24

JDBC Reactive Extensions

The Reactive Extensions are a set of methods that extend the JDBC standard to offer asynchronous database access.

The Reactive Extensions use non-blocking mechanisms for creating connection objects, executing SQL statements, fetching rows, committing transactions, rolling back transactions, closing Connection objects, and reading and writing BFILEs, BLOBs, and CLOBs.

This chapter includes the following topics:

- Overview of JDBC Reactive Extensions
- About Building an Application with Reactive Extensions
- Threading Model of Asynchronous Methods
- About the Flow API
- Using the FlowAdapters Class
- Streaming Row Data with the Reactor Library
- Streaming Row Data with the RxJava Library
- Streaming Row Data with the Akka Streams Library
- Limitations of JDBC Reactive Extensions

24.1 Overview of JDBC Reactive Extensions

The Reactive Extensions implement the Publisher and Subscriber types defined by the `java.util.concurrent.Flow` interfaces, which is the standard JDK representation of a reactive stream.

The Reactive Extensions use a single Java NIO Selector for nonblocking Database operations.

Requirements for Using JDBC Reactive Extensions

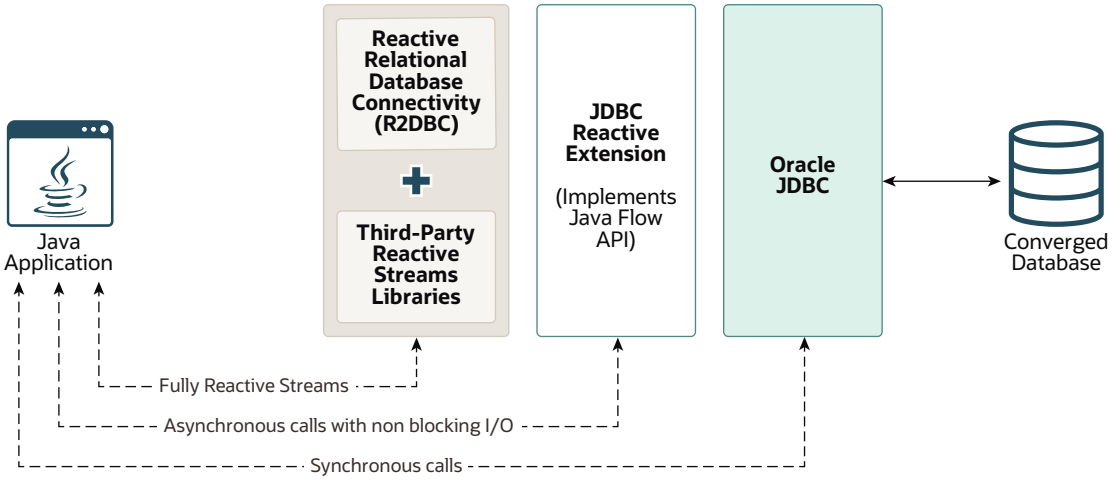
For using the JDBC Reactive Extensions, you must use the following:

- JDBC Thin Driver 21c or later for building connections
- JDK 11, or JDK 17 or later
- `ojdbc11.jar` or `ojdbc17.jar`

Architecture of JDBC Reactive Extensions

The following diagram illustrates the architecture of applications using JDBC Reactive Extensions:

Figure 24-1 Architecture of Applications Using JDBC Reactive Extensions



24.2 About Building an Application with Reactive Extensions

This section describes the steps that you must follow to build an application using the Reactive Extensions.

Building an application using the Reactive Extensions involves the same steps as building an application using the standard methods. But, in case of Reactive Extensions, you use the new asynchronous methods. This section describes how to use the various asynchronous methods in the following sections:

- Opening a Connection Using Asynchronous Methods
- Execution of SQL Statements with Asynchronous Methods
- About Fetching Row Data with Asynchronous Methods
- Reading LOB Data Using Asynchronous Methods
- Writing LOB Data Using Asynchronous Methods
- Committing a Transaction Using Asynchronous Methods
- Closing a Connection Using Asynchronous Methods

24.2.1 Opening a Connection Using Asynchronous Methods

The `OracleConnectionBuilder` interface provides methods for opening a connection asynchronously.

The `OracleConnectionBuilder.buildConnectionPublisherOracle` method returns a `Flow.Publisher<OracleConnection>` type. The Publisher emits a single Connection to a Subscriber. Once the Subscriber signals demand, the Publisher asynchronously opens a new Connection. The Published Connection is identical to a Connection that you can build using the `ConnectionBuilder.build` method.

The following example demonstrates how to asynchronously open a connection.

```
/**
 * Asynchronously opens a new connection
 * @param dataSource Datasource configured with a URL, User, and Password
```

```
   * @return A Publisher that emits a single connection
   * @throws SQLException If a database access error occurs before the
   * connection can be opened
   */
  Flow.Publisher<OracleConnection> openConnection(DataSource dataSource)
    throws SQLException {
    return dataSource.unwrap(OracleDataSource.class)
      .createConnectionBuilder()
      .buildConnectionPublisherOracle();
  }
```

24.2.2 Execution of SQL Statements with Asynchronous Methods

This section describes how to execute SQL statements with asynchronous methods.

The `OraclePreparedStatement` interface exposes methods for asynchronous SQL execution. Each asynchronous method performs a function that is analogous to the corresponding synchronous method of SQL execution. This relationship is expressed in the following table:

Table 24-1 Method Comparison

Synchronous Method	Asynchronous Method
`boolean execute`	`Flow.Publisher<Boolean> executeAsyncOracle`
`long executeLargeUpdate`	`Flow.Publisher<Long> executeUpdateAsyncOracle`
`long[] executeLargeBatch`	`Flow.Publisher<Long> executeBatchAsyncOracle`
`ResultSet executeQuery`	`Flow.Publisher<OracleResultSet> executeQueryAsyncOracle`

The following sections provide more information about the asynchronous methods:

- Standard SQL Statement Execution with the executeAsyncOracle Method
- DML Statement Execution with the executeUpdateAsyncOracle method
- Batch DML Statement Execution with the executeBatchAsyncOracle Method
- SQL Query Execution with the executeQueryAsyncOracle Method

24.2.2.1 Standard SQL Statement Execution with the executeAsyncOracle Method

This section describes the `executeAsyncOracle` method, which is equivalent to the standard `execute` method.

Any type of SQL statement can be executed by calling the `OraclePreparedStatement.executeAsyncOracle` method. This call returns a `Flow.Publisher<Boolean>` type. The Publisher emits a single Boolean and supports multiple Subscribers. If the Boolean value is `TRUE`, then it means that the SQL statement has resulted in row data, which is accessible from the `OraclePreparedStatement.getResultSet` method. If it

is `FALSE`, then it means that the SQL statement has returned an update count. The Boolean result is semantically equivalent to the `boolean` returned by the `execute` method.

```
/**
 * Asynchronously creates a new table by executing a DDL SQL statement
 * @param connection Connection to a database where the table is created
 * @return A Publisher that emits the result of executing DDL SQL
 * @throws SQLException If a database access error occurs before the DDL
 * SQL can be executed
 */
Flow.Publisher<Boolean> createTable(Connection connection)
  throws SQLException {

  PreparedStatement createTableStatement =
    connection.prepareStatement(
      "CREATE TABLE employee_names (" +
        "id NUMBER PRIMARY KEY, " +
        "first_name VARCHAR(50), " +
        "last_name VARCHAR2(50))");

  Flow.Publisher<Boolean> createTablePublisher =
    createTableStatement.unwrap(OraclePreparedStatement.class)
      .executeAsyncOracle();

  createTablePublisher.subscribe(
    // This subscriber will close the PreparedStatement
    new Flow.Subscriber<Boolean>() {
      public void onSubscribe(Flow.Subscription subscription) {
        subscription.request(1L);
      }
      public void onNext(Boolean item) { }
      public void onError(Throwable throwable) { closeStatement(); }
      public void onComplete() { closeStatement(); }
      void closeStatement() {
        try { createTableStatement.close(); }
        catch (SQLException closeException) { log(closeException); }
      }
    });

  return createTablePublisher;
}
```

24.2.2.2 DML Statement Execution with the executeUpdateAsyncOracle method

This section describes the `executeUpdateAsyncOracle` method, which is equivalent to the standard `executeLargeUpdate` method.

You can use the `OraclePreparedStatement.executeUpdateAsyncOracle` method to execute single (non-batch) DML statements. This call returns a `Flow.Publisher<Long>` type. The returned publisher emits a single `Long` value. This `Long` value indicates the number of rows updated or to be inserted by the DML statement. This `Long` value result is semantically equivalent to the `long` value returned by the standard `executeLargeUpdate` method.

```
/**
 * Asynchronously updates table data by executing a DML SQL statement
```

```
   * @param connection Connection to a database where the table data resides
   * @return A Publisher that emits the number of rows updated
   * @throws SQLException If a database access error occurs before the DML
   * SQL can be executed
   */
  Flow.Publisher<Long> updateData(Connection connection)
    throws SQLException {

    PreparedStatement updateStatement = connection.prepareStatement(
      "UPDATE employee_names SET " +
        "first_name = UPPER(first_name), " +
        "last_name = UPPER(last_name)");

    Flow.Publisher<Long> updatePublisher =
      updateStatement.unwrap(OraclePreparedStatement.class)
        .executeUpdateAsyncOracle();

    updatePublisher.subscribe(
      // This subscriber will close the PreparedStatement
      new Flow.Subscriber<Long>() {
        public void onSubscribe(Flow.Subscription subscription) {
          subscription.request(1L);
        }
        public void onNext(Long item) { }
        public void onError(Throwable throwable) { closeStatement(); }
        public void onComplete() { closeStatement(); }
        void closeStatement() {
          try { updateStatement.close(); }
          catch (SQLException closeException) { log(closeException); }
        }
      });

    return updatePublisher;
  }
```

24.2.2.3 Batch DML Statement Execution with the executeBatchAsyncOracle Method

This section describes the `executeBatchAsyncOracle` method, which is equivalent to the standard `executeLargeBatch` method.

You can use the `OraclePreparedStatement.executeBatchAsyncOracle` method to execute batch DML statements. This call returns a `Flow.Publisher<Long>` type. The returned publisher emits a `Long` value for each statement in the batch. The `Long` values indicate the number of rows updated by each DML statement. These `Long` value results are semantically equivalent to the `long[]` value returned by the standard `executeLargeBatch` method.

```
  /**
   * Asynchronously loads table data by executing a batch of DML SQL
statements.
   * @param connection Connection to a database where the table data resides.
   * @return A Publisher which emits the number of rows updated.
   * @throws SQLException If a database access error occurs before the DML
   * SQL can be executed.
   */
  Flow.Publisher<Long> createData(
```

```
  Connection connection, Iterable<Employee> employeeData)
  throws SQLException {

  PreparedStatement batchStatement = connection.prepareStatement(
    "INSERT INTO employee_names (id, first_name, last_name) " +
      "VALUES (?, ?, ?)");

  for (Employee employee : employeeData) {
    batchStatement.setLong(1, employee.id());
    batchStatement.setString(2, employee.firstName());
    batchStatement.setString(3, employee.lastName());
    batchStatement.addBatch();
  }

  Flow.Publisher<Long> batchPublisher =
    batchStatement.unwrap(OraclePreparedStatement.class)
      .executeBatchAsyncOracle();

  batchPublisher.subscribe(
    // This subscriber will close the PreparedStatement
    new Flow.Subscriber<Long>() {
      public void onSubscribe(Flow.Subscription subscription) {
        subscription.request(Long.MAX_VALUE);
      }
      public void onNext(Long item) { }
      public void onError(Throwable throwable) { closeStatement(); }
      public void onComplete() { closeStatement(); }
      void closeStatement() {
        try { batchStatement.close(); }
        catch (SQLException closeException) { log(closeException); }
      }
    });

  return batchPublisher;
}
```

24.2.2.4 SQL Query Execution with the executeQueryAsyncOracle Method

This section describes the `executeQueryAsyncOracle` Method, which is equivalent to the standard `executeQuery` method.

You can execute SQL query statements with the `OraclePreparedStatement.executeQueryAsyncOracle` method. This call returns a `Flow.Publisher<OracleResultSet>` type. The returned publisher emits a single `OracleResultSet` value. The `OracleResultSet` value provides access to row data that is resulted from the SQL query. This `OracleResultSet` is semantically equivalent to the `ResultSet` returned by the standard `executeQuery` method.

```
/**
 * Asynchronously reads table data by executing a SELECT SQL statement
 * @param connection Connection to a database where the table resides
 * @return A Publisher that emits the number of rows updated
 * @throws SQLException If a database access error occurs before the SELECT
 * SQL can be executed
 */
```

```
  Flow.Publisher<OracleResultSet> readData(Connection connection)
    throws SQLException {

    PreparedStatement queryStatement = connection.prepareStatement(
      "SELECT id, first_name, last_name FROM employee_names");

    Flow.Publisher<OracleResultSet> queryPublisher =
      queryStatement.unwrap(OraclePreparedStatement.class)
        .executeQueryAsyncOracle();

    // Close the PreparedStatement after the result set is consumed.
    queryStatement.closeOnCompletion();

    return queryPublisher;
  }
```

24.2.3 About Fetching Row Data with Asynchronous Methods

This section describes how to fetch row data with asynchronous methods.

The `OracleResultSet` interface exposes the `publisherOracle(Function<OracleRow, T>)` method for asynchronous row data fetching. The argument to this method is a mapping function for row data. The mapping function is applied to each row of the `ResultSet`. The method returns a `Flow.Publisher<T>` type, where T is the output type of the mapping function. The input type of the mapping function is `OracleRow`. An `OracleRow` represents a single row of the `ResultSet`, and exposes methods to access the column values of that row.

The following example demonstrates how you can fetch row data with asynchronous methods:

```
  /**
   * Asynchronously fetches table data by from a ResultSet.
   * @param resultSet ResultSet which fetches table data.
   * @return A Publisher which emits the fetched data as Employee objects.
   * @throws SQLException If a database access error occurs before table data
is
   * fetched.
   */
  Flow.Publisher<Employee> fetchData(ResultSet resultSet)
    throws SQLException {
    // Before closing the ResultSet with publisherOracle(..), first obtain a
    // reference to the ResultSet's Statement. The Statement needs to be
closed
    // after all data has been fetched.
    Statement resultSetStatement = resultSet.getStatement();

    Flow.Publisher<Employee> employeePublisher =
      resultSet.unwrap(OracleResultSet.class)
        .publisherOracle(oracleRow -> {
          try {
            return new Employee(
              oracleRow.getObject("id", Long.class),
              oracleRow.getObject("first_name", String.class),
              oracleRow.getObject("last_name", String.class));
          }
          catch (SQLException getObjectException) {
```

```
            // Unchecked exceptions thrown by a row mapping function will be
            // emitted to each Subscriber's onError method.
            throw new RuntimeException(getObjectException);
          }
        });

    employeePublisher.subscribe(
      // This subscriber will close the ResultSet's Statement
      new Flow.Subscriber<Employee>() {
        public void onSubscribe(Flow.Subscription subscription) {
          subscription.request(Long.MAX_VALUE);
        }
        public void onNext(Employee item) { }
        public void onError(Throwable throwable) { closeStatement(); }
        public void onComplete() { closeStatement(); }
        void closeStatement() {
          try { resultSetStatement.close(); }
          catch (SQLException closeException) { log(closeException); }
        }
      });

    return employeePublisher;
  }
```

Instances of `OracleRow`, which the mapping function receives as input, becomes invalid after the function returns. Restricting the access of `OracleRow` to the scope of the mapping function enables the driver to efficiently manage the memory that is used to store row data. If you need a persistent copy of an `OracleRow`, then you can use the `OracleRow.clone` method to create a new instance of `OracleRow`, which is backed by a copy of the original `OracleRow` data. The `OracleRow` returned by the `clone` method remains valid outside the scope of the mapping function and retains its data even after the database connection is closed.

The row mapping function must return a non-null value or throw an unchecked exception. If the mapping function throws an unchecked exception, then it is delivered to row data subscribers as an `onError` signal. The row data Publisher supports multiple Subscribers. Row data emission to multiple Subscribers follow certain policies as stated below:

- A Subscriber will not receive an `onNext` signal for row data emitted before the Subscriber received an `onSubscribe` signal.
- A Subscriber will not receive an `onNext` signal until demand has been signaled by all other Subscribers.

The following table demonstrates the event flow while working with multiple subscribers:

Table 24-2 Emission to Multiple Subscribers

Time	Event	Cause
0	SubscriberA receives an `onSubscribe` signal	A call to the row data Publisher's subscribe (Subscriber) method requested a Subscription for SubscriberA
1	SubscriberA requests 1 row	SubscriberA signaled demand on its Subscription

Table 24-2 (Cont.) Emission to Multiple Subscribers

Time	Event	Cause
2	SubscriberA receives data for the first row of the ResultSets	The row data Publisher fetched a row of data that was requested by SubscriberA
3	SubscriberB receives an `onSubscribe` signal	A call to the row data Publisher's subscribe (Subscriber) method requested a Subscription for SubscriberB
4	SubscriberA requests 1 row	SubscriberA signaled demand on its Subscription
5	SubscriberB requests 1 row	SubscriberB signaled demand on its Subscription
6	Both SubscriberA and SubscriberB receive data for the second row of the ResultSet	The row data Publisher fetched a row of data that was requested by both Subscribers
7	SubscriberA requests 1 row	SubscriberA signaled demand on its Subscription
8	No row data is emitted	The row data Publisher does not emit the next row until *ALL* subscribers have requested it.
9	SubscriberB requests 1 row	SubscriberB signaled demand on its Subscription
10	Both SubscriberA and SubscriberB receive data for the third row of the ResultSet	The row data Publisher fetched a row of data that was requested by both Subscribers

Note:

SubscriberB never received data for the first row. This is because SubscriberB subscribed after the first row was emitted. Also, no data was emitted at 8 seconds. This is because all Subscribers need to request the next row before it is emitted. At 8 seconds, SubscriberA had requested for the next row, but SubscriberB had not placed a request till then.

24.2.4 Reading LOB Data Using Asynchronous Methods

The `OracleBlob`, `OracleBFile`, `OracleClob`, and `OracleNClob` interfaces expose a `publisherOracle(long)` method for asynchronous reading of LOB data.

The argument to the `publisherOracle(long)` method is a position of the LOB from where the data is read. The `OracleBlob.publisherOracle(long)` and `OracleBFile.publisherOracle(long)` methods return a Publisher<byte[]> type. This Publisher emits segments of binary data that have been read from the LOB. The `OracleClob.publisherOracle(long)` and `OracleNClob.publisherOracle(long)` methods return a Publisher<String> type. This Publisher emits segments of character data that have been read from the LOB.

The following example demonstrates how to asynchronously read binary data from a LOB.

```
/**
 * Asynchronously reads binary data from a BLOB
 * @param connection Connection to a database where the BLOB resides
 * @param employeeId ID associated to the BLOB
 * @return A Publisher that emits binary data of a BLOB
 * @throws SQLException If a database access error occurs before the
 * BLOB can be read
 */
Flow.Publisher<byte[]> readLOB(Connection connection, long employeeId)
  throws SQLException {
  PreparedStatement lobQueryStatement = connection.prepareStatement(
    "SELECT photo_bytes FROM employee_photos WHERE id = ?");
  lobQueryStatement.setLong(1, employeeId);

  ResultSet resultSet = lobQueryStatement.executeQuery();
  if (!resultSet.next())
    throw new SQLException("No photo found for employee ID " + employeeId);

  OracleBlob photoBlob =
    (OracleBlob)resultSet.unwrap(OracleResultSet.class).getBlob(1);
  Flow.Publisher<byte[]> photoPublisher = photoBlob.publisherOracle(1);

  photoPublisher.subscribe(
    // This subscriber will close the PreparedStatement and BLOB
    new Flow.Subscriber<byte[]>() {
      public void onSubscribe(Flow.Subscription subscription) {
        subscription.request(Long.MAX_VALUE);
      }
      public void onNext(byte[] item) { }
      public void onError(Throwable throwable) { freeResources(); }
      public void onComplete() { freeResources(); }
      void freeResources() {
        try { lobQueryStatement.close(); }
        catch (SQLException closeException) { log(closeException); }
        try { photoBlob.free(); }
        catch (SQLException freeException) { log(freeException); }
      }
    });
  return photoPublisher;
}
```

You cannot configure the size of data segments emitted by the LOB publishers. The driver chooses a segment size that is optimized as per the `DB_BLOCK_SIZE` parameter of the database.

24.2.5 Writing LOB Data Using Asynchronous Methods

The `OracleBlob`, `OracleClob`, and `OracleNClob` interfaces expose a `subscriberOracle(long)` method for asynchronous writing of LOB data.

The argument to the `subscriberOracle(long)` method is a position of the LOB where the data is written. The `OracleBlob.subscriberOracle(long)` method returns a `Subscriber<byte[]>` type. This Subscriber receives segments of binary data that are written to the LOB. The

`OracleClob.subscriberOracle(long)` method and the `OracleNClob.subscriberOracle(long)` method return a `Subscriber<String>` type. These Subscribers receive segments of character data that are written to the LOB.

The following examples demonstrate how to asynchronously write binary data to a LOB.

```
  /**
   * Asynchronously writes binary data to a BLOB
   * @param connection Connection to a database where the BLOB resides
   * @param bytesPublisher Publisher that emits binary data
   * @return A CompletionStage that completes with a reference to the BLOB,
   * after all binary data is written.
   * @throws SQLException If a database access error occurs before the table
data is
   * fetched
   */
  CompletionStage<Blob> writeLOB(
    Connection connection, Flow.Publisher<byte[]> bytesPublisher)
    throws SQLException {

    OracleBlob oracleBlob =
      (OracleBlob) connection.unwrap(OracleConnection.class).createBlob();

    // This future is completed after all bytes have been written to the BLOB
    CompletableFuture<Blob> writeFuture = new CompletableFuture<>();

    Flow.Subscriber<byte[]> blobSubscriber =
      oracleBlob.subscriberOracle(1L,
        // This Subscriber will receive a terminal signal when all byte[]'s
        // have been written to the BLOB.
        new Flow.Subscriber<Long>() {
          long totalWriteLength = 0;
          @Override
          public void onSubscribe(Flow.Subscription subscription) {
            subscription.request(Long.MAX_VALUE);
          }
          @Override
          public void onNext(Long writeLength) {
            totalWriteLength += writeLength;
            log(totalWriteLength + " bytes written.");
          }
          @Override
          public void onError(Throwable throwable) {
            writeFuture.completeExceptionally(throwable);
          }
          @Override
          public void onComplete() {
            writeFuture.complete(oracleBlob);
          }
        });

    bytesPublisher.subscribe(blobSubscriber);
    return writeFuture;
  }
```

The `OracleBlob`, `OracleClob`, and `OracleNClob` interfaces also expose a `subscriberOracle(long, Subscriber<Long>)` method that performs the same function as the single-argument form of the `subscriberOracle(long)` method. However, the single-argument form also accepts a `Subscriber<Long>` type. The `Subscriber<Long>` type notifies a Subscriber to receive the result of write operations against the database. Each time an asynchronous write operation completes, the `Subscriber<Long>` type receives an `onNext` signal with the number of bytes or number of characters written by the operation. If an asynchronous write operation fails, the `Subscriber<Long>` type receives an `onError` signal. After the final write operation completes, the `Subscriber<Long>` receives an `onComplete` signal.

After the `CompletionStage<Blob>` returned by the `writeLOB` method completes, the resulting Blob object can be passed to the `insertLOB` method to have the BLOB data stored in a table.

The following examples demonstrate how to insert the data.

```
/**
 * Asynchronously inserts BLOB data into a table by executing a DML SQL
 * statement
 * @param connection Connection to a database where the table data resides
 * @param employeeId ID related to the BLOB data
 * @param photoBlob Reference to BLOB data
 * @return A Publisher that emits the number of rows inserted (always 1)
 * @throws SQLException If a database access error occurs before the DML
 * SQL can be executed
 */
Flow.Publisher<Long> insertLOB(
  Connection connection, long employeeId, Blob photoBlob)
  throws SQLException {

  PreparedStatement lobInsertStatement = connection.prepareStatement(
    "INSERT INTO employee_photos(id, photo_bytes) VALUES (? ,?)");
  lobInsertStatement.setLong(1, employeeId);
  lobInsertStatement.setBlob(2, photoBlob);

  Flow.Publisher<Long> insertPublisher =
    lobInsertStatement.unwrap(OraclePreparedStatement.class)
      .executeUpdateAsyncOracle();

  insertPublisher.subscribe(new Flow.Subscriber<Long>() {
    @Override
    public void onSubscribe(Flow.Subscription subscription) {
      subscription.request(1L);
    }
    @Override
    public void onNext(Long item) { }
    @Override
    public void onError(Throwable throwable) { releaseResources(); }
    @Override
    public void onComplete() { releaseResources(); }
    void releaseResources() {
      try { lobInsertStatement.close(); }
      catch (SQLException closeException) { log(closeException); }
      try { photoBlob.free(); }
      catch (SQLException freeException) { log(freeException); }
    }
  });
```

```
    return insertPublisher;
  }
```

24.2.6 Committing a Transaction Using Asynchronous Methods

The `OracleConnection` interface exposes the `commitAsyncOracle` and `rollbackAsyncOracle` methods for asynchronous transaction completion.

Both the `commitAsyncOracle` and `rollbackAsyncOracle` methods return a `Flow.Publisher<Void>` type. The Publishers do not emit any item, as signified by their `<Void>` type. The Publishers emit a single `onComplete` or `onError` signal to indicate whether the commit or rollback operation was completed successfully or not.

The following example demonstrates how to asynchronously commit a transaction.

```
/**
 * Asynchronously commits a transaction
 * @param connection Connection to a database with an active transaction
 * @return A Publisher that emits a terminal signal when the transaction
 * has been committed
 * @throws SQLException If a database access error occurs before the
 * transaction can be committed
 */
public Flow.Publisher<Void> commitTransaction(Connection connection)
  throws SQLException {
  return connection.unwrap(OracleConnection.class)
    .commitAsyncOracle();
}
```

The `commitAsyncOracle` and `rollbackAsyncOracle` methods perform the same function as the `Connection.commit` and `Connection.rollback` methods.

24.2.7 Closing a Connection Using Asynchronous Methods

The `OracleConnection` interface exposes the `closeAsyncOracle` method for closing an asynchronous connection.

The `closeAsyncOracle` method returns a `Flow.Publisher<Void>` type. The Publisher does not emit any item, as signified by its `<Void>` type. The Publisher emits a single `onComplete` or `onError` signal to indicate whether the connection was closed successfully or not.

The following example demonstrates how to asynchronously close a connection.

```
/**
 * Asynchronously closes a connection
 * @param connection Connection to be closed
 * @return A Publisher that emits a terminal signal when the connection
 * has been closed
 * @throws SQLException If a database access error occurs before the
 * connection can be closed
 */
Flow.Publisher<Void> closeConnection(Connection connection)
  throws SQLException {
  return connection.unwrap(OracleConnection.class)
```

```
        .closeAsyncOracle();
  }
```

The `closeAsyncOracle` method performs the same function as the `Connection.close` method.

24.3 Threading Model of Asynchronous Methods

This section describes the threading model of asynchronous methods.

When an asynchronous method is called, it performs as much work as possible on the calling thread, without blocking on a network read. An asynchronous method call returns immediately after a request is written to the network, without waiting for a response. If the write buffer of the operating system is not large enough to store a complete request, then the calling thread may become blocked until this buffer is flushed.

Once the write job is complete, the network channel is registered for I/O readiness polling. A single thread polls the network channels of all Oracle JDBC Connections in the same JVM. The I/O polling thread is named `oracle.net.nt.TcpMultiplexer`, and is configured as a daemon thread. This polling thread performs a blocking operation using a Selector.

When I/O readiness is detected for a network channel, the polling thread arranges for a worker thread to handle the event. The worker thread reads from the network and then notifies a Publisher that an operation is complete. Upon notification, the Publisher arranges worker threads that emit a signal to each of its Subscribers.

The `java.util.concurrent.Executor` interface manages the worker threads. The default Executor is the `java.util.concurrent.ForkJoinPool.commonPool` method. You can call the `OracleConnectionBuilder.executorOracle(Executor)` method to specify an alternative Executor.

24.4 About the Flow API

The `java.util.concurrent.Flow` types define the minimal set of operations that you can use to create a reactive stream.

You can write application code directly against the Flow API. However, you must implement the low-level signal processing in accordance with the reactive-streams specification as specified in the following link

https://github.com/reactive-streams/reactive-streams-jvm/blob/master/README.md

The JDBC Reactive Extensions APIs handle the low-level mechanics of the Flow API to the application using `java.util.concurrent.Flow.Publisher` and `java.util.concurrent.Flow.Subscriber`. Popular libraries such as Reactor, RxJava, and Akka-Streams interface with the `org.reactivestreams.Publisher` and `org.reactivestreams.Subscriber` types defined by the `reactive-streams-jvm` project as specified in the following link:

https://github.com/reactive-streams/reactive-streams-jvm/tree/master/api/src/main/java/org/reactivestreams

Although the `org.reactivestreams.Publisher` type and the `org.reactivestreams.Subscriber` type, and their `java.util.concurrent.Flow` counterparts declare the same interface, the Java compiler must still consider them to be distinct types. You can convert between the Flow type and the `org.reactivestreams` type using the `org.reactivestreams.FlowAdapters` class as specified in the following link

https://github.com/reactive-streams/reactive-streams-jvm/tree/master/api/src/main/java9/org/reactivestreams

Oracle recommends that you use the reactive streams libraries when interfacing with the Flow types exposed by the JDBC driver.

24.5 Using the FlowAdapters Class

The libraries covered in this section interfaces with the `org.reactivestreams` types described in the **About the Flow API** section.

The following code snippet shows how the `FlowAdapters` class can convert the `Flow.Publisher` types into the `org.reactivestreams` types.

Example 24-1 Conversion to org.reactivestreams Types

```
public static org.reactivestreams.Publisher<ResultSet> publishQuery(
  PreparedStatement queryStatement) {
  try {
    Flow.Publisher<OracleResultSet> queryPublisher =
      queryStatement.unwrap(OraclePreparedStatement.class)
        .executeQueryAsyncOracle();

    return FlowAdapters.toPublisher(queryPublisher);
  }
  catch (SQLException sqlException) {
    return createErrorPublisher(sqlException);
  }
}

public static <T> org.reactivestreams.Publisher<T> publishRows(
  ResultSet resultSet, Function<OracleRow, T> rowMappingFunction) {
  try {
    Flow.Publisher<T> rowPublisher =
      resultSet.unwrap(OracleResultSet.class)
        .publisherOracle(rowMappingFunction);

    return FlowAdapters.toPublisher(rowPublisher);
  }
  catch (SQLException sqlException) {
    return createErrorPublisher(sqlException);
  }
}
```

24.6 Streaming Row Data with the Reactor Library

The Reactor library defines a *Flux* type that represents a stream of many items and a *Mono* type that represents a stream of just one item.

The following example shows how to create a *Flux* of row data using the Reactive Extensions.

```
    private Publisher<Employee> queryAllEmployees(Connection connection) {

    return Flux.using(
```

```
        // Prepare a SQL statement.
        () -> connection.prepareStatement("SELECT * FROM emp"),

        // Execute the PreparedStatement.
        preparedStatement ->
          // Create a Mono which emits one ResultSet.
          Mono.from(publishQuery(preparedStatement))
            // Flat map the ResultSet to a Flux which emits many Rows. Each
row
            // is mapped to an Employee object.
            .flatMapMany(resultSet ->
              publishRows(resultSet, row -> mapRowToEmployee(row))),

        // Close the PreparedStatement after emitting the last Employee object
        prepareStatement -> {
          try {
            prepareStatement.close();
          }
          catch (SQLException sqlException) {
            throw new RuntimeException(sqlException);
          }
        });
    }
```

The `using` factory method creates a *Flux* that depends on a resource that must be released explicitly. In this case, the resource is a `PreparedStatement` instance that is released by calling the `close` method.

The first lambda argument creates the `PreparedStatement` instance. This lambda is executed before the *Flux* begins to emit items.

The second lambda argument uses the `PreparedStatement` instance to create a stream of row data that is mapped to an **Employee** object. First, the `PreparedStatement` instance is executed by the call to the `publishQuery` method. As the query execution Publisher emits a single `ResultSet`, it is adapted into a *Mono*. Once the query execution completes, the *Mono* emits the `ResultSet` into the lambda specified by the `flatMapMany` method. This lambda calls

the `publishRows` method with a function to map `OracleRow` objects into the **Employee** objects. This results in the `flatMapMany` method call returning a `Flux` of **Employee** objects, where each **Employee** is mapped from a row of the `ResultSet` object.

The third lambda argument closes the `PreparedStatement` instance. This lambda is executed after the *Flux* emits its last item.

24.7 Streaming Row Data with the RxJava Library

The RxJava library defines a *Flowable* type that represents a stream of many items, and a *Single* type that represents a stream of just one item.

The following example shows how to create a *Flowable* of row data using the Reactive Extensions.

```
    private Publisher<Employee> queryAllEmployees(Connection connection) {

      return Flowable.using(

        // Prepare a SQL statement

        () -> connection.prepareStatement("SELECT * FROM emp"),


        // Execute the PreparedStatement

        queryStatement ->

          // Create a Single which emits one ResultSet

          Single.fromPublisher(publishQuery(queryStatement))

            // Flat map the ResultSet to a Flowable which emits many rows,
where

            // each row is mapped to an Employee object

            .flatMapPublisher(resultSet ->

              publishRows(resultSet, oracleRow ->
mapRowToEmployee(oracleRow))),


        // Close the PreparedStatement after emitting the last Employee object

        PreparedStatement::close

      );

    }
```

The `using` factory method creates a *Flowable* that depends on a resource that must be released explicitly. In this case, the resource is a `PreparedStatement` instance that is released by calling the `close` method.

The first lambda argument creates the `PreparedStatement` instance. This lambda is executed before the *Flowable* begins to emit items.

The second lambda argument uses the `PreparedStatement` instance to create a stream of row data that is mapped to an **Employee** object. First, the `PreparedStatement` instance is executed by the `publishQuery` method call. As the query execution Publisher emits a single `ResultSet` object, it is adapted into a *Single*. Once the query execution completes, the *Single* emits the `ResultSet` object into the lambda specified by the `flatMapPublisher` method. This lambda calls the `publishRows` method with a function to map `OracleRow` objects into **Employee** objects. This results in the `flatMapPublisher` method call returning a *Flowable* of **Employee** objects, where each **Employee** is mapped from a row of the `ResultSet`.

The third method handle argument closes the `PreparedStatement` instance. This lambda is executed after the *Flowable* emits its last item.

24.8 Streaming Row Data with the Akka Streams Library

The Akka Streams library defines a *Source* type that represents a stream of items.

The following example shows how to create a *Source* of row data using the Reactive Extensions.

```
    private Source<Employee, NotUsed> queryAllEmployees(Connection
connection) {


      final PreparedStatement queryStatement;

      try {

        queryStatement = connection.prepareStatement("SELECT * FROM emp");

      }

      catch (SQLException prepareStatementFailure) {

        return Source.failed(prepareStatementFailure);

      }


      // Create a Source which emits one ResultSet

      return Source.fromPublisher(publishQuery(queryStatement))

        // Flat map the ResultSet to a Source which emits many Rows, where
each

        // Row is mapped to an Employee object

        .flatMapConcat(resultSet -> {

          Publisher<Employee> employeePublisher =

            publishRows(resultSet, oracleRow -> mapRowToEmployee(oracleRow));
```

```
            return Source.fromPublisher(employeePublisher);

        })

        // This Sink closes the PreparedStatement when the Source terminates

        .alsoTo(Sink.onComplete(result -> queryStatement.close()));

    }
```

A `PreparedStatement` instance is used to create a stream of row data that is mapped to an **Employee** object. First, the `PreparedStatement` instance is executed by the `publishQuery` method call. The query execution Publisher emits a single `ResultSet` object. This Publisher is adapted into a *Source*. Once the query completes, the *Source* emits the `ResultSet` object into the lambda specified by the `flatMapConcat` method.

This lambda calls the `publishRows` method with a function to map `OracleRow` objects into **Employee** objects. This results in the `flatMapConcat` method call returning a *Source* of **Employee** objects, where each **Employee** is mapped from a row of the `ResultSet` object.

The `PreparedStatement` instance is a resource that needs to be explicitly closed. This is handled by the `alsoTo` method call that specifies a *Sink* that closes the `PreparedStatement` instance when the *Source* emits an `onComplete` or `onError` signal.

24.9 Limitations of JDBC Reactive Extensions

This section describes the limitations of JDBC Reactive Extensions.

JDBC Reactive Extensions have the following limitations:

- You must access the asynchronous methods through the `java.sql.Wrapper.unwrap` method, instead of accessing those through a type cast. This ensures correct behavior of the asynchronous methods when you use them with proxy Oracle JDBC classes, for example, when you use asynchronous methods with connection pools.
- Reading large responses from the network may require blocking I/O bound operations. Blocking read operations can occur if the driver reads a response that is larger than the TCP Receive buffer size.
- Asynchronous SQL execution supports neither scrollable `ResultSet` types nor sensitive `ResultSet` types.