3
Calling Java Methods in Oracle Database

This chapter provides an overview and examples of calling Java methods that reside in Oracle Database. It contains the following sections:

3.1 Invoking Java Methods

The type of the Java application determines how the client calls a Java method. The following sections discuss each of the Java application programming interfaces (APIs) available for calling a Java method:

3.1.1 Using PL/SQL Wrappers

You can run Java stored procedures in the same way as PL/SQL stored procedures. In Oracle Database, Java is usually invoked through a PL/SQL interface.

To call a Java stored procedure, you must publish it through a call specification. The following example shows how to create, resolve, load, and publish a simple Java stored procedure that returns a `String`:

1. Define a class, `Hello`, as follows:

```
public class Hello
{
  public static String world()
  {
    return "Hello world";
  }
}
```

   Save the file as a `Hello.java` file.

2. Compile the class on your client system using the standard Java compiler, as follows:

```
javac Hello.java
```

   It is a good idea to specify the `CLASSPATH` on the command line with the `javac` command, especially when writing shell scripts or make files. The Java compiler produces a Java binary file, in this case, `Hello.class`.

You must determine the location at which this Java code will run. If you run `Hello.class` on your client system, then it searches the `CLASSPATH` for all the supporting core classes that `Hello.class` needs for running. This search should result in locating the dependent classes in one of the following:

- As individual files in one or more directories, where the directories are specified in the `CLASSPATH`
- Within `.jar` or `.zip` files, where the directories containing these files are specified in the `CLASSPATH`

3. Decide on the resolver for the `Hello` class.

   In this case, load `Hello.class` on the server, where it is stored in the database as a Java schema object. When you call the `world()` method, Oracle JVM locates the necessary supporting classes, such as `String`, using a resolver. In this case, Oracle JVM uses the default resolver. The default resolver looks for these classes, first in the current schema, and then in `PUBLIC`. All core class libraries, including the `java.lang` package, are found in `PUBLIC`. You may need to specify different resolvers. You can trace problems earlier, rather than at run time, by forcing resolution to occur when you use the `loadjava` tool.

4. Load the class on the server using the `loadjava` tool. You must specify the user name and password. Run the `loadjava` tool as follows:

   ```
   loadjava -user HR Hello.class
   Password: password
   ```

5. Publish the stored procedure through a call specification.

   To call a Java `static` method with a SQL call, you must publish the method with a call specification. A call specification defines the arguments that the method takes and the SQL types that it returns.

   In SQL*Plus, connect to the database and define a top-level call specification for `Hello.world()` as follows:

   ```
   sqlplus HR
   Enter password: password
   connected
   SQL> CREATE OR REPLACE FUNCTION helloworld RETURN VARCHAR2 AS
     2  LANGUAGE JAVA NAME 'Hello.world () return java.lang.String';
     3  /
   Function created.
   ```

6. Call the stored procedure, as follows:

   ```
   SQL> VARIABLE myString VARCHAR2(20);
   SQL> CALL helloworld() INTO :myString;
   Call completed.
   SQL> PRINT myString;

   MYSTRING
   ----------------------------------------
   Hello world

   SQL>
   ```

   The `call helloworld() into :myString` statement performs a top-level call in Oracle Database. SQL and PL/SQL see no difference between a stored procedure that is written in Java, PL/SQL, or any other language. The call specification provides a means to tie inter-language calls together in a consistent manner. Call specifications are necessary only for entry points that are called with triggers or SQL and PL/SQL calls. Furthermore, JDeveloper can automate the task of writing call specifications.

Related Topics

- Overview of Resolving Class Dependencies
  Many Java classes contain references to other classes, which is the essence of reusing code. A conventional JVM searches for `.class`, `.zip`, and `.jar` files within the directories specified in `CLASSPATH`.
- Schema Objects and Oracle JVM Utilities
- Developing Java Stored Procedures

3.1.1.1 Using PL/SQL Wrappers with JDK 11

Beginning with Oracle Database Release 23ai, The Java database object names can also contain a module component.

Java objects that are the members of a module, are stored in database objects with names in the following format:

```
<module_name>///<class_source_or_resource_name>
```

If a Java database object is not part of a module, that is, if it is part of the *unnamed module*, then the format is <*class_source_or_resource_name*>, as it was in the earlier database releases. In the `class_source_or_resource` portion of the name, the package delimiter period (`.`) is replaced by a forward slash (`/`) in the database object name, as in the earlier database releases.

No character replacement occurs in the module portion of the name. The fully modularized form of the database object class name is specified as the name of the top-level class, whose method is being called. For example, a call specification to call the method `world` in the class named `hello.Hello` in the module named `hello.in.there` can be the following:

```
CREATE OR REPLACE FUNCTION helloworld RETURN VARCHAR2 AS LANGUAGE JAVA NAME
'hello.in.there///hello.Hello.world () return java.lang.String';
```

As in the earlier database releases, class names with either period (`.`) or forward slash (`/`) delimiters are both accepted in the `class_name` portion of the database object name of the stored procedure being called. In the argument and return value portion of call specifications, module names are *not* specified, even if the argument or return value classes are members of a module.

All JDK and Oracle JVM system classes are themselves contained in modules in JDK11. Exceptionally, if the class name of a Java stored procedure to be invoked is one of the built-in system classes, then in the stored procedure definition, you can specify either the fully modularized database object name or just the class name portion as the class of the method to be called.

3.1.2 About JNI Support

The Java Native Interface (JNI) is a standard programming interface for writing Java native methods and embedding the JVM into native applications. The primary goal of JNI is to provide binary compatibility of Java applications that use platform-specific native libraries.

Native methods can cause server failure, violate security, and corrupt data. Oracle Database does not support the use of JNI in Java applications. If you use JNI, then your application is not 100 percent pure Java and the native methods require porting between platforms.

3.1.3 About Utilizing JDBC with Java in the Database

You can use Java Database Connectivity (JDBC) APIs from a Java client. These APIs establish a session with a given user name and password on the database, and run SQL queries against the database. All Oracle JDBC drivers communicate seamlessly with Oracle SQL and PL/SQL.

3.1.3.1 Using JDBC

JDBC is an industry-standard API that lets you embed SQL statements as Java method arguments. JDBC is based on the X/Open SQL Call Level Interface (CLI) and complies with the Entry Level of SQL-92 standard. Each vendor, such as Oracle, creates its JDBC implementation by implementing the interfaces of the standard `java.sql` package. Oracle provides the following JDBC drivers that implement these standard interfaces:

- The JDBC Thin driver, a 100 percent pure Java solution that you can use for either client-side applications or applets and requires no Oracle client installation.
- The JDBC OCI driver, which you use for client-side applications and requires an Oracle client installation.
- The server-side JDBC driver embedded in Oracle Database.

Using JDBC is a step-by-step process of performing the following tasks:

1. Obtaining a connection handle
2. Creating a statement object of some type for your desired SQL operation
3. Assigning any local variables that you want to bind to the SQL operation
4. Carrying out the operation
5. Optionally retrieving the result sets

This process is sufficient for many applications, but becomes cumbersome for any complicated statements. Dynamic SQL operations, where the operations are not known until run time, require JDBC. However, in typical applications, this represents a minority of the SQL operations.

> **See Also:**
>
> *Oracle Database JDBC Developer's Guide*

3.1.4 About Using the Command-Line Interface

The command-line interface to Oracle JVM is analogous to using the JDK or JRE shell commands. You can:

- Use the standard `-classpath` syntax to indicate where to find the classes to load
- Set the system properties by using the standard `-D` syntax

The interface is a PL/SQL function that takes a string (`VARCHAR2`) argument, parses it as a command-line input and if it is properly formed, runs the indicated Java method in Oracle JVM. To do this, PL/SQL package `DBMS_JAVA` provides the following functions:

- runjava
- runjava_in_current_session

runjava

This function takes the Java command line as its only argument and runs it in Oracle JVM. The return value is null on successful completion, otherwise an error message. The format of the command line is the same as that taken by the JDK shell command, that is:

```
[option switches] name_of_class_to_execute [arg1 arg2 ... argn]
```

You can use the option switches `-classpath`, `-D`, `-Xbootclasspath`, and `-jar`. This function differs from the `runjava_in_current_session` function in that it clears any Java state remaining from previous use of Java in the session, prior to running the current command. This is necessary, in particular, to guarantee that static variable values derived at class initialization time from `-classpath` and `-D` arguments reflect the values of those switches in the current command line.

```
FUNCTION runjava(cmdline VARCHAR2) RETURN VARCHAR2;
```

runjava_in_current_session

This function is the same as the `runjava` function, except that it does not clear Java state remaining from previous use of Java in the session, prior to executing the current command line.

```
FUNCTION runjava_in_current_session(cmdline VARCHAR2) RETURN VARCHAR2;
```

Syntax

The syntax of the command line is of the following form:

```
[-options] classname [arguments...]
[-options] -jar jarfile [arguments...]
```

Options

```
-classpath
-D
-Xbootclasspath
-Xbootclasspath/a
-Xbootclasspath/p
-cp
```

> **Note:**
>
> The effect of the first form is to run the main method of the class identified by classname with the arguments. The effect of the second form is to run the main method of the class identified by the `Main-Class` attribute in the manifest of the JAR file identified by JAR. This is analogous to how the JDK/JRE interprets this syntax.

Argument Summary

The following table summarizes the command-line arguments.

Table 3-1 Command Line Argument Summary

Argument	Description
classpath	Accepts a colon (:) separated (semicolon-separated on Windows systems) list of directories, JAR archives, and ZIP archives to search for class files. In general, the value of `-classpath` or similar arguments refer to file system locations as they would in a standard Java runtime. You also have an extension to this syntax to allow for terms that refer to database resident Java objects and sets of bytes.
D	Establishes values for system properties when there is no existing Java session state. The default behavior of the command-line interface, that is, the `runjava` function, is to terminate any existing Java session prior to running the new command. On the other hand, the alternative function, `runjava_in_current_session` leaves any existing session in place. So, values established with the `-D` option always take effect when `runjava` function is used, but the values may not take effect when `runjava_in_current_session` function is used.
Xbootclasspath	Accepts a colon (:) separated (semicolon-separated on Windows systems) list of directories, JAR archives, and ZIP archives. This option is used to set search path for bootstrap classes and resources.
`Xbootclasspath/a`	Accepts a colon (:) separated (semicolon-separated on Windows systems) list of directories, JAR archives, and ZIP archives. This is appended to the end of bootstrap class path.
`Xbootclasspath/p`	Accepts a colon (:) separated (semicolon-separated on Windows systems) list of directories, JAR archives, and ZIP archives. This is added in front of bootstrap class path.
`cp`	Acts as a synonym of `-classpath`.

> **Note:**
>
> System classes created by `create java system` are always used before using any file or folder that are found using the `-Xbootclasspath` option.

Related Topics

- About Using the Command-Line Interface

3.1.5 Overview of Using the Client-Side Stub

Oracle Database 10*g* introduced the client-side stub, formerly known as native Java interface, for calls to server-side Java code. It is a simplified application integration. Client-side and middle-tier Java applications can directly call Java in the database without defining a PL/SQL wrapper. The client-side stub uses the server-side Java class reflection capability.

In previous releases, calling Java stored procedures and functions from a database client required Java Database Connectivity (JDBC) calls to the associated PL/SQL wrappers. Each wrapper had to be manually published with a SQL signature and a Java implementation. This had the following disadvantages:

- The signatures permitted only Java types that had direct SQL equivalents
- Exceptions issued in Java were not properly returned

Starting from Oracle Database 12*c* Release 2 (12.2.0.1), you can use the Oracle JVM Web Services Call-Out Utility for generating the client-side stub.

Related Topics

- Architecture of Oracle JVM Web Services Call-Out Utility
  The Oracle JVM Web Services Call-Out utility consists of two phases: Client Stub Generation and Oracle JVM-Specific Artifact Generation.

3.1.5.1 Using the Default Service Feature

If you install Oracle Database client, then you need not specify all the details of the database server in the connection URL. Under certain conditions, Oracle Database connection adapter requires only the host name of the computer where the database is installed.

For example, in the JDBC connection URL syntax, that is:

```
jdbc:oracle:driver_type:[username/password]@[//]host_name[:port][:ORCL]
```

,the following have become optional:

- `//` is optional.
- `:port` is optional.

  You must specify a port only if the default Oracle Net listener port (1521) is not used.
- `:ORCL` or the service name is optional.

  The connection adapter for Oracle Database Client connects to the default service on the host. On the host, this is set to `ORCL` in the `listener.ora` file.

3.1.5.2 Testing the Default Service with a Basic Configuration

The following code snippet shows a basic configuration of the `listener.ora` file, where the default service is defined:

```
MYLISTENER = (ADDRESS_LIST=(ADDRESS=(PROTOCOL=tcp)(HOST=testserver1)(PORT=1521)))
DEFAULT_SERVICE_MYLISTENER=dbjf.app.myserver.com
SID_LIST_MYLISTENER = (SID_LIST=(SID_DESC=(SID_NAME=dbjf)
(GLOBAL_DBNAME=dbjf.app.myserver.com)(ORACLE_HOME=/test/oracle))
)
```

After defining the `listener.ora` file, restart the listener with the following command:

```
lsnrctl start mylistener
```

Now, any of the following URLs should work with this configuration of the `listener.ora` file:

- `jdbc:oracle:thin:@//testserver1.myserver.com.com`
- `jdbc:oracle:thin:@//testserver1.myserver.com:1521`
- `jdbc:oracle:thin:@testserver1.myserver.com`
- `jdbc:oracle:thin:@testserver1.myserver.com:1521`
- `jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=testserver1.myserver.com)(PORT=1521)))`
- `jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=testserver1.myserver.com)))`

- `jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=testserver1.myserver.com)(PORT=1521))(CONNECT_DATA=(SERVICE_NAME=)))`

3.2 How To Tell If You Are Running on the Server

You may want to write Java code that runs in a certain way on the server and in another way on the client. In general, Oracle does not recommend this. In fact, JDBC enable you to write portable code that avoids this problem, even though the drivers used in the server and client are different.

If you want to determine if your code is running on the server, then you can use the `System.getProperty ("oracle.jserver.version")` method.

The `getProperty()` method returns the following information:

- A `String` that represents Oracle Database release, if running on the server
- `null`, if running on the client

The following code snippet shows how to determine if you are running your code on the server:

```
{
...
/*
  * This code snippet checks if you are
  * running your code on the server
  */
        if (System.getProperty("oracle.jserver.version") != null)
        {
            /*
               * Java running in the database, already connected, use the
default
               * connection URL is ignored, assumed null
            */

          ods.setURL("jdbc:default:connection:");
        }
        else
        {
           /*
                  * Java not running in the database, you need a client URL
          */
    }
...
}
```

3.3 About Redirecting Output on the Server

You can pass Java output to SQL statements to provide more extensive control over the destination of output from Oracle JVM. Use the following APIs available in the `DBMS_JAVA` PL/SQL package to achieve this:

- set_output_to_sql
- remove_output_to_sql
- enable_output_to_sql

- disable_output_to_sql
- query_output_to_sql

set_output_to_sql

`set_output_to_sql` defines a named output specification that constitutes an instruction for executing a SQL statement, whenever output to the default `System.out` and `System.err` streams occurs. The specification is defined either for the duration of the current session, or till the `remove_output_to_sql` function is called with its ID. The SQL actions prescribed by the specification occur whenever there is Java output. This can be stopped and started by calling the `disable_output_to_sql` and `enable_output_to_sql` functions respectively. The return value of this function is null on success, otherwise an error message.

```
FUNCTION set_output_to_sql (id VARCHAR2,
stmt VARCHAR2,
bindings VARCHAR2,
no_newline_stmt VARCHAR2 default null,
no_newline_bindings VARCHAR2 default null,
newline_only_stmt VARCHAR2 default null,
newline_only_bindings VARCHAR2 default null,
maximum_line_segment_length NUMBER default 0,
allow_replace NUMBER default 1,
from_stdout NUMBER default 1,
from_stderr NUMBER default 1,
include_newlines NUMBER default 0,
eager NUMBER default 0) return VARCHAR2;
```

Table 3-2 describes the arguments the `set_output_to_sql` function takes.

Table 3-2 set_output_to_sql Argument Summary

Argument	Description
`id`	The name of the specification. Multiple specifications may exist in the same session, but each must have a distinct ID. The ID is used to identify the specification in the functions `remove_output_to_sql`, `enable_output_to_sql`, `disable_output_to_sql`, and `query_output_to_sql`.
`stmt`	The default SQL statement to execute when Java output occurs.
`bindings`	A string containing tokens from the set *ID, TEXT, LENGTH, LINENO, SEGNO, NL,* and *ERROUT.* This string defines how the SQL statement `stmt` is bound. The position of a token in the bindings string corresponds to the bind position in the SQL statement. The meanings of the tokens are: • ID is the ID of the specification. It is bound as a VARCHAR2. • TEXT is the text being output. It is bound as a VARCHAR2. • LENGTH is the length of the text. It is bound as a NUMBER. • LINENO is the line number since the beginning of session output. It is bound as a NUMBER. • SEGNO is the segment number within a line that is being output in more than one piece. It is bound as a NUMBER. • NL is a boolean indicating whether the text is to be regarded as newline terminated. It is bound as a NUMBER. The newline may or may not actually be included in the text, depending on the value of the `include_newlines` argument. • ERROUT is a boolean indicating whether the output came from `System.out` or `System.err`. It is bound as a NUMBER. The value is 0, if the output came from `System.out`.

Table 3-2 (Cont.) set_output_to_sql Argument Summary

Argument	Description
no_newline_stmt	An optional alternate SQL statement to execute, when the output is not newline terminated.
no_newline_bindings	A string with the same syntax as for the bindings argument discussed previously, describing how the no_newline_stmt is bound.
newline_only_stmt	An optional alternate SQL statement to execute when the output is a single newline.
newline_only_bindings	A string with the same syntax as for the bindings argument discussed previously, describing how the newline_only_stmt is bound.
maximum_line_segment_length	The maximum number of characters that is bound in a given execution of the SQL statement. Longer output sequences are broken up into separate calls with distinct SEGNO values. A value of 0 means no maximum.
allow_replace	Controls behavior when a previously defined specification with the same ID exists. A value of 1 means replacing the old specification. 0 means returning an error message without modifying the old specification.
from_stdout	Controls whether output from System.out causes execution of the SQL statement prescribed by the specification. A value of 0 means that if the output came from System.out, then the statement is not executed, even if the specification is otherwise enabled.
from_stderr	Controls whether output from System.err causes execution of the SQL statement prescribed by the specification. A value of 0 means that if the output came from System.err, then the statement is not executed, even if the specification is otherwise enabled.
include_newlines	Controls whether newline characters are left in the output when they are bound to text. A value of 0 means new lines are not included. But the presence of the newline is still indicated by the NL binding and the use of no_newline_stmt.
eager	Controls whether output not terminated by a newline causes execution of the SQL statement every time it is received, or accumulates such output until a newline is received. A value of 0 means that unterminated output is accumulated.

remove_output_to_sql

remove_output_to_sql deletes a specification created by set_output_to_sql. If no such specification exists, an error message is returned.

```
FUNCTION remove_output_to_sql (id VARCHAR2) return VARCHAR2;
```

enable_output_to_sql

enable_output_to_sql reenables a specification created by set_output_to_sql and subsequently disabled by disable_output_to_sql. If no such specification exists, an error message is returned. If the specification is not currently disabled, there is no change.

```
FUNCTION enable_output_to_sql (id VARCHAR2) return VARCHAR2;
```

disable_output_to_sql

disable_output_to_sql disables a specification created by set_output_to_sql. You can enable the specification by calling enable_output_to_sql. While disabled, the SQL statement

prescribed by the specification is not executed. If no such specification exists, an error message is returned. If the specification is already disabled, there is no change.

```
FUNCTION disable_output_to_sql (id VARCHAR2) return VARCHAR2;
```

query_output_to_sql

`query_output_to_sql` returns a message describing a specification created by `set_output_to_sql`. If no such specification exists, then an error message is returned. Passing `null` to this function causes all existing specifications to be displayed.

```
FUNCTION query_output_to_sql (id VARCHAR2) return VARCHAR2;
```

Another way of achieving control over the destination of output from Oracle JVM is to pass your Java output to an autonomous Java session. This provides a very general mechanism for propagating the output to various kinds of targets, such as disk files, sockets, and URLS. But, you must keep in mind that the Java session that processes the output is logically distinct from the main session, so that there are no other, unwanted interactions between them. To do this, PL/SQL package `DBMS_JAVA` provides the following functions:

- set_output_to_java
- remove_output_to_java
- enable_output_to_java
- disable_output_to_java
- query_output_to_java
- set_output_to_file
- remove_output_to_file
- enable_output_to_file
- disable_output_to_file
- query_output_to_file

set_output_to_java

`set_output_to_java` defines a named output specification that gives an instruction for executing a Java method whenever output to the default `System.out` and `System.err` streams occurs. The Java method prescribed by the specification is executed in a separate VM context with separate Java session state from the rest of the session.

```
FUNCTION set_output_to_java (id VARCHAR2,
class_name VARCHAR2,
class_schema VARCHAR2,
method VARCHAR2,
bindings VARCHAR2,
no_newline_method VARCHAR2 default null,
no_newline_bindings VARCHAR2 default null,
newline_only_method VARCHAR2 default null,
newline_only_bindings VARCHAR2 default null,
maximum_line_segment_length NUMBER default 0,
allow_replace NUMBER default 1,
from_stdout NUMBER default 1,
from_stderr NUMBER default 1,
include_newlines NUMBER default 0,
eager NUMBER default 0,
initialization_statement VARCHAR2 default null,
finalization_statement VARCHAR2 default null)return VARCHAR2;
```

Table 3-3 describes the arguments the `set_output_to_java` method takes.

Table 3-3 set_output_to_java Argument Summary

Argument	Description
`class_name`	The name of the class defining one or more methods.
`class_schema`	The schema in which the class is defined. A null value means the class is defined in the current schema, or PUBLIC.
`method`	The name of the method.
`bindings`	A string that defines how the arguments to the method are bound. This is a string of tokens with the same syntax as `set_output_to_sql`. The position of a token in the string determines the position of the argument it describes. All arguments must be of type INT, except for those corresponding to the tokens ID or TEXT, which must be of type `java.lang.String`.
`no_newline_method`	An optional alternate method to execute when the output is not newline terminated.
`newline_only_method`	An optional alternate method to execute when the output is a single newline.
`initialization_statement`	An optional SQL statement that is executed once per Java session prior to the first time the methods that receive output are executed. This statement is executed in same Java VM context as the output methods are executed. Typically such a statement is used to run a Java stored procedure that initializes conditions in the separate VM context so that the methods that receive output can function as intended. For example, such a procedure might open a stream that the output methods write to.
`finalization_statement`	An optional SQL statement that is executed once when the output specification is about to be removed or the session is ending. Like the `initialization_statement`, this runs in the same JVM context as the methods that receive output. It runs only if the initialization method has run, or if there is no initialization method.

remove_output_to_java

`remove_output_to_java` deletes a specification created by `set_output_to_java`. If no such specification exists, an error message is returned

```
FUNCTION remove_output_to_java (id VARCHAR2) return VARCHAR2;
```

enable_output_to_java

`enable_output_to_java` reenables a specification created by `set_output_to_java` and subsequently disabled by `disable_output_to_java`. If no such specification exists, an error message is returned. If the specification is not currently disabled, there is no change.

```
FUNCTION enable_output_to_java (id VARCHAR2) return VARCHAR2;
```

disable_output_to_java

`disable_output_to_java` disables a specification created by `set_output_to_java`. The specification may be re-enabled by `enable_output_to_java`. While disabled, the SQL statement prescribed by the specification is not executed. If no such specification exists, an error message is returned. If the specification is already disabled, there is no change.

```
FUNCTION disable_output_to_java (id VARCHAR2) return VARCHAR2;
```

query_output_to_java

`query_output_to_java` returns a message describing a specification created by `set_output_to_java`. If no such specification exists, an error message is returned. Passing `null` to this function causes all existing specifications to be displayed.

```
FUNCTION query_output_to_java (id VARCHAR2) return VARCHAR2;
```

set_output_to_file

`set_output_to_file` defines a named output specification that constitutes an instruction to capture any output sent to the default `System.out` and `System.err` streams and append it to a specified file. This is implemented using a special case of `set_output_to_java`. The argument `file_path` specifies the path to the file to which to append the output. The arguments `allow_replace`, `from_stdout`, and `from_stderr` are all analogous to the arguments having the same name as in `set_output_to_sql`.

```
FUNCTION set_output_to_file (id VARCHAR2,
file_path VARCHAR2,
allow_replace NUMBER default 1,
from_stdout NUMBER default 1,
from_stderr NUMBER default 1) return VARCHAR2;
```

remove_output_to_file

This function is analogous to `remove_output_to_java`.

```
FUNCTION remove_output_to_file (id VARCHAR2) return VARCHAR2;
```

enable_output_to_file

This function is analogous to `enable_output_to_java`.

```
FUNCTION enable_output_to_file (id VARCHAR2) return VARCHAR2;
```

disable_output_to_file

This function is analogous to `disable_output_to_java`.

```
FUNCTION disable_output_to_file (id VARCHAR2) return VARCHAR2;
```

query_output_to_file

This function is analogous to `query_output_to_java`.

```
FUNCTION query_output_to_file (id VARCHAR2) return VARCHAR2;
```

The following `DBMS_JAVA` functions control whether Java output appears in the `.trc` file:

- `PROCEDURE enable_output_to_trc;`
- `PROCEDURE disable_output_to_trc;`
- `FUNCTION query_output_to_trc return VARCHAR2;`

Redirecting the output to SQL*Plus Text Buffer

You can use the `DBMS_JAVA` package procedure `SET_OUTPUT` to redirect output to the SQL*Plus text buffer:

```
SQL> SET SERVEROUTPUT ON
SQL> CALL dbms_java.set_output(2000);
```

The minimum and default buffer size is 2,000 bytes and the maximum size is 1,000,000 bytes. In the following example, the buffer size is increased to 5,000 bytes:

```
SQL> SET SERVEROUTPUT ON SIZE 5000
SQL> CALL dbms_java.set_output(5000);
```

The output is displayed at the end of the call.