The Oracle Forms Java Importer

Technical White Paper
September 2001

ORACLE®

OVERVIEW

The Oracle Forms Developer and Oracle Forms Server product set provides a rich application framework to build and deploy advanced business applications. With the tight integration these product components have with the Oracle 8*i* Server products, the natural language of choice for application development is Oracle PL/SQL. Oracle PL/SQL is Oracle Corporation's procedural language extension to SQL, the standard data access language for object-relational databases. PL/SQL offers modern software engineering features such as data encapsulation, exception handling and information hiding coupled with the absolute ease of access to data stored in Oracle 8*i* database servers through the SQL standard language.

While Oracle PL/SQL is a powerful and productive development environment, it is sometimes necessary to integrate Oracle Forms applications with other external application services and providers. Usually these external applications are not written in Oracle PL/SQL so some form of integration capability is required. Traditionally, the interfaces and libraries exposed by external applications have usually been based on the C programming language. For these situations, the existing Oracle Forms integration mechanism, ORA_FFI has proved more than sufficient

With the ever-increasing momentum of the Java movement within the IT industry, many applications are now providing integration points in Java. Similarly, the number of standard Java libraries for enterprise applications is continually growing.

With this shifting of the landscape towards Java, and being ever cognizant of the need to further enhance the application integration capabilities of the Oracle Forms product set, a new feature, the Java Importer, has been introduced. The Java Importer facilitates the invocation of business logic written in Java from a Forms application.

Using the Java Importer, you can automatically generate PL/SQL packages and procedures that will allow you to create and use Java objects directly in your forms applications, all via the PL/SQL language facilities provided by Oracle Forms Developer and the runtime services provided by Oracle Forms Server.

This document provides a description of the Java Importer feature and its functionality.

THE JAVA IMPORTER

The Java Importer provides you with the facilities you need to create Forms applications that incorporate functionality contained in external Java classes.

The Java Importer is the name given to a set of components that has been added to the Oracle Forms Developer and Oracle Forms Server products. The Java Importer enables a Forms application to call out to Java to make use of code contained within compiled Java classes. Using the components of the Java Importer makes it possible to create PL/SQL packages for specified Java classes within a Forms application and to instantiate, use, and destroy the Java object instances when the Forms application is run.

COMPONENTS OF THE JAVA IMPORTER

The Java Importer consists of a set of components that together can be used to access Java classes from Oracle Forms applications and perform operations on them. These components are:

- The Java Importer Tool, which allows a developer to select and specify which Java classes they wish to access in their application.
- The Java Importer Generator, which creates PL/SQL packages that provide access to the specified Java classes.
- The ORA_JAVA package, which provides a set of convenience functions that assist a developer in working with the selected Java packages.
- The Oracle Forms Server JNI Bridge, which handles the low level interaction with the Java classes at runtime.

These Java Importer components include functionality that resides in both the Oracle Forms Developer Builder and the Oracle Forms Server.

THE JAVA IMPORTER AND ORACLE FORMS DEVELOPER

Oracle Forms Developer is the development component of the Oracle Forms product set and is used to build Oracle Forms applications.

The Java Importer Tool

The Java Importer Tool is a new dialog in Oracle Forms Developer that provides you with a way to select or specify the Java classes you wish to make use of in your Forms application. Once you have selected the required Java classes, the Java Importer Tool calls the PL/SQL Generator to create a PL/SQL package for each class you have selected. You will use this tool whenever you want to provide access to a Java class to your application. The Java Importer Tool can be run multiple times during a development session as new Java class access requirements are discovered; it is not necessary to identify all the classes needed at once.

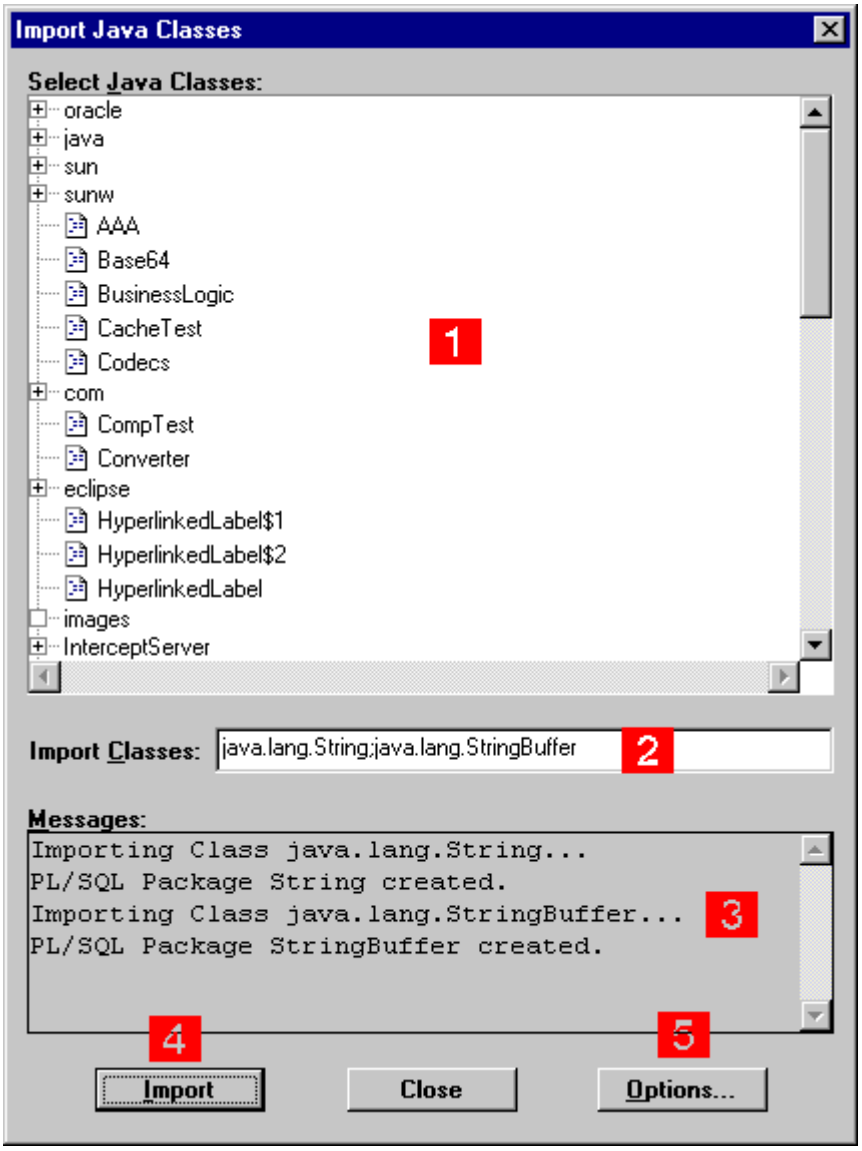

Figure [1] : The Java Importer Tool

The important sections of the Java Importer Tool for a developer to know are:

[1] Class Browser

The class browser lists all of the Java classes found on the CLASSPATH. The classes are ordered in the same way that they are represented in the CLASSPATH. A tree view is used to list the classes in a hierarchical format that corresponds to their package structures. The class hierarchy is navigated by opening and closing the tree branches, which represent the package levels. Class selections are made by clicking on the leaf nodes, which represent the actual Java classes. Multiple class selections can be made at one time using the shift key in conjunction with the mouse.

When a class is selected, the fully qualified name of the class is added to the Import Class list field.

The set of classes available for the Java Importer is determined when the Java Importer Tool is first invoked. Subsequent invocations of the Java Importer Tool reuse the same set of classes. Classes added to the CLASSPATH during an Oracle Forms Developer session will not be displayed in the class browser unless the Oracle Forms Developer session is stopped and restarted. If classes are added during an Oracle Forms Developer session and are accessible from the CLASSPATH, they may be imported by manually entering the fully qualified name of the class in the Import Classes list field.

[2] Import Classes List

The Import Classes list field displays the list of fully qualified classes that will be made accessible to the Forms application when the Import button is pressed. The list of classes is populated via selections made in the Class Browser or by directly entering the fully qualified class names of the Java classes to be made available into the Import Classes list field. Multiple class names are separated by the use of a semicolon.

[3] Messages Display

The Messages display is where the output of the PL/SQL Generator is sent. It displays the progress as the PL/SQL generation is performed and the result of the generation phase, including the name of the packages generated for each of the Java classes specified. It also displays any errors that occur during the PL/SQL generation phase.

[4] Import Button

The Import button starts the PL/SQL generation process for each of the Java classes specified in the Import Classes list field.

[5] Options Button

The Options button displays a dialog that is used to set the PL/SQL generation options. These options are described in the documentation that accompanies the Java Importer.

The Java Importer PL/SQL Generator

The Java Importer PL/SQL Generator performs the task of creating a PL/SQL package that exposes the methods identified in the class via PL/SQL functions and procedures. A separate PL/SQL package is created for each Java class you import. The generated PL/SQL packages are the way in which you interact with the Java classes you specified with the Java Importer Tool.

The PL/SQL Generator uses the reflection mechanism provided by the Java API to look inside a Java class and extract all of the field and method information from the class. The method information includes the name of the method, the return type, the method parameters and their type, and the method modifier. The field information includes the existence and details of any class level variables.

The PL/SQL packages will, where possible, mimic the Java classes in terms of naming of the variables, functions, and procedures. There are a number of situations where the Java names will be slightly modified, for instance, in the case of PL/SQL reserved word conflicts. The documentation accompanying the Java Importer contains a comprehensive discussion of the mechanics of name conflict resolution.

With the information extracted from the Java class, the PL/SQL Generator creates a PL/SQL package specification and body that represents the Java class. A separate PL/SQL package will be generated *for every* class you specify via the Java Importer Tool. The package specification contains the list of the functions and procedures that map to the Java methods. The package body contains PL/SQL code for each of the declared functions and procedures that perform the operations required to invoke the method on the identified Java class

The algorithm for creating the entries in the PL/SQL package is as follows:

- For each public constructor in the class has, a PL/SQL function called new is created. If the constructor has parameters, then the new function for that constructor has the same parameters. The new function returns a new object of type ORA_JAVA.JOBJECT which represents the newly instantiated Java object.

- For each public method with a return type of void, a PL/SQL procedure of the same name is created. If the method has parameters, then the procedure has the same parameters. The parameters will be typed using the mappings described in the documentation.

- For each public method with a return type that is not void, a PL/SQL function of the same name is created. If the method has parameters, then the function has the same parameters. The return type of the function will map directly to the return type of the Java method using the type mappings described in the documentation.

- For each public, static variable defined in the Java class, a PL/SQL package variable is created of the same name. When the PL/SQL package is first initialized, the value of the variable is extracted from the Java class and the PL/SQL package variable is set. When a variable in a Java class is declared as static and final then it becomes a constant.

- For each public variable defined in the Java class, a get function and a set procedure may be created to allow for the extraction and setting of the variable's value. The get function and set procedure are only created if the create get/set option is checked in the Java Importer option setting. The function and procedure will be named in accordance with the JavaBean naming convention, where the set procedure will be represented as `setVariableName` where VariableName is the name of the variable. Similarly, the get function will be represented as `getVariableName` where VariableName is the name of the variable.

The PL/SQL Generator performs type translations when it is generating the PL/SQL packages from the Java methods. Some examples of this type translation are:

- Parameters or return types that are specified as a Java String in the specified Java class are always mapped to PL/SQL varchar2 types in the generated PL/SQL package.

- A non primitive, non java.lang.String parameter or return type is always mapped to an ORA_JAVA.JOBJECT type in the generated PL/SQL package.

For a more details on the type translation that occurs at PL/SQL package generation time, please consult the accompanying documentation for the Java Importer.

Class and Instance Methods

The Java Importer enables you to work with both class (static) and instance methods of objects. Using the "new" functions in the generated PL/SQL packages, you can create instances of Java

classes and obtain references to those instances. Once you have a reference to an object returned from the new operation it is possible to perform operations on that specific object instance while it is still valid.

When you wish to invoke a method on a specific object that you have previously created, you pass the reference to the instance of the object to the instance methods in the PL/SQL package that corresponds with the Java class of the object. The desired method will be executed on the specific object instance you pass in as a parameter to the PL/SQL function or stored procedure.

PL/SQL functions and procedures that represent instance methods always take an initial parameter of type ORA_JAVA.JOBJECT that represents the actual object instance that methods should be executed on.

PL/SQL functions and procedures that represent class methods do not require an instance object as a parameter.

For example the Java String class contains a number of class methods that perform conversion of primitive Java types into a String representation. These methods do not operate on a specific instance of a String object but work generally on the String class. An example of this is the `valueOf` method that returns a String representation from a scalar value.

The String class also contains instance methods that operate on a specific instance of a String object. An example of this is the `length` method that returns the length of a specific String object.

```
public final class String
{
  . . .
  public static String valueOf(int i);
  . . .
  public int length();
  . . .
}
```

Figure [2]: Snippet of Java String class definition

The PL/SQL Generator would create the following PL/SQL functions for these Java methods in the String class.

```
PACKAGE STRING_ is

  FUNCTION valueOf(a0 NUMBER) RETURN VARCHAR2;
  . . .
  FUNCTION length(obj ORA_JAVA.JOBJECT) RETURN NUMBER;

END;
```

Figure [3]: Snippet of PL/SQL package spec for String

The valueOf PL/SQL function represents a Java class method and, as such, it does not require a parameter of type ORA_JAVA.JOBJECT to indicate the specific object instance on which it should invoke the valueOf method.

On the other hand, the length PL/SQL function represents an instance method. This requires that it be passed a reference to the actual instance of the String object on which that the length method should be invoked.

The Java Importer ORA_JAVA Package

To assist with the use of the Java Importer, a new package, ORA_JAVA is provided. This package provides a set of convenience built-ins that enable you to work with the Java Importer and the generated PL/SQL of an imported Java class.

The ORA_JAVA package provides convenience built-ins in the following areas:

- New PL/SQL type definitions
- Array creation and manipulation
- Runtime errors
- Java Exceptions thrown in accessed code
- Java object persistence

New PL/SQL Type Definitions

The ORA_JAVA package introduces two new data-types that can be used in your applications when working with the Java Importer.

ORA_JAVA.JOBJECT is a new data-type that is designed to store references to Java objects. Any time you create a Java object instance via the new function in a Java Importer generated PL/SQL package, you need to store the result in a variable of type JOBJECT. The JOBJECT data-type can be used in conjunction with the ORA_JAVA packages' persistence functions, described in the "Lifetime of Java Object" section, to manage the lifetime of the Java object instance.

ORA_JAVA.JARRAY is a new data-type that is designed to store references to Java arrays. Any time you create an array using the array built-ins in the ORA_JAVA package, you must store the result in a variable of type JARRAY. The JARRAY object is used to store all arrays, irrespective of the data type of the array elements. The JARRAY data-type is a subtype of the JOBJECT data-type and as such can be used with the persistence functions to control the lifetime of the array.

Array Creation and Manipulation

The Java Importer supports the use of arrays that are fully interoperable with Java arrays. The ORA_JAVA.JARRAY type is used to store references to created arrays. The ORA_JAVA.JARRAY type is a subtype of a ORA_JAVA.JOBJECT, so arrays can be persisted in the same manner as all other Java objects using the global reference functions.

Arrays can be created of any Java scalar type or of the java.lang.Object type. The ORA_JAVA package contains built-ins that allow you to create arrays of a specific type and of a designated length. Arrays can be returned from function calls in the generated PL/SQL packages.

The values of the elements of an array can be set using the provided ORA_JAVA.SET_<type>_ARRAY_ELEMENT built-ins.

The values of the elements of an array can be retrieved using the provided ORA_JAVA.GET_<type>_ARRAY_ELEMENT built-ins.

Another convenient array built-in is the ARRAY_LENGTH function that returns the maximum length of a specified array. This built-in is commonly used when an array object is returned from a Java method call and the length of the array is unknown.

Java Importer Runtime Errors

When Oracle Forms Server is working with the JVM, it is possible that errors may occur. Some examples of the types of errors that may occur are: the JVM could not be initialized for some reason

when an attempt was made to perform the task, or perhaps an array index that is specified for an ORA_JAVA.JARRAY variable is out of bounds. The documentation for the Java Importer provides a full list of reportable runtime errors.

When an error of this type occurs, it indicates that an error has happened within the Oracle Forms Server runtime process as it has attempted to perform an operation with the JVM. This causes a PL/SQL exception of type ORA_JAVA.JAVA_ERROR to be thrown, indicating that an unexpected result occurred. This PL/SQL exception can be detected using the standard PL/SQL exception handling mechanism. If you wish to obtain more information about which exact error occurred, the ORA_JAVA.LAST_ERROR function will return the runtime error information as a PL/SQL VARCHAR2.

Handling Java Exceptions from PL/SQL Code

The Java programming language uses the concept of exceptions to indicate that something has gone amiss when the program is executing. These types of exceptions are best thought of as something going wrong in the Java code itself. For example, attempting to invoke a method on an object that has not been instantiated will throw a java.lang.NullPointerException.

In Java parlance, exceptions are "thrown" when an abnormal condition is met. Similarly, exceptions are "caught" when code exists within an application that can recover from the abnormal condition.

The ORA_JAVA package provides you with the capability to work with these Java exceptions as they are thrown in the Java code that is called from your Forms application. When a Java exception is thrown inside the Java code as it is being executed, Oracle Forms Server will detect this and will raise a PL/SQL exception called ORA_JAVA.EXCEPTION_THROWN. This PL/SQL exception can be detected using the standard PL/SQL exception handling mechanism. The built-in ORA_JAVA.LAST_EXCEPTION can then be used to obtain a reference to the actual Java exception object that was thrown in the Java code. You should note that the built-in returns a reference to the *actual* Java exception object that was thrown. You can use this exception object just as you would any other Java object you had created.

The code snippet in Figure [4] demonstrates how to work with Java exceptions. The ORA_JAVA. EXCEPTION_THROWN PL/SQL exception is handled in the PL/SQL block. When this exception is detected, the actual Java exception is assigned to a local object. Using this object and

the imported java.lang.Exception package, the getMessage Java method is invoked on the exception object to display the actual error that was detected.

```
DECLARE
  exc ora_java.jobject;
  . . .
BEGIN
  [ do some operations ]
EXCEPTION
  WHEN ORA_JAVA.EXCEPTION_THROWN THEN
    exc := ORA_JAVA.LAST_EXCEPTION;
    MESSAGE(Exception.getMessage(exc));
    ORA_JAVA.CLEAR_EXCEPTION;
END;
```

Figure [4: Handling Java Exceptions in PL/SQL

Java Object Persistence

The ORA_JAVA package provides two built-ins that allow you to explicitly control the persistence of any Java objects you create. By default, any Java object that you create is valid only for the duration of the program unit you create it in. Once the program unit has completed, the Java objects are freed by the JVM. Using the persistence functions in the ORA_JAVA package, you can mark an object you create as global, which means that the object will not be freed by the JVM when the program unit ends. The object will remain valid until you explicitly unmark it as a global which allows the JVM to free the object when the next round of garbage collection runs.

To mark an object as a global reference, use the ORA_JAVA.NEW_GLOBAL_REF. This built-in takes the object you wish to make global as a parameter and returns a new object that is the global version of the original object. Since PL/SQL does not have global variables, you will need to store the returned global object in a package variable so that its value is kept.

To unmark an object as a global reference, use the ORA_JAVA.DELETE_GLOBAL_REF. This built-in takes the global object as a parameter and removes it as a global reference.

Using these built-ins changes that way that objects are managed by the JVM. You should take care that for any long-running process, you delete any global references you have created when you no longer have any use for them. Accumulating large numbers of global references without removing

them will increase the memory consumption of the JVM and will affect the scalability of your application.

A more detailed explanation of this process is given in the "Lifetime of Java Objects" section in this document.

THE JAVA IMPORTER AND ORACLE FORMS SERVER

Oracle Forms Server is the server side component of the Oracle Forms product set and it is used to run the Forms applications that have been developed in a thin client, network-oriented manner. Oracle Forms Server is responsible for three things: managing the lifecycle of a forms session for an end user, executing the application logic contained in a Forms module, and maintaining the transactional state of any data used in the application with an Oracle 8*i* database.

For the Java Importer feature, the functionality of Oracle Forms Server has been extended to provide facilities for the interaction with a standard Java runtime engine – the Java Virtual Machine (JVM). This coupling of Oracle Forms Server and JVM environments is achieved through the use of the Java Native Interface. The Java Native Interface (JNI) is a standard programming interface for writing Java native methods and embedding the Java virtual machine into native applications [1].

Using JNI, Oracle Forms Server can create an instance of a JVM when required, create instances of specified Java classes, invoke methods on Java objects, destroy Java objects it has created, and shutdown the JVM when the application exits.

A set of functions to interface directly with the JNI has been added to Oracle Forms Server. A PL/SQL package has been created to provide PL/SQL access to these new runtime functions. This package, named JNI, is used to facilitate the calling into Java via JNI from the Forms PL/SQL environment.

Coding in JNI is a low-level operation and fortunately, is not required to access Java classes from Oracle Forms Server. As described earlier, the PL/SQL Generator provided within Oracle Forms Developer generates the code required to interact with a specified Java class. The code that is generated makes use of the JNI package to perform all the operations required to instantiate objects, use, and destroy objects of the underlying the Java class. The JNI package is not intended for direct use by you as a developer and does not appear in the list of Built-in packages in Oracle Forms

Developer.

It is beneficial to understand what is happening under the covers when the generated PL/SQL packages are run. The code snippet in Figure [5] is a function that represents one of the constructors for the java.lang.String class that has been imported into an Oracle Forms application by the Java Importer Tool. This constructor takes a String object as its parameter and creates a new String object based on it. The JNI package is used to perform the lookup of the class-id associated with the java.lang.String class, perform the lookup of the initialization method for the String object that takes a String object as its argument, create an argument list to pass to the initialization method, and finally to construct an instance of the String class and return it to calling the program.

```
-- Constructor for signature (Ljava/lang/String;)V
FUNCTION new(a0 VARCHAR2) RETURN ORA_JAVA.JOBJECT IS
BEGIN
  cls := JNI.GET_CLASS('java/lang/String');
  mid :=JNI.GET_METHOD(FALSE, cls, '<init>',
                       '(Ljava/lang/String;)V');
  args := JNI.CREATE_ARG_LIST(1);
  JNI.ADD_STRING_ARG(args, a0);
  RETURN (JNI.NEW_OBJECT(cls, mid, args))
END;
```

Figure [5]: Code snippet of the generated package body for java.lang.String

You can see that there are many steps required to perform this relatively simple operation of creating a new String object. The PL/SQL Generator insulates you from having to write the code to accomplish all of these steps and provides real value in terms of productivity for you as a busy developer.

JAVA IMPORTER LIFECYCLE CONSIDERATIONS

This section contains information concerning the lifecycle of the various elements of the Java Importer including the Java Virtual Machine started and the Java objects used by the running Forms application. A good understanding of this information will assist you in making productive use of the Java Importer feature in your Forms applications.

The Java Virtual Machine

When an Oracle Forms application calls into Java via the Java Importer functionality, a dedicated Java Virtual Machine is created and attached to the Oracle Forms Server runtime process. The JVM is created only when a call is made to a Java class via the Java Importer. If the Java Importer feature is not used in your applications, a JVM is never started by the Oracle Forms Server runtime process.

Lifetime of Java Objects

The process of calling Java classes from Oracle Forms Server at runtime is governed by the standard JNI specification. JNI provides facilities to enable the execution of Java code in a JVM from native methods. In our case, the native method is the Oracle Forms Server runtime process that is executing the Oracle Forms application.

The JNI specification asserts that any objects created by a native method during the JNI session will be valid for the duration of the native method. Following from this, objects created during the native method's execution will be freed automatically when the native method completes and exits.

In the context of the Java Importer, the Oracle Forms Server runtime process is the native method that starts the JVM. The Oracle Forms Server runtime process will only stop when the Forms applications that are being executed by the end user are finished. Given this, the standard behavior of the JNI specification would mean that any objects created through the use of the Java Importer would never be removed from memory while the Oracle Forms Server runtime process is still active. This would cause the memory consumption of the Oracle Forms Server runtime process to increase over time and defeats the automatic memory management feature of Java.

To overcome this issue, the Java Importer utilizes a feature introduced with JDK 1.2 that allows for local frames to be created when working with JNI. A JNI local frame can be thought of as an isolated workspace that can be programmatically controlled via functions in JNI. Objects created within a JNI local frame are freed when the local frame is destroyed.

Oracle Forms Server creates a JNI local frame when a PL/SQL trigger unit is first executed and then destroys the local frame when the PL/SQL trigger unit ends. When a new object is created during the execution of a PL/SQL trigger, the object reference is created in the PL/SQL trigger's local frame. Conversely, when the PL/SQL trigger unit ends, the local frame is destroyed and all objects

created within that local frame are freed.

This programmed behavior causes the *default lifetime* of an object to be equivalent of that of the PL/SQL trigger unit that created it.

Controlling the Lifetime of Java Objects

It may be desirable at times to extend the lifetime of Java objects beyond the scope of the PL/SQL trigger unit that created them. To facilitate this, the ORA_JAVA built-ins NEW_GLOBAL_REF and DELETE_GLOBAL_REF can be used. These built-ins mark the specified Java object as global in the JVM. This results in them not being freed when the local frame in which they were created is destroyed.

When an object is marked as global through the use of the ORA_JAVA.NEW_GLOBAL_REF, it will exist in memory as a valid object until it is explicitly marked as not global through the use of the ORA_JAVA.DELETE_GLOBAL_REF. If an object is marked as global and never subsequently marked as not global it will never be freed from memory until the Oracle Forms Server runtime instance is stopped and the JVM is shutdown.

Java objects that are created through the constructors are stored as PL/SQL variables of type ORA_JAVA.JOBJECT. If a Java object is to be made persistent through the use of the NEW_GLOBAL_REF, then the PL/SQL variable that holds the reference must also exist beyond the lifetime of a local program unit. PL/SQL package variables should be used to store references to persistent Java objects.

For example, consider the case where a Java Vector object is to be used to collate data from several different locations in Forms application code. To do this, we need to:

- Create an instance of the Java object using the generated Vector PL/SQL package.
- Create a global reference from the Java object and store it in a package variable.
- Use the package variable whenever we want to reference the original Java Vector object to add an element to the Vector object.
- Destroy the global reference when it is not longer necessary to access the Vector object.

The PL/SQL code to perform this may look like the code snippets contained in Figure [6] and Figure [7].

```
PACKAGE globals AS
  vec ora_java.jobject;
END;
```

Figure [6]: PL/SQL package to store global references

```
DECLARE
  vec ora_java.jobject;
BEGIN
  vec := Vector.new;
  globals.vec := ORA_JAVA.NEW_GLOBAL_REF(vec);
END;
```

Figure [7]: PL/SQL trigger unit creating a persistent object

```
BEGIN
  . . .
  Vector.add (globals.vec,'some data');
  . . .
END;
```

Figure[8]: PL/SQL trigger unit accessing persistent object to add data

Using the NEW_GLOBAL_REF and DELETE_GLOBAL_REF are the only ways to manage the persistence of the Java object in the JNI space. Creating a PL/SQL package variable and assigning it the value returned from a constructor in a PL/SQL Generated package will not make the object persistent. If an invalid object is used with a function or procedure in a PL/SQL generated package that represents an instance method, this will cause the JNI operation to crash and may result in the corruption and stoppage of the Oracle Forms Server runtime process.

The golden rule is that if you wish to make use of a Java object instance beyond the scope of the PL/SQL trigger unit in which it is created, you must make it persistent by using the NEW_GLOBAL_REF built-in. It is equally important that you remember to delete the global references when they are no longer required so that the long-term impact on memory consumption of your objects is managed by the JVM.

WHEN TO USE JAVA CODE

The Java importer makes it simple to quickly integrate Java calls into your Forms applications, however, we recommend that you exercise caution with regards to when and where you leverage Java functions in this way.

Imported Java code can raise some issues of performance and maintenance which should be considered when deciding to use the functionality.

1. Memory: Each connected processes that calls imported Java code will have to instanciate it's own Java Virtual Machine (JVM) to run the imported Java code. Although the operating system will ensure that much of the memory required for this is actually shared, there is going to be a memory overhead for each and every runtime process that calls into Java code.
2. Execution performance: The Java Native Interface (JNI) layer that is used to communicate from the C based Forms engine and PL/SQL to the Java code is a bottleneck. Although delays caused by the JNI layer are small, and will not be noticed in operations such as sending an Email via Java Mail, the delays could add up to a significant amount if you attempted to use Java code to replace default Forms functionality. Examples of this might be to attempt to replace the default Forms Block functionality with calls out to Enterprise Java Beans (EJB) or Business Components for Java (BC4J). As a result of this we recommend that you don't try and attempt to replace large portions of default Forms functionality with imported code.
3. Maintenance: We recommend that you only use Java code where you are attempting to carry out and operation that can only be done in Java. If it is possible to carry out the same task using PL/SQL then your maintenance burden will be reduced if you continue to implement in PL/SQL. If you do implement a function in Java you will have to maintain both the Java code itself and the Forms interface to that code.

SUMMARY

The Java Importer provides you with the powerful ability to create Forms applications that utilize Java code contained in standard, compiled Java class libraries. The Java Importer automates the generation of PL/SQL functions and procedures to allow you to invoke Java methods on Java objects you create. A convenience package that contains built-ins that assists you with the

interoperation with Java objects is provided to speed up the development process. Finally, the Java Importer completely manages the lifecycle of the Java objects you use in your application with mechanisms to give you explicit control over the objects as you require it.

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
+1.650.506.7000
Fax +1.650.506.7200
http://www.oracle.com/