

```
pstmt.setObject(1, emp);
```

4. Run the statement, which updates the database.

```
pstmt.executeUpdate();
```

15.3.6 About the OracleData Interface

You can create a custom object class that implements the `oracle.jdbc.OracleData` and the `oracle.jdbc.OracleDataFactory` interfaces to make an Oracle object and its attribute data available to Java applications. The `OracleData` and `OracleDataFactory` interfaces are Oracle-specific and are not a part of the JDBC standard.

> **Note:**
>
> Starting from Oracle Database 12*c* Release 1 (12.1), the `OracleData` and the `OracleDataFactory` interfaces replace the `ORAData` and the `ORADataFactory` interfaces.

Understanding the OracleData Interface Features

The `OracleData` interface has the following advantages:

- It supports Oracle extensions to the standard JDBC types.
- It does not require a type map to specify the names of the Java custom classes you want to create.

- It provides better performance. `OracleData` works directly with `Datum` types, the internal format the driver uses to hold Oracle objects.

The `OracleData` and the `OracleDataFactory` interfaces perform the following:

- The `toJDBCObject` method of the `OracleData` class transforms the data into an `oracle.jdbc.*` representation.
- `OracleDataFactory` specifies a `create` method equivalent to a constructor for the custom object class. It creates and returns an `OracleData` instance. The JDBC driver uses the `create` method to return an instance of the custom object class to your Java application. It takes as input a `java.lang.Object` object and an integer indicating the corresponding SQL type code as specified in the `OracleTypes` class.

`OracleData` and `OracleDataFactory` have the following definitions:

```
package oracle.jdbc;
import java.sql.Connection;
import java.sql.SQLException;
public interface OracleData
{
   public Object toJDBCObject(Connection conn) throws SQLException;
}

package oracle.jdbc;
import java.sql.SQLException;
public interface OracleDataFactory
{
    public OracleData create(Object jdbcValue, int sqlType) throws SQLException;

}
```

Where *`conn`* represents the Connection object, *`jdbcValue`* represents an object of type `java.lang.object` that is to be used to initialize the Object being created, and *`sqlType`* represents the SQL type of the specified `Datum` object.

Retrieving and Inserting Object Data

The JDBC drivers provide the following methods to retrieve and insert object data as instances of `OracleData`.

You can retrieve the object data in one of the following ways:

- Use the following `getObject` method of the Oracle-specific `OracleResultSet` interface:

  ```
  ors.getObject(int col_index, OracleDataFactory factory
  );
  ```

  This method takes as input the column index of the data in your result set and an `OracleDataFactory` instance. For example, you can implement a `getOracleDataFactory` method in your custom object class to produce the `OracleDataFactory` instance to input to the `getObject` method. The type map is not required when using Java classes that implement `OracleData`.

- Use the standard `getObject(`*`index`*`,` *`map`*`)` method specified by the `ResultSet` interface to retrieve data as instances of `OracleData`. In this case, you must have an entry in the type map that identifies the factory class to be used for the given object type and its corresponding SQL type name.

You can insert object data in one of the following ways:

- Use the following `setObject` method of the Oracle-specific `OraclePreparedStatement` class:

  ```
  setObject(int bind_index, Object custom_object);
  ```

  This method takes as input the parameter index of the bind variable and an instance of `OracleData` as the name of the object containing the variable.

- Use the standard `setObject` method specified by the `PreparedStatement` interface. You can also use this method, in its different forms, to insert `OracleData` instances without requiring a type map.

The following sections describe the `getObject` and `setObject` methods.

To continue the example of an Oracle object `EMPLOYEE`, you might have something like the following in your Java application:

```
OracleData obj = ors.getObject(1, Employee.getOracleDataFactory());
```

In this example, `ors` is an instance of the `OracleResultSet` interface, `getObject` is a method in the `OracleResultSet` interface used to retrieve an `OracleData` object, and the `EMPLOYEE` is in column 1 of the result set. The `static Employee.getOracleDataFactory` method will return an `OracleDataFactory` to the JDBC driver. The JDBC driver will call `create()` from this object, returning to your Java application an instance of the `Employee` class populated with data from the result set.

> **Note:**
>
> - `OracleData` and `OracleDataFactory` are defined as separate interfaces so that different Java classes can implement them if you wish.
> - To use the `OracleData` interface, your custom object classes must import `oracle.jdbc.*`.

15.3.7 About Reading and Writing Data with an OracleData Implementation

This section describes how to read data from an Oracle object or write data to an Oracle object if your corresponding Java class implements `OracleData`.

Reading Data from an Oracle Object Using an OracleData Implementation

The following text summarizes the steps in reading data from an Oracle object into your Java application. These steps apply whether you implement `OracleData` manually or use Oracle JVM Web Service Call-Out utility to produce your custom object classes.

These steps assume you have already defined the Oracle object type, created the corresponding custom object class or had Oracle JVM Web Service Call-Out utility create it for you, and defined a statement object `stmt`.

1. Query the database to read the Oracle object into a result set, casting it to an Oracle result set.

   ```
   OracleResultSet ors = (OracleResultSet)stmt.executeQuery
                         ("SELECT Emp_col FROM PERSONNEL");
   ```

Where `PERSONNEL` is a one-column table. The column name is `Emp_col` of type `Employee_object`.

2. Use the `getObject` method of Oracle result set to populate an instance of your custom object class with data from one row of the result set. The `getObject` method returns a `java.lang.Object` object, which you can cast to your specific custom object class.

```
if (ors.next())
   Employee emp = (Employee)ors.getObject(1, Employee.getOracleDataFactory());
```

or:

```
if (ors.next())
   Object obj = ors.getObject(1, Employee.getOracleDataFactory());
```

This example assumes that `Employee` is the name of your custom object class and `ors` is the name of your `OracleResultSet` instance.

For example, if the SQL type name for your object is `EMPLOYEE`, then the corresponding Java class is `Employee`, which will implement `OracleData`. The corresponding Factory class is `EmployeeFactory`, which will implement `OracleDataFactory`.

Use this statement to declare the `EmployeeFactory` entry for your type map:

```
map.put ("EMPLOYEE", Class.forName ("EmployeeFactory"));
```

Then use the form of `getObject` where you specify the map object:

```
Employee emp = (Employee) rs.getObject (1, map);
```

If the default type map of the connection already has an entry that identifies the factory class to be used for the given object type and its corresponding SQL type name, then you can use this form of `getObject`:

```
Employee emp = (Employee) rs.getObject (1);
```

3. If you have `get` methods in your custom object class, then use them to read data from your object attributes into Java variables in your application. For example, if `EMPLOYEE` has `EmpName` of type `CHAR` and `EmpNum` of type `NUMBER`, provide a `getEmpName` method that returns a Java `String` and a `getEmpNum` method that returns an integer. Then call them in your Java application as follows:

```
String empname = emp.getEmpName();
int empnumber = emp.getEmpNum();
```

Writing Data to an Oracle Object Using an OracleData Implementation

The following text summarizes the steps in writing data to an Oracle object from your Java application. These steps apply whether you implement `OracleData` manually or use Oracle JVM Web Service Call-Out utility to produce your custom object classes.

These steps assume you have already defined the Oracle object type and created the corresponding custom object class.

> **Note:**
>
> The type map is not used when you are performing database `INSERT` and `UPDATE` operations.

1. If you have `set` methods in your custom object class, then use them to write data from Java variables in your application to attributes of your Java data type object.

   ```
   emp.setEmpName(empname);
   emp.setEmpNum(empnumber);
   ```

2. Write an Oracle prepared statement that updates an Oracle object in a row of a database table, as appropriate, using the data provided in your Java data type object.

   ```
   OraclePreparedStatement opstmt = conn.prepareStatement
      ("UPDATE PERSONNEL SET Employee = ? WHERE Employee.EmpNum = 28959);
   ```

   This assumes `conn` is your `Connection` object.

3. Use the `setObject` method of the `OraclePreparedStatement` interface to bind your Java data type object to the prepared statement.

   ```
   opstmt.setObject(1,emp);
   ```

   The `setObject` method calls the `toJDBCObject` method of the custom object class instance to retrieve an `oracle.jdbc.OracleStruct` object that can be written to the database.

> **Note:**
>
> You can use your Java data type objects as either `IN` or `OUT` bind variables.

15.3.8 Additional Uses of OracleData

The `OracleData` interface offers far more flexibility than the `SQLData` interface. The `SQLData` interface is designed to let you customize the mapping of only Oracle object types to Java types of your choice. Implementing the `SQLData` interface lets the JDBC driver populate fields of a custom Java class instance from the original SQL object data, and the reverse, after performing the appropriate conversions between Java and SQL types.

The `OracleData` interface goes beyond supporting the customization of Oracle object types to Java types. It lets you provide a mapping between Java object types and *any* SQL type supported by the `oracle.sql` package.

You may find it useful to provide custom Java classes to wrap `oracle.sql.*` types and then implement customized conversions or functionality as well. The following are some possible scenarios:

- Performing encryption and decryption or validation of data
- Performing logging of values that have been read or are being written
- Parsing character columns, such as character fields containing URL information, into smaller components
- Mapping character strings into numeric constants
- Making data into more desirable Java formats, such as mapping a `DATE` field to `java.util.Date` format
- Customizing data representation, for example, data in a table column is in feet but you want it represented in meters after it is selected
- Serializing and deserializing Java objects

For example, use `OracleData` to store instances of Java objects that do not correspond to a particular SQL object type in the database in columns of SQL type `RAW`. The `create` method in `OracleDataFactory` would have to implement a conversion from an object of type `oracle.sql.RAW` to the desired Java object. The `toJDBCObject` method in `OracleData` would have to implement a conversion from the Java object to an `oracle.sql.RAW` object. You can also achieve this using Java serialization.

Upon retrieval, the JDBC driver transparently retrieves the raw bytes of data in the form of an `oracle.sql.RAW` and calls the `create` method of `OracleDataFactory` to convert the `oracle.sql.RAW` object to the desired Java class.

When you insert the Java object into the database, you can simply bind it to a column of type `RAW` to store it. The driver transparently calls the `OracleData.toJDBCObject` method to convert the Java object to an `oracle.sql.RAW` object. This object is then stored in a column of type `RAW` in the database.

Support for the `OracleData` interfaces is also highly efficient because the conversions are designed to work using `oracle.sql.*` formats, which happen to be the internal formats used by the JDBC drivers. Moreover, the type map, which is necessary for the `SQLData` interface, is not required when using Java classes that implement `OracleData`.

Related Topics

- About the OracleData Interface

15.4 Object-Type Inheritance

Object-type inheritance allows a new object type to be created by extending another object type. The new object type is then a subtype of the object type from which it extends. The subtype automatically inherits all the attributes and methods defined in the supertype. The subtype can add attributes and methods and overload or override methods inherited from the supertype.

Object-type inheritance introduces **substitutability**. Substitutability is the ability of a slot declared to hold a value of type `T` in addition to any subtype of type `T`. Oracle JDBC drivers handle substitutability transparently.

A database object is returned with its most specific type without losing information. For example, if the `STUDENT_T` object is stored in a `PERSON_T` slot, Oracle JDBC driver returns a Java object that represents the `STUDENT_T` object.

This section covers the following topics:

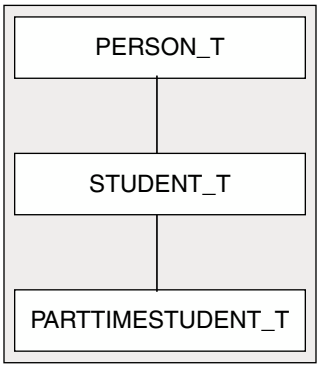
- About Creating Subtypes
- About Implementing Customized Classes for Subtypes
- About Retrieving Subtype Objects
- Creating Subtype Objects
- Sending Subtype Objects
- Accessing Subtype Data Fields
- Inheritance Metadata Methods

15.4.1 About Creating Subtypes

Create custom object classes if you want to have Java classes that explicitly correspond to the Oracle object types. If you have a hierarchy of object types, you may want a corresponding hierarchy of Java classes.

The most common way to create a database subtype in JDBC is to run a SQL `CREATE TYPE` command using the `execute` method of the `java.sql.Statement` interface. For example, you want to create a type inheritance hierarchy as depicted in the following figure:

Figure 15-1 Type Inheritance Hierarchy



The JDBC code for this can be as follows:

```
Statement s = conn.createStatement();
s.execute ("CREATE TYPE Person_T (SSN NUMBER, name VARCHAR2(30),
  address VARCHAR2(255))");
s.execute ("CREATE TYPE Student_T UNDER Person_t (deptid NUMBER,
  major VARCHAR2(100))");
s.execute ("CREATE TYPE PartTimeStudent_t UNDER Student_t (numHours NUMBER)");
```

In the following code, the `foo` member procedure in type `ST` is overloaded and the member procedure `print` overwrites the copy it inherits from type `T`.

```
CREATE TYPE T AS OBJECT (...,
  MEMBER PROCEDURE foo(x NUMBER),
  MEMBER PROCEDURE Print(),
  ...
  NOT FINAL;

CREATE TYPE ST UNDER T (...,
  MEMBER PROCEDURE foo(x DATE),         <-- overload "foo"
  OVERRIDING MEMBER PROCEDURE Print(),   <-- override "print"
  STATIC FUNCTION bar(...) ...
  ...
  );
```

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

> **See Also:**
>
> *Oracle Database Object-Relational Developer's Guide*

15.4.2 About Implementing Customized Classes for Subtypes

In most cases, a customized Java class represents a database object type. When you create a customized Java class for a subtype, the Java class can either mirror the database object type hierarchy or not.

You can use either the `OracleData` or `SQLData` solution in creating classes to map to the hierarchy of object types.

This section covers the following topics:

- About Using OracleData for Type Inheritance Hierarchy
- About UsingSQLData for Type Inheritance Hierarchy

15.4.2.1 About Using OracleData for Type Inheritance Hierarchy

Oracle recommends customized mappings, where Java classes implement the `oracle.sql.OracleData` interface. `OracleData` mapping requires the JDBC application to implement the `OracleData` and `OracleDataFactory` interfaces. The class implementing the `OracleDataFactory` interface contains a factory method that produces objects. Each object represents a database object.

The hierarchy of the class implementing the `OracleData` interface can mirror the database object type hierarchy. For example, the Java classes mapping to `PERSON_T` and `STUDENT_T` are as follows:

Person.java using OracleData

Code for the `Person.java` class which implements the `OracleData` and `OracleDataFactory` interfaces:

```
public static OracleDataFactory getOracleDataFactory()
  {
    return _personFactory;
  }

  public Person () {}

  public Person(NUMBER ssn, CHAR name, CHAR address)
  {
    this.ssn = ssn;
    this.name = name;
    this.address = address;
  }

  public Object toJDBCObject(OracleConnection c) throws SQLException
  {
    Object [] attributes = { ssn, name, address };
 Struct struct = c.createStruct("HR.PERSON_T", attributes);
    return struct;
  }
```

```
  public OracleData create(Object jdbcValue, int sqlType) throws SQLException
  {
    if (d == null) return null;
    Object [] attributes = ((STRUCT) d).getOracleAttributes();
    return new Person((NUMBER) attributes[0],
                      (CHAR) attributes[1],
                      (CHAR) attributes[2]);
  }
}
```

Student.java extending Person.java

Code for the `Student.java` class, which extends the `Person.java` class:

```
class Student extends Person
{
  static final Student _studentFactory = new Student ();

  public NUMBER deptid;
  public CHAR major;

  public static OracleDataFactory getOracleDataFactory()
  {
    return _studentFactory;
  }

  public Student () {}

  public Student (NUMBER ssn, CHAR name, CHAR address,
                  NUMBER deptid, CHAR major)
  {
    super (ssn, name, address);
    this.deptid = deptid;
    this.major = major;
  }

  public Object toJDBCObject(OracleConnection c) throws SQLException
  {
    Object [] attributes = { ssn, name, address, deptid, major };
    Struct struct = c.createStruct("HR.STUDENT_T", attributes);
    return struct;
  }

  public OracleData create(Object jdbcValue, int sqlType) throws SQLException
  {
    if (d == null) return null;
    Object [] attributes = ((STRUCT) d).getOracleAttributes();
    return new Student((NUMBER) attributes[0],
                       (CHAR) attributes[1],
                       (CHAR) attributes[2],
                       (NUMBER) attributes[3],
                       (CHAR) attributes[4]);
  }
}
```

Customized classes that implement the `OracleData` interface do not have to mirror the database object type hierarchy. For example, you could have declared the `Student` class without a superclass. In this case, `Student` would contain fields to hold the inherited attributes from `PERSON_T` as well as the attributes declared by `STUDENT_T`.

OracleDataFactory Implementation

The JDBC application uses the factory class in querying the database to return instances of `Person` or its subclasses, as in the following example:

```
ResultSet rset = stmt.executeQuery ("select person from tab1");
while (rset.next())
{
  rset.getOracleData(1,Person.getOracleDataFactory());
  ...
}
```

A class implementing the `OracleDataFactory` interface should be able to produce instances of the associated custom object type, as well as instances of any subtype, or at least all the types you expect to support.

In the following example, the `PersonFactory.getOracleDataFactory` method returns a factory that can handle `PERSON_T`, `STUDENT_T`, and `PARTTIMESTUDENT_T` objects, by returning `person`, `student`, or `parttimestudent` Java instances.

```
class PersonFactory implements OracleDataFactory
{
  static final PersonFactory _factory = new PersonFactory ();

  public static OracleDataFactory getOracleDataFactory()
  {
    return _factory;
  }

  public OracleData create(Object jdbcValue, int sqlType) throws SQLException
  {
    STRUCT s = (STRUCT) jdbcValue;
    if (s.getSQLTypeName ().equals ("HR.PERSON_T"))
      return Person.getOracleDataFactory ().create (jdbcValue, sqlType);
    else if (s.getSQLTypeName ().equals ("HR.STUDENT_T"))
      return Student.getOracleDataFactory ().create(jdbcValue, sqlType);
    else if (s.getSQLTypeName ().equals ("HR.PARTTIMESTUDENT_T"))
      return ParttimeStudent.getOracleDataFactory ().create(jdbcValue, sqlType);
    else
      return null;
  }
}
```

The following example assumes a table `tabl1`, such as the following:

```
CREATE TABLE tabl1 (idx NUMBER, person PERSON_T);
INSERT INTO tabl1 VALUES (1, PERSON_T (1000, 'HR', '100 Oracle Parkway'));
INSERT INTO tabl1 VALUES (2, STUDENT_T (1001, 'Peter', '200 Oracle Parkway', 101,
'CS'));
INSERT INTO tabl1 VALUES (3, PARTTIMESTUDENT_T (1002, 'David', '300 Oracle Parkway',
102, 'EE'));
```

15.4.2.2 About UsingSQLData for Type Inheritance Hierarchy

The customized classes that implement the `java.sql.SQLData` interface can mirror the database object type hierarchy. The `readSQL` and `writeSQL` methods of a subclass typically call the corresponding superclass methods to read or write the superclass attributes before reading or writing the subclass attributes. For example, the Java classes mapping to `PERSON_T` and `STUDENT_T` are as follows:

Person.java using SQLData

Code for the `Person.java` class, which implements the `SQLData` interface:

```
import java.sql.*;

public class Person implements SQLData
{
  private String sql_type;
  public int ssn;
  public String name;
  public String address;

  public Person () {}

  public String getSQLTypeName() throws SQLException { return sql_type; }

  public void readSQL(SQLInput stream, String typeName) throws SQLException
  {
    sql_type = typeName;
    ssn = stream.readInt();
    name = stream.readString();
    address = stream.readString();
  }

  public void writeSQL(SQLOutput stream) throws SQLException
  {
    stream.writeInt (ssn);
    stream.writeString (name);
    stream.writeString (address);
  }
}
```

Student.java extending Student.java

Code for the `Student.java` class, which extends the `Person.java` class:

```
import java.sql.*;

public class Student extends Person
{
  private String sql_type;
  public int deptid;
  public String major;

  public Student () { super(); }

  public String getSQLTypeName() throws SQLException { return sql_type; }

  public void readSQL(SQLInput stream, String typeName) throws SQLException
  {
    super.readSQL (stream, typeName);    // read supertype attributes
    sql_type = typeName;
    deptid = stream.readInt();
    major = stream.readString();
  }

  public void writeSQL(SQLOutput stream) throws SQLException
  {
    super.writeSQL (stream);        // write supertype
                                    // attributes
    stream.writeInt (deptid);
```

```
    stream.writeString (major);
  }
}
```

Although not required, it is recommended that the customized classes, which implement the `SQLData` interface, mirror the database object type hierarchy. For example, you could have declared the `Student` class without a superclass. In this case, `Student` would contain fields to hold the inherited attributes from `PERSON_T` as well as the attributes declared by `STUDENT_T`.

Student.java using SQLData

Code for the `Student.java` class, which does not extend the `Person.java` class, but implements the SQLData interface directly:

```
import java.sql.*;

public class Student implements SQLData
{
  private String sql_type;

  public int ssn;
  public String name;
  public String address;
  public int deptid;
  public String major;

  public Student () {}

  public String getSQLTypeName() throws SQLException { return sql_type; }

  public void readSQL(SQLInput stream, String typeName) throws SQLException
  {
    sql_type = typeName;
    ssn = stream.readInt();
    name = stream.readString();
    address = stream.readString();
    deptid = stream.readInt();
    major = stream.readString();
  }

  public void writeSQL(SQLOutput stream) throws SQLException
  {
    stream.writeInt (ssn);
    stream.writeString (name);
    stream.writeString (address);
    stream.writeInt (deptid);
    stream.writeString (major);
  }
}
```

15.4.3 About Retrieving Subtype Objects

In a typical JDBC application, a subtype object is returned as one of the following:

- A query result
- A PL/SQL `OUT` parameter
- A type attribute

You can use either the default mapping or the `SQLData` mapping or the `OracleData` mapping to retrieve a subtype.

Using Default Mapping

By default, a database object is returned as an instance of the `oracle.jdbc.OracleStruct` interface. This instance may represent an object of either the declared type or subtype of the declared type. If the `OracleStruct` interface represents a subtype object in the database, then it contains the attributes of its supertype as well as those defined in the subtype.

Oracle JDBC driver returns database objects in their most specific type. The JDBC application can use the `getSQLTypeName` method of the `OracleStruct` interface to determine the SQL type of the `STRUCT` object. The following code shows this:

```
// tab1.person column can store PERSON_T, STUDENT_T and PARTIMESTUDENT_T objects
ResultSet rset = stmt.executeQuery ("select person from tab1");
while (rset.next())
{
  oracle.sql.STRUCT s = (oracle.sql.STRUCT) rset.getObject(1);
  if (s != null)
    System.out.println (s.getSQLTypeName());    // print out the type name which
    // may be HR.PERSON_T, HR.STUDENT_T or HR.PARTTIMESTUDENT_T
}
```

Using SQLData Mapping

With `SQLData` mapping, the JDBC driver returns the database object as an instance of the class implementing the `SQLData` interface.

To use `SQLData` mapping in retrieving database objects, do the following:

1. Implement the container classes that implement the `SQLData` interface for the desired object types.
2. Populate the connection type map with entries that specify what custom Java type corresponds to each Oracle object type.
3. Use the `getObject` method to access the SQL object values.

   The JDBC driver checks the type map for an entry match. If one exists, then the driver returns the database object as an instance of the class implementing the `SQLData` interface.

The following code shows the whole SQLData customized mapping process:

```
// The JDBC application developer implements Person.java for PERSON_T,
// Student.java for STUDENT_T
// and ParttimeStudent.java for PARTTIMESTUDEN_T.

Connection conn = ...;  // make a JDBC connection

// obtains the connection typemap
java.util.Map map = conn.getTypeMap ();

// populate the type map
map.put ("HR.PERSON_T", Class.forName ("Person"));
map.put ("HR.STUDENT_T", Class.forName ("Student"));
map.put ("HR.PARTTIMESTUDENT_T", Class.forName ("ParttimeStudent"));

// tab1.person column can store PERSON_T, STUDENT_T and PARTTIMESTUDENT_T objects
ResultSet rset = stmt.executeQuery ("select person from tab1");
while (rset.next())
{
  // "s" is instance of Person, Student or ParttimeStudent
  Object s = rset.getObject(1);
```

```
  if (s != null)
  {
    if (s instanceof Person)
      System.out.println ("This is a Person");
    else if (s instanceof Student)
      System.out.println ("This is a Student");
    else if (s instanceof ParttimeStudent)
      System.out.pritnln ("This is a PartimeStudent");
    else
      System.out.println ("Unknown type");
  }
}
```

The JDBC drivers check the connection type map for each call to the following:

- `getObject` method of the `java.sql.ResultSet` and `java.sql.CallableStatement` interfaces
- `getAttribute` method of the `java.sql.Struct` interface
- `getArray` method of the `java.sql.Array` interface
- `getValue` method of the `oracle.sql.REF` interface

Using OracleData Mapping

With `OracleData` mapping, the JDBC driver returns the database object as an instance of the class implementing the `OracleData` interface.

Oracle JDBC driver needs to be informed of what Java class is mapped to the Oracle object type. The following are the two ways to inform Oracle JDBC drivers:

- The JDBC application uses the `getObject(int idx, OracleDataFactory f)` method to access database objects. The second parameter of the `getObject` method specifies an instance of the factory class that produces the customized class. The `getObject` method is available in the `OracleResultSet` and `OracleCallableStatement` interfaces.
- The JDBC application populates the connection type map with entries that specify what custom Java type corresponds to each Oracle object type. The `getObject` method is used to access the Oracle object values.

The second approach involves the use of the standard `getObject` method. The following code example demonstrates the first approach:

```
// tab1.person column can store both PERSON_T and STUDENT_T objects
ResultSet rset = stmt.executeQuery ("select person from tab1");
while (rset.next())
{
  Object s = rset.getObject(1, PersonFactory.getOracleDataFactory());
  if (s != null)
  {
    if (s instanceof Person)
      System.out.println ("This is a Person");
    else if (s instanceof Student)
      System.out.println ("This is a Student");
    else if (s instanceof ParttimeStudent)
      System.out.pritnln ("This is a PartimeStudent");
    else
      System.out.println ("Unknown type");
  }
}
```

15.4.4 Creating Subtype Objects

There are cases where JDBC applications create database subtype objects with JDBC drivers. These objects are sent either to the database as bind variables or are used to exchange information within the JDBC application.

With customized mapping, the JDBC application creates either `SQLData`-based or `OracleData`-based objects, depending on the approach you choose, to represent database subtype objects. With default mapping, the JDBC application creates `STRUCT` objects to represent database subtype objects. All the data fields inherited from the supertype as well as all the fields defined in the subtype must have values. The following code demonstrates this:

```
Connection conn = ...   // make a JDBC connection
...
Object[] attrs = {
  new Integer(1234), "HR", "500 Oracle Parkway", // data fields defined in
                                                 // PERSON_T
  new Integer(102), "CS",                        // data fields defined in
                                                 // STUDENT_T
  new Integer(4)                                 // data fields defined in
                                                 // PARTTIMESTUDENT_T
};
Struct s = conn.createStruct("HR.PARTTIMESTUDENT", attrs);
```

`s` is initialized with data fields inherited from `PERSON_T` and `STUDENT_T`, and data fields defined in `PARTTIMESTUDENT_T`.

15.4.5 Sending Subtype Objects

In a typical JDBC application, a Java object that represents a database object is sent to the databases as one of the following:

- A data manipulation language (DML) bind variable
- A PL/SQL `IN` parameter
- An object type attribute value

The Java object can be an instance of the `STRUCT` class or an instance of the class implementing either the `SQLData` or `OracleData` interface. Oracle JDBC driver will convert the Java object into the linearized format acceptable to the database SQL engine. Binding a subtype object is the same as binding a standard object.

15.4.6 Accessing Subtype Data Fields

While the logic to access subtype data fields is part of the customized class, this logic for default mapping is defined in the JDBC application itself. The database objects are returned as instances of the `oracle.jdbc.OracleStruct` class. The JDBC application needs to call one of the following access methods in the `STRUCT` class to access the data fields:

- `Object[] getAttribute()`
- `oracle.sql.Datum[] getOracleAttribute()`

Subtype Data Fields from the getAttribute Method

The `getAttribute` method of the `java.sql.Struct` interface is used in JDBC 2.0 to access object data fields. This method returns a `java.lang.Object` array, where each array element

represents an object attribute. You can determine the individual element type by referencing the corresponding attribute type in the JDBC conversion matrix. For example, a SQL `NUMBER` attribute is converted to a `java.math.BigDecimal` object. The `getAttribute` method returns all the data fields defined in the supertype of the object type as well as data fields defined in the subtype. The supertype data fields are listed first followed by the subtype data fields.

Subtype Data Fields from the getOracleAttribute Method

The `getOracleAttribute` method is an Oracle extension method and is more efficient than the `getAttribute` method. The `getOracleAttribute` method returns an `oracle.sql.Datum` array to hold the data fields. Each element in the `oracle.sql.Datum` array represents an attribute. You can determine the individual element type by referencing the corresponding attribute type in the Oracle conversion matrix. For example, a SQL `NUMBER` attribute is converted to an `oracle.sql.NUMBER` object. The `getOracleAttribute` method returns all the attributes defined in the supertype of the object type, as well as attributes defined in the subtype. The supertype data fields are listed first followed by the subtype data fields.

The following code shows the use of the `getAttribute` method:

```
// tab1.person column can store PERSON_T, STUDENT_T and PARTIMESTUDENT_T objects
ResultSet rset = stmt.executeQuery ("select person from tab1");
while (rset.next())
{
  oracle.sql.STRUCT s = (oracle.sql.STRUCT) rset.getObject(1);
  if (s != null)
  {
    String sqlname = s.getSQLTypeName();

    Object[] attrs = s.getAttribute();

    if (sqlname.equals ("HR.PERSON")
    {
      System.out.println ("ssn="+((BigDecimal)attrs[0]).intValue());
      System.out.println ("name="+((String)attrs[1]));
      System.out.println ("address="+((String)attrs[2]));
    }
    else if (sqlname.equals ("HR.STUDENT"))
    {
      System.out.println ("ssn="+((BigDecimal)attrs[0]).intValue());
      System.out.println ("name="+((String)attrs[1]));
      System.out.println ("address="+((String)attrs[2]));
      System.out.println ("deptid="+((BigDecimal)attrs[3]).intValue());
      System.out.println ("major="+((String)attrs[4]));
    }
    else if (sqlname.equals ("HR.PARTTIMESTUDENT"))
    {
      System.out.println ("ssn="+((BigDecimal)attrs[0]).intValue());
      System.out.println ("name="+((String)attrs[1]));
      System.out.println ("address="+((String)attrs[2]));
      System.out.println ("deptid="+((BigDecimal)attrs[3]).intValue());
      System.out.println ("major="+((String)attrs[4]));
      System.out.println ("numHours="+((BigDecimal)attrs[5]).intValue());
    }
    else
      throw new Exception ("Invalid type name: "+sqlname);
  }
}
rset.close ();
stmt.close ();
conn.close ();
```

15.4.7 Inheritance Metadata Methods

Oracle JDBC drivers provide a set of metadata methods to access inheritance properties. The inheritance metadata methods are defined in the `oracle.sql.StructDescriptor` and `oracle.jdbc.StructMetaData` classes.

The `StructMetaData` class provides inheritance metadata methods for subtype attributes. The `getMetaData` method of the `StructDescriptor` class returns an instance of `StructMetaData` of the type. The `StructMetaData` class contains the following inheritance metadata methods:

15.5 About Describing an Object Type

Oracle JDBC includes functionality to retrieve information about a structured object type regarding its attribute names and types. This is similar conceptually to retrieving information from a result set about its column names and types, and in fact uses an almost identical method.

This section covers the following topics:

- Functionality for Getting Object Metadata
- Retrieving Object Metadata

15.5.1 Functionality for Getting Object Metadata

The `oracle.sql.StructDescriptor` class includes functionality to retrieve metadata about a structured object type. The `StructDescriptor` class has a `getMetaData` method with the same functionality as the standard `getMetaData` method available in result set objects. It returns a set of attribute information, such as attribute names and types. Call this method on a `StructDescriptor` object to get metadata about the Oracle object type that the `StructDescriptor` object describes.

The signature of the `StructDescriptor` class `getMetaData` method is the same as the signature specified for `getMetaData` in the standard `ResultSet` interface. The signature is as follows:

```
ResultSetMetaData getMetaData() throws SQLException
```

However, this method actually returns an instance of `oracle.jdbc.StructMetaData`, a class that supports structured object metadata in the same way that the standard `java.sql.ResultSetMetaData` interface specifies support for result set metadata.

The following method is also supported by `StructMetaData`:

```
String getOracleColumnClassName(int column) throws SQLException
```

This method returns the fully qualified name of the `oracle.sql.Datum` subclass whose instances are manufactured if the `OracleResultSet` interface `getOracleObject` method is called to retrieve the value of the specified attribute. For example, `oracle.sql.NUMBER`.

To use the `getOracleColumnClassName` method, you must cast the `ResultSetMetaData` object, which that was returned by the `getMetaData` method, to `StructMetaData`.

- **Note:**

  In all the preceding method signatures, `column` is something of a misnomer. Where you specify a value of 4 for `column`, you really refer to the fourth attribute of the object.

15.5.2 Retrieving Object Metadata

Use the following steps to obtain metadata about a structured object type:

1. Create or acquire a `StructDescriptor` instance that describes the relevant structured object type.
2. Call the `getMetaData` method on the `StructDescriptor` instance.
3. Call the metadata getter methods, `getColumnName`, `getColumnType`, and `getColumnTypeName`, as desired.

Note:

If one of the structured object attributes is itself a structured object, repeat steps 1 through 3.

Example 15-1 Example

The following method shows how to retrieve information about the attributes of a structured object type. This includes the initial step of creating a `StructDescriptor` instance.

```
//
// Print out the ADT's attribute names and types
//
void getAttributeInfo (Connection conn, String type_name) throws SQLException
{

  // get the type descriptor
  StructDescriptor desc = StructDescriptor.createDescriptor (type_name, conn);

  // get type metadata
  ResultSetMetaData md = desc.getMetaData ();

  // get # of attrs of this type
  int numAttrs = desc.length ();

  // temporary buffers
  String attr_name;
  int attr_type;
  String attr_typeName;

  System.out.println ("Attributes of "+type_name+" :");
  for (int i=0; i<numAttrs; i++)
  {
    attr_name = md.getColumnName (i+1);
    attr_type = md.getColumnType (i+1);
    System.out.println (" index"+(i+1)+" name="+attr_name+" type="+attr_type);

    // drill down nested object
```

```
      if (attrType == OracleTypes.STRUCT)
      {
        attr_typeName = md.getColumnTypeName (i+1);

        // recursive calls to print out nested object metadata
        getAttributeInfo (conn, attr_typeName);
      }
    }
  }
```