Java Developers’ Perspective on Oracle Database 11g
Introduction

The Oracle database furnishes the following technologies for Java developers: JDBC drivers, embedded Java runtime in the database (a.k.a. OracleJVM), SQLJ pre-compiler and runtime, and the JPublisher utility. What's in Oracle Database 11g Release 1 and Release 2 for Java developers? Are you looking at application portability, ease-of-development (productivity), performance, security, reliability or ease of deployment? Focusing only on JDBC and the embedded Java runtime, this paper will first highlight Oracle database support for Java standards, then describe database features that bring development-time productivity, security, and performance to Java developer; finally the paper will describe database features that bring deployment-time performance, scalability, highly-availability and manageability to Java applications.

Portability: Java Standards Support

Application portability across platforms and vendors is the primary motivation for using Java. Oracle Database 11g Release 1 and Release 2 support Java standards primarily through JDBC and the Java runtime in the database.

JDBC 4.0 Support

The Oracle database 11g JDBC comes in two flavors: ojdbc5.jar for Java 5 (i.e., JDK 1.5) and ojdbc6.jar for Java 6 (i.e., JDK 1.6). The ojdbc6.jar supports the new JDBC 4.0 specification (dependency on JDK 1.6). Since the specification is publicly available¹, a complete description of JDBC 4.0 is beyond the scope of this paper however, here are the key features:

- Connection and Statement Enhancements
  The following enhancements are made to Connection and Statement interfaces:
  
  a) Validate a connection
     boolean isValid (int timeout): this method returns TRUE if the connection has not been closed and is still valid.
  
  b) Set/Get user-defined information and properties using SetClientInfo() and GetClientInfo() methods
     void setClientInfo(java.lang.String name, java.lang.String value)
     java.lang.String getClientInfo(java.lang.String name)
     java.util.Properties getClientInfoProperties()

  ¹ http://jcp.org/aboutJava/communityprocess/final/jsr221/index.html
c) Setting a statement poolable
   void setPoolable(boolean poolable)

d) Check whether a Statement object is poolable or not
   boolean isPoolable()

e) New Statement interface and class
   StatementEventListener interface on PooledConnection; and StatementEvent class

• Wrapper Interface
The Wrapper interface allows accessing vendor extensions to the following JDBC interfaces:
   java.sql.Connection, java.sql.DatabaseMetaData, java.sql.ParameterMetaData, java.sql.ResultSet,

The Wrapper interface defines an `unwrap()` method which takes an interface as an argument and
returns an object (or a proxy for the object) that implements the given interface. The returned
object is used to access the methods not exposed by the proxy.

The `isWrapperFor()` method returns true if the calling object implements the interface argument.
If `isWrapperFor()` returns true then calling `unwrap()` with the same argument should succeed.

• New Standard Datatypes
   java.sql.RowId: a standard Java type for mapping SQL ROWID Methods:
   java.sql.RowId xrowId = rs.getRowId(1);
   ps.setRowId(1, xrowId);

   java.sql.SQLXML2: a standard and driver-independant Java type for mapping of SQL XMLType
datatype methods
   SQLXML createXML() throws SQLException;
   public XMLStreamReader getXMLStreamReader();
   public XMLStreamWriter putXMLStreamWriter();
   public void setString(String text)

• SQL 2003 National Character Set types including:
   NCHAR, NCLOB: say goodbye to setFormOfUse;
   NVARCHAR and LONGNVARCHAR.

• LOB Enhancements
   The Connection interface has been enhanced with `createBlob`, `createClob`, and
   `createNClob` methods for creating empty Blob, Clob, and NClob objects:
   Blob aBlob = con.createBlob();
   The newly created Blob, aBlob contains no data; the `setXXX()` methods can be invoked on
   the CLOB interface to populate it:

   2 Currently not yet available
int numWritten = aBlob.setBytes(1, val);

Conversely, getBlob(), getClob(), getNClob() methods can be used for retrieving the LOB value from a ResultSet:
Blob blob = rs.getBlob(1);

• Exception Hierarchy
  JDBC 4.0 furnishes a complete rework of exception framework with:
  a) Support for Java SE For-each Loops: SQLException now implements java.langIterable<Throwable> allowing easy navigation of SQLExceptions
  b) Support for Java SE Chained Exceptions: SQLExceptions may have one or more causal relationships
  c) New SQLException Subclasses: SQLTransientException, and SQLNonTransientException

• RowSet (JSR-114) Enhancements
  JDBC 4.0 brings minor enhancements to the RowSet API such as getters and setters for NChar and RowId

OracleJVM Compliance with Java 5
The embedded Java VM in the database (a.k.a. OracleJVM) allows running Java SE applications and libraries directly in the Oracle database resulting in significant productivity and performance gains. Oracle database 11g releases support Java SE 5 (i.e., compatibility with JDK 1.5) and in addition, the RowSet (JSR-114) in server-side JDBC, and JMX for managing the session JVM.

Generic Database Features for Application Development

Beyond Java standards, Oracle Database 11g Release 1 and Release 2 expose generic database capabilities aiming at improving Java applications development. Features available through JDBC include: SYS.ANYDATA and SYS.ANYTYPE data types, LOB prefetching, SecureFiles LOBs, Client & Server Query Result cache, Query change notification. Features available through the embedded Java runtime include: like-JDK interface, output redirect, property interface, database resident JAR, and two-tier duration of Java session state.

SYS.ANYDATA and SYS.ANYTYPE
The SQL SYS.ANYDATA is a polymorph data type which, at a given time, contains an instance of a specific type description along with data. Over time, a SYS: ANYDATA column may contain any SQL type (i.e., VARCHAR, a NUMBER, a TIMESTAMP, FLOAT, etc). Furthermore, it can be persisted.
The SYS.ANYTYPE SQL type contains the type description of any SQL type (including objects types and collections) that can be transient or persistent, named or unnamed. It maps to oracle.sql.TypeDescriptor class.

Client Query Result Cache
Oracle Database 11g Release 2 furnish Table Annotation for caching query results sets in client memory (JDBC-OCI driver only) and synchronized with database changes as if the query was being continuously executed.

```
ALTER TABLE sales RESULT_CACHE (MODE FORCE);
```

When a table is annotated as cache worthy all queries run against the table become candidates for caching in the client/server cache. This eliminates the need for changing the application to add query level hints.

Retrieval of results set locally from the client memory is much faster than re-processing the query, and retrieving results from the disk; frequently executed queries experience a significant performance improvement when their results are cached. Internal testing of a standard benchmark shows response time improvements of up to 22% and a 6 times reduction in server CPU consumption.

Oracle Database 11g Release 1 only furnishes Client Query Result Cache with query annotation i.e., using the `/* + result_cache */` hint.

```
select /*+ result_cache */ * from employees
```

Server Query Result Cache

Results of queries can also be cached on the database side (within the shared pool). Users can annotate a query with the `/* + result_cache */` hint as in CQRC (see previous feature above) to indicate that result sets are to be cached.

Being on the database side, Server Query Result Cache is available to all database users irrespective of the language ad API however, unlike CQRC, it requires a database roundtrip.

The `RESULT_CACHE_MODE` initialization parameter controls whether the server query result cache is used for all queries (when possible), or only for annotated queries.

Prefetch in 1st Roundtrip – Saving Millions of Database Roundtrips per Day

For ad-hoc SQL queries, the new pre-fetch mechanism combines `parse, execute, and fetch` to retrieves the first batch of the results set in the first interaction with the database. Internal reliable testing shows 50% reduction in network roundtrip; 50% reduction in response time; 30% reduction of server and client CPU consumption. To put this reduction in perspective, a typical web retail application issuing 1 million ad-hoc queries per day (or a determined period of time) will see a reduction of 1 million roundtrips during the same period.

LOB Prefetching

To improve query performance of smaller LOBs, LOB data and length can be prefetched and cached while also fetching the locator.

SecureFiles LOB

SecureFiles (LOBs with STORE as SECUREFILE option), reduce the storage requirements for LOB data and increase performance in accessing it. Multiple SQL parameters control SecureFiles LOBS.
• DEDUPLICATE in CREATE TABLE and ALTER TABLE statements enables the application to specify that LOB data that are identical in two or more rows in LOB column will all share the same data block, thus saving disk space.

• COMPRESS turns on LOB compression.

• ENCRYPT turns on LOB encryption and optionally selects an encryption algorithm.

Each SecureFiles LOB column can have its own encryption specification, independent of the encryption of other columns (LOB or non-LOB).

Oracle JDBC supports SecureFiles LOB though new APIs including:

```java
oracle.sql.{BlobRegion/ClobRegion}
java.sql.{BlobShare/ClobShare}
oracle.sql.{setOptions/getOptions}
oracle.sql.{BLOB/CLOB}.{setSharedRegions/getSharedRegions}
oracle.sql.{CLOB/BLOB}.{fragment}{Insert/Delete/Move/Replace}
oracle.sql.{CLOB/BLOB}.{open/copy/getChunkSize}
```

Zero-Copy SecureFiles LOB

Zero-copy consists in bypassing the copy of LOB data into internal buffer on the server-side; as a result SecureFiles LOB operations are faster.

```java
setSecureFile(); Blob.getBytes();
```

Net Services: IPv6 Standard Support

IPv6 is the next generation IP protocol, designed to replace IPv4. As of Oracle Database 11g Release 2, Oracle furnishes a complete IPv6 support for single-instance databases running on dual-stack or IPv6-only host. The IP session layer has been abstracted to enable the listener to support IPv6 and IPv4 interfaces during the transition phases. A JDBC URL using IPv6 address will look like the following:

```java
jdbc:oracle:thin:@\(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
(CONNECT_DATA=(SERVICE_NAME=sales.acme.com))
```

Net Services: Fast Connect Time Failover and Retry

Faster failover can be achieved at connection establishment time by the use of new Net Services parameters. Rather than waiting for system level TCP timeouts, connect level timeouts can be set in seconds based on network latency.

```java
TCP.CONNECT_TIMEOUT enabled by default (60 sec.)
```

One can also set retry count for transparent retries at connection establishment time.

```java
RETRY_COUNT (default is 0)
```

See Net Services documentation for more details.
Native AQ Protocol
A new native AQ protocol (as opposed to a PL/SQL based interface) improves the performance of JMS-over-AQ operations up to 300% and a reduction of server CPU utilization (a consequence of non utilization of PL/SQL interface).

Query Change Notification
JDBC support for Database and Query Notification is a scalability mechanism, which let's a JDBC thread subscribe to notification of changes in a query result set. Using the feature involves the following steps:

(i) Registration: JDBC applications can register SQL queries (i.e., create registrations) using either the JDBC-style or PL/SQL-style registration.

(ii) SQL Query Association (Table Changes): associate SQL queries with the registrations.

(iii) RDBMS Notification: upon DDL and/or DML changes that impact the registered queries or their result sets, the Oracle database notifies the JDBC thread through a dedicated network connection. The driver transforms these notifications into Java events, received by the application.

Middle-tier may use this feature to invalidate and refresh data caches, timely. See more details on the APIs in the Oracle Database 11g JDBC doc.

Data Integrity and Encryption in JDBC-Thin

The Oracle database supports AES256, AES192, or AES128 encryptions algorithms. To protect data transfer from attacks, such as data modification, deleted packets, and replay attacks, a cryptographic digest, using MD5 or SHA-1 hashing algorithms is included with each message sent across the network. JDBC programs set the encryption and checksum algorithms as connection property; if the database does not accept such encryption or checksum, the connection attempt will fail.

sqlnet.ora

SQLNET.ENCRYPTION_TYPES_SERVER = (AES256, AES192, AES128)
SQLNET.ENCRYPTION_SERVER = accepted
jdbc

prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_LEVEL, AnoServices.ANO_REQUIRED);
prop.setProperties( OracleConnection.CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_TYPES, " + AnoServices.ENCRYPTION_AES256 + "," +AnoServices.ENCRYPTION_AES192 + ")");

Strong Authentication in JDBC-Thin

Strong authentication consists in Oracle authentication adapters, which support third-party and industry-standard authentication methods including: Kerberos, Remote Authentication Dial-In User Service (RADIUS), Distributed Computing Environment (DCE), and Secure Sockets Layer (SSL). In this release, JDBC-Thin supports Kerberos, RADIUS and SSL authentications. It works as follow:

#1: Configure and starts the third party authentication server (i.e., Radius, Kerberos) or set the keystore.

#2: Configure the database to use third party authentication
REMOTE_OS_AUTHENT = TRUE (INIT.ORA)
SQL> create user ssl_client1 identified externally as 'CN=Test Client2,C=US';
SQL> grant create session, connect, resource to ssl_client1;

#3: Configure Net services for Radius or Kerberos
sqlnet.authentication_services = (beq,kerberos5)
sqlnet.authentication_kerberos5_service = dbji
sqlnet.kerberos5_conf = /home/Jdbc/Security/kerberos/krb5.conf
sqlnet.kerberos5_keytab = /home/Jdbc/Security/kerberos/dbji.stacv41
sqlnet.kerberos5_conf_mit = true
sqlnet.kerberos5_cc_name = /tmp/krb5cc_5088

#4: Set connection property in JDBC application
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_AUTHENTICATION_SERVICES,"('+AnoServices.AUTHENTICATION_RADIUS+')");

OS Authentication in JDBC
The Oracle database 11g allows delegating client authentication to the Operating System. This permits users to authenticate and connect to the database using their OS credentials.

#1: Enable OS authentication in init.ora
REMOTE_OS_AUTHENT = TRUE

#2: Create a database user to allow an OS authenticated connection:
CREATE USER ops$w_rose IDENTIFIED EXTERNALLY;
GRANT CONNECT TO ops$w_rose;

#3: In your JDBC application
String url = "jdbc:oracle:thin:@stacv41.us.oracle.com:5521:dbja"
Driver driver = new oracle.jdbc.OracleDriver();
Properties props = new Properties();
Connection conn = DriverManager.getConnection( url, props);

Database Features for Application Deployment

Manageability is a key requirement in production environment for Java applications running against (i.e., JDBC, Java EE components) or within the Oracle database (i.e., Java in the database). The following manageability features are available through JDBC: Default Service, MBean for Logging, programmatic startup and shutdown. The following manageability features are available through the embedded Java runtime: JIT compiler, Like-JDK interface, Properties interface, JMX, Database Resident JARs, Output redirect, Two-tier duration of Java session state, and OracleJVM Utilities enhancements.

OracleJVM JIT Compiler
Java in the database has been introduced since 8i, and addresses a specific set of user-defined database functionality more efficiently than PL/SQL and Java EE. However, it has not yet been massively adopted because (i) it’s been perceived as slower than the JDK VM for pure java execution (even though the combination of SQL and Java runs faster in the database); (ii) the Native Compiler (a.k.a. NCOMP) which brings significant performance is rarely used due to a poor ease-of-use (i.e.,
explicit step and requires a C Compiler).

Prior to Oracle Database 11g, there were 3 Java execution modes: (i) Full Interpreted (out-of-the-box), (ii) System NCOMP (JAccelerator installed from the companion CD), and (iii) user NCOMP (full compiled). Most customers run Java fully interpreted mode (both system and user classes).

Starting with Oracle Database 11g Release 1, the OracleJVM furnishes a just-in-time compiler (JIT). The JIT dynamically and transparently produces native binaries from any Java classes existing in the database. The binary code are stored, avoiding recompilation. The JIT is enabled by default (out-of-the-box) and does not need a C compiler or further configuration. In house testing using industry standard benchmark shows an order of magnitude (10 x) speed-up.

**JDBC Default Service**

Starting with the XE release, the Oracle database comes with a new connection feature. Under certain conditions, the Oracle database connection adapter requires only the host name of the computer where the database is installed.

As a result, parts of the JDBC connection URL syntax become optional:

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

In this URL:

- `//` is optional.
- `port` is optional; specify a port only if the default Oracle Net listener port (1521) is not used.
- ORCL: default service name is optional; the connection adapter for the Oracle database client connects to the default service on the host, which is set to ORCL in the listener.ora file.

The example below shows a basic configuration of the listener.ora file, where the default service is defined.

**Example 3–2 Default Service Configuration in listener.ora**

```
MYLISTENER = (ADDRESS_LIST=  
  (ADDRESS=(PROTOCOL=tcp)(HOST=test555)(PORT=1521))  
  DEFAULT_SERVICE_MYLISTENER=dev.testserver.com  
  SID_LIST_MYLISTENER = (SID_DESC=(SID_NAME=dev)(GLOBAL_DBNAME=dev.testserver.com)(ORACLE_HOME=/test/oracle))  
)
```

Restart the listener with the following command:

```
> lsnrctl start mylistener
```

The following URLs should work with this configuration:

```
jdbc:oracle:thin:@//test555.testserver.com  
jdbc:oracle:thin:@/test555.testserver.com:1521  
jdbc:oracle:thin:@test555.testserver.com:1521  
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=test555.testserver.com)(PORT=1521)))  
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=test555.testserver.com)))  
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=test555.testserver.com)(PORT=1521))(CONNECT_DATA=(SERVICE_NAME=)))
```

Note: Default service is a new feature starting with Oracle Database 11g Release 1. If you use any older version of the Oracle database client to connect to the database, then you must specify the SID and port number.

**OracleJVM Like-JDK Interface**

Oracle Database 11g OJVM provides a command line interface which is similar to JDK/JRE Java shell
command. Similar to the JDK virtual machine (VM), this interface lets you run Java code directly from the file system without loading the code into the database.

The RUNJAVA command line interface:

- is analogous to JDK/JRE Java shell command
- Uses the standard -classpath syntax to indicate where to find the classes to load
- Can set the system properties using the JDK standard -D syntax
- Implemented by the following functions:
  
  ```plsql
  FUNCTION runjava(cmdline VARCHAR2) RETURN VARCHAR2;
  FUNCTION runjava_in_current_session(cmdline VARCHAR2) RETURN VARCHAR2;
  ```

The command line interface is a PL/SQL function that takes a string (VARCHAR2) argument and parses it as a command line input. If the input is properly formed, the interface runs the indicated Java method in the OracleJVM.

The PL/SQL package `DBMS_JAVA` provides the `runjava` function

- It takes the Java command line as its only argument and runs it in the OracleJVM; the format of the command line is the same as that taken by the JDK shell command
- `[option switches] name_of_class_to_execute arg1 arg2 ... argn`
- The switches are `-classpath`, `-D`, `-Xbootclasspath`, and `-jar`.

In addition, the RUNJAVA command line has been integrated with OJVMJAVA as illustrated by the following command:

```
ojvmjava -u scott/tiger -runjava -d edidrp7p1:1521:ORCL -t $ java -cp . Workers 621 Director 150000
```

Properties Interface

In any Java runtime, the `java.lang.System` class contains a table associating certain String keys to String values known as the `System.properties`. The values in this table are used in Java session initialization before execution of any user code. In releases prior to Oracle Database 11g, OracleJVM had no way to modify the default initial settings of System Properties.

To emulate the familiar JDK behavior of setting system properties, Oracle Database 11g enhances `DBMS_JAVA` with functions that set, retrieve, remove and display system properties. Within an RDBMS session you can maintain a set of values that are added to the system properties whenever a Java session is started within the database. The functions that manipulate system properties, work on the key-value pair methodology, where both the key and the value are of `VARCHAR2` type. Here are the `DBMS_JAVA` functions responsible for setting, retrieving, removing and displaying system properties.

Setting a key-value pair:

```plsql
FUNCTION set_property(name VARCHAR2, value VARCHAR2) RETURN VARCHAR2;
```

Retrieving the value in a key:

```plsql
FUNCTION get_property(name VARCHAR2) RETURN VARCHAR2;
```

Removing an existing key-value pair:

```plsql
FUNCTION remove_property(name VARCHAR2) RETURN VARCHAR2;
FUNCTION show_property(name VARCHAR2) RETURN VARCHAR2;
```

Database Resident JAR

Database resident JARs allow you to associate a Jar file with its contents (i.e., signed JARs).
To load a JAR into the database, `loadjava` has been enhanced with two new options:

- **-jarsasdbobjects**
  Indicates that JARs processed by the current `loadjava` command are to be stored in the database along with the classes and other objects they contain, and that knowledge of the association between the objects and the JAR is to be retained in the database.

- **-prependjarnames**
  Is used with the `-jarsasdbobjects` option. This option enables classes with the same names coming from different JARs to coexist in the same schema. It does this by preceding a version of the name of the JAR to the class name to produce a unique name for the database object.

Two-tier Duration of Java Session State

When a Java session terminates, various resources, such as UGA/PGA memory and Java static variable, are released. Prior to Oracle Database 11g, the only ways to terminate a Java session inside an RDBMS session were either by calling `System.exit` or similar `OracleRuntime` methods.

You can now terminate Java within an RDBMS session, with and without retention of system properties.

```sql
DBMS_JAVA has been enhanced by adding the following new functions
FUNCTION endsession RETURN VARCHAR2
FUNCTION endsession_and_related_state RETURN VARCHAR2
```

Using these functions, you can now terminate Java in a session while keeping the property settings so as to speed up Java reinitialization within the same RDBMS session. This feature also allows immediate termination of Java sessions, without requiring the use of `System.exit`.

Output Redirect

Java has two standard output streams called `System.out` and `System.err`. A Java program running on an OS shell, `System.out` and `System.err` correspond to the shell's `stdout` and `stderr`.

Prior to Oracle Database 11g, such a correspondence was missing. Hence OracleJVM used to write the output to a `.trc` file which could be redirected to `DBMS_OUTPUT` using `DBMS_JAVA.set_output`.

To provide you a better control over the destination of Java output generated by OracleJVM, 11g supports output redirection to SQL, autonomous Java sessions, or to a file. In addition to this in 11g you can also disable sending output to trace file.

To provide output redirection, the following new functions have been added to the PL/SQL `DBMS_JAVA` package:

```sql
FUNCTION set_output_to_sql (id VARCHAR2, stmt VARCHAR2)
• Executes the SQL statement specified as `stmt`, whenever output to the default `System.out` and `System.err` streams occurs. You may have multiple redirection specifications, each identified by a unique id.

FUNCTION set_output_to_java (id VARCHAR2, class_name VARCHAR2, class_schema VARCHAR2, method VARCHAR2)
• Defines a named output specification that gives an instruction for executing the Java method, `class_name.<method>` of the schema `<class_schema>`, whenever output to the default `System.out` and `System.err` streams occurs.

FUNCTION set_output_to_file (id VARCHAR2, file_path VARCHAR2)
• 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`.
```
FUNCTION remove_output_to_sql (id VARCHAR2)

- Deletes a specification created by set_output_to_sql. Similarly to remove specifications created by set_output_to_java and set_output_to_file you can call remove_output_to_java and remove_output_to_file respectively.

FUNCTION disable_output_to_sql (id VARCHAR2)

- Disables a specification created by set_output_to_sql. Similarly to disable specifications created by set_output_to_java and set_output_to_file you can call disable_output_to_java and disable_output_to_file respectively.

FUNCTION enable_output_to_sql (id VARCHAR2)

- Re-enables a specification created by set_output_to_sql. Similarly to enable specifications created by set_output_to_java and set_output_to_file you can call enable_output_to_java and enable_output_to_file respectively.

Programmatic Startup and Shutdown

An API allows a JDBC program with proper permission/privileges to programmatically start up and shut down the database.

Startup

```
prop1.setProperty("user", "sys");
prop1.setProperty("password", "manager");
prop1.setProperty("internal_logon", "sysdba");
...
stmt.executeUpdate("ALTER DATABASE MOUNT");
stmt.executeUpdate("ALTER DATABASE OPEN");
```

Shutdown

```
stmt.executeUpdate("ALTER DATABASE CLOSE NORMAL");
stmt.executeUpdate("ALTER DATABASE DISMOUNT");
stmt.close();
conn.shutdown(OracleConnection.DBSHUTDOWN_FINAL);
```

JMX in OracleJVM

The Java Management Extensions (JMX) API (javax.management package) allows managing and monitoring applications, system objects, devices, service-oriented networks, and the JVM (Java Virtual Machine). Java classes (MBeans) can be dynamically constructed and loaded to monitor and manage resources as they are installed. These MBeans are registered in an MBean server. A JMX agent consists in an MBean server, in which MBeans are registered, and a set of services for handling MBeans.

The JMX API includes remote access, which allows an external management program to interact with a running application.

In OracleJVM implementation, a new jmxserver role has been introduced to provide the required permissions to start and maintain a JMX agent in a session.

In addition, a new procedure 'dbms_java.start_jmx_agent' starts the agent in a specific session for the duration of the session.
To start JMX in a user session, one must first obtain \texttt{JMXSERVER} role from \texttt{SYS} or \texttt{SYSTEM}:

\begin{verbatim}
SQL> grant jmxserver to scott;
\end{verbatim}

Once a user is granted \texttt{JMXSERVER} role, the use can invoke the procedure

\begin{verbatim}
'dbms_java.start_jmx_agent (<port> <ssl> <authentication>') to startup JMX in a session.
SQL>call dbms_java.start_jmx_agent('9999', null, null);
\end{verbatim}

The JMX server runs as one or more daemon threads in the current session. It allows monitoring java code running in the session. With JMX support, OracleJVM moved from being a black box to a white box; using a standard JMX enabled console such as JConsole, one can monitor Java classes.

See the Oracle Database 11g Java Developers Guide for more details.

\textbf{JMX in JDBC: Logging MBean}

JDBC Logging can be configured either programatically or through Java system property. Using the standard \texttt{java.util.logging} framework and the \texttt{javax.management} framework, a new JDBC MBean allows turning JDBC logging on or off, programatically.

\begin{verbatim}
// get the MBean server
javax.management.MBeanServer mbs =
  java.lang.management.ManagementFactory.getPlatformMBeanServer();
// find out if logging is enabled or not
System.out.println("LoggingEnabled = " + mbs.getAttribute(name, "LoggingEnabled");
// enable logging
mbs.setAttribute(name, new javax.management.Attribute("LoggingEnabled", true));
// disable logging
mbs.setAttribute(name, new javax.management.Attribute("LoggingEnabled", false));
\end{verbatim}

\textbf{OracleJVM Utilities Enhancements}

The Oracle database 11g, brings significant enhancements to the command-line utilities for loading, dropping and running Java classes in the database. In addition, it furnishes a new utility (o JVMTC), which allows Java classes and Java archives (i.e., JAR ), to proactively resolve all references ahead of the runtime.

\textbf{OJVMTC: Stand-alone Reference Resolver}

OJVMTC is a new standalone class closure tool \texttt{ojvmtc} to ensure that a given list of file system resident classes and deployable JARs are closed as much as possible before \texttt{loadjava} invocation.

This tool simplifies deployment of Java applications in the Oracle Database. To close specific classes and Jar files, specify them in a colon separated list with the \texttt{-classpath} option.

The following example closes \texttt{jar1} and \texttt{jar2} found in the server specified by the connect string to resolve external references in \texttt{app2.jar}.

\begin{verbatim}
ojvmtc -server thin@scott/tiger:localhost:1521:orcl -classpath jar1:jar2 -list app2.jar
- list
- The \texttt{-list} parameter, specifies that any missing references are to be printed to \texttt{stdout}.
- You can also create a Jar file of all the classes and Jars that \texttt{ojvmtc} closes, using the \texttt{-jar} option.
\end{verbatim}
In the following example, rt.jar, classdir/lib1.jar, and classdir/lib2.jar are being used resolve references in app.jar.

```
ojvmtc -bootclasspath $JAVA_HOME/jre/lib/rt.jar
   -classpath classdir/lib1.jar:libdir/lib2.jar
   -jar set.jar app.jar
```

- The classes specified by -bootclasspath are also used for closure, but such classes, unlike the classes specified by -classpath, are not included in the closure set. In the example above, all the examined classes are added to set.jar, except those found in rt.jar.

**URL support for loadjava**

Prior to Oracle Database 11g, `loadjava` could not load Jars, class files, or resources from distant locations using *URLs*. A new option, `-proxy`, has been added to support this functionality:

```
-proxy host:port
```

Example:

```
loadjava -u scott/tiger -r -v -proxy proxy_server:1020 http://my.server.com/path/my.jar
```

The schema must be granted Java socket permission in order to access the remote source.

**List based operation with dropjava**

Earlier versions of `dropjava` required that the classes, JARs, source, and resources be present on the machine, where the client or server side utility is running. In Oracle Database 11g the `dropjava` tool has been enhanced with `-list` and `-listfile` options which facilitates deletion of the above mentioned objects, based on a list of classes, which may or may not exist on the client or the server machine.

**Enhancements in ojvmjava**

Starting with Oracle Database 11g Release 1, `ojvmjava` has been enhanced with

- Error and exception handling: recognize invalid commands and issue an error and reduce the amount of stack trace information produced for an exception when not in debug mode
- a new command `-connect` has been introduced to `ojvmjava` which allows changing connection without leaving `ojvmjava`; the `-connect` command drops the current connection and attempt to open another based on the following command options.

  - `-user (-u)` : User
  - `-password (-p)` : Password
  - `-thin (-t)` : Use the thin JDBC driver for connection
  - `-oci, -oci8 (-o)` : Use the native oci JDBC driver for connection
  - `-database (-d)` : Database descriptor.

Examples:

```
connect -o -u scott/tiger
connect -o -u scott/tiger@inst1
connect -u scott/tiger@localhost:1521:db -t
```

Java Developer's Perspective on Oracle Database 11g
Conclusion

Are you wondering about Oracle Database and Java? How to save million database roundtrips (or more) per day? Is Java in the database ready for mass adoption?

This paper has tried to respond to these questions and much more, by highlighting Java standards support, key development time and deployment time benefits such as in terms of productivity, portability, security, performance, ease of use, and manageability that Java developers will get from Oracle database 11g.