20
JDBC RowSets

This chapter contains the following sections:

20.1 Overview of JDBC RowSets

A RowSet is an object that encapsulates a set of rows from either Java Database Connectivity (JDBC) result sets or tabular data sources. RowSets support component-based development models like JavaBeans, with a standard set of properties and an event notification mechanism.

The JSR-114 specification includes implementation details for five types of RowSet:

- `CachedRowSet`
- `JdbcRowSet`
- `WebRowSet`
- `FilteredRowSet`
- `JoinRowSet`

Oracle JDBC supports all five types of RowSets through the interfaces and classes present in the `oracle.jdbc.rowset` package. RowSets support is uniform across all Oracle JDBC driver types. The standard Oracle JDBC Java Archive (JAR) files contain the `oracle.jdbc.rowset` package.

To use the Oracle RowSet implementations, you need to import either the entire `oracle.jdbc.rowset` package or specific classes and interfaces from the package for the required RowSet type. For client-side usage, you also need to include the standard Oracle JAR files like `ojdbc8.jar`, `ojdbc11.jar`, or `ojdbc17.jar` in the `CLASSPATH` environment variable.

This section covers the following topics:

20.1.1 RowSet Properties

The `javax.sql.RowSet` interface provides a set of JavaBeans properties that can be altered to access the data in the data source through a single interface. Example of properties are connection string, user name, password, type of connection, and the query string.

> **See Also:**
>
> The Java 2 Platform, Standard Edition (J2SE) Javadoc for a complete list of properties and property descriptions at http://docs.oracle.com/javase/1.5.0/docs/api/javax/sql/RowSet.html

The interface provides standard accessor methods for setting and retrieving the property values. The following code illustrates setting some of the `RowSet` properties:

```
...
rowset.setUrl("jdbc:oracle:oci:@");
rowset.setUsername("HR");
rowset.setPassword("hr");
rowset.setCommand("SELECT employee_id, first_name, last_name, salary FROM employees");
...
```

In this example, the URL, user name, password, and SQL query are set as the `RowSet` properties to retrieve the employee number, employee name, and salary of all the employees into the `RowSet` object.

20.1.2 Events and Event Listeners

RowSets support JavaBeans events. The following types of events are supported by the `RowSet` interface:

- `cursorMoved`

  This event is generated whenever there is a cursor movement. For example, when the `next` or `previous` method is called.

- `rowChanged`

  This event is generated when a row is inserted, updated, or deleted from the RowSet.

- `rowSetChanged`

  This event is generated when the whole RowSet is created or changed. For example, when the `execute` method is called.

An application component can implement a RowSet listener to listen to these RowSet events and perform desired operations when the event occurs. Application components, which are interested in these events, must implement the standard `javax.sql.RowSetListener` interface and register such listener objects with a `RowSet` object. A listener can be registered using the `RowSet.addRowSetListener` method and unregistered using the `RowSet.removeRowSetListener` method. Multiple listeners can be registered with the same `RowSet` object.

The following code illustrates the registration of a RowSet listener:

```
...
MyRowSetListener rowsetListener = new MyRowSetListener ();
// adding a rowset listener
rowset.addRowSetListener (rowsetListener);
...
```

The following code illustrates a listener implementation:

```
public class MyRowSetListener implements RowSetListener
 {
   public void cursorMoved(RowSetEvent event)
   {
     // action on cursor movement
   }

   public void rowChanged(RowSetEvent event)
   {
     // action on change of row
   }

   public void rowSetChanged(RowSetEvent event)
   {
     // action on changing of rowset
   }
 }// end of class MyRowSetListener
```

Applications that need to handle only selected events can implement only the required event handling methods by using the `oracle.jdbc.rowset.OracleRowSetListenerAdapter` class, which is an abstract class with empty implementation for all the event handling methods. In the following code, only the `rowSetChanged` event is handled, while the remaining events are not handled by the application:

```
...
rowset.addRowSetListener(new oracle.jdbc.rowset.OracleRowSetListenerAdapter ()
   {
     public void rowSetChanged(RowSetEvent event)
     {
       // your action for rowSetChanged
     }
   }
 );
...
```

20.1.3 Command Parameters and Command Execution

The `command` property of a `RowSet` object typically represents a SQL query string, which when processed would populate the `RowSet` object with actual data. Like in regular JDBC processing, this query string can take input or bind parameters. The `javax.sql.RowSet` interface also provides methods for setting input parameters to this SQL query. After the required input parameters are set, the SQL query can be processed to populate the `RowSet` object with data from the underlying data source. The following code illustrates this simple sequence:

```
 ...
  rowset.setCommand("SELECT first_name, last_name, salary FROM employees WHERE
employee_id = ?");
  // setting the employee number input parameter for employee named "Douglas"
  rowset.setInt(1, 199);
  rowset.execute();
  ...
```

In the preceding example, the employee number 199 is set as the input or bind parameter for the SQL query specified in the `command` property of the `RowSet` object. When the SQL query is processed, the `RowSet` object is filled with the employee name and salary information of the employee whose employee number is 199.

20.1.4 About Traversing RowSets

The `javax.sql.RowSet` interface extends the `java.sql.ResultSet` interface. The `RowSet` interface, therefore, provides cursor movement and positioning methods, which are inherited from the `ResultSet` interface, for traversing through data in a `RowSet` object. Some of the inherited methods are `absolute`, `beforeFirst`, `afterLast`, `next`, and `previous`.

The `RowSet` interface can be used just like a `ResultSet` interface for retrieving and updating data. The `RowSet` interface provides an optional way to implement a scrollable and updatable result set. All the fields and methods provided by the `ResultSet` interface are implemented in `RowSet`.

> **Note:**
>
> The Oracle implementation of `ResultSet` provides the scrollable and updatable properties of the `java.sql.ResultSet` interface.

The following code illustrates how to scroll through a RowSet:

```
/**
 *  Scrolling forward, and printing the empno in
 *  the order in which it was fetched.
 */
...
rowset.setCommand("SELECT empno, ename, sal FROM emp");
rowset.execute();
...
// going to the first row of the rowset
rowset.beforeFirst ();
while (rowset.next ())
  System.out.println ("empno: " +rowset.getInt (1));
```

In the preceding code, the cursor position is initialized to the position before the first row of the RowSet by the `beforeFirst` method. The rows are retrieved in forward direction using the `next` method.

The following code illustrates how to scroll through a RowSet in the reverse direction:

```
/**
 *  Scrolling backward, and printing the empno in
 *  the reverse order as it was fetched.
 */
//going to the last row of the rowset
rowset.afterLast ();
while (rowset.previous ())
  System.out.println ("empno: " +rowset.getInt (1));
```

In the preceding code, the cursor position is initialized to the position after the last row of the `RowSet`. The rows are retrieved in reverse direction using the `previous` method of `RowSet`.

Inserting, updating, and deleting rows are supported by the Row Set feature as they are in the Result Set feature. In order to make the Row Set updatable, you must call the `setReadOnly(false)` and `acceptChanges` methods.

The following code illustrates the insertion of a row at the fifth position of a Row Set:

```
...
/**
  * Make rowset updatable
  */
rowset.setReadOnly (false);
/**
 * Inserting a row in the 5th position of the rowset.
 */
// moving the cursor to the 5th position in the rowset
if (rowset.absolute(5))
{
  rowset.moveToInsertRow ();
  rowset.updateInt (1, 193);
  rowset.updateString (2, "Smith");
  rowset.updateInt (3, 7200);

  // inserting a row in the rowset
  rowset.insertRow ();

  // Synchronizing the data in RowSet with that in the database.
  rowset.acceptChanges ();
}
...
```

In the preceding code, a call to the `absolute` method with a parameter `5` takes the cursor to the fifth position of the RowSet and a call to the `moveToInsertRow` method creates a place for the insertion of a new row into the RowSet. The `update`*`XXX`* methods are used to update the newly created row. When all the columns of the row are updated, the `insertRow` is called to update the RowSet. The changes are committed through `acceptChanges` method.

20.2 About the CachedRowSet Interface

A `CachedRowSet` is a RowSet in which the rows are cached and the RowSet is disconnected, that is, it does not maintain an active connection to the database.

The `oracle.jdbc.rowset.OracleCachedRowSet` class is the Oracle implementation of `CachedRowSet`. It can interoperate with the standard reference implementation. The `OracleCachedRowSet` class, which is present in the `ojdbc8.jar`, `ojdbc11.jar`, and `ojdbc17.jar` files, implements the standard JSR-114 interface `javax.sql.rowset.CachedRowSet`.

In the following code, an `OracleCachedRowSet` object is created and the connection URL, user name, password, and the SQL query for the `RowSet` object is set as properties. The `RowSet` object is populated using the `execute` method. After the `execute` method has been processed, the `RowSet` object can be used as a `java.sql.ResultSet` object to retrieve, scroll, insert, delete, or update data.

```
...
RowSet rowset = new OracleCachedRowSet();
rowset.setUrl("jdbc:oracle:oci:@");
rowset.setUsername("HR");
rowset.setPassword("hr");
rowset.setCommand("SELECT employee_id, first_name, last_name, salary FROM employees");
```

```
rowset.execute();
while (rowset.next ())
{
  System.out.println("employee_id: " +rowset.getInt (1));
  System.out.println("first_name: " +rowset.getString (2));
  System.out.println("last_name: " +rowset.getString (3));
  System.out.println("sal: "   +rowset.getInt (4));
}
...
```

To populate a `CachedRowSet` object with a query, complete the following steps:

1. Instantiate `OracleCachedRowSet`.
2. Set the `Url`, which is the connection URL, `Username`, `Password`, and `Command`, which is the query string, properties for the `RowSet` object. You can also set the connection type, but it is optional.
3. Call the `execute` method to populate the `CachedRowSet` object. Calling `execute` runs the query set as a property on this RowSet.

```
    OracleCachedRowSet rowset = new OracleCachedRowSet ();
    rowset.setUrl ("jdbc:oracle:oci:@");
    rowset.setUsername ("HR");
    rowset.setPassword ("hr");
    rowset.setCommand ("SELECT employee_id, first_name, last_name, salary FROM
employees");
    rowset.execute ();
```

A `CachedRowSet` object can be populated with an existing `ResultSet` object, using the `populate` method. To do so, complete the following steps:

1. Instantiate `OracleCachedRowSet`.
2. Pass the already available `ResultSet` object to the `populate` method to populate the `RowSet` object.

```
    // Executing a query to get the ResultSet object.
    ResultSet rset = pstmt.executeQuery ();

    OracleCachedRowSet rowset = new OracleCachedRowSet ();
    // the obtained ResultSet object is passed to the populate method
    // to populate the data in the rowset object.
    rowset.populate (rset);
```

In the preceding example, a `ResultSet` object is obtained by running a query and the retrieved `ResultSet` object is passed to the `populate` method of the `CachedRowSet` object to populate the contents of the result set into the CachedRowSet.

> **Note:**
>
> Connection properties, like transaction isolation or the concurrency mode of the result set, and the bind properties cannot be set in the case where a pre-existent `ResultSet` object is used to populate the `CachedRowSet` object, because the connection or result set on which the property applies would have already been created.

The following code illustrates how an `OracleCachedRowSet` object is serialized to a file and then retrieved:

```
// writing the serialized OracleCachedRowSet object
{
  FileOutputStream fileOutputStream = new FileOutputStream("emp_tab.dmp");
  ObjectOutputStream ostream = new ObjectOutputStream(fileOutputStream);
  ostream.writeObject(rowset);
  ostream.close();
  fileOutputStream.close();
}

// reading the serialized OracleCachedRowSet object
{
  FileInputStream fileInputStream = new FileInputStream("emp_tab.dmp");
  ObjectInputStream istream = new ObjectInputStream(fileInputStream);
  RowSet rowset1 = (RowSet) istream.readObject();
  istream.close();
  fileInputStream.close();
}
```

In the preceding code, a `FileOutputStream` object is opened for an `emp_tab.dmp` file, and the populated `OracleCachedRowSet` object is written to the file using `ObjectOutputStream`. The serialized `OracleCachedRowSet` object is retrieved using the `FileInputStream` and `ObjectInputStream` objects.

`OracleCachedRowSet` takes care of the serialization of non-serializable form of data like `InputStream`, `OutputStream`, binary large objects (BLOBs), and character large objects (CLOBs). `OracleCachedRowSets` also implements metadata of its own, which could be obtained without any extra server round-trip. The following code illustrates how you can obtain metadata for the RowSet:

```
...
ResultSetMetaData metaData = rowset.getMetaData();
int maxCol = metaData.getColumnCount();
for (int i = 1; i <= maxCol; ++i)
   System.out.println("Column (" + i +") " + metaData.getColumnName(i));
...
```

Because the `OracleCachedRowSet` class is serializable, it can be passed across a network or between Java Virtual Machines (JVMs), as done in Remote Method Invocation (RMI). Once the `OracleCachedRowSet` class is populated, it can move around any JVM, or any environment that does not have JDBC drivers. Committing the data in the RowSet requires the presence of JDBC drivers.

The complete process of retrieving the data and populating it in the `OracleCachedRowSet` class is performed on the server and the populated RowSet is passed on to the client using suitable architectures like RMI or Enterprise Java Beans (EJB). The client would be able to perform all the operations like retrieving, scrolling, inserting, updating, and deleting on the RowSet without any connection to the database. Whenever data is committed to the database, the `acceptChanges` method is called, which synchronizes the data in the RowSet to that in the database. This method makes use of JDBC drivers, which require the JVM environment to contain JDBC implementation. This architecture would be suitable for systems involving a Thin client like a Personal Digital Assistant (PDA).

After populating the `CachedRowSet` object, it can be used as a `ResultSet` object or any other object, which can be passed over the network using RMI or any other suitable architecture.

Some of the other key-features of CachedRowSet are the following:

- Cloning a RowSet
- Creating a copy of a RowSet
- Creating a shared copy of a RowSet

CachedRowSet Constraints

All the constraints that apply to an updatable result set are applicable here, except serialization, because `OracleCachedRowSet` is serializable. The SQL query has the following constraints:

- References only a single table in the database
- Contains no join operations
- Selects the primary key of the table it references

In addition, a SQL query should also satisfy the following conditions, if new rows are to be inserted:

- Selects all non-nullable columns in the underlying table
- Selects all columns that do not have a default value

> **Note:**
>
> The `CachedRowSet` cannot hold a large quantity of data, because all the data is cached in memory. Oracle, therefore, recommends against using `OracleCachedRowSet` with queries that could potentially return a large volume of data.

Connection properties like, transaction isolation and concurrency mode of the result set, cannot be set after populating the RowSet, because the properties cannot be applied to the connection after retrieving the data from the same.

20.3 About the JdbcRowSet Interface

A `JdbcRowSet` is a RowSet that wraps around a `ResultSet` object. It is a connected RowSet that provides JDBC interfaces in the form of a Java Bean interface.

The Oracle implementation of `JdbcRowSet` is `oracle.jdbc.rowset.OracleJDBCRowSet`. The `Atherosclerosis` class, which is present in the `ojdbc8.jar`, `ojdbc11.jar`, and `ojdbc17.jar` files, implements the standard JSR-114 interface `javax.sql.rowset.JdbcRowSet`.

Table 20-1 shows how the `JdbcRowSet` interface differs from `CachedRowSet` interface.

Table 20-1 Comparison Between the JDBC Row Sets and the Cached Row Sets

RowSet Type	Serializable	Connected to Database	Movable Across JVMs	Synchronization of data to database	Presence of JDBC Drivers
JDBC	Yes	Yes	No	No	Yes
Cached	Yes	No	Yes	Yes	No

`JdbcRowSet` is a connected RowSet, which has a live connection to the database and all the calls on the `JdbcRowSet` are percolated to the mapping call in the JDBC connection, statement, or result set. A `CachedRowSet` does not have any connection to the database open.

`JdbcRowSet` requires the presence of JDBC drivers unlike a `CachedRowSet`, which does not require JDBC drivers during manipulation. However, both `JdbcRowSet` and `CachedRowSet` require JDBC drivers during population of the RowSet and while committing the changes of the RowSet.

The following code illustrates how a `JdbcRowSet` is used:

```
...
RowSet rowset = new Atherosclerosis();
rowset.setUrl("java:oracle:oci:@");
rowset.setUsername("HR");
rowset.setPassword("hr");
rowset.setCommand("SELECT empno, ename, sal FROM emp");
rowset.execute();
while (rowset.next())
{
  System.out.println("empno: " + rowset.getInt(1));
  System.out.println("ename: " + rowset.getString(2));
  System.out.println("sal: " + rowset.getInt(3));
}
...
```

In the preceding example, the connection URL, user name, password, and SQL query are set as properties of the `RowSet` object, the SQL query is processed using the `execute` method, and the rows are retrieved and printed by traversing through the data populated in the `RowSet` object.

20.4 About the WebRowSet Interface

A `WebRowSet` is an extension to `CachedRowSet`. It represents a set of fetched rows or tabular data that can be passed between tiers and components in a way such that no active connections with the data source need to be maintained.

The `WebRowSet` interface provides support for the production and consumption of result sets and their synchronization with the data source, both in Extensible Markup Language (XML) format and in disconnected fashion. This allows result sets to be shipped across tiers and over Internet protocols.

The Oracle implementation of `WebRowSet` is `oracle.jdbc.rowset.OracleWebRowSet`. This class, which is in the `ojdbc8.jar`, `ojdbc11.jar`, and `ojdbc17.jar` files, implements the standard JSR-114 interface `javax.sql.rowset.WebRowSet`. This class also extends the `oracle.jdbc.rowset.OracleCachedRowSet` class. Besides the methods available in `OracleCachedRowSet`, the `OracleWebRowSet` class provides the following methods:

- `public OracleWebRowSet() throws SQLException`

  This is the constructor for creating an `OracleWebRowSet` object, which is initialized with the default values for an `OracleCachedRowSet` object, a default `OracleWebRowSetXmlReader`, and a default `OracleWebRowSetXmlWriter`.

- `public void writeXml(java.io.Writer writer) throws SQLException`
  `public void writeXml(java.io.OutputStream ostream) throws SQLException`

These methods write the `OracleWebRowSet` object to the supplied `Writer` or `OutputStream` object in the XML format that conforms to the JSR-114 XML schema. In addition to the RowSet data, the properties and metadata of the RowSet are written.

- `public void writeXml(ResultSet rset, java.io.Writer writer) throws SQLException`
  `public void writeXml(ResultSet rset, java.io.OutputStream ostream) throws SQLException`

  These methods create an `OracleWebRowSet` object, populate it with the data in the given `ResultSet` object, and write it to the supplied `Writer` or `OutputStream` object in the XML format that conforms to the JSR-114 XML schema.

- `public void readXml(java.io.Reader reader) throws SQLException`
  `public void readXml(java.io.InputStream istream) throws SQLException`

  These methods read the `OracleWebRowSet` object in the XML format according to its JSR-114 XML schema, using the supplied `Reader` or `InsputStream` object.

The Oracle WebRowSet implementation supports Java API for XML Processing (JAXP) 1.2. Both Simple API for XML (SAX) 2.0 and Document Object Model (DOM) JAXP-conforming XML parsers are supported. It follows the current JSR-114 W3C XML schema for WebRowSet.

Applications that use the `readXml(...)` methods should set one of the following two standard JAXP system properties before calling the methods:

- `javax.xml.parsers.SAXParserFactory`

  This property is for a SAX parser.

- `javax.xml.parsers.DocumentBuilderFactory`

  This property is for a DOM parser.

The following code illustrates the use of `OracleWebRowSet` for both writing and reading in XML format:

```
import java.sql.*;
import java.io.*;
import oracle.jdbc.rowset.*;

...
String url = "jdbc:oracle:oci8:@";

Connection conn = DriverManager.getConnection(url,"HR","hr");
Statement stmt = conn.createStatement();
ResultSet rset = stmt.executeQuery("select * from employees");

// Create an OracleWebRowSet object and populate it with the ResultSet object
OracleWebRowSet wset = new OracleWebRowSet();
wset.populate(rset);

try
{
  // Create a java.io.Writer object
  FileWriter out = new FileWriter("xml.out");

  // Now generate the XML and write it out
  wset.writeXml(out);
}
catch (IOException exc)
{
  System.out.println("Couldn't construct a FileWriter");
}
```

```
System.out.println("XML output file generated.");

// Create a new OracleWebRowSet for reading from XML input
OracleWebRowSet wset2 = new OracleWebRowSet();

// Use Oracle JAXP SAX parser
System.setProperty("javax.xml.parsers.SAXParserFactory","oracle.xml.jaxp.JXSAXParserFacto
ry");

try
{
  // Use the preceding output file as input
  FileReader fr = new FileReader("xml.out");

  // Now read XML stream from the FileReader
  wset2.readXml(fr);
}
catch (IOException exc)
{
  System.out.println("Couldn't construct a FileReader");
}
...
```

> **Note:**
>
> The preceding code uses the Oracle SAX XML parser, which supports schema validation.

20.5 About the FilteredRowSet Interface

A `FilteredRowSet` is an extension to `WebRowSet` that provides programmatic support for filtering its content. This enables you to avoid the overhead of supplying a query and the processing involved. The Oracle implementation of `FilteredRowSet` is `oracle.jdbc.rowset.OracleFilteredRowSet`. The `OracleFilteredRowSet` class, which is present in the `ojdbc11.jar` and `ojdbc17.jar` files, implements the standard JSR-114 interface `javax.sql.rowset.FilteredRowSet`.

The `OracleFilteredRowSet` class defines the following new methods:

- `public Predicate getFilter();`

  This method returns a `Predicate` object that defines the filtering criteria active on the `OracleFilteredRowSet` object.

- `public void setFilter(Predicate p) throws SQLException;`

  This method takes a `Predicate` object as a parameter. The `Predicate` object defines the filtering criteria to be applied on the `OracleFilteredRowSet` object. The methods throws a `SQLException` exception.

The predicate set on an `OracleFilteredRowSet` object defines a filtering criteria that is applied to all the rows in the object to obtain the set of visible rows. The predicate also defines the criteria for inserting, deleting, and modifying rows. The set filtering criteria acts as a gating mechanism for all views and updates to the `OracleFilteredRowSet` object. Any attempt to update the `OracleFilteredRowSet` object, which violates the filtering criteria, throws a `SQLException` exception.

The filtering criteria set on an `OracleFilteredRowSet` object can be modified by applying a new `Predicate` object. The new criteria is immediately applied on the object, and all further views and updates must adhere to this new criteria. A new filtering criteria can be applied only if there are no reference to the `OracleFilteredRowSet` object.

Rows that fall outside of the filtering criteria set on the object cannot be modified until the filtering criteria is removed or a new filtering criteria is applied. Also, only the rows that fall within the bounds of the filtering criteria will be synchronized with the data source, if an attempt is made to persist the object.

The following code example illustrates the use of OracleFilteredRowSet. Assume a table, `test_table`, with two `NUMBER` columns, `col1` and `col2`. The code retrieves those rows from the table that have value of `col1` between `50` and `100` and value of `col2` between `100` and `200`.

The predicate defining the filtering criteria is as follows:

```
public class PredicateImpl implements Predicate
{
  private int low[];
  private int high[];
  private int columnIndexes[];

  public PredicateImpl(int[] lo, int[] hi, int[] indexes)
  {
    low = lo;
    high = hi;
    columnIndexes = indexes;
  }

  public boolean evaluate(RowSet rs)
  {
    boolean result = true;
    for (int i = 0; i < columnIndexes.length; i++)
    {
      int columnValue = rs.getInt(columnIndexes[i]);
      if (columnValue < low[i] || columnValue > high[i])
        result = false;
    }
    return result;
  }

// the other two evaluate(...) methods simply return true

}
```

The predicate defined in the preceding code is used for filtering content in an `OracleFilteredRowSet` object, as follows:

```
...
OracleFilteredRowSet ofrs = new OracleFilteredRowSet();
int low[] = {50, 100};
int high[] = {100, 200};
int indexes[] = {1, 2};
ofrs.setCommand("select col1, col2 from test_table");

// set other properties on ofrs like usr/pwd ...
...
ofrs.execute();
ofrs.setPredicate(new PredicateImpl(low, high, indexes));

// this will only get rows with col1 in (50,100) and col2 in (100,200)
```

```
while (ofrs.next()) {...}
...
```

20.6 About the JoinRowSet Interface

A `JoinRowSet` is an extension to `WebRowSet` that consists of related data from different RowSets.

There is no standard way to establish a SQL `JOIN` between disconnected RowSets without connecting to the data source. A `JoinRowSet` addresses this issue. The Oracle implementation of `JoinRowSet` is the `oracle.jdbc.rowset.OracleJoinRowSet` class. This class, which is in the `ojdbc11.jar` and `ojdbc17.jar` files, implements the standard JSR-114 interface `javax.sql.rowset.JoinRowSet`.

Any number of `RowSet` objects, which implement the `Joinable` interface, can be added to a `JoinRowSet` object, provided they can be related in a SQL `JOIN`. All five types of RowSet support the `Joinable` interface. The `Joinable` interface provides methods for specifying the columns based on which the `JOIN` will be performed, that is, the match columns.

A match column can be specified in the following ways:

- Using the `setMatchColumn` method

  This method is defined in the `Joinable` interface. It is the only method that can be used to set the match column before a `RowSet` object is added to a `JoinRowSet` object. This method can also be used to reset the match column at any time.

- Using the `addRowSet` method

  This is an overloaded method in `JoinRowSet`. Four of the five implementations of this method take a match column as a parameter. These four methods can be used to set or reset a match column at the time a `RowSet` object is being added to a `JoinRowSet` object.

In addition to the inherited methods, `OracleJoinRowSet` provides the following methods:

- ```
  public void addRowSet(Joinable joinable) throws SQLException;
  public void addRowSet(RowSet rowSet, int i) throws SQLException;
  public void addRowSet(RowSet rowSet, String s) throws SQLException;
  public void addRowSet(RowSet arowSet[], int an[]) throws SQLException;
  public void addRowSet(RowSet arowSet[], String as[]) throws SQLException;
  ```

  These methods are used to add a `RowSet` object to the `OracleJoinRowSet` object. You can pass one or more `RowSet` objects to be added to the `OracleJoinRowSet` object. You can also pass names or indexes of one or more columns, which need to be set as match column.

- ```
  public Collection getRowSets() throws SQLException;
  ```

  This method retrieves the `RowSet` objects added to the `OracleJoinRowSet` object. The method returns a `java.util.Collection` object that contains the `RowSet` objects.

- ```
  public String[] getRowSetNames() throws SQLException;
  ```

  This method returns a String array containing the names of the `RowSet` objects that are added to the `OracleJoinRowSet` object.

- ```
  public boolean supportsCrossJoin();
  public boolean supportsFullJoin();
  public boolean supportsInnerJoin();
  ```

```
public boolean supportsLeftOuterJoin();
public boolean supportsRightOuterJoin();
```

These methods return a boolean value indicating whether the `OracleJoinRowSet` object supports the corresponding `JOIN` type.

- `public void setJoinType(int i) throws SQLException;`

  This method is used to set the `JOIN` type on the `OracleJoinRowSet` object. It takes an integer constant as defined in the `javax.sql.rowset.JoinRowSet` interface that specifies the `JOIN` type.

- `public int getJoinType() throws SQLException;`

  This method returns an integer value that indicates the `JOIN` type set on the `OracleJoinRowSet` object. This method throws a `SQLException` exception.

- `public CachedRowSet toCachedRowSet() throws SQLException;`

  This method creates a `CachedRowSet` object containing the data in the `OracleJoinRowSet` object.

- `public String getWhereClause() throws SQLException;`

  This method returns a String containing the SQL-like description of the `WHERE` clause used in the `OracleJoinRowSet` object. This methods throws a `SQLException` exception.

The following code illustrates how `OracleJoinRowSet` is used to perform an inner join on two RowSets, whose data come from two different tables. The resulting RowSet contains data as if they were the result of an inner join on these two tables. Assume that there are two tables, an `Order` table with two `NUMBER` columns `Order_id` and `Person_id`, and a `Person` table with a `NUMBER` column `Person_id` and a `VARCHAR2` column `Name`.

```
...
// RowSet holding data from table Order
OracleCachedRowSet ocrsOrder = new OracleCachedRowSet();
...
ocrsOrder.setCommand("select order_id, person_id from order");
...
// Join on person_id column
ocrsOrder.setMatchColumn(2);
ocrsOrder.execute();

// Creating the JoinRowSet
OracleJoinRowSet ojrs = new OracleJoinRowSet();
ojrs.addRowSet(ocrsOrder);

// RowSet holding data from table Person
OracleCachedRowSet ocrsPerson = new OracleCachedRowSet();
...
ocrsPerson.setCommand("select person_id, name from person");
...
// do not set match column on this RowSet using setMatchColumn().
//use addRowSet() to set match column
ocrsPerson.execute();

// Join on person_id column, in another way
ojrs.addRowSet(ocrsPerson, 1);

// now we can go the JoinRowSet as usual
ojrs.beforeFirst();
while (ojrs.next())
```

```
System.out.println("order id = " + ojrs.getInt(1) + ", " + "person id = " +
ojrs.getInt(2) + ", " + "person's name = " + ojrs.getString(3));
...
```